

# **Energid Actin™ Developer's Guide**

Version 2.0



Energid Technologies  
124 Mount Auburn Street  
Suite 200 North  
Cambridge, MA 02138  
[www.energid.com](http://www.energid.com)



# Energid Actin™ Toolkit

## Version 2.0

### Trademarks


Energid, Actin, Selectin, Actin Control, Actin Design, and Actin Validate are trademarks of Energid Technologies. Linux is a trademark of Linus Torvalds. Windows, Visual Studio, and Visual C++ are trademarks of Microsoft Corporation. SolidWorks is a trademark of SolidWorks Corporation. 3DS Max is a trademark of Autodesk, Inc. Skype is a trademark of Skype Technologies S.A. All other trademarks mentioned in this document are property of their respective owners.






### Technical Support

Energid Technologies Corporation  
124 Mount Auburn Street  
Suite 200 North  
Cambridge, MA 02138  
Phone: (888) 547-4100  
Fax: (888) 522-5689  
<http://www.energid.com>  
[support@energid.com](mailto:support@energid.com)

Actin 2.0 libraries are compiled using Linux, Microsoft Visual C++ 2005, or Microsoft Visual C++ 2008. For Microsoft Visual C++, they use the multithreaded DLL runtime library.

## Table of Contents

1	Actin™ Capabilities Overview .....	14
1.1	Extensible Markup Language (XML) .....	14
1.2	Mathematical and Geometrical Tools.....	14
1.3	Automatic Kinematic Control.....	15
1.4	Dynamic Simulation .....	15
1.5	Parametric Studies .....	16
1.6	Monte Carlo Studies .....	16
1.7	Rendering.....	16
1.8	Machine Vision.....	16
1.9	Network Communications .....	16
1.10	Third-Party Integration .....	16
2	Quick Start .....	17
2.1	Loading a Stated System .....	17
2.2	Viewing a Stated System .....	17
2.3	Defining a Velocity-Control System .....	18
2.4	Defining a Position-Control System.....	20
2.5	Placing an End Effector .....	20
2.6	Moving the End Effector along a Path .....	22
2.7	Creating a Simulation .....	23
2.8	Additional Capability .....	24
2.9	Building .....	24
2.9.1	General Setup .....	25
2.9.2	Compiler Setup.....	25
2.9.3	Linker Setup .....	25
2.9.4	Libraries .....	26
3	ActinViewer .....	29
3.1	File Options .....	30
3.1.1	Open .....	30
3.1.2	Merge .....	30
3.1.3	State.....	31
3.1.4	Save Operations.....	31
3.2	Mouse Interaction Modes .....	31
3.2.1	Eyepoint Mode  .....	31

3.2.2	Guide Mode 	32
3.2.3	Center of Interest (COI) Mode 	32
3.3	Information Overlays.....	32
3.3.1	Bounding Volumes 	33
3.3.2	Mass Property Ellipsoids 	33
3.3.3	Show Frames 	33
3.4	Running Simulation.....	34
3.5	Controlling End-Effectors.....	35
3.5.1	Select Active End Effector.....	35
3.5.2	Edit and Add End Effectors.....	37
3.6	Working with Path Files.....	38
3.6.1	Record a Path.....	38
3.6.2	Playback Mode.....	38
3.6.3	Save and Load Path File.....	39
3.7	Manipulator Configuration.....	39
3.7.1	Changing Joint Positions.....	40
3.7.2	Changing End-Effector Positions.....	41
3.7.3	Changing the Fixed Link.....	41
3.8	Data Capture.....	43
3.8.1	Saving Captured Information in Different File Formats.....	43
3.8.2	Displaying Captured Information in Interactive Plots.....	47
3.9	Time Scaling.....	51
4	Guiding Software Principles.....	52
4.1	This Document.....	52
4.1.1	Overview.....	52
4.1.2	Component Representation.....	52
4.2	Source Code.....	52
4.2.1	Name Space.....	52
4.2.2	Classes.....	52
4.2.3	Identifiers.....	52
4.2.4	Protection.....	53
4.2.5	Virtualness.....	53
4.2.6	Constness.....	53

4.2.7	Pointers.....	53
4.2.8	Factory Methods.....	53
4.2.9	Multiple Inheritance.....	53
4.2.10	Units.....	53
4.2.11	Macros.....	54
4.2.12	Raster Data.....	54
4.2.13	Filenames.....	54
4.2.14	Extension Avoidance.....	54
4.2.15	Exception Handling.....	54
4.2.16	Friends.....	54
5	Fundamental Classes.....	55
5.1	Data Structures.....	55
5.1.1	Vector.....	55
5.1.2	Tensor.....	55
5.1.3	List.....	55
5.1.4	Map.....	55
5.1.5	Set.....	55
5.1.6	Tree.....	56
5.2	Basic Types.....	56
5.3	Basic Kinematics.....	56
5.3.1	Position.....	57
5.3.2	Orientation.....	58
5.3.3	Coordinate System Transformation.....	59
5.3.4	Rigid-Body Velocity.....	61
5.4	Basic Dynamics.....	62
5.4.1	Rigid-Body Acceleration.....	62
5.4.2	Rigid-Body Force.....	64
5.4.3	Rigid Body Mass Properties.....	65
6	XML.....	68
6.1	Overview.....	68
6.2	XML Objects.....	68
6.2.1	Basic Types for Simple Data.....	72
6.2.2	XML Object Containers for STL Containers.....	77
6.2.3	Compound XML Objects.....	85
6.2.4	Variable Compound XML Objects.....	90

6.3	XML Reading and Writing XML Objects.....	102
6.3.1	Top Level Interface for Reading and Writing.....	103
6.3.2	Direct Interface to the XML Reader and Writer.....	105
6.4	Schema.....	110
6.4.1	XML Namespaces.....	110
6.4.2	Schema Auto-Generation.....	113
7	The Link.....	117
7.1	Coordinate Systems.....	118
7.2	Link Kinematics.....	118
7.2.1	General Kinematics.....	118
7.2.2	Denavit-Hartenberg.....	119
7.2.3	General Joint Velocity and Acceleration.....	120
7.3	Mass Properties.....	123
7.4	Physical Extent.....	123
7.4.1	Composable Tree Structure.....	123
7.4.2	Shape Primitives.....	124
7.4.3	Fundamental Geometrical Shapes.....	124
7.4.4	Primary Shapes.....	126
7.5	Surface Properties.....	137
7.6	Bounding Volumes.....	138
7.7	Actuators.....	139
7.8	Actuator Database Interface.....	140
7.8.1	Selecting Actuator Components.....	141
7.8.2	Managing Database.....	146
7.9	Spring and Damper Properties.....	150
7.9.1	Approach.....	150
7.9.2	Implementation.....	151
7.9.3	Example.....	151
7.10	Child Links.....	152
7.11	Methods for Calculating Link Data.....	153
7.12	Example Code.....	154
7.12.1	Creating a Sphere-Shaped Link.....	154
8	The Manipulator.....	159
8.1	Reference Frames.....	160
8.2	Methods for Calculating Manipulator Data.....	161

8.3	Link and Manipulator References .....	162
8.4	Example Code.....	165
8.4.1	Creating a Bitmapped Base Link.....	165
8.4.2	Creating a Mechanism with One Joint .....	167
9	The Stated System.....	170
9.1	Description of EcStatedSystem .....	170
9.1.1	Description of EcManipulatorSystem .....	172
9.1.2	Description of EcManipulatorSystemState .....	174
9.2	Description of EcVisualizableStatedSystem .....	175
9.3	Example .....	176
10	Velocity Control.....	181
10.1	Algorithmic Description .....	181
10.1.1	Core Algorithmic Framework.....	181
10.1.2	Robust Extension .....	183
10.1.3	Reduced Control Calculation.....	184
10.1.4	End-Effector Error Filter.....	185
10.2	Implementation.....	188
10.2.1	Velocity Control System.....	188
10.3	Velocity Control Types .....	199
10.3.1	Singularity Avoidance .....	200
10.3.2	Torque Minimization .....	200
10.3.3	Collision Avoidance.....	201
10.3.4	Minimum Kinetic Energy Control .....	201
10.3.5	Minimum Potential Energy Control.....	201
10.3.6	Accuracy Optimization .....	201
10.3.7	Joint-Limit Avoidance .....	202
10.3.8	Strength Optimization .....	202
10.3.9	Statistical Error Reduction.....	204
10.3.10	Table Function .....	206
10.3.11	Composable Table Functions.....	210
10.4	End-Effector Descriptions .....	213
10.5	External-Force Optimization through Momentum Constraint.....	214
10.5.1	Organization.....	214
10.5.2	Derivation .....	215
10.5.3	Example .....	219

10.6	Control System Parameter Provision.....	219
10.6.1	Soft-Constraint End Effectors.....	220
10.7	Example Code.....	222
10.7.1	Minimum Potential Energy Control for the PUMA.....	222
11	Position Control.....	227
11.1	Position Control System Container.....	227
11.2	Position Control System.....	227
11.3	Position Control System With Look-Forward Simulation.....	230
12	Dynamic Simulation.....	234
12.1	Force Response.....	234
12.1.1	Architecture.....	234
12.1.2	Spring Displacement Model.....	235
12.1.3	Non-Conservative Forces.....	236
12.2	Articulated Dynamics.....	245
12.2.1	Composite Rigid-Body Inertia Simulation Algorithm.....	245
12.2.2	Articulated-Body Simulation Algorithm.....	250
12.2.3	Dynamics Example.....	252
12.3	Actuator Modeling.....	253
12.3.1	Dry (Stick-Slip) Friction for Actuators.....	253
12.3.2	Gear Backlash and Joint Elasticity.....	258
12.3.3	Power Conversion.....	261
12.4	Feedforward-Feedback Joint Controller.....	264
12.4.1	Feedback Proportional-Plus-Derivative Feedback Controller.....	264
12.4.2	Feedforward Dynamics.....	264
12.4.3	The Complete Implementation.....	267
12.4.4	Integration of Dynamic Simulation with the Kinematic Control System.....	267
12.5	Numerical Integration.....	269
12.5.1	Implementation.....	269
12.5.2	Integration of Base Motion.....	271
13	Collision Avoidance and Reasoning.....	274
13.1	Collision Avoidance Algorithm.....	274
13.2	Manipulator Self-Collision Avoidance.....	274
13.2.1	Example: Creating a Self-Collision Link Map.....	276
13.3	Manipulator-Manipulator Collision Avoidance.....	278
13.4	Manipulator-Environment Collision Avoidance.....	278



13.5	System Collision Exclusion .....	278
13.6	Collision Avoidance as a Function of Material Type .....	281
13.7	Collision Response .....	282
13.8	Example: Creating a Collision Avoidance Control Node.....	282
13.9	Distance Queries.....	285
13.9.1	Task Space Bounding Volumes .....	288
13.9.2	Bounding Volume Hierarchy .....	290
13.9.3	Traversing the hierarchy .....	291
13.9.4	Example: Adding a Bounding Volume Hierarchy to a shape .....	292
13.10	Penetration Depth Calculation.....	293
13.10.1	Sphere-Sphere .....	294
13.10.2	Capsule-Sphere .....	294
13.10.3	Capsule-Capsule .....	294
13.10.4	Box-Sphere .....	295
13.10.5	Box-Capsule.....	295
13.10.6	Lozenge-Sphere .....	295
13.10.7	Lozenge-Capsule.....	295
13.10.8	Lozenge-Lozenge.....	296
13.10.9	Cylinder-Sphere .....	296
13.10.10	Cylinder-Capsule .....	298
13.10.11	Cylinder-Half Space .....	302
13.10.12	Cone-Sphere .....	303
13.10.13	Cone-Capsule.....	305
13.10.14	Cone-Half Space.....	305
13.10.15	Implementations .....	306
13.11	Line Segment Intersection .....	307
13.11.1	Sphere .....	307
13.11.2	Cylinder .....	308
13.11.3	Capsule.....	310
13.11.4	Ellipsoid.....	310
13.11.5	Tetrahedron.....	311
13.11.6	Half Space.....	313
13.11.7	Oriented Box.....	314
13.11.8	Cone Frustum.....	316
13.11.9	Lozenge.....	317

13.11.10	Union .....	317
13.11.11	Intersection .....	318
14	Force Control and Grasping .....	319
14.1	Force Control .....	319
14.1.1	Design .....	319
14.1.2	Implementation .....	320
14.2	Grasping .....	323
14.2.1	Interface .....	323
14.2.2	Decision Tree .....	325
14.2.3	Organization of Each Grasping Algorithm .....	326
14.2.4	Prototype Grasp Creation .....	327
15	Data Capture .....	328
15.1	Path Saving and Following .....	328
15.1.1	State Path .....	328
15.1.2	Guide Frame Path .....	331
15.2	Storage and Display of Simulation Data .....	332
15.2.1	Design of Data Capture .....	332
15.2.2	Configuration Example .....	339
16	Studies .....	346
16.1	Parametric and Monte Carlo Studies .....	346
16.1.1	Design .....	346
16.1.2	Implementation .....	349
16.2	Simulation Visualization .....	353
16.3	Randomization and Monte Carlo Simulation .....	355
16.3.1	Randomization of System Properties .....	355
16.3.2	Randomization of State Variables .....	356
16.4	Mass Properties Randomization .....	357
16.4.1	Background .....	357
16.4.2	Mass .....	359
16.4.3	First Moment .....	360
16.4.4	Second Moment .....	362
16.4.5	Validity of Randomized Mass Properties .....	364
16.4.6	Ellipsoid Volume .....	364
16.4.7	Ellipsoids from Mass Properties .....	367
16.5	Coding Examples .....	368

16.5.1	Parametric Study.....	368
16.5.2	Monte Carlo Study.....	370
17	Rendering.....	371
17.1	Overview.....	371
17.2	EcWindow: Base Rendering Window Class.....	372
17.2.1	EcWindow::Impl.....	373
17.3	2D Rendering Windows.....	374
17.3.1	EcSimpleWindow.....	374
17.3.2	EcFBOWindow.....	375
17.4	3D Rendering Windows.....	376
17.4.1	EcSGWindow.....	376
17.4.2	EcRenderWindow.....	377
17.5	Qt-Based Classes.....	378
17.5.1	EcSGWidgetQt.....	378
17.5.2	EcCamera.....	378
17.5.3	EcBaseViewerMainWidget.....	380
17.5.4	EcBaseViewerMainWindow.....	380
17.6	User Input.....	381
17.6.1	EcSGBaseInputHandler.....	381
17.6.2	EcDefaultInputHandler.....	381
18	Plugin Interfaces.....	383
18.1	Control Plugins.....	383
18.2	Interface Plugins.....	385
18.2.1	EcPlugin.....	385
18.2.2	EcPluginManager.....	386
18.2.3	Viewer Plugins.....	386
19	Filters.....	387
19.1	Filter Enumeration.....	389
19.2	Filter Arguments Class.....	389
19.3	Filters Class.....	390
19.4	Filter Stream Class.....	391
19.5	Filter Inheritance.....	393
19.6	TEA Data Encryption Filter Details.....	393
19.7	Base 64 Encoding Filter Details.....	396
20	Network Operation.....	397

20.1	Low-level Sockets .....	398
20.1.1	Hierarchy .....	398
20.1.2	Base Socket Class .....	398
20.1.3	Tcp Socket .....	400
20.1.4	Udp Socket.....	402
20.2	Mid-Level Classes .....	403
20.2.1	Hierarchy .....	404
20.2.2	Protocols .....	404
20.2.3	Transports .....	409
20.3	CommFactory Utility Class .....	414
20.4	Complete Examples .....	415
20.4.1	Viewer.....	415
20.4.2	Tcp Client and Server with base64 encoding and bzip compression.....	416
20.4.3	Low-level UDP Example .....	419
21	Models and Other Format Loading.....	422
21.1	Overview of Converters.....	422
21.1.1	Point Polygon Format .....	423
21.2	Description for Adding New Converters .....	424
21.3	SolidWorks Plugin Converter.....	424
21.3.1	Model Setup.....	425
	Mass Properties .....	432
22	Analysis Tools .....	436
22.1	Using Simulink to Drive Actin with Desired End Effector Positions .....	436
23	Boost .....	447
1.1	What is Boost.....	447
1.2	Why use Boost.....	448
1.3	Overview of Boost libraries.....	448
1.3.1	Boost.Assign .....	448
1.3.2	Boost.Conversion .....	449
1.3.3	Boost.Filesystem .....	449
1.3.4	Boost.Foreach.....	450
1.3.5	Boost.Format .....	450
1.3.6	Boost.Iostreams .....	450
1.3.7	Boost.Numeric.....	451
1.3.8	Boost.Program_Options .....	452

1.3.9	Boost.Regex .....	454
1.3.10	Boost.Signals .....	455
1.3.11	Boost.Smart_Ptr .....	456
1.3.12	Boost.Thread .....	457
24	Bibliography .....	462

# 1 Actin™ Capabilities Overview

You are probably reading this because you want to control, design, or validate a robotic mechanism. There are tasks you want to accomplish, and you would like programming tools to support you. This is what the Energid Actin™ toolkit provides. Actin™ is an intuitive C++ software toolkit for manipulator control and simulation. Actin has three components that work together to meet your needs. Actin Control™, Actin Design™, and Actin Validate™. It allows you to control, design, and validate complex robotic mechanisms.

Actin Control's primary purpose is to calculate joint positions and rates that set robotic end effectors where you want them. It provides tools for geometric reasoning. It supports cooperation of multiple robotic manipulators. And it provides ways to capture and reason with camera imagery. It supports three-dimensional rendering and can be used for network TCP/IP communications that provide control.

Actin Design's primary purpose is enable designers with only initial ideas to bring them to practical reality as quickly as possible. Actin Design provides a library of actuators and other robotic components that can be used to quickly construct models for testing using Actin's control systems. It includes a plugin to other design tools, such as SolidWorks, that enable fast testing integrated with designers' favorite tools. Actin Design supports quick kinematic and dynamic tests to be combined with outmated control system construction.

Actin Validate's purpose is to assess the performance of complete robot designs for application to particular tasks. It can be used to kinematically and dynamically simulate physical environments. It also supports Monte Carlo simulation and Parameter Optimization analysis. Included are high-fidelity articulated dynamics and impact dynamics.

This section gives an overview of the capabilities in all of these packages. Specific capabilities to each version of Actin are provided in the package-specific documentation. Each package of Actin™ includes a set of libraries and header files that can be used with your C++ project to easily add manipulator-control and simulation capability. These toolkit components can be integrated into your existing code or used to build a new program. Both Linux and Windows are supported.

## 1.1 Extensible Markup Language (XML)

Components are configurable using XML, and you can easily connect your code with components from the Actin toolkit to build XML-configurable C++ objects. In addition to reading and writing themselves in XML, all XML-configurable objects can write their own validating schemas. So if you use the Actin™ toolkit to build your system, you will also be designing an XML language that can be used with other commercial software products.

## 1.2 Mathematical and Geometrical Tools

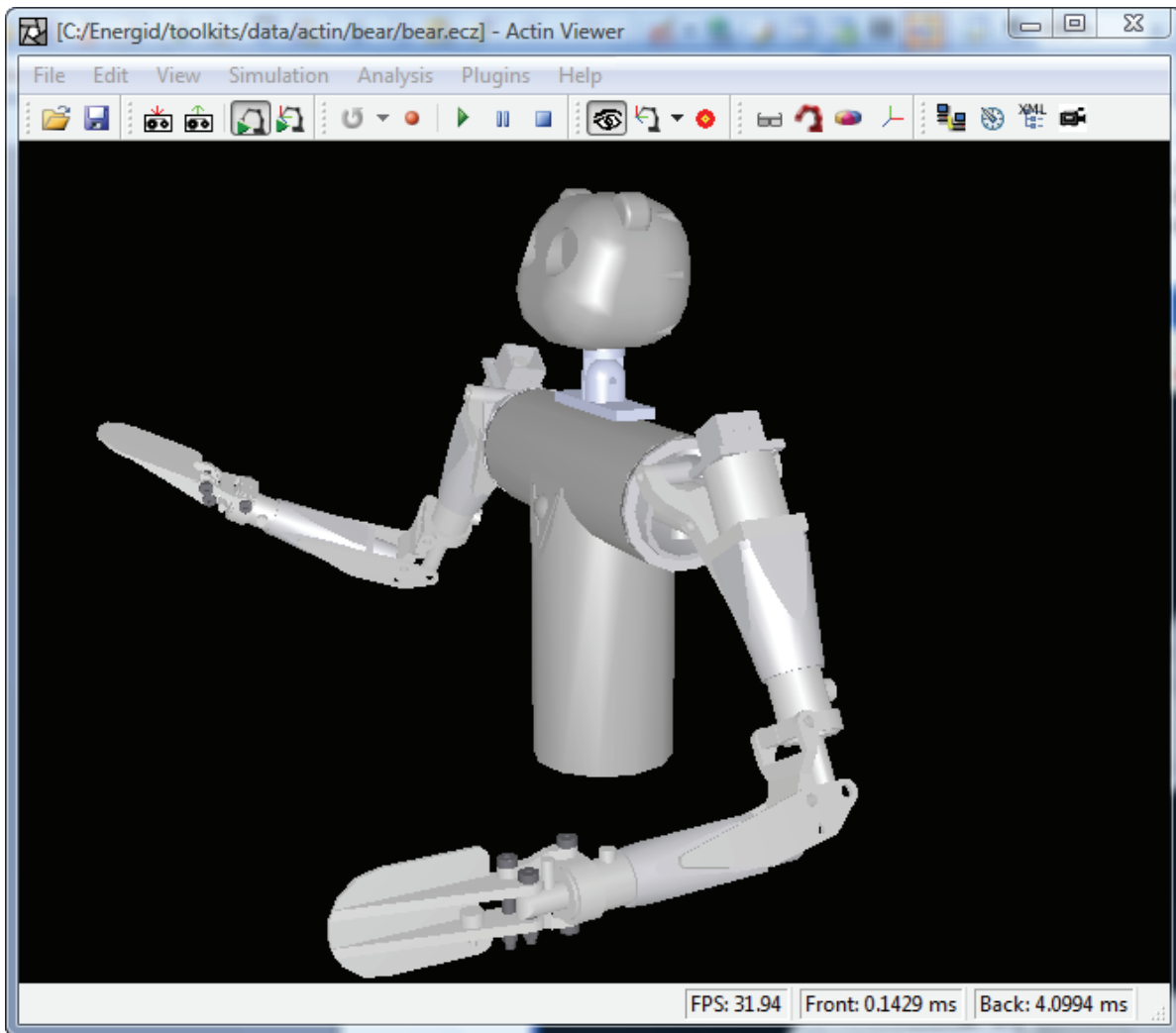
The Actin™ toolkit includes a number of tools for easy and efficient mathematical and geometric calculation. These include three-dimensional vector math and matrix routines. Conversion utilities for three-dimensional quantities are included. Orientations can be set from quaternions, Euler angles, Rodrigues parameters, angle-axis, direction cosine matrices, and so forth. These are all optimized for performance. With the Actin™ toolkit, you do not have to re-implement these basic functions.

### 1.3 Automatic Kinematic Control

Actin™ calculates the joint rates or positions to give desired hand velocities or positions. All is done automatically, based only on the manipulator model description. This is the strength of the Actin™ toolkit—the ability to control almost any robotic manipulator using just its kinematic description. Manipulators with any number of links, any number of bifurcations (branches), nonstandard joint types, and nonstandard end-effector types are supported.

### 1.4 Dynamic Simulation

Actin™ provides dynamic simulation capability. This includes full and accurate Newton-Euler rigid body dynamics on all articulated links and impact dynamics between obstacles. Dynamics are calculated for nontraditional joint types, as well. Both the Composite Rigid Body Inertia (CRBI) algorithm and the Articulated Body Inertia (ARBI) algorithm are implemented. The CRBI algorithm is an Order( $n^3$ ) method, which is efficient for mechanisms with few—less than 15 or so—degrees of freedom (DOF), while the ARBI algorithm is an Order( $n$ ) method, efficient for high-DOF mechanisms.



**Figure 1-1:** Dyanmic simulation of a humanoid robot. This robot model is provided courtesy of Vecna Robotics.

## **1.5 Parametric Studies**

Actin provides capability for parametric and Monte Carlo studies. A parametric takes discrete steps through changes in initial state or system parameters and tabulate simulation results. The design of the parametric study includes 1) representation changes to the initial state and system, and 2) a representation of the results of the simulation runs. A parametric study will allow the user to easily change in fixed increments initial configurations, control parameters, surface properties, weights, lengths, end effectors, motor torques, and actuator effectiveness, and tabulate the results of those changes. Results include measures of sensor saturation, visibility, speed, mobility, balance, end-effector placement, and manipulation.

## **1.6 Monte Carlo Studies**

A Monte Carlo study is performed by selecting random initial values for the system and state parameters. In addition, noise is input to sensor and actuator models. The noise models for the sensors and actuators is built into the classes that define them. The initial conditions for the system state are selected based on a set of probability density functions, as are the selected values for a time sequence of desired end-effector positions. In Actin, Monte Carlo studies can be used to perform parameter-optimization analysis to determine the best design values.

## **1.7 Rendering**

Actin™ provides cross-platform rendering and visualization capability. Any manipulator can be viewed through an easy-to-use interface that pops up a window with an animation. Any number of manipulators can be shown in the visualization. The specular properties of polygons can be set, polygons can be bit mapped, and any number of lights can be configured. These tools provide capability for intuitive debugging and for creating human-machine interfaces for remote supervision and teleoperation.

## **1.8 Machine Vision**

Actin™ includes methods for capturing images with a USB camera, firewire camera, or frame grabber. It also includes algorithms for analyzing captured images and using the results as information to feed back to the controller. The toolkit includes camera calibration algorithms that allow for the automatic calculation of camera parameters, such as focal length and position/orientation. These tools provide capability for making vision-based robotic control systems.

## **1.9 Network Communications**

The toolkit includes C++ classes for network communications. Sockets are implemented both for TCP/IP and UDP/IP communications. A networking stream class is implemented to allow the transmission of XML data from one network location to another. This allows front-end and back-end components to be implemented on different computers for remote supervision and teleoperation.

## **1.10 Third-Party Integration**

Actin supports integration with a variety of third-party software. It includes plug-in support for SolidWorks, integration with Matlab Simulink, integration with Skype, and the ability to load formats from 3D Studio Max and VRML.



## 2 Quick Start

If you would like to just start quickly and cover the details later, this section is for you. Example code will be presented for creating, writing, and reading a position control system for a Robotics Research Corporation RRC K-1207i manipulator.

### 2.1 Loading a Stated System

An example manipulator system, the RRC K-1207i, is defined in an example file that is distributed with the Actin™ software. This file, “rrcK1207i\_system.xml,” contains the description of an object of the class *EcStatedSystem*. To load this file, the C++ code in Text Box 2-1 can be used.

```
// declare an error return code
EcBoolean success;

// add the file location to the search path
EcFileUtil::addDirectory(EcFileUtil::getDataDirectory() +
    EcString("/actinExamples"));

// declare a filename
EcString filename="rrcK1207i_system.xml";

// declare a visualizable stated system object
EcVisualizableStatedSystem visStatedSystem;

// load the stated system from an XML file
success = visStatedSystem.readFromFile(filename);

// make sure it loaded properly
if(!success)
{
    EcWARN("Could not load stated system.\n");
    return;
}
```

**Text Box 2-1:** Example code for loading a manipulator description. This is Example Section #1 in the quick-start example code.

### 2.2 Viewing a Stated System

To see what this manipulator looks like, you can load the file “rrcK1207i\_system.xml” using the viewer that is provided with the toolkit. To enable viewing in your own code, you can use the class *EcRenderWindow*, as shown in Text Box 2-2 below

```

// instantiate a renderer
EcRenderWindow renderer;

// set the size of the window
renderer.setWindowSize(256,256);

// set the system
if(!renderer.setVisualizableStatedSystem(visStatedSystem))
{
    return;
}

// view the system
renderer.renderScene();

// pause one second
EcSLEEPMS(1000);

```

**Text Box 2-2:** Example code for visualizing the manipulator. This continues from the code shown in Text Box 2-1. It is Example Section #2 in the quick-start example code.

## 2.3 Defining a Velocity-Control System

The next task is to define a velocity control system. This is done using the code shown in Text Box 2-3. In this set of code, first a convenient reference is set for the manipulator. Then the link at the end of the kinematic chain starting with the manipulator is looked up. This is used to construct a frame end effector that is rigidly attached to this last link.

A velocity-control expression is constructed. An *EcControlExpressionCore* object is chosen to perform the inverse kinematics calculation. This object requires three child members: a matrix, a vector, and a scalar. The matrix is used to measure the joint rates, and the vector is used to determine desired change in joint values. The scalar parameter makes a tradeoff between the joint-rate measure and the joint value measure. In this case, the mass matrix is used to weight the joint rates and joint-limit avoidance is used to determine desired joint positions.

To this core velocity control algorithm, a joint-rate filter and an end-effector error filter are added. These prevent the joint rates and hand-motion error from exceeding specified bounds. Any number of filters can be chained together as shown to build a general control expression. The control expression and the end-effector definition are then added to the individual velocity control description. Since in this case there is only one manipulator being controlled, the velocity control system only has one description.

```

// make a convenient reference to the manipulator
const EcIndividualManipulator& manipulator =
    visStatedSystem.statedSystem().system().manipulators()[0];

// look up the last link
EcManipulatorLinkConstPointerVector linkPointerVector;
manipulator.collectLeafLinks(linkPointerVector);

// make a frame end effector
EcFrameEndEffector frameEnd;

// put the end effector into an end-effector set
frameEnd.setLinkIdentifier(
    EcXmlString(linkPointerVector[0]->label()));
frameEnd.setFrame(EcCoordinateSystemTransformation());
EcEndEffectorSet eeSet;
eeSet.addEndEffector(frameEnd);

// create a velocity-control core
EcControlExpressionCore expCore;

// set the matrix, vector, and scalar for the core
expCore.setMatrixElement(EcControlExpressionMassMatrix());
expCore.setVectorElement(EcControlExpressionJointLimitAvoidance());
expCore.setScalarElement(
    EcExpressionScalarConstant::objectWithValue(-1.0));

// add a joint-rate filter
EcControlExpressionJointRateFilter rateFilter;
EcExpressionGeneralColumn jointWeights;
jointWeights.assign(manipulator.jointDof(), 0.1);
rateFilter.setWeightsElement(jointWeights);
rateFilter.setUnfilteredRatesElement(expCore);

// add an end-effector error filter
EcControlExpressionEndEffectorErrorFilter eFilter;
EcExpressionGeneralColumn handWeights;
handWeights.assign(6, 1.0);
eFilter.setWeightsElement(handWeights);
eFilter.setUnfilteredRatesElement(rateFilter);
eFilter.setStopsAtLimits(EcTrue);

// put the system in a container and add it to the velocity control
// description
EcControlExpressionContainer container;
container.setTopElement(eFilter);

```

```

// add the expression and end effector to a velocity control
// description
EcIndividualVelocityControlDescription indVelContDesc;
indVelContDesc.setControlExpression(container);
indVelContDesc.setEndEffectorSet(eeSet);

// add the velocity control description to a velocity control
// system
EcVelocityControlSystem velContSys;
velContSys.addControlDescription(indVelContDesc);

```

**Text Box 2-3:** Example code for building a velocity control system. This continues from the code shown in Text Box 2-2 and is Example Section #3 in the quick-start example code.

## 2.4 Defining a Position-Control System

The velocity control system can now be added to a position control system. This is done using the code shown in Text Box 2-4. In this set of code, the velocity control system is set, the time step is set, and the maximum number of iterations per cycle is set (this is only used if the CPU is not able to run in real time). Then the two-pass flag is set. This generally should be used, as it protects the manipulator from inappropriate behavior at singularities.

```

// a variable holding the position control system
EcPositionControlSystem posContSys;

// set the velocity control system
posContSys.setVelocityControlSystem(velContSys);

// set the time step
posContSys.setTimeStep(0.012);

// set the maximum number of iterations
posContSys.setMaxIterations(16);

// set the use-two-passes flag
posContSys.setUseTwoPasses(EcTrue);

// set the stated system
posContSys.setStatedSystem(&visStatedSystem.statedSystem());

```

**Text Box 2-4:** Example code for building a position control system. This continues from the code shown in Text Box 2-3 and is Example Section #4 in the quick-start example code.

## 2.5 Placing an End Effector

The position control system can now be used to place an end effector. This is done using the code shown in Text Box 2-5. First the current pose of the end effector is calculated. This is offset in position while keeping the same orientation. This new hand pose is then set as the desired hand pose, and the position control system is polled for a new manipulator configuration every 20 milliseconds. The configurations are those calculated by the position control system to take the end effector to the desired location. Each of these configurations is rendered, and at the end, convergence to the desired pose is verified.

```

// get the current offset in system coordinates
EcCoordinateSystemTransformation
    initialPose=posContSys.actualPlacement(0,0);

// set the desired position to be offset from the current position
// by 0.5 m along the x-axis and 1.2 m along the z-axis
EcCoordinateSystemTransformation finalPose = initialPose;
finalPose.outboardTransformBy(EcVector(-0.5,0.0,-1.2));
posContSys.setDesiredPlacement(0,0,finalPose);

// a state to update and render, and an object to hold the
// final pose
EcManipulatorSystemState dynamicState;
EcCoordinateSystemTransformation calculatedFinalPose;

// set the system
if(!renderer.setVisualizableStatedSystem(visStatedSystem))
{
    return;
}

// execution parameters
EcU32 steps=100;
EcReal simRunTime = 2.0;
EcReal simTimeStep = simRunTime/steps;

// move to the desired pose, and render every timestep
for(ii=0;ii<steps;++ii)
{
    // get the current time
    EcReal currentTime=simTimeStep*ii;

    // calculate the state at current time
    posContSys.calculateState(currentTime,dynamicState);

    // show the manipulator in this position
    renderer.setState(dynamicState);
    renderer.renderScene();
    EcSLEEPMS(static_cast<EcU32>(1000*simTimeStep));
}

// check for accuracy
calculatedFinalPose = posContSys.actualPlacementVector()[0].
    offsetTransformations()[0];
if(!calculatedFinalPose.approxEq(finalPose,1e-5))
{
    EcWARN("Did not converge.\n");
    return;
}

```

**Text Box 2-5:** Example code for placing an end effector. This continues from the code shown in Text Box 2-4 and is Example Section #5 in the quick-start example code.

## 2.6 Moving the End Effector along a Path

The code shown in Text Box 2-6 uses the position control system to trace a path with the end effector. It sets a new, but fixed, orientation and a new, changing, position each time step. The orientation is about 180 degrees away from the starting orientation, and the position is a point along a circle with a 0.2 m radius. The end effector makes three loops around the circle. An image of what you should see when running this code is given in Figure 2-1.

```
// execution parameters
steps=400;
simRunTime = 10.0;
simTimeStep = simRunTime/steps;
EcReal radius=0.2;
EcU32 loops=3;
EcOrientation orient(0,0,0,1);
EcReal startingTime=posContSys.time();

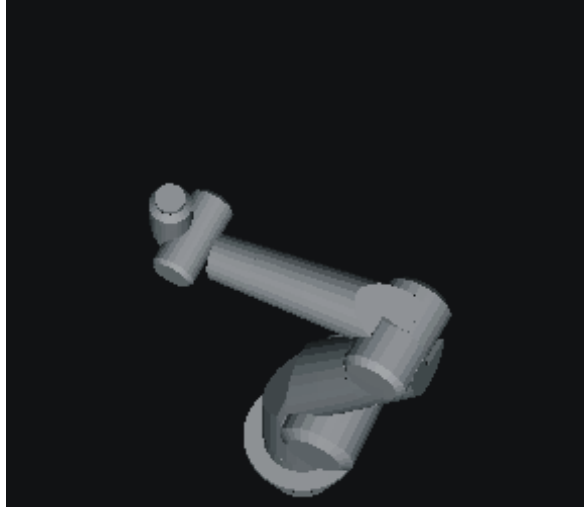
// move to the desired pose, and render the position every
// time step
for(ii=0;ii<steps;++ii)
{
    // get the current time
    EcReal currentTime=simTimeStep*ii;

    // set the pose
    EcCoordinateSystemTransformation pose;
    pose.setOrientation(orient);
    EcReal angle=Ec2Pi*loops*currentTime/simRunTime;
    EcVector offset=radius*EcVector(cos(angle),sin(angle),0);
    pose.setTranslation(finalPose.translation()+offset);
    posContSys.setDesiredPlacement(0,0,pose);

    // calculate the state at current time
    posContSys.calculateState(
        currentTime+startingTime,dynamicState);

    // show the manipulator in this position
    renderer.setState(dynamicState);
    renderer.renderScene();
    EcSLEEPMS(static_cast<EcU32>(1000*simTimeStep));
}
}
```

**Text Box 2-6:** Example code for tracing a path with the end effector. This continues from the code shown in Text Box 2-5 and is Example Section #6 in the quick-start example code.



**Figure 2-1:** An image of the RRC K-1207i tracing the circle when the example code in Text Box 2-6 is run. When the example code executes, a window pops up showing this image.

## 2.7 Creating a Simulation

A simulation can be created using the position control system. This is shown in the code in Text Box 2-7. A simulation animates and—optionally—dynamically simulates the manipulator. A simulation can be saved as XML in plain or compressed format. Both forms are saved in the example code below. Either form can be loaded by the ActinViewer that is provided with the toolkit and used for interactive manipulation. A screen capture of the ActinViewer showing an interactive session started by loading “quickStartSimulation.xml.gz” is given in Figure 2-2.

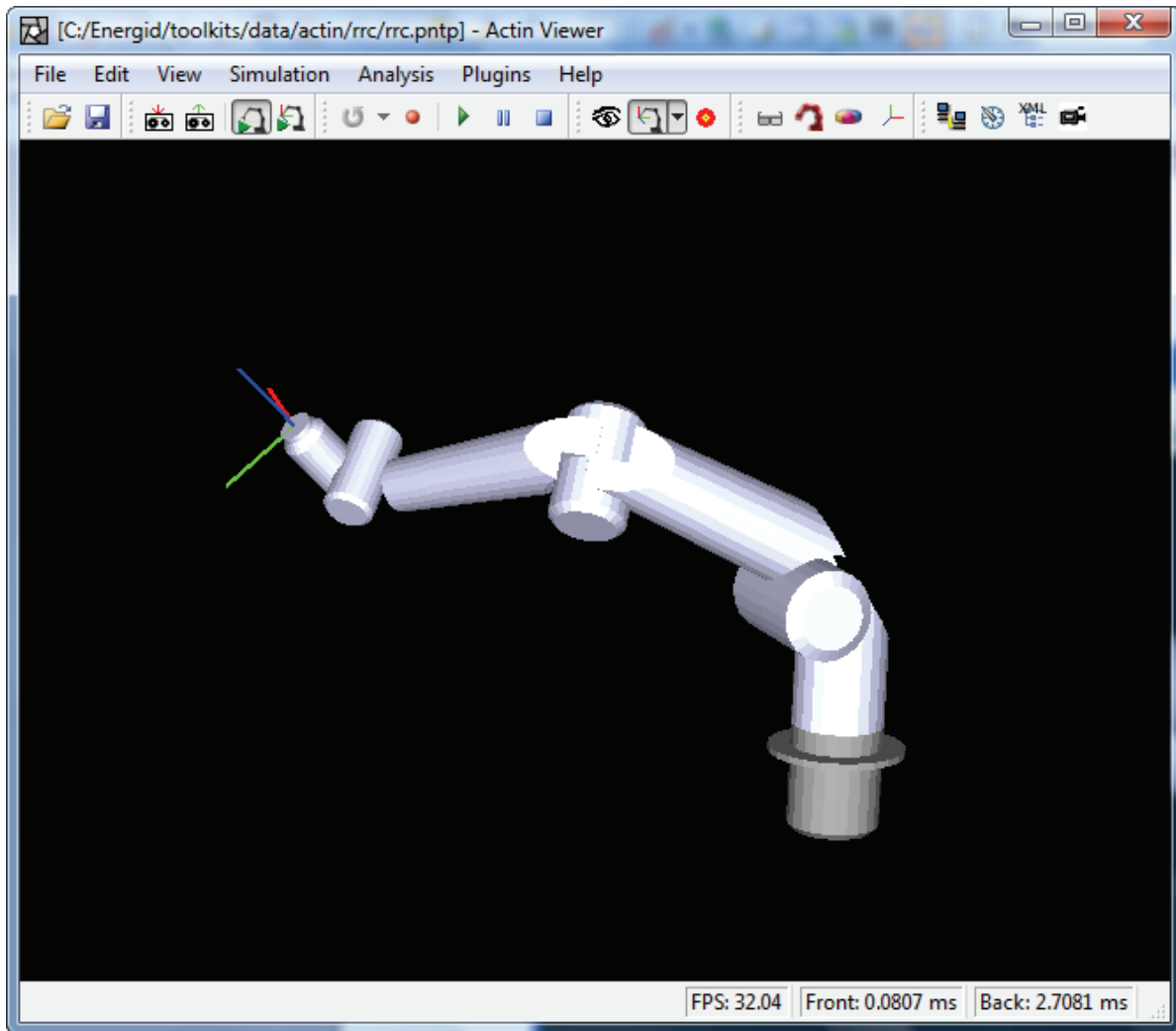
```
// a variable holding the simulation
EcSystemSimulation simulation;

// reset time to zero and add the position control system and
// visualization parameters
posContSys.setTime(0.0);
simulation.setFromPositionControlSystem(posContSys,
    visStatedSystem.visualizationParameters());

// save the simulation as a plain XML file
simulation.writeToFile("quickStartSimulation.xml");

// save the simulation as a compressed XML file
simulation.writeToFile("quickStartSimulation.xml.gz");
```

**Text Box 2-7** Example code for saving the system as a simulation. A simulation can be loaded and interactively controlled using the ActinViewer. This continues from the code shown in Text Box 2-6, and is Example Section #7 in the quick-start example code.



**Figure 2-2:** The ActinViewer can be used to load the simulation files created with Text Box 2-7. With the viewer, you can interactively place the end effector (by moving and rotating the red-green-blue frame shown), and the control system will automatically place the joints. This can be done with or without dynamic simulation. You can also change the view parameters, and save and replay paths.

## 2.8 Additional Capability

This quick-start example showed how to assign a control system to a manipulator. Only one manipulator was used with one end effector and one type of control system. The manipulator itself, which was a simple linear chain, was not defined, but rather loaded from a file. Dynamic simulation was not used. Network communication was not used, nor was visual feedback. The sections to follow describe these and the many other capabilities present in the Actin™ toolkit.

## 2.9 Building

To build a new application, the general process requires setting up the compiler options, setting up the linker options, and linking in the appropriate libraries.



## **2.9.1 General Setup**

The build instructions require that the environment variable named `EC_TOOLKITS` be specified. This variable should point to the location of the toolkits directory of the Actin install. If an Actin installer CD was used to install Actin, then the environment variable should already be defined.

## **2.9.2 Compiler Setup**

To compile the application source files, the compiler's include path must be defined, and several flags must be specified.

### **2.9.2.1 Include Path**

The include path must point to the Actin headers. If the `EC_TOOLKITS` environment variable is defined, then the include path can simply be set to “`{EC_TOOLKITS}\include`” in the project settings.

### **2.9.2.2 Flags**

There are six compiler flags that must be added to the project settings. A list of these flags follows:

- `/DBOOST_ALL_NO_LIB`
- `/DNOMINMAX`
- `/D_CRT_SECURE_NO_DEPRECATED`
- `/D_SCL_SECURE_NO_WARNINGS`
- `/DWIN32_LEAN_AND_MEAN`
- `/D_WIN32_WINNT`

Both `/D_CRT_SECURE_NO_DEPRECATED` and `/D_SCL_SECURE_NO_WARNINGS` are not strictly required for compilation; however, they remove warnings that can safely be ignored. Of all of the flags, only `/D_WIN32_WINNT` requires a value. It should be set to `0x501` if building on Windows XP, and it should be set to `0x600` if building on Windows Vista.

## **2.9.3 Linker Setup**

In order to link the application binary files, the linker's library path must be defined. Additionally, a linker flag is required.

### **2.9.3.1 Library Path**

The library path must point to the Actin libraries and their dependencies. If the `EC_TOOLKITS` environment variable is defined, then the library path can simply be set to “`{EC_TOOLKITS}\lib`” in the project settings.

### 2.9.3.2 Flags

In order to prevent linker errors, the flag /NODEFAULTLIB:MSVCRT must be specified in the project settings.

## 2.9.4 Libraries

It is important to determine which libraries need to be linked into the application. Although it is possible to determine the dependency libraries by trial-and-error, this section should help reduce the guess work.

### 2.9.4.1 Library Descriptions

The following table provides a short description for all of the libraries bundled with Actin.

Library Name	Source	Description
cluster	Energid	Library for modeling cluster
control	Energid	Library of controlling manipulators
convertSimulation	Energid	Library for converting different 3D model formats into a simulation
convertSystem	Energid	Library for converting different 3D model formats into a system
date_time	Boost	Library for easily working with dates and times
dynamicSystems	Energid	Library for modeling dynamic systems
excelXml	Energid	Library for working with Excel worksheets
filesystem	Boost	Library for working with file paths
filterStream	Energid	Library of filtered streams
foundCore	Energid	Library containing highly-reusable foundational components
function	Energid	Library containing utility functors
geometry	Energid	Library for composing complex 3D geometries
grasping	Energid	Library for modeling grasping algorithms
hardwareInterface	Energid	Library for working with serial connections
imageSensor	Energid	Library for modeling image sensors
inputDevice	Energid	Library for modeling input devices
interface	Energid	Library of simulation interfaces
iostreams	Boost	Library for creating filtered streams
manipulator	Energid	Library for modeling robotic manipulators
matrixUtilities	Energid	Library for working with matrices

measure	Energid	Library for measuring manipulators
program_options	Boost	Library for creating command line interfaces
regex	Boost	Library for native C++ regular expressions
serial	Energid	Library for working with serial connections
serialization	Boost	Library for easily serializing objects to/from streams
signals	Boost	Library for creating callbacks though signals and slots
simulation	Energid	Library for creating dynamic simulations
simulationAnalysis	Energid	Library for analyzing simulation results
simulationStudy	Energid	Library for creating simulation studies
socket	Energid	Library of higher-level socket classes
stream	Energid	Library of higher-level stream classes
thread	Boost	Library for creating multi threaded applications
transport	Energid	Library for transporting objects through a Skype connection
unitTestFramework	Boost	Library for unit testing software
visualization	Energid	Library of classes related to 3D visualization
vrml97	Energid	Library for working with Vrml97 3D file formats
walking	Energid	Library for modeling walking robots
xml	Energid	Library for marshaling objects to/from xml

**Table 2-1: The libraries bundled with Actin.**

### 2.9.4.2 Naming Convention

All of the Actin libraries contain debug and release variants. It is important that debug libraries are linked to debug applications and release libraries are linked to release applications. It is easy to distinguish the debug and release libraries, as the debug libraries have a “d” appended to the library name. For instance, the release version of the foundCore library is foundCore.lib, and the debug version of the library is foundCored.lib.

### 2.9.4.3 Dependency Chain

When linking in a library, it is necessary to also link in dependency libraries. The following table shows the dependency chain.

Library Name	Dependencies
cluster	xml
control	measure simulation walking

convertSimulation	convertSystem simulation
convertSystem	manipulator vrml97
date_time	NONE
dynamicSystems	matrixUtilities
excelXml	xml
filesystem	NONE
filterStream	iostreams foundCore
foundCore	thread
function	matrixUtilities
geometry	manipulator visualization matrixUtilities
grasping	control
hardwareInterface	xml
imageSensor	manipulator
inputDevice	manipulator
interface	control
iostreams	NONE
manipulator	convertSystem function geometry imageSensor
matrixUtilities	xml
measure	control manipulator
program_options	NONE
regex	NONE
serial	foundCore
serialization	NONE
signals	NONE
simulation	control convertSimulation grasping simulationAnalysis simulationStudy
simulationAnalysis	simulation
simulationStudy	manipulator signals
socket	foundCore
stream	socket
thread	NONE
transport	regex signals foundCore filterStream socket
unitTestFramework	NONE

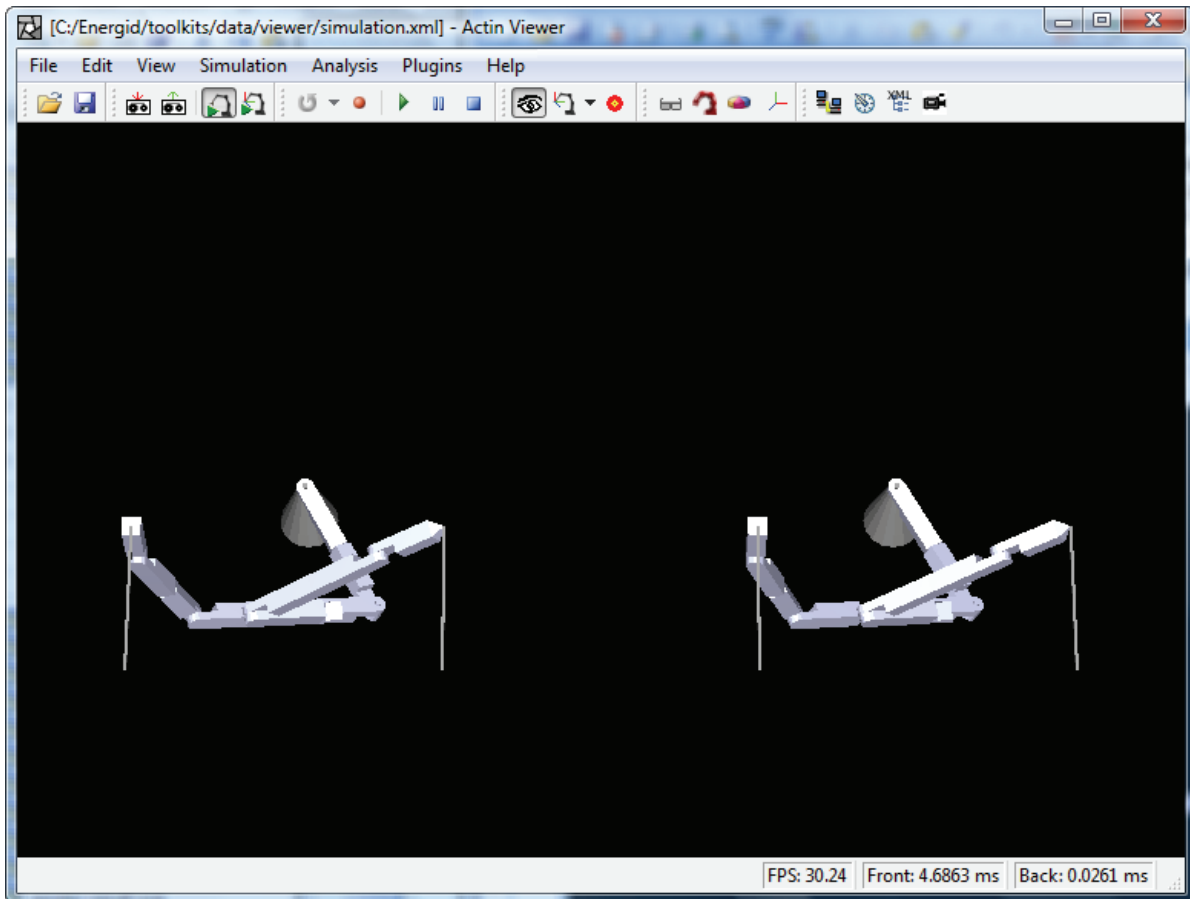
visualization	geometry
vrml97	foundCore
walking	control
xml	stream filterStream

**Table 2-2: Library dependencies.**

From recursive application of the table, an application that depends on the stream library will also need socket, foundCore, and thread.

### 3 ActinViewer

Energid’s ActinViewer serves as both a useful tool and an introduction to the Actin toolkit, as it is built using the toolkit and all the capabilities of ActinViewer are available through the toolkit. ActinViewer is a visualization tool for running simulations. ActinViewer can be used to both simulate motion of robots and to directly control physical robots if properly set up. This chapter is intended to be a complete guide for ActinViewer. ActinViewer is built on top of the Actin toolkit. For a graphical interface, it uses Qt so it is cross-platform and has been tested to run on Windows XP, Windows Vista, and many variants of Linux. Figure 3-1 shows ActinViewer in action with a system of two 12-DOF manipulators.



**Figure 3-1:** The system with two manipulators used to demonstrate ActinViewer features.

## 3.1 File Options

The following options are related to file operations.

### 3.1.1 Open

To open an Actin model file, select Open from the File menu or press Ctrl+O. This will bring up a File Open dialog. You can also open another file format and ActinViewer will convert that file into an Actin file format. The list of supported file formats is given in the table below.

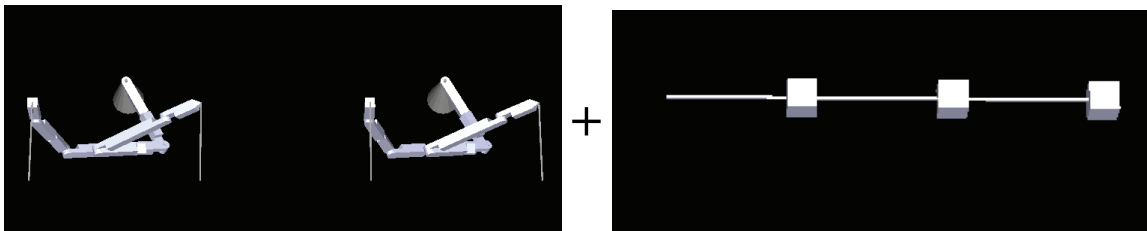
Supported format	File extensions
Energid	*.ecx; *.ecz
Xml	*.xml; *.xml.gz
3DS	*.3ds
ASE	*.ase
CFG	*.cfg
DTED	*.dted
SDTED	*.sdted
S3DS	*.s3ds
SASE	*.sase
Tecplot	*.stec
VRML97	*.wrl
OpenSceneGraph	*.osg; *.ive; *.zip
OpenFlight	*.flt
VEC	*.vec

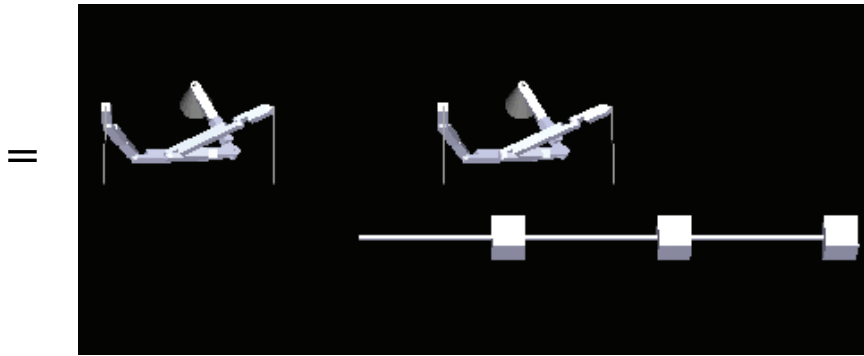
**Figure 3-2:** File formats supported by ActinViewer.

Another way to open a file is to drag a file from Windows Explorer (or equivalent program under Linux) and drop it inside ActinViewer. The drag-and-drop mechanism only allows supported file formats (determined by the file extension) to be dropped and opened in ActinViewer.

### 3.1.2 Merge

The merge operation allows another Actin model to merge with the currently open model. This allows you to combine two or more Actin models together to create a new model. With an open model, simply select Merge from the File menu or press Ctrl+M. This will bring up a File Open dialog so you can select the file to merge. An example of the merge operation of two Actin models is illustrated in the figure below.





**Figure 3-3:** Merge operation with two Actin models.

### 3.1.3 State

There are two options for state actions. The first option is “Load State ...” This will bring up a dialog that allows the user to load a state saved in a file into the simulation. This will cause all the manipulators in the system to move to the loaded state. The second option is “Save State ...” This allows the user to save the current state of the system so that it can be loaded later.

### 3.1.4 Save Operations

#### Save (Ctrl+S)

This option saves the currently open Actin model into the same file. If the model is not an Actin model, then it will bring up a dialog to let the user save the model as an Actin model in a new file.

#### Save Image As

Save a snapshot of the current view of ActinViewer. Currently .tif is the only supported image format.

#### Save Depth Buffer As

Save a snapshot of the depth buffer of the current view of ActinViewer. Currently .tif is the only supported image format.

## 3.2 Mouse Interaction Modes

Mouse interaction is an important part of ActinViewer. ActinViewer can be placed into one of three different mouse interaction modes, Eyepoint, Guide, or Center of Interest (COI).


### 3.2.1 Eyepoint Mode

The default mode is Eyepoint. This allows you to move your point of view around the object. Pressing the left mouse button and dragging changes the rotation about the center of interest. Using one of the following combinations will increase or decrease the eyepoint distance from the center of interest – scroll wheel, middle mouse button while dragging up/down, or left+right mouse buttons while dragging up/down.

While in this mode, you may temporarily switch to either guide or COI mode by pressing and holding down the Shift or Ctrl key respectively.

*Note - The eyepoint is always looking and rotating about the center of interest.*

### 3.2.2 Guide Mode

Guide mode places a set of red, green, and blue unit axes  into the scene to denote X, Y and Z respectively. This is used to ‘guide’ a selected end-effector to a desired position and orientation within the scene.

On first press, the Select End-Effector dialog box will be displayed to select an end-effector (EE). More detail on the Select End-Effector dialog can be found in the next section when we discuss controlling end-effectors. The currently selected EE will be used when in guide mode. To select a different EE, use the pull-down menu just to the right of the guide button icon.

This mode lets you translate (left mouse button) and rotate (right mouse button) as well as zoom in and out (scroll wheel / middle mouse button).

Pressing the Shift key will momentarily place you in Eyepoint mode.

*Note – if you are using a point end-effector, then rotating the guide frame will have no effect on EE placement. In addition, end-effector movement only occurs while the simulation is running. Press the Play button on the Simulation toolbar to start or continue the simulation.*

### 3.2.3 Center of Interest (COI) Mode

Sometimes it is desirable to focus on one particular area over another. Moving the COI allows one to more easily manipulate the eyepoint around this location. The COI is denoted by a reddish-orange sphere within the scene. The left mouse button will translate the COI along the X and Y axes, while the middle mouse wheel / button will translate along the Z axis.

Pressing the Shift key while in this mode will temporarily place you in Eyepoint mode.

## 3.3 Information Overlays

In ActinViewer, you can display some information visually by overlaying it on to the robots in the system. In this section, a model of the 7-DOF Mitsubishi PA-10 manipulator will be used for illustration purposes.



**Figure 3-4:** PA-10 manipulator with no overlay.



### 3.3.1 Bounding Volumes

Pressing the Bounding Volumes button on the toolbar shows the bounding volumes of all links of all manipulators in the system. Bounding volumes are displayed in a transparent green color. Bounding volumes can be of any shape but the most common is the capsule. Bounding volumes are used in collision reasoning to speed up distance calculations.



Figure 3-5: PA-10 manipulator with bounding volumes.

### 3.3.2 Mass Property Ellipsoids

Pressing the Mass Property Ellipsoids button will overlay the mass property ellipsoids onto the links of manipulators. This is a great visualization tool to help verify whether the mass properties (inertia matrices) are reasonable.

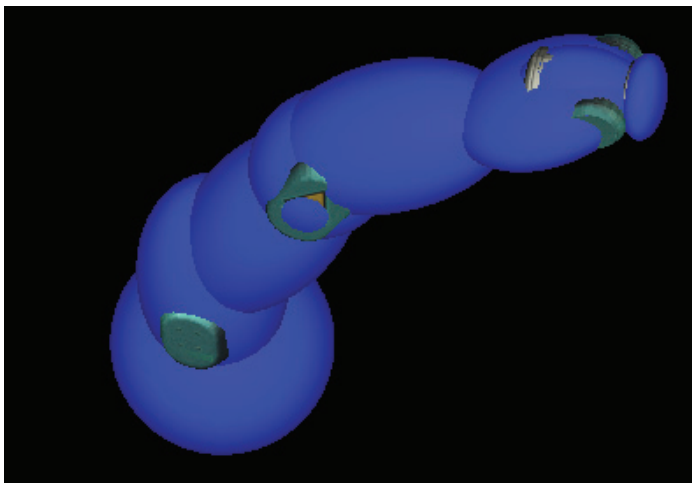
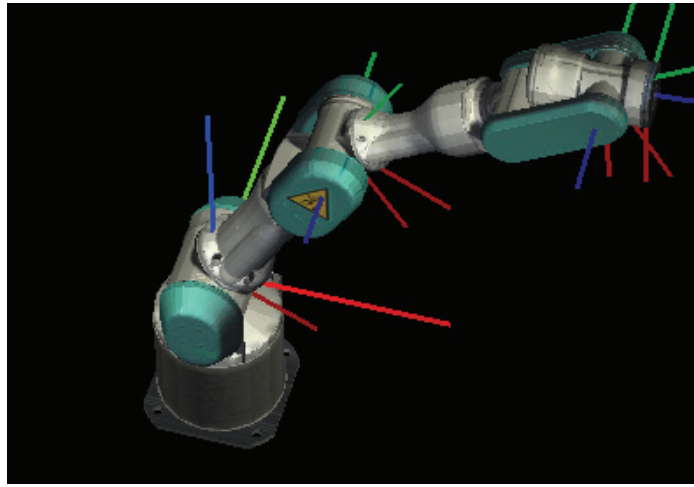


Figure 3-6: PA-10 manipulator with mass property ellipsoids.

### 3.3.3 Show Frames

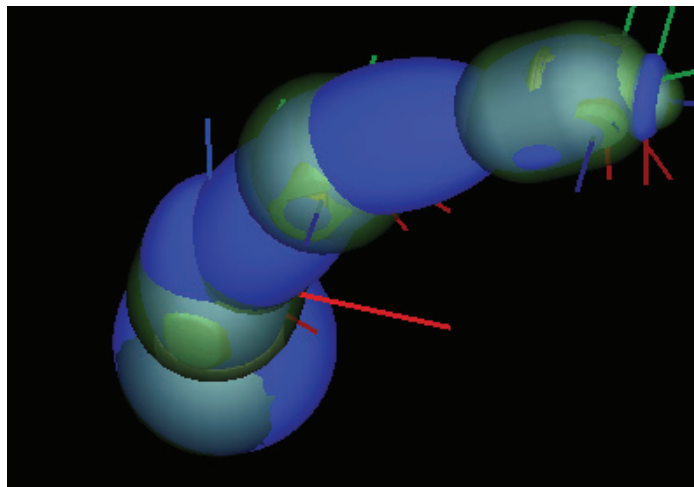
Sometimes it is very insightful to be able to see all the frames (primary and D-H) of the robot. Clicking on the Show Frames button will show all those frames as well as the world coordinate frame. The world coordinate frame is typically designated by the frame with the longest axes. For all

frames, the red, green, and blue axes represent the X, Y, and Z axes, respectively. The primary frame and the D-H frame of a link can occasionally coincide so they will appear as one frame. This is the case for all links of the PA-10 manipulator as shown below.







**Figure 3-7:** PA-10 manipulator with primary and D-H frames.

Naturally, these information overlays can be turned on/off independent of one another and therefore can be combined as shown in Figure 3-8



**Figure 3-8:** PA-10 manipulator with all information overlays.

### 3.4 Running Simulation

A key purpose of ActinViewer is to let the user run robot simulations, either dynamic or kinematic. To start a simulation, the user simply presses the play  button. The simulation will keep running. At this point, the user can select an active end-effector and control it using the mouse. This process is detailed in Section 3.5. The pause  button allows the user to pause the simulation. To resume the simulation, simply press the play  button again. To stop and reset the simulation, press the stop  button. This will bring the simulation back to the initial state.

## 3.5 Controlling End-Effectors

Using ActinViewer, it is possible to use the mouse to command one end-effector at a time to go to an arbitrary position and orientation in space as long as it's within the arm's workspace. The Guide Frame is what ActinViewer uses to specify the desired position of an end-effector. The figure below shows a Guide Frame (with red, green, and blue axes, which correspond to the X, Y, and Z axes, respectively).

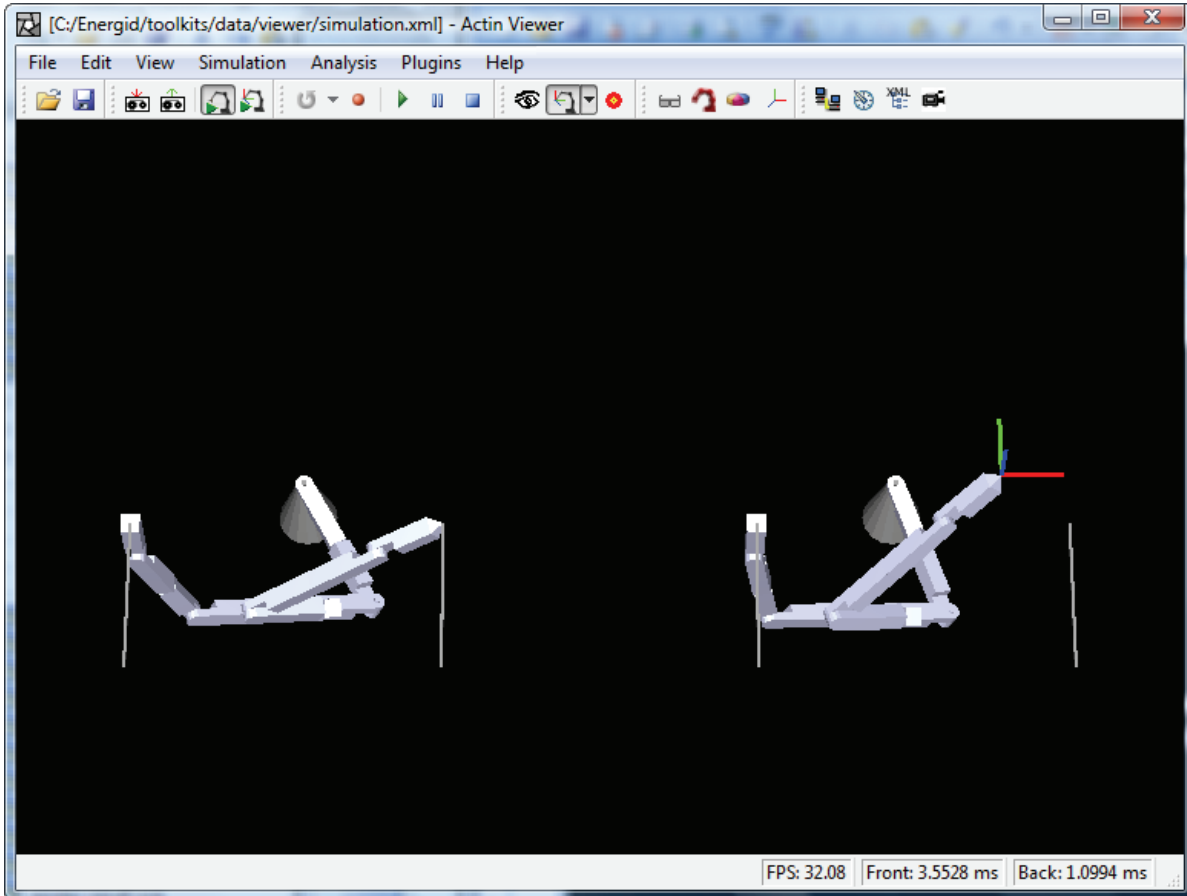

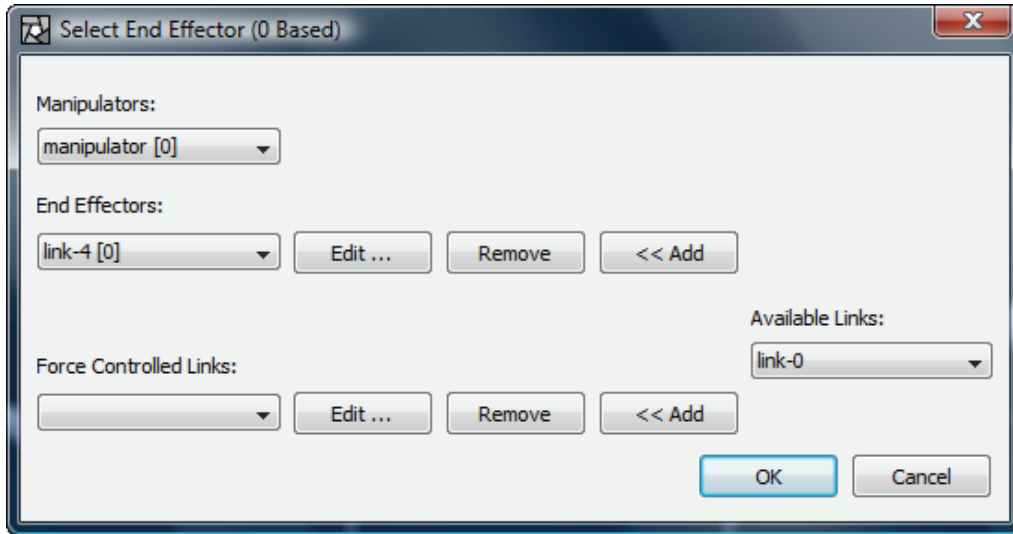


Figure 3-9: The Guide Frame is shown with the red, green, blue axes.



### 3.5.1 Select Active End Effector


Before you can move the Guide Frame, you need to first enter the Guide mode by pressing . The first time you enter the Guide mode, the Select End Effector dialog will show up. In this dialog, you can select which end-effector to be active (i.e. which one you want to follow the Guide Frame) by choosing from the End Effectors drop-down box. In addition, you can edit or remove any existing end-effector or add a new one.



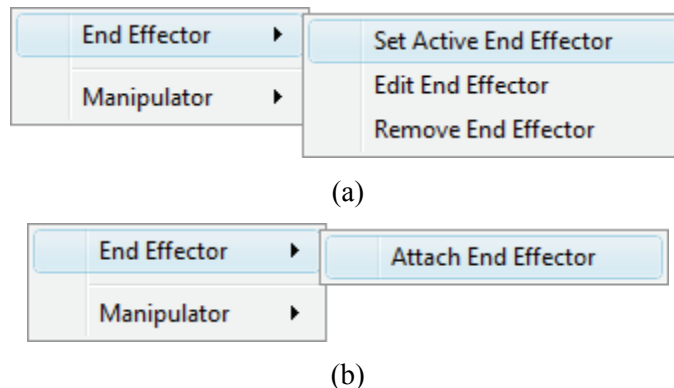
**Figure 3-10:** Select End Effector dialog.

Once OK is selected and if the simulation is running (see Section 3.4 on how to run simulations), you should be able to move the Guide Frame within ActinViewer by dragging with either the left or right mouse buttons depressed. Holding down the left mouse button will allow you to change the position of the Guide Frame and the right mouse button allows you to rotate the Guide Frame. Using one of the following combinations will move the Guide Frame in and out of ActinViewer – scroll wheel, middle mouse button while dragging up/down, or left+right mouse buttons while dragging up/down.

If ActinViewer is already in the Guide mode (the Guide mode button is already depressed like this ) or if an active end-effector has previously been selected, you can get to the Select End Effector dialog by pressing the down-arrow button on the right of the Guide mode  button.

This will drop down the  Set guide frames menu item. Select it to bring up the Select End Effector dialog.


Another, perhaps more convenient, way to select the active end-effector is to use the mouse picking process. Move the mouse so the cursor hovers above the link whose end-effector you would like to control. Then, right-clicking on it will bring up one of the two context menus shown in Figure 3-11, depending on whether there is an existing end-effector attached to the selected link.



**Figure 3-11:** End Effector context menu for (a) link with existing end-effector or (b) link without end-effector.

If there is already an end-effector attached to the selected link, the context menu in Figure 3-11(a) will show up and let you either set it as the active one, edit it, or remove it. If there is no end-effector for that link, then you have an option to add one to that link.

### 3.5.2 Edit and Add End Effectors

By pressing the edit  button in the Select End Effector dialog (Figure 3-10) or selecting Edit End Effector from the context menu in Figure 3-11(a), you can edit the properties of the selected end-effector. This will bring up the Edit End Effector dialog shown in Figure 3-12. Depending on the type of end-effector selected, the end-effector will either move to a specific position in space (with a point end-effector) or position and orientation (with a frame end-effector). The type of end-effector can be selected with the drop-down list below under the Edit End-Effector.

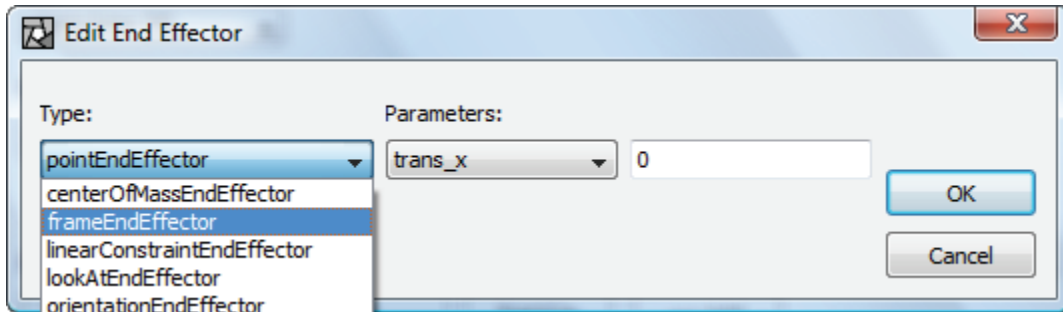
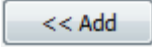
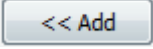
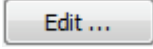
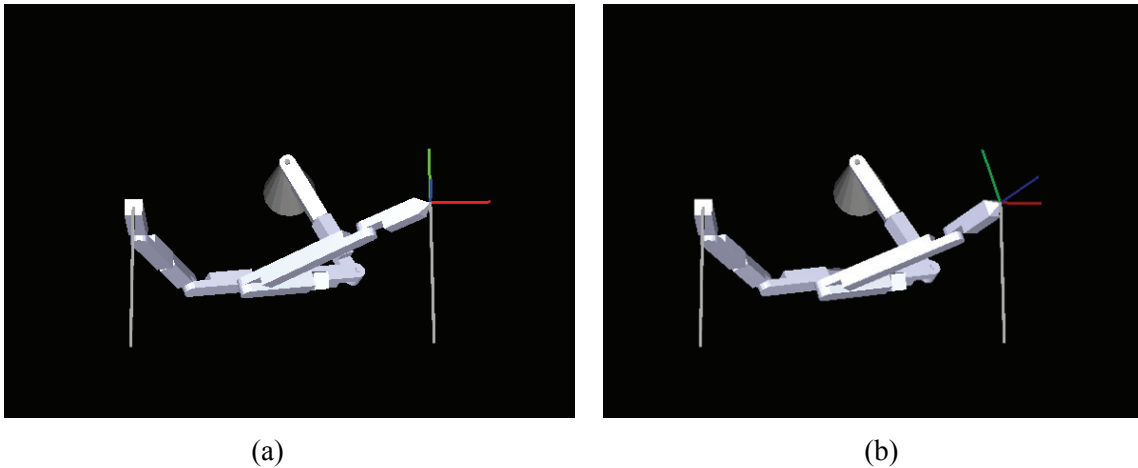


Figure 3-12: Edit End Effector dialog.

There are two ways to add an end-effector to a link. The first method is to use the add  button in the Select End Effector dialog (Figure 3-10). You will need to select which link to add an end-effector by selecting the desired link from the Available Links drop-down box on the right.

Pressing the  button will then add that link to the End Effectors drop-down box on the left. You can then select the newly added end-effector and press the  button to edit the end-effector's properties to the desired values. Alternatively, you can select Attach End Effector from the context menu in Figure 3-11(b). Simply move the mouse to hover above the desired link and right-click the mouse to bring up the context menu. Selecting the Attach End Effector will bring up the Add End Effector dialog, which looks just like the Edit End Effector dialog in Figure 3-12. Change the end-effector's properties as desired and press the OK button to add the end-effector. Or press the Cancel button to if you change your mind and cancel the add process.

The difference between a frame end-effector and a point end-effector can be seen in the images below. Note that with a point end-effector the orientation of the end-effector is arbitrary—only the position of the Guide Frame is important). With a frame end-effector (shown on the right) both position and orientation are considered—note that the last link is aligned along the blue axis of the Guide Frame.







**Figure 3-13:** Point and frame end-effectors. (a) The arm moved to the guide frame using a point end-effector (i.e. position only). (b) The arm moved to the guide frame using a frame end-effector (i.e. frame end-effector).





## 3.6 Working with Path Files


ActinViewer allows you to capture paths of the robot for future playback. This is very useful for certain applications.

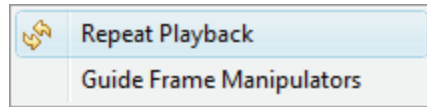
### 3.6.1 Record a Path

Pressing the record  button puts ActinViewer into the record mode. When in this mode, the record button will remain depressed like this  and the robot positions will be stored in memory until the record button is pressed again. Positions can be stored in one of two formats: Manipulator (Joint) mode, or Guide Frame mode. In Manipulator mode the viewer records all of the joint angles for the robot at each timestep, whereas in Guide Frame mode only the commanded gripper positions at each timestep are recorded. This means that a Guide Frame mode path file may result in different joint positions when rerun depending on the control method being used for the manipulators. For instance, a control system configured to minimize kinetic energy will result in different joint angle trajectories than a control system configured to minimize potential energy. A path file recorded in Manipulator mode, by contrast, is guaranteed to always give the same joint trajectories. By default ActinViewer records in Manipulator mode. You can enter Guide Frame mode by pressing  and can revert back to Manipulator Mode by pressing .

### 3.6.2 Playback Mode

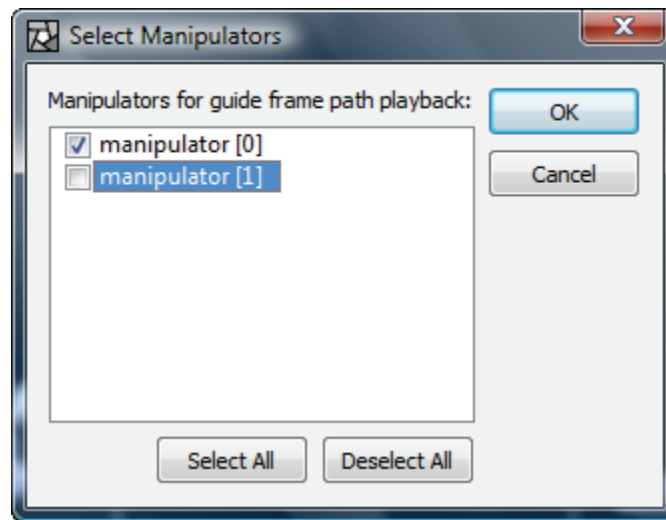
Once a path is recorded or loaded from a path file (see the next section on how to save and load a path file), the playback mode  button will be enabled . It is necessary to specify that you would like to playback the path and not just run a simulation. Press the playback mode button to enter the playback mode. The playback mode button will remain depressed . While in the playback mode, hitting the play  button will replay the path. Opening the dropdown list for the Playback mode button will bring up the options shown in Figure 3-14. The Repeat Playback option allows you to select whether or not the playback should be repeated. Repeat Playback means that at

the end of the path file, ActinViewer will go back to the beginning of the file and repeat the path. This repeat process will run indefinitely until the stop  button is hit.




**Figure 3-14:** Playback mode options.


If the recorded path is in the Guide Frame mode, it is possible to only allow certain manipulators to follow the path while letting other manipulators be controlled by the user using the mouse. For example, you may want a robot to follow a predefined end-effector path but also want the robot reactively avoid an obstacle whose motion you can control using the mouse. To do this, select the Guide Frame Manipulators option in Figure 3-14 to bring up the following dialog. Assuming that manipulator [0] is the robot and manipulator [1] is an obstacle, then selecting manipulator [0] and deselect manipulator [1] as depicted in Figure 3-15 will force the robot to follow the path while let the obstacle be controlled by the user using the mouse.



**Figure 3-15:** Select manipulators for Guide Frame path playback.


### 3.6.3 Save and Load Path File

The save path  button allows you to save a path just recorded. A Save File dialog box will appear asking for the name and location of the file to be saved.

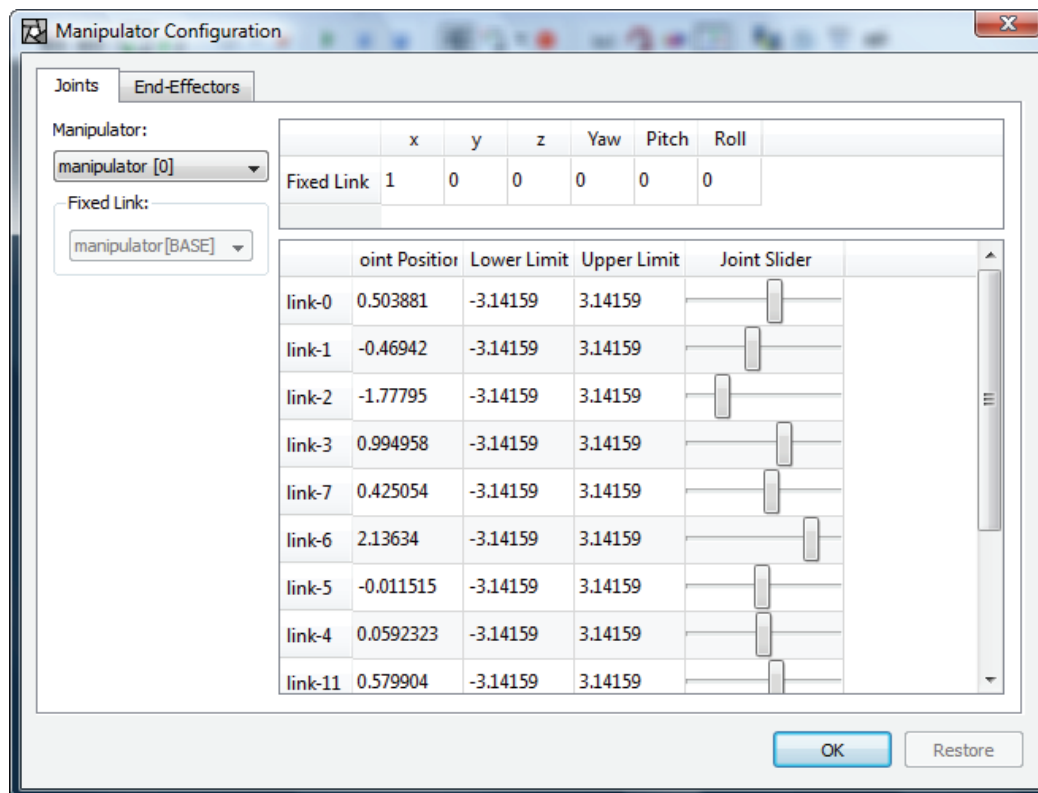
The load path  button allows you to load a previously saved path file. Once loaded, the playback/record mode buttons should automatically change to indicate whether the path is in Guide Frame mode or Manipulator mode. To playback the loaded path, just follow the instruction in Section 3.6.2 Playback Mode.

## 3.7 Manipulator Configuration

Oftentimes, the user may wish not only to know the numerical values of joint positions of a manipulator but also to move some joints to some specific values. He also may wish to do the same things with the end-effector placements. These tasks can be accomplished via the manipulator

configuration dialog. To invoke the dialog, click on the configuration button  in the toolbar. This dialog has two tabs: Joints and End-Effectors.

### 3.7.1 Changing Joint Positions



**Figure 3-16:** The "Joints" tab of the manipulator configuration dialog.

In “Joints” tab shown in Figure 3-16, the dialog displays the joint positions of a manipulator in a tabular format along with the lower and upper limit of each joint. The user can select to view the joint positions of another manipulator from the drop-down box on the left. Also displayed is the position of the “fixed link”. For manipulators with a fixed base, the fixed link is always the base link. Note that the drop-down list for “Fixed Link” is greyed out because this particular manipulator has a fixed base. Section 3.7.3 provides the details on the fixed link. The user can change the joint values in one of two ways. The first method is to edit a value directly into the cell. In the figure above, the value of “link-0” is being edited. A change that the user makes to a joint value will be validated against those limits before the change can take effect. Alternatively, the user can simply slide the slider in the rightmost column. Sliding to the left will decrease the joint value while sliding to the right does the opposite. Not only can the joint values be changed, the values of the base position and joint limits can be edited as well.



### 3.7.2 Changing End-Effector Positions

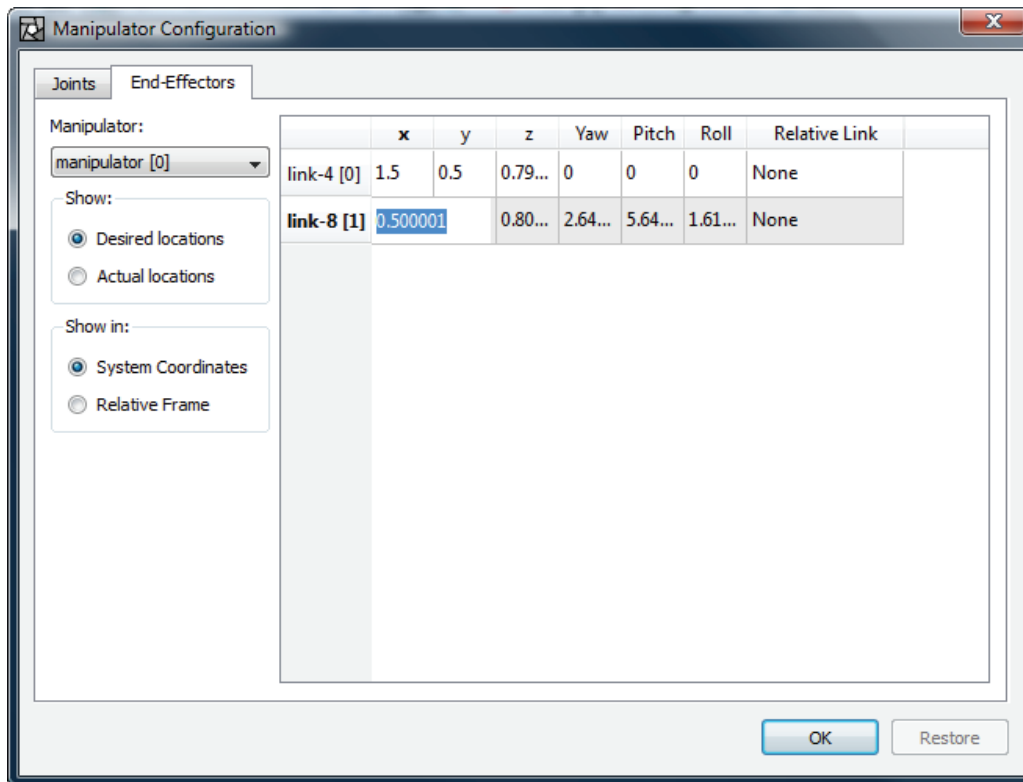


Figure 3-17: The "End-Effectors" tab of the manipulator configuration dialog.

The “End-Effectors” tab shown in Figure 3-17 is similar to the “Joints” tab. It displays all the positions (placements) of all end-effectors belonging to a manipulator. Again, the values of these placements can be directly edited in the cell. In the picture above, the X position of the end-effector attached to “link-8” is being edited. The rightmost column shows the links to which the end-effectors are relatively defined. If an end-effector is not relative to any link, then “None” will be shown. Although the two quantities are often equal, the desired and actual locations of an end-effector can sometimes be different. The user can select which of the two he wishes to examine by selecting the appropriate radio button on the left. Note, however, that if the actual positions are selected, he will NOT be able to edit the values in the table. The user can also choose to view the end-effector positions in either the system coordinate frame or in the the relative link’s frame.

### 3.7.3 Changing the Fixed Link

For manipulators with a floating base such as humanoid robots or the hexapod shown below, it is sometimes more desirable to change the joint positions with respect to a link other than the base link. This link is referred to as the “fixed link.” Figure 3-19 shows the “Joints” tab with a floating-base manipulator, allowing the fixed link drop-down list to be enabled. This provides the user a means to set the fixed link to be any link as desired by selecting from the drop-down list. When a link is selected to be the fixed link, the position (x, y, z, Yaw, Pitch, and Roll) of that link in the system coordinate frame will be shown in the top table on the right. When the value of any joint position is changed either through the slider or direct editing in the cell, the robot will move while the chosen link remains stationary. In this particular case, the link labeled “link\_0\_0\_0\_0”, which is one of the feet, is chosen as the fixed link in Figure 3-19.

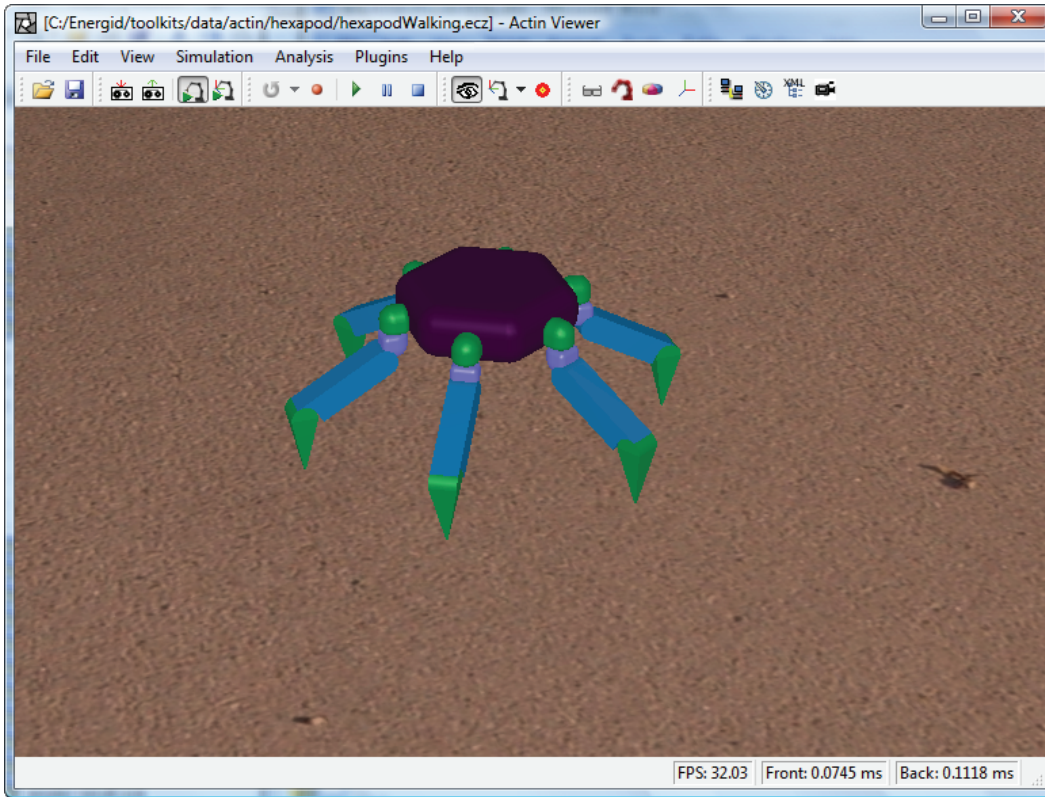


Figure 3-18: The hexapod used to illustrate the "fixed link" concept.

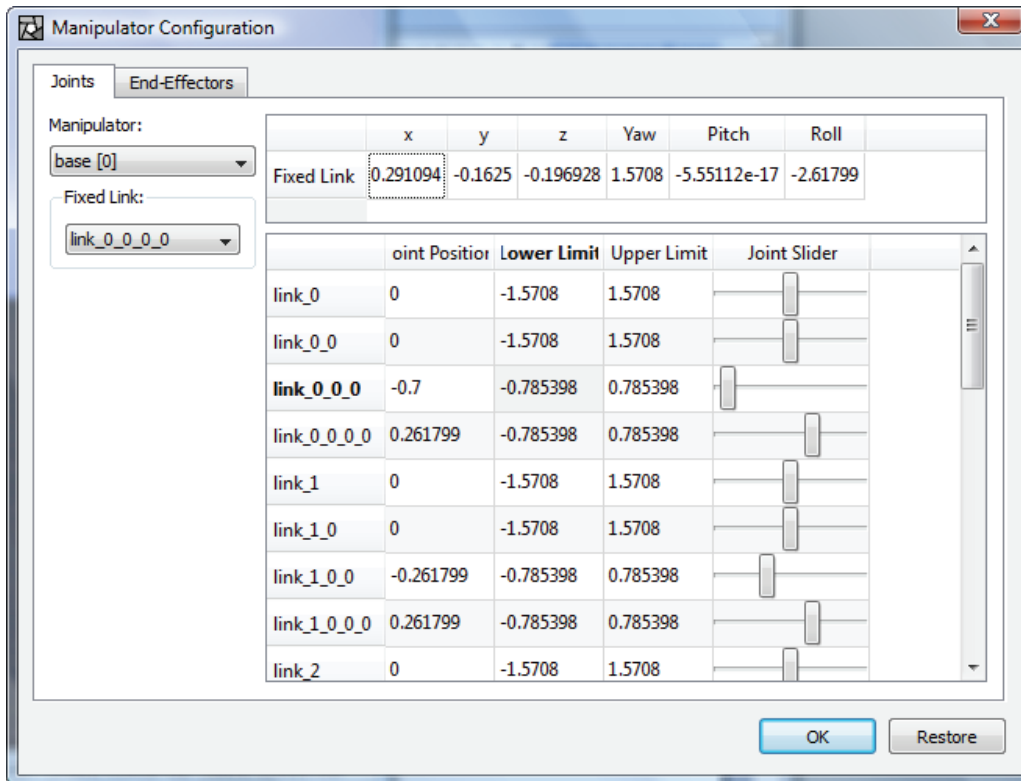
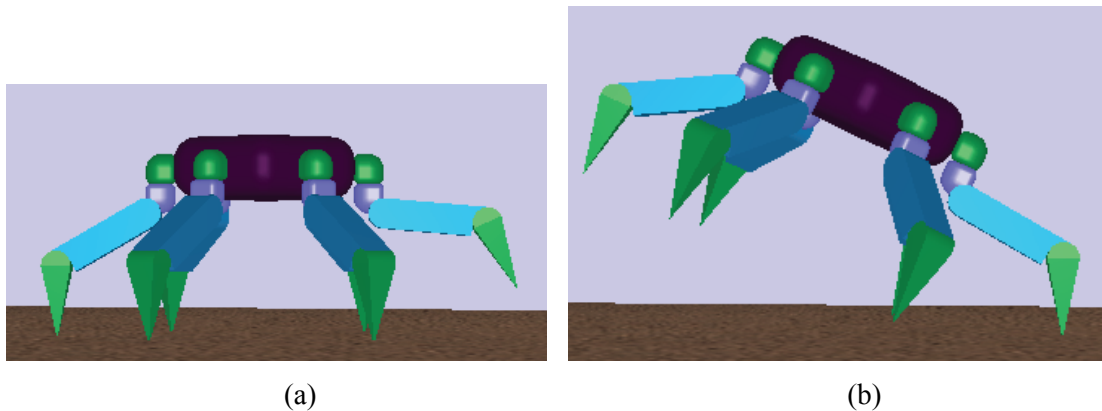


Figure 3-19: The "Joints" tab of the manipulator configuration dialog with a floating-base manipulator and the "fixed link" is set to a link other than the base link.

The fixed link concept is useful in some circumstances. For example, one may wish to visualize what the robot would look like if a joint position is changed while one of the feet is kept on the ground. To see the effects of the fixed link, consider the two configurations of the hexapod shown in Figure 3-20. The configuration on the left is a result of the normal case in which the base link is the fixed link. When the value on one of the joints in the right leg changes, the right leg moves up. In the configuration on the right, the right foot link is chosen to be fixed base. When the same joint changes its position, the right foot remains fixed to the ground while the whole robot moves up instead.



**Figure 3-20:** Two hexapod configurations with the exact same set of joint positions when (a) the base is the fixed link (b) the foot link on the right is the fixed link.

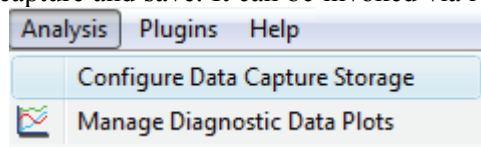
### 3.8 Data Capture

Data capture is a mechanism through which the user can select to store or display any data relevant to a simulation. The detailed discussion of the data capture mechanism is given elsewhere. This section is meant to provide instructions of how to conveniently set up data capture through ActinViewer.

Although information capture can be configured programmatically or through XML, the easiest way may be through interactive GUI. There are actually two separate but somewhat related features. The first one is using GUI to save the captured information in different file formats. The second is using GUI to display captured information in real-time scrolling plots. This section is intended as a guide on how to use these new GUI features using a system with two 12-dof manipulators for demonstration.

#### 3.8.1 Saving Captured Information in Different File Formats

The “Configure Data Capture” dialog shown in Figure 3-22 is where the user configures what information to capture and save. It can be invoked via Analysis->Configure Data Capture Storage in



the main menu.

As can be seen, there are three panels in the dialog.

1. The “System” panel consists of a tree view in which the system is shown and the “Save Options...” button. A system consists of a list of manipulators that in turn consists of a list of links.
2. The “Manipulator Data Capture Types” panel consists of a tree view that shows all the available manipulator-level data types that can be captured and the “Select All” and “Deselect All” buttons. Notice that if a link is selected in the tree view of the System panel, then this panel is disabled since it is not applicable to links.
3. The “Link Data Capture Types” panel consists of a tree view that shows all the available link-level data types that can be captured and the “Select All” and “Deselect All” buttons.

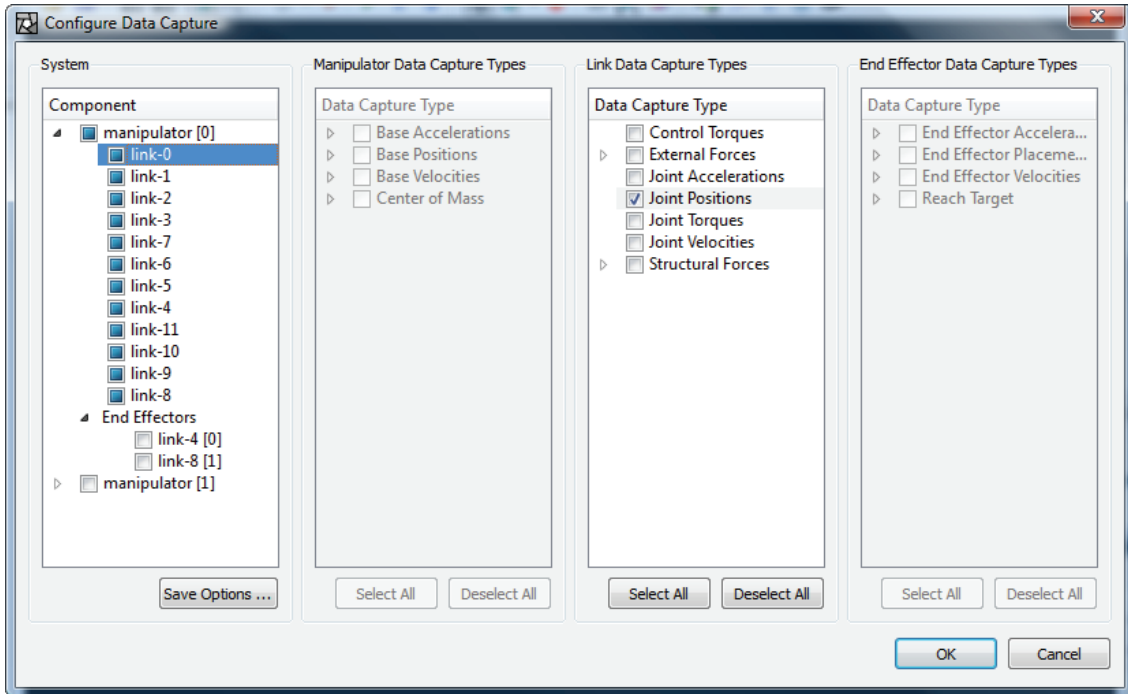
The “Select All” and “Deselect All” buttons are provided for convenience so that the user needs not to tirelessly click on every single checkbox in front of each data capture type. However, by following the tips below on how to select what information to capture will virtually eliminate the need for clicking these buttons.

1. Check/uncheck the checkbox in front of the desired manipulator to select/deselect all manipulator-level data types and all link-level data types of all links in the manipulator.
2. Check/uncheck the checkbox in front of the desired link to select/deselect all link-level data types of that link.
3. Click on the desired manipulator (not the checkbox in front of it) and check/uncheck the checkbox in front of a data capture type to select/deselect the data capture type. If the data capture type is a link-level one, this will select/deselect that data capture type for all the links in the manipulator.

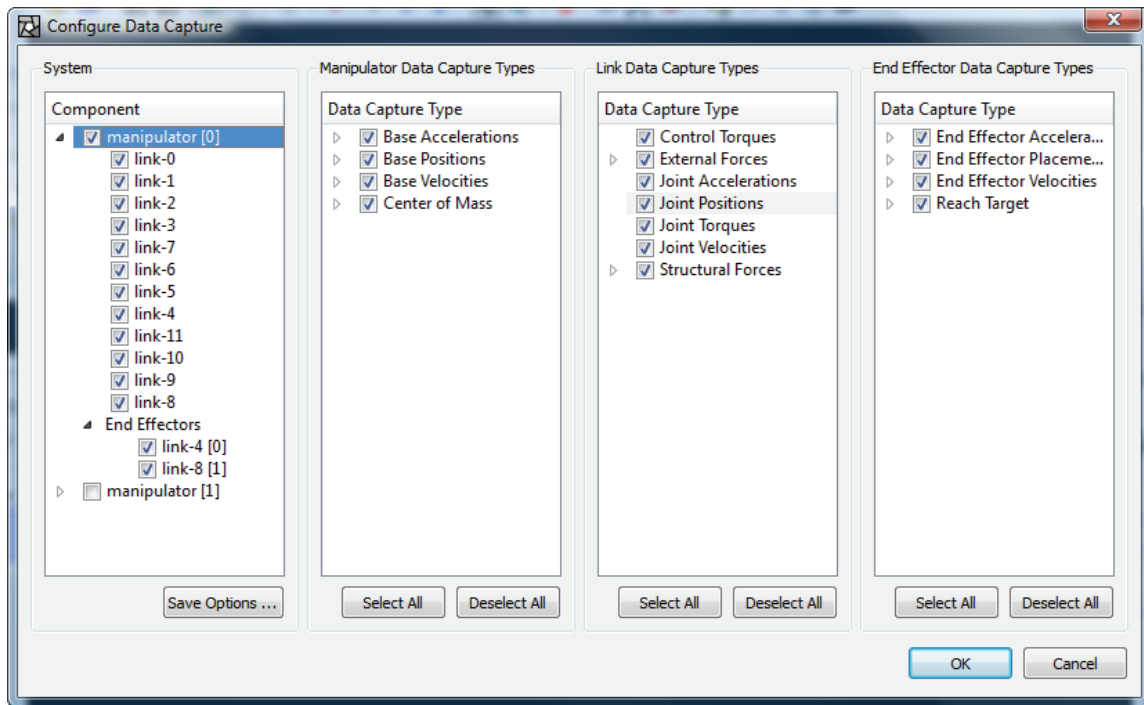
In the dialog in Figure 3-22, notice that there are three states for the checkbox –  unchecked,  partially checked, and  checked. The table below describes the meanings of these check states for all the components.

Check State	Manipulator	Link	Data Capture Type	
			Manipulator is highlighted	Link is highlighted
<input type="checkbox"/> Unchecked	No data type is selected for the manipulator or any of its links.	No data type is selected for the link.	<p>Manipulator-level data type: this data type is not selected for this manipulator.</p> <p>Link-level data type: this data type is not selected in any of its links.</p>	This link-level data type is not selected for this link.
<input type="checkbox"/> Partially Checked	Some manipulator-level data types or some link-level data types of some links are selected.	Some but not all link-level data types are selected for the link.	<p>Manipulator-level data type: this data type is selected for this manipulator but some elements are disabled.</p> <p>Link-level data type: this data type is selected for some but not all of its links.</p>	This link-level data type is selected for this link but some elements are disabled.
<input checked="" type="checkbox"/> Checked	All manipulator-level data types and all link-level data types of all links are selected.	All link-level data types are selected for the link.	<p>Manipulator-level data type: this data type is selected for this manipulator with no element disabled.</p> <p>Link-level data type: this data type is selected for all of its links.</p>	This link-level data type is selected for this link with no element disabled.

**Figure 3-21:** Meanings of different check states for different components.



(a)



(b)

**Figure 3-22:** A GUI dialog for configuring information capture. (a) Checkboxes in front of the first manipulator and its links are *partially* checked since only “Joint Positions” data type is selected. Also, the “Manipulator Data Capture Types” and “End Effector Data Capture Type” panels are disabled since link-0, which does not have manipulator-level or end-effector level data types, is active. (b) Checkboxes in front of the first manipulator and its links are checked since all the data capture types are selected.

Clicking on the “Save Options...” button in the “System” panel will take the user to the dialog depicted in Figure 3-23 that lets him choose how and where to save the simulation information outputs. The dialog shows all the available formats (with their corresponding file extensions) to which the simulation outputs can be saved. If none of the formats is chosen, then no information will be saved. Again, the “Select All” and “Deselect All” buttons are provided for convenience. The user can choose the file name and the location in which the output file(s) will reside. The location can be either directly entered by hand or more conveniently by clicking the “Browse...” button. The file extensions will be added automatically to the file name, depending on the selected formats. For example, in the configuration shown in Figure 3-23, the simulation information will be saved in two files: SimOutputs.m and SimOutputs.xml in the C:/Temp folder. Currently, the supported formats are XML, MATLAB, Mathematica, and comma-delimited text, which can be loaded into Excel.

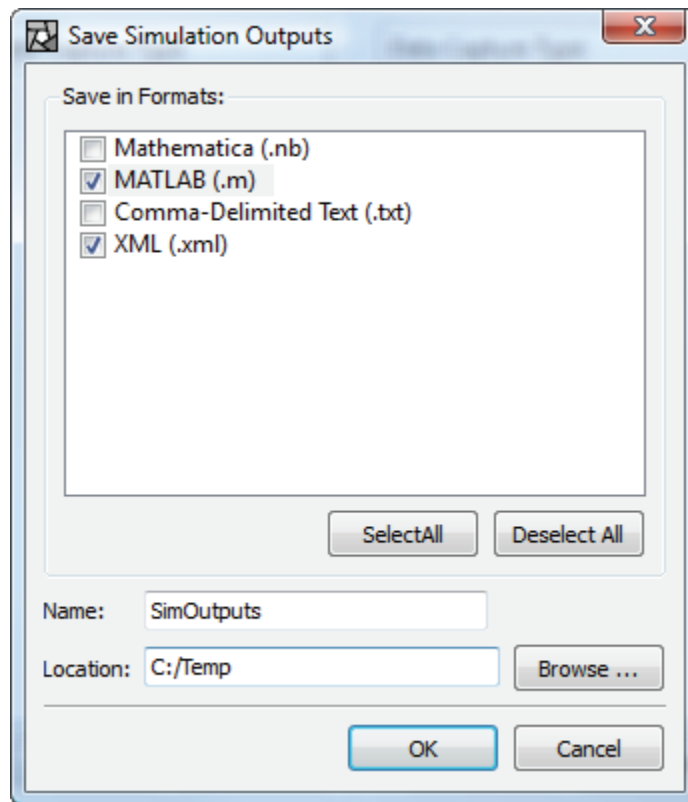
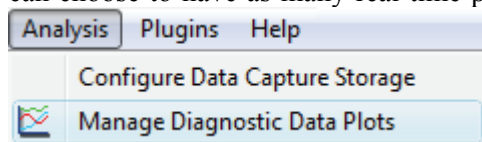


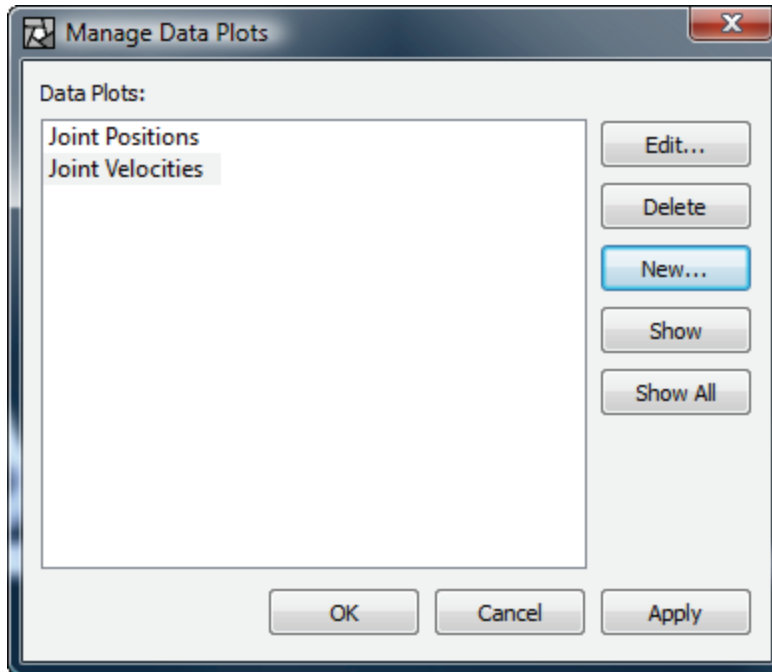
Figure 3-23: A GUI dialog for saving information in different formats.

### 3.8.2 Displaying Captured Information in Interactive Plots

The real-time displaying of captured information was designed to be as flexible as possible. The user can choose to have as many real-time plots as he wishes. Selecting Analysis -> Manage Data Plots



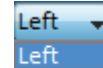
in the main menu brings up the dialog shown in Figure 3-24. Through this dialog, the user can create a new plot, edit or delete an existing plot, and show any plot that has been created.



**Figure 3-24:** A GUI dialog for managing real-time plots of captured information.

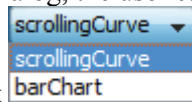
Clicking the “Edit...” or “New...” button will bring up the dialog shown in Figure 3-25, which is used to configure what information to capture and display in a plot. It is very similar to the dialog in Figure 3-22 with the following key differences.

1. This dialog does not have a “Save Options...” button.
2. This dialog has an entry to edit the title of the plot.
3. Through this dialog, the user can choose to have the captured information to be displayed on



either the left and/or the right Y-axis through the Y-Axis drop-down list. The captured data for both Y-axes are totally independent. They can even be the same.

4. Through this dialog, the user can choose the desired type of the plot through the Plot Type

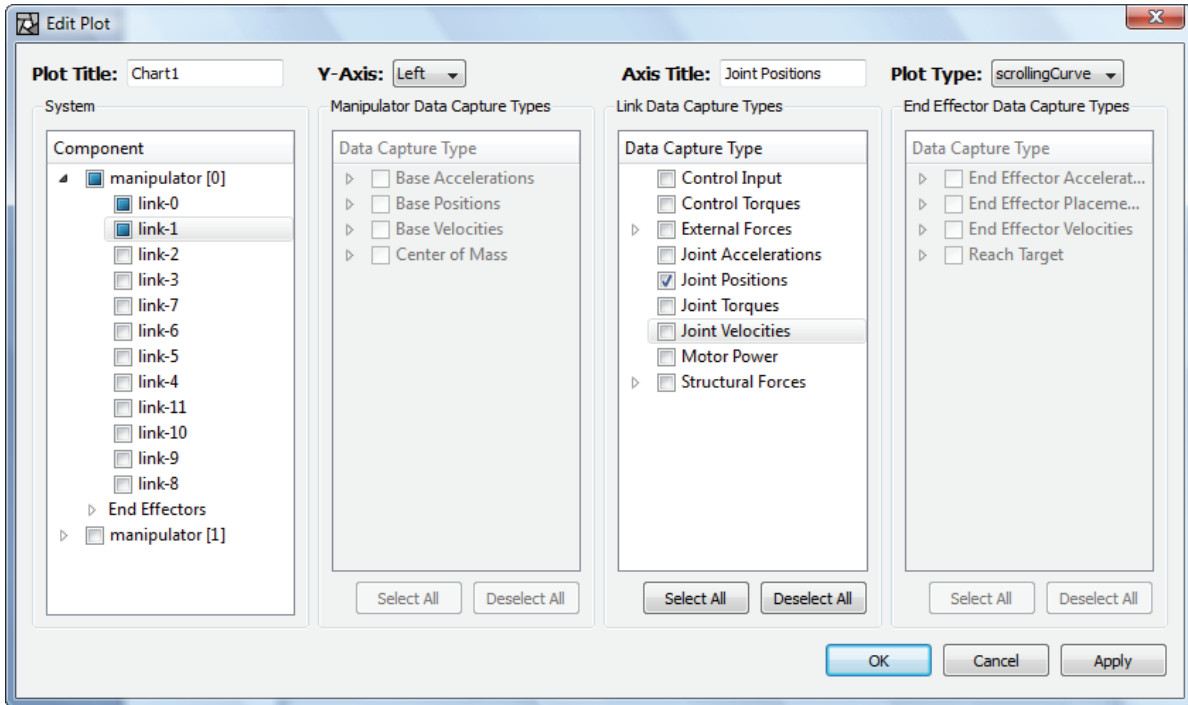


drop-down list. Currently, two plot types are supported– the scrolling plot and the bar chart.

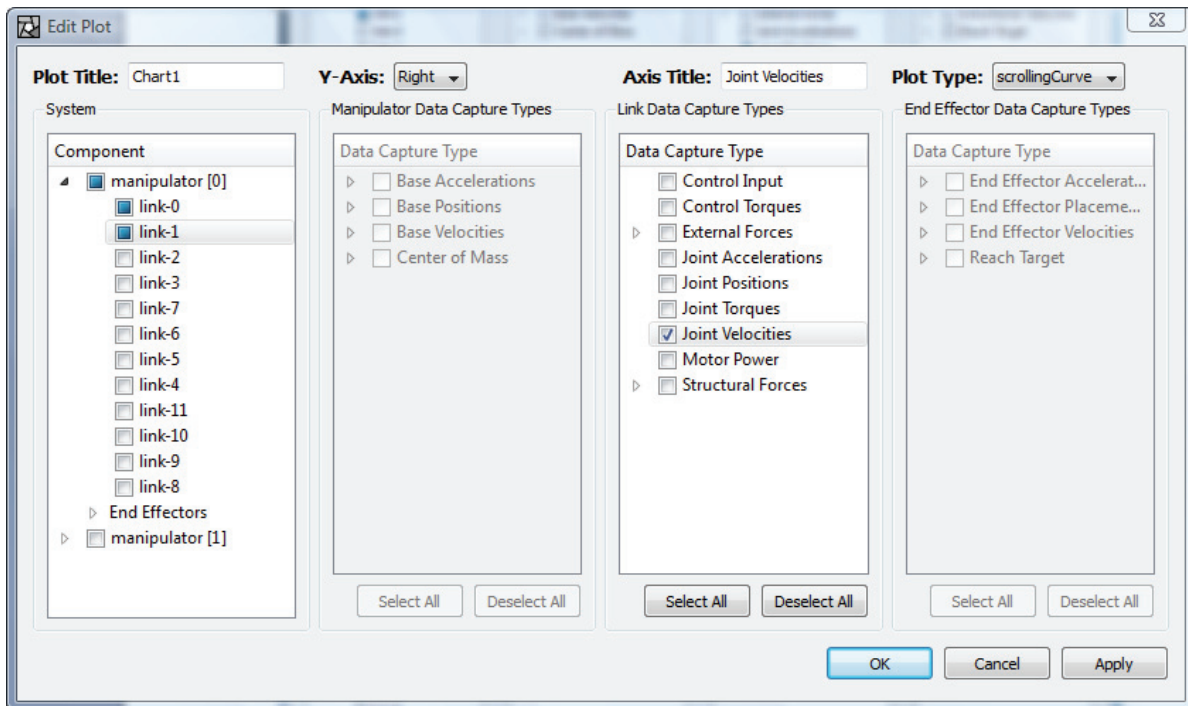
The tips on how to select what information to capture and the meanings of the states of the checkboxes are the same as those for Figure 3-22, which were described in the previous section.

Once the captured information for a plot has been configured, the plot can be created and shown by clicking the “Show” button in the dialog in Figure 3-24. The plot “widgets” were implemented using the Qwt toolkit (<http://qwt.sourceforge.net>), which is a collection of Qt widgets used in technical applications.





(a)

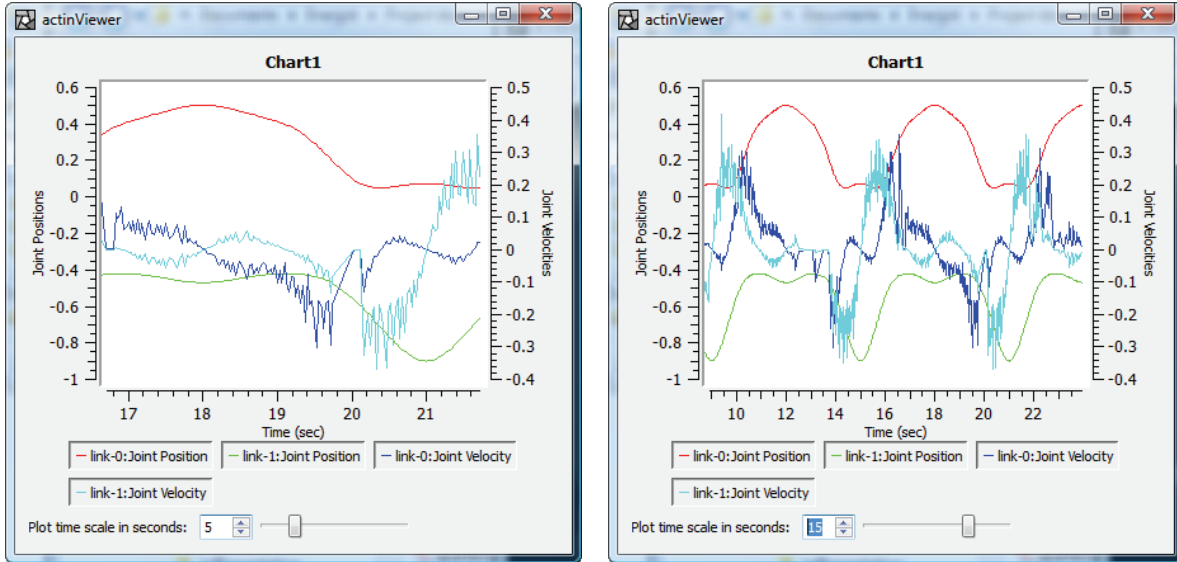


(b)

**Figure 3-25:** A GUI dialog for configuring information capture for a real-time plot (a) for the left Y-axis and (b) for the right Y-Axis. The configurations of the two axes can be done independently.

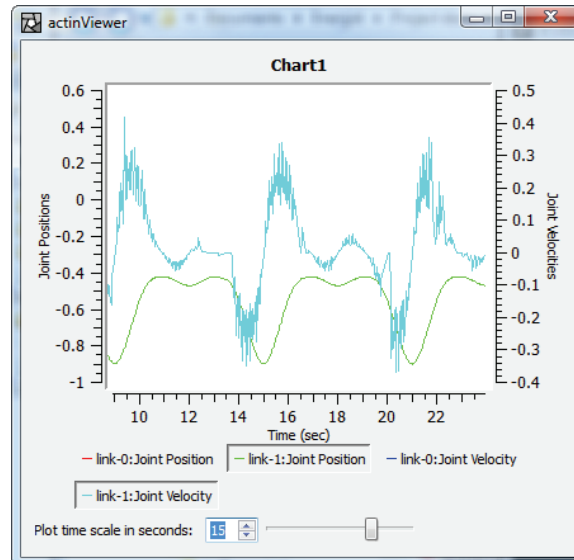
Figure 3-26 shows different views of the same real-time scrolling plot displaying the captured information that was configured by the dialog in Figure 3-25. Note the left and right Y-axes with different scales. Within the scrolling plot, the user can adjust the time scale interactively by using the

spinner or the slider. This will cause the plot to scroll at varying speed. The larger the time scale, the slower the plot scrolls. Figure 3-26 (a) and (b) show the same plot but with different time scales. Conveniently, the user can also choose to show or suppress any curve in the plot by clicking on its legend. In Figure 3-26 (c), the joint position (red curve) and joint velocity (blue curve) of link-0 are suppressed. Furthermore, multiple real-time plots can be displayed simultaneously when the simulation is running as illustrated in Figure 3-27. It should be cautioned here that displaying a real-time scrolling plot takes a fair amount of resources in terms of computational time and memory. Therefore, displaying too many scrolling plots all at once could potentially deteriorate the simulation run-time performance.



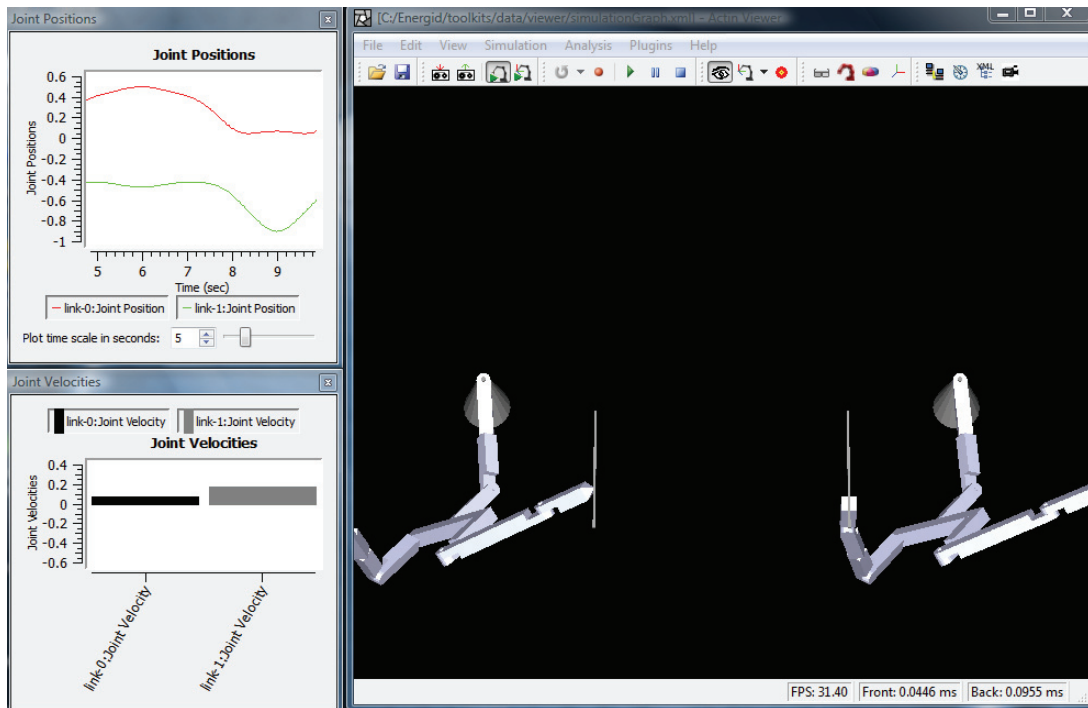
(a)

(b)



(c)

**Figure 3-26:** A real-time scrolling plot displaying captured information (a) with a time scale of 5 seconds; (b) with a time scale of 15 seconds; and (c) with a time scale of 15 seconds and with some curves suppressed.



**Figure 3-27:** Simulation is running with two real-time plots (one scrolling plot and one bar chart).

### 3.9 Time Scaling

The viewer includes configurable time scaling as part of the simulation. Time scaling is controlled through the member variable `m_TimeScaleFactor` of the class `EcViewerParameters`. Setting this parameter to a value less than one in the XML description of a simulation induces simulation execution at that fraction of wall-clock time, and setting it greater than one induces simulation at faster than wall-clock time.

## 4 Guiding Software Principles

### 4.1 This Document

Though Actin comes with a number of applications, it is primarily a software toolkit that can be integrated into any application. This section gives principles used in developing the toolkit.

#### 4.1.1 Overview

This document is intended to guide the reader in understanding the Actin™ code and how to use it to develop C++ software. It assumes that the reader has familiarity with C++ and the language used to describe its constructs. To make it readable, this document is not comprehensive. The many details left out of this document can be found in the class documentation, which is also provided with the toolkit.

#### 4.1.2 Component Representation

Quantities are represented in this document as described below:

- Class names are italicized, for example: *EcVector*.
- Variable names are italicized, for example: *m\_Position*.
- Filenames are quoted, for example: “ecVector.h”.
- Code is represented with a fixed font, for example: `vector.setX(2.0);`

### 4.2 Source Code

In developing this software toolkit, Energid Developers have defined and adhered to a set of guiding principles to make it easier to understand and use the toolkit. These are described below.

#### 4.2.1 Name Space

To prevent conflicts with other code that will be linked to ours by customers, all values with global scope are prefixed with the letters “Ec”. This includes class names, macros, and utility functions.

#### 4.2.2 Classes

With few exceptions, classes include implementation of the “big four” methods: the constructor, the destructor, the copy constructor, and the equal operator. The copy constructor and equal operator are deep (e.g., the contents of a pointer are copied, not the pointer value itself) except where noted in the class documentation and header files. If any of the big four are not implemented, they are protected in the header file.

Where relevant, classes have a *clone* operator. This is also called a virtual constructor—it returns a new’ed copy of itself as a pointer of its appropriate base class. Clone methods make a deep copy. This allows objects to be used as prototypes. Most classes also implement `operator==( )` for testing.

#### 4.2.3 Identifiers

All member functions besides constructors and destructors begin with a lower-case letter and use the camel-hump style, with each word in the name beginning with a capital, for example, *printResults*.

All member variables begin with the prefix “m\_”, followed by a capital letter if the variable represents a member object or basic type. For example, *m\_Range*.

Static member variables begin with the prefix “m\_the”. For example, *m\_theCount*. Class names begin with the prefix “Ec” followed by a capital and use the camel-hump style. For example, *EcPolygonRootFinder*.

Accessors use `const type& variableName()` or `getVariableName(const type& var)`. Mutators use `setVariableName(const type& var)`. (Basic types, like `int` and `double`, are typically passed by value, but objects are passed by reference.)

#### **4.2.4 Protection**

Member variables are always protected—no variables or methods are private. This allows you more flexibility when subclassing. Whenever there is a chance a method might be correctly called within another object, it is declared public, even if it is not used in a public manner in the toolkit code.

#### **4.2.5 Virtualness**

Member functions as a rule are declared virtual. This provides maximum flexibility in subclassing. A few special, basic classes (*EcVector*, *EcOrientation*, and *EcXmlBasicType*, *EcXmlVector*, *EcXmlOrientation*, *EcCoordinateSystemTransformation*, *EcGeneralForce*, and *EcGeneralMotion*) are nonvirtual to improve runtime.

#### **4.2.6 Constness**

All member functions that do not modify member data are declared `const`. It can be appropriate to have both `const` and non-`const` versions of a method, such as when returning `const` and non-`const` pointers or references. Accessors that return member variables return `const` references. Mutators pass `const` references. Static member variables that are not basic types (`int`, `double`, etc.) are `const`.

#### **4.2.7 Pointers**

Pointers are always set to `EcNULL` (which equals 0) when they are not valid. Note it is always safe to delete a null pointer to an object. It is not always safe to delete[] a null array pointer, and these should be checked first. Member pointer variables are prefixed with “m\_p”, for example *m\_pImage*.

#### **4.2.8 Factory Methods**

Objects are created in virtual factory methods. That is, “new” is rarely used outside of methods specifically for creating objects. Generally, factory methods should be prefixed with “new”. For example, *EcImage\* newImage()*. The use of factory methods allows you to subclass an object and replace member variables with subclassed versions.

#### **4.2.9 Multiple Inheritance**

Multiple inheritance is not used in the toolkit.

#### **4.2.10 Units**

All units are SI unless the variable or method name includes the units. For example, *lengthInches* would be the length in inches, while *length* would be the length in meters.

### **4.2.11 Macros**

Macros and macro-like functions are all named starting with “Ec” followed by upper-case letters. For example a macro to warn the user would be EcWARN.

### **4.2.12 Raster Data**

Raster data is ordered consistently using one of two methods.

- Image and image-like raster data are ordered such that image(x,y) has x incrementing left to right and y incrementing top to bottom (i.e., [column,row]). Position (0,0) is always in the upper left-hand corner.
- Matrices and other non-image mathematical formulations are ordered such that matrix(i,j) has i incrementing top to bottom and j incrementing left to right (i.e., [row,column]). As with images, position (0,0) is the upper left-hand corner entry.

### **4.2.13 Filenames**

Classes are defined in .h files and implemented in .cpp files. Each set of .h and .cpp files defines only one class. Filenames should be similar to class names, with “ec” as a prefix. So, “ecJointActuator.h” would be the filename for class *EcJointActuator*, and “ecPolygon.h” would be the filename for *EcPolygon*. In all cases, filenames start with a lower-case letter.

### **4.2.14 Extension Avoidance**

Microsoft-specific extensions are avoided in the toolkit, which builds under Windows and Linux.

### **4.2.15 Exception Handling**

In the toolkit, exception handling is avoided in favor of null pointer return in most cases. This allows general good practice (checking pointers) to overlap with error handling and leads to less cluttered, faster code. There are three areas in which exception handling may be used.

- When it is required by other (third party) software.
- When there is no appropriate return type to flag an error.
- When an error condition requires a lot of information or information that is different from nonexception cases.

### **4.2.16 Friends**

The use of friend classes is avoided in the toolkit. Friendship is not inherited, complicating reuse.

## 5 Fundamental Classes

### 5.1 Data Structures

The data structures that will be used in the description of the Actin™ toolkit include the vector, tensor, list, map and tree. In the description below, *constant time* refers to the ability to perform an operation in a fixed amount of time regardless of the number of elements in the data structure. The term *linear time* refers to the ability to perform a task in a time that is proportional to the number of elements in the data structure. The term *NULL* refers to any symbol representing emptiness—for C++ pointers this is usually the zero value.

#### 5.1.1 Vector

A vector data structure is a container of any data type that allows constant-time access to any element. Inserting or deleting elements requires linear time. It is equivalent to a standard array as used in C++, with the addition that part of this data structure is its size. Characteristically, data in a vector cannot be easily inserted or deleted. Vectors are used widely in the toolkit for storing and accessing data that is fixed in size, such as joint positions, joint rates, vertices, end-effectors, and so forth.

#### 5.1.2 Tensor

A tensor is an array of  $n$ -dimensional data that allows access in constant time for fixed  $n$ . A vector is a substitute for a one-dimensional tensor, but it is given its own category above because of its importance. Another important type of tensor is the array, which is a two-dimensional tensor. Data in a tensor cannot be easily inserted or deleted. Tensors are used widely in the toolkit for multidimensional data, such as images, Jacobians, link-collision maps, and so forth.

#### 5.1.3 List

A list data structure is a linear sequence of elements each of which holds a reference or pointer to its predecessor and its successor. The head of the list has a NULL predecessor and the tail has a NULL successor. All elements can be accessed sequentially in linear time, though linear time is also required to access any particular element. Elements can be inserted and deleted in constant time. Lists are not used widely in the toolkit.

#### 5.1.4 Map

A map data structure represents a one-to-one mapping between data types, often using a hash table. For example, a map might provide integers (the values) as a function of strings (the keys). A map represents this data in a way that allows constant- or log-time (depending on the implementation) access to a value given its key. Maps are used widely in the toolkit for organization. After loading, for each manipulator, links are mapped with string labels as keys.

#### 5.1.5 Set

A set data structure represents a collection of unique types. If a type is added to a set more than once, only one resides in the set. A set represents this data in a way that allows constant- or log-time (depending on the implementation) access to a given value in the set. Sets are used sparingly in the toolkit for organization.

### 5.1.6 Tree

A tree data structure is a collection of elements into a hierarchy such that each element has a unique parent but may have multiple children. The root of the tree has a NULL parent. Accessing any particular element in the tree may require log or linear time, depending on organization. Searches and operations can proceed in depth-first or breadth-first ordering. Trees are used widely in the toolkit to implement control systems that can be configured at run time.

## 5.2 Basic Types

The table below lists the basic data types used in the toolkit and their description. More complex classes are created by composing these. Note that all class names in this document are prefixed with “Ec”. This parallels their implementation in code to avoid namespace conflicts with other source code.

Class	Description
<i>EcReal</i>	64-bit floating-point value.
<i>EcU32</i>	32-bit unsigned integer.
<i>EcInt32</i>	32-bit signed integer.
<i>EcU16</i>	16-bit unsigned integer.
<i>EcInt16</i>	16-bit signed integer.
<i>EcU8</i>	8-bit unsigned integer.
<i>EcInt8</i>	8-bit signed integer.
<i>EcAngle</i>	A floating-point value in the range $(-\pi, \pi]$ . The sine and cosine of the angle are stored in the class for fast trigonometric processing.
<i>EcNonNegReal</i>	Non-negative 64-bit floating point value.
<i>EcString</i>	A variable-length array of Unicode characters.
<i>EcBoolean</i>	A Boolean value.

**Table 5-1:** Basic data types.

## 5.3 Basic Kinematics

For the most part, class details will not be described in this document, but rather in the class documentation. However, because of their importance throughout the toolkit, position, orientation, coordinate system transformations, and rigid-body motion will be described in detail.



### 5.3.1 Position

Position is defined using the *EcVector* class. This class holds three real values representing x, y, and z coordinates. It also provides a number of methods for working with vectors. Given three *EcVectors*, *a*, *b*, and *c*, the following table shows some of the more useful methods.

Method Examples	Description
<code>EcVector a(x, y, z);</code>	This constructor creates a vector holding the three specified values.
<code>a+=b; a-=b; a=b+c; a=b-c;</code>	Various math operators are defined among vectors.
<code>a*=r; a=r*b; a=b*r; a=b/r;</code>	Various math operations with scalars are also defined.
<code>a==b;</code>	Equality is defined, which returns <code>EcTrue</code> if both vectors are equal.
<code>a.approxEq(b, tol);</code>	Fuzzy equality is defined, which returns <code>EcTrue</code> if all the elements are within the tolerance of each other.
<code>r=a.dot(b); a=b.cross(c);</code>	Dot and cross product are defined.
<code>r=a.mag(); r=a.magSquared(); r=a.distanceTo(b); r=distanceSquaredTo(b);</code>	Various Euclidean norms are defined. The squared values are faster to calculate because they do not require a square root.
<code>a=b.unitVector(); a.normalize();</code>	Methods are included for normalizing vectors.
<code>r=a.x(); r=a.y(); r=a.z();</code>	Accessors for the three elements.
<code>a.setX(r); a.setY(r); a.setZ(r);</code>	Mutators for the three elements.
<code>r=a[0]; r=a[1]; r=a[2];</code>	Indexing can be used to get nonconst references to the three elements. These can be used for accessing the elements or setting them.
<code>a=EcVector::zeroVector();</code>	This method returns a const reference to a static member variable. It can be used for error recovery from a method that returns a reference to a vector.

**Table 5-2:** Select methods in the *EcVector* class. For a complete description, please see the class documentation.

### 5.3.2 Orientation

Orientation is defined using the *EcOrientation* class. Each member of this class holds four values, {w, x, y, and z} that define a quaternion that represents the orientation of an outboard frame, *j*, with respect to an inboard frame, *i*. This quaternion translates into a direction cosine matrix through the following formula:

$$R = \begin{bmatrix} 1 - 2y^2 - 2z^2 & 2xy - 2wz & 2xz + 2wy \\ 2xy + 2wz & 1 - 2x^2 - 2z^2 & 2yz - 2wx \\ 2xz - 2wy & 2yz + 2wx & 1 - 2x^2 - 2y^2 \end{bmatrix} \quad (4-1)$$

Note that, with this formalism, Q={1,0,0,0} corresponds to the identity matrix and the negative of a quaternion gives the same rotation as the original. (This formalism can be easily converted to the other common form by ordering the entries {x, y, z, and w}.)

The *EcOrientation* class provides a number of methods for extracting information about orientation, transforming orientations, and working with vectors. Its methods are defined in the table below. Given three *EcOrientations*, a, b, and c, the following table shows some of the more useful methods that can be used.

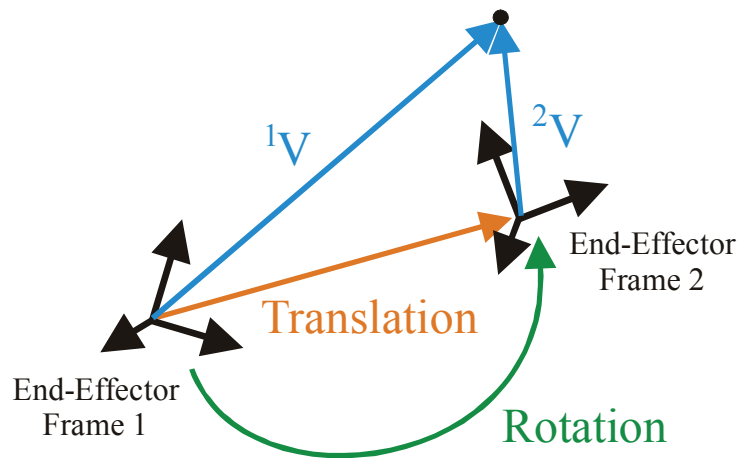
Method Examples	Description
<code>EcOrientation a(w, x, y, z);</code>	This constructor creates an orientation with the specified quaternion values. The four values are normalized to a unit vector.
<code>a.set(w, x, y, z);</code>	This assigns the specified quaternion values to a pre-existing orientation. The four values are normalized to a unit vector.
<code>a*=b; a=b*c;</code>	Various operators are defined among orientations. Orientation multiplication is not commutative.
<code>w=a*v;</code>	Transformation of vectors is defined.
<code>a==b;</code>	Equality is defined, which returns <code>EcTrue</code> if both quaternions are equal.
<code>a.approxEq(b, tol);</code>	Fuzzy equality is defined, which returns <code>EcTrue</code> if all the quaternion elements are within the tolerance of each other.
<code>a.invert(); a=b.inverse();</code>	Inversion routines are defined.
<code>a.setFrom321Euler(psi, th, phi); a.get321Euler(psi, th, phi); a.setFromAngleAxis(angle, axis); a.getAngleAxis(angle, axis); a.setFromDcmRows(r0, r1, r2); a.getDcmRows(r0, r1, r2);</code>	Various methods are available for setting the orientation from other representations and calculating the equivalent in other representations. More conversions are available than those shown here—please see the class documentation for more.

<pre>a=EcOrientation::identity();</pre>	<p>This method returns a const reference to a static member variable representing no orientation change. It can be used for error recovery from a method that returns a reference to a vector.</p>
---	--

**Table 5-3:** Select methods in the *EcOrientation* class. For a complete description, please see the class documentation.

### 5.3.3 Coordinate System Transformation

A coordinate system transformation includes both translation and reorientation. Our implementation class is *EcCoordinateSystemTransformation*, which holds one *EcVector* object and one *EcOrientation* object. These represent the position and orientation of a new frame (the outboard frame) expressed in a reference frame (the inboard frame). There are a number of ways to interpret the meaning of a coordinate system transformation. Our primary interpretation in this document is the following: It is a device for transforming quantities represented in the outboard frame to be represented in the inboard frame.



**Figure 5-1:** A coordinate system transformation is represented using the *EcCoordinateSystemTransformation* class, which includes an *EcVector* translation and an *EcOrientation* orientation. The *EcCoordinateSystemTransformation* can be viewed as representing frame 2 in frame 1 as shown above. The translation is represented in frame 1 coordinates. A vector represented in frame 2 can be multiplied by a coordinate system transformation to give the same point in frame 1 coordinates. That is, if  $v_2$  was the variable representing  ${}^2v$  as shown above, and  $c$  was the variable representing the coordinate system transformation giving frame 2 in frame 1, then “ $v_1=c*v_2;$ ” would calculate  ${}^1v$  as shown above.

To increase speed, simplified states of an *EcCoordinateSystemTransformation* are tracked and used when transforming quantities. Each *EcCoordinateSystemTransformation* has a mode, with four options as shown in the table below.

<b>EcCoordinateSystemTransformation Mode</b>	<b>Description</b>
ARBITRARY	The translation and orientation may take on any

	values.
NO_TRANSLATION	There is no change in translation. The translation vector equals {0,0,0}.
NO_ROTATION	There is no change in orientation. The orientation quaternion equals {1,0,0,0}.
NO_CHANGE	There is no change in position or orientation.

**Table 5-4:** The *EcCoordinateSystemTransformation* mode. This value is used to optimize calculations performed with the object.

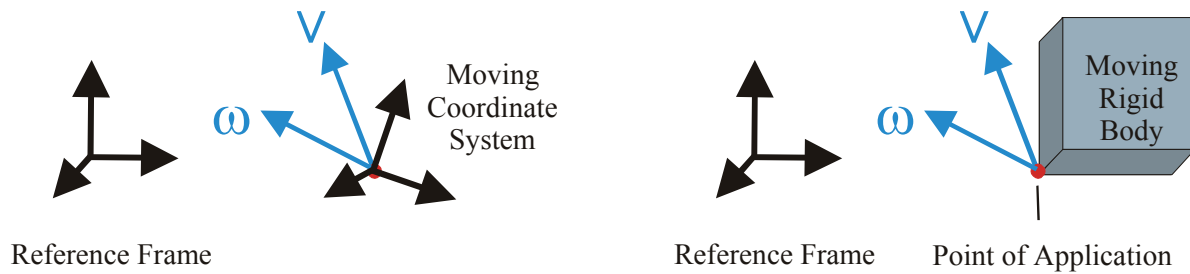
The most common methods provided by the *EcCoordinateSystemTransformation* class are shown in the table below, where a, b, and c represent *EcCoordinateSystemTransformation* objects.

Method Examples	Description
<code>EcCoordinateSystemTransformation (translation, orientation);</code>	This constructor creates a coordinate system transformation with the specified position and orientation.
<code>v=a.translation(); a.setTranslation(v); q=a.orientation(); a.setOrientation(q);</code>	Methods to get and set the position (an <i>EcVector</i> ) and the orientation (an <i>EcOrientation</i> ).
<code>a.mode();</code>	Access to the mode (as described through Table 5-3).
<code>a*=b; c=a*b;</code>	Transformation combination. If a represents frame A in reference coordinates and b represents frame B in frame A coordinates, then c=a*b represents frame B in reference coordinates.
<code>a==b;</code>	Equality is defined, which returns <i>EcTrue</i> if both transformations are equal.
<code>a.approxEq(b, tol);</code>	Fuzzy equality is defined, which returns <i>EcTrue</i> if all the position and orientation elements vary by less than the tolerance.
<code>a.invert(); a=b.inverse();</code>	Inversion routines are defined.
<code>a=EcOrientation::identity();</code>	This method returns a const reference to a static member variable representing no translation or orientation change. It can be used for error recovery from a method that returns a reference to a vector.

**Table 5-5:** Select methods in the *EcCoordinateSystemTransformation* class. For a complete description, please see the class documentation.

### 5.3.4 Rigid-Body Velocity

The motion of rigid bodies and reference frames is described in the toolkit using the *EcGeneralMotion* class. *EcGeneralMotion* is typedef'ed to *EcGeneralVelocity* to represent frame velocity. Each *EcGeneralVelocity* object has one *EcVector* representing linear velocity of a point and one *EcVector* representing angular velocity about that point. Each *EcGeneralVelocity* object must have a point of application and a frame of representation—these are implicit. This is illustrated in the figure below.



**Figure 5-2:** Reference-frame and rigid-body velocity are represented using *EcGeneralVelocity* objects, which comprise vectors of linear and angular velocities ( $\vec{v}$  and  $\vec{\omega}$ , respectively). These quantities are defined in an implicit reference frame at the origin of the moving frame or at an implicit point on a moving rigid body. *EcGeneralAcceleration* has a similar form.

Method Examples	Description
<code>EcGeneralVelocity(linear, angular);</code>	This constructor creates a general velocity from a linear and an angular component.
<code>a.linear(); a.setLinear(v);</code> <code>a.angular(); a.setAngular(w);</code>	Methods to get and set the linear and angular components (both <i>EcVectors</i> ).
<code>a+=b; a-=b; a=b+c; a=b-c;</code>	Various math operators are defined among <i>EcGeneralVelocity</i> objects.
<code>a*=r; a=r*b; a=b*r;</code>	Various math operations with scalars are also defined.
<code>a==b;</code>	Equality is defined, which returns <i>EcTrue</i> if both objects are equal.
<code>a.approxEq(b, tol);</code>	Fuzzy equality is defined, which returns <i>EcTrue</i> if

	all the position and orientation elements vary by less than the tolerance.
<pre>a.transformBy(xform); a.transformBy(orient); a.transformBy(vector);</pre>	Routines to transform the <i>EcGeneralVelocity</i> object by moving the point of application, changing the reference frame, or both.

**Table 5-6:** Select methods in the *EcGeneralMotion* class. *EcGeneralVelocity* shares this interface. For a complete description, please see the class documentation.

Tools are also provided for integrating *EcGeneralMotion* objects. Integrating *EcGeneralVelocity* to get an *EcCoordinateSystemTransformation* describing the position and orientation requires some calculation.

Let  $[Q_0, Q_1, Q_2, Q_3]^T$  represent a quaternion describing the position of a moving coordinate system and  $[\omega_0, \omega_1, \omega_2]^T$  represent the angular velocity of the moving coordinate system in reference frame (not moving frame) coordinates. The time derivative of the quaternion can be calculated using the following formula, which is a consequence of (4-1):

$$\begin{bmatrix} \dot{Q}_0 \\ \dot{Q}_1 \\ \dot{Q}_2 \\ \dot{Q}_3 \end{bmatrix} = \frac{1}{2} \begin{bmatrix} -Q_1 & -Q_2 & -Q_3 \\ Q_0 & Q_3 & -Q_2 \\ -Q_3 & Q_0 & Q_1 \\ Q_2 & -Q_1 & Q_0 \end{bmatrix} \begin{bmatrix} \omega_0 \\ \omega_1 \\ \omega_2 \end{bmatrix}. \quad (4-2)$$

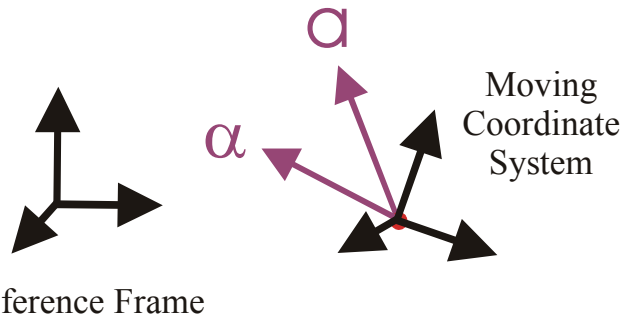
As a user of the toolkit, you do not need to work with this or other quaternion formulas. Equation (4-2) is implemented in the *EcOrientation* class. Let *q* be an *EcOrientation* object, *w* be angular velocity represented in reference-frame coordinates, and *vQdot* be a length-four real vector. Then `q.quaternionRateFromReferenceFrameAngularVelocity(w, vQdot);` will calculate the quaternion rate as shown in (4-2). Methods for calculating the quaternion rate from angular velocity represented in moving coordinates, calculating angular velocity from quaternion rates, and integrating angular velocity are all part of the implementation of *EcOrientation* in the toolkit—details are given in the class documentation.

## 5.4 Basic Dynamics

Just as with kinematics, for the most part, class details will not be described in this document, but rather in the class documentation. However, because of their importance, acceleration, force, and mass properties will be described in detail.

### 5.4.1 Rigid-Body Acceleration

As was mentioned previously, the motion of rigid bodies and reference frames is described in the toolkit using the *EcGeneralMotion* class. *EcGeneralMotion* is typedef'ed to *EcGeneralAcceleration* to represent acceleration. Each *EcGeneralAcceleration* object has one *EcVector* representing linear acceleration of a point and one *EcVector* representing angular acceleration about that point. Each As with *EcGeneralVelocity*, each *EcGeneralAcceleration* object must have an implicit point of application and a frame of representation. *EcGeneralAcceleration* is illustrated in the figure below.



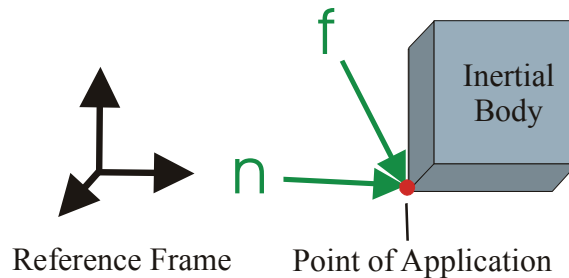
**Figure 5-3:** Reference-frame and Rigid-body acceleration are represented using *EcGeneralAcceleration* objects, which contain vectors for linear and angular velocity ( $\vec{a}$  and  $\vec{\alpha}$ , respectively). These are defined in an implicit reference frame at the origin of the moving frame or at an implicit point on a moving rigid body.

Method Examples	Description
<code>EcGeneralAcceleration(linear, angular);</code>	This constructor creates a general acceleration from a linear and an angular component.
<code>a.linear(); a.setLinear(a); a.angular(); a.setAngular(alpha);</code>	Methods to get and set the linear and angular components (both <i>EcVectors</i> ).
<code>a+=b; a-=b; a=b+c; a=b-c;</code>	Various math operators are defined among <i>EcGeneralAcceleration</i> objects.
<code>a*=r; a=r*b; a=b*r;</code>	Various math operations with scalars are also defined.
<code>a==b;</code>	Equality is defined, which returns <code>EcTrue</code> if two objects are equal.
<code>a.approxEq(b, tol);</code>	Fuzzy equality is defined, which returns <code>EcTrue</code> if all the position and orientation elements vary by less than the tolerance.
<code>a.transformBy(xform); a.transformBy(orient); a.transformBy(vector);</code>	Routines to transform the <i>EcGeneralAcceleration</i> object by moving the point of application, changing the reference frame, or both.

**Table 5-7:** Select methods in the *EcGeneralAcceleration* class. These are the same methods as for *EcGeneralVelocity*. For a complete description, please see the class documentation.

### 5.4.2 Rigid-Body Force

The sum of forces applied to a rigid body can be represented through a vector linear force and a vector moment applied to a point of application. This is illustrated in the figure below.



**Figure 5-4:** The force applied to a body is represented using *EcGeneralForce* objects, which contain vectors for linear force and angular moment ( $\vec{f}$  and  $\vec{n}$ , respectively). These are defined in an implicit reference frame at an implicit point of application.

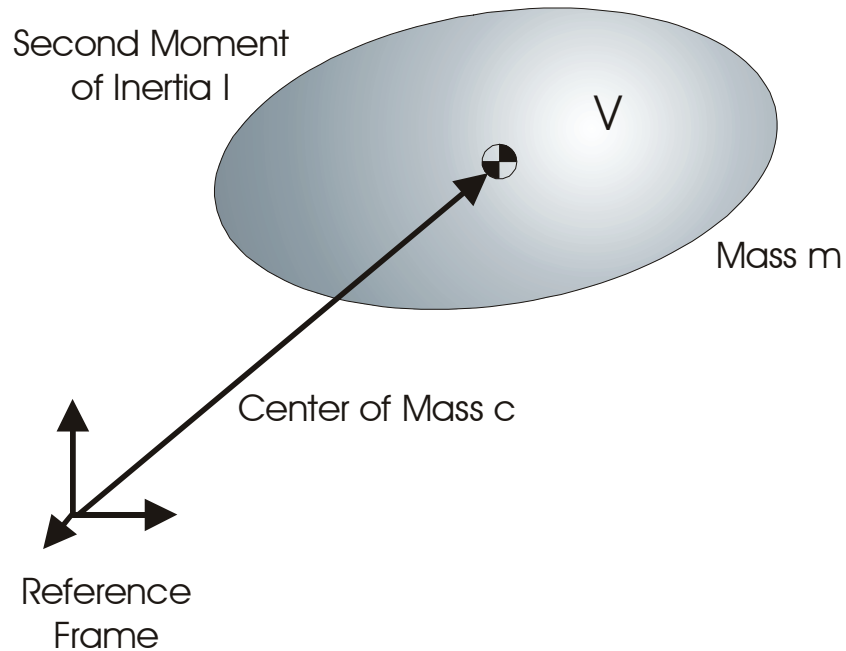
Method Examples	Description
<code>EcGeneralForce(linear, angular);</code>	This constructor creates a general force from a linear and an angular component.
<code>a.linear(); a.setLinear(f);</code> <code>a.angular(); a.setAngular(n);</code>	Methods to get and set the linear and angular components (both <i>EcVectors</i> ).
<code>a+=b; a-=b; a=b+c; a=b-c;</code>	Various math operators are defined among <i>EcGeneralForce</i> objects.
<code>a*=r; a=r*b; a=b*r;</code>	Various math operations with scalars are also defined.
<code>a==b;</code>	Equality is defined, which returns <i>EcTrue</i> if both are equal.
<code>a.approxEq(b, tol);</code>	Fuzzy equality is defined, which returns <i>EcTrue</i> if all the position and orientation elements vary by less than the tolerance.
<code>a.transformBy(xform);</code> <code>a.transformBy(orient);</code> <code>a.transformBy(vector);</code>	Routines to transform the <i>EcGeneralMotion</i> object by moving the point of application, changing the reference frame, or both.
<code>a.addLinear(f); a.addAngular(n);</code>	Routines to add linear and angular components.

**Table 5-8:** Select methods in the *EcGeneralForce* class. These are similar to those for *EcGeneralVelocity* and *EcGeneralAcceleration*.



### 5.4.3 Rigid Body Mass Properties

Rigid body mass properties include the 10 parameters needed for dynamics calculation: the scalar mass, the vector first moment of inertia, and the 3×3 symmetric second moment of inertia. For the first and second moments of inertia, there is an implied reference frame. The second moment is represented with point of application at the origin of the reference frame, not at the center of mass.



**Figure 5-5:** Rigid-body inertia includes the scalar mass  $m$ , the first moment of inertia, which is defined as the center of mass times the mass, and the second moment of inertia.

The mass of rigid body defined over volume  $V$  is given by

$$m = \int_V \rho dV, \quad (4-3)$$

where  $\rho$  is the mass density for the differential volume  $dV$ . This is a single scalar.

The first moment is given by

$$\vec{h} = \int_V \vec{r} \rho dV, \quad (4-4)$$

where  $\vec{r}$  is the vector pointing to the differential volume  $dV$ . The second moment is defined through a three-dimensional vector and is equal to  $m\vec{c}$ , where  $\vec{c}$  is the center of mass.

Second moment of inertia is defined through a symmetric 3x3 matrix:

$$I = \begin{bmatrix} I_{xx} & I_{xy} & I_{xz} \\ I_{xy} & I_{yy} & I_{yz} \\ I_{xz} & I_{yz} & I_{zz} \end{bmatrix}, \quad (4-5)$$

The values  $I_{xx}$ ,  $I_{yy}$ , and  $I_{zz}$  are the diagonal elements, given by

$$\begin{aligned} I_{xx} &= \int_V (y^2 + z^2) \rho dV \\ I_{yy} &= \int_V (x^2 + z^2) \rho dV \\ I_{zz} &= \int_V (x^2 + y^2) \rho dV \end{aligned} \quad (4-6)$$

The off-diagonal elements are defined as follows:

$$\begin{aligned} I_{xy} &= -\int_V xy \rho dV \\ I_{xz} &= -\int_V xz \rho dV \\ I_{yz} &= -\int_V yz \rho dV \end{aligned} \quad (4-7)$$

The second moment of inertia is represented through the class *EcSecondMoment*.

Method Examples	Description
<code>EcSecondMoment(jxx, jyy, jzz, jxy, jxz, jyz);</code>	This constructor creates a second moment from the entries in the matrix representation.
<code>a.jxx(); a.jyy(); a.jzz(); a.jxy(); a.jxz(); a.jyz();</code>	Methods to get individual components.
<code>a.set(jxx, jyy, jzz, jxy, jxz, jyz);</code>	Method to set values.
<code>a+=b; a-=b; a=b+c; a=b-c;</code>	Various math operators are defined.
<code>a*=r; a=r*b; a=b*r;</code>	Various math operations with scalars are also defined.
<code>a==b;</code>	Equality is defined, which returns <code>EcTrue</code> if both are equal.
<code>a.approxEq(b, tol);</code>	Fuzzy equality is defined, which returns <code>EcTrue</code> if all the position and orientation elements vary by

	less than the tolerance.
<code>a.transformBy(orient);</code>	Routines to transform the object through rotation.
<code>a.spatialMatrix();</code>	Gets the inertia as a 3x3 matrix.
<code>a.getPrincipalAxes();</code>	Gets the orientation of the frame aligned with the principal moments.

**Table 5-9** Select methods in the *EcSecondMoment* class.

The complete mass properties are represented by combining a scalar for mass, an *EcVector* for first moment, and an *EcSecondMoment* for second moment. This class, *EcRigidBodyMassProperties*, includes the methods shown in the table below.

Method Examples	Description
<code>EcRigidBodyMassProperties(mass, firstMoment, secondMoment);</code>	This constructor creates a second moment from components.
<code>a.mass(); a.firstMoment(); a.secondMoment();</code>	Methods to get individual components.
<code>a.set(mass, firstMoment, secondMoment);</code>	Method to set values.
<code>a+=b; a-=b; a=b+c; a=b-c;</code>	Various math operators are defined.
<code>a==b;</code>	Equality is defined, which returns <i>EcTrue</i> if both are equal.
<code>a.approxEq(b, tol);</code>	Fuzzy equality is defined, which returns <i>EcTrue</i> if all the position and orientation elements vary by less than the tolerance.
<code>a.transformBy(orientation); a.transformBy(translation); a.transformBy(xform);</code>	Routines to transform the object through rotation, translation, or general coordinate system transformation.
<code>a.calculateForce(velocity, acceleration);</code>	Calculates the force required to produce the specified frame velocity and frame acceleration.
<code>a.calculateAcceleration(velocity, force);</code>	Calculates the acceleration produced by the specified force when the body has the specified velocity.

**Table 5-10** Select methods in the *EcRigidBodyMassProperties* class.

## 6 XML

### 6.1 Overview

The Actin™ Toolkit provides extensive support for reading and writing XML configuration files. The toolkit contains XML objects that enable the developer to provide read and write support for any class member variable. By turning the class and member variables into XML objects, the member variables can be seamlessly read from or written to an XML file. These XML objects can also be member variables in other XML objects. Through this approach, complex class structures can have read and write capability throughout the class hierarchy.

This section is broken down into three components: 1) a description of the XML objects and how the developer can reuse them to create new objects with read and write capability, 2) a description of the XML reader and writer, and 3) a description of the XML schema auto-generator.

### 6.2 XML Objects

Through the use of the Actin™ Toolkit, the developer can create new C++ classes with XML read and write support. These classes can contain a spectrum of data types ranging from simple data to complicated hierarchies. For example, a class can include simple data, other classes made up of simple data, classes nested with other classes, and complex classes that are defined during run-time (i.e., polymorphically).

All XML objects build upon the abstract base class *EcXmlObject*. Table 6-1 shows the methods defined through this class. Through inheritance of *EcXmlObject*, read and write support is available to any class.

Method	Description
Big Four	Most classes within the code base require the “Big Four”: default constructor, destructor, copy constructor, and assignment operator.
==	Equivalence operator.
clone	Abstract method for cloning XML object.
selfTest	Abstract method for executing self test.
equality	Abstract virtual equality to an <i>EcXmlObject</i> pointer
newObject	Abstract virtual new to an <i>EcXmlObject</i> pointer
xmlInit	Initialize XML object.
read	Reads an XML object from an XML stream.
write	Writes an XML object to an XML stream.
writeSchema	Writes the schema for this object to an XML stream. See Section

	6.4 for more details.
isBasicType	Return true if object is a basic type; otherwise return false. The default is false. <i>EcXmlBasicType</i> and <i>EcXmlEnumType</i> override this method and return true. This is currently only used for schema generation and the <i>EcXmlVectorType</i> writer.
readFromStream	Read object from an istream. See Section 6.3 for more details.
writeToStream	Write object to an ostream. See Section 6.3 for more details.
readFromFile	Read object from a file. See Section 6.3 for more details.
writeToFile	Write object to a file. See Section 6.3 for more details.
readFromFilePlain	Read object from an uncompressed file. See Section 6.3 for more details.
writeToFilePlain	Write object to an uncompressed file. See Section 6.3 for more details.
readFromFileWithCompression	Read object from a compressed file. See Section 6.3 for more details.
writeToFileWithCompression	Write object to a compressed file. See Section 6.3 for more details.
readFromUrl	Read object from URL. See Section 6.3 for more details.
readFromBuffer	Read object from a buffer. See Section 6.3 for more details.
writeToBuffer	Write object to a buffer. See Section 6.3 for more details.
readFromCompressedBuffer	Read object from a compressed buffer. See Section 6.3 for more details.
writeToCompressedBuffer	Write object to a compressed buffer. See Section 6.3 for more details.
readFromTcpSocket	Read object from TCP socket. See Section 6.3 for more details.
writeToTcpSocket	Write object to TCP socket. See Section 6.3 for more details.
createSchema	Write a complete schema. This is called by <code>writeToPlainFile</code> , and it generates the schema for the XML object.
xmlInitialized	Get XML initialized flag. This flag is set upon read and write initialization.

setXmlInitialized	Set XML initialized flag. This flag is set upon read and write initialization.
specified	Get specified flag. This bit is set upon reading from an XML file or from the copy constructor.
setSpecified	Set specified flag. This bit is set upon reading from an XML file or from the copy constructor.

**Table 6-1:** List of methods in *EcXmlObject*. The blue highlighted methods are the typically overridden methods.

Although all of the methods are virtual and can be overridden through inheritance, the highlighted methods are likely candidates for being overridden. The *read*, *write*, *writeSchema*, and *xmlInit* methods are described in Section 6.2.4. The abstract methods, *clone*, *selfTest*, *equality*, and *newObject*, are described here and have the following prototypes.

```

    /// return cloned object
    virtual EcXmlObject* clone
    (
        ) const=0;

    /// perform self test
    virtual EcBoolean selfTest
    (
        ) const=0;

    /// equality - a virtual equality to an EcXmlObject pointer
    virtual EcBoolean equality
    (
        const EcXmlObject* other
        ) const=0;

    /// creates new object - a virtual new to an EcXmlObject pointer
    virtual EcXmlObject* newObject
    (
        ) const=0;

```

**Text Box 6-1:** The abstract method prototypes of *EcXmlObject*.

An example definition for the *clone* method is shown in Text Box 6-2. Each XML object needs its own *clone* definition.

```

EcXmlObject* EcXmlExample::clone
(
    ) const
{
    return ( new EcXmlExample(*this) );
}

```

**Text Box 6-2:** Example definition for the *clone* method.

An example definition for the *selfTest* method is shown in Text Box 6-3. This example contains testing for the read, write, copy constructor, assignment, and equivalence methods. The developer

can add testing for all aspects of the class. As code problems surface, the self-test can be updated to minimize the chance of the problem reoccurring.

```
EcBoolean EcXmlExample::selfTest
(
) const
{
    EcBoolean retVal=EcTrue;

    // create three test objects
    EcXmlExample* value1=new EcXmlExample;
    EcXmlExample* value2=new EcXmlExample;
    EcXmlExample* value3=new EcXmlExample;

    // get a test value for first test object
    *value1 = EcXmlExample::testObject();

    // test write/read for test object
    EcString filename=EcString("EcXmlExample.xml");

    value1->writeToFile(filename);
    value2->readFromFile(filename);

    // compare the original to the read value
    if(!(*value1==*value2))
    {
        retVal=EcFalse;
    }

    // test the assignment operator
    *value3=*value1;

    // compare the original to the copied value
    if(!(*value1==*value3))
    {
        retVal=EcFalse;
    }

    // test the copy constructor
    EcDELETE(value3);
    value3=new EcXmlExample(*value1);

    // compare the original to the copied value
    if(!(*value1==*value3))
    {
        retVal=EcFalse;
    }

    // clean up
    EcDELETE(value1);
    EcDELETE(value2);
    EcDELETE(value3);

    return retVal;
}
```

**Text Box 6-3:** Example definition for the selfTest method. This is in the XML example code.

The *equality* method is illustrated in Text Box 6-4 and performs a virtual test on the equality of the two XML objects.

```
EcBoolean EcXmlExample::equality
(
    const EcXmlObject* other
) const
{
    EcBoolean retVal=EcTrue;

    // cast XML object to EcXmlExample
    const EcXmlExample* cast=
        dynamic_cast<const EcXmlExample*>(other);

    // test equality
    if(!cast || !(*this==*cast))
    {
        retVal=EcFalse;
    }

    return retVal;
}
```

**Text Box 6-4:** Example definition for *equality* method.

The *newObject* method is illustrated in Text Box 6-5 and is a factory method for creating a new XML object.

```
EcXmlObject* EcXmlExample::newObject
(
) const
{
    return new EcXmlExample();
}
```

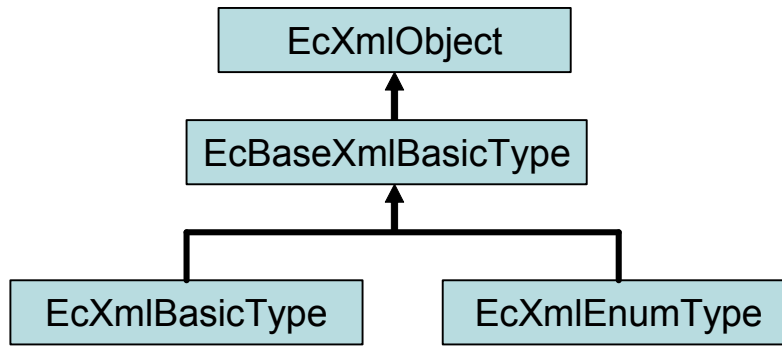
**Text Box 6-5:** Example definition for the *newObject* method.

The Actin™ Toolkit defines several types of XML objects: 1) basic types for simple data, 2) XML containers for Standard Template Library (STL) containers, 3) a static compound type and 4) variable compound types.

### **6.2.1 Basic Types for Simple Data**

There are two classes for defining basic data types: *EcXmlBasicType* and *EcXmlEnumType*. The basic XML object enables the reading and writing of simple data types such as strings and floats. The enumeration type enables the reading and writing of enumerations. The simple data managed by these simple XML objects represents the content (i.e. leaf nodes) in the XML files. Figure 6-1 illustrates the classes that provide support for enumerations and basic data such as strings and floats. Table 6-2 provides a summary description of these classes.





**Figure 6-1:** Inheritance diagram for simple XML objects. These objects provide read and write support for the simple member variables of a configurable XML object.

Class	Description	Example XML
<i>EcXmlBasicType</i> <ElementType>	XML basic type for adding C++ basic types to an XML description file. See Table 6-3 for a list of type definitions.	<alpha>1e-008</alpha>
<i>EcXmlEnumType</i> <ElementType>	XML enumeration type for adding self describing enumeration types to an XML description file. ElementType is an integral C++ type. See Table 6-5 for a list of type definitions.	<dhType>genPaul</dhType>

**Table 6-2:** XML types for defining simple member variables that have XML read/write support. The member variables are registered by the *registerComponents* method for the class, which needs to be a compound XML object as defined below (Table 6-17). These XML types are templates with many specific type definitions predefined at the bottom of the header files.

### 6.2.1.1 EcXmlBasicType

The following example, Text Box 6-6, illustrates how simple types can be used. The example variables, *input* and *output*, are XML objects using the *EcXmlBasicType* class. The list of type definitions is in Table 6-3. Although this example shows a legitimate way for using XML objects, generally the XML objects are class member variables. Table 6-4 shows the methods available through *EcXmlBasicType*.

```

// create a test value for writing
// EcXmlReal is a type definition of EcXmlBasicType<EcReal>
EcXmlReal output2=5.0;
EcXmlReal input2;

// write/read the test value
EcString filename = "EcXmlBasicType.xml";

output2.writeToFile(filename);
input2.readFromFile(filename);

// print the output and input
EcWARN("Example: EcXmlBaseType.\n");
std::cout << "Output: " << output2 << std::endl;
std::cout << "Input:  " << input2  << std::endl;

```

**Text Box 6-6:** Example code using *EcXmlBasicType* variables. Table 5-2 shows the format for the XML file. This is Example Section #2 in the XML example code.

Type Definition	Template Type
<i>EcXmlReal</i>	<i>EcReal</i>
<i>EcXmlU32</i>	<i>EcU32</i>
<i>EcXmlInt32</i>	<i>EcInt32</i>
<i>EcXmlU16</i>	<i>EcU16</i>
<i>EcXmlInt16</i>	<i>EcInt16</i>
<i>EcXmlU8</i>	<i>EcU8</i>
<i>EcXmlInt8</i>	<i>EcInt8</i>
<i>EcXmlAngle</i>	<i>EcAngle</i>
<i>EcXmlNonNegReal</i>	<i>EcNonNegReal</i>
<i>EcXmlBoolean</i>	<i>EcBoolean</i>
<i>EcXmlString</i>	<i>EcString</i>

**Table 6-3:** Type definitions for *EcXmlBasicType* contained in *ecXmlBasicType.h*.

Method	Description
Big Four	Most classes within the code base require the “Big Four”: default constructor, destructor, copy constructor, and assignment operator.
==	Equivalence operator.
cast operator	Cast to a basic type.
<	Less than operator.
clone	Clone XML object.
equality	Equality for an <i>EcXmlObject</i> pointer
newObject	Factory method for an <i>EcXmlObject</i> pointer
selfTest	Execute self test.
read	Reads an XML object from an XML stream.
write	Writes an XML object to an XML stream.
writeSchema	Write the schema for this object to an XML stream.
isBasicType	Returns true due to being a basic type
value	Get value of basic type.
setValue	Set value of basic type.
nullObject	Static method for getting an empty object.
testValue	Get a sample value for testing.
setValueFromString	Set the value from a string.

**Table 6-4:** List of methods in *EcXmlBasicType*. The methods for *EcXmlBasicType* are nonvirtual and inlined for speed. See the class documentation for details.

### 6.2.1.2 EcXmlEnumType

The following example, Text Box 6-7, illustrates how enumeration types can be used. The example variables, *input* and *output*, are XML objects using the *EcXmlEnumType* class. The list of type definitions is in Table 6-5. Table 6-6 shows the methods available through *EcXmlEnumType*.

```

// create a test value for writing
// EcXmlEnumU16 is a type definition of EcXmlEnumType<EcU16>
EcXmlEnumU16 output3=0;
EcXmlEnumU16 input3;

// create strings for enumerations
output3.setEnumString( 0, "Enum0" );
output3.setEnumString( 1, "Enum1" );
input3.setEnumString( 0, "Enum0" );
input3.setEnumString( 1, "Enum1" );

// write/read the test value
filename = "EcXmlEnumType.xml";

output3.writeToFile(filename);
input3.readFromFile(filename);

// print the output and input
EcWARN("Example: EcXmlEnumType.\n");
std::cout << "Output: " << output3.enumString() << std::endl;
std::cout << "Input:  " << input3.enumString() << std::endl;

```

**Text Box 6-7:** Example code using *EcXmlEnumType* variables. This is Example Section #3 in the XML example code.

Type Definition	Template Type
<i>EcXmlEnumU32</i>	<i>EcU32</i>
<i>EcXmlEnumInt32</i>	<i>EcInt32</i>
<i>EcXmlEnumU16</i>	<i>EcU16</i>
<i>EcXmlEnumInt16</i>	<i>EcInt16</i>
<i>EcXmlEnumU8</i>	<i>EcU8</i>
<i>EcXmlEnumNonNegReal</i>	<i>EcNonNegReal</i>

**Table 6-5:** Type definitions for *EcXmlEnumType* contained in *ecXmlEnumType.h..*

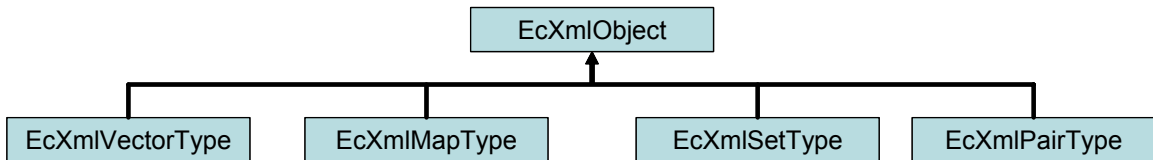
Method	Description
Big Four	Most classes within the code base require the “Big Four”: default constructor, destructor, copy constructor, and assignment operator.
==	Equivalence operator.

cast operator	Cast to a basic type
<	Less than operator.
clone	Clone XML object.
equality	Equality for an <i>EcXmlObject</i> pointer
newObject	Factory method for an <i>EcXmlObject</i> pointer
selfTest	Execute self test.
read	Reads an XML object from an XML stream.
write	Writes an XML object to an XML stream.
writeSchema	Write the schema for this object to an XML stream.
isBasicType	Returns true due to being a basic type
value	Get enumeration value.
setValue	Set enumeration value.
enumString	Return the enumeration string based on current value.
setEnumString	Set the specified enumeration string for the given value.
findEnumString	Lookup the given string and return the appropriate enumeration.
nullObject	Static method for getting an empty object.
testValue	Get a sample value for testing.

**Table 6-6:** List of methods in *EcXmlEnumType*. See the class documentation for details.

### 6.2.2 XML Object Containers for STL Containers

The Actin™ toolkit uses several STL containers. For read and write support, several XML containers were created to complement the STL. Figure 6-2 illustrates the classes that provide support for STL containers. Table 6-7 provides a summary description of these classes.



**Figure 6-2:** Inheritance diagram for STL XML objects. These objects provide read and write support to the STL containers.

Class	Description	Example XML
<i>EcXmlVectorType</i> <ElementType>	XML vector type for adding general vectors to an XML description files. See Table 6-8 for a list of type definitions.	<pre>&lt;jointTorques size="2"&gt;   &lt;element&gt;1&lt;/element&gt;   &lt;element&gt;-2&lt;/element&gt; &lt;/jointTorques&gt;</pre> <p>or</p> <pre>&lt;jointTorques size="2"&gt;   &lt;group&gt;1 -2&lt;/group&gt; &lt;/jointTorques&gt;</pre>
<i>EcXmlMapType</i> <KeyType, ValueType>	XML map type for adding general maps to an XML description file. See Table 6-10 for a list of type definitions.	<pre>&lt;integerMap&gt;   &lt;element&gt;     &lt;key&gt;string1&lt;/key&gt;     &lt;value&gt;1&lt;/value&gt;   &lt;/element&gt;   &lt;element&gt;     &lt;key&gt;string2&lt;/key&gt;     &lt;value&gt;2&lt;/value&gt;   &lt;/element&gt; &lt;/integerMap&gt;</pre>
<i>EcXmlSetType</i>	XML set type for adding general sets to an XML description file. See Table 6-12 for a list of type definitions.	<pre>&lt;setType size="3"&gt;   &lt;group&gt;4 7 8&lt;/group&gt; &lt;/setType&gt;</pre>
<i>EcXmlPairType</i>	XML pair type for adding general pairs to an XML description file. See Table 6-14 for a list of type definitions.	<pre>&lt;pairType&gt;   &lt;first&gt;string1&lt;/first&gt;   &lt;second&gt;string2&lt;/second&gt; &lt;/setType&gt;</pre>

**Table 6-7** XML types for defining STL container member variables that have XML read/write support. The member variables are registered by the *registerComponents* method for the class, which needs to be a compound XML object as defined below (Table 6-17). Most of these XML types are templates with many specific type definitions predefined at the bottom of the header files.

### 6.2.2.1 EcXmlVectorType

The following example, Text Box 6-8, illustrates how vector types can be used. The example variables, *input* and *output*, are XML objects using the *EcXmlVectorType* class. The list of type definitions is in Table 6-8. Table 6-9 shows the methods available through *EcXmlVectorType*.

```

// create a test value for writing
// EcXmlRealVector is a type definition of
EcXmlVectorType<EcXmlReal>
EcXmlRealVector output5;
EcXmlRealVector input5;

output5.resize(3);
output5[0]=0;
output5[1]=1;
output5[2]=2;

// write/read the test value
filename = "EcXmlVectorType.xml";

output5.writeToFile(filename);
input5.readFromFile(filename);

// print the output and input
EcWARN("Example: EcXmlVectorType.\n");
std::cout << "Output: " << output5[2] << std::endl;
std::cout << "Input:  " << input5[2] << std::endl;

```

**Text Box 6-8:** Example code using *EcXmlVectorType* variables. This is Example Section #5 in the XML example code.

Type Definition	Template Type
<i>EcXmlU32Vector</i>	<i>EcXmlU32</i>
<i>EcXmlU32VectorVector</i>	<i>EcXmlU32Vector</i>
<i>EcXmlInt32Vector</i>	<i>EcXmlInt32</i>
<i>EcXmlInt32VectorVector</i>	<i>EcXmlInt32Vector</i>
<i>EcXmlInt16Vector</i>	<i>EcXmlInt16</i>
<i>EcXmlU16Vector</i>	<i>EcXmlU16</i>
<i>EcXmlInt8Vector</i>	<i>EcXmlInt8</i>
<i>EcXmlU8Vector</i>	<i>EcXmlU8</i>
<i>EcXmlRealVector</i>	<i>EcXmlReal</i>
<i>EcXmlRealVectorVector</i>	<i>EcXmlRealVector</i>
<i>EcXmlRealVector3D</i>	<i>EcXmlRealVectorVector</i>
<i>EcXmlStringVector</i>	<i>EcXmlString</i>

<i>EcXmlStringVectorVector</i>	<i>EcXmlStringVector</i>
<i>EcXmlStringVector3D</i>	<i>EcXmlStringVectorVector</i>
<i>EcXmlBooleanVector</i>	<i>EcXmlBoolean</i>
<i>EcXmlBooleanVectorVector</i>	<i>EcXmlBooleanVector</i>

**Table 6-8:** Type definitions for *EcXmlVectorType* contained in *ecXmlVectorType.h*. *EcXmlVectorType* can be of any *EcXmlObject* type – not just basic types.

Method	Description
Big Four	Most classes within the code base require the “Big Four”: default constructor, destructor, copy constructor, and assignment operator.
==	Equivalence operator.
Constructor with vector size	Construct object and allocate vector size.
[]	Get/Set by index.
clone	Clone XML object.
equality	Equality for an <i>EcXmlObject</i> pointer
newObject	Factory method for an <i>EcXmlObject</i> pointer
selfTest	Execute self test.
read	Reads an XML object from an XML stream.
write	Writes an XML object to an XML stream.
writeSchema	Write the schema for this object to an XML stream.
pushBack	Push a copy of an object onto the vector.
vectorContainer	Get the vector container.
reserve	Reserve space in the vector container.
resize	Resize the vector container.
clear	Clear all entries in the vector container.
assign	Assign an object to a range of values.



size	Get the length of the vector.
leftRotate	Left rotate by the specified amount.
nullObject	Static method for getting an empty object.
testValue	Get a test value of the element type.

**Table 6-9:** List of methods in *EcXmlVectorType*. See the class documentation for details.

### 6.2.2.2 EcXmlMapType

The following example, Text Box 6-9, illustrates how map types can be used. The example variables, *input* and *output*, are XML objects using the *EcXmlMapType* class. The list of type definitions is in Table 6-10. Table 6-11 shows the methods available through *EcXmlMapType*.

```

// create a test value for writing
// EcXmlStringU32Map is of type EcXmlMapType<EcXmlString, EcXmlU32>
EcXmlStringU32Map output4;
output4.add(EcString("key"), 5);
EcXmlStringU32Map input4;

// write/read the test value
filename = "EcXmlMapType.xml";

output4.writeToFile(filename);
input4.readFromFile(filename);

// print the output and input
EcWARN("Example: EcXmlMapType.\n");
EcXmlU32 value;
output4.lookup(EcString("key"), value);
std::cout << "Output: " << value << std::endl;
input4.lookup(EcString("key"), value);
std::cout << "Input:  " << value  << std::endl;

```

**Text Box 6-9:** Example code using *EcXmlMapType* variables. This is Example Section #4 in the XML example code.

Type Definition	Template Type
<i>EcXmlStringStringMap</i>	< <i>EcXmlString</i> , <i>EcXmlString</i> >
<i>EcXmlStringInt32Map</i>	< <i>EcXmlString</i> , <i>EcXmlInt32</i> >
<i>EcXmlStringU32Map</i>	< <i>EcXmlString</i> , <i>EcXmlU32</i> >
<i>EcXmlStringRealMap</i>	< <i>EcXmlString</i> , <i>EcXmlReal</i> >
<i>EcXmlStringBooleanMap</i>	< <i>EcXmlString</i> , <i>EcXmlBoolean</i> >
<i>EcXmlStringStringRealMap</i>	< <i>EcXmlString</i> , <i>EcXmlStringRealMap</i> >
<i>EcXmlStringStringBooleanMap</i>	< <i>EcXmlString</i> , <i>EcXmlStringBooleanMap</i> >

**Table 6-10:** Type definitions for *EcXmlMapType* contained in *ecXmlMapType.h*. *EcXmlMapType* can be of any *EcXmlObject* type – not just basic types. The first type has the same restrictions as STL maps. That is, it is required to have a sorting operator.

Method	Description
Big Four	Most classes within the code base require the “Big Four”: default constructor, destructor, copy constructor, and assignment operator.
==	Equivalence operator.
clone	Clone XML object.
equality	Equality for an <i>EcXmlObject</i> pointer
newObject	Factory method for an <i>EcXmlObject</i> pointer
selfTest	Execute self test.
read	Reads an XML object from an XML stream.
write	Writes an XML object to an XML stream.
writeSchema	Write the schema for this object to an XML stream.
lookup	Lookup a value from a key.
lookupPointer	Lookup a value pointer from a key.
add	Add a key/value pair.

erase	Erase a key/value pair.
clear	Erase all key/value pairs, leaving an empty map.
mapContainer	Get the map container.
keyIndex	Get the index (the order in the map) of a key.
indexKey	Get the key at an index location (the order in the map).
size	Get the size of the map container.
nullObject	Static method for getting an empty object.
testKey	Get a test key of the element type.
testValue	Get a test value of the value type.

**Table 6-11:** List of methods in *EcXmlMapType*. See the class documentation for details.

### 6.2.2.3 EcXmlSetType

*EcXmlSetType* is an XML object containers for STL sets. The list of type definitions is in Table 6-12. Table 6-13 shows the methods available through *EcXmlSetType*.

Type Definition	Template Type
<i>EcXmlU32Set</i>	< <i>EcXmlU32</i> >

**Table 6-12:** Type definition for *EcXmlSetType* contained in *ecXmlSetType.h*. *EcXmlSetType* can be of any *EcXmlObject* type – not just basic types. The template type has the same restrictions as STL sets. That is, it is required to have a sorting operator.

Method	Description
Big Four	Most classes within the code base require the “Big Four”: default constructor, destructor, copy constructor, and assignment operator.
==	Equivalence operator.
clone	Clone XML object.
equality	Equality for an <i>EcXmlObject</i> pointer
newObject	Factory method for an <i>EcXmlObject</i> pointer

selfTest	Execute self test.
read	Reads an XML object from an XML stream.
write	Writes an XML object to an XML stream.
writeSchema	Write the schema for this object to an XML stream.
add	Add a key/value pair.
erase	Erase a key/value pair.
clear	Erase all key/value pairs, leaving an empty map.
setContainer	Get the set container.
size	Get the size of the map container.
nullObject	Static method for getting an empty object.

**Table 6-13:** List of methods in *EcXmlSetType*. See the class documentation for details.

#### 6.2.2.4 EcXmlPairType

*EcXmlPairType* is an XML object containers for STL pairs. The list of type definitions is in Table 6-14. Table 6-15 shows the methods available through *EcXmlPairType*.

Type Definition	Template Type
<i>EcXmlStringStringPair</i>	< <i>EcXmlString</i> , <i>EcXmlString</i> >
<i>EcXmlStringU32Pair</i>	< <i>EcXmlString</i> , <i>EcXmlU32</i> >
<i>EcXmlU32U32Pair</i>	< <i>EcXmlU32</i> , <i>EcXmlU32</i> >

**Table 6-14:** Type definition for *EcXmlPairType* contained in *ecXmlPairType.h*. *EcXmlPairType* can be of any *EcXmlObject* type – not just basic types. The first and second template types have the same restrictions as for STL pairs. That is, they are required to have a sorting operator.

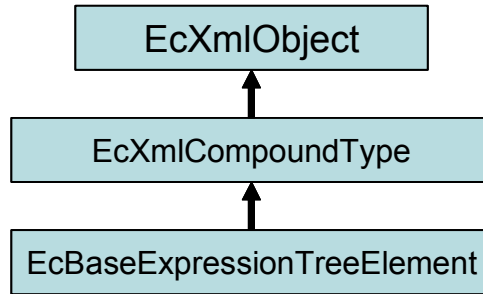
Method	Description
Big Four	Most classes within the code base require the “Big Four”: default constructor, destructor, copy constructor, and assignment operator.
==	Equivalence operator.

clone	Clone XML object.
equality	Equality for an <i>EcXmlObject</i> pointer
newObject	Factory method for an <i>EcXmlObject</i> pointer
selfTest	Execute self test.
read	Reads an XML object from an XML stream.
write	Writes an XML object to an XML stream.
writeSchema	Write the schema for this object to an XML stream.
first	Get the first value.
setFirst	Set the first value.
second	Get the second value.
setSecond	Set the second value.
set	Set both the first and second values.
pairContainer	Get the pair container.
nullObject	Static method for getting an empty object.

**Table 6-15:** List of methods in *EcXmlPairType*. See the class documentation for details.

### 6.2.3 Compound XML Objects

Compound XML objects enable a developer to build detailed objects using simple XML objects or other compound XML objects. Figure 6-3 illustrates the compound XML class. *EcBaseExpressionTreeElement* is the compound type needed for base expression tree containers. Table 6-16 provides a summary description of this class.



**Figure 6-3:** Inheritance diagram for the compound XML objects. Analogous to C++ class, this compound type can contain a combination of simple objects, STL containers, and other static and variable compound objects.

Name	Description
<i>EcXmlCompoundType</i>	Defines an abstract XML type with a fixed set of compound XML data. Each element in the set is registered in <i>registerComponents</i> . The <i>EcGeneralMotion</i> class is an example compound XML object containing two <i>EcXmlVectors</i> for linear and angular motion.
<i>EcBaseExpressionTreeElement</i>	Defines an abstract XML base class for the element types of <i>EcBaseExpressionTreeContainer</i> and the other variable compound types. It holds the pointer to the container for all expression elements. For example, <i>EcExpressionElement</i> inherits this base class.

**Table 6-16:** Base classes for defining compound XML object types. For a complete description, see the class documentation.

### 6.2.3.1 EcXmlCompoundType Description

Many classes in the toolkit have member variables that are statically defined at load-time, but are configurable during run-time (e.g., through a configuration file). The *EcXmlCompoundType* class is generally used for classes like these. Table 6-17 shows the methods of *EcXmlCompoundType*. The methods highlighted in blue are often overridden by the developer and are described in more detail in this section.

Method	Description
Big Four	Most classes within the code base require the “Big Four”: default constructor, destructor, copy constructor, and assignment operator.
==	Equivalence operator.

registerComponents	Abstract method for registering the components of the XML object.
xmlInit	Initialize XML object. It also calls <i>registerComponents</i> .
read	Reads an XML object from an XML stream.
write	Writes an XML object to an XML stream.
writeSchema	Write the schema for this object to an XML stream.
readValueFromSpecialToken	Read value from unregistered token. Specialized reading can be done here. The default method prints a warning that an unregistered token was encountered.
registerComponent	Register a single component.
componentMap	Get a reference to the component map.
registerAttributeComponent	Register a single attribute component.
attributeComponentMap	Get a reference to the component attribute map.
hasChildren	Return EcTrue if there are children elements.
hasAttributes	Return EcTrue if there are attributes in the element.
readValueFromToken	Reads an XML object corresponding to a token from a stream.
readAttributeFromToken	Reads an XML attribute corresponding to a token.
newComponentMap	Factor method for the component map.
createComponentMap	Create a component maps.
createAttributeComponentMap	Create attribute component map.

**Table 6-17:** List of methods in *EcXmlCompoundType*. The blue highlighted methods are the typically overridden methods. See the class documentation for details.

The abstract method, *registerComponents*, is used for registering the member variables, attributes, and enumerations of the class. If none of these items exist, the method can be empty. Text Box 6-10 shows an example definition.

```

void EcEndEffector::registerComponents
(
)
{
    // register components (string, XML object reference)
    registerComponent(EcLinkIdentifierToken, &m_LinkIdentifier);
    registerComponent(EcRelativeLinkDataToken, &m_RelativeLinkData);
    registerComponent(EcIsHardConstraintToken, &m_IsHardConstraint);
    registerComponent(EcEditingLabelsToken, &m_EditingLabels);
}

```

**Text Box 6-10:** Example definition for registerComponents. If *EcXmlExample* is subclassed, its *registerComponents* method will need to call *EcXmlExample*'s version.

The *registerComponents* method is called from the *xmlInit* method of the *EcXmlCompoundType* class. *xmlInit* does not generally need to be overridden – that is, the definition in *EcXmlCompoundType* is usually appropriate for its children. Text Box 6-11 shows an example definition for *xmlInit*. Class specific initializations can be added to this method. If *EcXmlExample* is further subclassed, the children of *EcXmlExample* need to call *EcXmlExample::xmlInit* instead of *EcXmlCompoundType::xmlInit*.

```

EcBoolean EcEndEffector::xmlInit
(
)
{
    if(EcXmlCompoundType::xmlInit() )
    {
        // place class specific XML initializations here
        return EcTrue;
    }
    else
    {
        return EcFalse;
    }
}

```

**Text Box 6-11:** Example definition for *xmlInit*.

Text Box 6-12 shows an example definition for *read*. The *read* method generally performs three actions:

1. Pre-load initializations such as freeing memory.
2. Read configuration file for the registered variables and their children. *EcXmlCompoundType::read* iterates over the registered variables. If *EcXmlExample* is subclassed, the subclass needs to call *EcXmlExample::read*.
3. Post-load initializations. At this point, the configuration file settings can be used for initialization.

Steps 1 and 3 are often omitted. With steps 1 and 3 omitted, the *EcXmlExample::read* method can also be omitted. With this omission, the *EcXmlCompoundType::read* method would be called in its place.



The *read* method can also be customized for speed. For example, the *EcXmlVectorVector* class is a highly used class with a customized *read* function for faster reading. The tradeoff is that less error checking is performed during reading. Section 6.3 describes these *read* functions.

```
EcBoolean EcEndEffector::read
(
    EcXmlReader& stream
)
{
    // Place load initializations here. For example, pointers to
members
// about to be loaded can be deleted. Oftentimes, nothing needs
to be
// done before reading.

    EcBoolean retVal= EcXmlCompoundType::read(stream);

    // Now that the configuration file settings have been loaded for
this
// object, further initializations can be performed here.

    return retVal;
}
```

**Text Box 6-12:** Example definition for the *read* method.

Text Box 6-13 shows an example definition for *write*. The *write* method generally performs two actions:

1. Write configuration file for registered variables and their children. *EcXmlCompoundType::write* iterates over the registered variables. If *EcXmlExample* is subclassed, the subclass needs to call *EcXmlExample::write*.
2. Write unregistered variables.

Step 2 is often unnecessary. If step 2 is omitted, the *EcXmlExample::write* method can also be omitted. With this omission, the *EcXmlCompoundType::write* method would be called in its place.

```
EcBoolean EcEndEffector::write
(
    EcXmlWriter& stream
) const
{
    EcBoolean retVal= EcXmlCompoundType::write(stream);

    // Manually write out unregistered variables here

    return retVal;
}
```

**Text Box 6-13:** Example definition for the *write* method.

Text Box 6-14 shows an example definition for *writeSchema*. The *writeSchema* method generally performs two actions that are analogous to the steps for the *write* method:

1. Write schema for registered variables and their children.  
*EcXmlCompoundType::writeSchema* iterates over the registered variables. If *EcXmlExample* is subclassed, the subclass needs to call *EcXmlExample::writeSchema*.
2. Write schema for unregistered variables.

Step 2 is often unnecessary. If step 2 is omitted, the *EcXmlExample::writeSchema* method can also be omitted. With this omission, the *EcXmlCompoundType::writeSchema* method would be called in its place.

```
EcBoolean EcEndEffector::writeSchema
(
  EcXmlSchema& stream
) const
{
  EcBoolean retVal= EcXmlCompoundType::writeSchema(stream);

  // Manually write out schema for unregistered variables here

  return retVal;
}
```

**Text Box 6-14:** Example definition for the *writeSchema* method.

The *readValueFromSpecialToken* method is only needed when unregistered variables are read. Generally, this method is left empty which is the default definition in *EcXmlCompoundType*.

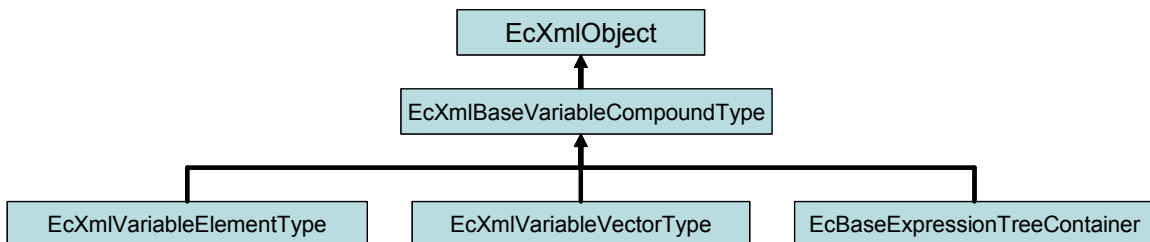
### 6.2.3.2 Adding Child Members to *EcXmlCompoundType*

The primary benefit of subclassing from *EcXmlCompoundType* lies in the ability to easily add new data to the object in form of a member variable that is subclassed from *EcXmlObject*. Any subclass of *EcXmlObject* can be added as a child member to a subclass of *EcXmlCompoundType* through the following procedure:

1. Add the new object as a class member variable and include it in the copy constructor and *operator=()* definitions.
2. Establish a new string token to define it
3. Register the new variable with the token inside the class

### 6.2.4 Variable Compound XML Objects

Variable compound XML objects enable a developer to build detailed objects with a selection of subclassed compound XML objects. Figure 6-4 illustrates the compound XML class. Table 6-18 provides a summary description of the variable compound classes.



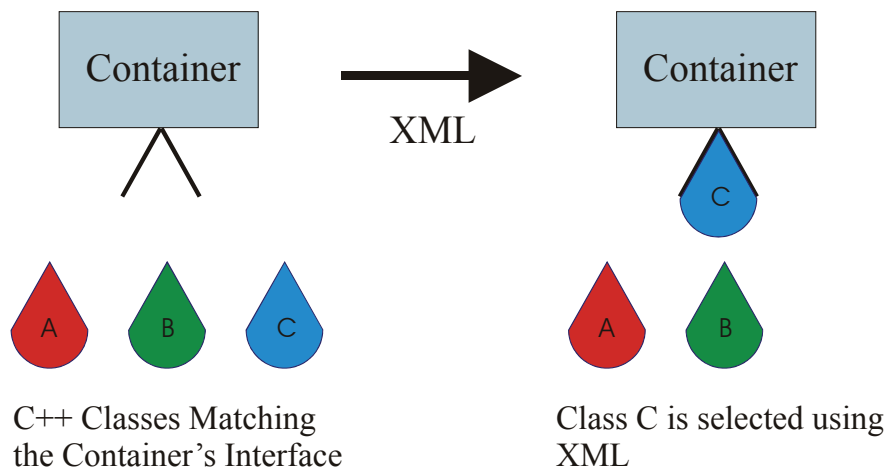
**Figure 6-4:** Inheritance diagram for the variable compound XML objects.

Name	Description
<i>EcXmlBaseVariableCompoundType</i>	Defines an abstract base class for the following three classes: <i>EcXmlVariableElementType</i> , <i>EcXmlVariableVectorType</i> , <i>EcBaseExpressionTreeContainer</i> . These classes are for building useful containers.
<i>EcXmlVariableElementType</i> <ExpressionType>	Defines a container that can hold any compound and variable compound object.
<i>EcXmlVariableVectorType</i> <ElementType>	Defines an abstract XML type with a vector of configurable (i.e., polymorphic) compound XML data types. Each element in the vector has a common base class as defined by <i>ElementType</i> . All options for the vector elements must be registered in <i>registerComponentCreators</i> or set through a factory method. The <i>EcEndEffectorVector</i> class is an example. The options for the vector are <i>EcPointEndEffector</i> , <i>EcFrameEndEffector</i> , and <i>EcXyEndEffector</i> with each using the <i>EcEndEffector</i> base class which is the <i>ElementType</i> .
<i>EcBaseExpressionTreeContainer</i> <ExpressionType>	Defines an abstract XML base class for a logical expression tree. Each element in the tree has a common base class of <i>ExpressionType</i> . All options for the tree elements must be registered in <i>registerComponentCreators</i> . The <i>EcExpressionContainer</i> class is an example. There are many expression element options defined in <i>EcExpressionContainer</i> with each having a common base class called <i>EcExpressionElement</i> which is the <i>ElementType</i> .

**Table 6-18:** Base classes for defining compound XML object types. For a complete description, see the class documentation.

#### 6.2.4.1 EcXmlBaseVariableCompoundType

*EcXmlBaseVariableCompoundType* is an abstract base class for *EcXmlVariableElementType*, *EcXmlVariableVectorType*, and *EcBaseExpressionTreeContainer*. Each of these classes is a container that holds other compound XML objects. These containers can be subclassed to develop new containers that have registered members which can be defined polymorphically as specified by the XML file. Figure 6-5 illustrates how the container is assigned its content.



**Figure 6-5:** Each of the containers defined through *EcXmlBaseVariableCompoundType* provides an interface. There may be multiple classes that can meet this interface—such as those illustrated as A, B, or C. These classes may be defined using dynamic link libraries that are written after the simulator is created or they can be added through a factory object. This will support easy enhancement of the simulator. The object used at run time is specified through an XML language.

Table 6-19 shows the methods of *EcXmlBaseVariableCompoundType*.

Method	Description
Big Four	Most classes within the code base require the “Big Four”: default constructor, destructor, copy constructor, and assignment operator.
==	Equivalence operator.
token	Abstract method for getting a string token which describes the object.
registerComponentCreators	Abstract method for registering the components creators of the XML object.
xmlInit	Initialize XML object. It also calls <i>registerComponentCreators</i> .
read	Reads an XML object from an XML stream.
write	Writes an XML object to an XML stream.
writeSchema	Write the schema for this object to an XML stream.
readValueFromToken	Abstract method that reads an XML object corresponding to a token from a stream. The three subclasses define this method, so the developer can generally use it as is.
registerComponentCreator	Register a single optional component.

newObjectFromToken	Create a new object from its token.
--------------------	-------------------------------------

**Table 6-19:** List of methods in *EcXmlBaseVariableCompoundType*. The blue highlighted methods are the typically overridden methods.

The *token* and *registerComponentCreators* methods are the only “blue” methods that are unique from the *EcXmlCompoundType* class. These abstract methods need a definition by the subclass. Example definitions are in Text Box 6-15 and Text Box 6-16.

```
const EcString& EcExpressionContainer::token
(
) const
{
    return EcExpressionContainerToken;
}
```

**Text Box 6-15:** Example definition for *token*. *classToken* is a static function defined by *EcXmlExample*.

```
void EcExpressionContainer::registerComponentCreators
(
)
{
    // register component creators (string, &creator())

    // register scalar constants
    registerComponentCreator
    (
        EcExpressionScalarConstant::classToken(),
        EcExpressionScalarConstant::creator
    );

    // register plus
    registerComponentCreator
    (
        EcExpressionPlus::classToken(),
        EcExpressionPlus::creator
    );
}
```

**Text Box 6-16:** Example definition for *registerComponentCreators*. The methods, *classToken* and *creator*, are defined below

Each registered variable needs a static *classToken* and *creator* method. Here are example definitions for these methods.

```

EcXmlObject* EcExpressionScalarConstant::creator
(
)
{
    EcXmlObject* retVal=new EcExpressionScalarConstant ();

    return retVal;
}

```

**Text Box 6-17:** Example definition for *creator*.

Every subclass of *EcXmlBaseVariableCompoundType* now has a member variable that is an instance of the class *EcXmlFactory*. A factory creates objects of any XML-configurable class from a string token.

The interface to *EcXmlFactory* is defined primarily through the following two methods:

```

virtual void registerComponentCreator
(
    const EcToken& token,
    EcXmlObjectCreator creator
);

virtual EcXmlObject* newObjectFromToken
(
    const EcToken& token
) const;

```

The first method, `registerComponentCreator()` registers a static creator method with a token. This can be called any number of times with new creator functions and new tokens. Once all the relevant creator functions have been registered, `newObjectFromToken()` can be called to allocate memory and create a new object of the type signaled by the token—provided this token had been used in an earlier registration. (The method returns NULL if no creator function is registered for the given token.)

This new use of *EcXmlFactory* standardizes the method of creating objects for all subclasses of *EcXmlBaseVariableCompoundType*. It also plays a new, important role. A developer creating a new system can register creators with a master factory that is used throughout the load process. So, rather than register new creator functions in remote portions of the XML database that defines the robot simulation, the creator function can be registered in one place.

The process of loading new object types using this new method is described through the following steps:

1. Create an *EcXmlFactory* object. Call it “factory,” say.
2. Register the new class type you want to load by calling `factory.registerComponentCreator()` with a creator function and a unique string token.
3. Create an *EcXmlReader* object. Call it “reader,” say.
4. Provide a pointer for the factory to the reader, through `reader.setFactoryPointer()`.
5. Create an object of the type you want to read. Call it “object,” say.
6. Read the object from an XML file that contains the new type of data you want to load using `object.read(reader)`.

### 6.2.4.2 EcXmlVariableElementType

The *EcXmlVariableElementType* class inherits the *EcXmlBaseVariableCompoundType* class and supports the development of containers that have registered members that are defined polymorphically as specified by the XML file. This class is unique in that it can contain one *EcXmlCompoundType*, *EcXmlVariableVectorType*, *EcXmlVariableElementType*, or *EcBaseExpressionTreeContainer* object. The table below shows the methods of *EcXmlVariableElementType*.

Method	Description
Big Four	Most classes within the code base require the “Big Four”: default constructor, destructor, copy constructor, and assignment operator.
==	Equivalence operator.
read	Reads an XML object from an XML stream.
write	Writes an XML object to an XML stream.
readValueFromToken	Reads an XML object corresponding to a token from a stream.
element	Get a pointer to the contained element.
setElement	Set the element through a copy.

**Table 6-20:** List of methods in *EcXmlVariableElementType*. The blue highlighted methods are the typically overridden methods.

### 6.2.4.3 EcXmlVariableVectorType

The *EcXmlVariableVectorType* class supports the development of vectors where each element is defined polymorphically as specified by the XML file. The table below shows the methods of *EcXmlVariableVectorType*.

Method	Description
Big Four	Most classes within the code base require the “Big Four”: default constructor, destructor, copy constructor, and assignment operator.
==	Equivalence operator.
Constructor with vector size	Construct object and allocate vector size.
[]	Get/Set by index.
registerComponentCreators	Abstract method for registering the components creators of the XML object.

xmlInit	Initialize XML object. It also calls <i>registerComponentCreators</i> .
read	Reads an XML object from an XML stream.
write	Writes an XML object to an XML stream.
writeSchema	Write the schema for this object to an XML stream.
readValueFromToken	Reads an XML object corresponding to a token from a stream.
pushBack	Push a copy of an object onto the vector.
vectorContainer	Get the vector container.
reserve	Reserve space in the vector container.
resize	Resize the vector container.
clear	Clear all entries in the vector container.
assign	Assign an object to a range of values.
size	Get the length of the vector.
registerComponentCreator	Register a single optional component.
newObjectFromToken	Get a reference to the component map.

**Table 6-21:** List of methods in *EcXmlVariableVectorType*. The blue highlighted methods are the typically overridden methods.

#### 6.2.4.4 EcBaseExpressionTreeContainer

The *EcBaseExpressionTreeContainer* class inherits the *EcXmlBaseVariableCompoundType* class and supports the development of containers that have registered members that are defined polymorphically as specified by the XML file. This container supports the development of tree structures such as unary and binary trees. The table below shows the methods of *EcBaseExpressionTreeContainer*.

Method	Description
Big Four	Most classes within the code base require the “Big Four”: default constructor, destructor, copy constructor, and assignment operator.
==	Equivalence operator.
read	Reads an XML object from an XML stream.
write	Writes an XML object to an XML stream.



writeSchema	Write the schema for this object to an XML stream.
setTopElementContainerToThis	Set the container pointer for the top element to this.
readValueFromToken	Reads an XML object corresponding to a token from a stream.
topElement	Get a pointer to the top expression.
setTopElement	Set the top expression through a copy.
setAndDeleteTopElementPointer	Set the top expression pointer and later deletes this pointer.

**Table 6-22:** List of methods in *EcBaseExpressionTreeContainer*. The blue highlighted methods are the typically overridden methods.

The *setTopElementContainerToThis* method is the only “blue” method that is unique from the other complex XML class methods. Here is an example definition of *setTopElementContainerToThis*.

```

void EcExpressionContainer::setTopElementContainerToThis
(
)
{
    // m_pTopElement points to the top element in the tree.
    // it is contained in EcBaseExpressionTreeContainer
    if (m_pTopElement)
    {
        m_pTopElement->setContainer(this);
    }
}

```

**Text Box 6-18:** Example definition for *setTopElementContainerToThis*.

#### 6.2.4.5 EcBaseExpressionTreeElement

The *EcBaseExpressionTreeElement* class inherits the *EcXmlCompoundType* class and is a base class for all the elements that get registered in *registerComponentCreators* of *EcBaseExpressionTreeContainer*. It holds a pointer to the container, and it contains the *get* and *set* methods for the container. The table below shows the methods of *EcBaseExpressionTreeElement*.

Method	Description
Big Four	Most classes within the code base require the “Big Four”: default constructor, destructor, copy constructor, and assignment operator.
==	Equivalence operator.
token	Abstract method for getting a string token which describes the

	object.
registerComponents	Registers components for class.
setContainer	Set the container.
container	Get the container.

**Table 6-23:** List of methods in *EcBaseExpressionTreeElement*. The blue highlighted methods are the typically overridden methods.

The `setContainer` method is generally overridden for branching classes such as for *EcExpressionBaseBinary*. Here is an example definition of `setContainer`.

```
void EcBaseExpressionTreeElement::setContainer
(
    const EcXmlBaseVariableCompoundType* container
)
{
    m_pContainer=container;

    // If EcXmlExample is a branching class (e.g., EcExpressionBaseBinary)
    // set containers for children here.
}
```

**Text Box 6-19:** Example definition for `setContainer`.

Below is an illustration of the `read`, `write`, `writeSchema`, and `readValueFromSpecialToken` methods from *EcExpressionBaseBinary*. The *EcExpressionBaseBinary* class is a branching class that does not have any registered variables of its own, but creates new elements based on the registered components of the container. The container class is *EcExpressionContainer*. These examples illustrate some of the variations that can occur.

```
EcBoolean EcExpressionBaseBinary::read
(
    EcXmlReader& stream
)
{
    // free memory and set the child pointers to null.
    EcDELETE(m_pLeft);
    EcDELETE(m_pRight);

    EcBoolean retVal=EcExpressionElement::read(stream);

    return retVal;
}
```

**Text Box 6-20:** Example definition for the *EcExpressionBaseBinary* `read` method.

```

EcBoolean EcExpressionBaseBinary::write
(
    EcXmlWriter& stream
) const
{
    // write all the composite types
    EcBoolean retVal=EcExpressionElement::write(stream);

    // write the left child
    if(m_pLeft)
    {
        stream.writeStartTag(m_pLeft->token());

        m_pLeft->write(stream);

        stream.writeEndTag();
    }

    // write the right child
    if(m_pRight)
    {
        stream.writeStartTag(m_pRight->token());

        m_pRight->write(stream);

        stream.writeEndTag();
    }

    return retVal;
}

```

**Text Box 6-21:** Example definition for the *EcExpressionBaseBinary* write method.

```
EcBoolean EcExpressionBaseBinary::writeSchema
(
  EcXmlSchema& stream
) const
{
  // write all the composite types
  EcBoolean retVal = EcExpressionElement::writeSchema(stream);

  if( container() )
  {
    // create left child
    container()->writeSchema(stream);

    // create right child
    container()->writeSchema(stream);
  }

  return retVal;
}
```

**Text Box 6-22:** Example definition for the *EcExpressionBaseBinary* *writeSchema* method.

```

EcBoolean EcExpressionBaseBinary::readValueFromSpecialToken
(
    const EcToken& token,
    EcXmlReader& stream
)
{
    EcBoolean retVal=EcTrue;

    if(container())
    {
        // create the element from a creator function in the container's map
        EcExpressionElement* element=
            dynamic_cast<EcExpressionElement*>(
                container()->newObjectFromToken(token));

        // if there was no registered function for the token or it was not
        // registered as an EcExpressionElement, an error occurred.
        if(element==0)
        {
            retVal=EcFalse;
        }
        else
        {
            // if the pointer is OK, set its container to this.
            element->setContainer(container());

            // and read the data from the stream
            element->read(stream);
        }

        // set the left and right elements in that order
        if(m_pLeft==0)
        {
            // we haven't yet set the left child--set it now
            m_pLeft=element;
        }
        else
        {
            // we have set the left element--set the right one
            EcDELETE(m_pRight);
            m_pRight=element;
        }
    }
    else
    {
        // warn the user
        EcWARN("Bad token: %s on line %d\n",
            token.c_str(),stream.lineCountOfFile());
        return EcFalse;
    }

    return retVal;
}

```

**Text Box 6-23:** Example definition for the *EcExpressionBaseBinary* *readValueFromSpecialToken* method.

#### 6.2.4.6 EcXmlFactory

The variable compound class containers have a creator map that holds the options for each container. The options are set through the *registerComponentCreator* method. The *EcXmlFactory* class enables

the developer to add options without having to subclass the registerComponentCreator. The table below shows the methods of *EcXmlFactory*.

Method	Description
Big Four	Most classes within the code base require the “Big Four”: default constructor, destructor, copy constructor, and assignment operator.
==	Equivalence operator.
registerComponentCreator	Add a token and creator to the factory map.
newObjectFromtoken	Given a token, return an object.

**Table 6-24:** List of methods in *EcXmlFactory*.

Here is example code for using the factory class.

```

// initialize the factory
EcXmlFactory factory;
factory.registerComponentCreator
(
    EcFirstClass::classToken(),
    EcFirstClass::creator
);
factory.registerComponentCreator
(
    EcSecondClass::classToken(),
    EcSecondClass::creator
);

// add factory to XML reader
EcXmlReader reader;
Reader.setFactoryPointer(&factory);

// read XML file and set test object
// XML file can contain new tokens as contained in factory above.
testObject.readFromFile(filename);

```

**Text Box 6-24:** Example code for XML factory class.

This concludes the description of the XML objects. The next section describes the XML reader and writer streams used by the XML objects.

### 6.3 XML Reading and Writing XML Objects

The previous section shows how to create XML objects. This section shows how to read and write the XML given an XML object. Many of the details of actually reading and writing XML are hidden in the XML objects. This section describes the interface to the XML reader and writer so that a developer can directly utilize this interface when necessary.

### 6.3.1 Top Level Interface for Reading and Writing

There are several options for reading and writing XML data including plain text, compressed text, URLs (read only), TCP streams, and general streams. The XML files can also contain XLinks which redirect the XML reader to a URL for getting XML fragments.

#### 6.3.1.1 Plain Text, Compressed Text, and URLs

Two primary functions exist in *EcXmlObject* for reading and writing XML files: `readFromFile` and `writeToFile`. Given an XML object and a file name with `.xml` or `.ecx` file extension, these functions can read and write the files. Depending on the file name, these functions can read/write a plain, compressed, or URL file. If the file name ends with “.gz” (the typical gzip suffix) or `.ecz`, then the file is inflated upon reading and compressed upon writing. If the file name starts with “<http://>”, then the URL is read.

Text Box 6-25 shows an example for reading and writing an XML file. Text Box 6-26 illustrates options for filename specification.

```
// write XML object to file
xmlObject.writeToFile(filename);

// read XML object from file
xmlObject.readFromFile(filename);
```

**Text Box 6-25:** Example code for reading and writing an XML file.

```
// Plain text
EcString filename="xmlObject.ecx";

// Compressed text
EcString filename="xmlObject.ecz";

// URL
EcString filename="http://www.energid.com/XML/xmlObject.xml";
```

**Text Box 6-26:** Example filenames supported by the toolkit. The URL option only works when reading XML files.

The compression option uses ZLIB. ZLIB compression is compatible with GZIP, so any third party GZIP utility can open these files.

#### 6.3.1.2 TCP Streams

TCP streams provide capability to read and write to a TCP socket. This is useful for a distributed environment. Text Box 6-27 contains an example usage for reading and writing to TCP streams.

```

// Create a socket
EcTCPSocket* pSocket = new EcTCPSocket();

// Connect to the server
pSocket->connect(EcString("127.0.0.1"), 6689);

// write XML object to socket
xmlObject.writeToTcpSocket(pSocket);

// read XML object from socket
xmlObject.readFromTcpSocket(pSocket);

```

**Text Box 6-27:** Example code for reading and writing an XML file using a TCP socket. *pSocket* is a pointer to a TCP socket. An IP address and port is all that is needed to open the TCP socket. The toolkit also provides mutex support which may be needed for controlling communication between a client and server.

### 6.3.1.3 Generic Streams

The XML reader and writer utilize the standard template library *istream* and *ostream* respectively. A developer can create new streams based on *istream* and *ostream* for XML reading and writing. The methods, *readFromStream* and *writeToStream*, are used for this purpose. These methods take a stream reference instead of a filename. The developer needs to open the stream before the call. Text Box 6-28 shows an example.

```

// Transfer internal data from xmlObject1 to xmlObject2 through a string.
// This is equivalent (albeit slower) to writing
// xmlObject1=xmlObject2

// create output string stream
std::ostream outputStream;

// put xmlObject1's data into string stream
xmlObject1.writeToStream(outputStream, "buffer");

// transfer data to xmlString.
// xmlString contains full contents of an XML file.
EcString xmlString = outputStream.str();

// create input string stream, and initialize with xmlString
std::istream inputStream(xmlString);

// put contents of xmlString into xmlObject2
xmlObject2.readFromStream(inputStream, "buffer");

```

**Text Box 6-28:** Example use of *readFromStream* and *writeToStream*. This example uses a string to transfer the contents of one XML file to another. *readFromBuffer* and *writeToBuffer* are available to perform this task more concisely. This is Example Section #9 in the XML example code.

Through the use of *readFromStream* and *writeToStream*, the developer can create new streams for use in the toolkit. Several streams have been created that are available to the user. The table below shows the list of options.



Stream Name	Include File	Open File Example
Compressed Input Stream	ecIfGzStream.h	EcIfGzStream ifs(filename);
Compressed Output Stream	ecOfGzStream.h	EcOfGzStream ofs(filename);
TCP I/O Stream	ecTcpStream.h	EcTcpStream tcp(pSocket);
HTTP Input Stream	ecIfHttpStream.h	EcIfHttpStream ifs(url);
Compressed I/O String Stream	ecGzStringStream.h	EcGzStringStream ifs(string);

**Table 6-25:** List of available streams.

### 6.3.1.4 XLinks

Similarly to HTML, XLink provides a hyperlink capability to XML using an attribute-based syntax. Although XLink has a wider range of capabilities, our code base supports the HTML-like absolute and relative unidirectional hyperlinks. The XML standard defines an XLink namespace for this task. Text Box 6-29 illustrates the usage of XLink.

```

1 <root xmlns:xlink = "http://www.w3.org/1999/xlink">
2   <include xlink:type = "simple" xlink:href =
3     "http://www.energid.com/XML/EcTableFunctionContainer.xml"/>
4   <include xlink:type = "simple" xlink:href =
5     "EcTableFunctionContainer.xml"/>
6 </root>

```

**Text Box 6-29:** XLink example.

Line 1 defines the namespace for XLink. Line 2 and Line 4 illustrate absolute and relative URLs respectively (these two lines are redundant – only one of them is needed). The absolute URL capability enables users to get XML files from any worldwide location. The relative URL capability provides a useful technique for putting “include” files in an XML file. This will enable the larger input files to be broken down by objects into more manageable files. These smaller files can be reused in several locations. Although the include files are fragments of a larger XML input file, each fragment must also be a well-formed XML file.

## 6.3.2 Direct Interface to the XML Reader and Writer

The XML reader and writer can be called directly for reading or constructing an XML file. This interface is sometimes needed when building new XML objects.

### 6.3.2.1 XML Reader

Table 6-26 shows the methods available for reading XML files. The *readXml* method reads events based on XML syntax and stores the information of that event in the *EcXmlReader* object. An event can be a start tag, empty tag, content, end tag, end-of-file, and error. The event type is returned by *readXml* and it can also be accessed through *mode*. Each tag (i.e., start, empty, and end tag) has an element name that can be accessed through *element*. If the event is a start tag or empty tag, attribute names and values may be available. The *numOfAttributes* method returns the number of attributes read. The attribute names and values can be returned through *attributeName* and *attributeValue*. The attribute name and value pairs are stored in a stack. Every time *attributeValue* is called, the

stack is popped and the next attribute pair is available. The *remainingNumOfAttributes* method is available for determining the current size of the attribute stack. If the event is content, then the *content*, *contentByWord*, and *contentCount* methods are available for getting the content. *content* returns the whole content as a string. *contentByWord* and *contentCount* are useful for getting list content one word at a time. Since content is returned as a string, string streams are useful for converting to other types of data.

A set of fast methods are available for quickly reading an XML file. Error checking and storage of XML data is turned off when using these methods. For example, the *element* method will return an empty string when only fast methods are used.

Method	Description
readXml	Finds the next start tag, content, end tag, or end of file – whichever is next in the stream.
mode	Gets the XML_READER_MODE from the last readXml call which can be EOF_FOUND, START_TAG, END_TAG, EMPTY_TAG, CONTENT, or ERROR_FOUND.
element	Gets start tag element name.
attributeName	Gets next attribute name.
attributeValue	Gets next attribute value. Prior to returning the value, the current attribute name and value are deleted from memory.
numOfAttributes	Gets number of attributes in start tag.
remainingNumOfAttributes	Gets the remaining number of attributes in memory.
content	Get content.
contentByWord	Get content by word. This is useful for content lists.
contentCount	Get the number of words in content.
filename	Get the filename.
lineCountOfFile	Get the current line number in file.
nextMatchingEndTag	Skip to next matching end tag in file. This is useful if the XML object does not recognize the element name in the start tag. The full tag is skipped by this method call.
stream	Get stream reference. The XML object can use the stream to parse the file using an alternate method.
getComplexStartTagFast	Fast approach for getting next start tag. It is assumed that the start tag has attributes. Returns element name.

getSimpleStartTagFast	Fast approach for getting next start tag. It is assumed that there are no attributes in start tag. Returns element name.
getAttributeValueFast	Fast approach for getting next attribute value. Returns next attribute value.
getLastAttributeValueFast	Needed for getting last attribute value. It reads up to content or next start tag. Returns last attribute value.
getContentByWordFast	Fast approach for getting content word by word. Returns next content word. The number of words needs to be known through the attributes or known a priori (e.g., <i>EcXmlVector</i> is always of size 3).
getEndTagFast	Fast approach for getting end tag.

**Table 6-26:** List of methods in *EcXmlReader*.

Text Box 6-30 illustrates some of the methods of *EcXmlReader*. This example resembles the read method of *EcXmlVectorVector* for reading one *EcXmlVector*.

```

// create XML reader (assume ifs, an istream, is available)
EcXmlReader stream(filename,ifs);

// read empty tag with 3 attributes (EcXmlVector)
EcXmlFileReader::XML_READER_MODE mode = stream.readXml();

// empty tag expected for EcXmlVector
if ( mode != EcXmlFileReader::EMPTY_TAG )
{
    EcWARN
    ("EcXmlVector Error: file not well formed.\n\tEmpty tag expected."
     "\n\tFile: %s\n\tLine: %d",
     stream.filename(), stream.lineCountOfFile());
}

// could warn if size != 3 which is necessary for EcXmlVector
EcU16 size = stream.numOfAttributes();

// loop through attributes
for( EcU32 ii=0; ii<size; ++ii)
{
    // the attribute name
    EcString token = stream.attributeName();

    if ( token == EcVectorXToken )
    {
        m_Vector[0]=atof(stream.attributeValue().c_str());
    }
    else if ( token == EcVectorYToken )
    {
        m_Vector[1]=atof(stream.attributeValue().c_str());
    }
    else if ( token == EcVectorZToken )
    {
        m_Vector[2]=atof(stream.attributeValue().c_str());
    }
}

// no need to read end tag, because this tag was empty

```

**Text Box 6-30:** Example of reading an XML file. This example was largely extracted from *EcXmlVectorVector* for the reading of one *EcXmlVector*.

### 6.3.2.2 XML Writer

Table 6-27 shows the methods available for writing XML files. The *writeStartTag* method writes out a start tag with the element name that is passed to it. When the root element is written (i.e., the first start tag), a header is written prior to the tag. The majority of elements in the toolkit are part of a namespace. There are a few elements that are generic and belong to the parent's namespace. For example, the "element" tokens of *EcXmlVectorVector* are not associated with any namespace. The *writeStartTagUsingParentNamespace* method is used for these start tags. A start tag can optionally have attributes. The attribute name is written using *setAttributeName*. Attribute values are written using the << output operator. If the start tag has children, the *EcXmlObject* *write* method traverses the children. If the start tag has basic content (e.g., int, float, ...), the << output operator is also used for writing the content. The << operator can be used repetitively for list content. The default number of columns for list data is 10, which can be changed through *setNumOfContentColumns*. The *writeEndTag* method closes the element.

<b>Method</b>	<b>Description</b>
writeStartTag	Write start tag.
writeStartTagUsingParentName space	Write start tag. The element does not have a namespace so the parent's namespace is used.
setAttributeName	Write attribute name.
<<	Output stream operator for writing data (i.e., attribute values and content).
setNumOfContentColumns	Sets the number of content columns. Used for lists.
writeEndTag	Write end tag.
indent	Get indent string.
setIndent	Set indent string.
defaultIndent	Get default indent string.
setDefaultIndent	Set default indent string.
header	Get header for XML file.
setHeader	Set header for XML file.
defaultHeader	Get default header for XML file.
setDefaultHeader	Set default header for XML file.
styleSheet	Get style-sheet name.
setStyleSheet	Set style-sheet name.
defaultStyleSheet	Get default style-sheet name.
setDefaultStyleSheet	Set default style-sheet name.
schemaInstance	Get schema instance.
setSchemaInstance	Set schema instance.
defaultSchemaInstance	Get default schema instance.
setDefaultSchemaInstance	Set default schema instance.

schemaLocation	Get schema location.
setSchemaLocation	Set schema location.
defaultSchemaLocation	Get default schema location.
setDefaultSchemaLocation	Set default schema location.
setLanguageSelection	Set XML language (e.g., MCML).
setDefaultLanguageSelection	Set default XML language.

**Table 6-27:** List of methods in *EcXmlWriter*.

Text Box 6-31 illustrates some of the methods of *EcXmlWriter*. This example resembles the write method of *EcXmlVectorVector* for writing one *EcXmlVector*.

```

// create XML writer (assume ofs, an ostream, is available)
EcXmlWriter stream(filename,ofs);

// write the start tag
stream.startTag(EcElementToken);

// write the attribute name and value for X.
stream.attributeName(EcVectorXToken);
stream<<x;

// write the attribute name and value for Y.
stream.attributeName(EcVectorYToken);
stream<<y;

// write the attribute name and value for Z.
stream.attributeName(EcVectorZToken);
stream<<z;

// No content ... empty tag
stream.endTag();

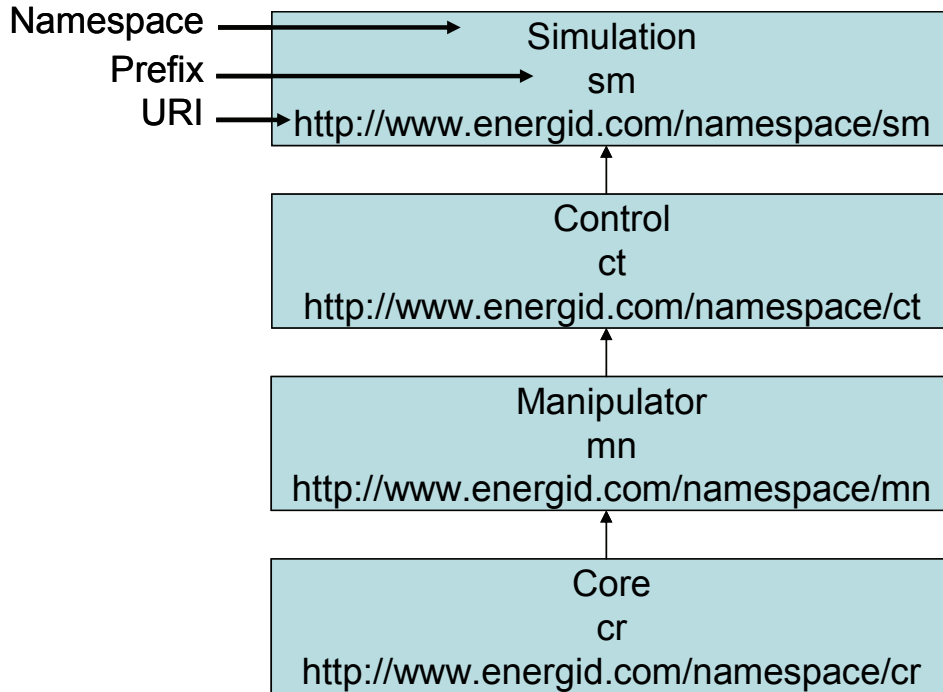
```

**Text Box 6-31:** Example of writing an XML file. This example was largely extracted from *EcXmlVectorVector* for the printing of one *EcXmlVector*.

## 6.4 Schema

### 6.4.1 XML Namespaces

The Actin™ Toolkit uses several XML namespaces. These namespaces impact the schema auto-generator in several ways, so a brief discussion of XML namespaces is warranted. The toolkit has four namespaces as shown in Figure 6-6.



**Figure 6-6:** The toolkit contains four XML namespaces: simulation, control, manipulator, and core. The upper namespaces import the lower namespaces. The prefix matches the last token of the URI.

The four XML namespaces were created to support the robotics simulation, control, and manipulator and core algorithms. If the developer uses the toolkit to create a simulation, then the “simulation” namespace is the root namespace, which means that the root element of the XML file is in the “simulation” namespace. In general, the “simulation” namespace imports the other namespaces in the order presented in Figure 6-6. Intermediate namespaces can also be skipped; for example, the “simulation” namespace elements can have “core” elements as their immediate children.

The XML writer within the toolkit places namespaces on all elements, but not on attributes. It is assumed that attributes belong to the namespace of their element, unless explicitly placed in another namespace. All elements and attributes are defined as tokens in the token header files for their respective project. The XML namespace is embedded within the token using the following syntax.

```
const EcToken tokenContainer = "URI#token";
```

**Text Box 6-32:** Token syntax.

Here is an example:

```
const EcToken EcTableFunctionInterpolatorToken =  
    "http://www.energid.com/namespace/cr#tableFunctionInterpolator";
```

**Text Box 6-33:** Token example.

This notation (i.e., the use of the ‘#’ delimiter) is a common notation used in literature. Since namespace prefixes are re-definable within the XML file, use of the URI is often preferable to the prefix. This notation is also convenient for our purposes, because it is easy to parse and place in an *EcToken* object. *EcToken* is a class that contains a string for the token and a string for the URI.

XML namespaces are specified as URIs. Since these URIs are more verbose than is desired for typical input file syntax, an abbreviated string is used in the form of a namespace prefix for each element. The prefixes are fully defined within the XML file. Although the prefixes could be anything as defined in the XML file, use of the namespace prefixes is recommended. But if a user chooses a different prefix (say because of a conflict), the XML reader and toolkit would work just fine.

Let us look at a few examples of XML namespace usage in our XML files. Here is the root element for “simulation.xml” which initializes the simulator.

```
<simulation xmlns="http://www.energid.com/namespace/sm">
```

**Text Box 6-34:** Simulation namespace example using a default namespace.

The “simulation” element is part of the “simulation” namespace as defined by the namespace attribute. The attribute defines “simulation” as the default namespace. This is done by specifying no namespace prefix with the “simulation” URI. Alternatively, the tag could have looked like this

```
<sm:simulation xmlns:sm="http://www.energid.com/namespace/sm">
```

**Text Box 6-35:** Simulation namespace example with a specified prefix.

Both notations work equivalently.

The “simulation” element has a “maxIterations” child element in the “control” namespace as shown below:

```
<ct:maxIterations  
    xmlns:ct="http://www.energid.com/namespace/ct">16</ct:maxIterations>
```

**Text Box 6-36:** Control namespace example.

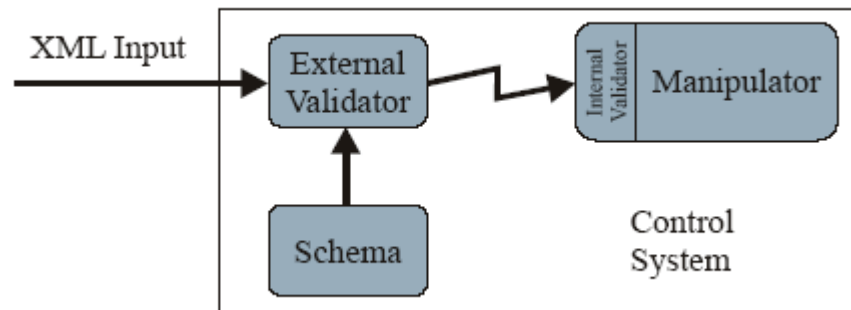
The namespace prefix is defined to be “ct” for the “control” namespace. Since this is the first occurrence of this namespace, it needs to be defined in the start tag. As a coding standard, our code base only writes one default namespace. Therefore, all imported namespaces have an explicitly defined namespace prefix. A user can redefine the default namespace to be any of the four namespaces.

The XML reader is less restrictive than the writer. For example, XML namespaces are optional, and multiple default namespaces are allowed by the XML reader.



## 6.4.2 Schema Auto-Generation

An XML schema defines a class of XML documents in much the same way a C++ class defines a class of objects. Taking this analogy further, an XML file is an instance document of the schema just like a C++ object is an instance of its class. Multiple schemas importing multiple namespaces can be combined to define a larger class of XML documents; much the same way inheritance works in C++. Through the capabilities available through a schema, flexible XML vocabularies can be defined that mirror the complex data structures defined in the toolkit. Although XML files are not compiled like C++ code, there are third-party XML tools that analyze the XML instance document along with the schema to provide validation and error checking (Figure 6-7). These third-party tools are quite beneficial to the toolkit.



**Figure 6-7:** Validation process.

The Actin™ Toolkit is unique in that it contains the definition of the XML instance files in its code and it auto-generates the files as requested by the user. A *writeToFile* method defined in Section 6.3 is available to the user for writing the contents of an object to an XML file. In order to auto-generate the XML file, the code base needs to contain the same information that a schema contains. The toolkit also auto-generates the schema while it writes the XML file. As the code base is reconfigured or upgraded with new classes, the schema will automatically be maintained within the code base. Through this process, the XML files and schemas can be auto-generated and then validated by a third party validator.

A schema file contains the definition for one XML namespace. Using the simulation as an example, see Figure 6-6, the schema auto-generation creates a schema file for each of the four namespaces.

The table below shows the methods available for auto-generating a schema.

Method	Description
<code>writeStartTag</code>	Register start tag and open an element. Optionally, a sequence, choice, or unbounded mode can be defined through the <i>startTag</i> method. Sequence is generally the default. Elements in a sequence must follow a certain order. The schema by default orders elements alphabetically. Choice is generally used by the variable types like expression trees. For example, each branch uses one choice from available options. Unbounded is used by vectors such as <i>EcXmlVectorType</i> . The vector length is not known a priori.
<code>writeStartTagUsingParentName space</code>	Register start tag. Place element in parent's namespace.

setAttributeName	Register attribute name.
setType	This method is context based. If an attribute value is expected due to previous use of attributeName, then this registers the type of the attribute value. If attributeName was not previously called (i.e., startTag was called), the content type is registered. For content, two calls to type signify that the content is a list. For example, if the content is a list of doubles, one call to type registers the double type, the next call upgrades the type to double-list.
writeEndTag	Close the element
openGroup	Open a group. Many compound elements only have one group associated with the children elements. Branch elements for an expression tree, for example, need two groups for their children. Each branch can open a new group through this method.
closeGroup	Close the group
write	Write out the schema files using everything registered under the previous method calls.
isRegistered	Register an element so that it is not recursively called. This is needed for the branch example. For example, a branch can recursively contain another branch. The branch only needs to be registered once.

**Table 6-28:** List of methods in *EcXmlSchema*.

Text Box 6-37 illustrates a simple schema generator which is largely extracted from *EcXmlVectorVector*. The following methods are contained in this example: *EcXmlSchema*, *startTag*, *attributeName*, *type*, *endTag*, and *write*.

```

// create XML schema object
EcXmlSchema stream;

// register the start tag
stream.startTag(EcElementToken);

// register the attribute name and value for X.
stream.attributeName(EcVectorXToken);
stream.type(typeid(EcReal));

// register the attribute name and value for Y.
stream.attributeName(EcVectorYToken);
stream.type(typeid(EcReal));

// register the attribute name and value for Z.
stream.attributeName(EcVectorZToken);
stream.type(typeid(EcReal));

// close the tag
stream.endTag();

// write the schema file
stream.write(filename);

```

**Text Box 6-37:** Example of creating an XML schema. This example was largely extracted from *EcXmlVectorVector* for the schema creation of one *EcXmlVector*.

Text Box 6-38 illustrates a more complex example for creating a schema. The *EcBaseExpressionTreeContainer* class provides branching capabilities where each branch can contain an element from the container (see Text Box 6-22). Each branch in the schema contains a choice group, which in turn contain all of the registered components of the container. This schema auto-generation example also has the challenge of recursion. Each branch can contain another branch and so forth. The *isRegistered* method facilitates the registering of these recursive components so that all elements are only traversed once.

```

// if container has any components, then create schema for container.
if ( m_CreatorMap.size() > 0 )
{
    // get an iterator for the map, starting at the first pair
    EcXmlCreatorMap::const_iterator iter=m_CreatorMap.begin();

    // open schema group for this object
    // this is needed to support multiple branching (See Text Box 5-24)
    stream.openGroup();

    // loop through all the entries in the map
    while(iter!=m_CreatorMap.end())
    {
        // get element
        ExpressionType* element =
            dynamic_cast<ExpressionType*>((iter->second) ());

        element->setContainer(this);

        // write the start tag
        // CHOICE selected because each branch generally takes one option
        stream.startTag(iter->first, EcXmlSchemaElementType::CHOICE);

        // if already registered, bypass writing schema for element
        if( stream.isRegistered(iter->first, typeid(*element).name())
            == EcFalse )
        {
            // write schema for element
            element->writeSchema(stream);

            // write the end tag
            stream.endTag();
        }

        // clean element
        EcDELETE(element);

        // increment to next element
        ++iter;
    }

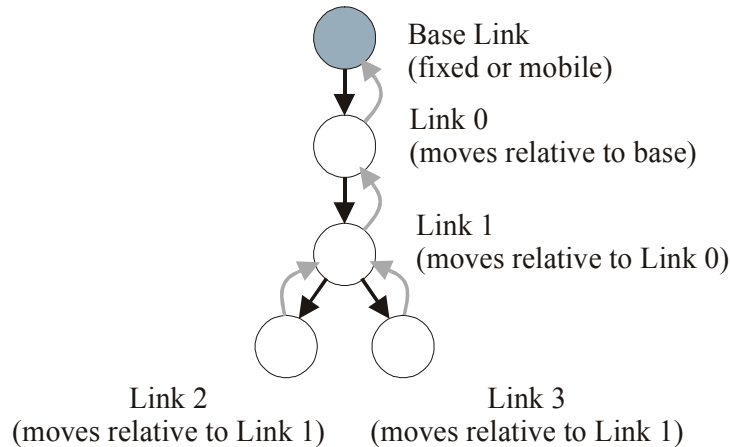
    // close schema group for this object
    stream.closeGroup();
}

```

**Text Box 6-38:** More complex example of creating an XML schema. This example is extracted from *EcBaseExpressionTreeContainer*. It calls *openGroup*, *closeGroup*, *writeStartTag*, *isRegistered*, *writeSchema*, and *writeEndTag*.

## 7 The Link

The link is the fundamental object used to construct manipulators. A manipulator constructed from links has the tree structure shown in the figure below.



**Figure 7-1:** A manipulator is composed of links in a tree structure, where each link moves relative to its parent in the tree. Each link has access to all of its children and to its parent.

The link is defined in the *EcManipulatorLink* class. Each *EcManipulatorLink* corresponds to a single joint on the manipulator, the joint's actuation, and the physical portion of the manipulator immediately outboard from the joint.

This approach to representing links is conceptually different from the traditional approach. Traditionally, robotic links are connected with joints. The links lie between the joints. In this formulation, the link conceptually contains the joint. The distal frame of one link is rigidly attached to the proximal frame of a child. This provides a more flexible approach to representing kinematic properties. It allows multiple formalisms (such as Paul or Craig's Denavit-Hartenberg notation) to be used internally to the link. It also supports the representation of new types of joints.

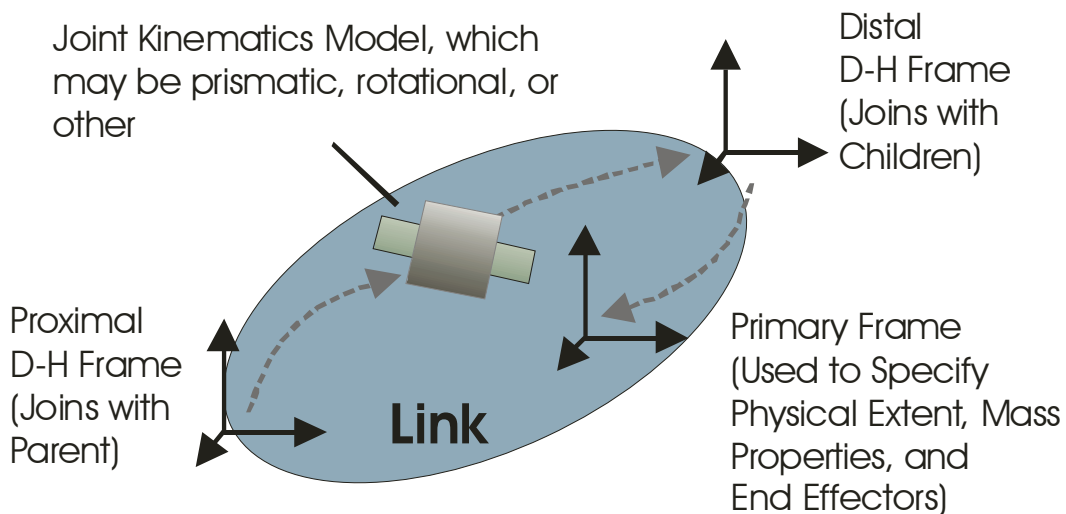
As a summary, each *EcManipulatorLink* holds the following categories of information, which will be described in this section.

- Joint kinematic description
- Mass properties
- Actuation parameters
- Physical extent
- Surface properties
- Spring and damper properties
- Child Links
- Methods for calculating link data

## 7.1 Coordinate Systems

The link model uses three reference frames. The proximal D-H frame is rigidly attached to the parent. The distal D-H frame aligns with the proximal D-H frame of the children. It moves with respect to the proximal D-H frame when the joint value of the link changes. The primary frame is rigidly attached to the distal D-H frame, but with a fixed offset. It is used to define link properties, such as mass and physical extent. It is also used as a reference when defining end effectors. These frames are illustrated in the figure below.

The proximal and distal D-H frames are so named because they correspond to the Denavit-Hartenberg (D-H) frames when this formalism is used when describing the joints. Denavit-Hartenberg notation is supported in the toolkit, but not required.



**Figure 7-2:** The link reference frames. The proximal D-H frame is rigidly attached to the parent. The distal D-H frame aligns with the proximal D-H frame of any child links. The primary frame is rigidly attached to the distal D-H frame, but with a fixed offset.

## 7.2 Link Kinematics

### 7.2.1 General Kinematics

To describe link kinematics (position, velocity, and acceleration), the toolkit supports general joint types with one degree of freedom. Thus, not only are rotational and prismatic joints supported, but also screw-type joints, motion along a rail, and any other type of motion that can be parameterized by a scalar.

To enable this generalization, a base link kinematics class, *EcLinkKinematics*, is used that allows new joint kinematic types to be easily added to the code through subclassing. Each new joint type need only specify the outer frame position/orientation, velocity, and acceleration relative to its inner frame as the function of the joint position, joint rate, and joint acceleration. The *EcLinkKinematics*

class also provides the description of the primary frame with respect to the distal frame (as shown in Figure 7-2).

The description of velocity and acceleration in the subclass are optional, however, and for efficiency only. For convenience, basic methods of velocity and acceleration calculations were implemented in the *EcLinkKinematics* class in a general form so only the transformation introduced by the new joint type is needed when adding a new joint type. The derivation of the methods for calculating the velocity and acceleration is described in the following sections.

With these implementations, all a developer needs to do to kinematically and dynamically control and simulate any one-DOF joint type is to add a function describing the location of the outer frame with respect to the inner frame as a function of joint position. This is a unique strength of the Actin™ Toolkit.

### 7.2.1.1 Approach

The general joint interface has three methods:

```
virtual const EcCoordinateSystemTransformation& calculateTransform
(
    EcReal jointValue
) const;

virtual const EcGeneralMotion& calculateVelocity
(
    EcReal jointValue,
    EcReal jointVelocity
) const;

virtual const EcGeneralAcceleration& calculateAcceleration
(
    EcReal jointValue,
    EcReal jointVelocity,
    EcReal jointAcceleration
) const;
```

**Text Box 7-1:** The methods that define the interface to the *EcLinkKinematics* class.

As shown in Text Box 7-1, *calculateTransform* is a virtual function that has to be implemented for each new joint type. This function returns the *EcCoordinateSystemTransformation* representing the distal D-H frame in the proximal D-H frame (see Figure 7-1). The other two, *calculateVelocity* and *calculateAcceleration* may be implemented for efficiency, but can work by default when *calculateTransform* is implemented.

### 7.2.2 Denavit-Hartenberg

One option for describing the link kinematics is to use the Denavit-Hartenberg notation through the *EcDenavitHartenberg* class. *EcDenavitHartenberg* is subclassed from *EcLinkKinematics* and the methods shown in Text Box 7-1 are implemented for a broad class of robotic joints that includes prismatic and rotational joints.

The toolkit supports the two most common D-H approaches, as found in robotics textbooks. The first is Paul's Denavit-Hartenberg notation [1], which uses the kinematic sequence {z-rotation, z-

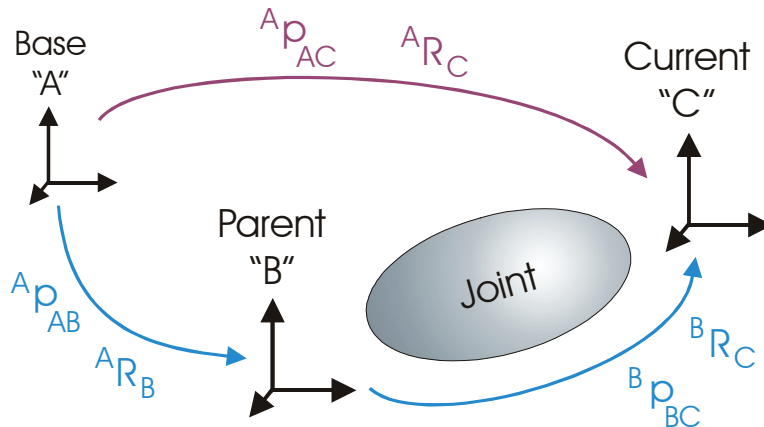
translation, x-rotation, x-translation}. The second is Craig's notation [2], using the kinematic sequence {x-rotation, x-translation, z-rotation, z-translation}. The Denavit-Hartenberg class supports general axes of rotation and translation, as well. These options, which are specified through XML, allow utmost flexibility in describing links with rotational or prismatic joints.

### 7.2.3 General Joint Velocity and Acceleration

Other types of link kinematics can be defined by specifying the methods in Text Box 7-1. To describe new link kinematics, the developer must *calculateTransform* as shown in Text Box 7-1. The other methods are optional. The developer will typically be able to calculate velocity and acceleration in closed form for faster processing, but a default approximation method is implemented that always works, albeit with more CPU time. As a reference, this section describes how the default method works when there is not specific method defined in the subclass.

#### 7.2.3.1 Mathematical Derivation

Let three frames, base, parent, and current, be labeled A, B, and C, respectively. Let quantities represented in a frame have that frame as a leading superscript, i.e.,  ${}^A p_{AC}$  is the vector from the origin of A to the origin of C represented in frame A. Orientations such as  ${}^A R_C$  represent frame C in frame A (i.e., a value represented in frame C can be represented in frame A by  ${}^A \mathbf{v} = {}^A R_C {}^C \mathbf{v}$ ). Relative position and orientation are illustrated in the figure below.



**Figure 7-3:** Illustration of the relevant frames and their representations to general velocity and acceleration calculation. A is the reference frame, B is the joint's proximal frame, and C is the joint's distal frame.

The position and orientation of frame C with respect to frame A is straightforward.

$${}^A p_{AC} = {}^A p_{AB} + {}^A R_B {}^B p_{BC} \quad (7-1)$$



$${}^A R_C = {}^A R_B {}^B R_C. \quad (7-2)$$

Using these, the linear and angular velocity can be derived. The time derivative of (7-1) and (7-2) above give (using the fact that  $\dot{R} = \mathbf{W} R$ , where  $\mathbf{W}$  is the cross-product matrix for  $\omega$ , the angular velocity):

$${}^A \dot{p}_{AC} = {}^A \dot{p}_{AB} + {}^A \omega_{AB} \times {}^A R_B {}^B p_{BC} + {}^A R_B {}^B \dot{p}_{BC} \quad (7-3)$$

and

$${}^A \omega_{AC} = {}^A \omega_{AB} + {}^A R_B {}^B \omega_{BC}. \quad (7-4)$$

Using these, the linear and angular acceleration can be derived. Taking the time derivative of equations (7-3) and (7-4) gives

$${}^A \ddot{p}_{AC} = {}^A \ddot{p}_{AB} + {}^A \dot{\omega}_{AB} \times {}^A R_B {}^B p_{BC} + {}^A \omega_{AB} \times ({}^A \omega_{AB} \times {}^A R_B {}^B p_{BC}) + 2 {}^A \omega_{AB} \times {}^A R_B {}^B \dot{p}_{BC} + {}^A R_B {}^B \ddot{p}_{BC}. \quad (7-5)$$

and

$${}^A \dot{\omega}_{AC} = {}^A \dot{\omega}_{AB} + {}^A \omega_{AB} \times {}^A R_B {}^B \omega_{BC} + {}^A R_B {}^B \dot{\omega}_{BC}. \quad (7-6)$$

Expressing all the vector quantities in a single frame gives

$$\ddot{p}_{AC} = \ddot{p}_{AB} + \dot{\omega}_{AB} \times p_{BC} + \omega_{AB} \times (\omega_{AB} \times p_{BC}) + 2 \omega_{AB} \times \dot{p}_{BC} + \ddot{p}_{BC} \quad (7-7)$$

and

$$\dot{\omega}_{AC} = \dot{\omega}_{AB} + \omega_{AB} \times \omega_{BC} + \dot{\omega}_{BC}. \quad (7-8)$$

### 7.2.3.2 Default Velocity Calculation

The function *calculateVelocity* returns  ${}^C \dot{p}_{BC}$  and  ${}^C \omega_{BC}$ , the linear and angular velocity of the current frame with respect to the parent frame, represented in the current frame. To calculate  ${}^C \dot{p}_{BC}$ , it uses finite differencing on (7-2). That is, it takes the difference of two positions at different times and divides by the time difference. The same approach can be used with angular velocity, but a small modification is needed. Because orientation is specified using quaternions, a conversion is needed to calculate the angular velocity  ${}^C \omega_{BC}$  from the time derivative of the quaternion.

Quaternions offer numerical stability while requiring only four numerical values for description. This contrasts with numerical instability for Euler angles (a small rotational rate can give unbounded Euler-angle derivatives) and the need for nine numerical values for a rotation matrix, also known as a direction cosine matrix (DCM).

There are multiple formalisms for quaternions. The Actin™ toolkit uses the one described by Shoemake [3], which is more common for robotic—but less common for aeronautical—applications. In this formalism, a quaternion representing frame  $j$  in frame  $i$  is given by

$${}^i Q_j = \begin{bmatrix} Q_0 \\ Q_1 \\ Q_2 \\ Q_3 \end{bmatrix} \quad (7-9)$$

The quaternion values are such that the quaternion can be converted to a rotation matrix through the following formula:

$${}^i R_j = \begin{bmatrix} 1-2Q_2^2-2Q_3^2 & 2Q_1Q_2-2Q_0Q_3 & 2Q_1Q_3+2Q_0Q_2 \\ 2Q_1Q_2+2Q_0Q_3 & 1-2Q_1^2-2Q_3^2 & 2Q_2Q_3-2Q_0Q_1 \\ 2Q_1Q_3-2Q_0Q_2 & 2Q_2Q_3+2Q_0Q_1 & 1-2Q_1^2-2Q_2^2 \end{bmatrix} \quad (7-10)$$

With this formalism,  $q=\{1,0,0,0\}$  corresponds to the identity matrix. (This formalism can be easily converted to the most common aeronautical form simply by moving the first entry to the last position.)

The quaternion rate can be converted to angular velocity in the moving frame through the following formula:

$$\begin{bmatrix} \omega_0 \\ \omega_1 \\ \omega_2 \end{bmatrix} = 2 \begin{bmatrix} -Q_1 & Q_0 & Q_3 & -Q_2 \\ -Q_2 & -Q_3 & Q_0 & Q_1 \\ -Q_3 & Q_2 & -Q_1 & Q_0 \end{bmatrix} \begin{bmatrix} \dot{Q}_0 \\ \dot{Q}_1 \\ \dot{Q}_2 \\ \dot{Q}_3 \end{bmatrix} \quad (7-11)$$

This equation allows finite differencing to be used on the quaternion values to estimate the quaternion time derivative and convert this estimate into an estimate of angular velocity.

### 7.2.3.3 Default Acceleration Calculation

To define the frame acceleration for a new type of joint, the last terms in equations (7-7) and (7-8) can be used, since the other terms in (7-7) and (7-8) are known.  $\ddot{p}_{BC}$  and  $\dot{\omega}_{BC}$  can be readily calculated from the time derivative of transformation from frame B to frame C. The function *calculateAcceleration* returns  ${}^c \ddot{p}_{BC}$  and  ${}^c \dot{\omega}_{BC}$  represented in the current frame, which can be obtained from time derivative of the function *calculateVelocity*. Note that *calculateVelocity* has two independent input variables *jointValue* and *jointVelocity*. Let this function be denoted as  $f(x, \dot{x})$ . Also note that the frame velocity returned from *calculateVelocity* is represented in the current frame. Let  $q(x, \dot{x}) = R(x)f(x, \dot{x})$  be the velocity represented in the parent frame. The acceleration can then be approximated by the following:

$$\begin{aligned} \frac{dq(x, \dot{x})}{dt} &= \frac{\partial q(x, \dot{x})}{\partial x} \dot{x} + \frac{\partial q(x, \dot{x})}{\partial \dot{x}} \ddot{x} \approx \frac{q(x + \Delta, \dot{x}) - q(x, \dot{x})}{\Delta} \dot{x} + \frac{q(x, \dot{x} + \Delta) - q(x, \dot{x})}{\Delta} \ddot{x} \\ &= \frac{R(x + \Delta)f(x + \Delta, \dot{x}) - R(x)f(x, \dot{x})}{\Delta} \dot{x} + \frac{R(x)f(x, \dot{x} + \Delta) - R(x)f(x, \dot{x})}{\Delta} \ddot{x} \end{aligned} \quad (7-12)$$

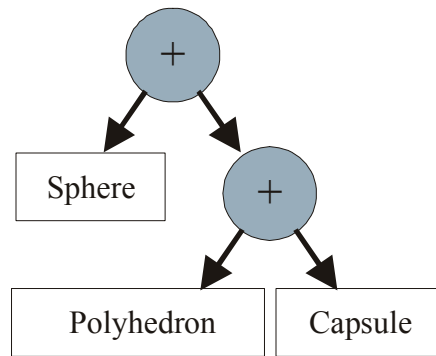
for small  $\Delta$ . The result of (7-12) must be represented in the current frame through effectively pre-multiplying by  $R^{-1}(x)$ .

### 7.3 Mass Properties

The link's mass properties are described within the link. This data includes the 10 parameters needed for dynamics calculation: the scalar mass, the vector first moment of inertia, and the 3×3 symmetric second moment of inertia. The values are represented in the primary frame, not at the center of mass. Mass properties are represented efficiently in the code in a special class for rigid-body mass properties. This class includes methods for adding mass properties and transforming mass properties to a different reference frame.

### 7.4 Physical Extent

The physical extent of the link is represented as a combination of geometric primitives through a tree-based approach. Geometric primitives will be stored in the tree structure as shown in Figure 7-4.



**Figure 7-4:** A tree structure representing a manipulator link's physical extent. Point-polygon, sphere, and tetrahedron are geometric primitive shapes.

#### 7.4.1 Composable Tree Structure

The tree structure for representing geometric extents is composable as runtime. A physical description of each link can be read or written as XML. A shape container class has been developed to support this, which can be easily modified (either directly or through a dll) to support additional geometric shape primitives or processing nodes. The shape container class (*EcShapeContainer*) inherits from *EcBaseExpressionTreeContainer*<*EcShape*>.

The tree structure can hold operations (both binary and unary) as intermediate nodes and shape primitives as leaf nodes. Currently there are two binary operations that are supported, union and intersection. The framework can also support unary operations.

## 7.4.2 Shape Primitives

Shape primitives are used to represent physical extents in the toolkit. Primitives can be used to represent bounding volumes or used to construct the representation of the physical extent itself.

## 7.4.3 Fundamental Geometrical Shapes

To build the primitives, a few fundamental geometric components other than lines and points have been implemented within the framework. These are described below.

### 7.4.3.1 Triangle

Triangles are frequently used for shape representation since fast implementations of distance and intersection tests exist for them. All polyhedrons can be reduced to a collection of triangles. Through tessellation, other shape primitives can always be reduced to a polyhedron composed of a collection of triangles. The code uses a fast formulation, as developed in [4].

A triangle is represented using a vector origin and two vector edges as follows:

Member Data	Class/Type	Description	Restrictions
m_Origin	<i>EcXmlVector</i>	The origin of the triangle: ( <b>B</b> )	None.
m_Edge0	<i>EcXmlVector</i>	The first edge vector describing the triangle: ( <b>E<sub>0</sub></b> )	None.
m_Edge1	<i>EcXmlVector</i>	The second edge vector describing the triangle: ( <b>E<sub>1</sub></b> )	None.

**Table 7-1:** *EcTriangle* data structure.

### 7.4.3.2 Rectangle

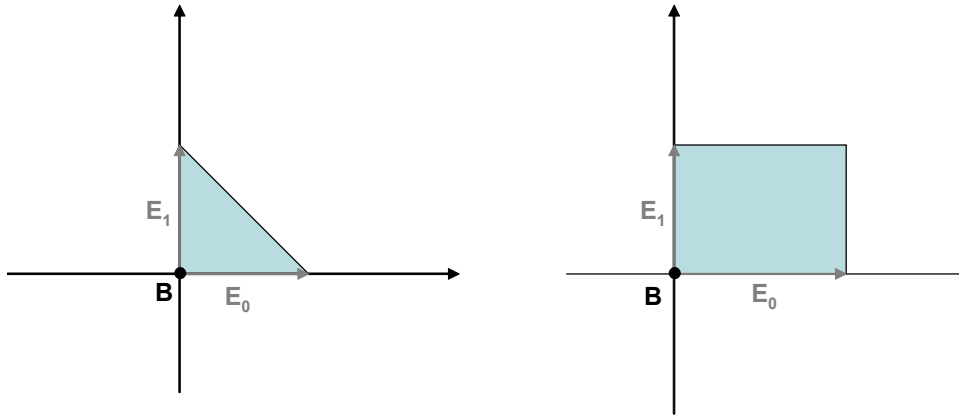
A rectangle is another fundamental shape used to construct the geometrical primitives. It is also available for general use in the toolkit. Figure 7-5 shows a rectangle with the space partitions that need to be considered for distance calculations.

The representation of a rectangle in the framework is similar to that for a triangle, as shown in the table that follows

Member Data	Class/Type	Description	Restrictions
m_Origin	<i>EcXmlVector</i>	The origin of the rectangle	None.
m_Edge0	<i>EcXmlVector</i>	The first edge vector describing the rectangle.	None.
m_Edge1	<i>EcXmlVector</i>	The second edge vector describing the rectangle.	None.

**Table 7-2:** *EcRectangle* data structure.

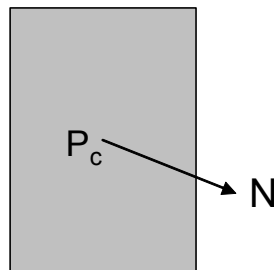
The representation of a triangle and a rectangle are described graphically below.



**Figure 7-5:** Representation of a Triangle (*EcTriangle*) and a Rectangle (*EcRectangle*).

### 7.4.3.3 Plane

A plane is a construct used extensively in the framework. A plane is defined by a normal vector  $N$  and a base point  $P_c$ . By convention, planes return a signed distance with the normal vector pointing out of the front of the plane.



**Figure 7-6:** A plane defined by a base point  $P_c$  and a normal vector  $N$ .

The data structure for describing a plane is as follows:

Member Data	Class/Type	Description	Restrictions
m_BasePoint	<i>EcXmlVector</i>	The base point of the plane.	None.
m_NormalVector	<i>EcXmlVector</i>	The normal vector of the plane.	None.

**Table 7-3:** *EcPlane* data structure.

#### 7.4.4 Primary Shapes

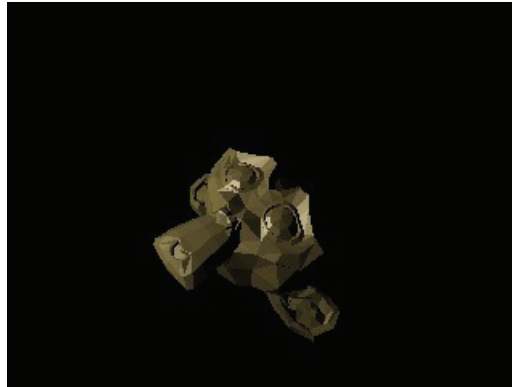
All shape primitives are derived from the *EcShape* virtual base class. Objects derived from *EcShape* can compute distances and intersections to other objects derived from *EcShape*. The architecture accommodates both intersection and distance methods. Distance can be used to calculate intersection, but intersection is generally less costly to calculate directly.

Member Data	Class/Type	Description	Restrictions
m_pPhysicalExtent	<i>EcBasePhysicalExtent</i>	Holds a pointer to a physical extent representation of the shape. All shapes can be represented as a physical extent (i.e. a polyhedron that is formed from a collection of polygons)	None.
m_pShapeContainer	<i>EcXmlVariableCompoundType</i>	A pointer to a shape container. A shape container knows how to read and write an extent expression tree.	None.

**Table 7-4:** *EcShape* data structure.

Shapes in the toolkit are described in the following subsections.

#### 7.4.4.1 Polyhedron



**Figure 7-7:** A polyhedron.

Polyhedrons are represented as a collection of polygons. Methods are available to compute the distance between polyhedrons and other shapes (including other polyhedrons). A polyhedron is represented in code using an *EcBasePhysicalExtent* class. All other shapes can always be turned into an *EcBasePhysicalExtent* by calling the *physicalExtent* method in the *EcShape* base class. The table below describes the most frequently used methods.

Method	Description
points setPoints	Get/set the collection of points for this polyhedron.
polygons setPolygons	Get/set the polygons for this polyhedron. The polygon arrays are lists of indices into the point array. This is done for space savings so that each point (an <i>EcVector</i> ) needs only to be stored once.
normals setNormals	Get/set a collection of normal vectors, one for each polygon that describes the surface of the polygon.
convexHull	Creates the polyhedron as a convex hull of the physical extent passed in. This is useful for converting a non-convex polyhedron into a convex polyhedron.

**Table 7-5:** Polyhedron data structure.

```

//-----
// Make a very simple one polygon physical extent
//-----
EcPhysicalExtent polyExt ;
// make a point set
EcPointCollection polyPoints;
polyPoints.pushBack(EcVector(3.0,0.0,0.0));
polyPoints.pushBack(EcVector(5.0,0.0,0.0));
polyPoints.pushBack(EcVector(0.0,3.0,0.0));

// add the point collection to the physical extent
polyExt.setPoints(polyPoints);

// create a polygon collection (just one for this example)
EcPolygonCollection polygons;
EcPolygon polygon;
polygon.addPointIndex(0);
polygon.addPointIndex(1);
polygon.addPointIndex(2);

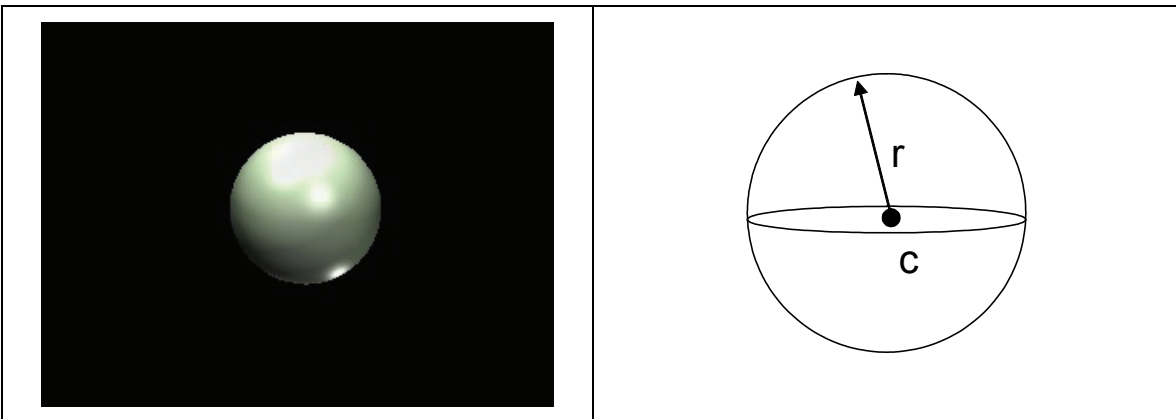
// set a surface key. This is used to find the surface
// properties of the material
polygon.setSurfaceKey("metal");

// add the polygon to the polygon collection
polygons.pushBack(polygon);
// add the polygon collection to the physical extent
polyExt.setPolygons(polygons);

```

**Text Box 7-2:** Defining a physical extent.

#### 7.4.4.2 Sphere



**Figure 7-8:** A sphere.

A sphere is represented as a center and a radius. Distance and intersection calculations for spheres are straightforward. Distance can be calculated by computing the distance from a point to an object



and subtracting the radius. This is very efficient for calculating distances, and it can also be used with some shapes to rapidly compute a penetration distance needed for force feedback.

Method	Description
center setCenter	Gets/sets the center of the sphere. .
radius setRadius	Gets/sets the radius of the sphere

**Table 7-6:** *EcSphere* data structure.

```

// -----
// Create a sphere shape
// -----

EcSphere sphere;

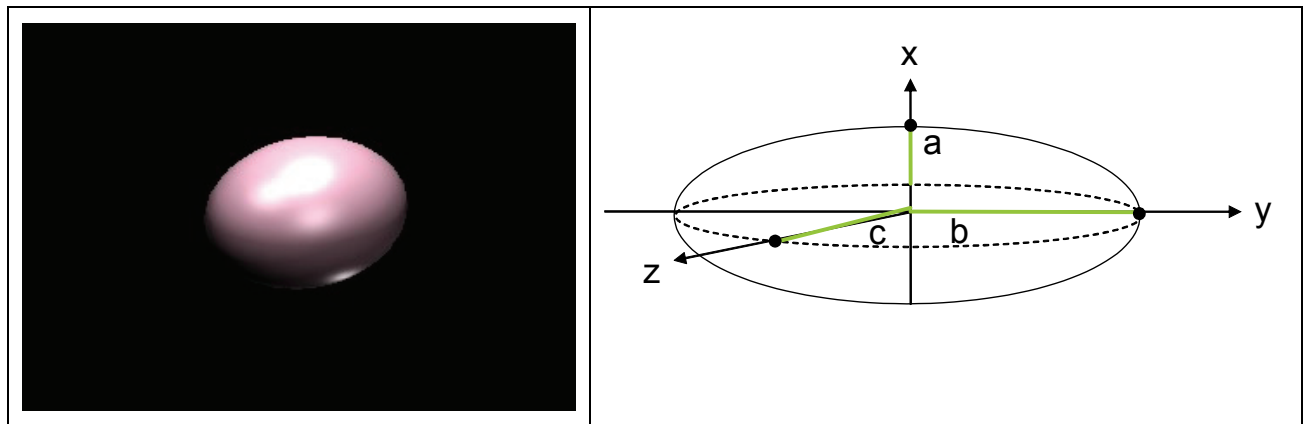
// set the center point in the primary frame of the link
sphere.setCenter(EcVector(0,0,0));

// set the radius
sphere.setRadius(2.0);

```

**Text Box 7-3:** Defining a sphere.

### 7.4.4.3 Ellipsoid



**Figure 7-9:** An ellipsoid.

```

// -----
// Create an ellipsoid shape
// -----
EcEllipsoid ellipsoid;

// set the <A,B,C> values. This example creates
// a sphere represented as an ellipsoid
ellipsoid.setA(1.0);
ellipsoid.setB(1.0);
ellipsoid.setC(1.0);

// set the ellipsoid at the origin of the primary frame
EcCoordinateSystemTransformation xf =
    EcCoordinateSystemTransformation::nullObject();
ellipsoid.setXform(xf);

```

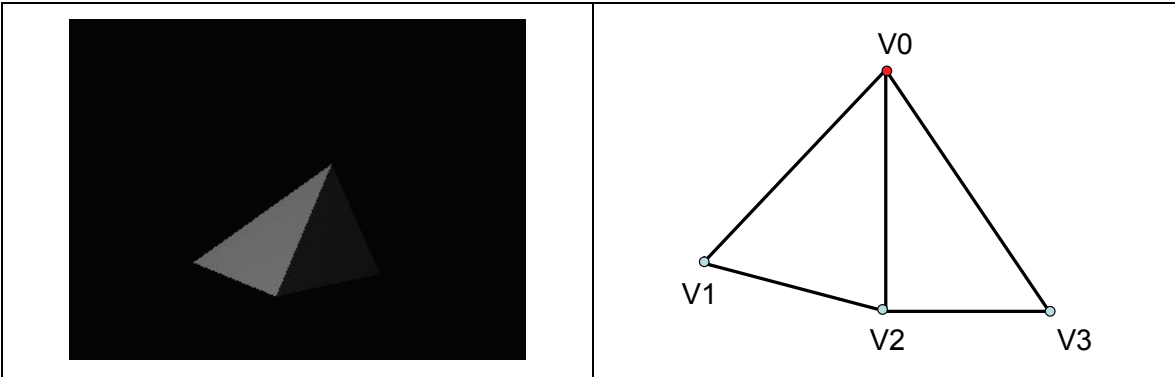
**Text Box 7-4:** Defining an ellipsoid.

To improve processing time, the data structure describing the ellipsoid contains two representations. The computation of the distance between an ellipsoid and another shape is, however, difficult in general. The implementation of the ellipsoid is given in the table below.

Member Data	Class/Type	Description	Restrictions
m_A	<i>EcReal</i>	A of: $\frac{x^2}{a^2} + \frac{y^2}{b^2} + \frac{z^2}{c^2} = 1$	None.
m_B	<i>EcReal</i>	B of: $\frac{x^2}{a^2} + \frac{y^2}{b^2} + \frac{z^2}{c^2} = 1$	None.
m_C	<i>EcReal</i>	C of: $\frac{x^2}{a^2} + \frac{y^2}{b^2} + \frac{z^2}{c^2} = 1$	None.
m_CoordXform	<i>EcCoordinateSystemTransformation</i>	The orientation of the ellipsoid.	None.
m_AMatrix	<i>EcSpatialMatrix</i>	A of: $(X - C)^T A (X - C) = 1$	None.
m_InvAMatrix	<i>EcSpatialMatrix</i>	Inv(A) of: $(X - C)^T A (X - C) = 1$	None.

**Table 7-7:** *EcEllipsoid* data structure.

### 7.4.4.4 Tetrahedron



**Figure 7-10:** A tetrahedron.

A tetrahedron is a three-dimensional simplex. In most instances, distances between tetrahedrons and other shapes can be computed by separating the tetrahedron into its four constituent triangles (*EcTriangle*). The tetrahedron implementation is shown through the following table.

Member Data	Class/Type	Description	Restrictions
m_v0	<i>EcXmlVector</i>	Base vertex of the tetrahedron.	None.
m_Edge1	<i>EcXmlVector</i>	Edge 1 of the tetrahedron.	None.
m_Edge2	<i>EcXmlVector</i>	Edge 2 of the tetrahedron.	None.
m_Edge3	<i>EcXmlVector</i>	Edge 3 of the tetrahedron.	None.

**Table 7-8:** *EcTetrahedron* data structure.

```
// -----
// Create a tetrahedron shape
// -----

// -----
// A tetrahedron can be constructed from four points in
// space, as described in the figure above. These are defined in the
// primary frame.
// -----

EcVector v0 = EcVector(3,3,0);
EcVector v1 = EcVector(1,0,0);
EcVector v2 = EcVector(5,0,0);
EcVector v3 = EcVector(2,0,-2);
EcTetrahedron tet(v0,v1,v2,v3);
```

**Text Box 7-5:** Defining a tetrahedron.

### 7.4.4.5 Oriented Box

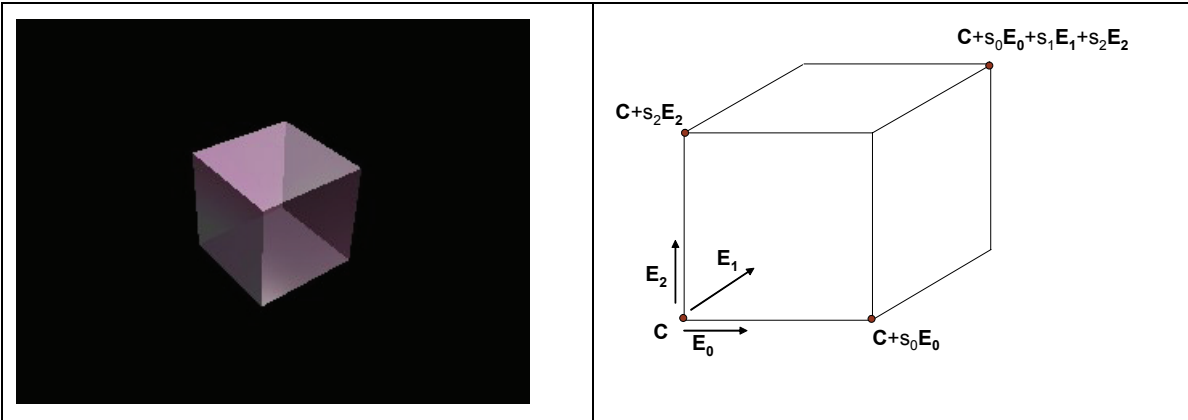


Figure 7-11: An oriented bounding box.

Implementation of the oriented bounding box is described through the following table.

Member Data	Class/Type	Description	Restrictions
m_Extents	<i>EcXmlVector</i>	The extents of the bounding box $(s_0, s_1, s_2)$ . The length of a side is $2s_i$ where $i = 0, 1, 2$ .	None.
m_Axes	<i>EcXmlVectorVector</i>	The axes of the bounding box	None.
m_Center	<i>EcXmlVector</i>	The center of the box.	None.

Table 7-9: *EcBox* data structure.

```

//-----
// create a unit cube box
// -----
EcBoundingBox bbox;

// set the center at the origin
EcVector center = EcVector(0,0,0);
bbox.setCenter(center);

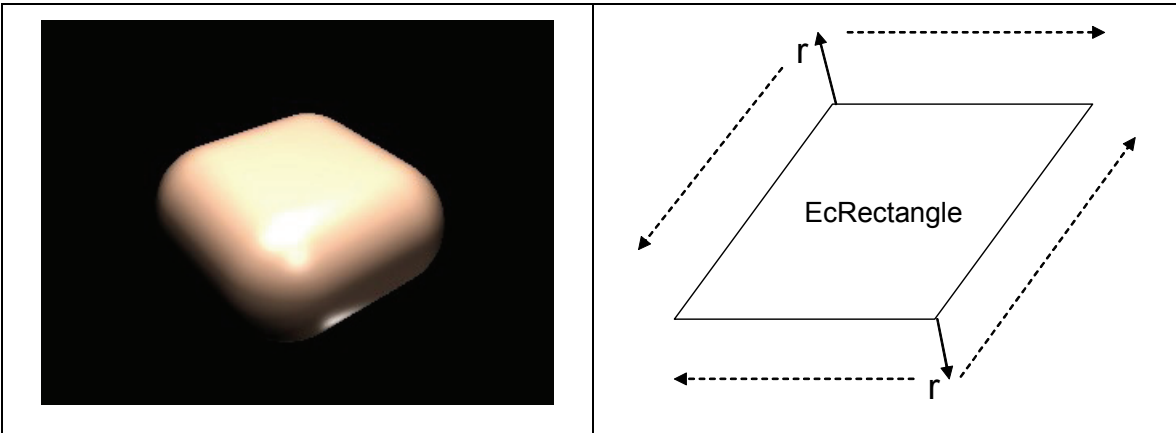
// set the axes (these should be orthogonal unit vectors)
EcXmlVectorVector axes;
EcVector axis1 = EcVector(1,0,0);
EcVector axis2 = EcVector(0,1,0);
EcVector axis3 = EcVector(0,0,1);
bbox.setAxes(axes);

// set the extents
EcVector extents = EcVector(1,1,1);
bbox.setExtents(extents);

```

**Text Box 7-6:** Defining a bounding box.

**7.4.4.6 Lozenge**



**Figure 7-12:** A lozenge.

A lozenge can be thought of as the square analog to a capsule, where instead of defining the surface as a fixed radial distance from a line segment, a lozenge is described by a fixed radial distance from the surface of a square. This construct allows for efficient distance and collision checks. It can be used in lieu of the more computationally expensive ellipsoid primitive. Its implementation is described through the table below.

Member Data	Class/Type	Description	Restrictions
m_Rectangle	<i>EcRectangle</i>	The rectangular center of the lozenge.	None.
m_Radius	<i>EcXmlReal</i>	The radial distance from the rectangle to a point on the surface of the lozenge.	None.

**Table 7-10:** *EcLozenge* data structure.

```

//-----
// create a lozenge
//-----
EcLozenge lozenge;

// set the rectangle
EcRectangle rect;
// a square in the xy plane
rect.setEdge0(EcVector(1,0,0));
rect.setEdge1(EcVector(0,1,0));

lozenge.setRectangle(rect);

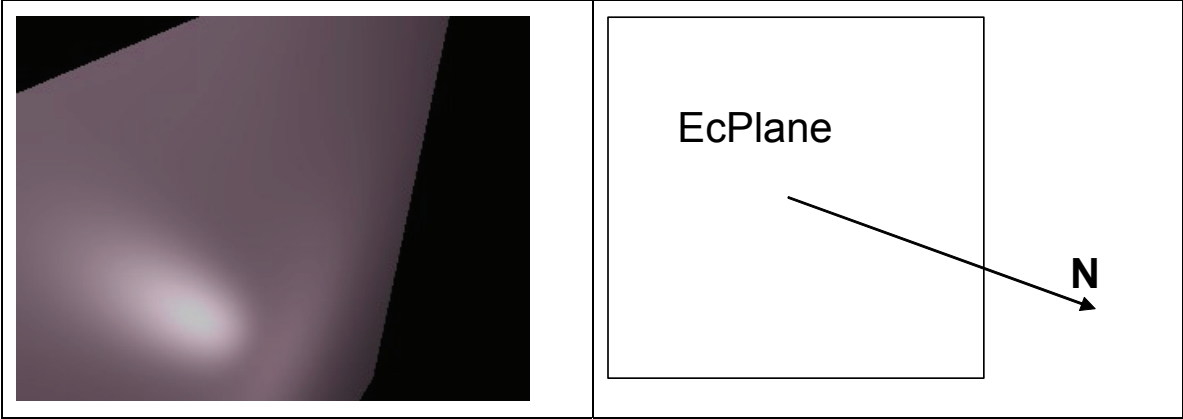
// set the the corner at the origin
rect.setOrigin(EcVector::zeroVector());

// set the radius
lozenge.setRadius(2.5);

```

**Text Box 7-7:** Defining a lozenge.

**7.4.4.7 Half Space**



**Figure 7-13:** An illustration of a half space.

A halfspace bisects all of space into two halves. A halfspace is represented by a bisecting plane. The normal vector of the plane points out of the halfspace.

Member Data	Class/Type	Description	Restrictions
m_Plane	<i>EcPlane</i>	The separating plane of the halfspace. The normal of the plane points out of the halfspace.	None.

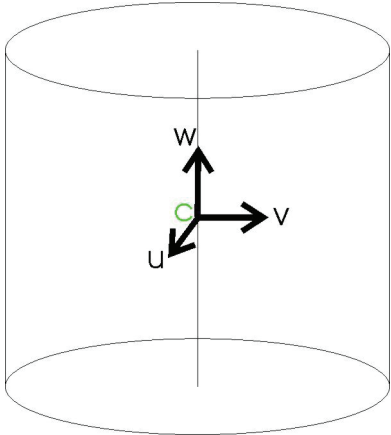
**Table 7-11:** *EcHalfspace* data structure.

```
//-----  
// Create a Halfspace  
//-----  
  
EcHalfSpace halfSpace;  
  
// set the normal to point in the positive x direction  
halfSpace.setNormal(EcVector::xVector());  
  
// set the base point at the origin. All points left of the  
// y-z plane are in the halfspace  
halfSpace.setBasePoint(EcVector::zeroVector());
```

**Text Box 7-8:** Defining a halfspace.

### 7.4.4.8 Cylinder

A cylinder can be described by a line segment  $\mathbf{k}$  and a radius  $r$ . Let  $\mathbf{w} = \mathbf{k}/\|\mathbf{k}\|$  and  $\{\mathbf{u}, \mathbf{v}, \mathbf{w}\}$  be a coordinate frame at the cylinder center  $\mathbf{c}$  (refer to the figure below).



**Figure 7-14:** Coordinate frame  $\{\mathbf{u}, \mathbf{v}, \mathbf{w}\}$  attached at center of the cylinder.

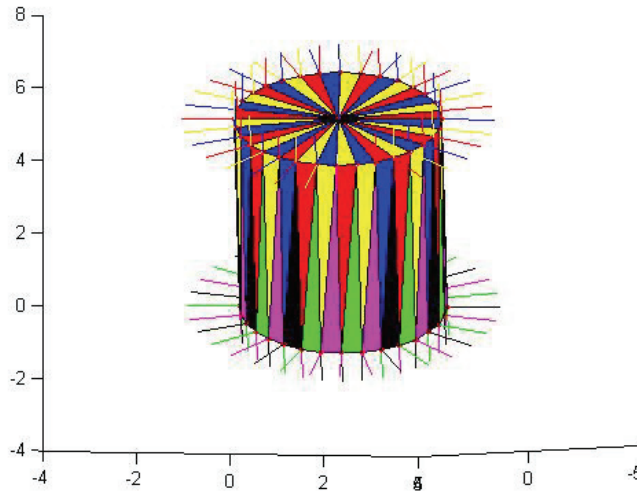
The points on the cylinder surface can be conveniently parameterized as [3b]:

$$\mathbf{x}(\theta, t) = \mathbf{c} + r \cos \theta \mathbf{u} + r \sin \theta \mathbf{v} + t \mathbf{w} \tag{35}$$

Where  $\theta \in [0, 2\pi)$ ,  $|t| \leq \frac{\|\mathbf{k}\|}{2}$ . A *support point* is the point with maximum distance in a given direction, as required by the GJK algorithm. For a cylinder, the support point is the point on this cylinder that yields the maximum dot product with a given vector  $\mathbf{d}$ . It can be shown that this point is

$$\mathbf{s}(\mathbf{d}) = \mathbf{c} + r \frac{\mathbf{d} - (\mathbf{d} \cdot \mathbf{w})}{\sqrt{\|\mathbf{d}\|^2 - (\mathbf{d} \cdot \mathbf{w})^2}} \mathbf{w} + \text{sign}(\mathbf{d} \cdot \mathbf{w}) \frac{\|\mathbf{k}\|}{2} \mathbf{w} \quad (36)$$

The triangle-meshed physical extent is shown in the figure below. The normal for each vertex is also plotted. The ability to perform this triangularization was added to the code base.



**Figure 7-15:** Triangle meshed physical extent for the cylinder. Normal vectors are also shown.

#### 7.4.4.9 Cone

A conical frustum shape can be described by a line segment  $\mathbf{k}$  and two radii,  $r_0$  and  $r_1$ . When  $r_0 = r_1$ , this is reduced to the cylindrical case. The support point can only appear on the edge of the two end discs. By checking the maximum projection values of the two end discs to the given vector  $\mathbf{d}$ :

$$\text{val}_0 = \mathbf{d} \cdot \mathbf{c} + r_0 \sqrt{\|\mathbf{d}\|^2 - (\mathbf{d} \cdot \mathbf{w})^2} - (\mathbf{d} \cdot \mathbf{w}) \frac{\|\mathbf{k}\|}{2} \quad (37)$$

and

$$\text{val}_1 = \mathbf{d} \cdot \mathbf{c} + r_1 \sqrt{\|\mathbf{d}\|^2 - (\mathbf{d} \cdot \mathbf{w})^2} + (\mathbf{d} \cdot \mathbf{w}) \frac{\|\mathbf{k}\|}{2} \quad (38)$$



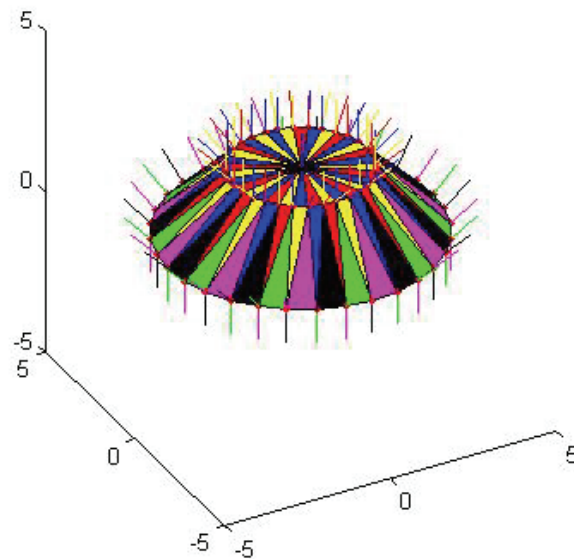
One can conclude that, if  $val_0 > val_1$ ,

$$\mathbf{s}(\mathbf{d}) = \mathbf{c} + r_0 \frac{\mathbf{d} - (\mathbf{d} \cdot \mathbf{w})}{\sqrt{\|\mathbf{d}\|^2 - (\mathbf{d} \cdot \mathbf{w})^2}} \mathbf{w} - \frac{\|\mathbf{k}\|}{2} \mathbf{w}. \quad (39)$$

Otherwise,

$$\mathbf{s}(\mathbf{d}) = \mathbf{c} + r_1 \frac{\mathbf{d} - (\mathbf{d} \cdot \mathbf{w})}{\sqrt{\|\mathbf{d}\|^2 - (\mathbf{d} \cdot \mathbf{w})^2}} \mathbf{w} + \frac{\|\mathbf{k}\|}{2} \mathbf{w}. \quad (40)$$

The triangle-meshed physical extent is shown in the figure below. The normal for each vertex is also plotted. The ability to perform this triangularization was added to the code base.



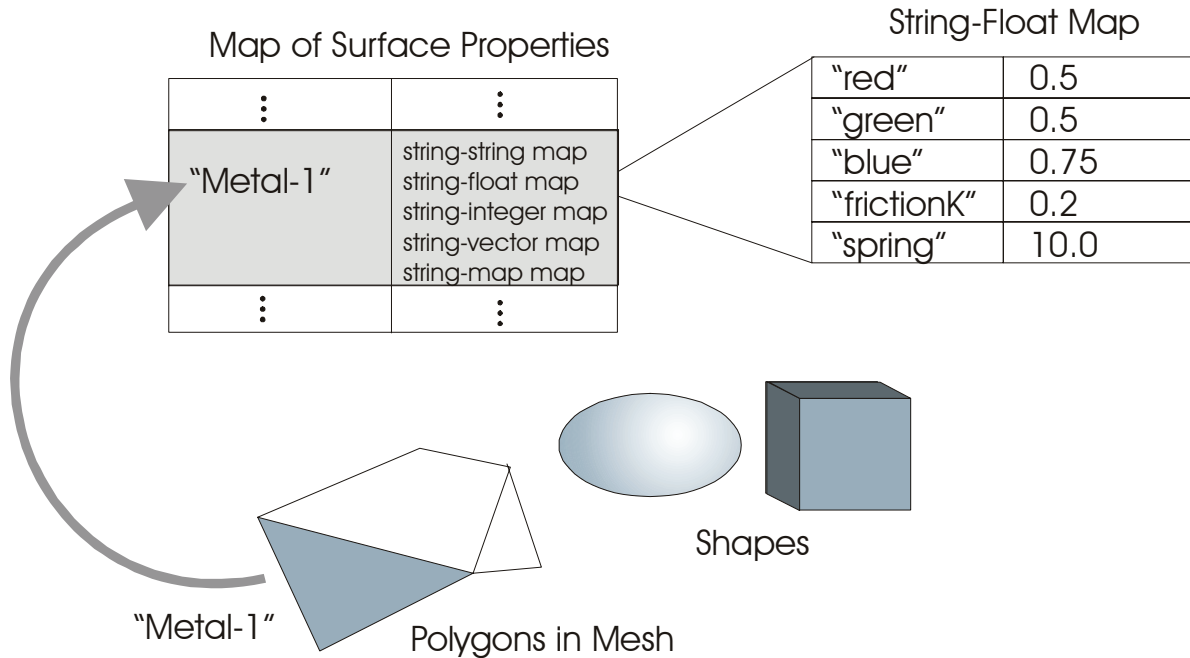
**Figure 7-16:** Triangle meshed physical extent for the cone shape. Normal vectors (as are needed for rendering and spatial reasoning) are also shown.

## 7.5 Surface Properties

The toolkit allows the specification of detailed surface properties. The specification of surface properties allows improved rendering, simulation, and intelligent reasoning. To represent surface properties, the toolkit uses an approach influenced by the Synthetic Environment Data Representation and Interchange Specification (SEDRIS) [5] for specifying environmental properties.

The surface-property specification works as follows: Every link maintains a map of surface properties referenced by a string token. Each surface property in turn holds one string-string map, one string-floating-point map, and one string-integer map. These allow the user to specify arbitrary

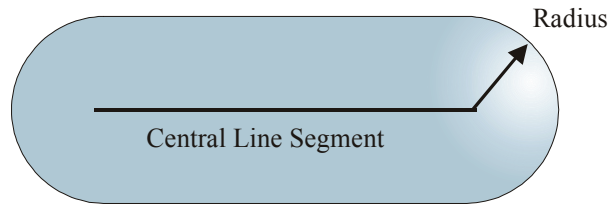
strings, floating-point values, and integers in an intuitive general way. This is illustrated in the figure below.



**Figure 7-17:** Surface property description. Each link holds a string-to-surface-property map. Polygons specify their surface property using a string reference into the map. Each surface property maintains three separate maps: string-string, string-float, string-integer, string-float-vector, string-integer-vector, and string-map maps. These maps are configurable by the user through XML. They can represent virtually any properties, including hardness, flexibility, fragility, and so forth. The example shown here is color.

## 7.6 Bounding Volumes

A link can have several bounding volumes for fast collision detection and distance queries. These will be discussed in detail in section 12- Collision Avoidance. By default, every link maintains a capsule bounding volume, defined as all points within a specified radius of a three-dimensional line segment. This is illustrated in the figure below. A capsule bounding volume is defined for every link on every manipulator. It can be specified by the user through the XML language. The capsule is resized when the points describing the polygon mesh are loaded. This way the bounding volume is always valid. Capsule bounding volumes offer easy transformation and easy intersection testing, and can also be used to bound environmental obstacles for obstacle-avoidance control.



**Figure 7-18:** Bounding volumes are described using capsules, which are defined as all points within some radius of a three-dimensional line segment. The line segment can be positioned and oriented in any way. Every link on every arm has a capsule bounding volume. Capsules are also used to bound obstacles in the environment.

## 7.7 Actuators

The actuator parameters that are described as part of the link include the motor friction, motor inertia, gear ratio, and joint limits. In addition, stopper dynamics are represented using a repulsive force or torque that is proportional to the incursion within a specified zone of the hard stop.

Member Data	Class/Type	Description	Restrictions
m_Inertia	<i>EcXmlReal</i>	Actuator inertia	None.
m_FrictionCoefficient	<i>EcXmlReal</i>	Actuator viscous friction coefficient	None.
m_GearRatio	<i>EcXmlReal</i>	Actuator gear ratio	None.
m_LowerLimit	<i>EcXmlReal</i>	Lower joint limit	None.
m_UpperLimit	<i>EcXmlReal</i>	Upper joint limit	None.
m_StopperZone	<i>EcXmlReal</i>	The range within which a force from the joint limit has a physical effect	None.
m_MinTorque	<i>EcXmlReal</i>	The minimum torque that can be applied to the joint by the actuator.	None.
m_MaxTorque	<i>EcXmlReal</i>	The maximum torque that can be applied to the joint by the actuator.	None.
m_StopperSpring Coefficient	<i>EcXmlReal</i>	The coefficient of the best spring approximation to the joint limit.	None.
m_StopperDamping	<i>EcXmlReal</i>	The coefficient of the best damper approximation to	None.

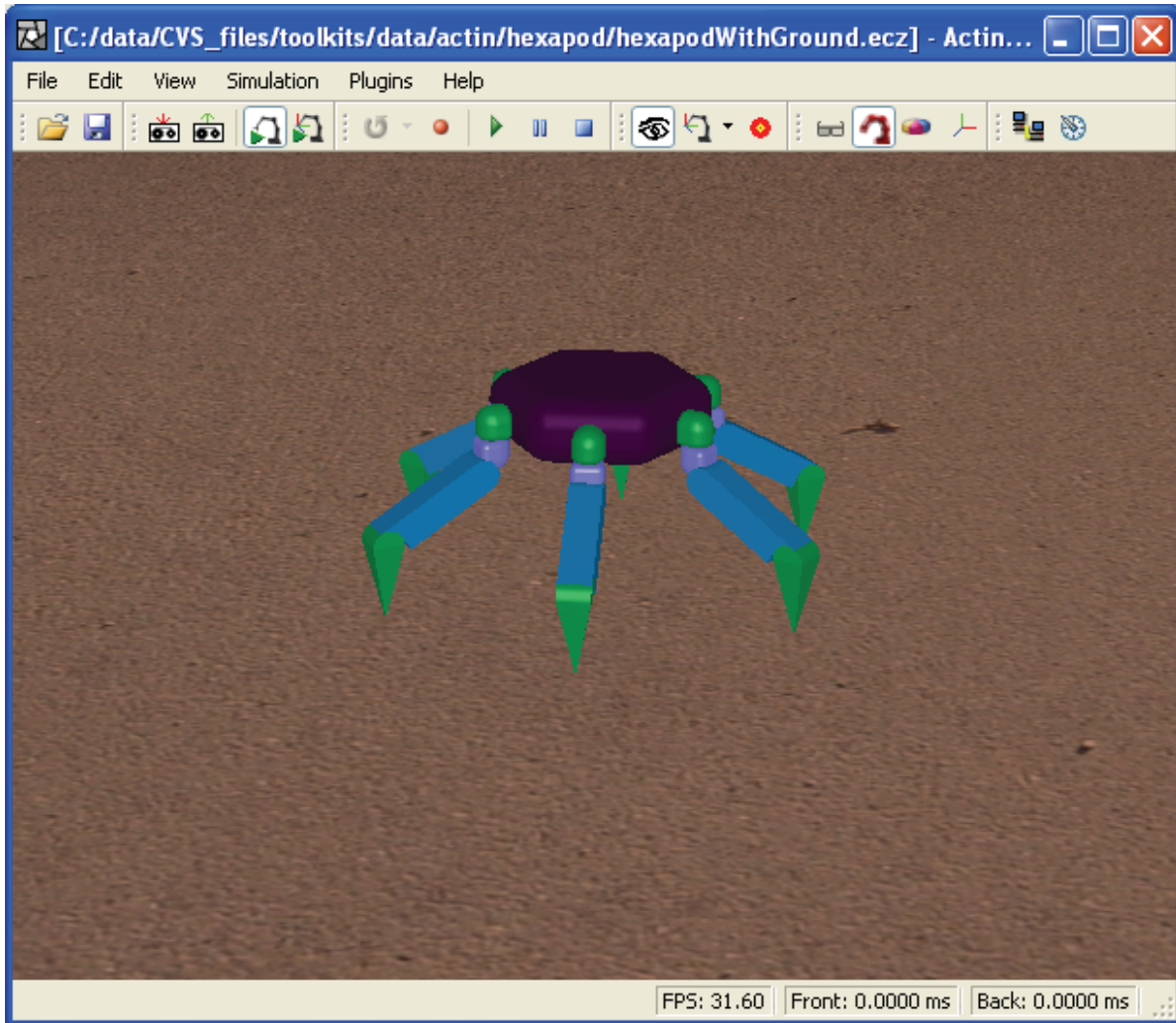
Coefficient		the joint limit.	
-------------	--	------------------	--

**Table 7-12 Actuator Description.**

## 7.8 Actuator Database Interface

For use in robot design, Actin includes an actuator-related plugin to assist robot designers when trying to size motors and gearheads for a given manipulator. For each joint actuator, the user can select a motor and chain gearheads in series to be used in the actuator from a database through GUI. This allows the designer to quickly and easily change the actuator components of any actuator and rerun the simulation to see if the new components meet the requirements.

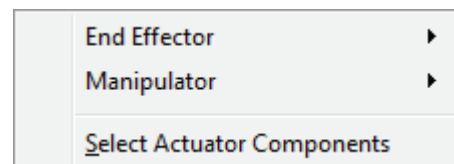
In the backend of the plugin is a relational database that contains the data of actuator components. For each component, the data include the manufacturer, the part number (or order number), the model, and a set of specifications. The presence of the database is transparent to the user since he never has to interact with it directly, only through GUI. Let's look at how to use the actuator plugin. First, the actuator plugin must be loaded. The plugin file is 'ecActuatorPlugin.dll.' Once the plugin is loaded, the Design menu will be added. Figure 7-19 shows ActinViewer with the actuator plugin loaded and with a hexapod model.

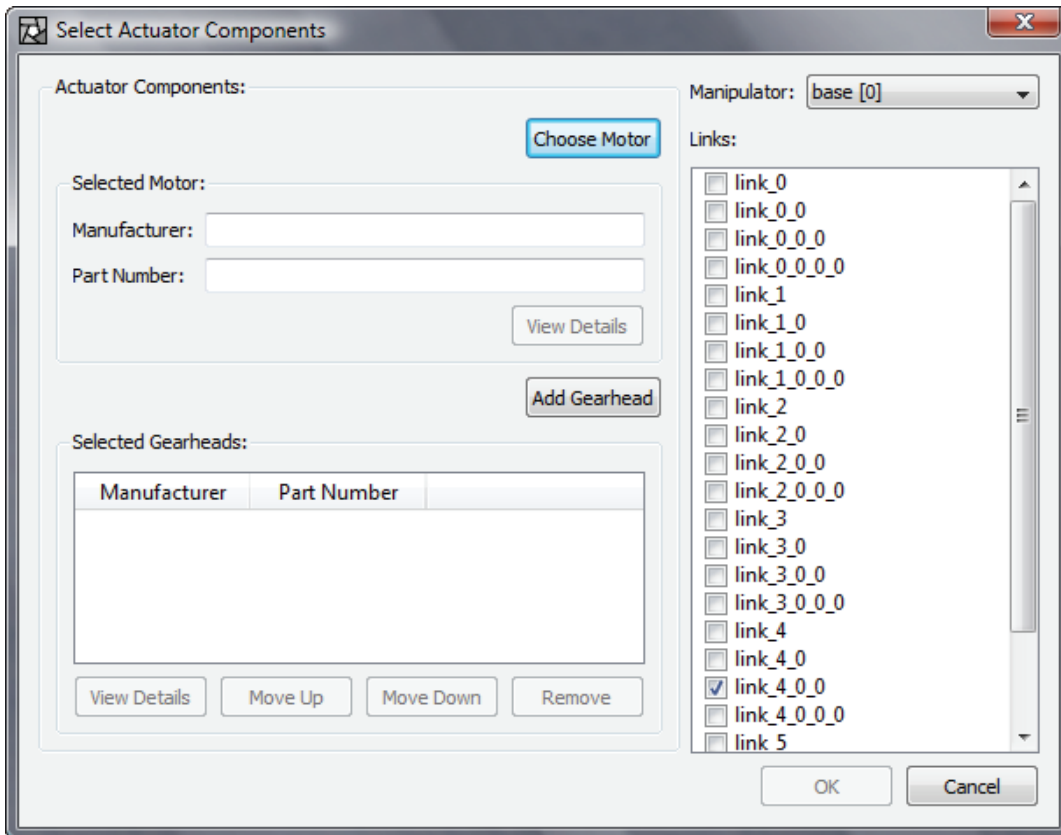


**Figure 7-19:** Actuator plugin allows the designer to quickly change components of joint actuators from by selecting them from database.

### 7.8.1 *Selecting Actuator Components*

To select components of a joint actuators, the user can right-click on the desired link to bring up the context menu shown on the right. The user then selects “Select Actuator Components,” which will bring up the following dialog.



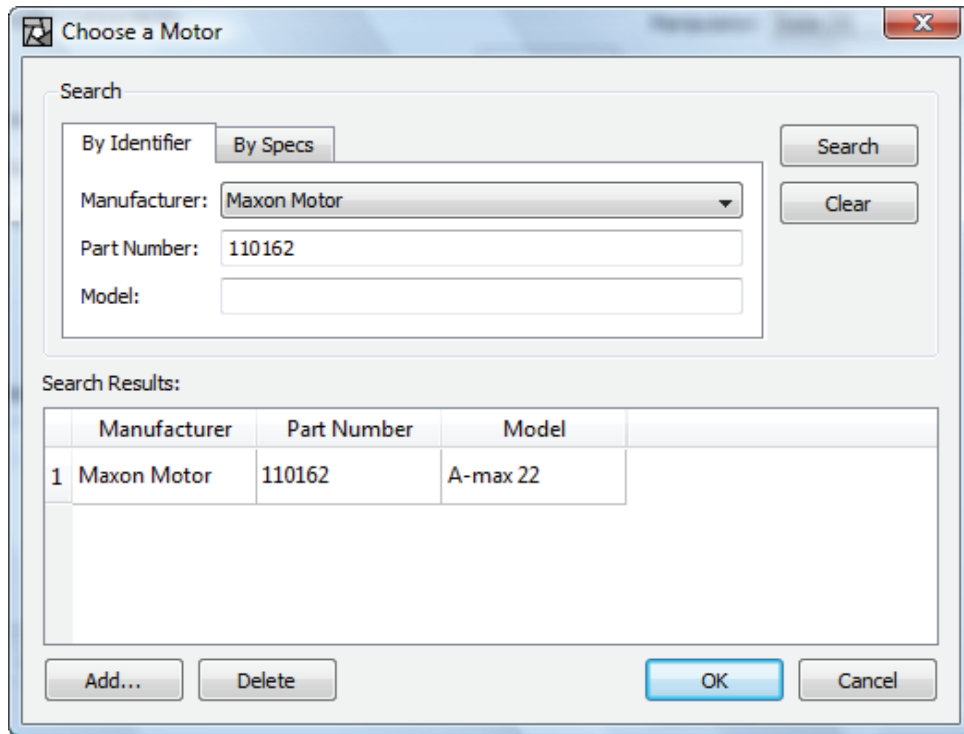


**Figure 7-20:** Dialog for selecting actuator components. Note that the desired link (link\_4\_0\_0 in this case) is selected by default.

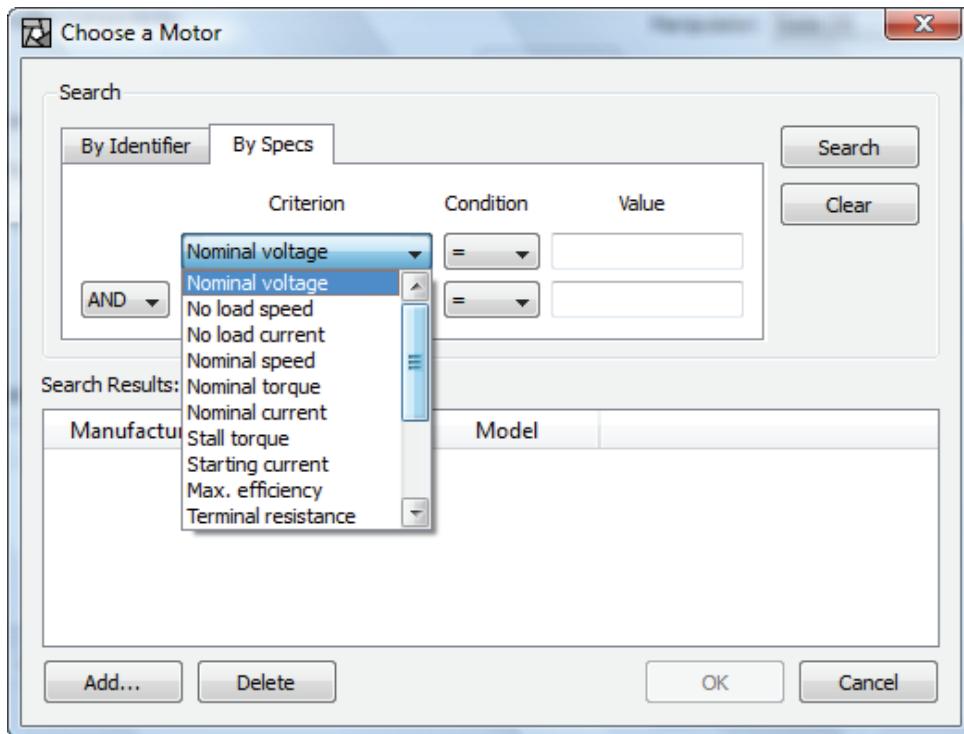
The “Select Actuator Components” in Figure 7-20 lets the user choose the motor and a series of gearheads as components of a joint actuator. To choose a motor, click on the “Choose Motor” button and the “Choose Motor” dialog in Figure 7-21 will be brought up. With this dialog, the user can choose a motor by searching the database. Motors can be searched either by identification or by specifications. If the motor that the user is looking for does not exist in the database, he can have an option to add it to the database. Adding items to the database along with editing and deleting items will be discussed later in Section 7.8.2 Managing Database.

Still in the “Choose Motor” dialog, assume that the desired motor is in the database. If the search for the desired motor yields one result, the OK button will be enabled (see Figure 7-21), allowing the user to select that motor. If two or more motors are found by the search, the OK button will not be enabled until the use clicks on motor to signify that it is the one he is looking for. Clicking the OK button will select the desired motor and indicate that it will be added to the joint actuator.

For gearheads, the user has an option to chain gearheads in series. To add a gearhead, click the Add Gearhead button in Figure 7-20. This will bring up a dialog almost identical to the “Choose Motor” dialog. But instead of choosing a motor, the user will choose a gearhead by searching in the same manner as choosing a motor.

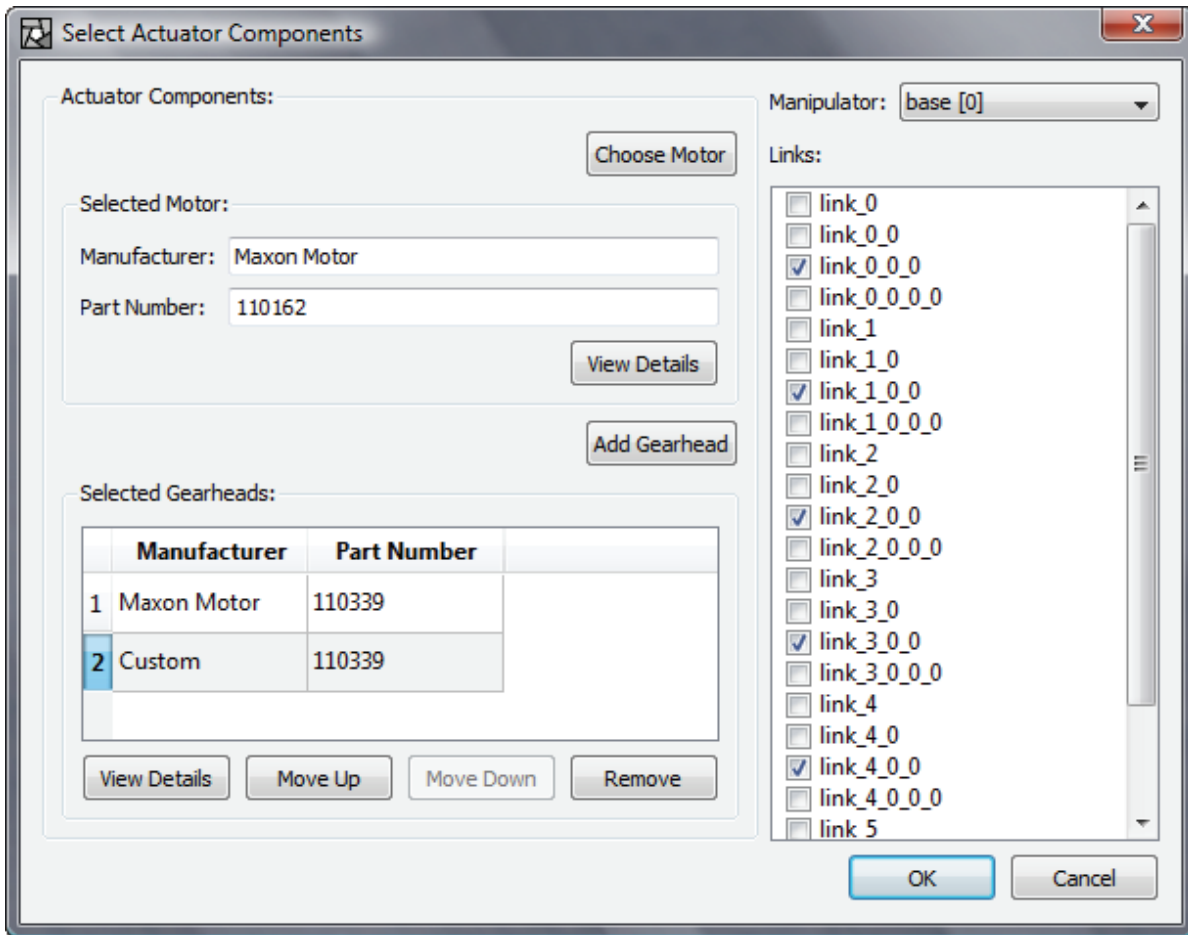


(a)



(b)

**Figure 7-21:** Dialog to choose a motor from database. The user can search by (a) identifier or (b) specifications. If the desired motor does not exist in the database, the user can add it to the database.



**Figure 7-22:** After choosing a motor and gearheads, the user can view the details of each component. For gearheads, they can be moved up/down the chain or be removed. The user can also apply these components to actuators of multiple links.

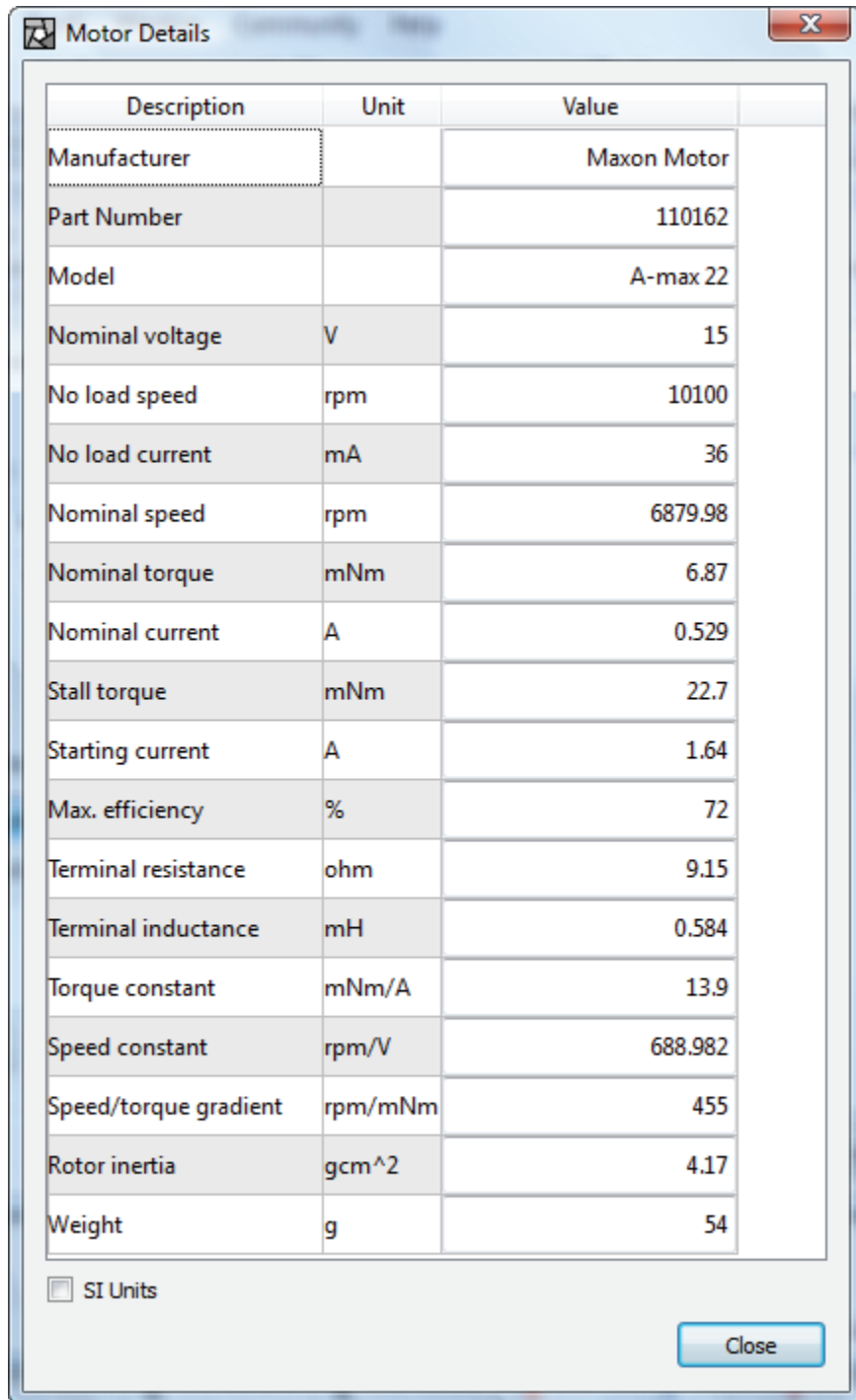
Figure 7-22 depicts the “Select Actuator Components” dialog after the desired motor and gearheads have been chosen. The details of each component can be inspected by clicking the View Details button. The ‘Details’ dialogs in Figure 7-23 and Figure 7-24 show the details of a motor and a gearhead, respectively. The user can view the specifications in SI units by checking the ‘SI Units’ checkbox.

The right pane of the dialog in Figure 7-22 shows a drop-down list of manipulators and a list of all links for the current manipulator. First the user select which manipulator these components will be applied to by choosing from the drop-down list. For the selected manipulator, the user can apply the selected components to the joint actuators of multiple links simultaneously by checking the checkboxes before the desired link labels. This is very convenient especially for robots with symmetry such as the hexapod robot in this example where in all likelihood one type of actuator is likely to apply to the same joint of all six legs.

Once all the desired links have been chosen and the OK button is clicked, the properties of the motor and gearheads will be used to construct a joint actuator and the joint actuator will be added to the desired links. The user can repeat the ‘select actuator components’ process until all the actuators have been completed.



Note that for selecting a motor is necessary but adding gearheads is optional. This allows direct-drive actuators to be composed.



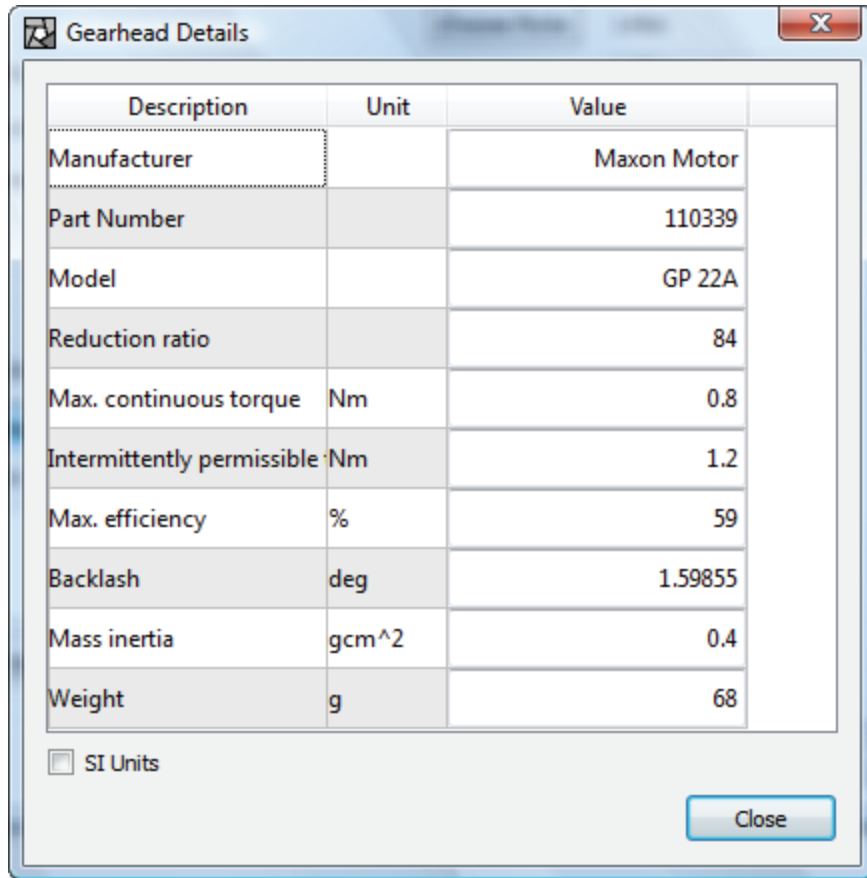
The image shows a dialog box titled "Motor Details" with a close button in the top right corner. It contains a table with three columns: "Description", "Unit", and "Value". The table lists various motor specifications. At the bottom of the dialog, there is a checkbox labeled "SI Units" which is currently unchecked, and a "Close" button.

Description	Unit	Value
Manufacturer		Maxon Motor
Part Number		110162
Model		A-max 22
Nominal voltage	V	15
No load speed	rpm	10100
No load current	mA	36
Nominal speed	rpm	6879.98
Nominal torque	mNm	6.87
Nominal current	A	0.529
Stall torque	mNm	22.7
Starting current	A	1.64
Max. efficiency	%	72
Terminal resistance	ohm	9.15
Terminal inductance	mH	0.584
Torque constant	mNm/A	13.9
Speed constant	rpm/V	688.982
Speed/torque gradient	rpm/mNm	455
Rotor inertia	gcm <sup>2</sup>	4.17
Weight	g	54

SI Units

Close

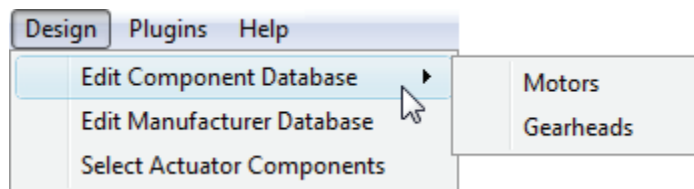
**Figure 7-23:** Dialog showing the details of a motor. The user can switch to SI units by checking the checkbox.



**Figure 7-24:** Dialog showing the details of a gearhead. The user can switch to SI units by checking the checkbox.

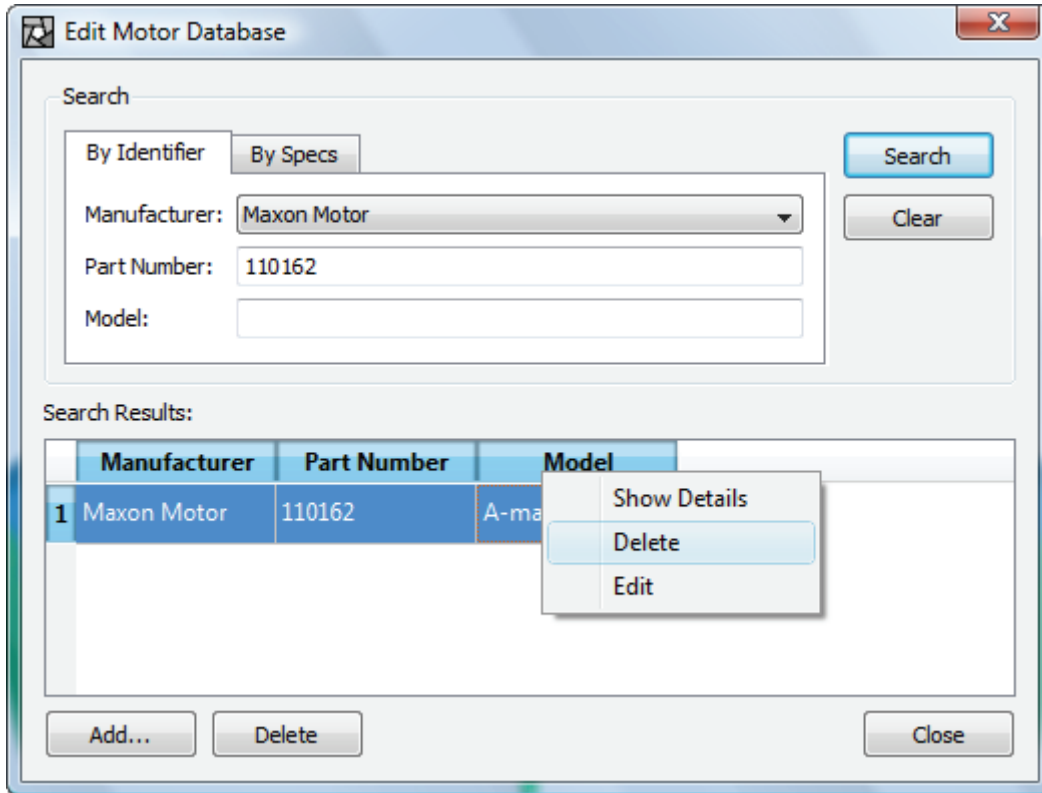
### 7.8.2 Managing Database

In addition to allowing the user to select components for actuators, the actuator plugin provides the means for the user to manage the entries in the database. This is essential since it is highly unlikely that the database will contain every component desired by the designer. Figure 7-25 shows the Design menu items that can be used to manage the database.



**Figure 7-25:** Design menu items to edit database. The user can edit motor, gearhead, or manufacturer database.

To edit the motor database, select Design->Edit Component Database->Motors from the menubar. This will bring up the “Edit Motor Database” dialog shown in Figure 7-26.



**Figure 7-26:** The “Edit Motor Database” dialog allows the user to add, delete, or edit motors to/from/in the database. This dialog is similar to “Choose Motor” dialog in Figure 7-21.

Using the “Edit Motor Database” dialog, the user can add a motor to the database by clicking the “Add...” button, which will bring up the dialog shown in Figure 7-27. There, after the user enters the required motor details and clicks the Submit button, the motor will be added to the database. If the Cancel button is clicked, the motor will not be added and the entered data will be lost.

To delete or edit a motor in the database, first the user needs to search for the desired motor. Once the desired motor shows up in the search results, the user can right-click on that motor and select either to delete or edit the motor. Note that the user can follow the same procedures just described to add/edit/delete a motor but in the “Choose Motor” dialog show in Figure 7-21.

Description	Unit	Value
Manufacturer*		Maxon Motor
Part Number*		0
Model*		
Nominal voltage	V	0
No load speed*	rad/s	0
No load current*	A	0
Nominal speed*	rad/s	0
Nominal torque*	Nm	0
Nominal current	A	0
Stall torque*	Nm	0
Starting current	A	0
Max. efficiency*		1
Terminal resistance	ohm	0

SI Units \* required fields

Reset Submit Cancel

**Figure 7-27:** Add a motor to database. The user enters the details of the motor to be added. \* appended to the field description indicates that the field is required.

To edit the gearhead database, Design->Edit Component Database->Motors from the menubar. The procedures of adding/editing/deleting a gearhead are identical to those of a motor previously described. Figure 7-28 shows the dialog for entering the details of a gearhead and adding it to the database.

Description	Unit	Value
Manufacturer*		Maxon Motor
Part Number*		0
Model*		
Reduction ratio*		1
Max. continuous torque	Nm	0
Intermittently permissible	Nm	0
Max. efficiency*	%	100
Backlash	deg	0
Mass inertia*	gcm <sup>2</sup>	0
Weight	g	0

SI Units \* required fields

Reset Submit Cancel

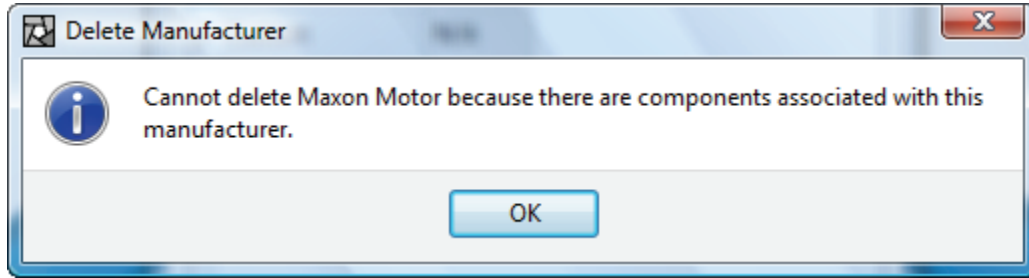
**Figure 7-28:** Add a gearhead to database. The user enters the details of the gearhead to be added. \* appended to the field description indicates that the field is required.

	Name	Country
1	Maxon Motor	USA
2	Custom	N/A

Add Delete Close

**Figure 7-29:** Edit manufacturer database.

Finally, the user can manage the database for manufacturers by selecting Design->Edit Manufacturer Database, which will bring up the dialog shown in Figure 7-29. A new manufacturer can be added by clicking the Add button. A new row will be added to the end of the table that allows the user to enter the information. The user can press the Tab key to go to the next column. Once all information has been entered, press the Enter key to submit the new manufacturer into the database. To delete a manufacturer, select the desired manufacturer and click the Delete button. However, if there are still some components associated with this manufacturer, it will not be removed and the warning message in Figure 7-30 will be displayed. Those components must be removed from the database before the manufacturer can be deleted.



**Figure 7-30:** A warning message is issued if the user is attempting to delete a manufacture whose components still exist in the database.

## 7.9 Spring and Damper Properties

In this section, the mechanism is described for specifying springs between (and among) joints. These springs and dampers are modeled during dynamic simulation and can be used by the control system as well.

### 7.9.1 Approach

The spring and damper are defined in a general sense. The spring torque applied to joint  $i$  is a function of any number of associated joint angles, as follows:

$$\tau_i = k_i \sum_{j=0}^{n_i-1} a_{ij} (q_{ij} - d_{ij}), \quad (7-13)$$

where  $k_i$  is a spring constant for joint  $i$ ,  $a_{ij}$  is weighting factor  $j$ ,  $d_{ij}$  is offset  $j$ ,  $q_{ij}$  is associated joint angle  $j$ ,  $n_i$  is the number of associated joints.

In the simplest nontrivial case, the torque at joint  $i$  is just a function of joint  $i$  itself. That is,

$$\tau_i = k_i (q_i - d_i). \quad (7-14)$$

The damper torque applied on each joint is similarly defined as

$$\tau_i = k_i \sum_{j=0}^{n_i-1} a_{ij} \dot{q}_{ij}, \quad (7-15)$$

where  $k_i$  is the damper constant,  $a_{ij}$  is the weighting factor,  $\dot{q}_{ij}$  is the joint rate of the  $j^{\text{th}}$  associated joint and  $n_i$  is the number of associated joints for joint  $i$ .

### 7.9.2 Implementation

For implementation, a class named *EcSpringProperties* contains the following information:

Member Data	Class/Type	Description	Restrictions
m_SpringConstant	<i>EcXmlReal</i>	The spring constant $k_i$	None.
m_WeightingFactors	<i>EcXmlRealVector</i>	A vector of spring weights, $a_{i1}, a_{i2}, \dots, a_{i(n_i-1)}$	None.
m_SpringOffsets	<i>EcXmlRealVector</i>	A vector of offsets, $d_{i1}, d_{i2}, \dots, d_{i(n_i-1)}$	None.
m_AssociatedJoints	<i>EcXmlStringVector</i>	List of joint labels	None.

**Table 7-13:** *EcSpringProperties*.

A method for evaluating (6-13) is implemented in this class as well. An instance of *EcSpringProperties* resides in *EcManipulatorLink* as a member variable.

The class for the damper is named *EcDamperProperties*. It contains all the variables introduced in (6-15), as shown in the table below and a method for evaluating (6-15). An instance of this class also resides in *EcManipulatorLink* as a member variable.

Member Data	Class/Type	Description	Restrictions
m_DamperConstant	<i>EcXmlReal</i>	The damper constant $k_i$	None.
m_WeightingFactors	<i>EcXmlRealVector</i>	A vector of damper weights, $a_{i1}, a_{i2}, \dots, a_{i(n_i-1)}$	None.
m_AssociatedJoints	<i>EcXmlStringVector</i>	List of joint labels	None.

**Table 7-14:** *EcDamperProperties*.

### 7.9.3 Example

**Figure 7-31** shows an example used for this task. A spring in the form of (6-14) is assigned to each of the joints with a small spring constant and a zero spring offset. Each joint actuator has friction associated with it. The simulation started with the pendulum in its initial configuration as shown in the left sub-figure of Figure 7-31. It moved into the steady-state configuration shown in the right

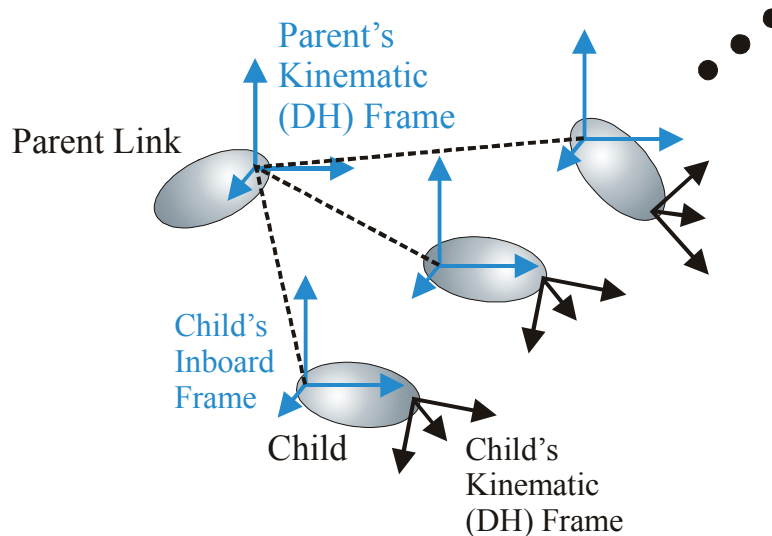
sub-figure of Figure 7-31, where all the joint angles are zero. Gravity in this simulation is set to zero. This is the file “springPendulum.xml” in the examples directory.



**Figure 7-31:** An example used to demonstrate springs and dampers. The sub-figure on the left shows the initial condition of the pendulum and the one on the right shows the resting configuration. This simulation is contained in the file “springPendulum.xml” in the examples directory.

## 7.10 Child Links

Any link can have any number of child links. The child links’ placements are defined such that the inboard frame of the child aligns with the kinematic (Denavit-Hartenberg) frame of the parent. This is illustrated in the figure below. Every child has a pointer to its parent.



**Figure 7-32:** Each link can have any number of child links. The child links are placed such that their inboard frames align with the parent’s kinematic (DH) frame. Each child’s degree of freedom then places its kinematic frame with respect to the parent’s kinematic frame.



## 7.11 Methods for Calculating Link Data

The links provide methods for calculating its position, velocity, and other properties. These methods are summarized in the table below:

Method	Description
subtreeLinkCount	Returns the number of links in the subtree with this link as the root (includes this link in the count).
mapLinks	Maps all links in the subtree with this as the root (includes this link in the map). Pointers to links are placed in a vector and a label-to-link-pointer map is created.
dhFrameInSystem	Calculates and returns a reference to a coordinate system transformation describing the kinematic (DH) frame for this link in system coordinates. Takes an active state as its input and modifies it. The active state can be reused to quickly calculate other components.
primaryFrameInSystem	Calculates and returns a reference to a coordinate system transformation describing the primary frame for this link in system coordinates. Takes an active state as its input and modifies it. The active state can be reused to quickly calculate other components.
dhFrameVelocityInSystem	Calculates and returns a reference to a general velocity object describing the kinematic (DH) frame motion in system coordinates (with point of application at the DH-frame origin). Takes an active state as its input and modifies it. The active state can be reused to quickly calculate other components.
dhFrameVelocityInLocal	Calculates and returns a reference to a general velocity object describing the kinematic (DH) frame motion in DH-frame coordinates (with point of application at the DH-frame origin). Takes an active state as its input and modifies it. The active state can be reused to quickly calculate other components.
primaryFrameVelocityInSystem	Calculates and returns a reference to a general velocity object describing the primary frame motion in system coordinates (with point of application at the primary-frame origin). Takes an active state as its input and modifies it. The active state can be reused to quickly calculate other components.
dhFrameAccelerationInLocal	Calculates and returns a reference to a general acceleration object describing the kinematic (DH) frame acceleration in DH-frame coordinates (with point of application at the DH-frame origin). Takes an active state as its input and modifies it. The active state can be reused to quickly calculate other components.

dhFrameForceInLocal	Calculates and returns a reference to a general force object describing the linear and angular force that must be applied to the kinematic (DH) frame in DH-frame coordinates (with point of application at the DH-frame origin) to achieve the motion described in the active state that is input. The active state can be reused to quickly calculate other components.
crbi	Calculates the composite rigid-body inertia of the subtree with this link at its root (i.e., this and all outboard links) represented in the link's D-H frame. Takes an active state as its input and modifies it. The active state can be reused to quickly calculate other components.
arbd	Calculates the articulated-body dynamics (articulated-body inertia and bias force) for the subtree with this link at its root. The articulated-body dynamics are represented in the link's D-H frame. Takes external forces and joint torques as input. Also takes an active state as its input and modifies it. The active state can be reused to quickly calculate other components.
collectLeafLinks	Collects all leaf links (links with no children) in the subtree with this link as its root.

**Table 7-15:** List of prominent data-calculation methods in *EcManipulatorLink*.

## 7.12 Example Code

### 7.12.1 Creating a Sphere-Shaped Link

Code for creating a simple sphere-shaped link is shown in Text Box 7-9. This code creates the link from scratch and saves it as “sphereLink.xml.” This file can be loaded with the ActinViewer.

The code in Text Box 7-9 does the following: It instantiates the link object. It then adds Denavit-Hartenberg kinematics and a joint actuator. The label for the link is set to “exampleLink,” and the mass is set to that of a sphere with mass 10.0 kg and radius 1.0 m.

```

// declare an error return code
EcBoolean success;

// declare a link object
EcManipulatorLink link;

// set the kinematics
EcDenavitHartenberg dh;
link.setLinkKinematics(dh);

// set the joint actuator
EcJointActuator act=
    EcJointActuator(0.004,0.001,30.0,-1000,1000,-10.0,10.0,0.08,1.0,2.0);
link.setJointActuator(act);

// set the link label
link.setLabel("exampleLink");

// set the mass properties to that of a sphere
EcRigidBodyMassProperties massProperties=
    EcRigidBodyMassProperties::uniformSolidSphere(10.0,1.0);
link.setMassProperties(massProperties);

// set the shape as a sphere
EcSphere sphere;
sphere.setRadius(1.0);
link.setShape(sphere);

// save the link as a plain XML file
success=link.writeToFile("sphereLink.xml",EcManip::EcManipulatorLinkToken);

// make sure it saved properly
if(!success)
{
    EcWARN("Could not save link.\n");
    return;
}

```

**Text Box 7-9:** Example code for creating a sphere-shaped link. This is Example Section #1 in the link example code.

The code in Text Box 7-10 creates four different geometric shapes and saves them as links. These links can be loaded and rendered using the ActinViewer.

```

// give the link a tetrahedron shape and save it
EcVector v1(0.0,0.0,0.0);
EcVector v2(1.0,0.0,0.0);
EcVector v3(0.5,1.0/sqrt(12.0),sqrt(2.0/3.0));
EcVector v4(0.5,sqrt(3.0)/2.0,0.0);
EcTetrahedron tetrahedron(v1,v2,v3,v4);
link.setShape(tetrahedron);
success=link.writeToFile("tetrahedronLink.xml",
                        EcManip::EcManipulatorLinkToken);

// give the link a capsule shape and save it
EcLineSegment segment(EcVector::zeroVector(),
                      EcVector::xVector(1.0));
EcCapsule capsule(segment,0.5);
link.setShape(capsule);
success=link.writeToFile("capsuleLink.xml",
                        EcManip::EcManipulatorLinkToken);

// give the link a lozenge shape and save it
EcRectangle rectangle(
    EcVector::zeroVector(),
    EcVector::xVector(1.0),
    EcVector::yVector(2.0));
EcLozenge lozenge(rectangle,0.5);
link.setShape(lozenge);
success=link.writeToFile("lozengeLink.xml",
                        EcManip::EcManipulatorLinkToken);

// give the link an ellipsoid shape and save it
EcEllipsoid ellipsoid(1.0,2.0,3.0);
link.setShape(ellipsoid);
success=link.writeToFile("ellipsoidLink.xml",
                        EcManip::EcManipulatorLinkToken);

```

**Text Box 7-10:** Example code for creating links with tetrahedral, capsule, lozenge, and ellipsoidal shapes. This is Example Section #2 in the link example code.

The code in Text Box 7-11 creates four different geometric shapes and saves them as links. These links can be loaded and rendered using the ActinViewer.

```

// an extent to hold the polyhedron
EcPolyPhysicalExtent extent;

// a variable to hold the points
EcXmlVectorVector points;

EcVector lengthVector=EcVector::xVector(0.5);
EcVector widthVector =EcVector::yVector(1.0);
EcVector heightVector=EcVector::zVector(2.0);

// points, front then back
points.pushBack(lengthVector+widthVector-heightVector);
points.pushBack(lengthVector+widthVector+heightVector);
points.pushBack(lengthVector-widthVector+heightVector);
points.pushBack(lengthVector-widthVector-heightVector);
points.pushBack(-lengthVector+widthVector-heightVector);
points.pushBack(-lengthVector+widthVector+heightVector);
points.pushBack(-lengthVector-widthVector+heightVector);
points.pushBack(-lengthVector-widthVector-heightVector);

// set the points
extent.setPoints(points);

// polygons
EcPolygonWithKeyVector polygons;

// front (+x)
EcPolygonWithKey polygon;
polygon.setSurfaceKey("surface");
polygon.setPointIndices(4, 0,1,2,3);
polygons.pushBack(polygon);

// right (+y)
polygon.setPointIndices(4, 4,5,1,0);
polygons.pushBack(polygon);

// bottom (+z)
polygon.setPointIndices(4, 6,2,1,5);
polygons.pushBack(polygon);

// left
polygon.setPointIndices(4, 3,2,6,7);
polygons.pushBack(polygon);

// top
polygon.setPointIndices(4, 3,7,4,0);
polygons.pushBack(polygon);

// back
polygon.setPointIndices(4, 4,7,6,5);
polygons.pushBack(polygon);

extent.setPolygons(polygons);
link.setShape(extent);
success=link.writeToFile("polyhedralLink.xml",
    EcManip::EcManipulatorLinkToken);

```

**Text Box 7-11:** Example code for creating a link with a polyhedral shape. This is Example Section #3 in the link example code.

The code in Text Box 7-12 builds on the code in Text Box 7-11 to change the color of the link. The appearance properties are set in the surface using a string-number map. The file that is saved can be loaded and rendered using the ActinViewer for comparison with the link created through the code in Text Box 7-11.

```
// create a surface and set the spectral properties
EcSurfaceProperty surface;

// ambient color parameters
surface.add(EcManip::EcAmbientRedToken,0.4);
surface.add(EcManip::EcAmbientGreenToken,0.4);
surface.add(EcManip::EcAmbientBlueToken,0.8);
surface.add(EcManip::EcAmbientAlphaToken,0.0);

// diffuse color parameters
surface.add(EcManip::EcDiffuseRedToken,0.3);
surface.add(EcManip::EcDiffuseGreenToken,0.3);
surface.add(EcManip::EcDiffuseBlueToken,0.3);
surface.add(EcManip::EcDiffuseAlphaToken,0.3);

// specular color parameters
surface.add(EcManip::EcSpecularRedToken,0.0);
surface.add(EcManip::EcSpecularGreenToken,0.0);
surface.add(EcManip::EcSpecularBlueToken,0.0);
surface.add(EcManip::EcSpecularAlphaToken,0.0);

// emmissive color parameters
surface.add(EcManip::EcEmissionRedToken,0.05);
surface.add(EcManip::EcEmissionGreenToken,0.05);
surface.add(EcManip::EcEmissionBlueToken,0.5);
surface.add(EcManip::EcEmissionAlphaToken,0.0);

// shininess parameter
surface.add(EcManip::EcShininessToken,5.0);

// add the surface to map of surfaces
EcStringSurfacePropertyMap surfaces;
surfaces.add(EcXmlString("surface"),surface);

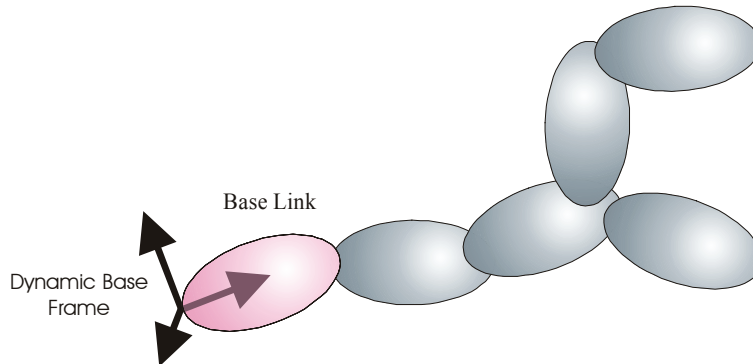
// set the surface properties in the extent
extent.setSurfaces(surfaces);

link.setShape(extent);
success=link.writeToFile("bluePolyhedralLink.xml",
    EcManip::EcManipulatorLinkToken);
```

**Text Box 7-12:** Example code for creating a link with a polyhedral shape. This is Example Section #4 in the link example code. It builds on the code shown in Text Box 7-11.

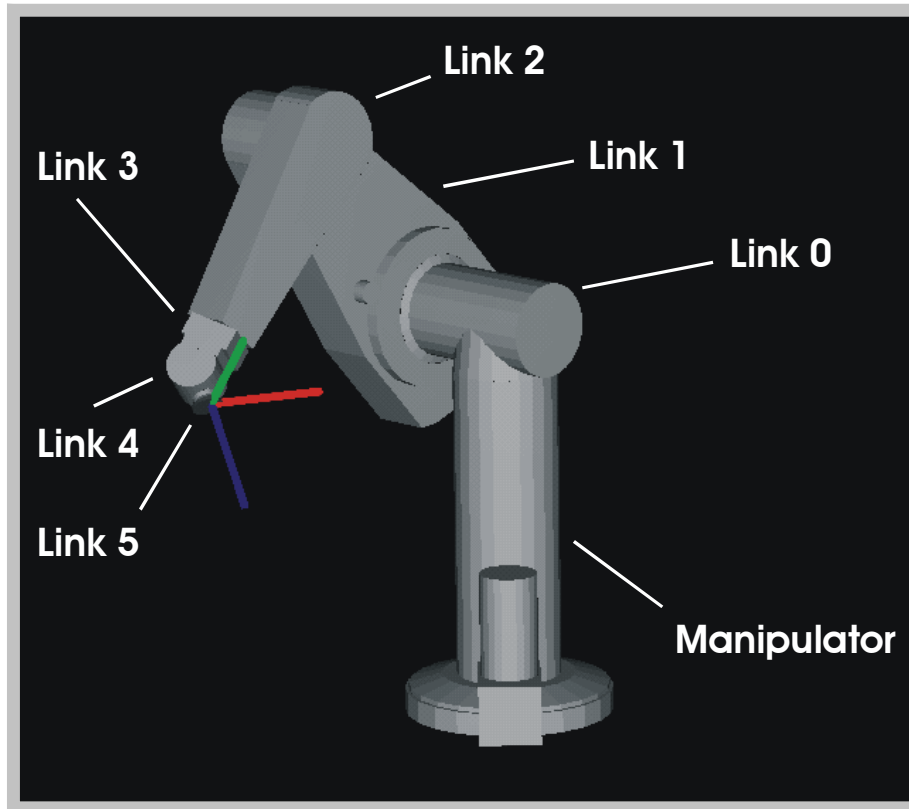
## 8 The Manipulator

The robotic mechanism is defined by connecting links as described in the previous section. In the construction of a manipulator, every link has a parent except for one, the base link. The base link can either be fixed with respect to the environment or free moving. In addition to serving a special role as the root of the link tree defining the manipulator and defining its location, the C++ class describing the base link also provides a programming interface to the manipulator. The organization of the manipulator is illustrated in the figure below.



**Figure 8-1:** A manipulator is organized as a tree of links. One link on the manipulator serves a special role as the base link. It does not have a joint associated with it. Instead, the location of the base frame determines its location. In code, the class describing the base link is a subclass of the normal link type and stores extra information on the entire manipulator.

The organization is illustrated for a physical manipulator through Figure 8-2, which shows a PUMA manipulator. The *EcIndividualManipulator* object in this case represents the base of the manipulator, which does not move as the joint values change. This same object also provides the programming interface to the entire robot.



**Figure 8-2:** The PUMA has six degrees of freedom and is represented using seven rigid bodies. The manipulator object contains a description of the base. The first moving link (link 0) is a child of the manipulator link. The rest of the arm is defined using a serial kinematic chain, with each link containing one child link except for the last link (link 5), which has no children.

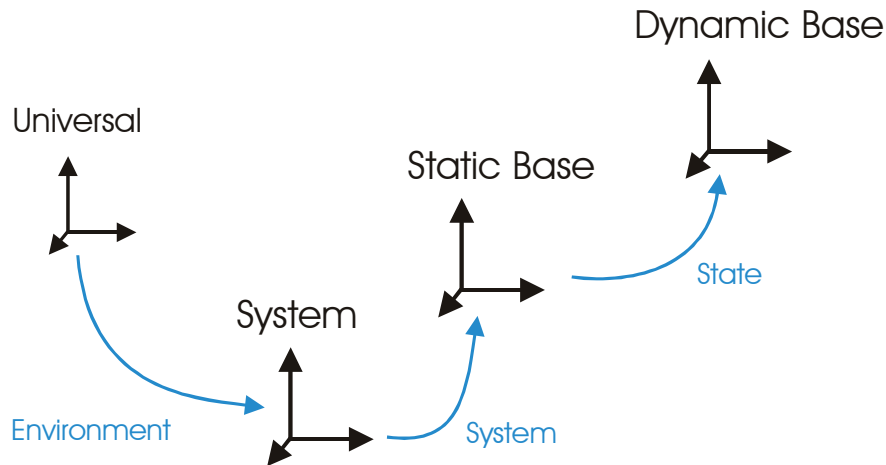
This chapter describes the organization of the link tree and the details on the interface that the base link provides.

## 8.1 Reference Frames

The location of every link in the manipulator is defined through the following process: First, the location of the base link is defined with respect to a reference frame, then the location of each child link is defined with respect to its parent. Every link in the manipulator has the base link as an ancestor in the tree, so this uniquely specifies the location of every link.

The location of the base of each manipulator is specified through a sequence of transformations, as shown in Figure 8-3. The entire manipulator system is represented in the system frame, which is defined with respect to a universal reference frame as part of the environmental specification. Then, each manipulator has a static base frame whose location with respect to the system frame is specified as part of the manipulator (system) description. The location of the base link is then specified through a coordinate system describing the location of the dynamic base frame with respect to the static frame. This coordinate system changes with each time step for mobile manipulators and remains constant for fixed-based manipulators.





**Figure 8-3:** Location of a manipulator’s base frame (as shown in Figure 8-1). The dynamic base frame location is identified through a sequence of transformations. The location of the static base frame is part of the system description, and the relative location of the dynamic base frame is part of the state. For mobile manipulators, the motion of the dynamic base frame is integrated with the motion of the joints.

## 8.2 Methods for Calculating Manipulator Data

The base link is an object of the class *EcIndividualManipulator*, which is subclassed from *EcManipulatorLink*. The base link provides all the information available from a regular link, plus information on the complete manipulator. It includes all the methods shown in Table 7-15 plus those shown in the table below.

Method	Description
absoluteBoundingSphere	Gets a reference to a sphere that bounds the manipulator independent of its state.
manipulatorIndex	Gets the index for the manipulator.
jointDof	Gets the number of degrees of freedom in the manipulator excluding base motion.
jointAndBaseDof	Gets the number of degrees of freedom in the manipulator including base motion.
distanceTo	Computes the distance from this to another manipulator.
checkIntersect	Determines if this and another manipulator intersect.
checkSelfCollision	Determines if any pair of links that form this manipulator intersect.

crbiSpatialCholeskyDecomposition	Gets the Cholesky Decomposition (L such that $L*L^T=M$ ) of the 6x6 spatial representation of the composite rigid-body inertia (as represented in the DH frame).
linkByIdentifier	Returns a pointer to the link with the specified string identifier. Returns NULL if no link exists with that identifier.
linkByIndex	Returns a pointer to the link with the specified integer index. Returns NULL if no link exists with that index (as would happen if the index were larger than the manipulator DOF).
propagateState	Propagates a manipulator's state forward in time. Stops joints at joint limits and at collisions.
mapManipulator	Builds maps for quick access to links based on indices and labels.
surfaces	Gets the surface property collection for the manipulator.
selfCollisionLinkMap	Gets self-collision information for the manipulator.
lookup	Looks up a surface property.
addCapsuleBoundingVolumeToLinks	Adds a capsule bounding volume to the bounding volume hierarch for each link, if it does not already exist
canCollide	Determines whether two links can collide.
lookup	Looks up a surface property.

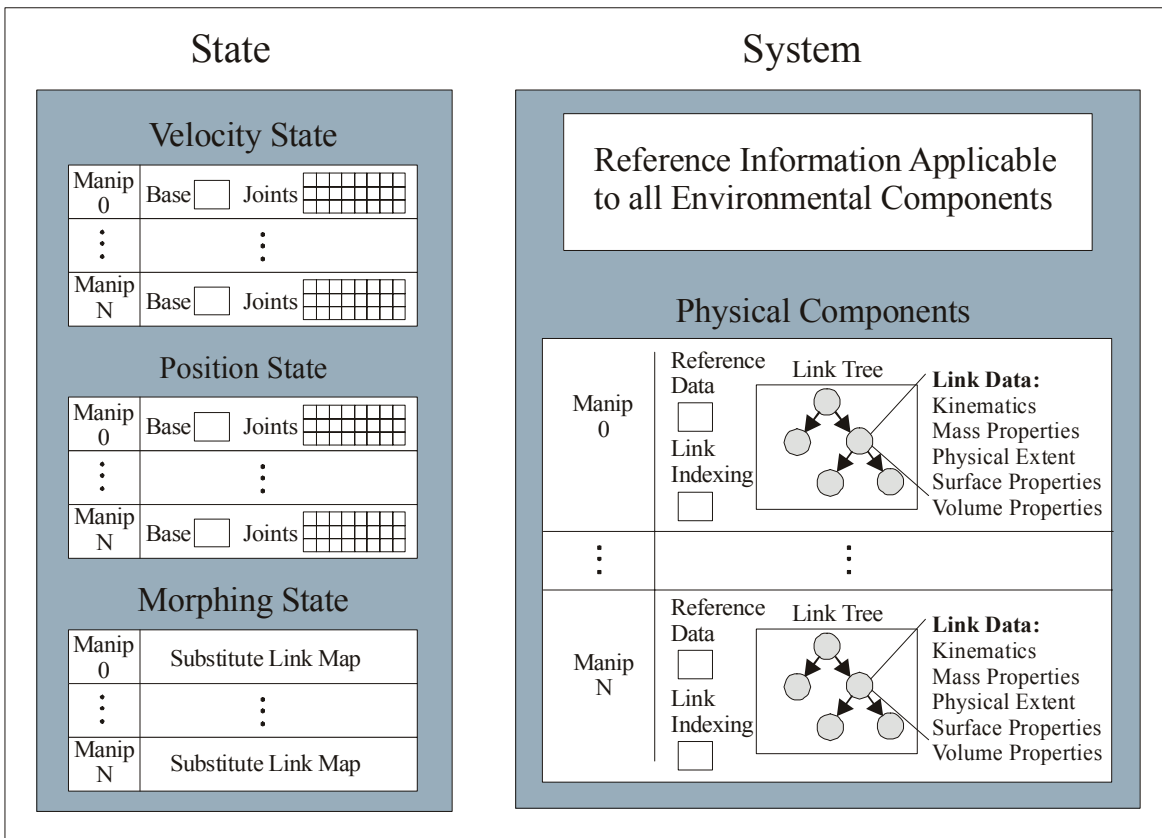
**Table 8-1:** List of prominent data-calculation methods in *EcIndividualManipulator* that complement those in the parent class, *EcManipulatorLink*.

### 8.3 Link and Manipulator References

To reduce XML file size, TCP/IP bandwidth, and rendering time, Actin includes link and manipulator references. When components of a robotic mechanism are repeated there is redundancy if all components are explicitly represented. In addition, if a simulation uses multiple mechanisms of the same type, then explicit representation of each mechanism produces redundancy.

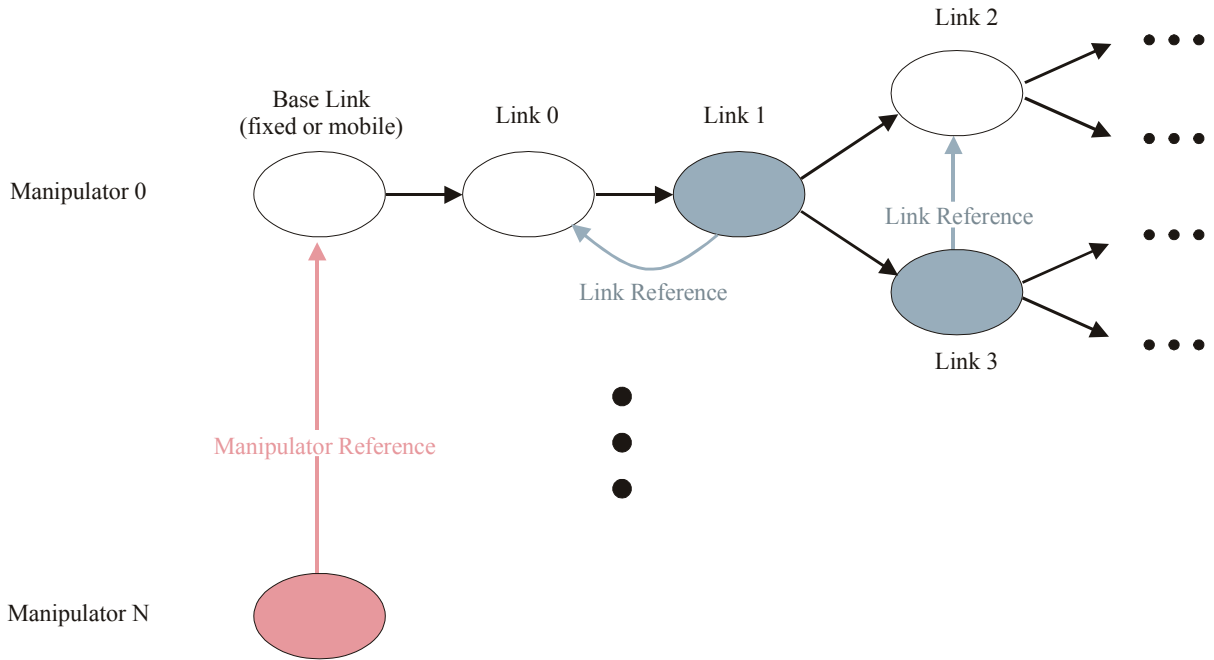
All components of the simulation are represented using the dichotomy of system and state in a single stated-system class (*EcStatedSystem*). This class is illustrated in the figure below. The system remains constant, timestep to timestep, while the state changes. The bulk of the footprint of the simulation, both in file and memory size, lies in the system. In particular, it is the physical extent of the manipulators and their environment which takes the most resources. These are typically represented through polygon meshes, with many thousands of polygons.

## Stated System



**Figure 8-4:** Organization of the physical system in a simulation. All data is contained in one of two groups, the system or the state. The system remains constant during a simulation, while the state changes timestep to timestep.

The organization shown in the figure above allows referencing on the redundant parts of the system, both links and entire manipulators. The new organization of the manipulator system, which includes these references, is shown in the figure below.



**Figure 8-5:** Organization of the system (shown on the right-hand side of Figure 8-4) using references. Any link in any manipulator can reference another link in the same manipulator, and any manipulator can reference another manipulator.

The organization shown in the figure above is implemented by having each link hold a link-reference description, and each manipulator hold a manipulator-reference description. These two data sets are implemented through classes with the profile shown in the table below.

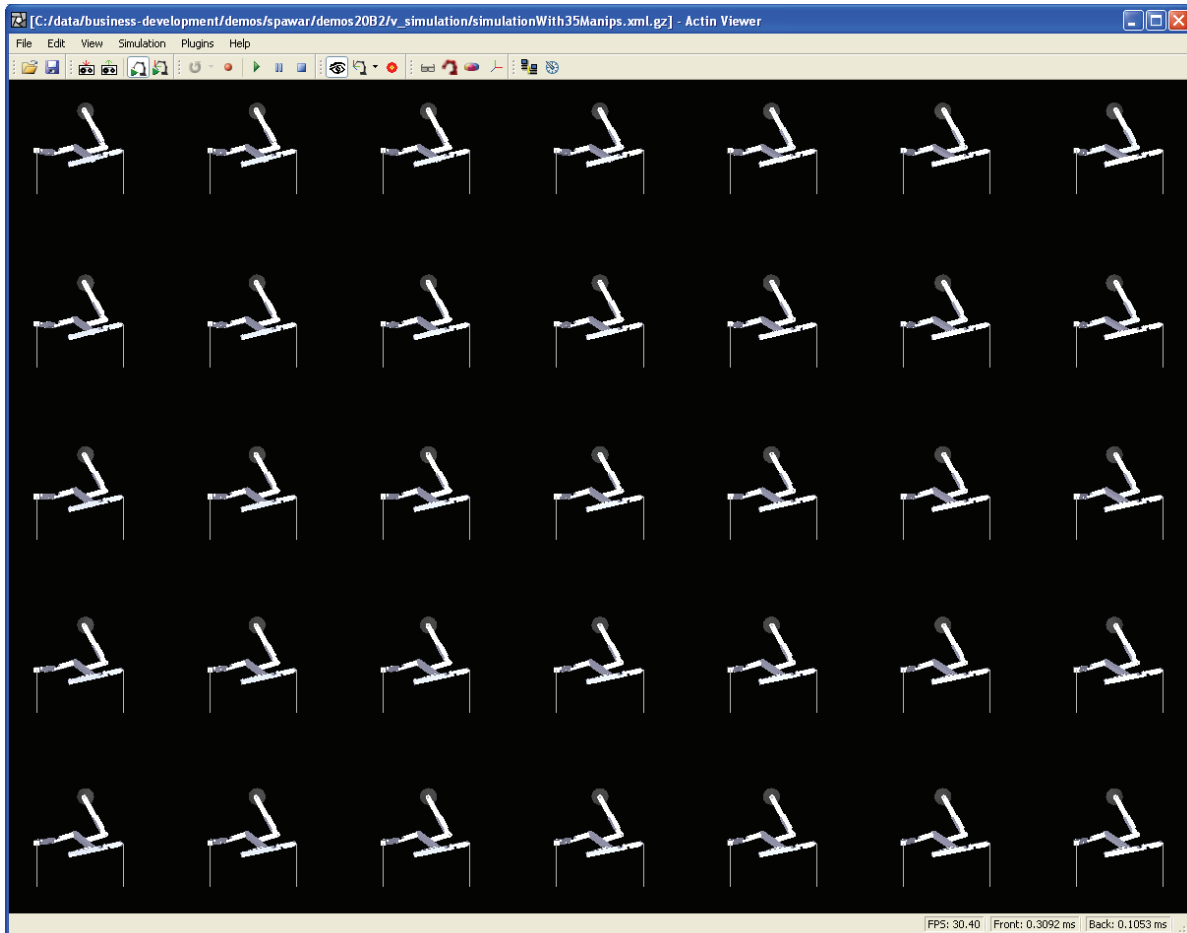
Class	Flag	Reference String
<i>EcManipulatorReferenceDescription</i>	m_IsManipulatorReferenced	m_ReferencedManipulatorLabel
<i>EcLinkReferenceDescription</i>	m_IsShapeReferenced	m_ReferencedLinkLabel

**Table 8-2:** Manipulator and link reference information is established through the two classes shown. Each contains a flag and the string label for the referenced entity.

Each link holds a pointer to a link reference. If the link is not referenced, this pointer is set to point to the parent link itself. The reference pointer is always used to access the shape container. With this implementation, the physical extent of the link is seamlessly represented in the same way, whether or not the link is referenced. Beyond the shape container, no other aspect of the link is referenced. That is, it uses its own link kinematics and actuator properties. Thus, the entire kinematic structure of the robot must still be explicitly represented, even when using link references.

If a manipulator is referenced, the manipulator uses the shape for the base link of the referenced manipulator and also references the children of the referenced manipulator. This allows an arbitrarily complex mechanism to be represented using a single *EcIndividualManipulator* object with no children.

The figure below shows a specific example of the improvement produced by referencing. The file size for this simulation, which includes 35 identical 12-dof mechanisms, is 1 MB without referencing and 53 KB with referencing.



**Figure 8-6:** With the new link and manipulator referencing, this simulation (using 35 independently controlled 12-dof mechanisms) has a file size of only 53 KB and loads almost instantaneously. This simulation uses both link and manipulator referencing.

## 8.4 Example Code

### 8.4.1 Creating a Bitmapped Base Link

Code for creating a simple box-shaped link is shown in Text Box 8-1 and Text Box 8-2. This code creates an *EcIndividualManipulator* object with six sides and bitmaps an image to it. The image is “robotImage.jpg,” which is delivered with the toolkit. The manipulator is saved as “bitmapBox.xml.” This file can be loaded with the ActinViewer.

The code in Text Box 8-1 does the following: It instantiates the manipulator object and gives it a string label. It then adds a single surface property. This surface property includes the name of the texture that will be used as an image for mapping. It builds a set of three-dimensional vertices to define the physical extent, and it builds a set of two-dimensional map points into the texture image.

```

// create a manipulator object
EcIndividualManipulator manipulator;

// set the label
manipulator.setLabel("box");

// surface properties with a texture
EcStringSurfacePropertyMap surfaces;
EcSurfaceProperty surface;
surface.add(EcManip::EcAmbientRedToken,0.99);
surface.add(EcManip::EcAmbientGreenToken,0.99);
surface.add(EcManip::EcAmbientBlueToken,0.99);
surface.add(EcManip::EcSpecularRedToken,0.9);
surface.add(EcManip::EcSpecularGreenToken,0.9);
surface.add(EcManip::EcSpecularBlueToken,1.0);
surface.add(EcManip::EcTextureFilenameToken,"robotImage.jpg");
surfaces.add(EcXmlString("box"),surface);

// set the extent
EcPolyPhysicalExtent baseExtent;
baseExtent.setSurfaces(surfaces);

// make a point set
EcXmlVectorVector points;

// build the box
EcVector top=EcVector(0.0, 1, 0.0);
EcVector bottom=EcVector(0.0, -1, 0.);
EcReal er=0.5;
points.pushBack(top+er*EcVector::xVector()+er*EcVector::zVector());
points.pushBack(top+er*EcVector::xVector()-er*EcVector::zVector());
points.pushBack(top-er*EcVector::xVector()-er*EcVector::zVector());
points.pushBack(top-er*EcVector::xVector()+er*EcVector::zVector());

points.pushBack(bottom+er*EcVector::xVector()+er*EcVector::zVector());
points.pushBack(bottom+er*EcVector::xVector()-er*EcVector::zVector());
points.pushBack(bottom-er*EcVector::xVector()-er*EcVector::zVector());
points.pushBack(bottom-er*EcVector::xVector()+er*EcVector::zVector());

baseExtent.setPoints(points);

// set the raster map points
// map the entire texture to all sides of the box
EcPlanarVectorVector rasterMapPoints;
rasterMapPoints.pushBack(EcPlanarVector(0.0,0.0));
rasterMapPoints.pushBack(EcPlanarVector(0.0,1.0));
rasterMapPoints.pushBack(EcPlanarVector(1.0,1.0));
rasterMapPoints.pushBack(EcPlanarVector(1.0,0.0));
baseExtent.setRasterMapPoints(rasterMapPoints);

```

**Text Box 8-1:** Start of example code for creating a bit-mapped box manipulator. This is the start of Example Section #1 in the manipulator example code.

The code in Text Box 8-2 does the following: It constructs the polygons defining the manipulator by referencing the three-dimensional vertices and the two-dimensional raster-map points. It sets the manipulator's shape, then saves the manipulator as an XML file.

```

// ...continued

// set the base polygons

EcPolygonWithKey polygon;
EcPolygonWithKeyVector polygons;

polygon.setSurfaceKey("box");

// map the entire texture to all sides of the box
polygon.setRasterMapIndices(4,0,1,2,3);

// build the box
EcU32 ba=0;

polygon.setPointIndices(4,0+ba,1+ba,2+ba,3+ba);
polygons.pushBack(polygon);

polygon.setPointIndices(4,4+ba,7+ba,6+ba,5+ba);
polygons.pushBack(polygon);

polygon.setPointIndices(4,7+ba,3+ba,2+ba,6+ba);
polygons.pushBack(polygon);

polygon.setPointIndices(4,3+ba,7+ba,4+ba,0+ba);
polygons.pushBack(polygon);

polygon.setPointIndices(4,0+ba,4+ba,5+ba,1+ba);
polygons.pushBack(polygon);

polygon.setPointIndices(4,1+ba,5+ba,6+ba,2+ba);
polygons.pushBack(polygon);

baseExtent.setPolygons(polygons);

manipulator.setShape(baseExtent);
manipulator.setIsFixedBase(EcTrue);

// write the manipulator to a file
EcBoolean success=manipulator.writeToFile("bitmapBox.xml",
    EcManip::EcIndividualManipulatorToken);

```

**Text Box 8-2:** Continuation of example code for creating a bit-mapped box manipulator. This is part of Example Section #1 in the manipulator example code.

### **8.4.2 Creating a Mechanism with One Joint**

Code for adding a single joint to the manipulator created in Text Box 8-1 and Text Box 8-2 is shown in Text Box 8-3 and Text Box 8-4. The manipulator is saved as “oneJointManipulator.xml.” This file can be loaded with the ActinViewer.

The code in Text Box 8-3 creates and sets the polygons in the physical extent. The code in Text Box 8-4 sets the surface properties for the physical extent and uses the physical extent to model the link. It then, creates the link kinematics and mass properties and connects the new link to the base created through Text Box 8-1 and Text Box 8-2.

```

// an extent to hold the polyhedron
EcPolyPhysicalExtent extent;

// a variable to hold the linkPoints
EcXmlVectorVector linkPoints;

EcVector lengthVector=EcVector::xVector(0.2);
EcVector widthVector =EcVector::yVector(0.2);
EcVector heightVector=EcVector::zVector(0.2);

// linkPoints, front then back
linkPoints.pushBack(lengthVector+widthVector-heightVector);
linkPoints.pushBack(lengthVector+widthVector+heightVector);
linkPoints.pushBack(lengthVector-widthVector+heightVector);
linkPoints.pushBack(lengthVector-widthVector-heightVector);
linkPoints.pushBack(-lengthVector+widthVector-heightVector);
linkPoints.pushBack(-lengthVector+widthVector+heightVector);
linkPoints.pushBack(-lengthVector-widthVector+heightVector);
linkPoints.pushBack(-lengthVector-widthVector-heightVector);

// set the linkPoints
extent.setPoints(linkPoints);

// linkPolygons
EcPolygonWithKeyVector linkPolygons;

// front (+x)
EcPolygonWithKey linkPolygon;
linkPolygon.setSurfaceKey("surface");
linkPolygon.setPointIndices(4, 0,1,2,3);
linkPolygons.pushBack(linkPolygon);

// right (+y)
linkPolygon.setPointIndices(4, 4,5,1,0);
linkPolygons.pushBack(linkPolygon);

// bottom (+z)
linkPolygon.setPointIndices(4, 6,2,1,5);
linkPolygons.pushBack(linkPolygon);

// left
linkPolygon.setPointIndices(4, 3,2,6,7);
linkPolygons.pushBack(linkPolygon);

// top
linkPolygon.setPointIndices(4, 3,7,4,0);
linkPolygons.pushBack(linkPolygon);

// back
linkPolygon.setPointIndices(4, 4,7,6,5);
linkPolygons.pushBack(linkPolygon);

extent.setPolygons(linkPolygons);

// continued...

```

**Text Box 8-3:** Start of example code for adding a child link to the manipulator created in the last section. This is the start of Example Section #2 in the manipulator example code.



```

// ...continued

// clear the old surface properties
surface.clear();

// ambient color parameters
surface.add(EcManip::EcAmbientRedToken,0.4);
surface.add(EcManip::EcAmbientGreenToken,0.4);
surface.add(EcManip::EcAmbientBlueToken,0.8);

// specular color parameters
surface.add(EcManip::EcSpecularRedToken,0.7);
surface.add(EcManip::EcSpecularGreenToken,0.5);
surface.add(EcManip::EcSpecularBlueToken,0.5);

// shininess parameter
surface.add(EcManip::EcShininessToken,50.0);

// add the surface to map of surfaces
surfaces.clear();
surfaces.add(EcXmlString("surface"),surface);

// set the surface properties in the extent
extent.setSurfaces(surfaces);

EcManipulatorLink link;
link.setShape(extent);

// set the Denavit-Hartenberg link kinematics data
EcDenavitHartenberg dh;
dh.setJointType(0);
dh.setDhAlpha(EcHalfPi);
dh.setDhD(0.6);
dh.setDhA(0.0);
link.setLinkKinematics(dh);

// set the actuator
link.setJointActuator(EcJointActuator::testObject());

// set the label
link.setLabel("link1");

// set center-of-mass mass properties (using a rectangular-prism model)
EcRigidBodyMassProperties massProp;
massProp.set(1.0,EcVector(0,0,0),EcSecondMoment(1,1,1));

// set the mass properties
link.setMassProperties(massProp);

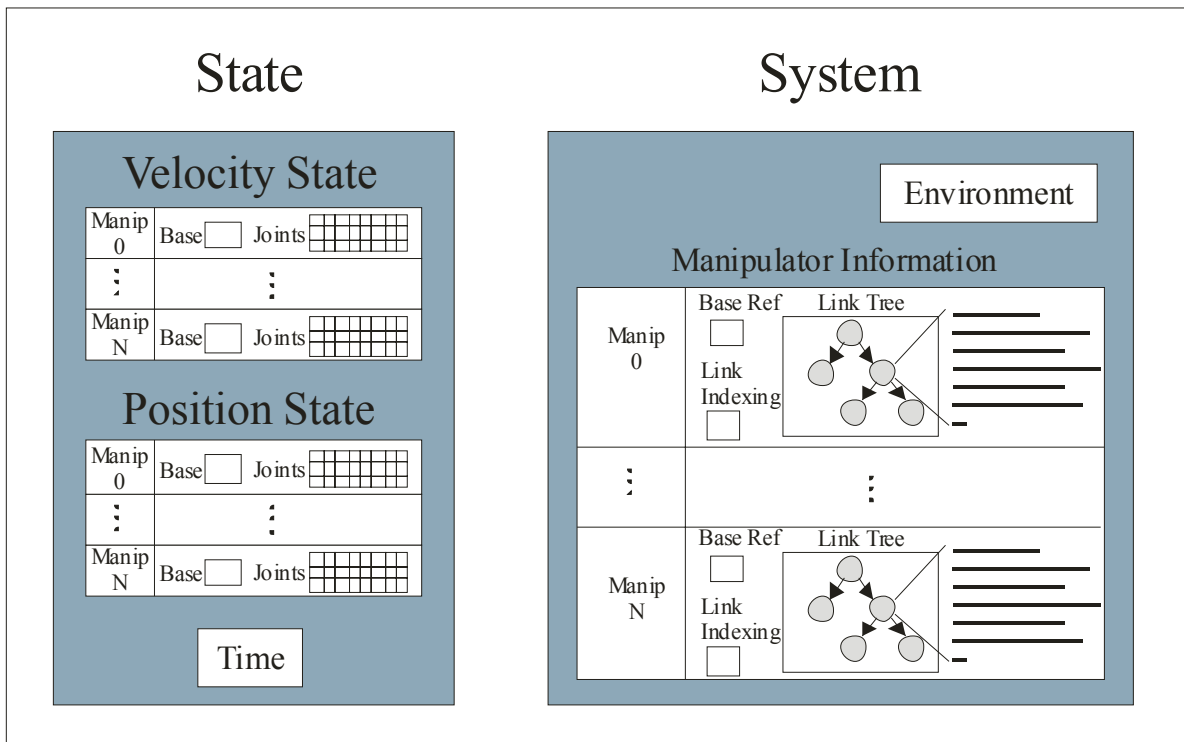
// write out the result
manipulator.addChild(link);
success=manipulator.writeToFile("oneJointManipulator.xml",
    EcManip::EcIndividualManipulatorToken);

```

**Text Box 8-4:** Continuation of example code for creating a one-joint manipulator. This is part of Example Section #2 in the manipulator example code.

## 9 The Stated System

The complete manipulator structure is described through a dichotomy: system and state. The system remains the same, time step to time step, while the state changes. The system may change upon unique events, but does not change with each time step. The system is decomposed into any number of manipulators, each of which is represented through any number of links. Manipulators have both a unique integer and string identifier, and each link has a unique integer and string identifier within the manipulator. Link integer identifiers use depth-first ordering in the tree. The state is decomposed into a velocity and a position state, each of which is organized by manipulator and link ID. Figure 9-1 below illustrates this organization. Separating system and state allows easy logging, check pointing, and storage of the information that changes with time.



**Figure 9-1:** The representation of the manipulators and the environment are organized into a system and state. The state changes from time step to time step, while the system remains static outside of exceptional events.

### 9.1 Description of EcStatedSystem

The stated system is made up of a system and state. Table 9-1 shows the registered member variables and Table 9-2 shows the primary methods of the class.

Member Data	Class/Type	Description	Restrictions
m_pSystem	<i>EcManipulator System</i>	Pointer to static system data.	None
m_State	<i>EcManipulator SystemState</i>	State data that changes frequently.	None
m_Processing Parameters	<i>EcSystem Processing Parameters</i>	Parameters used for processing the system during simulation and control (maximum number of GJK iterations).	None

**Table 9-1:** Stated system registered member variable description. This is class *EcStatedSystem*. See the code documentation for a complete list of member variables and descriptions.

Method	Description
propagateSelfTo	Propagate the internal state in time. This keeps the same velocity state and integrates the position state using forward Euler. Diagnostic data is optionally returned which indicates whether a collision occurred or a joint limit was hit.
propagateTo	Propagate the input state in time. This keeps the same velocity state and integrates the position state using forward Euler. Diagnostic data is optionally returned which indicates whether a collision occurred or a joint limit was hit.
propagateSingleStateBy	Propagate a single manipulator state by delta-time. This keeps the same velocity state and integrates the position state using forward Euler. The manipulator number is given by index. Diagnostic data is optionally returned which indicates whether a collision occurred or a joint limit was hit.
addCompatibleState	Creates a state that is a compatible size with the system.
setToMidpointState	Creates a state that is a compatible size with the system and it initializes each joint position to be half way between the limits.
validateTopLevelDimensions	Ensures a match in the numbers of manipulators in the system and state.
validateLowLevelDimensions	Ensures that the sizes of manipulators match by verifying that the number of joint positions and velocities in the state match the number of degrees of freedom in the system.
aboutToModifySystem	Since the system is generally large, it is not copied when passed around. Instead, a pointer to the system is copied from object to object. A counter is kept to determine how many objects are in

	use of the system so that it can be deleted properly. New pointers to new or modified systems are created when a system is about to be modified or created. The counter is set to zero at that time.
absoluteBoundingSphere	Calculates a bounding sphere. This bounding sphere is a function of the base positions and orientations, but not the joint positions.
checkForCollisions	Checks for manipulator collisions given the current state. It checks for self collisions first, then manipulator to manipulator collisions.
gjkDistanceQuery	Get the shortest distance between two <i>EcShape</i> objects. This is called from <i>checkForCollisions</i> for each manipulator.

**Table 9-2:** List of prominent methods in *EcStatedSystem*.

The propagation methods are used by the simulation and the position control system to propagate the simulation state and the control system state. As the state is propagated, a check for collisions is performed.

The system is stored in *EcStatedSystem* as a pointer. To save on memory, this pointer is shared by other objects that have the exact same system. If a system in a particular object is about to change, the *aboutToModifySystem* method needs to be called to give the object a unique state object for changing. Otherwise, the system would be changed in every object connected to the system pointer. *EcStatedSystem* contains a reference counter pointer which provides information about the system pointer. If the reference counter pointer is null, then a call to *aboutToModifySystem* creates a new system and reference counter pointer, and it initializes the counter to zero. A zero count means that only one object has access to the system. If the reference pointer has been created and is set to zero, a call to *aboutToModifySystem* does nothing; the object contains the only reference to the system and can freely modify the system without affecting any other objects. If the counter is greater than zero, a call to *aboutToModifySystem* disconnects the object from that system and creates a cloned copy of the system that it can modify. The object also disconnects from the counter and obtains a new one which is initialized to zero.

### 9.1.1 Description of *EcManipulatorSystem*

The manipulator system is made up of a vector of manipulators, an environment, and a vector of link interaction processors. Table 9-3 shows the member variables and Table 9-4 shows the primary methods of the class.

Member Data	Class/Type	Description	Restrictions
m_Manipulators	<i>EcIndividualManipulator</i> <i>Vector</i>	Vector of manipulators	None
m_Environment	<i>EcSystemEnvironment</i>	Environment for the system. This includes the gravity vector, and a transformation from a universal frame to the	None

		system frame.	
m_LinkInteraction Vector	<i>EcLinkInteractionsVector</i>	Vector of interaction models between links. The vector options are currently a spring and damper force model and a collision force container which can be upgraded with new force models.	None

**Table 9-3:** Description of registered manipulator system member variables. This is class *EcManipulatorSystem*. See the code documentation for a complete list of member variables and descriptions.

Method	Description
jointDof	Gets the number of joint degrees of freedom in the system excluding the position and orientation of the base.
jointAndBaseDof	Gets the number of joint degrees of freedom in the system including the position and orientation of the base.
isCompatible	Check system size compatibility with state.
absoluteBoundingSphere	Calculates a bounding sphere. This bounding sphere is a function of the base positions and orientations, but not the joint positions.
numberOfManipulators	Get the number of manipulators in the system.
setManipulatorBackPointers	Sets the back pointer to the stated system in the manipulators so that the system knows which stated system it is a part of.

**Table 9-4:** List of prominent methods in *EcManipulatorSystem*.

Most of the information stored in the system is contained in the manipulator vector which is described in the previous section on manipulators. The environment contains general information such as the coordinate system transformation from the universal frame to the system frame, and a gravity vector. Table 9-5 provides a description of these variables.

While the propagation methods of the stated system call a method for detecting collisions, the link interaction vector provides a mechanism for creating forces between links and objects that can be dynamically simulated. Currently, the link interaction vector can contain a spring and damper model (i.e., *EcSpringAndDamperBetweenLinks*) or a generic force container (i.e., *EcLinkCollisionForce*). The generic force container holds two additional options: *EcMassSpringCollisionForceProcessor* and *EcRestitutionModelCollisionForceProcessor*. New models can be added to the link interaction vector and the generic force container through a dynamic library (e.g., DLL under Windows)—please see Section 18. The link interaction vector elements describe the types of interactions that will be dynamically simulated. For example, if a manipulator is required to bounce a ball, two link interactions need to be described: the interaction between the manipulator and the ball, and the

interaction between the ball and the floor. The link interaction vector assumes that all interactions are between links of manipulators. In this example, the ball and floor are described as trivial manipulators for the purpose of enabling the link interaction processors to calculate the force between objects.

Member Data	Class/Type	Description	Restrictions
m_CoordSysXForm	<i>EcCoordinateSystemTransformation</i>	The transformation from the universal frame to the system frame	None
m_UpGravityVector	<i>EcXmlVector</i>	The acceleration due to gravity in the up direction in the system frame	None

**Table 9-5:** Complete list of environment system member variables. This is class *EcSystemEnvironment*.

### 9.1.2 Description of *EcManipulatorSystemState*

The manipulator state is made up of a vector of joint positions and velocities, and a time tag. Table 9-6 shows the member variables and Table 9-7 shows the primary methods of the class. The state data is primarily manipulated by the propagation methods of the stated system class.

Member Data	Class/Type	Description	Restrictions
m_PositionStates	<i>EcPositionStateVector</i>	Vector of manipulator position states. This is a vector of joint position vectors. The top level vector is for separating the joint positions for each manipulator.	None
m_VelocityStates	<i>EcVelocityStateVector</i>	Vector of manipulator velocity states. This is also a vector of joint velocity vectors.	None
m_Time	<i>EcXmlTime</i> Same as <i>EcXmlReal</i>	Time associated with the state	None

**Table 9-6:** Complete list of manipulator system member variables.

Method	Description
linearInterpolation	Sets <i>this</i> to be a linear interpolation between two other state values. Returns true for success and false if the state times are equivalent.
validateTopLevelDimensions	Ensures that the sizes of position states match the sizes of velocity states. The position states take precedence. (i.e., when there is a discrepancy, the velocity state is modified.)
validateLowLevelDimensions	Ensures that the size of the joint position vector matches the size of joint velocity vector. The position vector takes precedence. (i.e., when there is a discrepancy, the velocity vector is modified.)

**Table 9-7:** List of prominent methods in *EcManipulatorSystemState*.

## 9.2 Description of EcVisualizableStatedSystem

The visualizable stated system is made up of a stated system, and visualization parameters. The visualization parameters enable the rendering of the stated system. Table 9-8 shows the list of member variables.

Member Data	Class/Type	Description	Restrictions
m_PovParameters	<i>EcPovParameters</i>	Parameters describing the kinematics of the view such as eye point, center of interest, and field of view.	None
m_LightParameters	<i>EcLightParameters</i>	Parameters describing the scene lighting. <i>EcLight</i> contains a description of numerous lighting parameters.	None
m_RenderParameters	<i>EcRenderParameters</i>	Parameters describing the rendering process such as near and far clipping distance and anti-aliasing parameters	None
m_DisplayOptions	<i>EcDisplayOptions</i>	Display options such as collision, joint limit, and end effector options.	None
m_GUIObject Parameters	<i>EcGUIObjectParameters</i>	GUI parameters such as size of the guide frame and center	None

		of interest	
--	--	-------------	--

**Table 9-8:** Complete list of visualization parameters member variables. This is for class *EcVisualizationParameters*.

### 9.3 Example

Code for creating a visualizable stated system for a two manipulator system is described in this section. The example starts by creating the system for the stated system. The system needs a definition for the manipulators, the environment, and the link interaction vector. Text Box 9-1 illustrates the creation of the manipulators. Since an illustration for creating a manipulator is described in detail in the manipulator section of this user's guide, the details for creating a manipulator in this example are hidden in the manipulator *testObject* method.

```

// create a first manipulator

// see the manipulator example for more details on creating a manipulator
EcIndividualManipulator manip1=EcIndividualManipulator::testObject();

EcCoordinateSystemTransformation xform;
xform.setTranslation(EcVector(1.0,1.0,0.0));
xform.setOrientation(EcOrientation(1.0,0.0,0.0,0.0));

manip1.setCoordSysXForm(xform);

// create a second manipulator

EcIndividualManipulator manip2=EcIndividualManipulator::testObject();

xform.setTranslation(EcVector(-1.0,1.0,0.0));
xform.setOrientation(EcOrientation(1.0,0.0,0.0,0.0));

manip2.setCoordSysXForm(xform);

// add the two test manipulators to the system

EcIndividualManipulatorVector manipulators;
manipulators.pushBack(manip1);
manipulators.pushBack(manip2);

```

**Text Box 9-1:** Start of example code for creating a stated system. In this section, the system manipulators are created. This is Example Section #1 in the stated system example code.

Text Box 9-2 illustrates how the environment for the system is created. The environment contains a coordinate transformation and a gravity vector.

```

xform.setTranslation(EcVector(1.0,1.0,0.0));
xform.setOrientation(EcOrientation(1.0,0.0,0.0,0.0));

EcSystemEnvironment env;

env.setCoordSysXForm(xform);
env.setUpGravityVector(EcVector(0.0,9.8,0.0));

```

**Text Box 9-2:** Example creation of the system environment. This is Example Section #2 in the stated system example code.



Text Box 9-3 illustrates how the link interaction vector is created. This example creates one link interaction element between two links.

```
EcSpringAndDamperBetweenLinks forceBetweenLink;
forceBetweenLink.setManipulatorOneIndex(0);
forceBetweenLink.setManipulatorOneIndex(1);
forceBetweenLink.setLinkOneIndex(1);
forceBetweenLink.setLinkTwoIndex(1);
xform.setToIdentity();
forceBetweenLink.setFrameOffsetOne(xform);
forceBetweenLink.setFrameOffsetTwo(xform);
forceBetweenLink.setAngularDamperConstant(0.1);
forceBetweenLink.setAngularSpringConstant(1.2);
forceBetweenLink.setAngularSpringOffset(1.0);
forceBetweenLink.setLinearDamperConstant(0.1);
forceBetweenLink.setLinearSpringConstant(2.0);
forceBetweenLink.setLinearSpringOffset(12.0);

EcLinkInteractionsVector forceBetweenLinkVector;
forceBetweenLinkVector.pushBack(forceBetweenLink);
```

**Text Box 9-3:** Example creation of the system link interaction vector. This is Example Section #3 in the stated system example code.

Text Box 9-4 illustrates how the manipulator vector, environment, and link interaction vector are placed into the system.

```
EcManipulatorSystem system;

system.setManipulators(manipulators);
system.setEnvironment(env);
system.setLinkInteractionsVector(forceBetweenLinkVector);
```

**Text Box 9-4:** Now that the manipulators, environment, and link interaction vector are in place, the system can be created. This is Example Section #4 in the stated system example code.

Text Box 9-5 illustrates how the joint position and velocity states are created and how they are placed into the state.

```

EcPositionStateVector    positionStates;
EcVelocityStateVector    velocityStates;

EcU32 numManip=system.manipulators().size();
for(EcU32 ii=0;ii<numManip;++ii)
{
    // create position state holder
    EcPositionState posState;

    // get the number of degrees of freedom in the manipulator
    EcU32 dof=system.manipulators()[ii].jointDof();

    // set the position state
    posState.setCoordSysXForm
        (EcCoordinateSystemTransformation::nullObject());
    EcXmlRealVector values(12);
    values[0]=0.503881;
    values[1]=-0.46942;
    values[2]=-1.77795;
    values[3]=0.994958;
    values[4]=0.425054;
    values[5]=2.13634;
    values[6]=-0.011515;
    values[7]=0.0592323;
    values[8]=0.579904;
    values[9]=0.826406;
    values[10]=-0.334235;
    values[11]=0.299118;

    posState.setJointPositions(values);

    // create velocity state holder
    EcVelocityState velState;

    // set the velocity states
    velState.setGeneralVelocity(EcGeneralVelocity::nullObject());

    // assign zero to each velocity
    values.assign(dof,EcXmlReal(0.0));
    velState.setJointVelocities(values);

    // push the states onto the list
    positionStates.pushBack(posState);
    velocityStates.pushBack(velState);
}

// set the state
EcManipulatorSystemState state;

state.setPositionStates(positionStates);
state.setVelocityStates(velocityStates);

```

**Text Box 9-5:** Create and initialize the state. This is Example Section #5 in the stated system example code.

Given a system and state, Text Box 9-6 shows how a stated system is created.

```

EcStatedSystem statedSystem;

statedSystem.setSystem(system);
statedSystem.setState(state);

```

**Text Box 9-6:** Example for creating a stated system. This is Example Section #6 in the stated system example code.

A visualizable stated system needs a stated system and visualization parameters for rendering. The visualization parameters include point-of-view, light, and rendering parameters. Text Box 9-8 illustrates how the point-of-view visualization parameters are created.

```

EcPovParameters pov;
pov.setEyepoint(EcVector(0.0, -6.0, 15.0));
pov.setCenterOfInterest(EcVector(0.0, 1.0, 0.0));
pov.setFieldOfView(0.15);

```

**Text Box 9-7:** Example creation of the point-of-view parameters of the visualizable stated system. This is Example Section #7 in the stated system example code.

Text Box 9-8 illustrates how the light visualization parameters are created.

```

EcLight light;
light.setIsOn(EcTrue);
light.setAmbient(EcColor(0.1, 0.1, 0.1, 1.0));
light.setDiffuse(EcColor(0.75, 0.75, 0.75, 1.0));
light.setSpecular(EcColor(0.75, 0.75, 0.75, 1.0));
light.setPosition(EcVector(0.0, 0.0, 1.0));
light.setIsPositional(EcTrue);
light.setSpotDirection(EcVector(0.0, 0.0, -1.0));
light.setSpotExponent(0.0);
light.setSpotCutoff(EcPi);
light.setConstantAttenuation(1.0);
light.setLinearAttenuation(0.0);
light.setQuadraticAttenuation(0.0);

EcLightParameters lights;
EcLightVector lightVector;
lightVector.pushBack(light);
lights.setLights(lightVector);

```

**Text Box 9-8:** Example creation of the light parameters of the visualizable stated system. This is Example Section #8 in the stated system example code.

Text Box 9-9 illustrates how the render visualization parameters are created.

```

EcRenderParameters render;
render.setUseDefaults(EcTrue);
render.setNearClippingDistance(0.1);
render.setFarClippingDistance(20.0);
render.setNumJitterPoints(1);
render.setAccuPixelRange(1);

```

**Text Box 9-9:** Example creation of the render parameters of the visualizable stated system. This is Example Section #9 in the stated system example code.

Text Box 9-10 illustrates how the point-of-view, light, and render parameters come together to form the visualization parameters.

```
EcVisualizationParameters vizParameters;

vizParameters.setPovParameters(pov);
vizParameters.setLightParameters(lights);
vizParameters.setRenderParameters(render);
```

**Text Box 9-10:** Now that the point-of-view, light, and render parameters are in place, the visualization parameters object can be created. This is Example Section #10 in the stated system example code.

Text Box 9-11 illustrates how the stated system and visualization parameters come together to form the visualizable stated system.

```
EcVisualizableStatedSystem vizStatedSystem;

vizStatedSystem.setVisualizationParameters(vizParameters);
vizStatedSystem.setStatedSystem(statedSystem);
```

**Text Box 9-11:** Example creation of the visualizable stated system. This is Example Section #11 in the stated system example code.

The example code for the stated system concludes by rendering the visualizable stated system, as shown in the following text box:

```
// instantiate a renderer
EcRenderWindow renderer;

// set the size of the window
const EcU32 size = 320;
renderer.setWindowSize(2*size, size);

// view the system
renderer.setVisualizableStatedSystem(vizStatedSystem);
renderer.renderScene();

// pause for 1 second
EcSLEEPMS(3000);
renderer.closeScene();
```

**Text Box 9-12:** Rendering the visualizable stated system. This is Example Section #12 in the stated system example code.

## 10 Velocity Control

Velocity control is the study of how to move the joints to properly move the hand. It includes a variety of techniques, such as pseudoinverse control [6], weighted pseudoinverse control [7], augmented Jacobian techniques [8], [9], extended Jacobian techniques [10], [11], and projection methods [12], [13]. Velocity control contrasts with position control, which is the study of how to position the joints to properly place the hand. Both types of control are needed for dexterous manipulator operation. Velocity and position control are also often called, respectively, inverse velocity kinematics and inverse position kinematics.

Velocity control is the central strength of the Actin™ toolkit. It uses a patented approach that combines powerful XML-based configuration methods with the generic velocity-control algorithms described in [14]. It applies to virtually all types of kinematically redundant and bifurcating manipulators. Kinematically redundant arms are those with more degrees of freedom in arm motion than degrees of freedom in hand motion. Bifurcating manipulators are those that branch by having at least one link connect to more than one extension (or child link).

### 10.1 Algorithmic Description

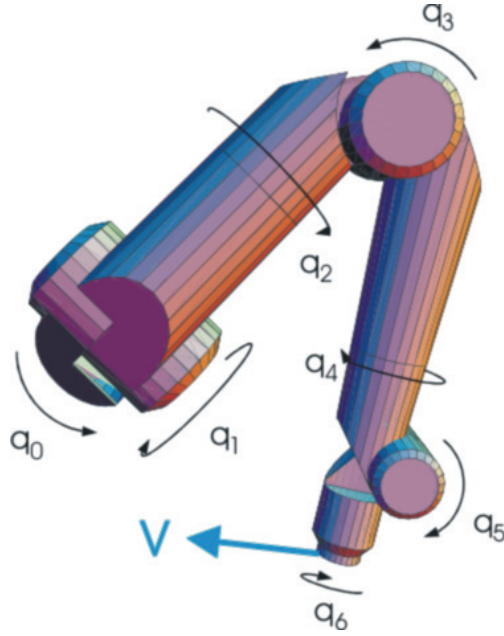
Engid Technologies' Actin™ approach is multifaceted, with algorithmic, language, and software-implementation components. This section describes the algorithms.

#### 10.1.1 Core Algorithmic Framework

The core velocity framework is based on the manipulator Jacobian equation:

$$\mathbf{V} = \mathbf{J}(\mathbf{q})\dot{\mathbf{q}}. \quad (10-1)$$

Here  $\mathbf{V}$  is an  $m$ -length vector representation of the motion of the hand or hands (usually some combination of linear and angular velocity referenced to points rigidly attached to parts of the manipulator);  $\mathbf{q}$  is the  $n$ -length vector of joint positions (with  $\dot{\mathbf{q}}$  being its time derivative); and  $\mathbf{J}$  is the  $m \times n$  manipulator Jacobian, a function of  $\mathbf{q}$ . (For spatial arms with a single end effector,  $\mathbf{V}$  is often the frame velocity with three linear and three angular components. In this document, it takes on a larger meaning that includes the concatenation of point, frame, or other motion of multiple end-effectors.) This is illustrated in the figure below.



**Figure 10-1:** An illustration, based on the RRC K-1207i manipulator, of the parameters for velocity control. The column vector  $\mathbf{V}$  represents hand motion (for positioning and orienting, it would be  $6 \times 1$ ), and  $\mathbf{q}$  represents the concatenated joint values (for the RRC K-1207i shown, it would be  $7 \times 1$ ). The Jacobian  $\mathbf{J}(\mathbf{q})$  is the matrix that makes (10-1) true for all possible values of  $\dot{\mathbf{q}}$ . Note  $\mathbf{V}$  can represent a concatenation of values for multiple end effectors.

For any physical manipulator that is not self-connecting, a manipulator Jacobian can be defined to make equation (10-1) true. When the manipulator is kinematically redundant, the dimension of  $\mathbf{V}$  is less than the dimension of  $\mathbf{q}$  ( $m < n$ ), and (10-1) is underconstrained when  $\mathbf{V}$  is specified. By using  $\mathbf{V}$  to represent relative motion, (10-1) can support self-connecting mechanisms by setting the relative motion to zero.

The velocity control question is the following: given a desired hand motion  $\mathbf{V}$ , what are the joint rates  $\dot{\mathbf{q}}$  that best achieve this motion? To answer this, the framework is built on the method described in [14], which uses a scalar  $\alpha$ , a matrix function  $\mathbf{W}(\mathbf{q})$ , and a scalar function  $f(\mathbf{q})$  to solve for  $\dot{\mathbf{q}}$  given  $\mathbf{V}$  through the following formula:

$$\dot{\mathbf{q}} = \begin{bmatrix} \mathbf{J} \\ \mathbf{N}_J^T \mathbf{W} \end{bmatrix}^{-1} \begin{bmatrix} \mathbf{V} \\ -\alpha \mathbf{N}_J^T \nabla f \end{bmatrix}, \quad (10-2)$$

where  $\nabla f$  is the gradient of  $f$  and  $\mathbf{N}_J$  is an  $n \times (n-m)$  set of vectors that spans the null space of  $\mathbf{J}$ . That is,  $\mathbf{J}\mathbf{N}_J = 0$ , and  $\mathbf{N}_J$  has rank  $(n-m)$ . Both  $\nabla f$  and  $\mathbf{N}_J$  are generally functions of  $\mathbf{q}$ . By changing the values of  $\alpha$ ,  $\mathbf{W}$ , and  $f$ , many new and most established velocity-control techniques can be implemented.

The Actin™ Toolkit, however, goes beyond the formulation in (10-2) to create a more general framework. Instead of insisting on the use of the gradient of a function, a general column vector

$\mathbf{F}(\mathbf{q})$  is used. Not all vector functions are gradients. This minor, but important, modification yields the following formula:

$$\dot{\mathbf{q}} = \begin{bmatrix} \mathbf{J} \\ \mathbf{N}_J^T \mathbf{W} \end{bmatrix}^{-1} \begin{bmatrix} \mathbf{V} \\ -\alpha \mathbf{N}_J^T \mathbf{F} \end{bmatrix}. \quad (10-3)$$

Equation (10-3) is the core velocity-control algorithm used in the Actin™ Toolkit. Mathematically, it achieves the desired  $\mathbf{V}$  while minimizing  $\frac{1}{2} \dot{\mathbf{q}}^T \mathbf{W} \dot{\mathbf{q}} + \alpha \mathbf{F}^T \dot{\mathbf{q}}$ . The parameters  $\alpha$ ,  $\mathbf{W}$  and  $\mathbf{F}$  can be defined using XML to give many different types of velocity control.

### 10.1.2 Robust Extension

The framework provided by (10-3) is generally not robust in the presence of kinematic singularities. This is not a problem with (10-3) in particular, but with velocity-control techniques in general. The problem arises because of the limited information on the manipulator provided by the Jacobian. (Given robot parameters, one can calculate the Jacobian, but given a Jacobian, one cannot calculate robot parameters.)

To resolve the robustness problem with velocity control, one must return to the manipulator. This is done indirectly in our technique by recalculating predicted values for  $\mathbf{V}$  as  $\mathbf{q}$  changes. To do this, let the function  $\mathbf{V}$  be defined as  $\mathbf{V}(\mathbf{q}, \dot{\mathbf{q}})$ . This function is defined independently of equation (10-1), because it is possible to calculate  $\mathbf{V}$  directly as a function of  $\mathbf{q}$  and  $\dot{\mathbf{q}}$  faster than by calculating  $\mathbf{J}$  and evaluating (10-1).  $\mathbf{V}(\mathbf{q}, \dot{\mathbf{q}})$  is used to refine the value found through (10-3) by scaling it.

At any point in time, real or simulated, let  $\mathbf{q}_0$  be the joint values,  $\mathbf{V}_0$  be the desired hand motion, and  $\dot{\mathbf{q}}_0$  be the result of applying (10-3). Then, at time  $\Delta t$  later, the motion of the hand,  $\mathbf{V}_1$ , produced by  $\dot{\mathbf{q}}_0$  can be found as follows:

$$\mathbf{V}_1 = \mathbf{V}(\mathbf{q}_0 + \Delta t \dot{\mathbf{q}}_0, \dot{\mathbf{q}}_0). \quad (10-4)$$

Potential problems caused by the occurrence of a kinematic singularity can be determined by comparing  $\mathbf{V}_1$  to  $\mathbf{V}_0$ . The assumption in velocity control is that  $\mathbf{V}_1$  and  $\mathbf{V}_0$  will be similar, and their divergence proves this assumption invalid. An effective way to address this problem when it occurs is to scale  $\dot{\mathbf{q}}_0$  to be small enough that the assumption becomes true again. Let the function  $\beta'(\mathbf{V}_0, \mathbf{V}_1)$  be this scaling term, so that the final joint-rate value is  $\dot{\mathbf{q}} = \beta'(\mathbf{V}_0, \mathbf{V}_1) \dot{\mathbf{q}}_0$ . Because—through (10-4)— $\mathbf{V}_1$  is a function of  $\mathbf{q}_0$  and  $\dot{\mathbf{q}}_0$ , and therefore—through (10-3)—is a function of  $\mathbf{q}_0$  and  $\mathbf{V}_0$ , the final joint-rate value can be written as

$$\dot{\mathbf{q}} = \beta(\mathbf{q}, \mathbf{V}) \begin{bmatrix} \mathbf{J} \\ \mathbf{N}_J^T \mathbf{W} \end{bmatrix}^{-1} \begin{bmatrix} \mathbf{V} \\ -\alpha \mathbf{N}_J^T \mathbf{F} \end{bmatrix}, \quad (10-5)$$

where  $\beta(\mathbf{q}, \mathbf{V})$  is calculated using the intermediate quantities in the equation described above. Equation (10-5) is the algorithmic framework for the robust extension.

In the Actin™ Toolkit,  $\beta(\mathbf{q}, \mathbf{V})$  can be defined using two metrics. The first metric is a measure of joint rate motion, and the second metric is a measure of end-effector motion error due to the linearization performed in using the Jacobian.

### 10.1.3 Reduced Control Calculation

The general core velocity-control algorithm shown in (10-3) can be enhanced for some types of control by taking advantage of the fact that the matrix  $\mathbf{W}$  occurs only in the term  $\mathbf{N}_j^T \mathbf{W}$ , and the vector  $\mathbf{F}$  occurs only in the term  $\mathbf{N}_j^T \mathbf{F}$ . It is sometimes possible to avoid calculating  $\mathbf{W}$  and  $\mathbf{F}$  explicitly and speed up the calculation, and this is the idea behind the reduced control calculation. For reduced control calculation, there is an alternative core algorithm.

In the normal core algorithm,  $\alpha$ ,  $\mathbf{W}$  and  $\mathbf{F}$  are explicitly defined as part of the control expression using XML. In the reduced calculation,  $\alpha$ ,  $\mathbf{A}$  and  $\mathbf{B}$  are used instead, where  $\alpha$  has the same meaning and  $\mathbf{A}$  and  $\mathbf{B}$  are defined as follows:

$$\mathbf{A} = \mathbf{N}_j^T \mathbf{W}. \quad (10-6)$$

$$\mathbf{B} = \mathbf{N}_j^T \mathbf{F}. \quad (10-7)$$

These are used to calculate  $\dot{\mathbf{q}}$  through the following:

$$\dot{\mathbf{q}} = \begin{bmatrix} \mathbf{J} \\ \mathbf{A} \end{bmatrix}^{-1} \begin{bmatrix} \mathbf{V} \\ -\alpha \mathbf{B} \end{bmatrix}. \quad (10-8)$$

This provides a more efficient implementation when  $\mathbf{A}$  and  $\mathbf{B}$  can be calculated more efficiently than  $\mathbf{W}$  and  $\mathbf{F}$ . This is often true when the degree of redundancy ( $n-m$ ) is small.

An example of the control expression in the form of  $\mathbf{A}$  is implemented where  $\mathbf{W}$  is taken as the manipulator inertia matrix ( $\mathbf{M}$ ) to produce minimum-kinetic-energy control. Consider the manipulator dynamic model in the absence of external forces, given by the following [15]:

$$\boldsymbol{\tau} = \mathbf{M}(\mathbf{q})\ddot{\mathbf{q}} + \mathbf{C}(\mathbf{q}, \dot{\mathbf{q}})\dot{\mathbf{q}} + \mathbf{G}(\mathbf{q}). \quad (10-9)$$

To find  $\mathbf{N}_j^T \mathbf{M}$  for use in (10-6) with  $\mathbf{W} = \mathbf{M}$ , it follows from (10-9) that each column in null-space  $\mathbf{N}_j$  can be treated as a set of joint accelerations ( $\ddot{\mathbf{q}}$ ). The corresponding row in  $\mathbf{A}$ , then, is the joint torque ( $\boldsymbol{\tau}$ ) needed to achieve these joint accelerations ( $\ddot{\mathbf{q}}$ ), with joint velocity ( $\dot{\mathbf{q}}$ ) and gravity ( $\mathbf{G}(\mathbf{q})$ ) set to zero. This problem can be solved using the Newton-Euler dynamics algorithm, which goes through an outward recursion calculating the vector force and moment acting on each link, followed by an inward recursion to calculate the scalar force and torque needed for each joint. This is usually more efficient than calculating the inertia matrix ( $\mathbf{M}$ ), as Newton-Euler inverse dynamics is a linear-time operation (in the number of joints), while calculating the mass matrix is a quadratic-time operation. This algorithm needs to be performed  $n-m$  times, and therefore this new approach is especially efficient when the degree of redundancy ( $n-m$ ) is small. (Note this implementation of minimum-kinetic-energy control is included as part of the Actin™ Toolkit, discussed more below.)



The implicit use of  $\mathbf{F}$  in calculating  $\mathbf{B}$  is also an important improvement for many problems. An example is its use in finding directional derivatives. If  $\mathbf{F}$  is the gradient of a function ( $\nabla f$ ), then  $\mathbf{B}$  is just the directional derivatives of the function along the columns of  $\mathbf{N}_j$ . For the gradient case,  $\mathbf{F}$  can be evaluated as follows:

$$\nabla f = \left[ \frac{\partial f}{\partial x_1}, \frac{\partial f}{\partial x_2}, \dots, \frac{\partial f}{\partial x_n} \right]^T \quad (10-10)$$

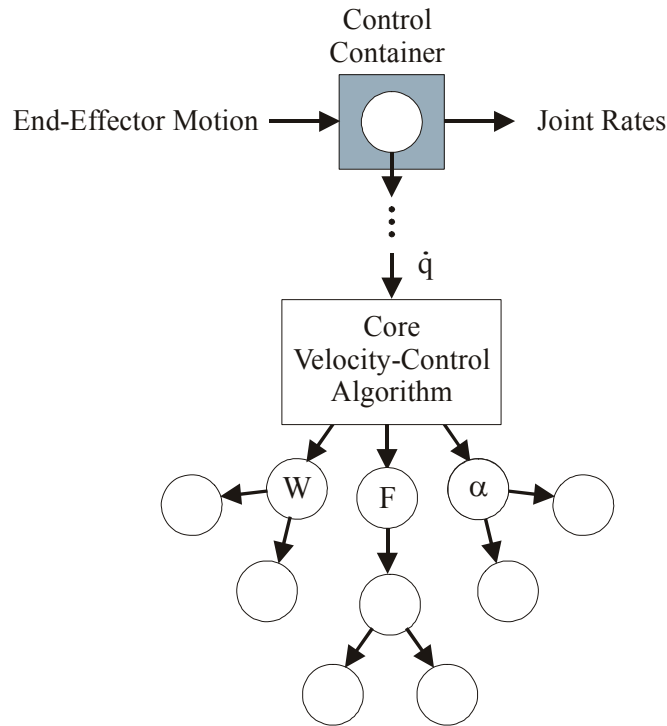
When using finite differencing, (10-10) involves evaluating the function  $f$   $n+1$  times. If reduced control calculation is used instead, the terms of  $\mathbf{N}_j^T \nabla f$  can be evaluated by directional derivative of the function  $f$  along the direction of the columns of  $\mathbf{N}_j$ . That is:

$$\mathbf{n}_i^T \nabla f = \lim_{h \rightarrow 0} \frac{f(\mathbf{x} + h\mathbf{n}_i) - f(\mathbf{x})}{h}, \quad (10-11)$$

where  $\mathbf{n}_i$  is the  $i^{\text{th}}$  column of  $\mathbf{N}_j$ . Approximating (10-11) with a small fixed value of  $h$  only takes  $(n-m+1)$  evaluations of the function  $f$ , saving  $m$  evaluations. In the case of a low degree of redundancy, the computation of  $\mathbf{n}_i^T \nabla f$  with this new approach is very efficient.

#### **10.1.4 End-Effector Error Filter**

The organization of the kinematic control system is as shown in the figure below. It is organized as a tree that is configured at runtime, with joint rates and other parameters passed from children to parents in the tree as matrices. The control system algorithm itself, the control system parameters, and all filters are nodes in this tree.



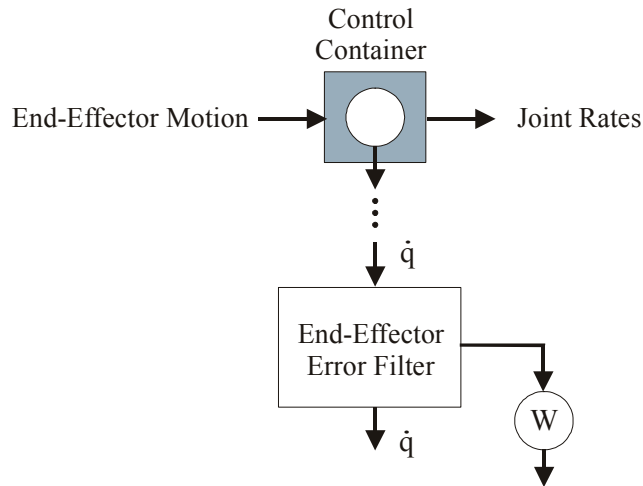
**Figure 10-2:** The kinematic control system is organized as a tree that is composed at runtime, configured using XML.

One component for this tree that is used for virtually all manipulators is an end-effector error filter, which prevents excessive end-effector error. If a measure of the difference between the desired and the actual end-effector velocities exceeds a threshold, then the manipulator is slowed or stopped. This measure,  $\mu$ , is calculated by taking the weighted inner product of the difference between desired and actual velocities, through

$$\mu = (\mathbf{V}_d - \mathbf{V}_a)^T \mathbf{W} (\mathbf{V}_d - \mathbf{V}_a), \quad (10-12)$$

where  $\mathbf{V}_d$  is the concatenation of desired end-effector velocities and  $\mathbf{V}_a$  is the concatenation of actual end-effector velocities.

The weighting matrix  $\mathbf{W}$  is established as a node in the control-tree description. This is illustrated in Figure 10-12.

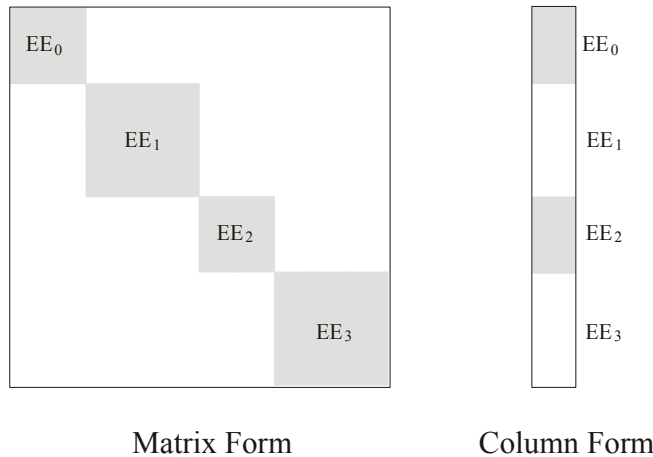


**Figure 10-3:** The end-effector error filter is one component that can be placed in the control tree. It reduces the speed of the manipulator as a function of the metric on end-effector error given by (10-12), which is a function of weighting matrix  $W$ .

This weighting matrix is implemented as a function of parameters set within each end-effector, including the following six end-effector types:

1. Frame (*EcFrameEndEffector*)
2. Point (*EcPointEndEffector*)
3. XY (*EcXyEndEffector*)
4. Orientation (*EcOrientationEndEffector*)
5. Center of Mass (*EcCenterOfMassEndEffector*)
6. Spatial Momentum (*EcSpatialMomentumEndEffector*)
7. Sliding (*EcSlidingEndEffector*)
8. Planar (*EcPlanarEndEffector*)
9. Look At (*EcLookAtEndEffector*)
10. Linear Constraint (*EcLinearConstraintEndEffector*)

Each of these now has configurable weights associated with it. These weights are in matrix form and are appended to build a general block-diagonal weighting matrix or, in special cases, a column of weights, as shown in the figure below.



**Figure 10-4** The weighting matrix  $\mathbf{W}$  can be calculated from the end effectors in one of two forms. It can either be an explicit matrix or a column that is treated as if it were a diagonal matrix. The column form is more efficient when strict-diagonal weighting is used.

Neither of these forms provides a way to weight cross terms formed by multiplying a component of error for one end effector with a component for another. For this,  $\mathbf{W}$  can be explicitly set as a matrix through the XML description. (The downside of setting  $\mathbf{W}$  explicitly being that end effectors cannot be added and removed individually at runtime.)

## 10.2 Implementation

In the Actin™ Toolkit, equations (10-3), (10-5), and (10-8) are implemented using a tree structure. This tree structure exists in the C++ code and is defined using XML. It is described in detail in this section.

### 10.2.1 Velocity Control System

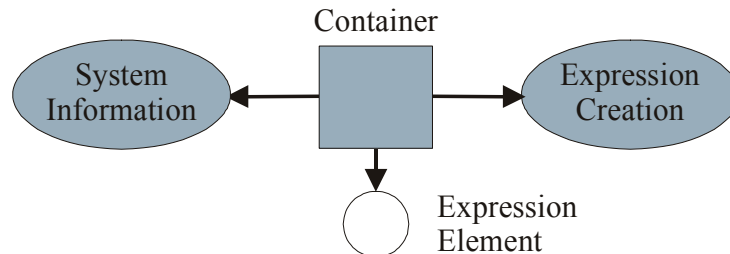
Large, fielded control systems are typically complex, with many patches and logical paths added by the development team. Though based on logically encapsulated equations from textbooks and academic papers, fielded control systems transcend them. The Actin™ Toolkit embraces this phenomenon by organizing the control system, the robust extension, and all supporting mathematics into a flexible logical tree that can be represented in XML.

This logical tree can better represent the additions and modifications that otherwise might be informally added to the control system. It also provides a method of organization that supports dynamic programming—the storage of subproblem solutions to prevent duplicate calculation. This dynamic-programming/logical-tree approach to velocity control is among the most important components of the Actin™ Toolkit. Though the algorithmic framework in (10-3) is central to its success, through the chosen software architecture, the control system is bigger than equation (10-3). It is a combination of flexible software with a powerful algorithm.

The logical tree is built from two types of entities: containers and expression elements. These two families of objects are described through XML and are connected dynamically in code to produce unique functions that give joint rates for any manipulator state and desired hand motion. Containers and elements are described below.

### 10.2.1.1 Containers

A control expression container performs three tasks. It 1) holds a single expression element, which is the root of a tree, 2) knows how to create any type of expression element that may lie in the tree beneath it, and 3) provides access to system information. This is illustrated in the figure below.



**Figure 10-5:** The three roles of the container. Each container holds a single expression element, a pointer to system information, and the ability to create any type of expression element in the tree below it from a string token.

### 10.2.1.2 Expression Elements

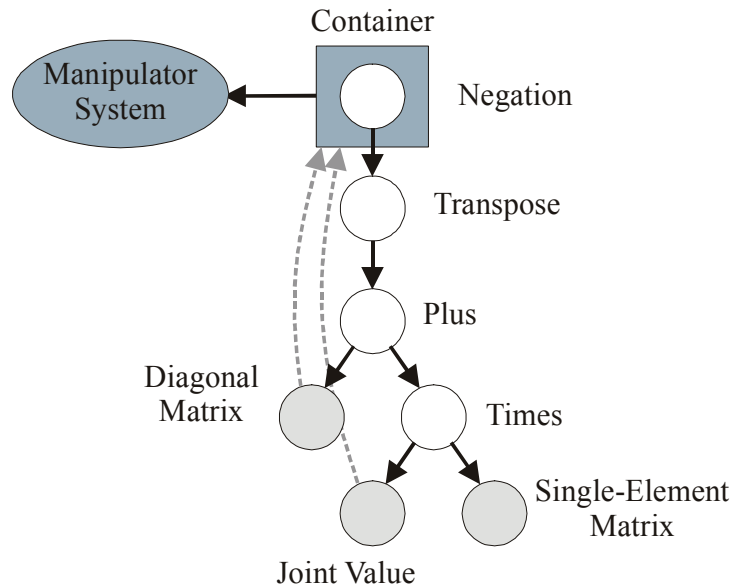
Expression elements are the building blocks of the control system tree. All expression elements have the following in common:

- Every expression element returns a two-dimensional array of values when queried.
- Every expression element holds a pointer to the top-most container in the tree.
- Using the container pointer, every expression element can read and write its description (including the tree below it) from and to an XML stream.

There are two categories of expression elements: branch and leaf elements. Branch elements have children that are expression elements. Unary branch elements have a single child, binary branch elements have two children, and multiary branch elements have more than two children. Leaf elements have no children. The control expression tree is formed using branch elements terminated by leaf elements. In the C++ code, each type of expression element is represented using a different class. In the XML description, each expression element uses a unique string identifier of its type.

The most basic expression elements are simple mathematical operations, such as multiplication and addition. All operations are supported on two-dimensional arrays in an intuitive manner. Addition is performed element-by-element and multiplication is performed as matrix multiplication. When the dimensions of quantities do not agree, the operation is performed on the maximum compatible subset. The figure below illustrates a simple expression tree that returns an array that is a function of a manipulator joint value.

The memory management in the tree is self organizing. Standard Template Library (STL) data structures are allocated from the heap based on the size of values returned from child expression elements. In so doing, as long as the children return the same size data (the typical case), no memory is allocated on subsequent use of the tree. This allows fast operation of the control system. Despite very high flexibility, there is no dynamic memory allocation at run time.



**Figure 10-6:** An example expression tree. A container holds the top element, which is the root of a tree that is the root of a tree composed of branch (white) and leaf (gray) elements. All elements hold a pointer to the container (only two are illustrated). For programming, the container object holds a map of functions to create any type of expression element from a string token.

### 10.2.1.2.1 Basic Mathematical and Logical Control Tree Elements

There are two types of expressions used in the expression tree (as shown in Figure 10-6), those that return values that are pure functions of the returned values from the tree below and those that analyze the robotic system to calculate their return values. The former, which can be leaf or branch nodes, are described in the two tables below.

Type	Class	Meaning
And	<i>EcExpressionAnd</i>	Logical <i>and</i> (element by element)
Cosine	<i>EcExpressionCosine</i>	cos(x) (element by element)
Element Inverse	<i>EcExpressionElementInverse</i>	1/x (element by element)
Greater Than	<i>EcExpressionGreaterThan</i>	> (element by element)
Greater Than Or Equal To	<i>EcExpressionGreaterThanOrEqualTo</i>	≥ (element by element)
Less Than	<i>EcExpressionLessThan</i>	< (element by element)
Less Than Or Equal To	<i>EcExpressionLessThanOrEqualTo</i>	≤ (element by element)

Log	<i>EcExpressionElementLog</i>	Natural log (element by element)
LogN	<i>EcExpressionElementLogN</i>	Arbitrary log (Element by Element)
Minus	<i>EcExpressionMinus</i>	Subtraction
Negative	<i>EcExpressionNegative</i>	-x
Or	<i>EcExpressionOr</i>	Logical <i>or</i> (element by element)
Plus	<i>EcExpressionPlus</i>	Addition
Pow	<i>EcExpressionElementPow</i>	$a^x$ (for each element)
Root	<i>EcExpressionElementRoot</i>	$\sqrt[x]{a}$ (for each element)
Sine	<i>EcExpressionSine</i>	sin(x) (element by element)
Times	<i>EcExpressionTimes</i>	Matrix multiplication
Transpose	<i>EcExpressionTranspose</i>	$(\cdot)^T$ – Matrix transpose

**Table 10-1:** Basic branch elements, their C++ class, and their meaning.

Type	Class	Meaning
Constant	<i>EcExpressionScalarConstant</i>	Constant floating-point
Single Element Column	<i>EcExpressionSingleElementColumn</i>	Column with one nonzero value
General Column	<i>EcExpressionGeneralColumn</i>	Column with arbitrary values
Single Element Matrix	<i>EcExpressionSingleElementMatrix</i>	Matrix with one nonzero value
Identity Matrix	<i>EcExpressionIdentityMatrix</i>	The identity matrix
Diagonal Matrix	<i>EcExpressionDiagonalMatrix</i>	Matrix with arbitrary diagonal

**Table 10-2:** Basic leaf elements, their C++ class, and their meaning.

These basic mathematical operations enable the user to specify many different matrix, vector, and scalar functions within the control tree.

Note these expression elements are not limited to data from the expression tree. Some elements use data that is contained within its own XML description. That is, the value returned by any expression element is a function both of the expression tree below it and of its own parameters. For example, a diagonal-matrix expression loads its diagonal entries directly from the XML description, not from the expression tree.

### 10.2.1.2.2 System-Aware Control Tree Leaf Elements

There are also a large number of predefined matrix, vector, and scalar functions of the state of the robotic system. These leaf elements are shown in the table below.

Type	Class	Meaning
Accuracy Measure Gradient	<i>EcControlExpression...</i> <i>ErrorSensitivity</i>	The gradient of the error sensitivity for a single joint.
Collision Avoidance	<i>EcControlExpression...</i> <i>CollisionAvoidanceAB</i>	A set of joint rates that can be used to drive the manipulator away from self collision and collision with other manipulators.
Error Reduction	<i>EcControlExpression...</i> <i>ErrorReduction</i>	The gradient of an error of sensitivity to a statistical distribution of joint errors.
Joint Limit Avoidance	<i>EcControlExpression...</i> <i>JointLimitAvoidance</i>	A column of joint rates that can be used to drive a manipulator away from joint limits
JointTorqueSquared Gradient	<i>EcControlExpression...</i> <i>GravitationalTorqueGradient</i>	The gradient of the gravitational torque squared on a single joint.
Mass Matrix	<i>EcControlExpression...</i> <i>MassMatrix</i>	The manipulator inertia matrix.
Mass Matrix AB	<i>EcControlExpression...</i> <i>MassMatrixAB</i>	The manipulator null space basis multiplied by the inertia matrix.
Obstacle Avoidance	<i>EcControlExpression...</i> <i>ObstacleAvoidance</i>	A column of joint rates that can be used to drive the arm away from environmental obstacles.
Obstacle Avoidance AB	<i>EcControlExpression...</i> <i>ObstacleAvoidanceAB</i>	The manipulator null space basis times a column of joint rates that can be used to drive the arm away from environmental obstacles.



Potential Energy Gradient	<i>EcControlExpression... PotentialEnergyGradient</i>	The gradient of the potential energy.
Singularity Avoidance	<i>EcControlExpression... SingularityAvoidance</i>	The gradient of a measure of singularity proximity.
Strength Optimization	<i>EcControlExpression... StrengthOptimization</i>	The gradient of a measure of strength along a specified direction.
Joint Value	<i>EcControlExpression... JointValue</i>	The value for any joint on any manipulator.
Table Function	<i>EcControlExpression... ColumnTableFunction</i>	An arbitrary, data-specified table function, with any number of domain dimensions and any number of range dimensions.

**Table 10-3:** Leaf Elements that are system aware, their C++ class, and their meaning.

### 10.2.1.2.3 Core Velocity-Control Elements and Converters

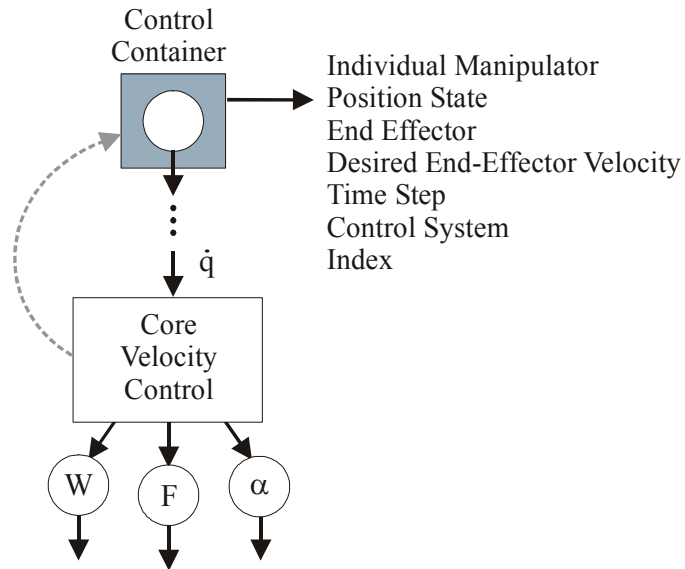
The expressions in Table 10-3 are used to build the parameter inputs to the velocity control algorithms. The velocity control algorithms themselves are implemented as elements in the control tree as well. There are two velocity-control elements: 1) a core expression, which implements equation (10-3) and 2) an AB core expression, which implements (10-8). There are also elements for converting **W** and **F** parameters to **A** and **B** parameters, as defined through equations (10-6) and (10-7). These expressions are summarized in the table below.

Type	Class	Meaning
Core	<i>EcControlExpressionCore</i>	The core algorithm defined through (10-3).
AB Core	<i>EcControlExpressionABCore</i>	Implementation of the reduced core algorithm defined through (10-8).
AB Matrix Converter	<i>EcControlExpression... MatrixToAB</i>	Converts a weighting matrix to an <b>A</b> parameter through (10-6).
AB Vector Converter	<i>EcControlExpression... VectorToAB</i>	Converts a weighting vector to a <b>B</b> parameter through (10-7).

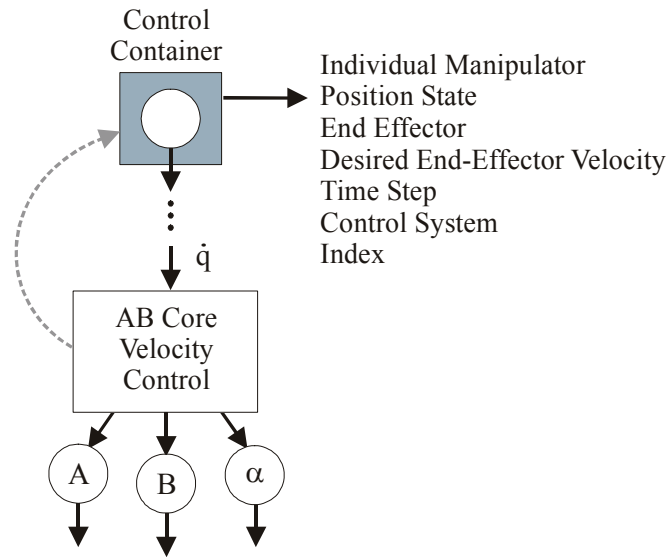
**Table 10-4:** Core elements and converters, their C++ class, and their meaning.

The core control systems benefit from the flexibility of the expression tree in several ways. The matrix, vector, and scalar functions that are used in equations (10-3) and (10-8) are described using the expression tree. Also, the output joint rates can be modified in implementing  $\beta(\mathbf{q}, \mathbf{V})$ , the robustness function, just as any other expression in the tree.

All the expressions in Table 10-4 require information on the manipulator system, end effectors, and desired end-effector motion that it accesses through the top-level container. This is illustrated, along with the operation of the expressions in the figures below.

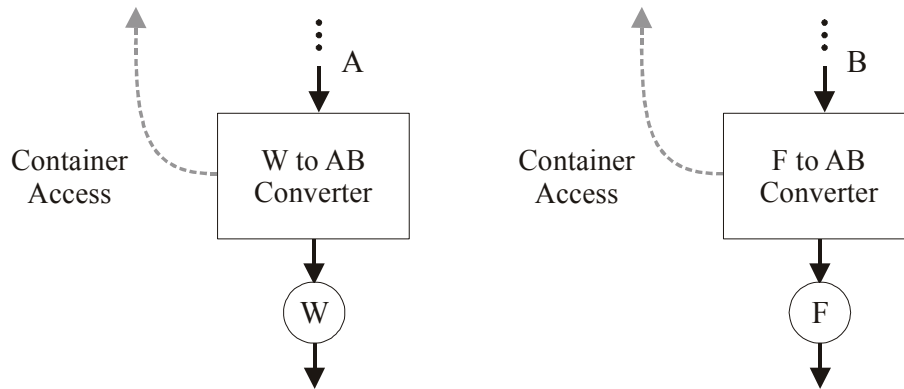


**Figure 10-7:** The core velocity control system (*EcControlExpressionCore*). It has three children in the control tree—one each for the matrix, vector, and scalar used in equation (10-3). This is used with system information provided through the top-level container to calculate joint rates.



**Figure 10-8:** The AB core velocity control system (*EcControlExpressionABCore*). It also has three children in the control tree—one each for the **A** parameter, the **B** parameter, and scalar used in equation (10-8). A typical control tree will contain one of either the core shown in Figure 10-7 or the AB core shown here.

Expressions that provide the **W** and **F** inputs to an *EcControlExpressionCore* object can be converted into the **A** and **B** inputs to the *EcControlExpressionABCore* object using the converters shown in the figure below. With these, it is always possible to use the AB core. However, when both **W** and **F** are explicitly calculated, it is generally more efficient to use the regular core. The AB core can be very efficient when **A** or **B** are calculated directly.



**Figure 10-9:** *EcControlExpressionMatrixToAB* (left) and *EcControlExpressionVectorToAB* (right) can convert matrix weights to **A** and **B** parameters, using the formulas in equations (10-6) and (10-7).

#### 10.2.1.2.4 Filters

The robust extension to the core velocity-control algorithm is implemented using several types of expression elements. One type restricts the weighted infinity norm of joint rates. (The infinity norm

returns the absolute value of the largest-magnitude element in the vector.) Let  $\mathbf{W}_J$  be a weighting matrix on the joint rates and  $t_J$  be a threshold, then the following defines  $\beta_J(\dot{\mathbf{q}}, \mathbf{q}, \mathbf{V})$ :

$$\begin{aligned} \beta_J(\dot{\mathbf{q}}, \mathbf{q}, \mathbf{V}) &= 1, & \|\mathbf{W}_J \dot{\mathbf{q}}\|_\infty < t_J \\ &= \frac{t_J}{\|\mathbf{W}_J \dot{\mathbf{q}}\|_\infty}, & \text{otherwise} \end{aligned} \quad (10-13)$$

Another is an end-effector error metric, a weighted two-norm on the difference between the expected and the actual end-effector motion. Let  $\mathbf{W}_E$  be a weighting matrix on the end-effector error, and  $t_E$  be a threshold. Then let

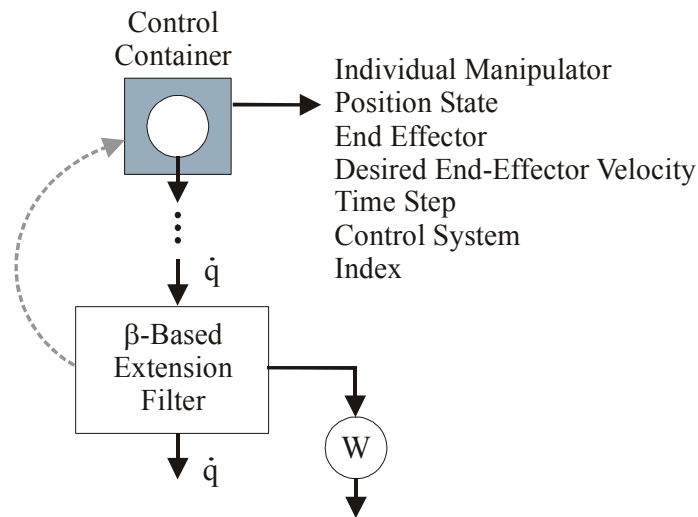
$$\mu_E(\mathbf{V}_0, \mathbf{V}_1) = \sqrt{(\mathbf{V}_1 - \mathbf{V}_0)^T \mathbf{W}_E (\mathbf{V}_1 - \mathbf{V}_0)} \quad (10-14)$$

measure the error between the expected and the actual end-effector velocity. With this,  $\beta_E(\dot{\mathbf{q}}, \mathbf{q}, \mathbf{V})$  is defined as follows:

$$\begin{aligned} \beta_E(\dot{\mathbf{q}}, \mathbf{q}, \mathbf{V}) &= 1, & \mu_E(\mathbf{V}_0, \mathbf{V}_1) < t_E \\ &= \frac{t_E}{\mu_E(\mathbf{V}_0, \mathbf{V}_1)}, & \text{otherwise} \end{aligned} \quad (10-15)$$

In addition, to support absolute stopping where appropriate (such as at workspace boundaries), a check is made for each manipulator to see if its direction of motion is correct. If a hand is actually moving away from the desired direction (where *away* is defined using  $\mathbf{W}_E$ ), the arm is stopped.

Another filter simulates one step ahead and stops the arm if a joint limit or other restriction is encountered. And yet another measures the ratio of end-effector motion to joint-rate motion and slows the manipulator accordingly. Multiple filters can be daisy-chained to give flexible filtering. The form of the filters is shown in the figure below.



**Figure 10-10:** The form of the robust extension elements. A joint-rate filter is shown here, which implements equation (10-13). The weighting matrix  $W$  is used to measure the incoming joint rates and produce a robust output (which can be the input to another filter).

The filters that are available in the Actin™ Toolkit are shown in the table below.

Type	Class	Meaning
Joint Rate Filter	<i>EcControlExpression...</i> <i>JointRateFilter</i>	Limits joint rates. Implements equation (10-13).
End-Effector Error Filter	<i>EcControlExpression...</i> <i>EndEffectorErrorFilter</i>	Limits end-effector error using equation (10-15).
End-Effector Motion Filter	<i>EcControlExpression...</i> <i>EndEffectorMotionFilter</i>	Limits the ratio of a measure of joint rates to hand motion.
Simulation Filter	<i>EcControlExpression...</i> <i>SimulationFilter</i>	Simulates one step ahead and recovers from problems.

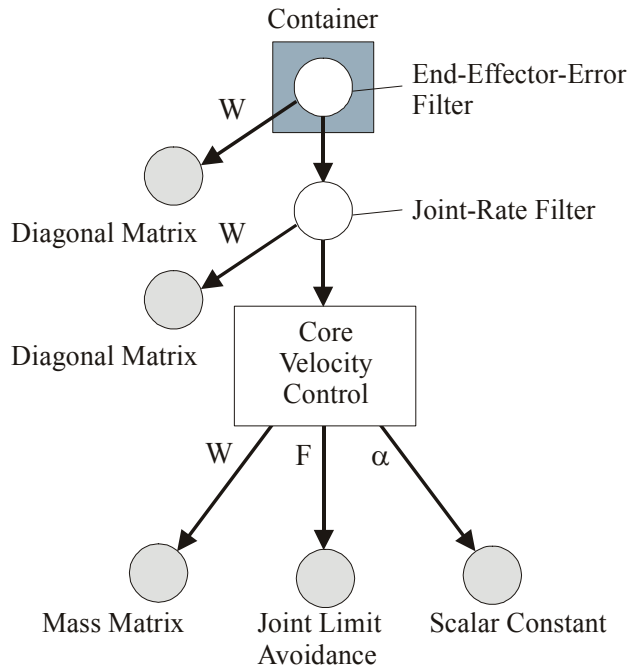
**Table 10-5** Filter expression elements, their C++ class, and their meaning.

### 10.2.1.3 Example

To create a velocity control tree, the following steps should be taken:

- 1) Either a basic core or an AB core should be chosen to implement the velocity control system.
- 2) A matrix  $W$ , vector  $F$ , and scalar  $\alpha$  should be chosen to represent the desired control method, which will minimize  $\frac{1}{2} \dot{\mathbf{q}}^T W \dot{\mathbf{q}} + \alpha F^T \dot{\mathbf{q}}$ .
- 3) An algorithm should be chosen to implement a  $\beta(\mathbf{q}, \mathbf{V})$  that gives robust behavior.
- 4) If the basic core is used, expressions must be constructed to give the desired  $W$ ,  $F$ , and  $\alpha$ . If the AB core is used, expressions must be constructed giving  $A = N_J^T W$ ,  $B = N_J^T F$ , and  $\alpha$ .
- 5) A filter must be constructed to give the desired value of  $\beta(\mathbf{q}, \mathbf{V})$ .

If as an answer to step 1), the standard core is used; as an answer to step 2),  $F$  is chosen to move away from joint limits and  $W$  is chosen to minimize kinetic energy, and  $\alpha$  is chosen to make a simple tradeoff; and as an answer to 3) a union of joint-rate filtering and end-effector-error filtering is chosen, then the resulting control tree is shown below. This control system will robustly avoid joint limits and when distant from limits will minimize kinetic energy.



**Figure 10-11:** An example control tree that would avoid joint limits and minimize kinetic energy when not operating near limits. There are eight control expressions and one container. The column vector of joint rates returned by the joint-rate filter at the top of the tree would drive the manipulator.

An example XML description of the control tree shown in Figure 10-11 is shown in the figure below. This system is created in code as part of the quick-start description, in Text Box 2-3.

```

- <ct:controlExpressionContainer>
- <ct:endEffectorErrorFilter>
  <ct:stopsAtLimits>1</ct:stopsAtLimits>
  <ct:threshold>1</ct:threshold>
- <ct:unfilteredRates>
- <ct:jointRateFilter>
  <ct:threshold>1</ct:threshold>
- <ct:unfilteredRates>
- <ct:controlCore>
- <ct:matrix>
  <ct:massMatrix />
</ct:matrix>
- <ct:scalar>
  <ct:scalarConstant>-0.5</ct:scalarConstant>
</ct:scalar>
- <ct:vector>
- <ct:jointLimitAvoidance>
  <ct:avoidanceZone>0.5</ct:avoidanceZone>
  <ct:exponent>3</ct:exponent>
  <ct:maximum>50</ct:maximum>
</ct:jointLimitAvoidance>
</ct:vector>
</ct:controlCore>
</ct:unfilteredRates>
- <ct:weights>
- <ct:diagonalMatrix>
  <ct:columnSize>6</ct:columnSize>
- <ct:diagonal size="6">
  <ct:group>0.1 0.1 0.1 0.1 0.1 0.1</ct:group>
</ct:diagonal>
  <ct:rowSize>6</ct:rowSize>
</ct:diagonalMatrix>
</ct:weights>
</ct:jointRateFilter>
</ct:unfilteredRates>
- <ct:weights>
- <ct:generalColumn>
- <ct:column size="6">
  <ct:group>10 10 10 10 10 10</ct:group>
</ct:column>
</ct:generalColumn>
</ct:weights>
</ct:endEffectorErrorFilter>
</ct:controlExpressionContainer>

```

**Figure 10-12:** An XML description of a control tree that would avoid joint limits and minimize kinetic energy when not operating near limits, as illustrated in Figure 10-11 above. There are eight control expressions and one container.

### 10.3 Velocity Control Types

The control system is flexible and able to implement a wide variety of algorithms. Included with the toolkit are all the parameters defined through Table 10-3. This section describes the details behind these. Control types include singularity avoidance, torque minimization, obstacle avoidance, fault

tolerance, minimum-kinetic-energy control, minimum-potential-energy control, accuracy optimization, and joint-limit avoidance.

### 10.3.1 Singularity Avoidance

A finite-differencing tool was developed for singularity avoidance. This takes a pointer to function object, and uses this function to numerically calculate its gradient with respect to joint variables using the finite difference method. This is a very flexible and powerful, though somewhat costly approach. Generally, it is better to explicitly calculate the gradient whenever possible. Since the explicit calculation is not always possible, finite differencing is used for singularity avoidance.

A manipulator experiences a kinematic singularity whenever the Jacobian loses rank. To frame the desire to avoid singularities into a solution method, a function is needed that is large at singularities and small away from singularities. For this, the damped inverse of the product of the singular values of a weighted Jacobian is used. That is, the optimization function is the following:

$$f(\mathbf{q}) = \frac{1}{\bar{\sigma}_1 \bar{\sigma}_2 \cdots \bar{\sigma}_n + \varepsilon}. \quad (10-16)$$

where  $\varepsilon$  is a damping factor and  $\bar{\sigma}_i$  is singular value  $i$  of the weighted Jacobian  $\mathbf{J}_w$ :

$$\mathbf{J}_w = \mathbf{D}_T \mathbf{J} \mathbf{D}_J, \quad (10-17)$$

where  $\mathbf{D}_T$  and  $\mathbf{D}_J$  are diagonal matrices. The weighting matrices are necessary to meaningfully define the singular values of the Jacobian.

The singular values are not explicitly calculated, as this is too costly. Instead the Cholesky decomposition of  $\mathbf{J}_w \mathbf{J}_w^T$  is taken, and the product of the diagonal terms is used. The product of these diagonal terms equals the product of the singular values.

The gradient of the function given in (10-16) is numerically calculated for use with the basic core through the class *EcControlExpressionSingularityAvoidance*. For use with the AB core, it can be combined with *EcControlExpressionVectorToAB*. These can be used with a positive definite weighting matrix and a positive scalar to drive the manipulator to minimize the function and therefore move away from kinematic singularities.

### 10.3.2 Torque Minimization

For torque minimization, the function to be optimized is defined as the following

$$f(\mathbf{q}) = g_i^2(\mathbf{q}), \quad (10-18)$$

the square of the gravitational torque or force on joint  $i$ . To calculate the gradient, an explicit gradient calculation method (i.e., finite differencing is not used for this) for gravitational joint torque/force based on composite rigid body inertia is used. Using this calculation method, the gradient of the torque squared is given by

$$\nabla f(\mathbf{q}) = 2g_i(\mathbf{q})\nabla g_i(\mathbf{q}). \quad (10-19)$$

This is implemented in class *EcControlExpressionGravitationalTorqueGradient* for use with the basic core. For use with the AB core, it can be combined with *EcControlExpressionVectorToAB*.



Using this value as the vector input to the core control system with a positive definite weighting matrix and a positive scalar produces control that minimized the magnitude of the gravitational torque on joint  $i$ . This can be used to prevent stress on a particular joint, or it could be used for failure recovery after a free-swinging failure [16]

### 10.3.3 Collision Avoidance

High fidelity collision avoidance is a key component of the Actin™ Toolkit. It is important enough to merit its own chapter later in this document.

### 10.3.4 Minimum Kinetic Energy Control

To minimize kinetic energy, the matrix parameter to the control system is set to be the manipulator mass matrix, and the scalar parameter is set to zero. That is,

$$\mathbf{W} = \mathbf{M}(\mathbf{q}), \quad (10-20)$$

$$\alpha = 0. \quad (10-21)$$

In this case, when  $\alpha = 0$ , the vector parameter is not relevant. There are two classes for implementing this. For use with the basic core, *EcControlExpressionMassMatrix* returns the manipulator mass matrix— $\mathbf{M}(\mathbf{q})$  in equation (10-9)—directly for use as weighting parameter  $\mathbf{W}$ . The manipulator mass matrix is calculated using the method of Composite Rigid-Body Inertia (developed in [15] and named in [17]), which is extended to apply to bifurcating arms. The same algorithm is used to calculate the mass matrix for minimum-kinetic-energy control as is used to calculate the composite-rigid-body dynamics. For use with the AB core, an  $\mathbf{A}$  parameter corresponding to the manipulator mass matrix as  $\mathbf{W}$  is calculated by treating the Jacobian-null-space basis vectors as accelerations and calculating the corresponding joint torques. For this the iterative Newton Euler algorithm is used [18].

### 10.3.5 Minimum Potential Energy Control

To minimize the potential energy of the arm, the gradient of the potential energy is used. The gradient of potential energy is in fact equal to the vector of gravitational joint torques— $\mathbf{G}(\mathbf{q})$  in equation (10-9). To calculate the gravitational joint torques (forces for prismatic joints) the composite rigid-body inertia of the links is used. This method is implemented for the basic core in the class *EcControlExpressionPotentialEnergyGradient*.

### 10.3.6 Accuracy Optimization

For accuracy optimization in the presence of position errors in a single joint, an  $\mathbf{F}$  value is provided for the basic core through the class *EcControlExpressionErrorSensitivity*. For this calculation, a weight  $w_i$  is assigned to each of  $E$  end-effectors on the manipulator, and a focus joint,  $j$ , is identified. Then an error sensitivity function is defined as follows:

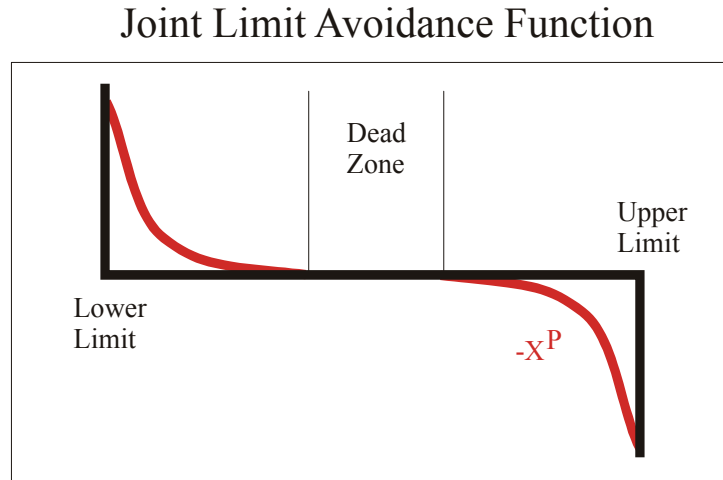
$$f(\mathbf{q}) = \sum_{i=1}^E w_i \left\| \mathcal{J}_j \mathbf{v}_i \right\|^2, \quad (10-22)$$

where  $\mathcal{J}_j \mathbf{v}_i$  is the velocity of end-effector  $i$  due to unit motion of joint  $j$ .

The gradient of this function is then found using finite differencing, and this is used as the vector parameter for the core velocity control system. With the matrix parameter being a diagonal matrix and the scalar parameter a positive value, this serves to minimize the sensitivity to errors.

### 10.3.7 Joint-Limit Avoidance

For joint-limit avoidance, for each joint, a rate term is independently calculated that is a polynomial function of the proximity to a limit. This is illustrated in Figure 10-12. Class *EcControlExpressionJointLimitAvoidance* implements this weighting function. This is used with a negative scalar value to drive the manipulator away from joint limits.



**Figure 10-13:** The vector elements for joint-limit avoidance are set using polynomials with a user-configurable exponent and dead-zone.

### 10.3.8 Strength Optimization

Strength optimization is achieved by employing a technique based on the strength formulation in [19]. In this discussion, torque is used in the general sense, meaning force for a sliding joint. For an arbitrary set of end-effector forces, a vector of joint torques can be found. The goal of strength optimization is to adjust the manipulator joint positions in such a way as to minimize the normalized joint torques resulting from the imposed forces while simultaneously placing the hand(s). This is accomplished by directing the joints toward the direction that yields the maximum reactive strength in the direction opposing the applied forces. Strength here is defined as the maximum force or moment that the manipulator can exert on its environment at the point of resolution of an end effector.

The basic formulation of velocity control used to relate joint rates to hand velocity is given by equation (10-1). Again,  $\mathbf{V}$  is a concatenation of hand velocities,  $\dot{\mathbf{q}}$  is the vector of joint rates and  $\mathbf{J}(\mathbf{q})$  is the manipulator Jacobian.

If friction and gravitational loads are neglected, the power flow at the joints must equal power flow at the end effector. By relating hand velocities  $\mathbf{V}$  and corresponding wrench loads  $\mathbf{F}$  with joint rates  $\dot{\mathbf{q}}$  and joint torques  $\mathbf{T}$ , this property is expressed as follows:

$$\mathbf{V}^T \mathbf{F} = -\dot{\mathbf{q}}^T \mathbf{T} \quad (10-23)$$

Using (10-1) with the fact that (10-23) must be true for all values of  $\dot{\mathbf{q}}$  yields

$$\mathbf{T} = -\mathbf{J}(\mathbf{q})^T \mathbf{F} \quad (10-24)$$

Equation (10-24) relates wrench loads exerted on the system with joint torques required to prevent manipulator motion due to those loads.

When assessing the total strength of a manipulator, torque capacities for each joint must be known. The torque capacity is typically limited by the motor's ability to supply torque or the gearhead's ability to transmit torque to the joint. The torque capacity  $\hat{T}$  for a given joint can be expressed as

$$\hat{T} = \min(t_a G, t_t) \quad (10-25)$$

Where  $t_a$  is the actuator (corresponding to motor) torque,  $G$  is the gear ratio (the ratio of actuator velocity to joint velocity—typically  $G$  is greater than unity) and  $t_t$  is the transmission torque. The torque capacity for a joint may differ depending on direction.

To optimize manipulator configuration for strength, let  $\mathbf{x}$  be a desired end-effector force, as would be used with  $\mathbf{F}=\mathbf{x}$  in (10-24). Let  $\hat{T}_{\max_j}$  be the maximum torque capacity, and  $\hat{T}_{\min_j}$  be the minimum torque capacity (the largest magnitude in the negative direction) for joint  $j$ . For an  $n$ -joint system, a measure of strength  $f(\mathbf{q})$  is given by

$$f(\mathbf{q}) = \sum_{j=0}^{n-1} \left[ \frac{|T_j - \mu_j|}{\|\hat{T}_j\|} \right]^r, \quad (10-26)$$

where  $T_j$  is the actual torque required to achieve  $\mathbf{F}=\mathbf{x}$  using (10-24),  $\mu_j$  is the mean of the torque capacity of joint  $j$ , given by:

$$\mu_j = \frac{\hat{T}_{\max_j} + \hat{T}_{\min_j}}{2}, \quad (10-27)$$

and  $\|\hat{T}_j\|$  is the range of torque capacity for joint  $j$ , given by:

$$\|\hat{T}_j\| = \hat{T}_{\max_j} - \hat{T}_{\min_j}. \quad (10-28)$$

The user-defined parameter  $r$  specifies the order of the function to optimize. Note that  $r=2$  corresponds to a sum of squares approach and  $r \rightarrow \infty$  approaches an infinity-norm approach, where the strength is defined only by the weakest joint.

The gradient of  $f(\mathbf{q})$  is found using finite differencing, forcing the joints in the direction of minimum normalized torque and, hence, maximum strength. This gradient is calculated in the expression tree using the class *EcControlExpressionStrengthOptimization*. Because it is a gradient, it is for direct use with the basic core velocity control system. It can be converted to a  $\mathbf{B}$  parameter for use with an AB core by combining it with an *EcControlExpressionVectorToAB* expression.

### 10.3.9 Statistical Error Reduction

For accuracy optimization in the presence of high frequency noise on multiple joints, an approach is used that exploits the statistical properties of the joint noise as well as the Euclidean Space nature of the tools the manipulator is using.

#### 10.3.9.1 The Function for Optimization

Let the velocity error in the manipulator's joints,  $\dot{\mathbf{q}}_e$ , be a random variable having covariance  $\mathbf{C}_e$ . That is, the expected value of the quadratic form is given by

$$E[\dot{\mathbf{q}}_e \cdot \dot{\mathbf{q}}_e^T] = \mathbf{C}_e \quad (10-29)$$

It is assumed that the controller prevents low-frequency drift to enforce  $E(\dot{\mathbf{q}}_e) = \mathbf{0}$ . It is desired to minimize the expected value of a quadratic measure of the hand velocities due to this error. Let the error in the hand velocities be  $\mathbf{V}_e$ , now also a random variable. From (10-1), the relationship between  $\dot{\mathbf{q}}_e$  and  $\mathbf{V}_e$  is

$$\mathbf{V}_e = \mathbf{J}(\mathbf{q})\dot{\mathbf{q}}_e. \quad (10-30)$$

Let a measure of the hand error be defined as

$$\mu_e = \mathbf{V}_e^T \mathbf{A} \mathbf{V}_e, \quad (10-31)$$

for some prescribed constant matrix  $\mathbf{A}$ . Note  $\mu_e$  is a random variable. The goal, then, is to minimize the expected value of  $\mu_e$ . That is, the function to be minimized is given by

$$f(\mathbf{q}) = E[\mu_e] \quad (10-32)$$

Combining the three above equations gives

$$f(\mathbf{q}) = E[\dot{\mathbf{q}}_e^T \mathbf{J}^T \mathbf{A} \mathbf{J} \dot{\mathbf{q}}_e]. \quad (10-33)$$

To put this in a form that can be used for optimization, the following formula is used, which applies for any column vector  $\mathbf{x}$  and any matrix  $\mathbf{M}$ ,

$$\mathbf{x}^T \mathbf{M} \mathbf{x} = \text{tr}(\mathbf{M} \mathbf{x} \mathbf{x}^T). \quad (10-34)$$

This gives

$$f(\mathbf{q}) = E[\text{tr}(\mathbf{J}^T \mathbf{A} \mathbf{J} \dot{\mathbf{q}}_e \dot{\mathbf{q}}_e^T)]. \quad (10-35)$$

Here,  $\text{tr}(\cdot)$  is the sum of the diagonal entries (the trace operator). Using  $E(\dot{\mathbf{q}}_e) = \mathbf{0}$ , this gives

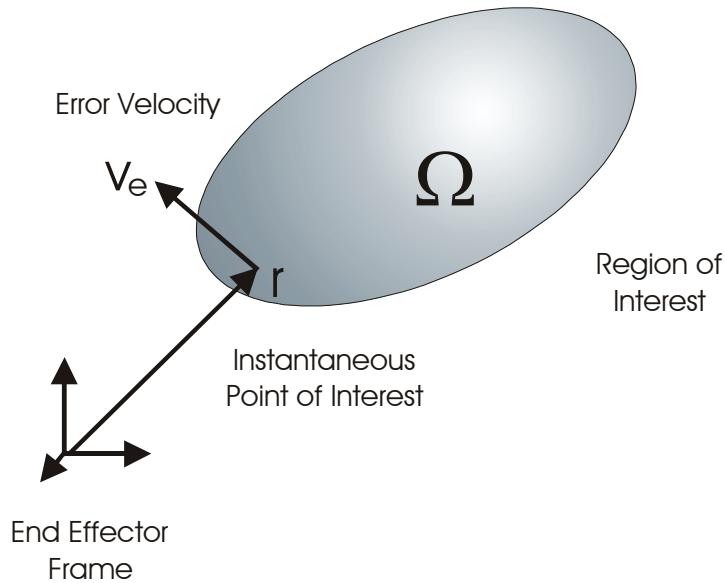
$$f(\mathbf{q}) = \text{tr}(\mathbf{J}^T \mathbf{A} \mathbf{J} \mathbf{C}_e), \quad (10-36)$$

a straightforward function of configuration.

### 10.3.9.2 Evaluating the A Matrix

The matrix  $\mathbf{A}$  as used above can be set in a number of ways. One particularly powerful approach is to use Euclidean-space regions of interest rigidly attached to the hands. Such a region might be the physical extent of a tool, for example.

Let the Euclidean-space region be labeled  $\Omega$ . Then  $\mu_e$  can be cast to minimize an integral of the velocity error squared over  $\Omega$ . Let  $\vec{r}$  be a point in  $\Omega$ , and let  $\rho(\vec{r})$  be a nonnegative interest density function defined over  $\Omega$  (specifying, for example, that the tip of a screwdriver is more important than the shaft). Let  $\vec{v}_e(\vec{r})$  be defined as the error velocity at point  $\vec{r}$ , a random variable. These are illustrated in the figure below.



**Figure 10-14:** Illustration of the terms used to measure the statistical error of a Euclidean Space region. The region is rigidly attached to one end-effector frame.

With this, what is desired is to minimize

$$f(\mathbf{q}) = E \left[ \int_{\Omega} \rho(\vec{r}) \|\vec{v}_e(\vec{r})\|^2 d\omega \right]. \quad (10-37)$$

That is, the expected value of a weighted integral of the velocity error squared of all the points in the region of interest. This is a comprehensive and intuitive measure.

It is not practical to calculate the integral shown in (10-37) in real time. However, there is a way to cast it into the form of (10-31), which will allow (10-36) to be used with no on-line integration. To do this, an analogy is used between (10-37) and kinetic energy of a rigid body. If  $\vec{v}_e(\vec{r})$  were the velocity over the region  $\Omega$  and  $\rho(\vec{r})$  were the mass, then the integral in (10-37) would give twice the kinetic energy of the body.

Let the following be defined in analogy to mass, first moment of inertia, and second moment of inertia:

$$m_{\Omega} = \int_{\Omega} \rho(\vec{r}) d\omega, \quad (10-38)$$

$$h_{\Omega} = \int_{\Omega} \rho(\vec{r}) \vec{r} d\omega, \quad (10-39)$$

$$I_{\Omega} = \int_{\Omega} \rho(\vec{r}) \mathbf{R}^T \mathbf{R} d\omega, \quad (10-40)$$

where  $\mathbf{R}$  is the cross product matrix for  $\vec{r}$  (i.e.,  $\mathbf{R}\mathbf{x} = \vec{r} \times \mathbf{x}$  for any column vector  $\mathbf{x}$ ).

With these values, let the following matrix be defined:

$$\mathbf{J}_{\Omega} = \begin{bmatrix} m_{\Omega} \mathbf{I} & \mathbf{H}_{\Omega} \\ \mathbf{H}_{\Omega}^T & \mathbf{I}_{\Omega} \end{bmatrix}, \quad (10-41)$$

where  $\mathbf{I}$  is the 3×3 identity matrix and  $\mathbf{H}_{\Omega}$  is the cross-product matrix for  $h_{\Omega}$ . This produces, for a single frame end effector,

$$\int_{\Omega} \rho(\vec{r}) \|\vec{v}_e(\vec{r})\|^2 d\omega = \mathbf{V}_e^T \mathbf{J}_{\Omega} \mathbf{V}_e. \quad (10-42)$$

This is the same form as (10-31), allowing the optimization of (10-37) to be performed using (10-36) with  $\mathbf{A} = \mathbf{J}_{\Omega}$ .

In general, there will be N end effectors, and the matrix  $\mathbf{A}$  will be block diagonal:

$$A = \begin{bmatrix} [\mathbf{J}_{\Omega,1}] & & & \mathbf{0} \\ & [\mathbf{J}_{\Omega,2}] & & \\ & & \ddots & \\ \mathbf{0} & & & [\mathbf{J}_{\Omega,N}] \end{bmatrix}. \quad (10-43)$$

### 10.3.9.3 Implementation

Statistical error reduction is implemented through the *EcControlExpressionErrorReduction* class. This gives a vector weight for use with the basic core velocity control system. It can be converted to a  $\mathbf{B}$  parameter for use with an AB core by combining it with an *EcControlExpressionVectorToAB* expression.

### 10.3.10 Table Function

An arbitrary linearly interpolated table function can be inserted into the control tree. This table function can map any  $n$ -dimensional domain to an  $m$ -dimensional range. It is implemented through

the class *EcControlExpressionColumnTableFunction*. The implementation details for the table function are defined below.

### 10.3.10.1 Regular and Irregular Table Functions

The XML input structure for the table function is useful in describing the inner workings of the base class algorithms for the table function. The figure below shows an example XML input structure.

```
1 <tableFunctionInterpolator>
2   <independentData size="1">
3     <element searchOption = "bisection"
4         lowerBoundaryOption = "extrapolate"
5         upperBoundaryOption = "extrapolate">
6       <independentVector size="4">
7         1.0  2.0  3.0  4.0
8       </independentVector>
9     </element>
10  </independentData>
11  <dependentData size="1">
12    <element size="4">
13      1.0  4.0  9.0  16.0
14    </element>
15  </dependentData>
16 </tableFunctionInterpolator>
```

**Text Box 10-1:** Example of a table function XML structure

Under the root element (e.g., *tableFunctionInterpolator* line 1), there are two children. The first child, “*independentData*” (line 2), describes the independent data and the second child, “*dependentData*” (line 11), describes the dependent data.

The independent data describes the domain of the function. It can have any number of values each for any number of dimensions. As a simple example, the function  $z=x^2$  defined over  $x=[0, 2]$  would have a single dimension and might have independent data points of (0.0, 1.0, 2.0). It could also have data points of (0.0, 0.5, 1.0, 1.5, 2.0) for finer resolution. The function  $z=x^2 y$ , on the other hand, would need two dimensions of data points, such as (0.0, 1.0, 2.0) for  $x$  and (0.0, 0.25, 0.5, 0.75, 1.0) for  $y$ .

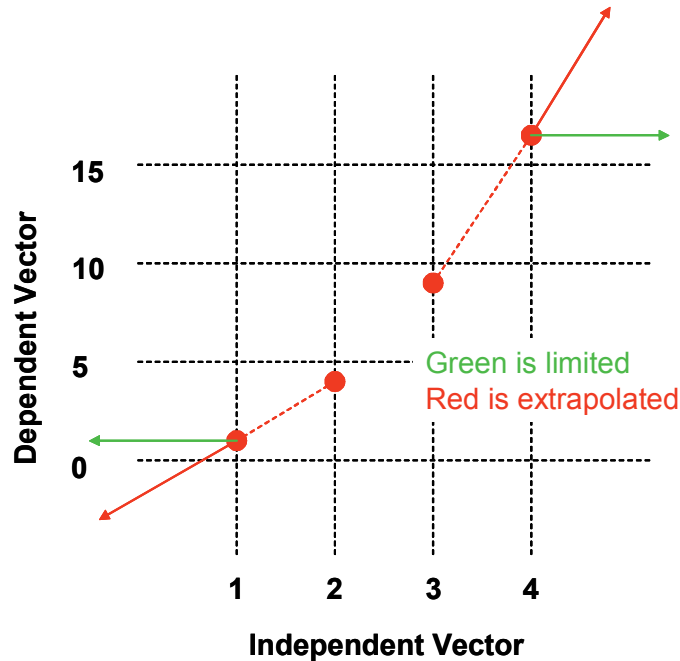
The dependent data describes the range of the function. It provides a function value for each combination of independent data points. To define the function  $z=x^2 y$  over the independent data points (0.0, 1.0, 2.0) for  $x$  and (0.0, 0.25, 0.5, 0.75, 1.0) would require  $3 \times 5 = 15$  dependent data values.

The  $z=x^2$  example above, defined over  $x=[1, 4]$ , contains only one dimension as specified on line 2. This could serve as a template—if more dimensions were necessary, lines 3–9 could be copied and inserted after line 9 to add another dimension. Several parameters further specify an independent variable: *searchOption*, *lowerBoundaryOption*, *upperBoundaryOption*, *independentVector*.

The search option (line 3) has 3 alternatives that define how the independent vector is searched. Note that the independent data must be strictly increasing or decreasing for these options to work. The first option is “linear”, which can be very efficient if the independent variable state does not change much between interpolator evaluations. With each evaluation, the search begins from the

previous search result. Although a linear search algorithm generically accesses data in order(N) time (where N is the number of gridded data points), the search is complete in constant time if the state is not changing. This works well in specific applications. If the state is quite dynamic, however, the method of bisection is more appropriate. This second option accesses data in log(N) time. Option “linear” or “bisection” is required for data with irregular grid spacing. The third option, “equal\_spaced”, is suited for data with regular grid spacing. It always accesses data in constant time, but puts constraints on how the data can be represented.

On occasion, the independent variable state can lie outside of the independent vector bounds. The user can specify how the interpolator evaluates this condition using the upper and lower boundary options (lines 4–5). The options are “limit” and “extrapolate” as illustrated in the figure below.



**Figure 10-15:** Example of the “limit” and “extrapolate” options. The green line (light gray in a printed document) shows the *limited* behavior and the red line (dark gray in a printed document) is the *extrapolated* behavior.

The independent data in Figure 10-15 is captured in lines 6–8 of Text Box 10-1, and the dependent data is contained in lines 12–14. When the dependent data has more than one dimension, it is important to sort the independent variables such that they map properly to the dependent data. For example, the first independent vector maps to the first dimension (sometimes viewed as column data), the second independent vector maps to the second dimension (sometimes viewed a row data), and so forth.

The best way to represent this is as follows: every dimension (n) is built by repeated sampling of the previous dimension (n-1). Starting with one dimension:  $a = x$ . If  $x = (0, 1, 2)$ , then the independent data would be  $a=(0, 1, 2)$ . For two dimensions, the data builds upon the 1-D data set. So, given  $a = x + y$ , if  $x = (0, 1, 2)$  and  $y = (0, 2, 4)$ , then  $a$  is given by the following:

```
0 1 2 (same as 1-D)
2 3 4
```



4 5 6

The data can be represented as a single vector,  $(0, 1, 2, 2, 3, 4, 4, 5, 6)$ , which is analogous to how the 2-D data is stored in memory. Notice that the first independent variable (in this case  $x$ ) is aligned with the fastest moving index.

For three dimensions, the data builds upon the 2-D data set. So, given  $a = x+y+z$  with  $x=(0, 1, 2)$ ,  $y=(0, 2, 4)$ , and  $z=(0, 3, 6)$ ,  $a$  is given by the following:

```
0 1 2 (same as 2-D)
2 3 4
4 5 6
```

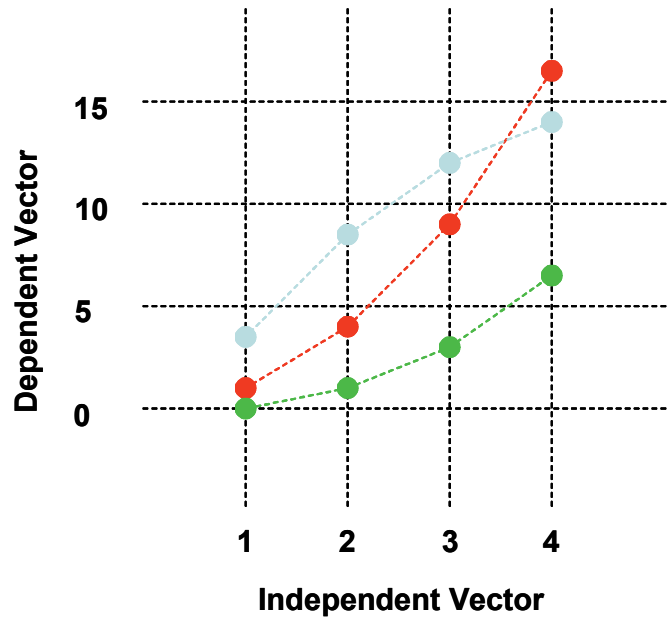
```
3 4 5
5 6 7
7 8 9
```

```
6 7 8
8 9 10
10 11 12
```

or

$(0, 1, 2, 2, 3, 4, 4, 5, 6, 3, 4, 5, 5, 6, 7, 7, 8, 9, 6, 7, 8, 8, 9, 10, 10, 11, 12)$ .

In many applications, several dependent data sets can be defined with a single definition for the independent data. This is illustrated in Figure 10-16. For example, in a six degree-of-freedom floating-base manipulator, an experiment might produce three scalar force components and three scalar moment components at each grid point. If all the parameters are saved in the same structure, then looking up each one independently can be inefficient. Although the example in Text Box 10-1 contains one dependent data set (lines 11–15), many more (with no limit other than that imposed by computer resources) can be added similarly to how the independent data can be expanded. The table function can perform the interpolation for the “2 through N” data sets more quickly because many intermediate calculations from the first set are saved and reused.

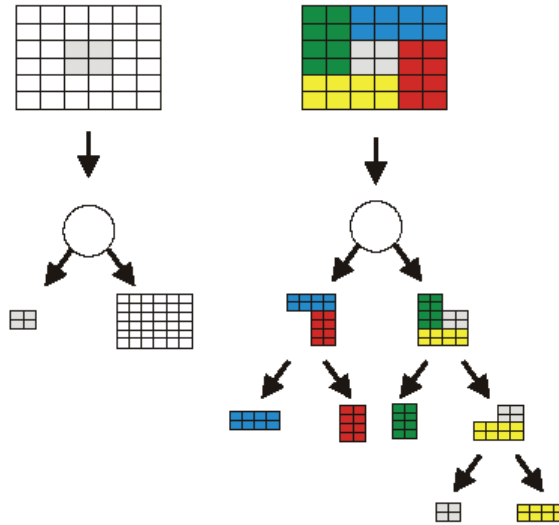


**Figure 10-16:** Data set example with one independent variable and three dependent variables

Although this table function capability is generic and powerful, it has a few limitations. For example, the basic table function must have blocked data. Specifically, it cannot have variable length boundaries, and it cannot have holes in the data. Also, if patches of the data are irregular spaced, the whole data set must be searched using an irregular-spaced search option (i.e., linear or bisection). It is for these reasons that the composable table functions were created, as described in the next section. For many applications, however, these limitations are not present, and the basic table function capability is useful without composition.

### **10.3.11 Composable Table Functions**

The ultimate purpose of the table function is to give the user a generic capability for adding complex or data-driven functions to the expression tree. To make this capability general, a composite table function capability was created, where the function domain is broken into subsets with each subset having its own table function definition. Figure 10-17 illustrates this approach.



**Figure 10-17:** Composite table function examples. The example to the left illustrates overlapping subsets where the center has special features. The example to the right illustrates an alternative way of dissecting the example on the left with no overlapping subsets. In each of the examples, the final subsets have their own table function definitions. The composite table function can handle both examples.

The composite table function contains regular or irregular table functions at the leaf nodes (i.e., bottom nodes in tree). Note that although the root nodes (i.e., top node of the tree) in Figure 10-17 contain blocked data, the data does not have to be blocked and it can have holes in it. Only the components on a leaf node, if defined using a regular or irregular table function, need to be blocked. And the toolkit supports extension to nonblocked leaves. Through this approach, the limitations of the regular and irregular table functions are overcome.

A composite table function defined through XML has a structure as shown in Text Box 10-2. This example has a nested branch on the left (or upper) branch and a table function on the right (or bottom) branch. The data between the “tableFunctionInterpolator” tags would be that required for table functions as described previously (e.g., Text Box 10-1).

```

1  <tableFunctionContainer>
2  <tableFunctionBranch>
      <!-- left branch -->
3  <tableFunctionBranch>
4  <tableFunctionInterpolator>
5  ...
6  </tableFunctionInterpolator>
7  <tableFunctionInterpolator>
8  ...
9  </tableFunctionInterpolator>
10 </tableFunctionBranch>
      <!-- right branch -->
11 <tableFunctionInterpolator>
12 ...
13 </tableFunctionInterpolator>
14 </tableFunctionBranch>
15 </tableFunctionContainer>

```

**Text Box 10-2:** Example of a composable table function XML structure.

The binary search tree enables the data to be nested to any depth. This is done through the *EcBaseExpressionTreeContainer* class, which contains a composable table function data type. This class can contain two objects that are in turn composable table functions. This approach provides a powerful framework for generic table functions.

The examples in Figure 10-17 illustrate a few considerations that are addressed through the composable table function. The left composite contains two overlapping table functions. The binary search tree first determines which table function contains the independent variable state with the left branch taking precedence over the right branch. Since the state can lie in multiple table functions when overlapping exists, it is particularly important to note that the left (top in the XML) branch takes precedence. If the state lies in neither table function, the binary search tree looks for the first table function (again searching from left to right [or top to bottom in XML]) that can extrapolate to the state. In this example, the left table function would be defined with the “limit” option, and the right table function would be defined with the “extrapolate” option. This would enable the right table function to be used for extrapolation. Through this approach, all possible locations of the state can be evaluated.

In the right example in Figure 10-17, the search is performed similarly to the left example. A more sophisticated definition of the boundary options is needed though to properly enable the extrapolation algorithm. To perform an interpolation evaluation equivalently to the left example, all table function boundaries that are not touching a neighboring table function would need a boundary option set to “extrapolate”, while the other joint boundary options are set to “limit”. Through this approach, the left and right example can produce the same numerical results.

## 10.4 End-Effector Descriptions

The array  $\mathbf{V}$ , as used above, represents the motion of all the manipulator's end effectors. In the Actin™ design, a special class holds the description of an end-effector set, which contains any number of end effectors. The end effectors can represent any type of constraint. Implemented end-effector types include frame, 3D point, 3D orientation, 2D point, center of mass, and spatial momentum. More types can be added using the toolkit or the plugin interface.

To allow this general approach, many of calculations needed for velocity control are performed in the end-effector class. The public methods that must be implemented to define a new end effector are given in the table below.

Member Function	Meaning
doc	Returns the end effector's degrees of constraint. For a point end effector, it returns 3. For a frame end effector, 6.
insertJacobianComponent	Builds a strip of the Jacobian. The height of the strip equals the value returned by <i>doc</i> .
insertSparsityComponent	Builds a strip of the sparsity description of the Jacobian. A value of true in this strip means the corresponding position in the Jacobian is always zero. The height of the strip equals the value returned by <i>doc</i> .
calculatePlacement	Calculates a placement value for the end effector. The placement is described through an <i>EcCoordinateSystemTransformation</i> , which may have different meanings for different end-effector types.
calculateVelocity	Calculates end-effector velocity. The result is a real vector of length equal to the value returned by <i>doc</i> . The velocity will have different meanings for different end-effector types.
calculateAcceleration	Calculates end-effector acceleration. The result is a real vector of length equal to the value returned by <i>doc</i> . The acceleration will have different meanings for different end-effector types.
filterEndEffectorVelocity	Calculates an end-effector velocity that drives the end effector toward a guide placement. The guide frame is always represented in system coordinates.
minimumTime	Calculates the minimum time that will be used to move from one frame to another.
difference	Calculates a measure of the difference between two placement frames that uses Euclidean distance as its baseline. That is, the difference between two frames is the Euclidean distance between them plus optional additional factors related to orientation change.

**Table 10-6:** Member functions that are implemented to define a new type of end effector.

Through this approach, any new end effector can be added as a subclass of *EcEndEffector*, provided these member functions are implemented in the new class. These member functions allow the end-effectors to create their own Jacobians and position controllers. Subclasses of *EcEndEffector* that are provided with the Actin™ toolkit include *EcFrameEndEffector*, *EcPointEndEffector*, and *EcXyEndEffector*.

## 10.5 External-Force Optimization through Momentum Constraint

A robotic mechanism moving without external forces must conserve linear and angular momentum [31]. A control law that is consistent with this requirement will move the robot in a manner consistent with dynamic movement in free space without external force.

### 10.5.1 Organization

The spatial momentum constraint was added to the control-software framework as a new end effector. As background, in the software framework, the end effectors for any given manipulator are represented as an end-effector set, which contains any number of any types of end effectors. Implemented end-effector types include 3D frame, 3D point, 2D point, center of mass, and linear constraint. Any new end-effector type, such as momentum constraint, can be added just by adding a new class, subclassed from *EcEndEffector*, to define it.

To allow this general approach, many of calculations needed for velocity control are performed in the end-effector class. The public methods that must be implemented are given in the table below.

Member Function	Meaning
doc()	Returns the end effector's degrees of constraint. For a point end effector, this would return three. For spatial momentum constraint, it would return six.
insertJacobianComponent()	Builds a strip of the Jacobian. The height of the strip equals the value returned by doc().
insertSparsityComponent()	Builds a strip of the sparsity description of the Jacobian. A value of true in this strip means the corresponding position in the Jacobian is always zero.
calculatePlacement()	Calculates a placement value for the end effector. The placement is described through an <i>EcCoordinateSystemTransformation</i> , which may have different meanings for different end-effector types.
calculateVelocity()	Calculates end-effector velocity. The result is a real vector of length equal to the value returned by doc(). The velocity has different meanings for different end-effector types.
filterEndEffectorVelocity()	Calculates an end-effector velocity that drives the end effector toward a guide placement. The guide frame is always represented in system coordinates.
minimumTime()	Calculates the minimum time that will be used to move from one

	frame to another.
difference()	Calculates a measure of the difference between two placement frames. It uses Euclidean distance as its baseline. That is, the difference between two frames is the Euclidean distance between them plus optional additional factors related to orientation change.

**Table 10-7:** Member functions that must be implemented to define a new type of end effector.

Of these, the method implementation that takes the most effort is insertJacobianComponent(), which adds a strip of numbers to the Jacobian, as illustrated in the figure below.

$$J = \begin{bmatrix} \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times \end{bmatrix}$$

**Figure 10-18:** Each end effector class must be able to write an appropriate strip of numbers into the Jacobian. This is done in the method insertJacobianComponent() in Table 10-7. The biggest challenge in adding spatial-momentum constraint as an end effector lies in calculating this data.

### 10.5.2 Derivation

To determine the set of numbers that must be inserted into the Jacobian as shown in Figure 10-18, let  $\mathbf{M}$  be the spatial momentum of the robotic mechanism, a 6x1 column vector formed by augmenting linear momentum,  $\vec{\mu}$ , with angular momentum  $\vec{\lambda}$ :

$$\mathbf{M} = \begin{bmatrix} \vec{\mu} \\ \vec{\lambda} \end{bmatrix}. \quad (10-44)$$

Let the matrix  $\mathbf{J}_M$  be defined through the following equation:

$$\mathbf{M} = \mathbf{J}_M \dot{\mathbf{q}}^*, \quad (10-45)$$

$\mathbf{J}_M$  is the inserted strip of the Jacobian as shown in Figure 10-18, and  $\dot{\mathbf{q}}^*$  gives the joint rates augmented by the base spatial velocity, i.e.,

$$\dot{\mathbf{q}}^* = \begin{bmatrix} \dot{\mathbf{q}} \\ \cdots \\ \mathbf{V}_b \end{bmatrix}, \quad (10-46)$$

where

$$\dot{\mathbf{q}} = \begin{bmatrix} \dot{q}_0 \\ \vdots \\ \dot{q}_{N-1} \end{bmatrix}, \quad (10-47)$$

and

$$\mathbf{V}_b = \begin{bmatrix} \vec{v}_b \\ \vec{\omega}_b \end{bmatrix}. \quad (10-48)$$

Here,  $\vec{v}_b$  is the linear velocity of the robot base and  $\vec{\omega}_b$  is its angular velocity.

### 10.5.2.1 Transformations

To calculate  $\mathbf{J}_M$ , let the following be defined: For any reference frames  $i$  and  $j$  that are rigidly connected, Let  ${}^j\mathbf{P}_{j \rightarrow i}$  be the cross-product matrix for  ${}^j p_{i \rightarrow j}$ , the vector from the origin of frame  $i$  to the origin of frame  $j$ , expressed in frame  $j$ . And let  ${}^j\mathbf{R}_i$  be the rotation matrix expressing frame  $i$  in frame  $j$ . Using this, let the matrices  ${}^F\mathbf{T}_{i \rightarrow j}$  and  ${}^A\mathbf{T}_{i \rightarrow j}$  be defined as follows:

$${}^F\mathbf{T}_{i \rightarrow j} = \begin{bmatrix} {}^j\mathbf{R}_i & 0 \\ {}^j\mathbf{P}_{j \rightarrow i} {}^j\mathbf{R}_i & {}^j\mathbf{R}_i \end{bmatrix}, \quad (10-49)$$

and

$${}^A\mathbf{T}_{i \rightarrow j} = \begin{bmatrix} {}^j\mathbf{R}_i & {}^j\mathbf{P}_{j \rightarrow i} {}^j\mathbf{R}_i \\ 0 & {}^j\mathbf{R}_i \end{bmatrix}. \quad (10-50)$$

Let  $\mathbf{M}_j$  be the spatial momentum of a body defined in a reference frame  $j$ . Similarly, let  $\mathbf{F}_j$ ,  $\mathbf{V}_j$ , and  $\mathbf{A}_j$  be the spatial force, velocity, and acceleration of a body in reference frame  $j$ .

Transformation  ${}^F\mathbf{T}_{i \rightarrow j}$  produces the following equalities for force and momentum transformation:

$$\mathbf{M}_j = {}^F\mathbf{T}_{i \rightarrow j} \mathbf{M}_i, \quad (10-51)$$



$$\mathbf{F}_j = {}_F\mathbf{T}_{i \rightarrow j} \mathbf{F}_i, \quad (10-52)$$

And  ${}_A\mathbf{T}_{i \rightarrow j}$  produces the following for velocity and acceleration transformation:

$$\mathbf{V}_j = {}_A\mathbf{T}_{i \rightarrow j} \mathbf{V}_i. \quad (10-53)$$

$$\mathbf{A}_j = {}_A\mathbf{T}_{i \rightarrow j} \mathbf{A}_i. \quad (10-54)$$

An important identity is the following:

$${}_A\mathbf{T}_{i \rightarrow j}^T = {}_F\mathbf{T}_{j \rightarrow i}, \quad \forall i, j. \quad (10-55)$$

### 10.5.2.2 Rigid-Body Momentum

For any rigid body, let  $\vec{\omega}$  be the angular velocity,  $\vec{v}$  be the linear velocity,  $m$  be the mass,  $\mathbf{H}$  be the cross-product matrix for the first moment of inertia, and  $\mathbf{J}$  be the second moment of inertia (a separate entity from the manipulator Jacobian  $\mathbf{J}$ ). Then, the linear and angular momentum equations are given by the following:

$$\vec{\mu} = \mathbf{H}^T \vec{\omega} + m\vec{v}. \quad (10-56)$$

$$\vec{\lambda} = \mathbf{J} \vec{\omega} + \mathbf{H} \vec{v}. \quad (10-57)$$

Let the 6×6 rigid-body inertia be defined as follows:

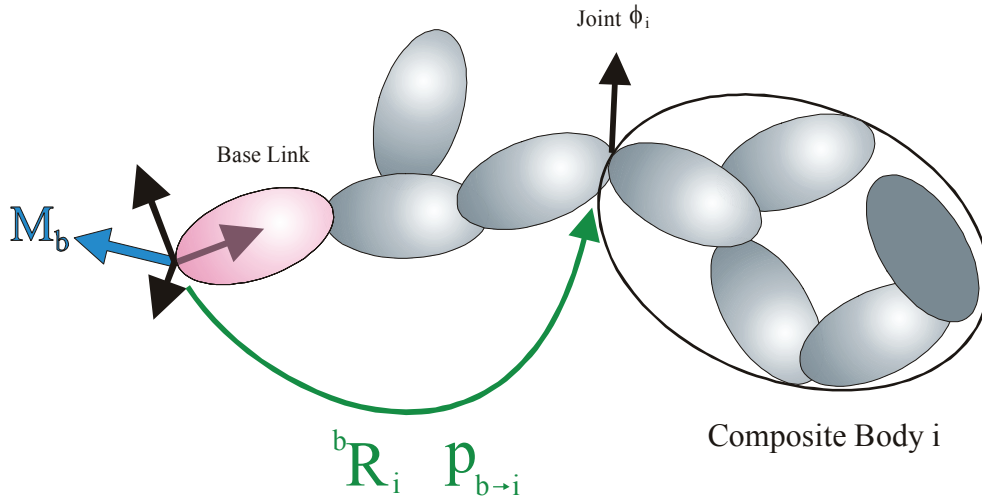
$$\mathbf{I}^C = \begin{bmatrix} m\mathbf{I} & \mathbf{H}^T \\ \mathbf{H} & \mathbf{J} \end{bmatrix}, \quad (10-58)$$

where  $\mathbf{I}$  is the 3×3 identity matrix. With this, the spatial momentum equation for any link can be represented as

$$\mathbf{M} = \mathbf{I}^C \mathbf{V}. \quad (10-59)$$

### 10.5.2.3 Summing the Effect of Each Joint

The effect of all joint rates must be combined to calculate the momentum of the entire manipulator. This can be done by calculating the composite rigid-body inertia of all links outboard from each joint in turn. The momentum produced by the motion of a single joint in isolation is the momentum produced by the action of the joint on this composite body. For combination with the momentum produced by the action of other joints, the momentum so calculated must be transformed to a common reference frame. The figure below illustrates this calculation.



**Figure 10-19:** The momentum produced by the rate of joint  $i$  can be calculated by collecting all outboard bodies into a composite rigid body. The momentum so calculated is an additive term, and must be transformed to a common frame. Illustrated here is transformation to the base frame.

To express the combination mathematically, let a matrix  $\mathbf{D}$  be defined as follows:

$$\mathbf{D} = \begin{bmatrix} \phi_0^T \mathbf{I}_0^C ({}^A \mathbf{T}_{b \rightarrow 0}) \\ \phi_1^T \mathbf{I}_1^C ({}^A \mathbf{T}_{b \rightarrow 1}) \\ \vdots \\ \phi_{n-1}^T \mathbf{I}_{n-1}^C ({}^A \mathbf{T}_{b \rightarrow n-1}) \end{bmatrix}, \quad (10-60)$$

where  $\mathbf{I}_i^C$  is the rigid-body inertia of the composite body formed by joining all links outboard and including link  $i$ . The index  $b$  represents the base frame.

With this, the total momentum due to the manipulator's joint rates,  $\widehat{\mathbf{M}}_b$ , can be found as follows:

$$\widehat{\mathbf{M}}_b = \mathbf{D}^T \dot{\mathbf{q}}. \quad (10-61)$$

Remaining is the momentum contribution due to the linear and angular velocity of the base. Using (10-59), this component is calculated as follows:

$$\widetilde{\mathbf{M}}_b = \mathbf{I}_b^C \mathbf{V}_b, \quad (10-62)$$

where  $\mathbf{I}_b^C$  is the  $6 \times 6$  rigid-body inertia of the entire composite manipulator, including the base link and  $\mathbf{V}_b$  is the frame velocity of the base link.

The total manipulator momentum is the sum of these two components, the spatial momentum due to the action of the joints and the spatial momentum due to base motion:

$$\tilde{\mathbf{M}}_b = \widehat{\mathbf{M}}_b + \check{\mathbf{M}}_b, \quad (10-63)$$

This gives

$$\mathbf{M}_b = \mathbf{D}^T \dot{\mathbf{q}} + \mathbf{I}_b^C \mathbf{V}_b, \quad (10-64)$$

or

$$\mathbf{M}_b = \begin{bmatrix} \mathbf{D}^T & \vdots & \mathbf{I}_b^C \end{bmatrix} \begin{bmatrix} \dot{\mathbf{q}} \\ \cdots \\ \mathbf{V}_b \end{bmatrix}, \quad (10-65)$$

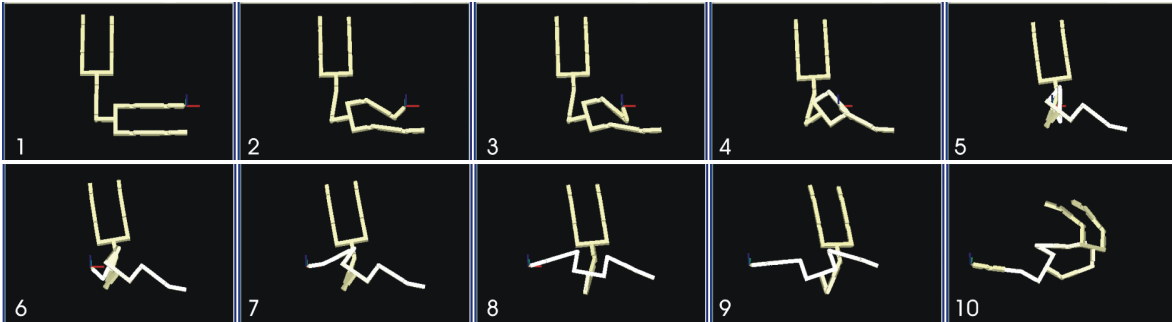
which is exactly the form of equation (10-45). Thus

$$\mathbf{J}_M = \begin{bmatrix} \mathbf{D}^T & \vdots & \mathbf{I}_b^C \end{bmatrix}. \quad (10-66)$$

With this component inserted into Jacobian, as shown in Figure 10-18, spatial momentum can be constrained, and control is implemented that requires no external force on the manipulator.

### 10.5.3 Example

Shown in the figure below is a simple 21-joint (27 dof) mechanism moving under spatial-momentum constraint. Notice that movement of the end effector from one side to the other shifts the other parts of the mechanism in an equal and opposite manner. Behavior is similar in kinematic and dynamic modes.



**Figure 10-20:** An illustration of a 21-joint test mechanism being controlled with spatial momentum constraint. A point end effector is moved from the right-hand side to the left-hand side.

## 10.6 Control System Parameter Provision

The robot control system comprises end effectors as constraints and a parameterized optimization algorithm. It is possible to exchange end effectors and optimization parameters while controlling a robotic manipulator. However, exchanging these quantities requires reconfiguring the control system, which usually includes dynamic memory allocation. It is not feasible to do this every time step.

The timestep-by-timestep transmission of data related to the end effector constraints is performed through a desired coordinate system. This information can be passed without reconfiguration. Prior

to the last quarterly period, though, there was no corresponding method of data transmission for the optimization process. To change optimization parameters, the entire control system had to be exchanged.

The use of a data map is available for passing information to the optimization portion of the control system. This data map organizes information in six maps:

- String-String
- String-Real
- String-Integer
- String-Real Vector
- String-Integer Vector
- String-Data Map

Through these components, most types of data can easily be passed to the optimization system. The use of a string-data-map component allows data to be organized hierarchically.

### 10.6.1 *Soft-Constraint End Effectors*

The core velocity framework Energid uses in the rapid-prototyping system it is developing for NASA is based on the manipulator Jacobian equation:

$$\mathbf{V} = \mathbf{J}(\mathbf{q})\dot{\mathbf{q}}, \quad (10-67)$$

where  $\mathbf{V}$  is an  $m$ -length vector representation of the motion of the end effectors (usually some combination of linear and angular velocity referenced to points rigidly attached to parts of the manipulator);  $\mathbf{q}$  is the  $n$ -length vector of joint positions (with  $\dot{\mathbf{q}}$  being its time derivative); and  $\mathbf{J}$  is the  $m \times n$  manipulator Jacobian, a function of  $\mathbf{q}$ .

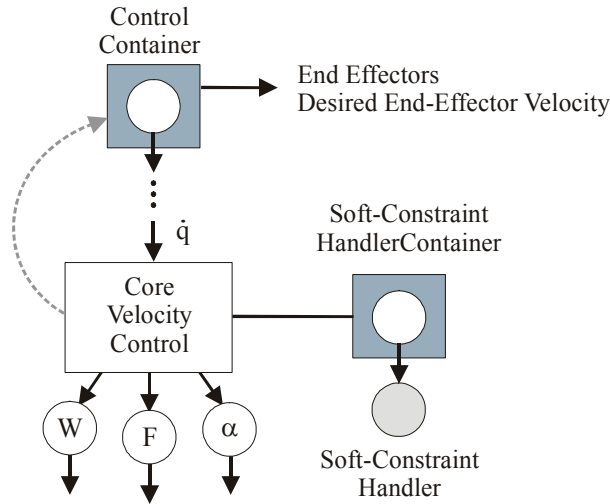
To solve for  $\dot{\mathbf{q}}$  given  $\mathbf{V}$ , the framework is built on that described in [32], which uses a scalar  $\alpha$ , a matrix function  $\mathbf{W}(\mathbf{q})$ , and a vector function  $\mathbf{F}(\mathbf{q})$  to solve for  $\dot{\mathbf{q}}$  given  $\mathbf{V}$  through the following formula:

$$\dot{\mathbf{q}} = \begin{bmatrix} \mathbf{J} \\ \mathbf{N}_J^T \mathbf{W} \end{bmatrix}^{-1} \begin{bmatrix} \mathbf{V} \\ -\alpha \mathbf{N}_J^T \mathbf{F} \end{bmatrix}. \quad (10-68)$$

where  $\mathbf{N}_J$  is an  $n \times (n-m)$  set of vectors that spans the null space of  $\mathbf{J}$ . That is,  $\mathbf{J}\mathbf{N}_J = \mathbf{0}$ , and  $\mathbf{N}_J$  has rank  $(n-m)$ . By changing the values of  $\alpha$ ,  $\mathbf{W}$ , and  $\mathbf{F}$ —which can be configured using XML—many new and most established velocity-control techniques can be implemented. Mathematically, it achieves the desired  $\mathbf{V}$  while minimizing  $\frac{1}{2}\dot{\mathbf{q}}^T \mathbf{W}\dot{\mathbf{q}} + \alpha \mathbf{F}^T \dot{\mathbf{q}}$ .

In the last quarter, Energid implemented an easy way to move end-effectors from the constraint portion of this formulation (i.e.,  $\mathbf{J}$  and  $\mathbf{V}$ ) to the optimization portion (i.e.,  $\alpha$ ,  $\mathbf{W}$ , and  $\mathbf{F}$ ). Every end-effector now has a flag indicating whether it is a hard or a soft constraint. When an end effector is flagged as a soft constraint, it is not included in the Jacobian equation. Instead, it is incorporated into  $\alpha$ ,  $\mathbf{W}$ , and  $\mathbf{F}$  through a soft-constraint handler.

The soft-constraint handler is an object that resides in a container that is a member variable of the class implementing the core velocity algorithm. This is illustrated in the figure below.



**Figure 10-21:** An illustration of the control system with the soft-constraint handler. The core velocity control algorithm resides within the XML-configurable control system. The class that defines that algorithm now holds a container for a soft-constraint handler. The soft constraint handler changes  $\alpha$ ,  $\mathbf{W}$ , and  $\mathbf{F}$  to trade-off end-effector movement with other optimization criteria.

There are many possible ways to implement soft constraints. As desired, new objects for the container shown in the figure above can be added by users of the toolkit by creating an appropriate dynamic library (e.g., DLL under Windows). This method adds minimization of a sum-of-squares error metric to the optimization criteria.

Let  $\mathbf{J}_s(\mathbf{q})$  be the Jacobian formed from all the soft-constrain end effectors in use. And let  $\mathbf{V}_s$  be the concatenated desired velocity of those end effectors. Then an approach to handling the soft constraints is to minimize a measure of the soft end-effector error  $\mathbf{E}_s$ , defined as follows:

$$\mathbf{E}_s = \mathbf{J}_s(\mathbf{q})\dot{\mathbf{q}} - \mathbf{V}_s \quad (10-69)$$

Let the measure of this error be defined through a weighted quadratic form. That is, for some positive definite weighting matrix  $\mathbf{W}_s$ , define the measure to be minimized as

$$e_s = \mathbf{E}_s^T \mathbf{W}_s \mathbf{E}_s \quad (10-70)$$

The method of (10-68) minimizes

$$e_o = \frac{1}{2} \dot{\mathbf{q}}^T \mathbf{W} \dot{\mathbf{q}} + \alpha \mathbf{F}^T \dot{\mathbf{q}} \quad (10-71)$$

based on values of  $\alpha$ ,  $\mathbf{F}$ , and  $\mathbf{W}$  that have already been defined through the control-system database. To integrate the soft constraints, the default handler makes a trade-off based on a scalar parameter,  $\sigma$ . It simply changes  $\alpha$ ,  $\mathbf{F}$ , and  $\mathbf{W}$  to minimize the following:

$$e = e_o + \sigma e_s \quad (10-72)$$

The method by which  $\alpha$ ,  $\mathbf{F}$ , and  $\mathbf{W}$  change can be established through the following derivation: Combining the three above equations gives

$$e = \frac{1}{2} \dot{\mathbf{q}}^T \mathbf{W} \dot{\mathbf{q}} + \alpha \mathbf{F}^T \dot{\mathbf{q}} + \sigma \mathbf{E}_s^T \mathbf{W}_s \mathbf{E}_s. \quad (10-73)$$

Using (27) gives

$$e = \frac{1}{2} \dot{\mathbf{q}}^T \mathbf{W} \dot{\mathbf{q}} + \alpha \mathbf{F}^T \dot{\mathbf{q}} + \sigma (\mathbf{J}_s(\mathbf{q}) \dot{\mathbf{q}} - \mathbf{V}_s)^T \mathbf{W}_s (\mathbf{J}_s(\mathbf{q}) \dot{\mathbf{q}} - \mathbf{V}_s). \quad (10-74)$$

This gives

$$e = \frac{1}{2} \dot{\mathbf{q}}^T (\mathbf{W} + 2\sigma \mathbf{J}_s^T \mathbf{W}_s \mathbf{J}_s) \dot{\mathbf{q}} + (\alpha \mathbf{F}^T - 2\sigma \mathbf{J}_s^T \mathbf{W}_s \mathbf{V}_s) \dot{\mathbf{q}} + \sigma \mathbf{V}_s^T \mathbf{V}_s. \quad (10-75)$$

The final term is constant, and not relevant to the optimization process. The first two terms take the same form as those in (10-69). Thus, the handler can minimize  $e$  by changing  $\alpha$ ,  $\mathbf{F}$ , and  $\mathbf{W}$  as follows:

$$\begin{aligned} \mathbf{W} &\leftarrow \mathbf{W} + 2\sigma \mathbf{J}_s^T \mathbf{W}_s \mathbf{J}_s \\ \mathbf{F} &\leftarrow \alpha \mathbf{F} - 2\sigma \mathbf{J}_s^T \mathbf{W}_s \mathbf{V}_s. \\ \alpha &\leftarrow 1 \end{aligned} \quad (10-76)$$

This is how the new soft-constraint handler includes the soft-constraints in the optimization process.

## 10.7 Example Code

### 10.7.1 Minimum Potential Energy Control for the PUMA

In this section, a velocity control system is constructed for a PUMA 560 manipulator and used to drive a point end effector. A velocity control system was constructed as part of the quick-start description in Chapter 2, which is useful for comparison. The quick-start example used the RRC K-1207i arm with a position controller constructed around a velocity controller using the standard core. This example uses the PUMA, controls the arm motion directly through the velocity control system, and uses an AB core.

#### 10.7.1.1 Loading and Displaying the PUMA

An example manipulator system for the PUMA 560 is defined in an example file that is distributed with the Actin™ software. This file, “puma\_system.xml,” contains the description of an object of the class *EcStatedSystem*. C++ code for loading and displaying this model using the class *EcRenderWindow* is given in Text Box 10-3 below.

```

// declare an error return code
EcBoolean success;

// declare a filename
EcString filename="../../../../data/actinExamples/puma_system.xml";

// declare a visualizable stated system object
EcVisualizableStatedSystem visStatedSystem;

// load the stated system from an XML file
success = visStatedSystem.readFromFile(filename);

// make sure it loaded properly
if(!success)
{
    EcWARN("Could not load stated system.\n");
    return;
}

// instantiate a renderer
EcRenderWindow renderer;

// set the size of the window
renderer.setWindowSize(256,256);

// set the system
if(!renderer.setVisualizableStatedSystem(visStatedSystem))
{
    return;
}

// view the system
renderer.renderScene();

// after a pause, close the window
EcSLEEPMS(1000);
renderer.closeScene();

```

**Text Box 10-3:** Example code for loading and rendering the PUMA model. This is Example Section #1 in the velocity-control example code.

### 10.7.1.2 Defining a Velocity-Control System

A velocity control system is defined for the PUMA through the code in Text Box 10-4. In this set of code, first a convenient reference is set for the manipulator. Then the link at the end of the kinematic chain starting with the manipulator is looked up. This is used to construct a point end effector that is rigidly attached to this last link.

A velocity-control expression is constructed. An *EcControlExpressionCoreAB* object is chosen to perform the inverse kinematics calculation. This object requires three child members: an **A** parameter, a **B** parameter, and a scalar. In this case, **A** is chosen by projecting the manipulator mass matrix onto the Jacobian null space, and **B** is chose by projecting the gradient of potential energy onto the Jacobian null space.

To this core velocity control algorithm, a joint-rate filter and an end-effector error filter are added. These prevent the joint rates and hand-motion error from exceeding specified bounds. The control expression and the end-effector definition are then added to the individual velocity control description. Since in this case there is only one manipulator being controlled, the velocity control system only has one description.



```

// make a convenient reference to the manipulator
const EcIndividualManipulator& manipulator =
    visStatedSystem.statedSystem().system().manipulators()[0];

// look up the last link
EcManipulatorLinkConstPointerVector linkPointerVector;
manipulator.collectLeafLinks(linkPointerVector);

// make a point end effector
EcPointEndEffector pointEnd;

// put the end effector into an end-effector set
pointEnd.setLinkIdentifier(EcXmlString(linkPointerVector[0]->label()));
EcEndEffectorSet eeSet;
eeSet.addEndEffector(pointEnd);

// create a velocity-control core
EcControlExpressionABCORE abCore;

// create a vector-to-AB converter with a potential-energy gradient child
EcControlExpressionVectorToAB vectorToAB;
vectorToAB.setChild(EcControlExpressionPotentialEnergyGradient());

// set the matrix, vector, and scalar for the core
abCore.setMatrixElement(EcControlExpressionMassMatrixAB());
abCore.setVectorElement(vectorToAB);
abCore.setScalarElement(EcExpressionScalarConstant::objectWithValue(0.05));

// add a joint-rate filter
EcControlExpressionJointRateFilter rateFilter;
EcExpressionGeneralColumn jointWeights;
jointWeights.assign(manipulator.jointDof(), 0.1);
rateFilter.setWeightsElement(jointWeights);
rateFilter.setUnfilteredRatesElement(abCore);

// add an end-effector error filter
EcControlExpressionEndEffectorErrorFilter eFilter;
EcExpressionGeneralColumn handWeights;
handWeights.assign(6, 0.1);
eFilter.setWeightsElement(handWeights);
eFilter.setUnfilteredRatesElement(rateFilter);
eFilter.setStopsAtLimits(EcTrue);

// add the system to the velocity control description
EcControlExpressionContainer container;
container.setTopElement(eFilter);

// add the expression and end effector to a velocity control description
EcIndividualVelocityControlDescription indVelContDesc;
indVelContDesc.setControlExpression(container);
indVelContDesc.setEndEffectorSet(eeSet);

// add the velocity control description to a velocity control system
EcVelocityControlSystem velContSys;
velContSys.addControlDescription(indVelContDesc);

// set the stated system
EcStatedSystem statedSystem=visStatedSystem.statedSystem();
velContSys.setStatedSystem(&statedSystem);

```

**Text Box 10-4:** Example code for building a velocity control system. This continues from the code shown in Text Box 10-3 and is Example Section #2 in the velocity-control example code.

### 10.7.1.3 Using the Velocity-Control System

The velocity control system defined through Text Box 10-4 is used to calculate joint velocities from end-effector velocities in Text Box 10-5. First the renderer is initialized, then parameters are established for a simulation. A desired end-effector velocity is constructed—in this case, there is one manipulator with one end effector. Then a simulation is run, with joint rates corresponding to the desired end-effector velocity calculated and integrated each time step.

```
// set the system
if(!renderer.setVisualizableStatedSystem(visStatedSystem))
{
    return;
}

// execution parameters
EcU32 steps=100;
EcReal simRunTime = 4.0;
EcReal simTimeStep = simRunTime/steps;

// set the desired velocity of the end effector
EcXmlRealVector values;
values.pushBack(-0.1);
values.pushBack(0.0);
values.pushBack(0.0);

// add it to a collection
EcXmlRealVectorVector collection;
collection.pushBack(values);

// create a desired end-effector velocity object
EcManipulatorEndEffectorVelocity endEffectorVelocity(collection);
EcManipulatorEndEffectorVelocityVector endEffectorVelocityVector;
endEffectorVelocityVector.pushBack(endEffectorVelocity);

// move to the desired pose, and render every time step
for(ii=0;ii<steps;++ii)
{
    // get the current time
    EcReal currentTime=simTimeStep*ii;

    // set the desired end-effector velocity
    velContSys.setEndEffectorVelocities(endEffectorVelocityVector);

    // calculate joint rates and update the manipulator state
    velContSys.calculate();
    statedSystem.setVelocityStates(velContSys.calculatedVelocityStates());

    // propagate the state
    statedSystem.propagateSelfTo(currentTime);

    // show the manipulator in this position
    renderer.setState(statedSystem.state());
    renderer.renderScene();
    EcSLEEPMS(1000*simTimeStep);
}
```

**Text Box 10-5:** Example code for building a velocity control system. This continues from the code shown in Text Box 10-4 and is Example Section #3 in the velocity-control example code.

## 11 Position Control

The position control system builds upon the velocity control system to give robust position control to the end effectors. A position control container is supplied as an interface to the position control system to enable the addition of new position control algorithms. Currently, the position control container includes two position control classes: *EcPositionControlSystem* and *EcPositionControlSystemWithSimulation*. The second class builds upon the first to give look-forward capability. Other position control classes can be created and added to the container through the plugin interface (see Section 18).

### 11.1 Position Control System Container

The position control container is subclassed from *EcXmlVariableElementType* which is a template defined as a position control system type (i.e., *EcPositionControlSystem*). This container holds a compound XML object and enables the user to add new position control systems as desired.

Most of the methods available to the container enable data to be passed back and forth to the position control system defined in the container. Table 11-1 defines some of these methods.

Method	Description
calculateState	Given a time, this method propagates the control system and state, and updates the end effector placements. This method returns an updated state through the argument list for rendering.
actualPlacementVector	Gets the achieved end effector placements
setActualPlacementVector	Sets the actual end effector placements
desiredPlacementVector	Gets the commanded end effector placements
setDesiredPlacementVector	Sets the commanded end effector placements
velocityControlSystem	Gets the velocity control system
setVelocityControlSystem	Sets the velocity control system
statedSystem	Gets the stated system
setStatedSystem	Sets the stated system

**Table 11-1** Listing of primary methods available to the developer. The position control container has many methods for passing data to the position and velocity control system. Check the code documentation for a complete list and description.

### 11.2 Position Control System

The position control system provides basic position control for the end effectors by building upon the velocity control system. Given the joint velocities from the velocity control system, the position control system uses Euler integration to calculate the joint positions. Once the joint positions are

calculated, the end effector positions are known. The control system checks for joint limit exceedance and collisions as it is iterated forward in time. It also zeros the joint rate commands if a singularity is detected.

Table 11-2 shows the member variables that are registered for XML access and Table 11-3 shows the primary methods available to the developer.

Member Data	Class/Type	Definition	Restrictions
m_IsOn	<i>EcXml...</i> <i>Boolean</i>	Flag indicating if the position control system is turn on	None
m_maxFinal... PropagationSize	<i>EcXmlU32</i>	Maximum propagation size in time steps for propagating the state for rendering. This is only used if maxIterations is reached.	None
m_MaxIterations	<i>EcXmlU32</i>	Maximum number of time step iterations per cycle. This is only used if the CPU is not able to run in real-time.	None
m_TimeStep	<i>EcXmlReal</i>	Time step for control system	None
m_UseTwoPasses	<i>EcXml...</i> <i>Boolean</i>	This protects against control problems at singularities. If there is no singularity detected, the results of the two passes are averaged to improve accuracy.	None
m_VelocityControl... System	<i>EcVelocity...</i> <i>Control...</i> <i>System</i>	Velocity control system	None
m_Collision... AvoidanceMode	<i>EcXml...</i> <i>EnumU32</i>	Collision avoidance mode	3 Options See class description
m_Collision... BreakdownThreshold	<i>EcXmlReal</i>	Threshold specifying when a material will break down and allow an object to pass through. Typically this value is set to 1, meaning the system will stop at all link collisions.	[0,1]
m_BVHPrecision	<i>EcXmlU32</i>	Level in the bounding volume hierarchy to use for collision detection.	[0,#BV Levels]

**Table 11-2** XML inputs to the position control system. This is class *EcPositionControlSystem*. A definition of the other member variables is available in the code documentation.

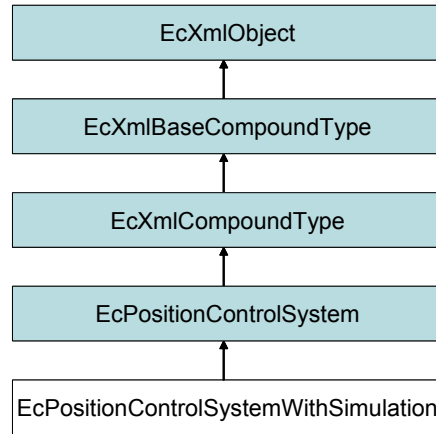
Method	Description
calculateState	Given a time, this method propagates the control system and state, and updates the end effector placements. This method returns an updated state through the argument list for rendering.
actualPlacementVector	Gets the achieved end effector placements
setActualPlacementVector	Sets the actual end effector placements
desiredPlacementVector	Gets the commanded end effector placements
setDesiredPlacementVector	Sets the commanded end effector placements
velocityControlSystem	Gets the velocity control system
setVelocityControlSystem	Sets the velocity control system
statedSystem	Gets the stated system
setStatedSystem	Sets the stated system
timeStep	Gets the time step
setTimeStep	Sets the time step
time	Gets the control system time
setTime	Sets the control system time
maxIterations	Gets the maximum iterations value
setMaxIterations	Sets the maximum iterations value
maxFinalPropagationSize	Gets the maximum propagation size
setMaxFinalPropagationSize	Sets the maximum propagation size
useTwoPasses	Gets the two pass flag
setUseTwoPasses	Sets the two pass flag
isOn	Gets the position control activation flag
setIsOn	Sets the position control activation flag

**Table 11-3:** Listing of primary methods available to the developer. Check the code documentation for a complete list and description.

Text Box 2-4 and Text Box 2-5 give a useful example for defining the commanded and actual end effector positions and then propagating the position control system for driving the end effectors to the commanded positions.

### 11.3 Position Control System With Look-Forward Simulation

The position control system with a look-forward simulation class is subclassed from *EcPositionControlSystem* (see Figure 11-1). The *setDesiredPlacementVector* method is overridden to look-forward before setting the internal command placements. If the simulation is not able to produce the commanded placements, the internal command placement is set to the current placement. This produces a halt.



**Figure 11-1:** Class inheritance diagram for *EcPositionControlSystemWithSimulation*.

Table 11-4 shows the XML configuration parameters and Table 11-5 shows the primary methods available to the developer.

Member Data	Class/Type	Definition	Restrictions
m_Simulation	<i>EcSystem... Simulation</i>	Simulation definition for position control system.	None
m_SimRunTime	<i>EcXmlReal</i>	Maximum simulation run-time for looking forward. If zero, the run-time is estimated.	None
m_SimRunTime... ScaleFactor	<i>EcXmlReal</i>	Scale factor for increasing the simulation run time beyond the estimate. Only used if simRunTime is zero.	None
m_Steps	<i>EcXmlU32</i>	Number of steps to iterate through simRunTime.	None
m_Tol	<i>EcXmlReal</i>	Tolerance for testing the convergence of the actual end effector positions with the commanded positions.	None

**Table 11-4:** XML inputs to the look-forward position control system. See Text Box 11-1 for a list of defaults. A definition of the other member variables is available in the code documentation.

```

<ct:positionControlSystemWithSimulation>
  <ct:isOn>1</ct:isOn>
  <ct:maxFinalPropagationSize>2</ct:maxFinalPropagationSize>
  <ct:maxIterations>16</ct:maxIterations>
  +<ct:simulationControl>
    <ct:simulationRunTime>0</ct:simulationRunTime>
    <ct:simulationRunTimeScaleFactor>10
      </ct:simulationRunTimeScaleFactor>
    <ct:simulationSteps>30</ct:simulationSteps>
    <ct:simulationTolerance>0.0001</ct:simulationTolerance>
    <ct:timeStep>0.012</ct:timeStep>
    <ct:useTwoPasses>1</ct:useTwoPasses>
  +<ct:velocityControlSystem>
</ct:positionControlSystemWithSimulation>

```

**Text Box 11-1:** Configuration file snippet for position control system with simulation. This includes the base class inputs from *EcPositionControlSystem*. The inputs with a “+” have a more complex definition that is expandable into a hierarchy of XML tags.

There are five user inputs added to the position control system configuration through this class as defined in Table 11-4. *simulationControl* is the most complicated input. It contains a complete definition of the simulation. In general, it is defined similarly to the top level simulation that includes position control, except that this simulation under position control does not include another simulation within its position control. In other words, the simulation is not included recursively. *simulationRunTime* defines how far to look ahead. If the position successfully converges on the commanded position prior to *simulationRunTime*, the simulation will exit early which reduces the processing time needed for looking ahead. There is an alternative method for determining the run time which is established through *simulationRunTimeScaleFactor*. If *simulationRunTime* is zero, this method becomes active. Internal to the code, there is a method called *minimumTime* which estimates the minimum time necessary to achieve the commanded position. Since the simulation needs to run for an undetermined time longer than the minimum time, there is a scale factor available for increasing the calculated time. The *simulationRunTime* approach is the default, since its use is more intuitive when used interactively such as with the ActinViewer. The value *simulationSteps* defines the number of time steps to use when marching forward through the simulation. The value *simulationTolerance* defines the difference required between the commanded and achieved position. If this tolerance is achieved, the position controller proceeds to execute the command. If the tolerance is not achieved, the position controller commands a halt.

Method	Description
setDesiredPlacementVector	This method checks to set that the desired placement vector is achievable and then sets it. If it is not achievable, the internal desired placement vector is set to the current position which stops any motion.
setSafePlacementVector	This method sets the internal desired placement vector to the current end effector positions.

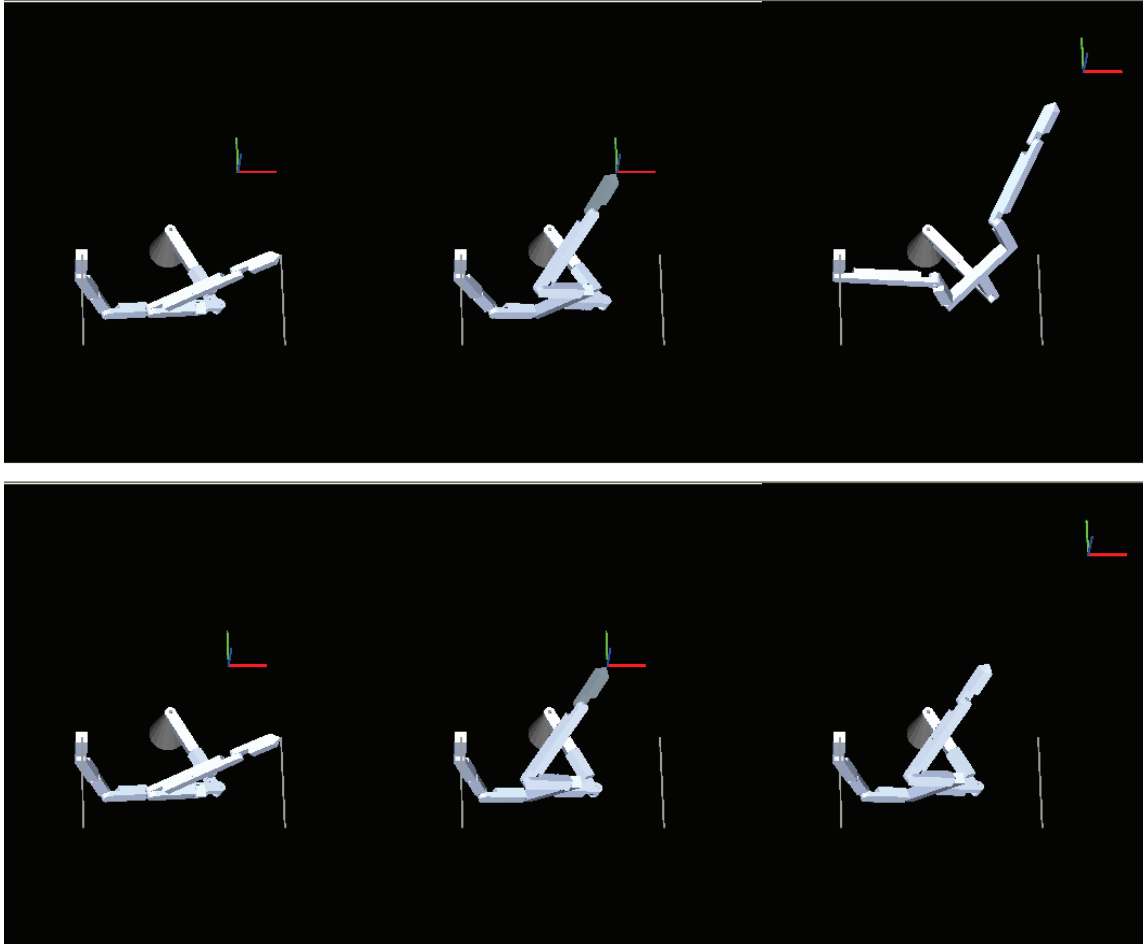
setSimulation	Sets the simulation description.
setSimRunTime	Sets the maximum simulation run time for looking ahead.
setSimRunTimeScaleFactor	Sets the scale factor parameter.
setSteps	Sets the number of steps to iterate through the simulation time.
setTol	Sets the tolerance parameter.

**Table 11-5:** Listing of primary methods available to the developer for the look-forward position control system. Check the code documentation for a complete list and description.

The method, *setDesiredPlacementVector*, takes a desired placement vector, tests it with the simulation, and optionally sets the desired placement vector or calls the *setSafePlacementVector* method. The *setSafePlacementVector* is called if the simulation predicts that the desired placement vector is not achievable. The *setSafePlacementVector* sets the desired placement vector to the current placement vector (which produces a halt). Both the *setDesiredPlacementVector* and *setSafePlacementVector* methods are virtual and can be overridden through inheritance. The *setSafePlacementVector* method can be overridden to produce different commands, as needed for the user's specific task, when a failure to converge occurs.

Other than the "set" methods for setting the internal parameters, the interface to the position control system with simulation class is the same as the basic position control system class. For that reason, Text Box 2-4 Text Box 2-5 also provide an instructive example for this class. Figure 10-2 shows an illustration of how the two position control algorithms react to position commands.





**Figure 11-2:** Illustration of position control with look-forward simulation. The top row of pictures shows three end effector placements under the control of the basic position control system. The first picture shows the starting location with a guide frame at a desired end effector position. The second picture shows that the achieved position converges on the desired position. The third picture shows a desired end effector position that is out of reach. The manipulator reaches as far as it can and stops at its joint and link limits. The bottom row shows a similar command sequence using the position control system with a look-forward simulation. The third picture shows that the manipulator was halted because the simulation determined that the commanded position was out of reach. This same concept is used to aid in avoiding collisions.

## 12 Dynamic Simulation

### 12.1 Force Response

Accurate force response for the dynamic simulation is a key requirement for testing collision avoidance as well as force control and grasping. Since the application requires modeling of force as a function of material type for grasping, manipulating and other dexterous operations, a physics-based force response model that returns a force as a function of the force applied is needed. This requires the application to forgo the myriad fast methods available for handling collision response that have been developed using a coefficient of restitution model. Coefficient of restitution models are common in the video game industry where fast execution times are of more concern than accurate physical representation.

#### 12.1.1 Architecture

Since several models may be employed for force response, and developers of the toolkit may want to add their own, the architecture was constructed to be easily extendable.

All force processors inherit from *EcBaseLinkInteractions*. A vector of link interactions (*EcLinkInteractionsVector*) is traversed in *EcDynamicSimulatorSystem* at the time step specified for the dynamic simulator system. Note that this time step is independent of the simulation time step for the individual manipulators. If a new force processor is developed, it needs only to subclass *EcBaseLinkInteractions*, and be registered with the *EcLinkInteractionsVector*. This can be done by outside developers of the toolkit using the plugin interface.

The force processing is done in *EcLinkCollisionForce*, which subclasses *EcBaseLinkInteractions* as discussed. The description of the class is below.

Member Data	Class/Type	Description	Restrictions
CollisionForceProc... Container	<i>EcCollisionForce... ProcessorContainer</i>	Holds a container for the collision force processor	None

**Table 12-1:** *EcLinkCollisionForce* class description.

By employing a container class for the collision force processor, *EcLinkCollisionForce* has the greatest flexibility. *EcCollisionForceProcessorContainer* is a standard container that subclasses *EcBaseExpressionTreeContainer<EcBaseCollisionForceProcessor>*. The base collision force processor has the following members.

Member Data	Class/Type	Description	Restrictions
pContainer	<i>EcCollisionForce... ProcessorContainer</i>	A pointer to the collision force processor container	None
DissipativeForce...	<i>EcDissipativeForce</i>	A container holding the	None

Container	<i>ProcessorContainer</i>	class for computing dissipative (non-conservative) forces	
-----------	---------------------------	---	--

**Table 12-2:** *EcBaseCollisionForceProcessor* class description.

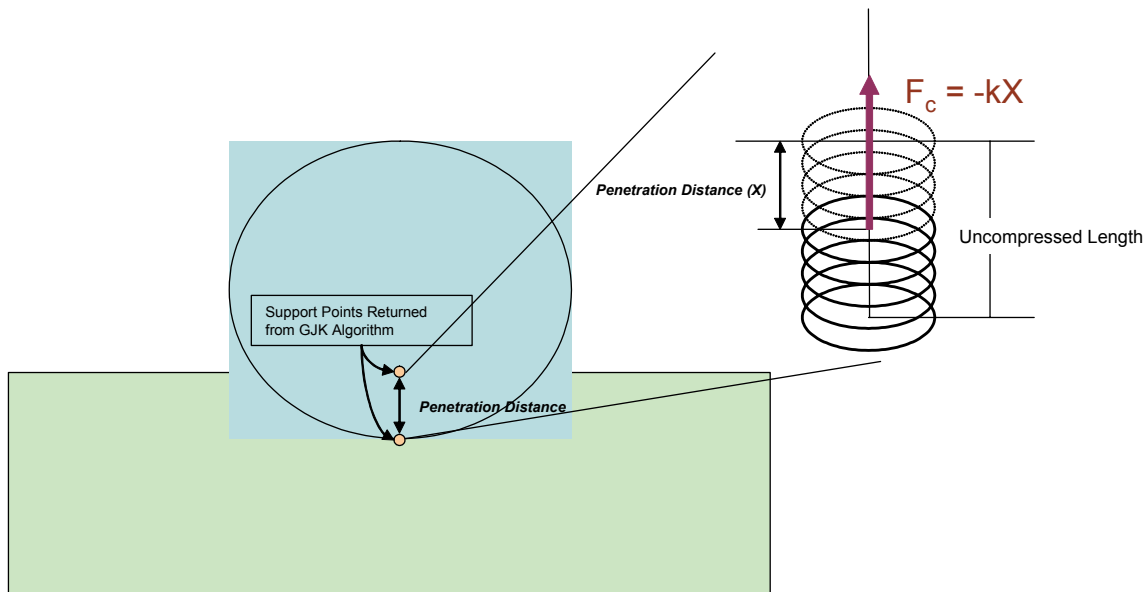
An *EcDissipativeForceProcessorContainer* is a standard container that subclasses *EcBaseExpressionTreeContainer*<*EcBaseDissipativeForceProcessor*>.

By architecting the system in this way, developers have complete control over the force feedback algorithms used. It accommodates, for example, changing the way that frictional forces are calculate without having to change other aspects of the force calculation.

### 12.1.2 Spring Displacement Model

A simple spring displacement model for the collision response was implemented that relies on knowledge of the penetration distance between two intersecting physical extents. This model has the advantage of returning force as a function of force applied and is well suited for the robotics application domain.

Figure 12-1 shows a graphical representation of the displacement model. The line of action **1** is the line normal to both colliding surfaces and points in the direction of the resulting force for Surface 1, and in the direction opposite the force for Surface 2. The line of action is obtained as a byproduct of the GJK penetration depth calculation.



**Figure 12-1:** Spring Displacement Model for Force Computation.

The force is computed in system coordinates as.

$$F_c = -kX\mathbf{1} \quad (12-1)$$

Where  $X$  is the penetration depth,  $\mathbf{1}$  is the line of action, and  $k$  is the resultant spring constant for the two surfaces. The spring constant for each physical extent is described in XML as

**ec\_surface\_tension\_spring\_constant** and is part of the surface properties description. The resultant spring constant  $k$  is obtained by:

$$k = \frac{k_1 k_2}{k_1 + k_2} \quad (12-2)$$

Where  $k_1$  is the spring constant for shape 1 and  $k_2$  is the spring constant for shape 2.

### 12.1.3 Non-Conservative Forces

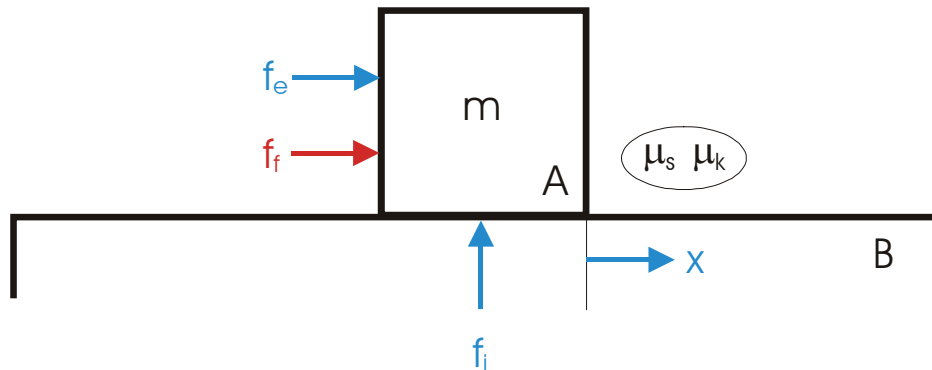
#### 12.1.3.1 Design

Modeling static friction is difficult [20]. A number of approaches are available for approximation, but these usually require optimization (such as in [21]) or the addition of new states (such as in [22]). This document describes a design for implementing a static and kinetic Coulomb friction model tailored for use with a robotic manipulator. The technique is based on the concept of a breaking spring.

#### 12.1.3.2 One-Dimensional Model

To illustrate the breaking-spring approach to friction modeling, this section shows first how the technique would be applied to a one-dimensional problem. The next section will extend the technique to three dimensions.

Let two interacting surfaces be labeled A and B, as shown in the figure below. The normal force applied by surface B on surface A is  $f_i$ . The horizontal location of the block is represented through  $x$ ; the external horizontal force applied to the block is  $f_e$ , and the friction force applied to the block is  $f_f$ . The coefficients of static and kinetic friction are  $\mu_s$  and  $\mu_k$ , respectively. The mass of the block is  $m$ .



**Figure 12-2:** A one-dimensional example. Object A can move horizontally relative to object B.

The Coulomb friction model gives the following constraints:

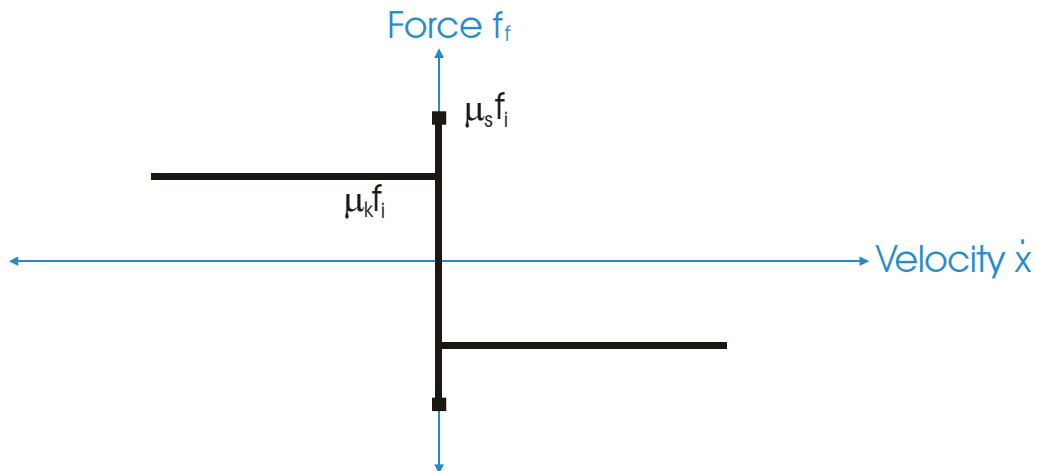
$$|f_f| < \mu_s f_i \text{ when } \dot{x} = 0, \quad (12-3)$$

$$f_f = -\text{sign}(\dot{x}) \mu_k f_i \text{ when } \dot{x} \neq 0, \quad (12-4)$$

along with the constraint that friction can do no work:

$$f_f \dot{x} \leq 0. \quad (12-5)$$

The constraint from equations (12-3) and (12-4) gives force as a function of velocity as follows:



**Figure 12-3:** The relationship between force and velocity for the one-dimensional Coulomb friction model.

To form an analogy between this one-dimensional example and the dynamic manipulator model used in the toolkit, the friction force will be estimated as a function of the position  $x$  and velocity  $\dot{x}$ . For this, the friction model will be assigned a state that takes two values, kinetic and static. The kinetic mode will represent object A moving relative to object B, and static mode will represent object A stationary relative to object B.

#### **12.1.3.2.1 In the Kinetic Friction State**

In kinetic mode, the friction will be calculated as

$$f_f = -\text{sign}(\dot{x}) \mu_k f_i. \quad (12-6)$$

This is exactly consistent with the constraints of equations (12-3) to (12-5).

#### **12.1.3.2.2 In the Static Friction State**

The static friction state, the friction force will be represented using a spring-damper response. The spring and damper parameters will be  $k_s$  and  $\lambda_s$ . Both of these will be strictly nonnegative. The force will then be calculated as follows in static mode:

$$f_f = -k_s x - \lambda_s \dot{x}. \quad (12-7)$$

The transfer function for this system is

$$\frac{X}{F_e} = \frac{\frac{1}{m}}{s^2 + \frac{\lambda_s}{m}s + k_s}. \quad (12-8)$$

Let  $\lambda_s$  be calculated as

$$\lambda_s = 2\hat{m}\sqrt{k_s}, \quad (12-9)$$

Where  $\hat{m}$  is the estimate of  $m$ . With this value, (12-72) is critically damped when  $m = \hat{m}$ . The damping factor,  $\zeta$ , is given by

$$\zeta = \frac{\hat{m}}{m} \quad (12-10)$$

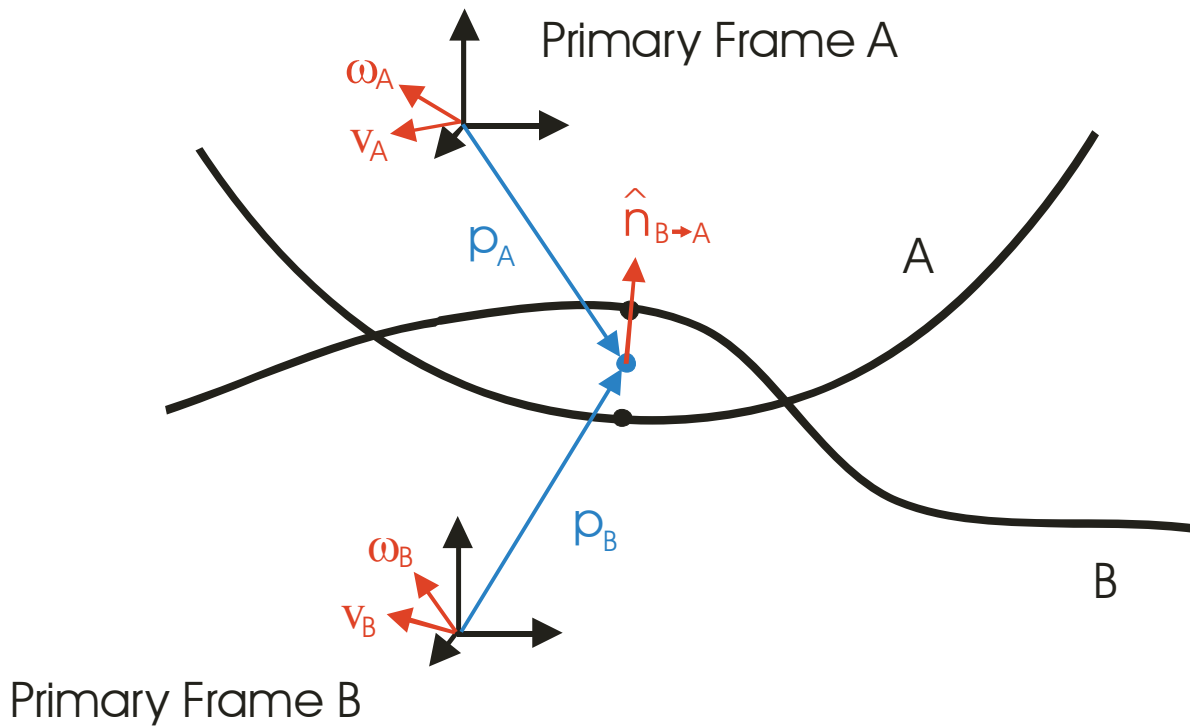
The settling time for an underdamped system is inversely proportional to the damping factor.

### 12.1.3.2.3 *Choosing the Friction State*

Initially, the friction state is static. If the force calculated through (12-71) ever exceeds  $\mu_s f_i$ , then the mode changes to kinetic. If the sign of  $\dot{x}$  changes, the mode changes to static.

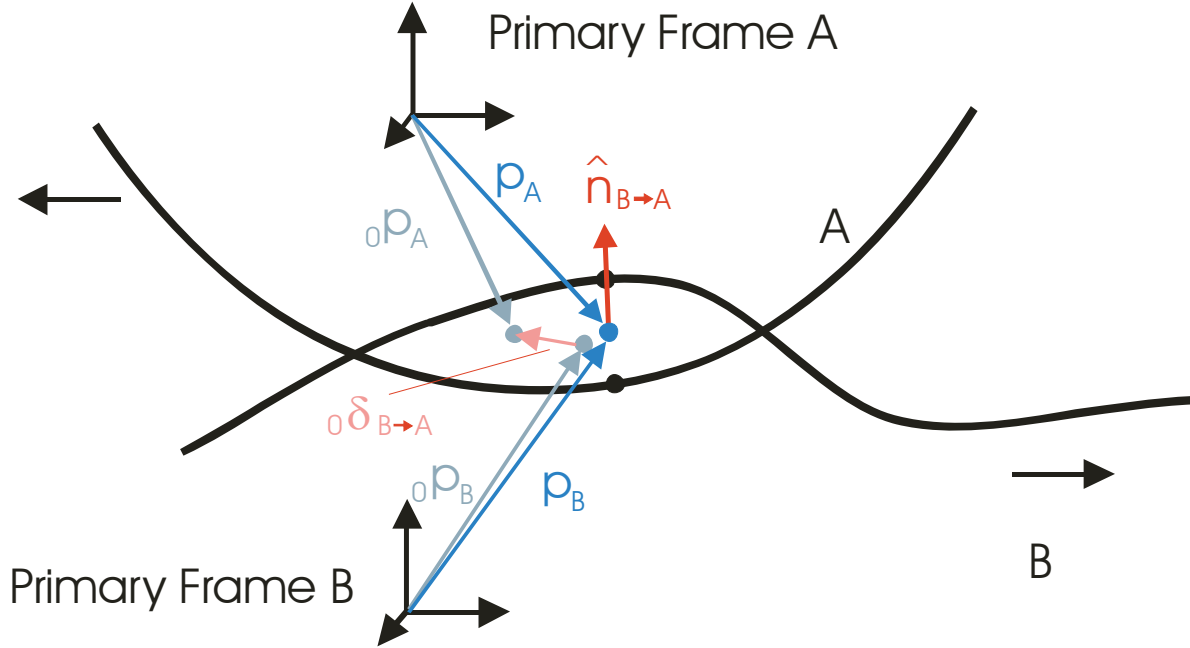
### 12.1.3.3 **Three-Dimensional Model**

For the three-dimensional model, the situation is illustrated in the figure below. The contact point is the spatial point midway between the deepest intersections of the two objects. The location of this point as represented in the two object primary frames is given by  $\vec{p}_A$  and  $\vec{p}_B$ . The normal  $\hat{n}_{B \rightarrow A}$  is a unit vector along the axis defined by the two intersection points.



**Figure 12-4:** For two intersecting 3D objects, the midpoint between the deepest intersection points is represented in both primary frames as  $\vec{p}_A$  and  $\vec{p}_B$ .

For calculating the friction between the two objects, the locations of the contact point in the two reference frames ( $\vec{p}_A$  and  $\vec{p}_B$ ) are saved as  ${}_0\vec{p}_A$  and  ${}_0\vec{p}_B$ . This is illustrated in the figure below. As the two objects move, this point moves distinctly for the two objects, staying constant in each object's primary frame. The location of the point is stored for the first time step that is part of a static-friction mode.



**Figure 12-5:** For two intersecting 3D objects, the midpoint between the deepest intersection points is represented in both primary frames and this representation moves as the objects move.

The following quantities are saved each from the first occurrence of a static mode:

- The locations of the contact points in primary frame coordinates,  $\vec{p}_A$  and  $\vec{p}_B$ , as  ${}^0\vec{p}_A$  and  ${}^0\vec{p}_B$ .
- Frame A represented in frame B,  ${}^B\mathbf{T}_A$ .
- The general velocity of frame A with respect to frame B,  ${}^B\mathbf{V}_A$ .

### 12.1.3.3.1 In the Kinetic Friction State

Let the linear kinetic friction coefficient be  $\mu_k$  and the angular kinetic friction coefficient be  $\mu_{a,k}$ . In the kinetic friction state, the force will be calculated as described below.

Let the general force applied by object B to object A as a nonfrictional response to the collision of objects A and B be defined as  ${}^B\mathbf{F}_A$ . If this general force is expressed at the collision point, the value is  ${}^B\mathbf{F}_A^c$ , which has two vector components, linear force  ${}^B\vec{f}_A^c$  and angular moment of force  ${}^B\vec{n}_A^c$ . With these values, let the normal force magnitude be defined as follows:

$$f_i = {}^B\vec{f}_A^c \cdot \hat{n}_{B \rightarrow A}, \quad (12-11)$$

where  $\hat{n}_{B \rightarrow A}$  is the normal defined by the intersection points (see Figure 12-5).

The general velocity of frame A with respect to frame B expressed at the contact point is given by  ${}^B\mathbf{V}_A^c$ . This has two vector components, linear velocity  ${}^B\vec{v}_A^c$  and angular velocity  ${}^B\vec{\omega}_A^c$ . The linear



component of the velocity of B relative to A at the contact point perpendicular to the normal is given by

$${}^B\vec{v}_A^\perp = {}^B\vec{v}_A^c - ({}^B\vec{v}_A^c \cdot \hat{n}_{B \rightarrow A}) \hat{n}_{B \rightarrow A}. \quad (12-12)$$

And the angular component of the velocity of B relative to A at the contact point along the normal is given by

$${}^B\vec{\omega}_A^n = ({}^B\vec{\omega}_A^c \cdot \hat{n}_{B \rightarrow A}) \hat{n}_{B \rightarrow A}. \quad (12-13)$$

With this, the force applied to object B by object A at the contact point is

$$\vec{f}_{B \rightarrow A} = -\frac{f_i \mu_k}{\|{}^B\vec{v}_A^\perp\|} {}^B\vec{v}_A^\perp. \quad (12-14)$$

And the angular moment applied to object B by object A is

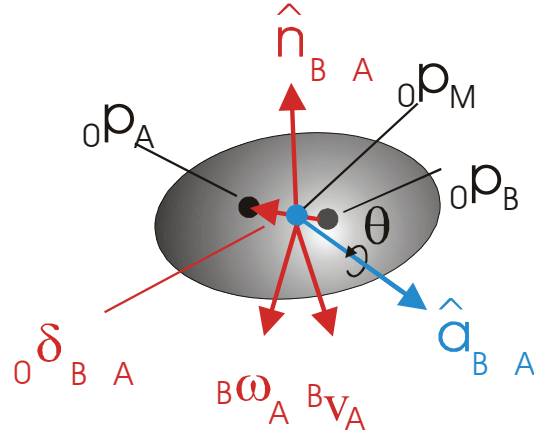
$$\vec{n}_{B \rightarrow A} = -\text{sign}({}^B\vec{\omega}_A^c \cdot \hat{n}_{B \rightarrow A}) f_i \mu_a \hat{n}_{B \rightarrow A}. \quad (12-15)$$

### 12.1.3.3.2 *In the Static Friction State*

For the static case, let the following parameters be defined:

- $\hat{n}_{B \rightarrow A}$  is a unit vector along the axis defined by the two current intersection points;
- ${}_0\vec{p}_A$  and  ${}_0\vec{p}_B$  are the locations of the start of the most recent static-friction state;
- ${}_0\vec{\delta}_{B \rightarrow A}$  is the different between  ${}_0\vec{p}_A$  and  ${}_0\vec{p}_B$  when they are represented in common coordinates using the current locations of objects A and B;
- ${}^B\vec{v}'_A$  and  ${}^B\vec{\omega}'_A$  are the linear and angular velocity of object A with respect to object B expressed at the point intermediate between  ${}_0\vec{p}_A$  and  ${}_0\vec{p}_B$  when they are represented in common coordinates;
- $\hat{a}_{B \rightarrow A}$  and  $\theta$  are the angle and axis between the orientation of object A with respect to B at the start of the current static mode and the current orientation of object A with respect to object B; and
- $\mu_{s,t}$  and  $\mu_{s,n}$  are the coefficients of static friction tangential and normal, respectively, to the contact between objects A and B.
- $k_{s,t}$  and  $k_{s,n}$  are the spring coefficients used to model tangential and normal static friction, respectively, between objects A and B.
- $\lambda_{s,t}$  and  $\lambda_{s,n}$  are the damper coefficients used to model tangential and normal static friction, respectively, between objects A and B.
- $k_{s,a}$  and  $\lambda_{s,a}$  are the spring and damper coefficients for the angular component of static friction.

These quantities are illustrated in the figure below.



**Figure 12-6:** For two intersecting 3D objects, the midpoint between the deepest intersection points is represented in both primary frames. The mean is  $0\vec{p}_M$ .

Let  $0\vec{\delta}_{B \rightarrow A}^n$  be defined as the normal component of  $0\vec{\delta}_{B \rightarrow A}$ , i.e.,

$$0\vec{\delta}_{B \rightarrow A}^n = (0\vec{\delta}_{B \rightarrow A} \cdot \hat{n}_{B \rightarrow A}) \hat{n}_{B \rightarrow A}. \quad (12-16)$$

And let  $0\vec{\delta}_{B \rightarrow A}^\perp$  be the perpendicular component, calculated as

$$0\vec{\delta}_{B \rightarrow A}^\perp = 0\vec{\delta}_{B \rightarrow A} - 0\vec{\delta}_{B \rightarrow A}^n. \quad (12-17)$$

Similarly, let the normal and perpendicular components of the rotational axis  $\hat{a}_{B \rightarrow A}$  be defined as follows:

$$\hat{a}_{B \rightarrow A}^n = (\hat{a}_{B \rightarrow A} \cdot \hat{n}_{B \rightarrow A}) \hat{n}_{B \rightarrow A} \quad (12-18)$$

and

$$\hat{a}_{B \rightarrow A}^\perp = \hat{a}_{B \rightarrow A} - \hat{a}_{B \rightarrow A}^n. \quad (12-19)$$

The normal and perpendicular component of linear velocity are defined as follows

$$B\vec{v}_A'^n = (B\vec{v}_A' \cdot \hat{n}_{B \rightarrow A}) \hat{n}_{B \rightarrow A} \quad (12-20)$$

and

$$B\vec{v}_A'^\perp = B\vec{v}_A' - B\vec{v}_A'^n. \quad (12-21)$$

And the normal component of angular velocity is defined as follows

$$B\vec{\omega}_A'^n = (B\vec{\omega}_A' \cdot \hat{n}_{B \rightarrow A}) \hat{n}_{B \rightarrow A}. \quad (12-22)$$

With these, the force applied on object B by object A at the contact point is

$$\vec{f}_{B \rightarrow A} = -\left(k_{s,t}({}_0\bar{\delta}_{B \rightarrow A}^\perp) + \lambda_{s,t}({}^B\bar{v}_A'^\perp)\right) - \left(k_{s,n}({}_0\bar{\delta}_{B \rightarrow A}^n) + \lambda_{s,n}({}^B\bar{v}_A'^n)\right). \quad (12-23)$$

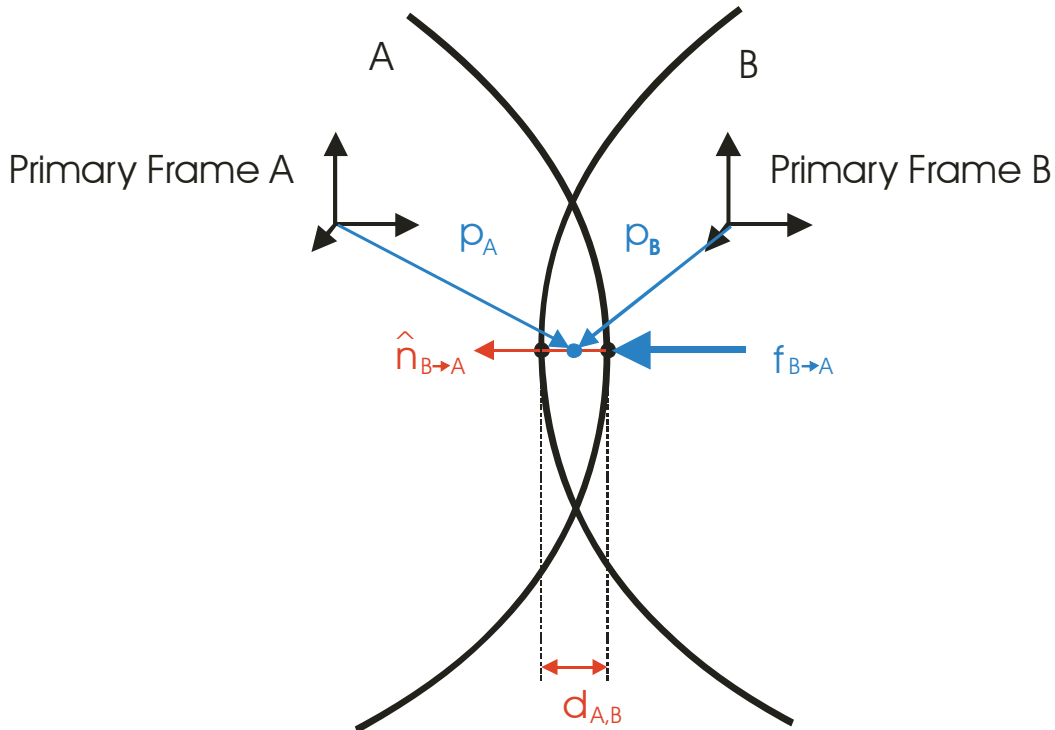
And the applied moment is

$$\vec{n}_{B \rightarrow A} = -k_{s,a}\theta(\hat{a}_{B \rightarrow A}^n) - \lambda_{s,a}({}^B\bar{\omega}_A'^n). \quad (12-24)$$

### 12.1.3.3.3 Choosing the Friction State

Initially, the friction state is static. If the magnitude of the normal force calculated through (12-23) ever exceeds  $\mu_{s,t}f_i$ , then the mode changes to kinetic. Similarly, if the magnitude of the angular moment in (12-24) ever exceeds  $\mu_{s,n}f_i$ , then the mode changes to kinetic. If  $\vec{f}_{B \rightarrow A}$  changes by more than 90 degrees while the direction of  $\vec{n}_{B \rightarrow A}$  changes in the same time step, the mode changes to static.

### 12.1.3.3.4 Collision Reaction



**Figure 12-7:** Each of two colliding objects applies a repelling force to the other. That force is calculated as a function of the penetration distance and the estimated normal between the two surfaces.

At any instant in time, let the penetration distance between bodies A and B be  $d_{A,B}$ , and let the normal between the surfaces, from B to A, be  $\hat{n}_{B \rightarrow A}$ , and the collision points in frames A and B be  $\vec{p}_A$  and  $\vec{p}_B$ . With this, a linear spring/damper repelling force applied by B to A at a given instance would be calculated as follows:

$$\vec{f}_{B \rightarrow A} = (k_r d_{A,B} + \lambda_r \dot{d}_{A,B}) \hat{n}_{B \rightarrow A}. \quad (12-25)$$

This linear force expressed in frame as a general force becomes

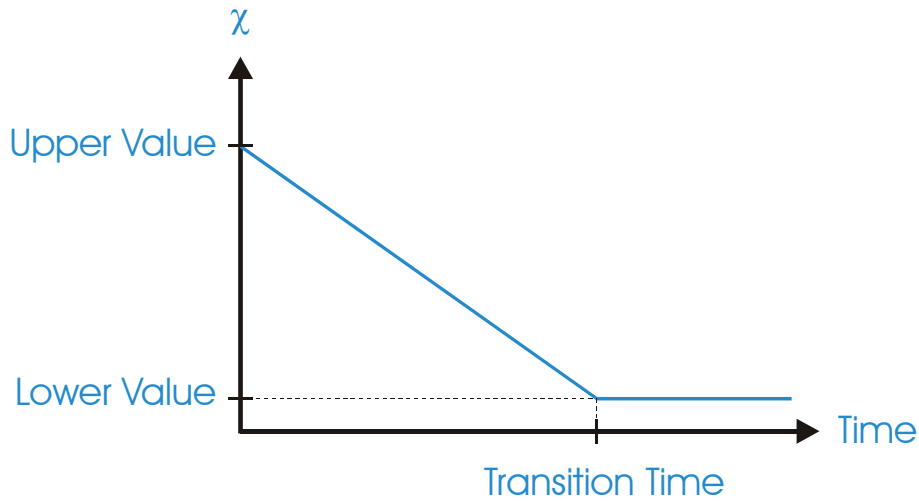
$${}^A \mathbf{F}_{B \rightarrow A} = \begin{bmatrix} \vec{f}_{B \rightarrow A} \\ \vec{p}_A \times \vec{f}_{B \rightarrow A} \end{bmatrix}. \quad (12-26)$$

Before application to body A in the simulation, this value is smoothed to integrate the effect of multiple or poorly defined collision points, as might occur when a flat face intersects another flat face. The smoothed value of  ${}^A \mathbf{F}_{B \rightarrow A}$  is  ${}^A \tilde{\mathbf{F}}_{B \rightarrow A}$ . This smoothing process is implemented using a smoothing parameter,  $\chi_k \in [0,1]$ , as follows:

$${}^A \tilde{\mathbf{F}}_{B \rightarrow A}^{k-1} = \mathbf{0}; \quad {}^A \tilde{\mathbf{F}}_{B \rightarrow A}^k = (1 - \chi_k) {}^A \tilde{\mathbf{F}}_{B \rightarrow A}^{k-1} + \chi_k {}^A \mathbf{F}_{B \rightarrow A}^k. \quad (12-27)$$

The collision point and normal are smoothed similarly.

Time step 0 corresponds to time  $t_c$ , the first instance of an ongoing collision between A and B. If the objects separate and recollide, the time step is again 0, and the smoothing process starts over. The smoothing value transitions from a high value to a low value as the time progresses, as shown below.



**Figure 12-8:** The smoothing parameter transitions from the high to the low value over a specified period of time during a continuous collision event.

## 12.2 Articulated Dynamics

Actin™ includes two dynamic simulation methods: the Articulated Body Inertia Algorithm and the Composite Rigid-Body Inertia Algorithm. The Order( $N$ ) Articulated Body Inertia algorithm is best for very large manipulators, while the Order( $N^3$ ) Composite Rigid-Body Inertia algorithm is best for smaller manipulators. These techniques are implemented for both fixed-base and free-base manipulators. This section describes these techniques for general free-base manipulators.

### 12.2.1 Composite Rigid-Body Inertia Simulation Algorithm

For fixed-base manipulators, dynamic simulation is implemented using an adaptation of the composite rigid-body algorithm [23], [24] for bifurcating manipulators. This algorithm runs in Order( $N^3$ ) time, for  $N$  links.

The fixed-base composite rigid-body inertia algorithm uses the following dynamics equation:

$$\boldsymbol{\tau} = \mathbf{M}(\mathbf{q})\ddot{\mathbf{q}} + \mathbf{C}(\mathbf{q})\dot{\mathbf{q}} + \mathbf{G}(\mathbf{q}) + \mathbf{B}, \quad (12-28)$$

where  $\boldsymbol{\tau}$  is the column vector of joint torques/forces,  $\mathbf{M}(\mathbf{q})$  is the manipulator inertia matrix,  $\mathbf{q}$  is the vector of joint position,  $\mathbf{C}(\mathbf{q})$  represents the Coriolis forces,  $\mathbf{G}(\mathbf{q})$  represents gravitational forces, and  $\mathbf{B}$  represents the effect of external forces applied to the arm's links. This equation, as shown, is only valid for a manipulator with a fixed base. When the base is free, it must be modified.

#### 12.2.1.1 Background

For any frames  $i$  and  $j$  that are rigidly connected, Let  ${}^j\mathbf{P}_{j \rightarrow i}$  be the cross-product matrix for  ${}^j p_{i \rightarrow j}$ , the vector from the origin of frame  $i$  to the origin of frame  $j$ , expressed in frame  $j$ . And let  ${}^j\mathbf{R}_i$  be the rotation matrix expressing frame  $i$  in frame  $j$ . Using this, let the matrices  ${}^F\mathbf{T}_{i \rightarrow j}$  and  ${}^A\mathbf{T}_{i \rightarrow j}$  be defined as follows:

$${}^F\mathbf{T}_{i \rightarrow j} = \begin{bmatrix} {}^j\mathbf{R}_i & 0 \\ {}^j\mathbf{P}_{j \rightarrow i} & {}^j\mathbf{R}_i \end{bmatrix}, \quad (12-29)$$

and

$${}^A\mathbf{T}_{i \rightarrow j} = \begin{bmatrix} {}^j\mathbf{R}_i & {}^j\mathbf{P}_{j \rightarrow i} & {}^j\mathbf{R}_i \\ 0 & & {}^j\mathbf{R}_i \end{bmatrix}. \quad (12-30)$$

These transformations produce the following equalities:

$$\mathbf{F}_j = {}^F\mathbf{T}_{i \rightarrow j} \mathbf{F}_i, \quad (12-31)$$

and

$$\mathbf{A}_j = {}^A\mathbf{T}_{i \rightarrow j} \mathbf{A}_i. \quad (12-32)$$

For any rigid body, let  $\vec{f}$  be the vector force applied to the link,  $\vec{n}$  be the moment,  $\vec{\omega}$  be the angular velocity,  $\vec{v}$  be the linear velocity,  $\vec{f}_a$  be an a priori external force applied to the body,  $\vec{n}_a$  be an a priori moment applied to the body,  $m$  be the mass,  $\mathbf{H}$  be the cross-product matrix for the first moment of inertia, and  $\mathbf{J}$  be the second moment of inertia. Then, the force/moment equations are given by the following:

$$\vec{f} = \mathbf{H}^T \dot{\vec{\omega}} + \vec{\omega} \times \mathbf{H}^T \vec{\omega} + m\dot{\vec{v}} - \vec{f}_e. \quad (12-33)$$

$$\vec{n} = \mathbf{J} \dot{\vec{\omega}} + \vec{\omega} \times \mathbf{J} \vec{\omega} + \mathbf{H} \dot{\vec{v}} - \vec{n}_e. \quad (12-34)$$

Let the 6×6 rigid-body inertia be defined as follows:

$$\mathbf{I}^C = \begin{bmatrix} m\mathbf{I} & \mathbf{H}^T \\ \mathbf{H} & \mathbf{J} \end{bmatrix}. \quad (12-35)$$

And let a bias frame force be defined as

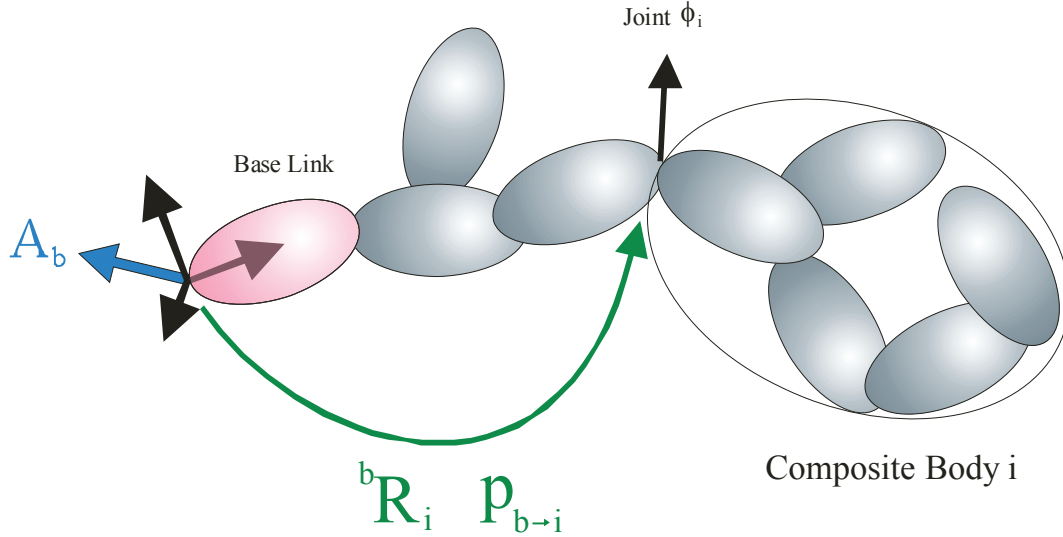
$$\mathbf{B} = \begin{bmatrix} \vec{\omega} \times \mathbf{H}^T \vec{\omega} \\ \vec{\omega} \times \mathbf{J} \vec{\omega} \end{bmatrix} + \begin{bmatrix} -\vec{f}_a \\ -\vec{n}_a \end{bmatrix}. \quad (12-36)$$

With this, the rigid-body dynamics can be represented as

$$\mathbf{F} = \mathbf{I}^C \mathbf{A} + \mathbf{B}. \quad (12-37)$$

### 12.2.1.2 Effect of Base Acceleration on Joint Torque

When the base link is free to move, the force on an acceleration of the base affects the manipulator dynamics. The affect of the base acceleration on any joint can be found by fixing all other joints and finding the torque required by the joint to sustain the acceleration on the composite rigid body outboard from the joint. This is illustrated in Figure 12-9.



**Figure 12-9:** The torque produced on joint  $i$  due to the acceleration of the base is the torque required to accelerate all the outboard links from the joint. This is an additive term, found by assuming all other joints are stationary.

For the base joint, let the a priori external force and moment be divided into two components. Let  $\vec{f}_e$  be an external force applied to the base,  $\vec{n}_e$  be an external moment applied to the body,  $\vec{f}_m$  be the force applied by child links to the base, and  $\vec{n}_m$  be the moment applied by child links. With this,  $\vec{f}_a = \vec{f}_e + \vec{f}_m$  and  $\vec{n}_a = \vec{n}_e + \vec{n}_m$ . Then the bias force is given by

$$\mathbf{B} = \begin{bmatrix} \vec{\omega} \times \mathbf{H}^T \vec{\omega} \\ \vec{\omega} \times \mathbf{J} \vec{\omega} \end{bmatrix} + \begin{bmatrix} -\vec{f}_e - \vec{f}_m \\ -\vec{n}_e - \vec{n}_m \end{bmatrix}. \quad (12-38)$$

The torque on joint  $i$  due to acceleration  $\mathbf{A}_b$  can be found by assuming an otherwise stationary manipulator, with all other rates and accelerations zero. With this assumption, the base frame acceleration can be expressed in frame  $i$  using (12-32), the frame force required to move the outboard composite rigid body calculated using (12-37), and the joint torque calculated by taking the inner product of this force with  $\phi_i$ . This gives the torque on joint  $i$  due to base acceleration as the following:

$$\tau_i^A = \phi_i^T \mathbf{I}_i^C ({}^A \mathbf{T}_{b \rightarrow i}) \mathbf{A}_b. \quad (12-39)$$

Let the matrix  $\mathbf{D}$  be defined as follows:

$$\mathbf{D} = \begin{bmatrix} \phi_0^T \mathbf{I}_0^C ({}^A \mathbf{T}_{b \rightarrow 0}) \\ \phi_1^T \mathbf{I}_1^C ({}^A \mathbf{T}_{b \rightarrow 1}) \\ \vdots \\ \phi_{n-1}^T \mathbf{I}_{n-1}^C ({}^A \mathbf{T}_{b \rightarrow n-1}) \end{bmatrix}. \quad (12-40)$$

This is evaluated in code by calculating  $\mathbf{I}_i^C \phi_i$  as the frame force produced by unit acceleration of joint  $i$  with all other joints stationary ( $\mathbf{I}_i^C$  is symmetric), then transforming this force to the base frame using the identity  ${}^A \mathbf{T}_{i \rightarrow j}^T = {}^F \mathbf{T}_{j \rightarrow i}$ ,  $\forall i, j$ .

With this definition, the manipulator dynamics equation becomes

$$\boldsymbol{\tau} = \mathbf{M}(\mathbf{q})\ddot{\mathbf{q}} + \mathbf{C}(\mathbf{q})\dot{\mathbf{q}} + \mathbf{G}(\mathbf{q}) + \mathbf{D} \mathbf{A}_b + \mathbf{B}. \quad (12-41)$$

Note that, because gravitational  $\mathbf{G}(\mathbf{q})$  is explicit,  $\mathbf{A}_b$  is the *seen* (rather than *felt*) value.

### 12.2.1.3 Changing Reference Frames for Matrix $\mathbf{D}$

Because  $\mathbf{D}$  applies to the base acceleration, it has a frame of representation, just as the base acceleration does. The formula to change the frame of expression for  $\mathbf{D}$  from  $j$  to  $i$  is given by the following (with  ${}^A \mathbf{T}_{i \rightarrow j}$  defined through (71)):

$${}^i \mathbf{D} = {}^j \mathbf{D} ({}^A \mathbf{T}_{i \rightarrow j}). \quad (12-42)$$

### 12.2.1.4 Effect Joint Accelerations on the Base Acceleration on Joint Torque

When the base link is free to move, assuming an otherwise stationary manipulator, the force that must be exerted on composite rigid body  $i$  for acceleration only by joint  $i$  is given by

$$F_i = \mathbf{I}_i^C \phi_i \ddot{q}_i. \quad (12-43)$$

Expressing this in the base frame, and changing the sign to represent the effective force *applied to* the base gives

$${}^b F_i = -{}^F \mathbf{T}_{i \rightarrow b} \mathbf{I}_i^C \phi_i \ddot{q}_i. \quad (12-44)$$

Summing the contributions of all links and using the identity  ${}^F \mathbf{T}_{i \rightarrow j}^T = {}^A \mathbf{T}_{j \rightarrow i}$ ,  $\forall i, j$  gives the following remarkable reuse of the matrix  $\mathbf{D}$  to calculate the force  $F_{ma}$  applied to the base as a result of manipulator joint accelerations.

$$\mathbf{F}_{ma} = -\mathbf{D}^T \ddot{\mathbf{q}}. \quad (12-45)$$

Let the total force applied by the manipulator to the base be

$$\mathbf{F}_m = \mathbf{F}_{ma} + \mathbf{F}_{mc} + \mathbf{F}_{mg} + \mathbf{F}_{me}, \quad (12-46)$$

where  $\mathbf{F}_{mc}$ ,  $\mathbf{F}_{mg}$ , and  $\mathbf{F}_{me}$  represent the force due to Coriolis and centripetal terms, gravity, and external forces, respectively.

If  $\mathbf{F}_e$  represents the external forces applied to the base link directly, then (78) gives



$$\mathbf{F}_{ma} + \mathbf{F}_e = \mathbf{I}_b^C \mathbf{A}_b - \mathbf{F}_{mc} - \mathbf{F}_{mg} - \mathbf{F}_{me}, \quad (12-47)$$

where  $\mathbf{I}_b^C$  is the composite rigid-body inertia of the entire manipulator, including the base.

Substituting in (86) gives

$$\mathbf{D}^T \ddot{\mathbf{q}} + \mathbf{I}_b^C \mathbf{A}_b = \mathbf{F}_e + \mathbf{F}_{mc} + \mathbf{F}_{mg} + \mathbf{F}_{me}. \quad (12-48)$$

Combining this with (82) gives a new manipulator dynamics equation

$$\begin{bmatrix} \mathbf{I}_b^C & \mathbf{D}(\mathbf{q})^T \\ \mathbf{D}(\mathbf{q}) & \mathbf{M}(\mathbf{q}) \end{bmatrix} \begin{bmatrix} \mathbf{A}_b \\ \ddot{\mathbf{q}} \end{bmatrix} = \begin{bmatrix} \mathbf{F}_e + \mathbf{F}_{mc} + \mathbf{F}_{mg} + \mathbf{F}_{me} \\ \boldsymbol{\tau} - \mathbf{C}(\mathbf{q})\dot{\mathbf{q}} - \mathbf{G}(\mathbf{q}) + \mathbf{B} \end{bmatrix}. \quad (12-49)$$

For  $N$  joint degrees of freedom,  $\mathbf{I}_b^C$  is  $6 \times 6$ ,  $\mathbf{D}(\mathbf{q})$  is  $N \times 6$ , and  $\mathbf{M}(\mathbf{q})$  is  $N \times N$ . Solving this for  $\mathbf{A}_b$  and  $\ddot{\mathbf{q}}$  and time integrating these gives the free-base composite rigid-body algorithm. In concatenating  $\mathbf{A}_b$  and  $\ddot{\mathbf{q}}$ ,  $\mathbf{A}_b$  is placed on top of  $\ddot{\mathbf{q}}$  to simplify the calculation of the Cholesky Decomposition of the left-hand matrix in (90). Note that this matrix, which must be inverted, is guaranteed not to be singular for a real system (otherwise acceleration could be achieved with no force).

### 12.2.1.5 Solving for the Accelerations

Let

$$\overline{\mathbf{M}}(\mathbf{q}) = \begin{bmatrix} \mathbf{I}_b^C & \mathbf{D}(\mathbf{q})^T \\ \mathbf{D}(\mathbf{q}) & \mathbf{M}(\mathbf{q}) \end{bmatrix}, \quad (12-50)$$

$$\overline{\mathbf{q}} = \begin{bmatrix} \mathbf{A}_b \\ \ddot{\mathbf{q}} \end{bmatrix}, \quad (12-51)$$

and

$$\overline{\boldsymbol{\tau}} = \begin{bmatrix} \mathbf{F}_e + \mathbf{F}_{mc} + \mathbf{F}_{mg} + \mathbf{F}_{me} \\ \boldsymbol{\tau} - \mathbf{C}(\mathbf{q})\dot{\mathbf{q}} - \mathbf{G}(\mathbf{q}) + \mathbf{B} \end{bmatrix}. \quad (12-52)$$

With these definitions, (90) becomes

$$\overline{\mathbf{M}}(\mathbf{q}) \overline{\mathbf{q}} = \overline{\boldsymbol{\tau}} \quad (12-53)$$

an  $(N+6)$ -dimensional fully constrained linear equation. Cholesky Decomposition (decomposition into a lower-triangular square root) is ideal for solving this for  $\overline{\mathbf{q}}$  because  $\overline{\mathbf{M}}(\mathbf{q})$ , like  $\mathbf{M}(\mathbf{q})$ , is positive definite. In this approach,  $\overline{\mathbf{M}}(\mathbf{q})$  is decomposed as follows:

$$\bar{\mathbf{M}} = \bar{\mathbf{L}} \bar{\mathbf{L}}^T, \quad (12-54)$$

with  $\bar{\mathbf{L}}$  lower triangular.

The constness of  $\mathbf{I}_b^C$  can be exploited in the calculation of  $\bar{\mathbf{L}}$ . Let  $\mathbf{I}_b^C$  be decomposed as

$$\mathbf{I}_b^C = \mathbf{L}_b \mathbf{L}_b^T, \quad (12-55)$$

with  $\mathbf{L}_b$  lower triangular.  $\mathbf{L}_b$  is constant and only needs to be calculated once.

Let  $\mathbf{E}$  be the  $N \times 6$  matrix satisfying the following:

$$\mathbf{L}_b \mathbf{E}^T = \mathbf{D}^T, \quad (12-56)$$

which can be solved using back substitution with  $\mathbf{L}_b$ , and let  $\mathbf{L}_M$  be defined such that

$$\mathbf{M}(q) - \mathbf{E} \mathbf{E}^T = \mathbf{L}_M \mathbf{L}_M^T. \quad (12-57)$$

This can be solved using Cholesky decomposition on an  $N \times N$  matrix.

With these values,  $\bar{\mathbf{L}}$  is evaluated as

$$\bar{\mathbf{L}} = \begin{bmatrix} \bar{\mathbf{L}}_b & \mathbf{0} \\ \mathbf{E} & \mathbf{L}_M \end{bmatrix}. \quad (12-58)$$

Using (76) and (96),

$$\bar{\mathbf{L}}_b = \begin{bmatrix} \sqrt{m} \mathbf{I} & \mathbf{0} \\ \frac{1}{\sqrt{m}} \mathbf{H} & \bar{\mathbf{L}}_J \end{bmatrix}, \quad (12-59)$$

Where  $\bar{\mathbf{L}}_J$  is lower triangular, and

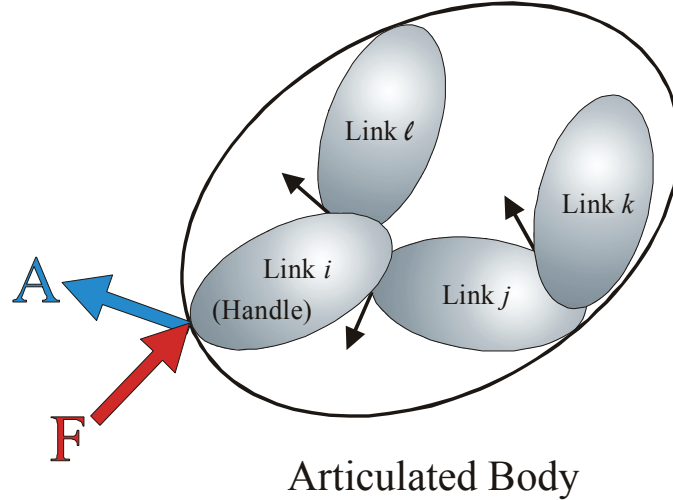
$$\bar{\mathbf{L}}_J \bar{\mathbf{L}}_J^T = \mathbf{J} - \frac{1}{m} \mathbf{H} \mathbf{H}^T. \quad (12-60)$$

## 12.2.2 Articulated-Body Simulation Algorithm

The articulated-body algorithm [25], [26] is used for simulation in the software toolkit, running in Order( $N$ ) time. This algorithm allows fast simulation of very complex mechanisms.

### 12.2.2.1 Articulated Body Inertia

An articulated body is a collection of connected articulating links, as illustrated in the figure below. Any single rigid body within the articulated body can be used as a handle for defining the relationship between force and acceleration.



**Figure 12-10:** The torque produced on joint  $i$  due to the acceleration of the base is the torque required to accelerate all the outboard links from the joint. This is an additive term, found by assuming all other joints are stationary.

Associated with any handle is a  $6 \times 6$  articulated-body inertia  $\mathbf{I}^A$  that satisfies the following equation for any physically realizable frame acceleration ( $6 \times 1$ )  $\mathbf{A}$ :

$$\mathbf{F} = \mathbf{I}^A \mathbf{A} + \mathbf{B}. \quad (12-61)$$

Here,  $\mathbf{F}$  is the  $6 \times 1$  frame force that must be applied to the articulated body to achieve  $6 \times 1$  frame acceleration  $\mathbf{A}$ . (For vector force  $\vec{f}$ , moment  $\vec{n}$ , linear acceleration  $\vec{a}$ , and angular acceleration  $\vec{\alpha}$ , represented as  $3 \times 1$  column vectors,  $\mathbf{F} = [\vec{f}^T \vec{n}^T]^T$  and  $\mathbf{A} = [\vec{a}^T \vec{\alpha}^T]^T$ .) The  $6 \times 1$  frame force  $\mathbf{B}$  is a bias force that is a function of external forces on the links, internal forces on the links, gravity, and link velocities.  $\mathbf{B}$  represents all contributors to the force on the link except acceleration  $\mathbf{A}$ . It is the force required to produce no acceleration of the handle.

The iterative formulas to calculate  $\mathbf{I}^A$  and  $\mathbf{B}$  are given by the following:

$$\mathbf{I}_k^A = (\mathbf{I}_j^A + \mathbf{I}_i^A - \frac{1}{\phi_j^T \mathbf{I}_j^A \phi_j} \mathbf{I}_j^A \phi_j \phi_j^T \mathbf{I}_j^A), \quad (12-62)$$

$$\mathbf{B}_k = \frac{1}{\phi_j^T \mathbf{I}_j^A \phi_j} (\tau_j - \phi_j^T \mathbf{B}_j) \mathbf{I}_j^A \phi_j + \mathbf{B}_j + \mathbf{B}_i. \quad (12-63)$$

When the base is free, these equations can be continued to the base link. Then, at the base link, the frame acceleration will be given by

$$\mathbf{A}_b = (\mathbf{I}_b^A)^{-1} (\mathbf{F}_b - \mathbf{B}_b). \quad (12-64)$$

Here,  $\mathbf{A}_b$  is the acceleration that is felt, rather than seen. If the frame is stationary with respect to a gravitational field, then  $\mathbf{A}_b$  will show acceleration upward.

### 12.2.3 Dynamics Example

An example manipulator system is provided with the toolkit showing the use of dynamic simulation. The XML configuration of this system is shown in Text Box 12-1, and

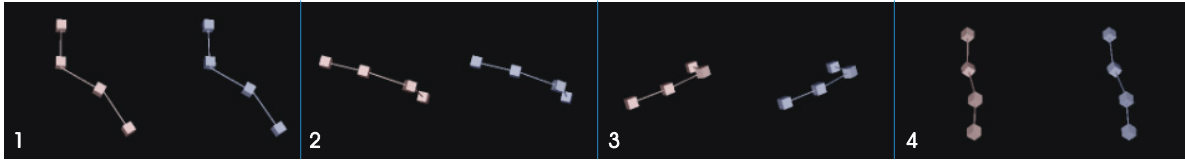
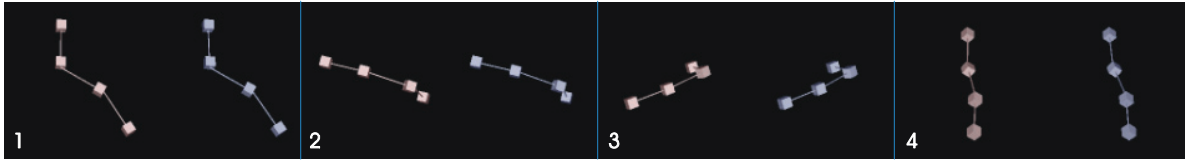


Figure 12-11 shows side-by-side comparisons of the two algorithms simulating a free-base three-link mechanism.

```
<dynamicSimulatorSystem>
  <dynamicSimSystemTimestep>0.001</dynamicSimSystemTimestep>
  <dynamicSimulators size="2">
    <element>
      <integrationMethod>adamsBashforth</integrationMethod>
      <integrationOrder>2</integrationOrder>
      <isOn>1</isOn>
      <simulationType>arbi</simulationType>
      <stateVariableBound>1000000000000000</stateVariableBound>
      <timeBetweenMassMatrixEvaluations>0</timeBetweenMassMatrixEvaluations>
      <timeStep>0.002</timeStep>
    </element>
    <element>
      <integrationMethod>adamsBashforth</integrationMethod>
      <integrationOrder>2</integrationOrder>
      <isOn>1</isOn>
      <simulationType>crbi</simulationType>
      <stateVariableBound>1000000000000000</stateVariableBound>
      <timeBetweenMassMatrixEvaluations>0</timeBetweenMassMatrixEvaluations>
      <timeStep>0.002</timeStep>
    </element>
  </dynamicSimulators>
</dynamicSimulatorSystem>
```

**Text Box 12-1:** Dynamic simulation configuration.



**Figure 12-11:** A side-by-side comparison of the dynamically simulated free-base mechanism using the Articulated Rigid-Body Algorithm and the Composite Rigid-Body Algorithm. The two algorithms give the same results.

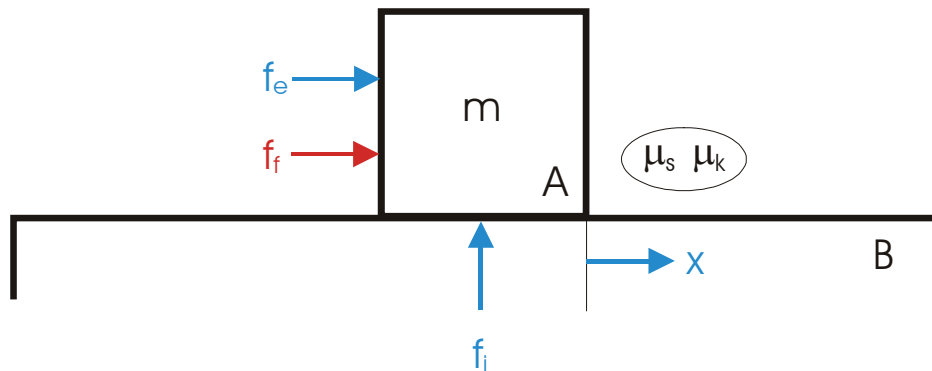
## 12.3 Actuator Modeling

### 12.3.1 Dry (Stick-Slip) Friction for Actuators

Dry friction is difficult to model. Without great care, the discretization errors in numerical integration can lead to friction forces that actually produce work or motion when none would actually occur. To avoid these difficulties, we have developed a novel dry friction model that combines the concepts developed by other researchers ([27],[28]) with a breaking spring-damper representation.

#### 12.3.1.1 Approach

Let two interacting surfaces be labeled A and B, as shown in the figure below. The normal force applied by surface B on surface A is  $f_i$ . The horizontal location of the block is represented through  $x$ ; the external horizontal force applied to the block is  $f_e$ , and the friction force applied to the block is  $f_f$ . The coefficients of static and kinetic friction are  $\mu_s$  and  $\mu_k$ , respectively. The mass of the block is  $m$ .



**Figure 12-12:** A one-dimensional example. Object A can move horizontally relative to object B. The Coulomb friction model gives the following constraints:

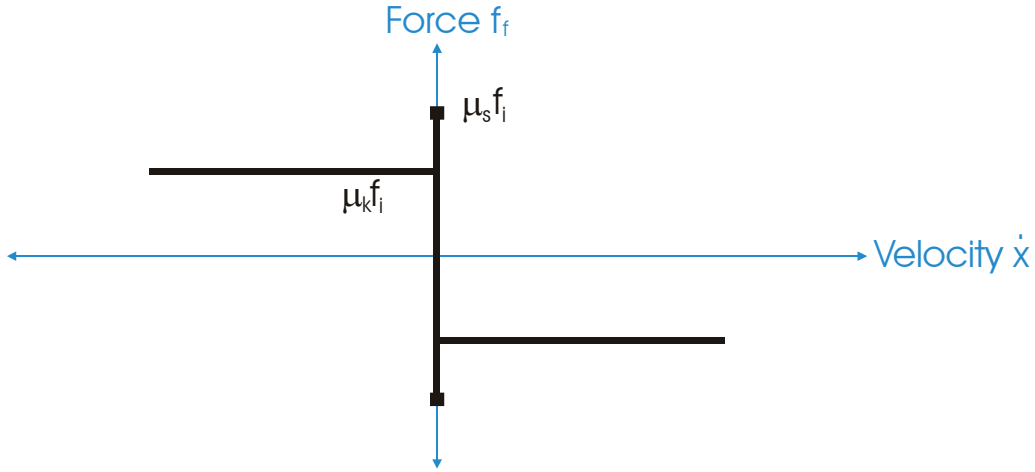
$$|f_f| < \mu_s f_i \text{ when } \dot{x} = 0, \quad (12-65)$$

$$f_f = -\text{sign}(\dot{x}) \mu_k f_i \text{ when } \dot{x} \neq 0, \quad (12-66)$$

along with the constraint that friction can do no work:

$$f_f \dot{x} \leq 0. \quad (12-67)$$

The constraint from equations (12-65) and (12-66) gives force as a function of velocity as follows:



**Figure 12-13:** The relationship between force and velocity for the one-dimensional Coulomb friction model.

In case of an actuator, let  $\dot{\theta}_m$  be the motor angular velocity;  $T$  be the total torque applied by or to motor;  $T_f$ ,  $T_{slip}$ , and  $T_{stick}$  denote the total, slip, and stick friction torques, respectively;  $T_s$  be the saturation torque of stick friction;  $T_k$  be the kinetic slip friction torque, and  $D_{\dot{\theta}}$  be the limiting angular velocity in the stick and slip regions. Then, the friction torque can be described as a combination of two components, both of which cannot occur simultaneously, as

$$T_f = T_{slip} + T_{stick} \quad (12-68)$$

where

$$T_{slip} = \begin{cases} 0, & -D_{\dot{\theta}} < \dot{\theta}_m < D_{\dot{\theta}} \\ -\text{sign}(\dot{\theta}_m) T_k, & \text{Otherwise} \end{cases} \quad (12-69)$$

and

$$T_{stick} = \begin{cases} -T_s, & T'_{stick} < -T_s \\ T'_{stick}, & -T_s < T'_{stick} < T_s \\ T_s, & T_s < T'_{stick} \end{cases} \quad (12-70)$$

with

$$T'_{stick} = \begin{cases} -T, & -D_{\dot{\theta}} < \dot{\theta}_m < D_{\dot{\theta}} \\ 0, & \text{Otherwise} \end{cases} \quad (12-71)$$

Essentially, if the motor velocity is within the limiting velocity  $D_{\dot{\theta}}$ , the motor is considered to be in the stick region. According to Karnopp and Choek, the motor angular velocity would have to be forced to zero if the momentum falls within a stick region. If, however, this was to be implemented, a sudden change of velocity could cause discontinuity in dynamic simulation. On the other hand, if the velocity was not forced to zero, the motor would continue to move even within stick region in which the total torque should be entirely countered by the stick friction. To circumvent this problem, we added a breaking spring-damper to help stop the motion. The breaking-spring damper torque is described by

$$T_{sd} = -k_s(\theta_m - \theta_{mc}) - k_d\dot{\theta}_m \quad (12-72)$$

where  $k_s$  is the spring constant;  $k_d$  the damper constant; and  $\theta_{mc}$  the motor angular displacement at the stick region. The purpose of this breaking spring-damper torque is to help bring the motor back to  $\theta_{mc}$ . With this, the stick friction is modified to be

$$T'_{stick} = \begin{cases} -T_s, & T'_{stick} < -T_s \\ T'_{stick} + T_{sd}, & -T_s < T'_{stick} < T_s \\ T_s, & T_s < T'_{stick} \end{cases} \quad (12-73)$$

### 12.3.1.2 Implementation

The dry friction model described in equations (12-68) to (12-73) is implemented in the *EcJointActuator* class. The methods listed in Table 12-3 are related for the implementation. One notable is that the saturation torque of the slip friction is not defined as absolute but rather as a percentage of that of the stick friction.

Methods	Description
<i>staticCoulombFriction</i> / <i>setStaticCoulombFriction</i>	Gets/sets the saturation torque of stick friction $T_s$
<i>kineticCoulombFrictionPercentage</i> / <i>setKineticCoulombFrictionPercentage</i>	Gets/sets the saturation torque of slip friction $T_k$ as a percentage of $T_s$
<i>staticCoulombSpringCoefficient</i> / <i>setStaticCoulombSpringCoefficient</i>	Gets/sets the spring constant used in the breaking spring/damper $k_s$
<i>staticCoulombDamperCoefficient</i> / <i>setStaticCoulombDamperCoefficient</i>	Gets/sets the damper constant used in the breaking spring/damper $k_d$
<i>limitingVelocityStickRegion</i> /	Gets/sets the limiting velocity indicating whether the

<i>setLimitingVelocityStickRegion</i>	actuator is in the stick or slip region $D_{\dot{\theta}}$
---------------------------------------	--

**Table 12-3:** Methods of *EcJointActuator* relevant to computing dry friction.

The implementation is mostly straightforward except for a few details. First, the center for the breaking spring-damper model must be updated as needed during run time. This required a state for actuators separate from manipulators. As such, a new class *EcActuatorState* was created to contain all necessary state information for actuators. In this case, *EcActuatorState* contains the center of the breaking spring-damper and a flag indicating whether dry friction is in the stick or slip region. The center position gets updated whenever the motor enters the stick region.

Secondly, obtaining the correct value of the motor torque  $T$  used in equation (12-71) is not trivial. This is the total torque (excluding the dry friction itself) applied to or by the motor, which include the input torque from the motor, the actuator viscous friction torque, the torque from stopper dynamics, and the torque from manipulator dynamics (e.g. from Coriolis and gravitational forces). It is the torque from manipulator dynamics that adds a level of complexity to the calculation of dry friction.

Two methods for manipulator dynamics calculation exist in our software toolkits. One is composite-rigid body inertia (CRBI) and the other articulated-body inertia (ARBI). Therefore, the support for both of these two methods must be provided. For the CRBI dynamics algorithm implemented in the *EcCrbAccelerationTool* class, the torque from manipulator dynamics was already explicitly computed in *EcTotalTorqueTool*. Thus, including the dry friction model is simple and does not increase any computation burden. For ARBI, this torque was not computed explicitly and therefore its calculation had to be added. An instance of *EcTotalTorqueTool* was added to *EcArbiAccelerationTool* to compute the torque from manipulator dynamics. This addition will undoubtedly decrease the speed of ARBI algorithm. However, since the torque calculation runs in  $O(N)$  time, the drop-off in performance is not expected to be significant.

Adding the dry friction model to joint actuators did require some changes in the way the total torques were calculated. Originally, the *EcTotalTorqueTool::torques()* method was intended to compute the total torques that would be required to achieve the specified motion (acceleration). This took into account the Coriolis/centripetal and gravity effects and actuator dynamics including viscous friction, stopper dynamics, and motor inertia as well. This method was primarily used in dynamic simulation and feed-forward joint control of the manipulator. However, the types of torques required for dynamic simulation and feed-forward joint control are not identical. A new method called *setTotalTorqueType()* was added to *EcTotalTorqueTool* for setting the desired torque type. The argument to the *setTotalTorqueType()* method is an enumeration type with the three options shown in Table 12-4. This method should be called prior to a call to the *torques()* method.

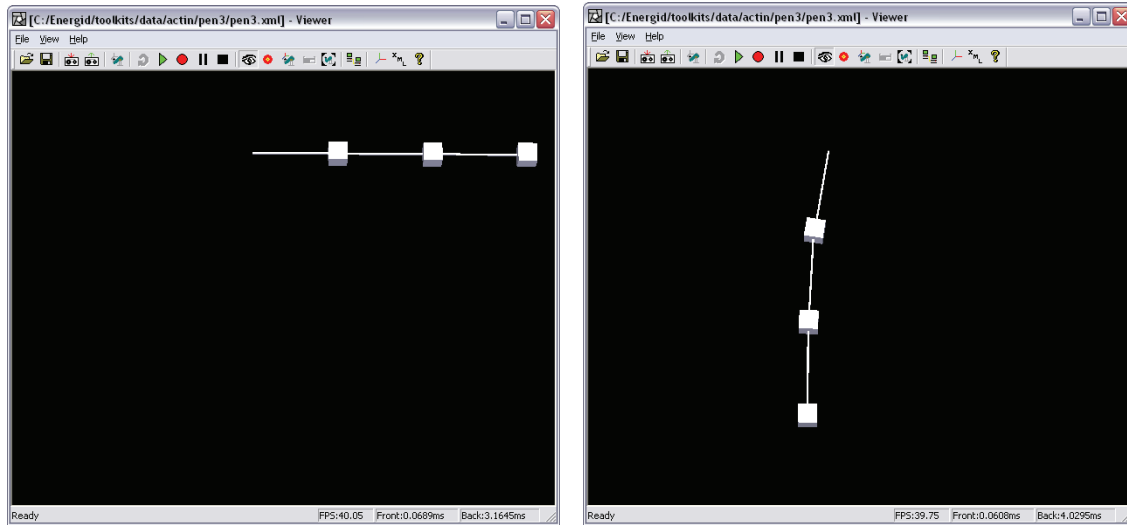
Total Torque Type	Description
TOTAL_TORQUE	Computes the torques due to manipulator dynamics and all of actuator dynamics. General purpose for computing torques required to achieve the desired acceleration.
TOTAL_LINEAR_TORQUE	Computes the torques due to manipulator dynamics and linear parts of actuator dynamics. Used in feed-forward joint control.
MANIPULATOR_TORQUE	Computes the torques due to manipulator dynamics only. Used in dynamic simulation. The torques from actuator dynamics are explicitly added later.



**Table 12-4:** The three options for computing torques in *EcTotalTorqueTool*.

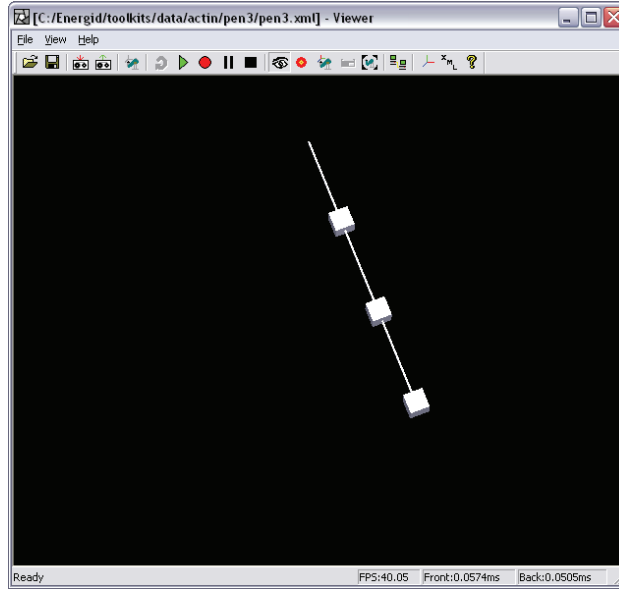
### 12.3.1.3 Example

Figure 12-14 and Figure 12-15 demonstrate the effect of dry friction. A dry friction torque in the form of equation (12-68) is assigned to each of the joint actuators. Each joint actuator also has a viscous friction associated with it. In Figure 12-14, the simulation starts with the initial configuration shown in the left subfigure. Due to gravity, the pendulum swings downwards and finally rests at the configuration shown in the right subfigure. Without dry friction, the pendulum would only asymptotically approach the vertical configuration.



**Figure 12-14:** An example used to demonstrate dry friction. The subfigure on the left shows the initial condition of the pendulum and the one on the right shows the resting configuration.

Figure 12-15 shows the same pendulum but with a different initial configuration. At this configuration, gravity is not sufficient to overcome the dry friction to cause the pendulum to move. Thus, the pendulum simply stays at the initial configuration.



**Figure 12-15:** Initial condition with less amount of gravitation torques. In this case, gravity cannot overcome the dry friction so the pendulum does not move.

### 12.3.2 Gear Backlash and Joint Elasticity

As mentioned earlier, the complexity that gear back and joint elasticity imposes on manipulator dynamics lies in the fact that they increase the number of states that need to be integrated. In addition, the full impact of gear backlash and joint elasticity on manipulator dynamics can be overly complex unless right assumptions are made to simplify it.

#### 12.3.2.1 Approach

In the Phase I software, the following Lagrange-Euler equation of motion of a fixed-base robotic manipulator with rigid joints, including joint inertias and friction, was utilized:

$$\boldsymbol{\tau} = [\mathbf{M}(\mathbf{q}) + \mathbf{R}^2 \mathbf{J}_m] \ddot{\mathbf{q}} + \mathbf{C}(\mathbf{q}) \dot{\mathbf{q}} + \mathbf{G}(\mathbf{q}) + \mathbf{F}(\mathbf{q}, \dot{\mathbf{q}}) + \mathbf{B} \quad (12-74)$$

where  $\boldsymbol{\tau}$  is the column vector of joint torques/forces;  $\mathbf{M}(\mathbf{q})$  is the manipulator inertia matrix;  $\mathbf{q}$  is the vector of joint position;  $\mathbf{C}(\mathbf{q})$  represents the Coriolis forces;  $\mathbf{G}(\mathbf{q})$  represents gravitational forces;  $\mathbf{J}_m$  and  $\mathbf{R}$  are diagonal matrices of motor inertias and gear ratios, respectively;  $\mathbf{F}(\mathbf{q}, \dot{\mathbf{q}})$  is the friction forces propagated from joint actuators; and  $\mathbf{B}$  represents the effect of external forces applied to the arm's links. The fact that the joint actuators are assumed rigid allows the actuator motion to be a simple function of the joint motion, which helps simplify the equations of motion.

$$\begin{aligned} \dot{\mathbf{q}}_m &= \mathbf{R} \dot{\mathbf{q}} \\ \ddot{\mathbf{q}}_m &= \mathbf{R} \ddot{\mathbf{q}} \end{aligned} \quad (12-75)$$

where  $\mathbf{q}_m$  is the vector of motor position.

When joints are allowed to be non-rigid, the relationships in equation (12-75) no longer hold true. Consequently, the equation of motion (12-74) is no longer valid. In this work, the following

assumptions used by Spong [29] in his development of a dynamic model for manipulator with elastic joints will be used.

1. The kinetic energy of the rotor is due mainly to its own rotation.
2. The rotor/gear inertia is symmetric about the rotor axis of rotation.

With these assumptions, the equations of motion for a manipulator with non-rigid joints can be written as

$$\begin{aligned} \mathbf{M}(\mathbf{q})\ddot{\mathbf{q}} + \mathbf{C}(\mathbf{q})\dot{\mathbf{q}} + \mathbf{G}(\mathbf{q}) + \mathbf{B} &= \mathbf{R}\boldsymbol{\tau} \\ \mathbf{J}_m\ddot{\mathbf{q}}_m + \mathbf{C}_m\dot{\mathbf{q}}_m + \boldsymbol{\tau}_f + \boldsymbol{\tau} &= \boldsymbol{\tau}_m \end{aligned} \quad (12-76)$$

where  $\boldsymbol{\tau}_m$  is the column vector of input torques from motors;  $\mathbf{C}_m$  is a diagonal matrix whose elements are viscous coefficients, respectively, of motor, shaft, and gear assemblies;  $\boldsymbol{\tau}_f$  is the actuator dry friction torques; and  $\boldsymbol{\tau}$  is a column vector of output torques from actuators due to non-rigidity of the joints. In general,  $\boldsymbol{\tau}$  can be any function of the joint position and velocity and motor position and velocity.

$$\boldsymbol{\tau} = \mathbf{f}(\mathbf{q}, \dot{\mathbf{q}}, \mathbf{q}_m, \dot{\mathbf{q}}_m). \quad (12-77)$$

For simple joint flexibility, the output torque can be computed from

$$\boldsymbol{\tau} = \mathbf{K}_f(\mathbf{R}^{-1}\mathbf{q}_m - \mathbf{q}) + \mathbf{C}_f(\mathbf{R}^{-1}\dot{\mathbf{q}}_m - \dot{\mathbf{q}}) \quad (12-78)$$

where  $\mathbf{K}_f$  and  $\mathbf{C}_f$  are diagonal matrices of joint flexibility stiffness constants and joint flexibility damper constants, respectively.

The gear backlash model can be added by constructing  $\boldsymbol{\tau}$  that reflects the lost-motion effect of gear backlash. In essence, neither torque nor motion is transmitted from motor to link in the “backlash mode.” The approach taken here is borrowed from a study by Yang and Fu [30]. Let’s consider only one joint actuator and one link. The joint position of the link is denoted by  $q$  and the motor position by  $q_m$ . The gear ratio of the joint is denoted by  $r$ . Without loss of generality, let’s consider a planar mating set of gears and assume that initially the two gears are in contact with each other. The driving gear is attached to the motor and the driven gear to the link. If either gear moves such that two contacting gear teeth start to break away, then the two gears become out of contact. In the case, the motion is said to be in the “backlash mode.” The condition under which the motion is in the backlash mode is identified by

$$rq - b < q_m < rq \quad (12-79)$$

where  $b$  is the backlash amount, which is defined in terms of the motor displacement. If the moving gear continues to its course, it will eventually reach the other gear and the contact is reestablished. The motion and torque is then transmitted between the motor and the link. There are two conditions under which the contact between the two gears is established. The condition in equation (12-80) is referred to as the “positive contact mode.”

$$q_m \geq rq \quad (12-80)$$

The “negative contact mode” is defined by the following condition.

$$q_m \leq rq - b \quad (12-81)$$

Combining the conditions in equations (12-79) – (12-81) with the torque resulted from joint flexibility in equation (12-78), the total output torque between the motor and the link can be written as

$$\tau = \begin{cases} k_f(r^{-1}q_m - q) + c_f(r^{-1}\dot{q}_m - \dot{q}), & \text{positive contact} \\ 0, & \text{backlash mode} \\ k_f(r^{-1}(q_m + b) - q) + c_f(r^{-1}\dot{q}_m - \dot{q}), & \text{negative contact} \end{cases} \quad (12-82)$$

### 12.3.2.2 Implementation

Like the dry friction model, the output torque from equation (12-82) is implemented in the *EcJointActuator* class. Important methods related to joint flexibility and gear backlash are listed in Table 12-5. The presence of the *isRigid* flag is critical to correct and efficient implementation and dynamic simulation. If this flag is set to true, then the effect of gear backlash and joint elasticity will essentially be ignored.

Method	Description
<i>isRigid</i> / <i>setIsRigid</i>	Gets/sets the flag indicating whether or not this joint is rigid. If the effect of gear backlash and/or joint elasticity is to be included, this flag must be false.
<i>jointFlexibilityStiffness</i> / <i>setJointFlexibilityStiffness</i>	Gets/sets the joint flexibility spring constant $k_f$
<i>jointFlexibilityDamping</i> / <i>setJointFlexibilityDamping</i>	Gets/sets the joint flexibility damping constant $c_f$
<i>backlash</i> / <i>setBacklash</i>	Gets/sets the backlash amount $b$

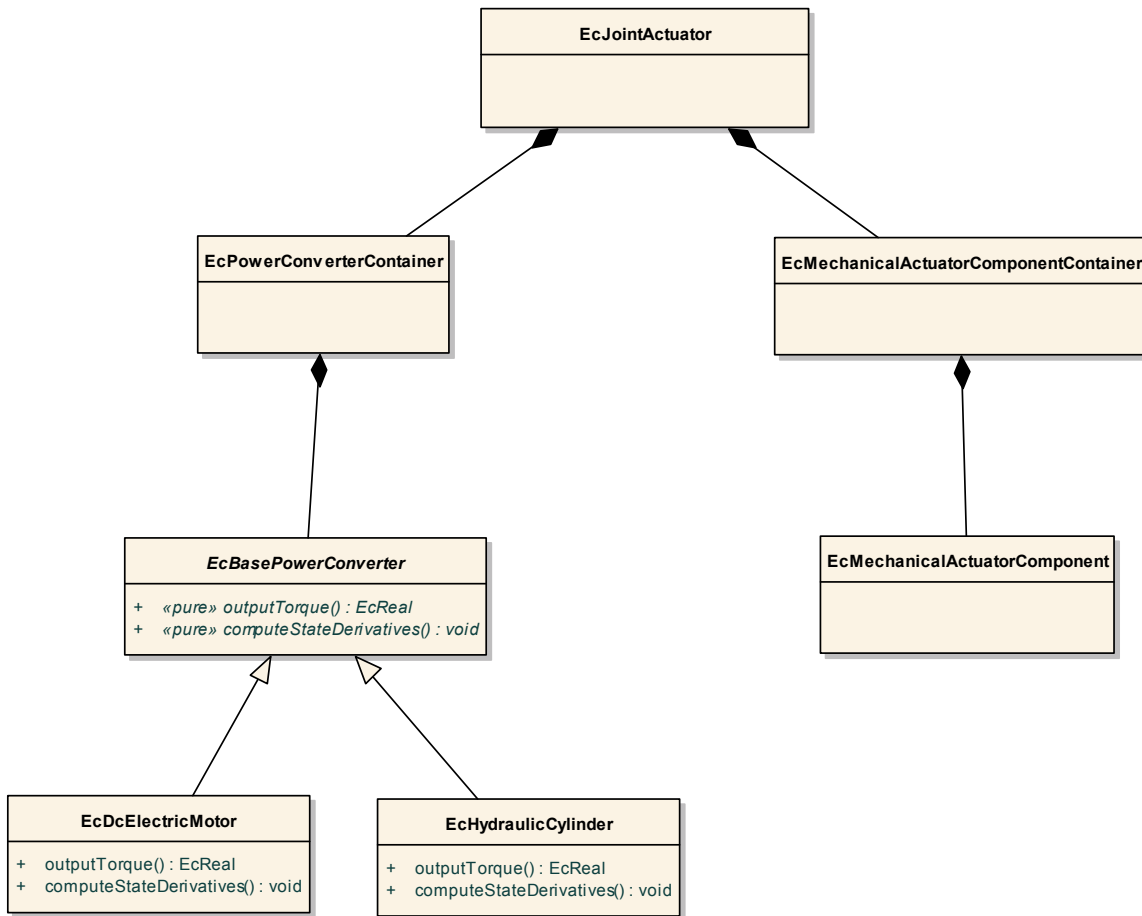
**Table 12-5:** Methods in *EcJointActuator* for implementation of dynamics of gear backlash and joint elasticity.

The implementation of dynamics from gear backlash and joint elasticity is significantly more complicated than that of dry friction because the increase in the number of states. The integration of motor motion is taken care of by a general-purpose integrator presented earlier. Three general-purpose integrators were added to *EcIndividualDynamicSimulator* to process integration of joint motion, base motion, and motor motion.

For motor motion integration, care must be taken to achieve computational efficiency for joints whose gear backlash and joint elasticity are not present. Dynamic simulation should not unnecessarily incur extra computation associated with integrating those extra states. The *m\_IsRigid* flag in *EcJointActuator* helps avoid the extra computation. If *m\_IsRigid* is set to true, the integration of the motor state of that particular joint can simply be skipped. With the fine control at the joint level instead of the manipulator level, this enables manipulators to have both rigid and non-rigid joints.

### 12.3.3 Power Conversion

Essentially, a joint actuator is broken into two parts – the power converter and the mechanical component. The mechanical component is responsible for the motion of the actuator and includes all mechanical properties including inertia, friction (Coulomb and viscous), stiffness, flexibility due to compliance, lost motion (backlash), etc. These properties had already been defined and successfully simulated. The power converter represents conversion from hydraulic or electric power to mechanical power in terms of force (torque) and motion.



**Figure 12-16:** Class diagram of joint actuator.

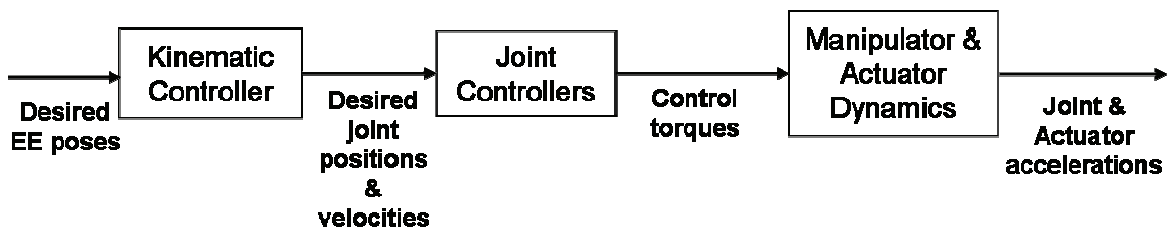
The class diagram for joint actuators is shown in Figure 12-16. In order to be able to easily swap components, we adopt the notion of container, which is used throughout the Actin toolkit, when designing the actuator classes. A joint actuator is represented by *EcJointActuator* class. *EcJointActuator* consists of *EcPowerConverterContainer* and *EcMechanicalActuatorComponentContainer*. *EcPowerConverterContainer* contains an instance of a class that is derived from *EcBasePowerConverter*, which is a base class for all power converters. Currently, two power converter classes have been implemented. The first is *EcDcElectricMotor*, for permanent-magnet DC motors, which are widely used as prime movers for actuators in the field of robotics. The second is *EcHydraulicCylinder* that represents the linear hydraulic cylinder commonly used in many powerful industrial robots. The use of container allows us the flexibility to use any type of power converter in the simulation. Other types of power converters can also be modeled and implemented by deriving from *EcBasePowerConverter* when the need arises.

The two methods of *EcBasePowerConverter* that need to be implemented in all of its derived classes along with their descriptions are listed in Table 12-6.

Method	Description
<i>outputTorque</i>	Computes and returns the converted torque from the control input and state.
<i>computeStateDerivatives</i>	Computes the derivatives of the state variables of the power converter.

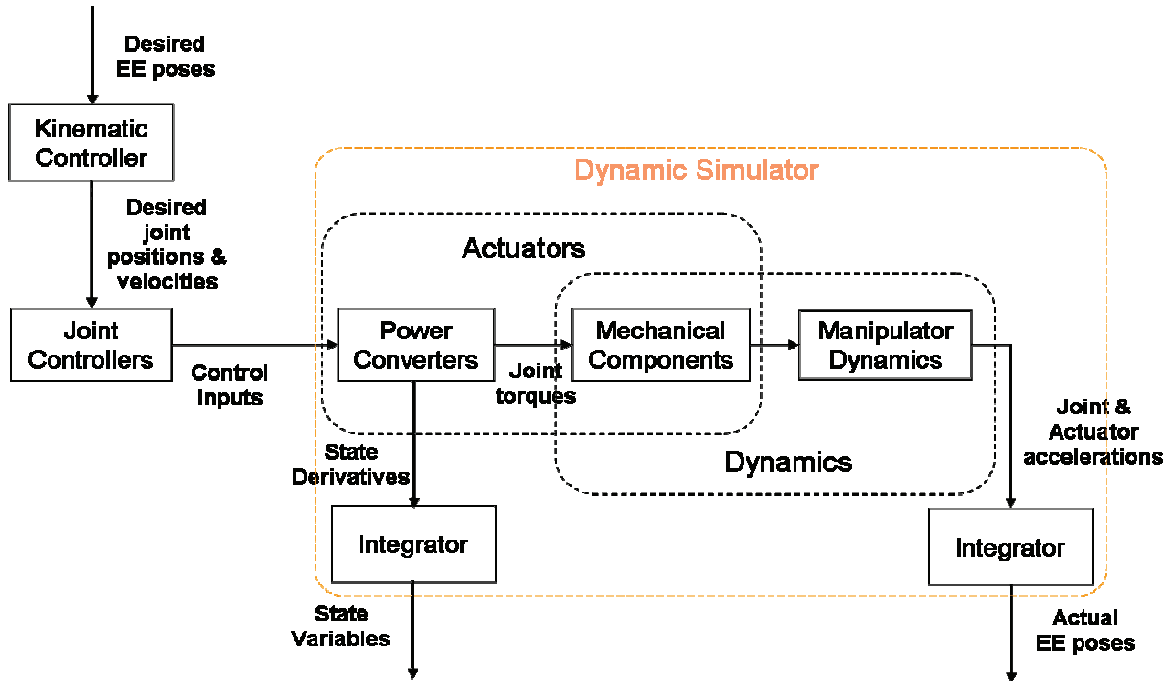
**Table 12-6:** Critical methods of *EcBasePowerConverter*.

With the inclusion of power converters, we must add support for simulating their dynamic behaviors in our original dynamic simulation and control framework. Figure 12-17 shows the rudimentary control of manipulators without power conversion. First, a kinematic controller computes the desired joint positions and velocities from the desired end-effector poses. Joint controllers then take the desired joint positions and velocities as inputs and compute the control torques. These torques are used in the dynamic simulation to calculate the joint accelerations, which are in turn integrated to obtain the actual joint positions and velocities.



**Figure 12-17:** Rudimentary dynamic simulation and control.

With the addition of the power converter part to the actuator model, the dynamic simulation and control framework needs to be modified. The joint controllers need to output the control inputs to the actuators instead of the control torques. The power converters then compute the output torques and feed them to the mechanical components and manipulator dynamics. In addition, the power converters need to compute the derivatives of their state variables. These derivatives will then be integrated as part of dynamic simulation. The schematic of this advanced dynamic simulation and control is given in Figure 12-18.



**Figure 12-18:** Advanced dynamic simulation and control to support power conversion.

Note that in Figure 12-18, the derivatives of the power converter state variables need to be integrated to complete the dynamic simulation. Currently, the Actin toolkit supports Adams-Bashforth-Moulton integration, which is well-suited for solving differential equations with second derivatives (e.g. accelerations). The Adams-Bashforth-Moulton method uses Adams-Bashforth predictor to integrate the second derivatives (accelerations) to obtain the first derivatives (velocities) and Adams-Moulton corrector to obtain the variables (positions). For integrating the derivatives of the power converter state variables, we will use the Adams-Bashforth predictor due to its simplicity and effectiveness in real-time simulation applications. The formulae for first-order to fourth-order Adams-Bashforth predictors are listed in Table 12-7. Here  $\mathbf{q}_k$  and  $\dot{\mathbf{q}}_k$  are a vector of state variables and a vector of the derivatives of the state variables at the time step  $k$ , respectively.

Order	Update Formula
1	$\mathbf{q}_{k+1} = \mathbf{q}_k + \Delta t \dot{\mathbf{q}}_k$
2	$\mathbf{q}_{k+1} = \mathbf{q}_k + \frac{\Delta t}{2} (3\dot{\mathbf{q}}_k - \dot{\mathbf{q}}_{k-1})$
3	$\mathbf{q}_{k+1} = \mathbf{q}_k + \frac{\Delta t}{12} (23\dot{\mathbf{q}}_k - 16\dot{\mathbf{q}}_{k-1} + 5\dot{\mathbf{q}}_{k-2})$
4	$\mathbf{q}_{k+1} = \mathbf{q}_k + \frac{\Delta t}{24} (55\dot{\mathbf{q}}_k - 59\dot{\mathbf{q}}_{k-1} + 37\dot{\mathbf{q}}_{k-2} - 9\dot{\mathbf{q}}_{k-3})$

**Table 12-7:** Adams-Bashforth Predictors.

## 12.4 Feedforward-Feedback Joint Controller

Actin provides a feedforward-feedback controller for use in dynamic simulation or application to hardware. This controller is described below.

### 12.4.1 Feedback Proportional-Plus-Derivative Feedback Controller

The first step in building the Feedforward-Feedback controller was to establish the desired accelerations of the joints. This was done using proportional-plus-derivative (PD) control, which takes the following form for each joint:

$$\ddot{q}_d = K_v(\dot{q}_d - \dot{q}) + K_p(q_d - q), \quad (12-83)$$

where  $q$  is the actual joint value and  $q_d$  is the desired joint value. In the Laplace domain, this gives the familiar second-order system dynamics:

$$T = \frac{1}{s^2 + K_v s + K_p}, \quad (12-84)$$

The second order dynamic system is well understood. A desired damping ratio  $\xi$  (typically a value near 1.0) and an undamped natural frequency  $\omega_n$  (typically a value higher than the natural frequencies of the uncontrolled system) can be established by setting

$$K_p = \omega_n^2 \quad (12-85)$$

and

$$K_v = 2\xi\omega_n. \quad (12-86)$$

The parameters  $\xi$  and  $\omega_n$  were made configurable components in the XML description of the joint controller. Their use in (12-83) gives for each joint the desired acceleration at runtime.

### 12.4.2 Feedforward Dynamics

Given a set of desired joint accelerations calculated using the equations above, the next question is how to produce those accelerations by applying motor torque at the joints. This requires an analysis of the robot dynamics. This analysis and the resulting formula for calculating joint torques are described below.

#### 12.4.2.1 Rigid-Body Dynamics

For any rigid body, let  $\vec{f}$  be the vector force applied to the link,  $\vec{n}$  be the moment,  $\vec{\omega}$  be the angular velocity,  $\vec{v}$  be the linear velocity,  $\vec{f}_e$  be an a priori external force applied to the body,  $\vec{n}_e$  be an a priori moment applied to the body,  $m$  be the mass,  $\mathbf{H}$  be the cross-product matrix for the first moment of inertia, and  $\mathbf{J}$  be the second moment of inertia. Then, the force/moment equations are given by the following:



$$\vec{f} = \mathbf{H}^T \dot{\vec{\omega}} + \vec{\omega} \times \mathbf{H}^T \vec{\omega} + m\dot{\vec{v}} - \vec{f}_e \quad (12-87)$$

and

$$\vec{n} = \mathbf{J} \dot{\vec{\omega}} + \vec{\omega} \times \mathbf{J} \vec{\omega} + \mathbf{H} \dot{\vec{v}} - \vec{n}_e. \quad (12-88)$$

Let the 6×6 rigid-body inertia be defined as follows:

$$\mathbf{I}^C = \begin{bmatrix} m\mathbf{I} & \mathbf{H}^T \\ \mathbf{H} & \mathbf{J} \end{bmatrix}. \quad (12-89)$$

And let a bias frame force be defined as

$$\mathbf{B} = \begin{bmatrix} \vec{\omega} \times \mathbf{H}^T \vec{\omega} \\ \vec{\omega} \times \mathbf{J} \vec{\omega} \end{bmatrix} + \begin{bmatrix} -\vec{f}_e \\ -\vec{n}_e \end{bmatrix}. \quad (12-90)$$

With this, the rigid-body dynamics in (5) and (6) can be represented as

$$\mathbf{F} = \mathbf{I}^C \mathbf{A} + \mathbf{B}, \quad (12-91)$$

Where  $\mathbf{F} = [\vec{f}^T \vec{n}^T]^T$  and  $\mathbf{A} = [\dot{\vec{v}}^T \dot{\vec{\omega}}^T]^T$  are 6×1 representations of spatial force and acceleration, respectively.

#### 12.4.2.2 Articulated Dynamics

The above discussion addressed a rigid body. For an articulated mechanism, the composite-rigid-body dynamics are represented using the following equation:

$$\boldsymbol{\tau} = \mathbf{M}(\mathbf{q})\ddot{\mathbf{q}} + \mathbf{C}(\mathbf{q})\dot{\mathbf{q}} + \mathbf{G}(\mathbf{q}) + \mathbf{D}(\mathbf{q})\mathbf{A}_b + \mathbf{B}, \quad (12-92)$$

where  $\boldsymbol{\tau}$  is the column vector of joint torques/forces,  $\mathbf{M}(\mathbf{q})$  is the manipulator inertia matrix,  $\mathbf{q}$  is the vector of joint positions,  $\mathbf{C}(\mathbf{q})$  represents the Coriolis forces,  $\mathbf{G}(\mathbf{q})$  represents gravitational forces,  $\mathbf{D}(\mathbf{q})$  is a function of configuration that linearly transforms base acceleration into joint torques, and  $\mathbf{B}$  represents the effect of external forces applied to the manipulator's links. Because gravitational  $\mathbf{G}(\mathbf{q})$  is explicit,  $\mathbf{A}_b$  is the *seen* (rather than *felt*) value.

The matrix  $\mathbf{D}(\mathbf{q})$  has an additional quality in that it can be used to calculate the force  $\mathbf{F}_{ma}$  applied to the base as a result of manipulator joint accelerations through

$$\mathbf{F}_{ma} = -\mathbf{D}^T \ddot{\mathbf{q}}. \quad (12-93)$$

Let the total force applied by the manipulator to the base be

$$\mathbf{F}_m = \mathbf{F}_{ma} + \mathbf{F}_{mc} + \mathbf{F}_{mg} + \mathbf{F}_{me}, \quad (12-94)$$

where  $\mathbf{F}_{mc}$ ,  $\mathbf{F}_{mg}$ , and  $\mathbf{F}_{me}$  represent the force due to Coriolis and centripetal terms, gravity, and preexisting external forces, respectively.

If  $\mathbf{F}_e$  represents the external forces applied to the base link directly for control, then (9) gives

$$\mathbf{F}_{ma} + \mathbf{F}_e = \mathbf{I}_b^C \mathbf{A}_b - \mathbf{F}_{mc} - \mathbf{F}_{mg} - \mathbf{F}_{me}, \quad (12-95)$$

where  $\mathbf{I}_b^C$  is the composite rigid-body inertia of the entire manipulator, including the base, treated as a rigid body.

Substituting in (12-93) gives

$$\mathbf{D}^T \ddot{\mathbf{q}} + \mathbf{I}_b^C \mathbf{A}_b = \mathbf{F}_e + \mathbf{F}_{mc} + \mathbf{F}_{mg} + \mathbf{F}_{me}. \quad (12-96)$$

Combining this with (10) gives the following as the manipulator dynamics equation

$$\begin{bmatrix} \mathbf{I}_b^C & \mathbf{D}(\mathbf{q})^T \\ \mathbf{D}(\mathbf{q}) & \mathbf{M}(\mathbf{q}) \end{bmatrix} \begin{bmatrix} \mathbf{A}_b \\ \ddot{\mathbf{q}} \end{bmatrix} = \begin{bmatrix} \mathbf{F}_e + \mathbf{F}_{mc} + \mathbf{F}_{mg} + \mathbf{F}_{me} \\ \boldsymbol{\tau} - \mathbf{C}(\mathbf{q})\dot{\mathbf{q}} - \mathbf{G}(\mathbf{q}) - \mathbf{B} \end{bmatrix}. \quad (12-97)$$

For  $N$  joint degrees of freedom,  $\mathbf{I}_b^C$  is  $6 \times 6$ ,  $\mathbf{D}(\mathbf{q})$  is  $N \times 6$ , and  $\mathbf{M}(\mathbf{q})$  is  $N \times N$ . This equation could be used to calculate forward dynamics by solving for  $\mathbf{A}_b$  and  $\ddot{\mathbf{q}}$  and integrating them.

### 12.4.2.3 The Mathematics of Calculating Feedforward Joint Torque

But the problem at hand is not forward dynamics. Rather, for calculating control torques on a mobile robot, based on (15), what is desired is to find a value for  $\boldsymbol{\tau}$  that gives a prescribed  $\ddot{\mathbf{q}}$  while forcing  $\mathbf{F}_e = 0$  and letting  $\mathbf{A}_b$  be arbitrary. That is, we have control over the joint torques  $\boldsymbol{\tau}$ , but no ability to apply force directly to any link. To make a joint controller, we want to specify  $\ddot{\mathbf{q}}$ , but don't care (at least directly) about  $\mathbf{A}_b$ .

To find these joint torques,  $\mathbf{F}_e$  can be set to 0 and  $\mathbf{A}_b$  can be solved from the top six rows of (12-97) as follows:

$$\mathbf{A}_b = [\mathbf{I}_b^C]^{-1} (\mathbf{F}_{mc} + \mathbf{F}_{mg} + \mathbf{F}_{me} - \mathbf{D}(\mathbf{q})^T \ddot{\mathbf{q}}_d), \quad (12-98)$$

where  $\ddot{\mathbf{q}}_d$  is the desired set of joint accelerations.  $\mathbf{I}_b^C$  is guaranteed to be positive definite, and therefore invertible, for any physically realizable system. Otherwise, as a direct consequence of (12-97), it would be possible to have acceleration of the robot without force, in violation of Newton's Second Law.

With  $\mathbf{A}_b$  established, the joint torques can then be solved from the bottom  $N$  rows of (12-97) as follows:

$$\boldsymbol{\tau} = \mathbf{D}(\mathbf{q})\mathbf{A}_b + \mathbf{M}(\mathbf{q})\ddot{\mathbf{q}}_d + \mathbf{C}(\mathbf{q})\dot{\mathbf{q}} + \mathbf{G}(\mathbf{q}) + \mathbf{B}. \quad (12-99)$$

With this, the feedforward control torques are established.

#### 12.4.2.4 Feedforward Implementation

Energid implemented a method, though the class *EcTotalTorqueTool*, for calculating the inverse dynamics. That is, given a set of desired motion  $\{\mathbf{q}, \dot{\mathbf{q}}, \ddot{\mathbf{q}}, \mathbf{A}_b\}$  and pre-existing external forces  $\{\mathbf{F}_{me}\}$ , it calculates a set of control torques/forces  $\{\mathbf{F}_e, \boldsymbol{\tau}\}$  to achieve the desired motion.

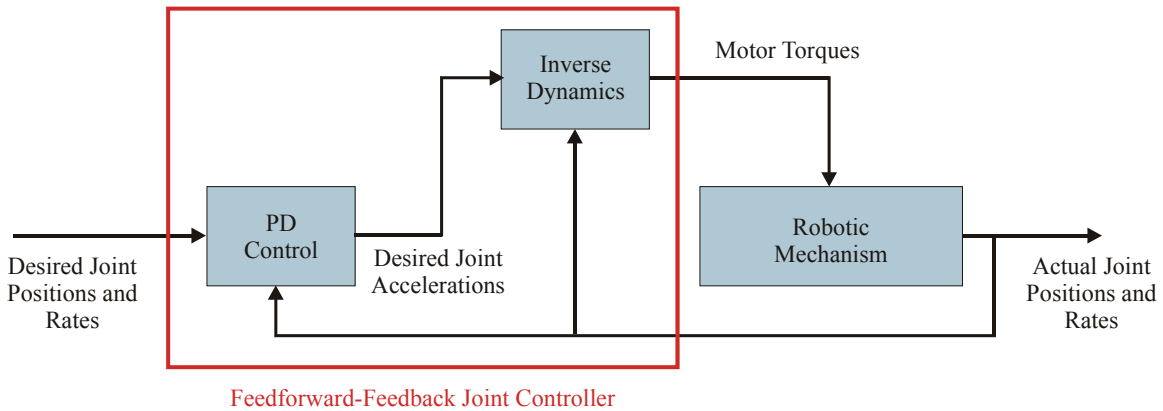
The torques as given in (12-99) are calculated through two calls to this method. The first call calculates the forces  $\{\mathbf{F}_{e,0}, \boldsymbol{\tau}_0\}$  that achieve the motion  $\{\mathbf{q}, \dot{\mathbf{q}}, \ddot{\mathbf{q}} = \ddot{\mathbf{q}}_d, \mathbf{A}_b = \mathbf{0}\}$ . This is used to establish the true base acceleration through the following equation:

$$\mathbf{A}_b = -[\mathbf{I}_b^C]^{-1} \mathbf{F}_{e,0}, \quad (12-100)$$

where the matrix inverse is calculated using a special form of Cholesky Decomposition tailored to the special structure of the  $6 \times 6$  rigid-body inertia matrix. With this, the desired torques,  $\boldsymbol{\tau}$ , can be calculated using (12-92).

#### 12.4.3 The Complete Implementation

Putting together the calculations of the previous section gives the feedforward-feedback controller design shown in the figure below.

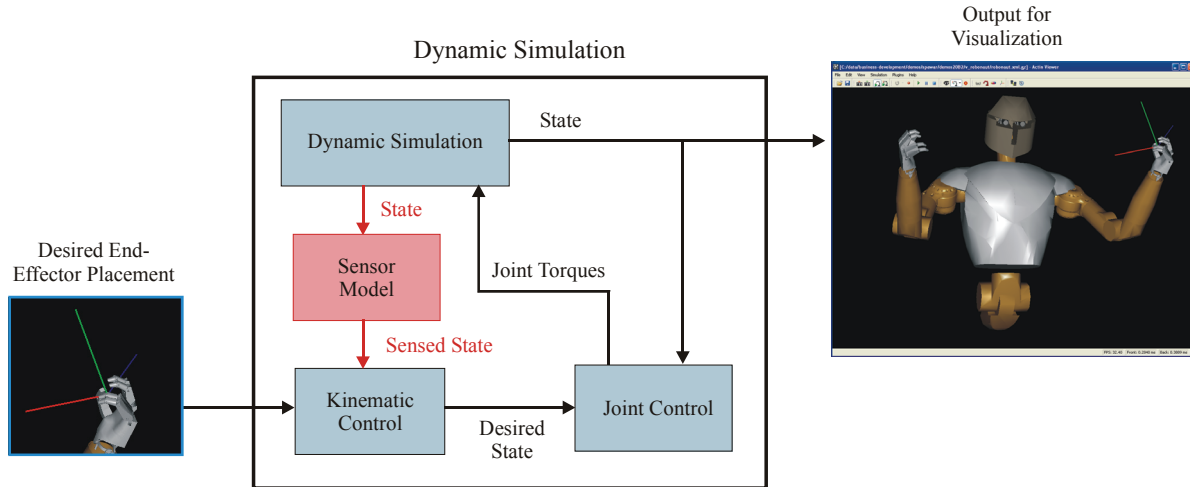


**Figure 12-19:** The organization of the feedforward-feedback joint controller. PD control is used to calculate desired joint accelerations. These are provided to an inverse dynamics algorithm (implemented to support for both free- and fixed-base mechanisms), which calculates motor torques. These motor torques are applied to the robotic mechanism to produce the actual joint positions and rates.

#### 12.4.4 Integration of Dynamic Simulation with the Kinematic Control System

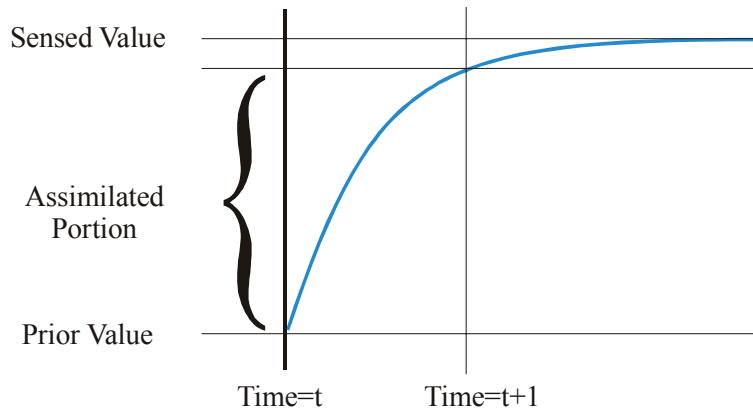
Previously, Energid's simulation software had not used feedback from the dynamic simulation to the kinematic controller. The assumption had been that the joint controllers would always successfully effect the desired state produced by the kinematic controller. This is a very good assumption for fixed-base mechanisms, but it is not always true for mobile mechanisms, where a discrepancy can easily form between the dynamic simulation and the idealized controller (due to a grasp or foot position slipping, for example).

To address this, in the last quarter Energid implemented a feedback process between the dynamic simulation and the kinematic controller. This is illustrated in the figure below, which shows (in red) the newly implemented connection between the dynamic simulation and the kinematic controller. This connection models a sensed value of the state that includes the position and orientation of the base link. This sensed value is used by the kinematic controller to change its own internal state to reflect the reality provided by the sensors. The implementation of the sensor is just a state capture currently—more detailed models can be added by toolkit users.



**Figure 12-20:** The new connection between the dynamic simulation and the kinematic controller is shown (in red). The dynamic simulation state is used to model a sensed state, which is input to the kinematic control module for use in correcting its own internal model.

The feedback from the sensor is incorporated into the kinematic controller through an exponentially convergent process. This non-instantaneous assimilation reduces dynamic coupling between the kinematic controller and the dynamic simulation. To implement it, an assimilation factor was added to the position controller’s XML description. This assimilation factor,  $f$ , is the proportion of feedback that is assimilated per second, as illustrated in the figure below.



**Figure 12-21:** To reduce dynamic coupling between the dynamic simulation and the kinematic controller, feedback information from the dynamic simulation is assimilated over time. The rate of assimilation is prescribed using a parameter that specifies the proportion of the difference between the prior known value and the sensed value that is assimilated in one second. The process is shown here for a scalar value, as would be used for base position. Base orientation is assimilated in a similar manner, but by interpolating two quaternions over the surface of a unit hypersphere.

The assimilation process is implemented as follows. Given a time difference of  $\Delta t \neq 0$  from the last assimilation step, an assimilation proportion  $\alpha$  for that specific time difference is calculated from the one-second assimilation proportion  $f$  as

$$\alpha = 1 - (1 - f)^{\Delta t}, \quad (12-101)$$

where  $0 \leq f \leq 1$ .

This is used to calculate a new value for state variable  $x$  as follows:

$$x_n = \alpha x_s + (1 - \alpha)x_o, \quad (12-102)$$

where  $x_n$  is the new value,  $x_o$  is the old value, and  $x_s$  is the sensed value. For orientation, which cannot be decomposed into a set of independent scalars, the same process is used, but the interpolation is performed with quaternions over the surface of a unit hypersphere.

## 12.5 Numerical Integration

### 12.5.1 Implementation

The embedded numerical integration resides inside the *EcIntegratorContainer* class, which contains an instance of a class derived from *EcBaseIntegrator*. Some of the methods of *EcBaseIntegrator* are listed in the table below. The *integrateOneStep()* method is pure virtual and must be implemented by the derived class. The implementation of Adams-Bashforth/Adams-Moulton integration now resides inside *EcAdamsBashforthMoulton* class. Other numerical integration techniques such as Runge-Kutta can be implemented by subclassing *EcBaseIntegrator*.

Method	Description
<i>integrateOneStep()</i>	Performs the integration one time step forward
<i>integrationOrder()</i>	Returns the integration order
<i>setIntegrationOrder()</i>	Sets the integration order
<i>timeStep()</i>	Returns the integration time step
<i>setTimeStep()</i>	Sets the integration time step

**Table 12-8:** Methods for *EcBaseIntegrator*.

As mentioned earlier, the *EcAdamsBashforthMoulton* implements the Adams-Bashforth predictor and the Adams-Moulton corrector inside its *integrateOneStep()* method. The Adams-Bashforth predictor is used to integrate the second derivatives (accelerations) to get the first derivatives (velocities). The Adams-Bashforth algorithms rely on current and past state and derivative values to calculate the future state. For example, first-order Adams-Bashforth is just basic forward Euler. The table below gives the Adams-Bashforth update formulas.

Because past derivatives are needed, an Adams-Bashforth integrator using an order higher than one must begin with a different method. Lower orders are used to build up to higher orders. That is, the

integrator starts with the first-order method, then on the next step uses the second-order method, and so on, until the desired order is reached.

Order	Update Formula
1	$\dot{\mathbf{q}}_{k+1} = \dot{\mathbf{q}}_k + \Delta t \ddot{\mathbf{q}}_k$
2	$\dot{\mathbf{q}}_{k+1} = \dot{\mathbf{q}}_k + \frac{\Delta t}{2} (3\ddot{\mathbf{q}}_k - \ddot{\mathbf{q}}_{k-1})$
3	$\dot{\mathbf{q}}_{k+1} = \dot{\mathbf{q}}_k + \frac{\Delta t}{12} (23\ddot{\mathbf{q}}_k - 16\ddot{\mathbf{q}}_{k-1} + 5\ddot{\mathbf{q}}_{k-2})$
4	$\dot{\mathbf{q}}_{k+1} = \dot{\mathbf{q}}_k + \frac{\Delta t}{24} (55\ddot{\mathbf{q}}_k - 59\ddot{\mathbf{q}}_{k-1} + 37\ddot{\mathbf{q}}_{k-2} - 9\ddot{\mathbf{q}}_{k-3})$

**Table 12-9:** Adams-Bashforth Predictors. These are used to integrate the joint accelerations to find the joint rates. The joint rates at the next time point (time point k+1) are functions of current and past values of joint rates and accelerations. The integrator using these formulas was separated into independent code.

Adams-Moulton correctors are used to integrate the first derivatives (velocities) to get the positions. Correctors require future (as well as both past and current) derivatives, which is why they can be applied to integrate the first derivatives but not the second derivatives. The Adams-Moulton correctors are shown in the table below. In all cases the same order is used for the predictor as for the corrector.

Order	Update Formula
1	$\mathbf{q}_{k+1} = \mathbf{q}_k + \Delta t \dot{\mathbf{q}}_{k+1}$
2	$\mathbf{q}_{k+1} = \mathbf{q}_k + \frac{\Delta t}{2} (\dot{\mathbf{q}}_{k+1} + \dot{\mathbf{q}}_k)$
3	$\mathbf{q}_{k+1} = \mathbf{q}_k + \frac{\Delta t}{12} (5\dot{\mathbf{q}}_{k+1} + 8\dot{\mathbf{q}}_k - \dot{\mathbf{q}}_{k-1})$
4	$\mathbf{q}_{k+1} = \mathbf{q}_k + \frac{\Delta t}{24} (9\dot{\mathbf{q}}_{k+1} + 19\dot{\mathbf{q}}_k - 5\dot{\mathbf{q}}_{k-1} + \dot{\mathbf{q}}_{k-2})$

**Table 12-10:** Adams-Moulton Correctors. These are used to integrate the joint rates to get the joint positions. Note the Adams-Moulton correctors require the derivative value at the new time step, where the Adams-Bashforth predictors only require them at previous time steps.

The general-purpose integrator is then used in the dynamic simulation of manipulators' joint states and actuators' motor states. Its use on the general motion (base motion) is more complicated and is presented in the next section.

## 12.5.2 Integration of Base Motion

Separating numerical integration of joint motion is straightforward. Separation for base motion integration, on the other hand, was more difficult because it involves integrating angular acceleration and velocity to obtain orientation, which is normally represented by a quaternion. Typically, the orientation of the base motion is integrated using the following steps.

1. The angular acceleration (a  $3 \times 1$  vector) is integrated to obtain the angular velocity (also a  $3 \times 1$  vector) using the Adams-Bashforth predictor.
2. An “applicable” angular velocity  $s$  then calculated from the angular velocity from Step (1) using a modified version of the Adams-Moulton corrector.
3. The applicable angular velocity is then used to compute the orientation (a  $4 \times 1$  quaternion) using the `EcOrientation::integrateReferenceFrameAngularVelocity()` method.

This integration approach, however, does not lend itself to direct use the new `EcBaseIntegrator::integrateOneStep()` method described in previous section. The reason is that the `integrateOneStep()` method expects the accelerations, velocities, and positions to be of the same size. However, the orientation (represented by a quaternion) is not of the same size as the angular velocity and acceleration.

### 12.5.2.1 Approach

There are multiple formalisms for quaternions, and we use the one described by Shoemake [4c], which is more common for robotic applications. In this formalism, a quaternion representing frame  $j$  in frame  $i$  is given by

$${}^i Q_j = \begin{bmatrix} Q_0 \\ Q_1 \\ Q_2 \\ Q_3 \end{bmatrix} \quad (12-103)$$

The quaternion values are such that the quaternion can be converted to a rotation matrix through the following formula:

$${}^i R_j = \begin{bmatrix} 1 - 2Q_2^2 - 2Q_3^2 & 2Q_1Q_2 - 2Q_0Q_3 & 2Q_1Q_3 + 2Q_0Q_2 \\ 2Q_1Q_2 + 2Q_0Q_3 & 1 - 2Q_1^2 - 2Q_3^2 & 2Q_2Q_3 - 2Q_0Q_1 \\ 2Q_1Q_3 - 2Q_0Q_2 & 2Q_2Q_3 + 2Q_0Q_1 & 1 - 2Q_1^2 - 2Q_2^2 \end{bmatrix} \quad (12-104)$$

Note that, with this formalism,  $q = \{1, 0, 0, 0\}$  corresponds to the identity matrix. Angular velocity will be represented using the traditional three-element vector that is aligned with the instantaneous axis of rotation with magnitude equal to the angular rate of change. Angular velocity is given by  $\vec{\omega}$ . The time derivative of a quaternion can be calculated from the angular velocity using the following formula:

$$\begin{bmatrix} \dot{Q}_0 \\ \dot{Q}_1 \\ \dot{Q}_2 \\ \dot{Q}_3 \end{bmatrix} = \frac{1}{2} \begin{bmatrix} -Q_1 & -Q_2 & -Q_3 \\ Q_0 & -Q_3 & Q_2 \\ Q_3 & Q_0 & -Q_1 \\ -Q_2 & Q_1 & Q_0 \end{bmatrix} \begin{bmatrix} \omega_0 \\ \omega_1 \\ \omega_2 \end{bmatrix}. \quad (12-105)$$

The angular-velocity vector used in (12-105) is represented in the local coordinates of the moving frame.

Taking the derivative of (12-105) gives the following formula for quaternion acceleration:

$$\begin{bmatrix} \ddot{Q}_0 \\ \ddot{Q}_1 \\ \ddot{Q}_2 \\ \ddot{Q}_3 \end{bmatrix} = \frac{1}{2} \begin{bmatrix} -Q_1 & -Q_2 & -Q_3 \\ Q_0 & -Q_3 & Q_2 \\ Q_3 & Q_0 & -Q_1 \\ -Q_2 & Q_1 & Q_0 \end{bmatrix} \begin{bmatrix} \dot{\omega}_0 \\ \dot{\omega}_1 \\ \dot{\omega}_2 \end{bmatrix} + \frac{1}{2} \begin{bmatrix} -\dot{Q}_1 & -\dot{Q}_2 & -\dot{Q}_3 \\ \dot{Q}_0 & -\dot{Q}_3 & \dot{Q}_2 \\ \dot{Q}_3 & \dot{Q}_0 & -\dot{Q}_1 \\ -\dot{Q}_2 & \dot{Q}_1 & \dot{Q}_0 \end{bmatrix} \begin{bmatrix} \omega_0 \\ \omega_1 \\ \omega_2 \end{bmatrix} \quad (12-106)$$

Substituting  $\dot{Q}_i$  from (12-105) into (12-106) yields

$$\begin{bmatrix} \ddot{Q}_0 \\ \ddot{Q}_1 \\ \ddot{Q}_2 \\ \ddot{Q}_3 \end{bmatrix} = \frac{1}{2} \begin{bmatrix} -Q_1 & -Q_2 & -Q_3 \\ Q_0 & -Q_3 & Q_2 \\ Q_3 & Q_0 & -Q_1 \\ -Q_2 & Q_1 & Q_0 \end{bmatrix} \begin{bmatrix} \dot{\omega}_0 \\ \dot{\omega}_1 \\ \dot{\omega}_2 \end{bmatrix} - \frac{1}{4} (\omega_0^2 + \omega_1^2 + \omega_2^2) \begin{bmatrix} Q_0 \\ Q_1 \\ Q_2 \\ Q_3 \end{bmatrix} \quad (12-107)$$

The quaternion acceleration is contributed by two components—one from the angular acceleration and another from the angular velocity. Thus, even if the angular acceleration is identically zero, the quaternion rate will not be zero if the angular velocity is present. The quaternion acceleration due to the angular velocity can be viewed as similar to the centripetal acceleration of a particle moving in a circle with a constant angular velocity. In this case, a quaternion is a point on a four-dimensional unit hypersphere.

If the angular velocity and acceleration are expressed in the *stationary* coordinate system, then the time derivative of a quaternion can be calculated from the angular velocity using the following formula:

$$\begin{bmatrix} \dot{Q}_0 \\ \dot{Q}_1 \\ \dot{Q}_2 \\ \dot{Q}_3 \end{bmatrix} = \frac{1}{2} \begin{bmatrix} -Q_1 & -Q_2 & -Q_3 \\ Q_0 & Q_3 & -Q_2 \\ -Q_3 & Q_0 & Q_1 \\ Q_2 & -Q_1 & Q_0 \end{bmatrix} \begin{bmatrix} \omega_0 \\ \omega_1 \\ \omega_2 \end{bmatrix} \quad (12-108)$$

Taking the derivative of (12-108) gives the following formula for quaternion acceleration:



$$\begin{bmatrix} \ddot{Q}_0 \\ \ddot{Q}_1 \\ \ddot{Q}_2 \\ \ddot{Q}_3 \end{bmatrix} = \frac{1}{2} \begin{bmatrix} -Q_1 & -Q_2 & -Q_3 \\ Q_0 & Q_3 & -Q_2 \\ -Q_3 & Q_0 & Q_1 \\ Q_2 & -Q_1 & Q_0 \end{bmatrix} \begin{bmatrix} \dot{\omega}_0 \\ \dot{\omega}_1 \\ \dot{\omega}_2 \end{bmatrix} + \frac{1}{2} \begin{bmatrix} -\dot{Q}_1 & -\dot{Q}_2 & -\dot{Q}_3 \\ \dot{Q}_0 & \dot{Q}_3 & -\dot{Q}_2 \\ -\dot{Q}_3 & \dot{Q}_0 & \dot{Q}_1 \\ \dot{Q}_2 & -\dot{Q}_1 & \dot{Q}_0 \end{bmatrix} \begin{bmatrix} \omega_0 \\ \omega_1 \\ \omega_2 \end{bmatrix} \quad (12-109)$$

Substituting  $\dot{Q}_i$  from (12-108) into (12-109) yields the following formula for stationary-coordinate-system representation:

$$\begin{bmatrix} \ddot{Q}_0 \\ \ddot{Q}_1 \\ \ddot{Q}_2 \\ \ddot{Q}_3 \end{bmatrix} = \frac{1}{2} \begin{bmatrix} -Q_1 & -Q_2 & -Q_3 \\ Q_0 & Q_3 & -Q_2 \\ -Q_3 & Q_0 & Q_1 \\ Q_2 & -Q_1 & Q_0 \end{bmatrix} \begin{bmatrix} \dot{\omega}_0 \\ \dot{\omega}_1 \\ \dot{\omega}_2 \end{bmatrix} - \frac{1}{4} (\omega_0^2 + \omega_1^2 + \omega_2^2) \begin{bmatrix} Q_0 \\ Q_1 \\ Q_2 \\ Q_3 \end{bmatrix} \quad (12-110)$$

### 12.5.2.2 Implementation

The quaternion rate equations (12-105) and (12-108) are implemented in the *EcOrientation*. The following two methods implement the quaternion acceleration equations (12-107) and (12-110).

Method	Description
quaternionAccelerationFromLocalFrameMotion()	Compute the quaternion acceleration using equation (12-107).
quaternionAccelerationFromReferenceFrameMotion()	Compute the quaternion acceleration using equation (12-110).

**Table 12-11:** Two new methods in *EcOrientation* for computing quaternion acceleration.

Once quaternion rates and accelerations are available, they can be used in the general-purpose integrator to dynamically simulate the base motion of manipulators.

## 13 Collision Avoidance and Reasoning

The Actin™ toolkit provides fast, robust algorithms for collision avoidance and reasoning based on material type. The toolkit supports avoidance of manipulator self-collisions, manipulator-manipulator collisions, and manipulator environment collisions. This chapter begins by describing the algorithms and data structures used for fast collision avoidance and reasoning. Code samples describing how to setup a collision avoidance system can be found throughout this chapter.

### 13.1 Collision Avoidance Algorithm

The collision avoidance algorithm employs a gradient-based method which seeks to minimize the function with the general form as follows:

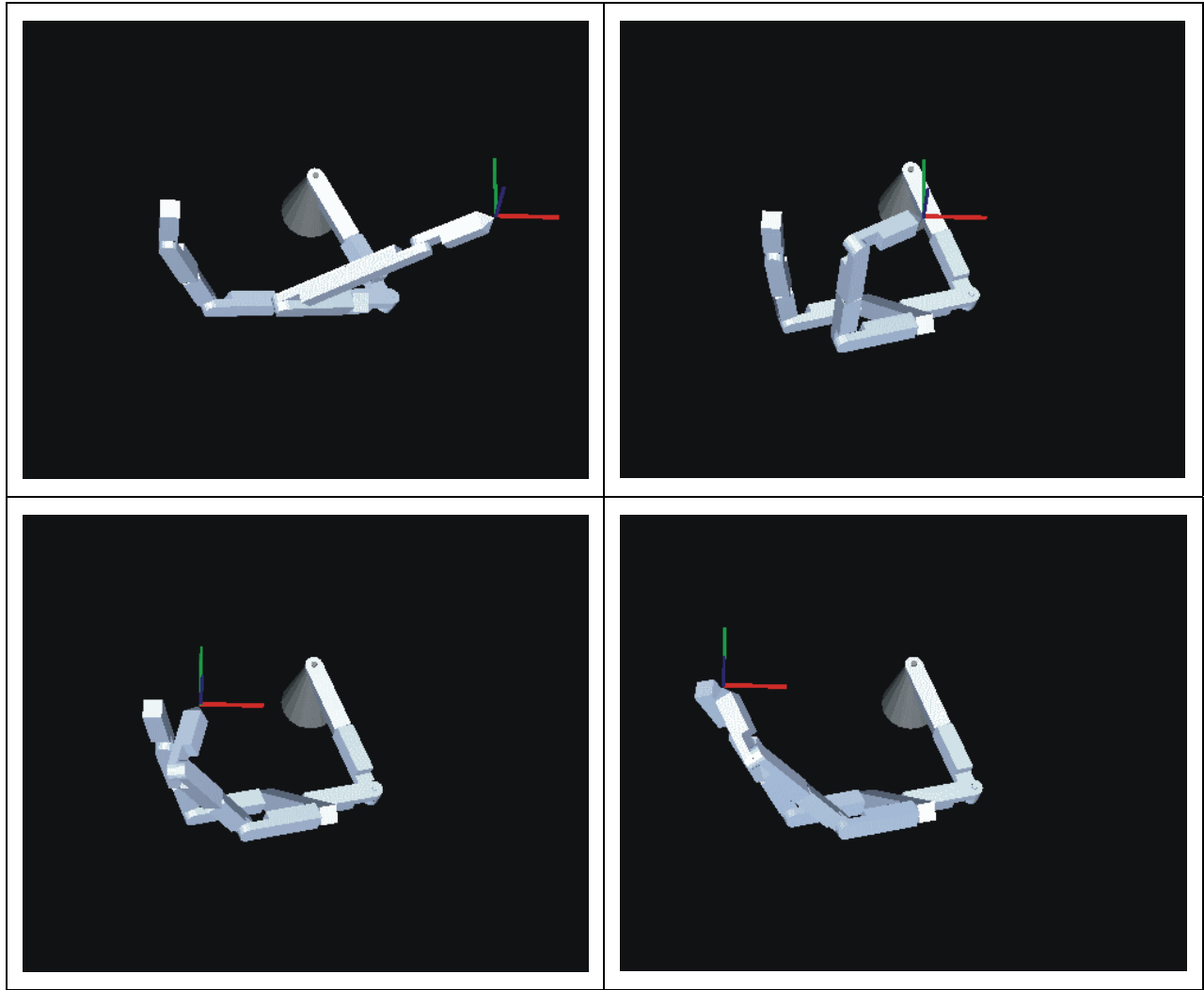
$$f(\mathbf{q}) = \sum_{i=1}^N \sum_{j=1}^B F^p(i, j), \quad (13-1)$$

where  $N$  is the number of links in the manipulator,  $B$  is the number of obstacles in the environment,  $p$  is a user-defined exponent, and  $F(i, j)$  is a measure of the proximity of the bounding volume of  $i$  to the bounding volume of link  $j$ .  $F(i, j)$  is zero when the distance is larger than a user-specified threshold, and within the threshold, it is just a scalar constant times the minimum distance needed to move one bounding volume to take it outside the threshold distance from the other bounding volume.

This function is used with finite differencing to calculate its gradient, which is then used as the vector parameter to the core control system to drive the manipulators in a direction that avoids collisions. Additional factors are added to this function (for instance a weighting based on material type) that will be discussed in the sections below.

### 13.2 Manipulator Self-Collision Avoidance

Manipulator self-collision occurs when one or more links of a manipulator collide during a commanded operation. For manipulator control, avoiding such types of collisions is imperative for fielded systems. Self-collisions can cause damage to the links or damage to the environment in the event that the robot loses balance and falls out of its task space. The figure below describes a self-collision avoidance example. Note how the left terminal link moves out of the way of the end effector as it approaches.



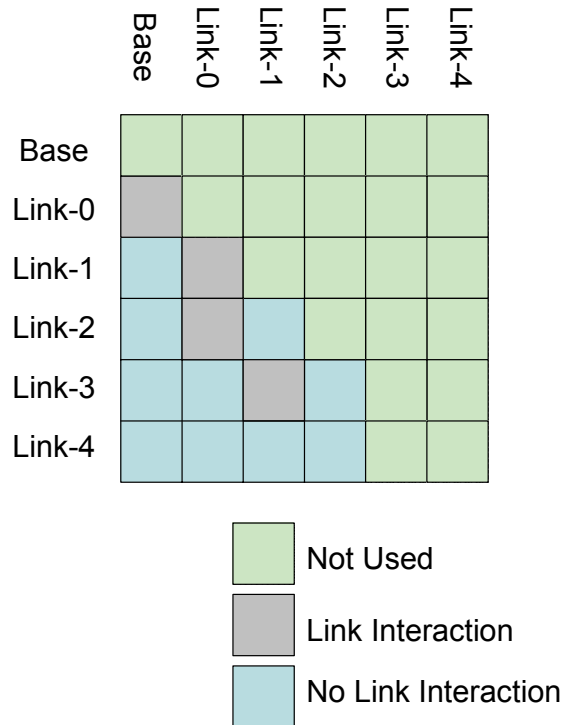
**Figure 13-1:** A sequence showing self-collision avoidance (left-to right starting in upper left).

The algorithm for manipulator self-collision is as follows:

$$f(\mathbf{q}) = \sum_{i=1}^N \sum_{j=i}^N \delta(i, j) \eta(i, j) F^p(i, j), \quad (13-2)$$

Where  $N$  is the number of links in the manipulator,  $p$  is a user-defined exponent and  $F(i, j)$  is a measure of the proximity of link  $i$  to link  $j$ . There are two additional terms that were not in the original obstacle avoidance equation. They are  $\delta(i, j)$ , which is a function describing the link interaction between link  $i$  and link  $j$  and  $\eta(i, j)$  which is a weighting function that is related to the material types of the two links.

In code,  $\delta(i, j)$  is described as the manipulator self-collision link map (*EcManipulatorSelfCollisionLinkMap*) which is derived from an *EcXmlStringStringBooleanMap* class, and is user-defined. It describes whether two links can interact. If there is no possibility that the two links can collide, then the map will return an *EcFalse* for that link pair, otherwise it returns *EcTrue*. Figure 13-2 shows an example self-collision link map for a five link manipulator. In reality, the implementation does not include entries for the symmetric query.



**Figure 13-2:** An example manipulator self-collision map for a 5 link manipulator.

### 13.2.1 Example: Creating a Self-Collision Link Map

The code snippet below describes how one would go about creating the manipulator self-collision link map. This specific example shows part of the creation of the link map for the 12 link example manipulator. Note that when the map is created (with the call to *buildMapFromSystem*), all values are set to *EcFalse* by default. Only those entries that need to be set to true need to be set explicitly.

```

// create the link map
EcManipulatorSelfCollisionLinkMap map;

// size the map to the size of the manipulator
map.buildMapFromSystem(m_TestManipulator);

// have the collision avoidance system avoid the end-effectors
map.setLinkCollisionCanditaterByIdentifier("link-0", "link-2", EcTrue);
map.setLinkCollisionCanditaterByIdentifier("link-0", "link-3", EcTrue);
map.setLinkCollisionCanditaterByIdentifier("link-0", "link-4", EcTrue);
map.setLinkCollisionCanditaterByIdentifier("link-0", "link-5", EcTrue);
map.setLinkCollisionCanditaterByIdentifier("link-0", "link-6", EcTrue);
map.setLinkCollisionCanditaterByIdentifier("link-0", "link-7", EcTrue);
map.setLinkCollisionCanditaterByIdentifier("link-0", "link-8", EcTrue);
map.setLinkCollisionCanditaterByIdentifier("link-0", "link-9", EcTrue);
map.setLinkCollisionCanditaterByIdentifier("link-0", "link-10", EcTrue);
map.setLinkCollisionCanditaterByIdentifier("link-0", "link-11", EcTrue);
map.setLinkCollisionCanditaterByIdentifier("link-1", "manipulator", EcTrue);
map.setLinkCollisionCanditaterByIdentifier("link-1", "link-3", EcTrue);
map.setLinkCollisionCanditaterByIdentifier("link-1", "link-4", EcTrue);
map.setLinkCollisionCanditaterByIdentifier("link-1", "link-5", EcTrue);

.
.
.

map.setLinkCollisionCanditaterByIdentifier("link-11", "manipulator", EcTrue);
map.setLinkCollisionCanditaterByIdentifier("link-11", "link-4", EcTrue);
map.setLinkCollisionCanditaterByIdentifier("link-11", "link-5", EcTrue);
map.setLinkCollisionCanditaterByIdentifier("link-11", "link-6", EcTrue);
map.setLinkCollisionCanditaterByIdentifier("link-11", "link-7", EcTrue);
map.setLinkCollisionCanditaterByIdentifier("link-11", "link-8", EcTrue);
map.setLinkCollisionCanditaterByIdentifier("link-11", "link-9", EcTrue);
map.setLinkCollisionCanditaterByIdentifier("link-11", "link-2", EcTrue);
map.setLinkCollisionCanditaterByIdentifier("link-11", "link-1", EcTrue);
map.setLinkCollisionCanditaterByIdentifier("link-11", "link-0", EcTrue);

// set the link map for this manipulator
m_TestManipulator.setSelfCollisionLinkMap(map);

// map the collision candidates
m_TestManipulator.mapLinkCollisionCandidates();

```

**Text Box 13-1:** Creating a manipulator self-collision link map.

The weighting function  $\eta(i, j)$  is described as:

$$\eta(i, j) = \mu_p m(i, j) \quad (13-3)$$

Where  $m(i, j)$  is a function that varies from [0..1] as a function of the material type and  $\mu_p$  is a function of the certainty that the described material type is, in fact, the material type. For manipulator self-collision  $\mu_p$  is 1 since it is assumed the manipulator has complete knowledge of the

material type of its links. This may not be the case for manipulator environment interactions however.

The manipulator self-collision algorithm is described below.

*checkSelfCollision*(manip M)

1. **for** each link  $u \in M$
2.     **for** each link  $v \in M$
2.     if ( $M - > \delta(u, v)$ )
3.          $\rho = dist(u, v)$
4.          $f_+ = \eta(u, v) \left[ \frac{\rho_{avoid} - \rho}{\rho_{avoid}} \right]^n B$

**Listing 1** Manipulator Self-collision algorithm

### 13.3 Manipulator-Manipulator Collision Avoidance

Manipulator-manipulator collisions occur between robots sharing a task space or working cooperatively to achieve some goal. The computational complexity of determining and preventing a manipulator-manipulator collision between two or more complex arms can be significant. As such, optimizations need to be made to insure that collisions can be avoided in a timely manner. For each manipulator in the system, the function to minimize is the following:

$$f(\mathbf{q}) = \sum_{i=1}^L \sum_{j=1}^M \sum_{k=1}^{L(M)} \eta(i, k) F^p(i, k), \quad (13-4)$$

Where L is the number of links in the manipulator, M is the number of manipulators and N is the number of links in manipulator j. Logic for the computation of  $f(\mathbf{q})$  is similar to listing one, with the added layer of querying multiple manipulators, and additional proximity testing required for efficiency.

### 13.4 Manipulator-Environment Collision Avoidance

Manipulator environment collisions are similar to manipulator-manipulator collisions with the exception that environmental objects are guaranteed not to move. This allows us to use data structures that can be preprocessed at load time. A bounding box tree (BB-Tree) data structure was chosen to quickly determine which of the environment obstacles (models as fixed base manipulators with no links) are close by to the manipulator querying them.

### 13.5 System Collision Exclusion

The concept of collision map is extended to the system level. Essentially, the system has a map that describes whether a collision between two links from *different* manipulators or a collision between two manipulators should be excluded from calculations. It does not interfere with the self-collision map, which concerns collisions within one manipulator. If there is little or no possibility that the two links or the two manipulators can collide or if a collision between the two links or the two

manipulators is not of great importance and can be ignored, then the map can be set to exclude the collision computation between that link or manipulator pair. A great example of its usage is a collision between two static objects that are in contact in the scene. For example, we may model a ceiling and a wall with boxes as two separate manipulators. The ceiling and the wall always touch each another and thus a collision always occurs. However, this collision can and indeed should be ignored; otherwise, a collision is always reported and the controller will prevent any robot in the scene from moving.

For each manipulator in the system, the objective function to minimize is

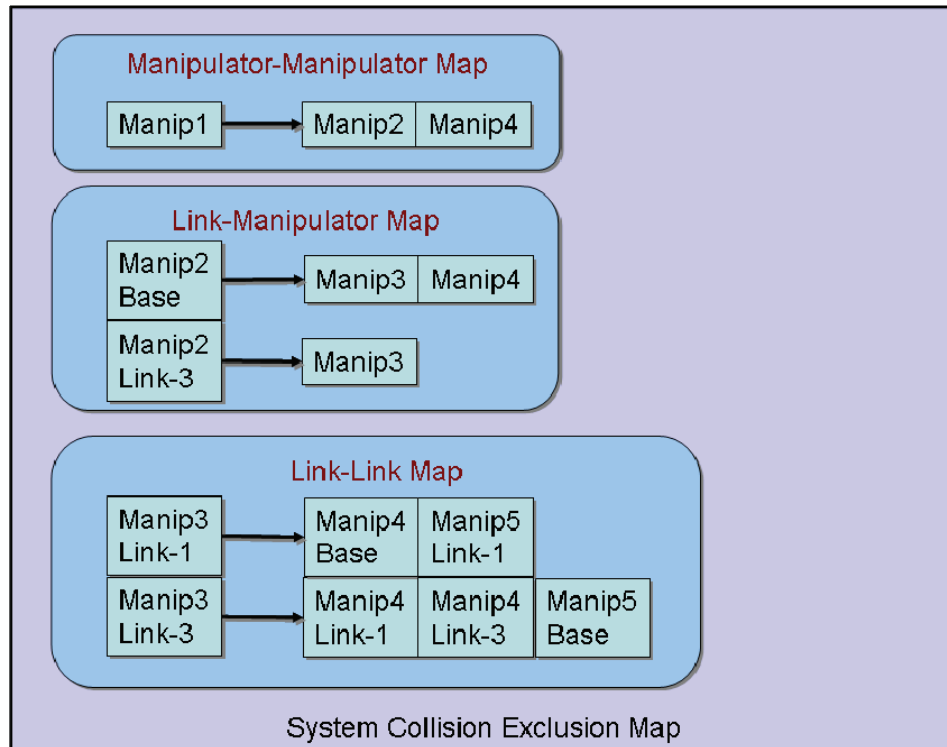
$$f(\mathbf{q}) = \sum_{i=1}^L \sum_{j=1}^M \sum_{k=1}^{L(j)} \delta(i,k) \eta(i,k) F^p(i,k)$$

where  $L$  is the number of links in the manipulator;  $M$  is the number of manipulators in the system, and  $L(j)$  is the number of links in manipulator  $j$ ;

The exclusion of some collision pairs is accomplished using a system collision exclusion map. For fast queries, which are critical to the implementation of the system collision exclusion map, the map has three internal maps as depicted in Figure 13-3 for an example map. The manipulator-manipulator map excludes the collision calculations of all links between two manipulators. The link-manipulator map indicates no collision between a link of a manipulator and all links of another manipulator. Lastly, the link-link map is at the highest granularity, allowing one to specify interaction at the link level.

In this example, there is no link interaction between (these links cannot collide):

- All links of Manip1 and all links of Manip2
- All links of Manip1 and all links of Manip4
- “Base” of Manip2 and all links of Manip3
- “Base” of Manip2 and all links of Manip4
- “Link-3” of Manip2 and all links of Manip3
- “Link-1” of Manip3 and “Base” of Manip4
- “Link-1” of Manip3 and “Link-1” of Manip5
- “Link-3” of Manip3 and “Link-1” of Manip4
- “Link-3” of Manip3 and “Link-3” of Manip4
- “Link-3” of Manip3 and “Base” of Manip5



**Figure 13-3:** An example of a system collision exclusion map.

Text Box 13-2 shows a code snippet describing how one would go about programmatically creating the system collision exclusion map depicted in Figure 13-3. To set the map in the system, just call *EcStatedSystem::setCollisionExclusionMap* or *EcManipulatorSystem::setCollisionExclusionMap* method. Text Box 13-3 illustrates how to query whether the collision calculation between two links of two different manipulators is excluded. Note that the method *buildMapFromSystem* must be called to construct internal data structures, using the information from the system for which this map is intended, before the map is ready for queries.

```
// create a collision exclusion map
EcSystemCollisionExclusionMap colMap;

// exclude collision between manipulators
colMap.excludeManipulatorCollisionCandidates(1,2);
colMap.excludeManipulatorCollisionCandidates(1,4);

// exclude collisions between links and manipulators
colMap.excludeLinkCollisionCandidates(2, "Base", 3);
colMap.excludeLinkCollisionCandidates(2, "Base", 4);
colMap.excludeLinkCollisionCandidates(2, "Link-3", 3);

// exclude collisions between links of different manipulators
colMap.excludeLinkCollisionCandidate(3, "Link-1", 4, "Base");
colMap.excludeLinkCollisionCandidate(3, "Link-1", 5, "Link-1");
colMap.excludeLinkCollisionCandidate(3, "Link-3", 4, "Link-1");
colMap.excludeLinkCollisionCandidate(3, "Link-3", 4, "Link-3");
colMap.excludeLinkCollisionCandidate(3, "Link-3", 5, "Base");

// create a simulation
EcSystemSimulation simulation;
```



```
// set the map in the system
simulation.statedSystem().setCollisionExclusionMap(colMap);
```

**Text Box 13-2:** Creating the system collision exclusion map shown in Figure 13-3.

```
////////////////////////////////////
// example queries about collision exclusion
////////////////////////////////////
// create a system (assuming the system have all the links and the manipulators
required in the map).
EcManipulatorSystem system;

// build the map from the system
colMap.buildMapFromSystem(system);

// this should return true
colMap.isLinkCollisionCandidateExcluded(1, "Link-1", 2, "Link-2");

// this should return true
colMap.isLinkCollisionCandidateExcluded(2, "Base", 3, "Link-2");

// this should return false
colMap.isLinkCollisionCandidateExcluded(3, "Base", 4, "Link-2");

// this should return false
colMap.isLinkCollisionCandidateExcluded(3, "Link-1", 5, "Link-3");
```

**Text Box 13-3:** Querying collision exclusion from the map in Figure 13-3.

The system collision exclusion map has been added as part of the system (*EcManipulatorSystem*). A method called *canCollide* has also been added to *EcManipulatorSystem*. *canCollide* returns true if the collision between the two links is not excluded or false otherwise. It performs the same functionality as the method of the same name in *EcIndividualManipulator* but at the system level, instead of the manipulator level. *canCollide* is automatically called whenever a distance between two links is queried. If it returns false, then the distance calculation between those two links is omitted.

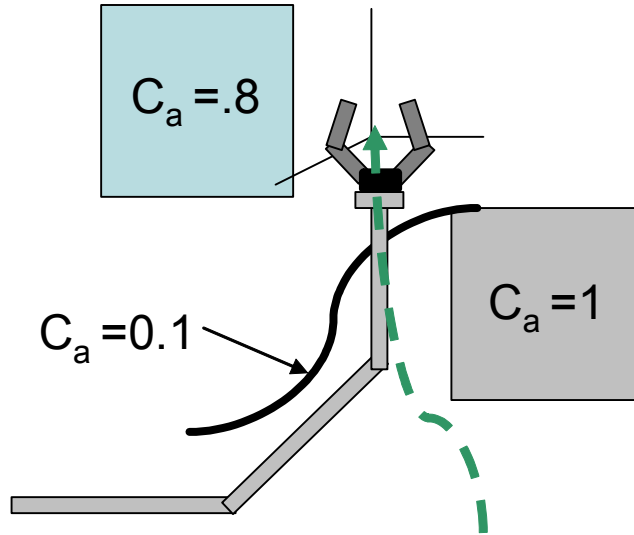
## 13.6 Collision Avoidance as a Function of Material Type

A strength of the Actin<sup>TM</sup> toolkit is that it can take advantage of the material type at both the collision avoidance and the collision response stages. This is done through the flexible surface properties associated with each polygon (in polyhedron representations) or each shape (for homogenous shape primitives). The material type can be assumed to be known with certainty (for manipulator self-collision) or with some probability (based on visual feedback of the environment).

The collision avoidance algorithm accommodates a term which is proportional to the fragility of the object. This term is a parameter in the surface properties called **ec\_collision\_avoidance\_metric**. This term can be a direct value specified through XML (i.e. a fragility constant) or a derived quantity based on more fundamental properties of the object. Figure 13-4 shows a manipulator traversing a path through several objects with different avoidance coefficients. An additional multiplicative term  $\mu_p$  is related to the perception of the robot and is expressed as:

$$\mu_p = \Omega(A) \quad (13-5)$$

Where  $\Omega(A)$  is a [0..1] function returned by the vision module indicating the certainty of the perceived material type. This term is 1 for all controlled environments, but in cases where visual feedback is used to explore new environments, the value could be set by the vision system.



**Figure 13-4:** A commanded action will result in secondary control that minimizes impact with fragile objects. The objects to avoid have high  $c_a$  coefficients.

If not otherwise specified in the XML, surface properties default to a collision avoidance coefficient of 1.

### 13.7 Collision Response

Once a collision occurs, the collision response algorithm determines if the collision should result in a stop or a pass-through. For Version 1.0 of Actin™, this is determined by the surface properties of the interacting links. The `ec_collision_avoidance_metric` parameter is currently used for this determination. Note that this collision response is different from the collision response used by the dynamic simulation. The determination of pass-through or stop is part of the control system and can be used even if dynamic simulation is turned off.

### 13.8 Example: Creating a Collision Avoidance Control Node

The code below creates a collision avoidance control node and adds it to the control system as the vector portion of the control algorithm. In this example, the manipulator will avoid other manipulators and environment obstacles without avoiding collisions with itself. To turn on self-collision avoidance, simply change the line

```
colAvoidance.setCheckSelfCollision(EcFalse);
```

to

```
colAvoidance.setCheckSelfCollision(EcTrue);
```

```

// -----
// Set the Control Description
// -----
EcControlExpressionContainer container0;
// the scalar constant - use for the scalar weight
EcExpressionScalarConstant valueSC;

// set the scalar constant
valueSC.setScalar(0.005);

// the vector
EcControlExpressionCollisionAvoidanceAB colAvoidance;
// set the avoidance distance, beyond which we don't concern ourselves with
collisions
colAvoidance.setAvoidanceDistance(0.05);

// set the value of the function at the boundary
colAvoidance.setBoundaryValue(30);

// tell the system to avoid collisions with the environment,
// which includes both manipulators and obstacles in the task space
colAvoidance.setCheckEnvironmentCollisions(EcTrue);

// tell the system to avoid self collisions.
colAvoidance.setCheckSelfCollision(EcFalse);

// set the exponent for the collision avoidance function
colAvoidance.setExponent(2);

// the matrix
EcControlExpressionMatrixToAB contExpMatToAB;
EcControlExpressionMassMatrix massMatrix;
contExpMatToAB.setChild(massMatrix);

// Diagonal Matrix - use for the matrix weight
EcExpressionDiagonalMatrix valueDD;
valueDD.setRowSize(manip.jointDof());
valueDD.setColumnSize(manip.jointDof());

```

**Text Box 13-4:** Creating a collision avoidance control node Listing 1 of 3.

```

EcXmlRealVector diagonal;
EcU32 ii;
for(ii=0;ii<manip.jointDof();ii++)
{
    diagonal.pushBack(0.1);
}
valueDD.setDiagonal(diagonal);
// Core
EcControlExpressionABCore expCore0;

expCore0.setMatrixElement(contExpMatToAB);
expCore0.setVectorElement(colAvoidance);
expCore0.setScalarElement(valueSC);

// Joint Rate Filter
EcControlExpressionJointRateFilter rateFilter0;
EcExpressionGeneralColumn valueDF;

EcXmlRealVector diagonalF;
for(ii=0;ii<manip.jointDof();ii++)
{
    diagonalF.pushBack(0.10);
}
valueDF.setColumn(diagonalF);
rateFilter0.setWeightsElement(valueDF);
rateFilter0.setUnfilteredRatesElement(expCore0);

// End-Effector Error Filter
EcControlExpressionEndEffectorErrorFilter eFilter0;
EcExpressionGeneralColumn valueDFE;
EcXmlRealVector diagonalFE;
for(ii=0;ii<6;ii++)
{
    diagonalFE.pushBack(10.0);
}
for(ii=6;ii<eeSet.doc();ii++)
{
    diagonalFE.pushBack(1.0);
}
valueDFE.setColumn(diagonalFE);
eFilter0.setWeightsElement(valueDFE);
eFilter0.setUnfilteredRatesElement(rateFilter0);
eFilter0.setStopsAtLimits(EcTrue);

```

**Text Box 13-5:** Creating a collision avoidance control node Listing 2 of 3.

```

// put them in the container
container0.setTopElement(eFilter0);
indVcDescription.setControlExpression(container0);

vcDescriptions.pushBack(indVcDescription);

// add a velocity control system
EcVelocityControlSystem velContSys;
velContSys.setControlDescriptions(vcDescriptions);
velContSys.initializeEndEffectorVelocities();

//set the time step
velContSys.setSystemTimeStep(0.008);

posContSystem.setVelocityControlSystem(velContSys);

// set the time step
posContSystem.setTimeStep(0.008);

// set the maximum number of iterations
posContSystem.setMaxIterations(16);

// set the use-two-passes flag
posContSystem.setUseTwoPasses(EcTrue);

posContSystem.setCollisionAvoidanceMode
(EcStatedSystem::MANIPULATORCOLLISIONAVOIDANCE);

posContSystem.setCollisionBreakdownThreshold(EcReal(0.4));

```

**Text Box 13-6:** Creating a collision avoidance control node Listing 3 of 3.

### 13.9 Distance Queries

A key component of obstacle avoidance is the ability to measure the distance between manipulator links and the environment. In the event of intersection, the penetration depth must also be determined for accurate physical modeling. Distance queries are made through an *EcShapeQueryData* object that contains an *EcShapeQueryDescriptor* and an *EcShapeQueryResult*. Querying options are specified in the *EcShapeQueryDescriptor* object. If only query is needed for the intersection (often a less costly operation than computing the distance), set *m\_QueryDistance* to *EcFalse*. The result of the query is returned in the *EcShapeQueryResult* object.

By setting distance queries up in this way, developers can subclass them to add new data without having to change the interface. Table 13-1 shows the query data class key methods and Table 13-2 and Table 13-3 shows the descriptor class and the result class.

Method	Description
queryDescriptor setQueryDescriptor	Get/set a query descriptor. The query descriptor is the data structure containing the query information.

queryResult setQueryResult	Get/set the query result. The data structure holds the result of a query (i.e. distance)
-------------------------------	--

**Table 13-1:** Shape query data class.

Method	Description
computeDistance setComputeDistance	Get/set the compute distance flag. If false, the query operation will check intersection without necessarily computing the distance.
xform setXform	Get/set the coordinate system transformation that transforms shape 2 from the local primary frame of shape 2's link to the local primary frame of shape 1.
shape1Pointer getShape1Pointer	Get/set the pointer to the first shape that is being queried.
shape2Pointer getShape2Pointer	Get/set the pointer to the second shape that is being queried.
minimumDistanceThreshold setMinimumDistanceThreshold	Get/set the minimum distance threshold. This is the distance that the distance algorithm uses to decide whether or not to move down in the bounding volume hierarchy. If the distance between the two shapes is less than this value, and if the bounding volumes being queried are not at the root level, then the algorithm moves down in the hierarchy and recomputes the distance. The value is typically set to 0.
precisionLevel setPrecisionLevel	Get/set the precision level. This is the level in the bounding volume hierarchy to use as the <b>root</b> level. The default is 0, but it is often useful to use a value > 0 for speed. If, for instance, the root level is a complex polyhedron, and level one is a bounding capsule, then setting the precision level to 1 will greatly reduce the computational burden thus speeding up the simulation.

**Table 13-2:** Query descriptor.

Method	Description
distance setDistance	Get/set the distance between the two shapes pointed to by <i>shape1Pointer</i> and <i>shape2Pointer</i> in the query descriptor.
collisionOccured setCollisionOccured	Get/set a flag indicating if a collision occurred between the two shapes pointed to by <i>shape1Pointer</i> and <i>shape2Pointer</i> in the query descriptor.

intersectingPoints	Returns the points on the surface of the shapes that were involved in the intersection, if one occurred.
intersectionVolume	Returns a scalar describing the volume of intersection. This is not currently implemented and is reserved for future releases.

**Table 13-3:** Query result.

The *EcShape* class contains a static function *query* that will return an *EcShapeQueryResult* object given an *EcShapeQueryData* input.

```

// create a sphere as shape 1
EcSphere sphere = EcSphere::testObject();

// set the center
sphere.setCenter(EcVector(0,10,0));
// set the radius
sphere.setRadius(2.0);

// create a capsule as shape2
EcCapsule capsule = EcCapsule::testObject();

// set the line segment
capsule.setLineSegment(EcLineSegment(EcVector(0,0,0),EcVector(5,0,0));
// set the radius
capsule.setRadius(1.0);

// create a query data descriptor
EcShapeQueryData queryData;

// set values in the query descriptor

// set the pointer to shape 1
queryData.queryDescriptor().setShape1Pointer(&sphere);
// set the pointer to shape 2
queryData.queryDescriptor().setShape2Pointer(&capsule);
// set the compute distance flag to true
queryData.queryDescriptor().setComputeDistance(EcTrue);

// the transformation for transform the primary frame of shape 2
// to the primary frame of shape 1;
EcCoordinateSystemTransformation xf
    =EcCoordinateSystemTransformation::nullObject();

// set the transform, in this case, both shapes are assumed to be
// in the same reference frame
queryData.queryDescriptor().setXform(xf);

// set the precision level for the bounding volume hierarchy.
// here we use 0, meaning that we want the distance returned to be
// the distance to the shapes at the base of the hierarchy. In this
// example there are no bounding volumes. Level zero consists of the
// sphere and the capsule.
queryData.queryDescriptor().setDistanceQueryPrecision(0);

```

**Text Box 13-7:** Performing a distance query Listing 1 of 2.

```

EcReal distance;
// call the static function to compute the query
if (EcShape::query(m_QueryData,activeState))
{
// if successful, get the distance
distance = m_QueryData.queryResult().distance();
}

```

**Text Box 13-8:** Performing a distance query Listing 2 of 2.

Besides the ability to query the distance between any two shape primitives or polyhedrons, it is also possible to query distances between any two link, or manipulators. See the description of the *EcLink* and the *EcIndividualManipulator* classes for a description of how to use these methods. In all cases, distance queries come down to queries between shape primitives and/or polyhedrons.

### 13.9.1 Task Space Bounding Volumes

In order to reduce the computational burden associated with computing the proximity function for all pairs of manipulators and links, the number of pairs queried is reduced by taking advantage of spatial coherence. Spatial coherence makes use of the known locations of the manipulators in the system (i.e. locality in space). This is opposed to temporal coherence where a slow moving link will move a small distance over the resolution of a time step (i.e. locality in time). Temporal coherence is exploited in the collision detection algorithm described in Section 12

Before computing the proximity function for a pair of manipulators, the absolute bounding sphere of each is checked. If the distance between the bounding spheres exceeds the distance threshold  $D$ , the pair is skipped. For this test to work, the algorithm must insure that the bounding sphere completely encloses the task space of the manipulator. The following algorithm is used to compute a bounding sphere that encloses the task space.

*upperBoundSubmanipulatorExtentDistance()*

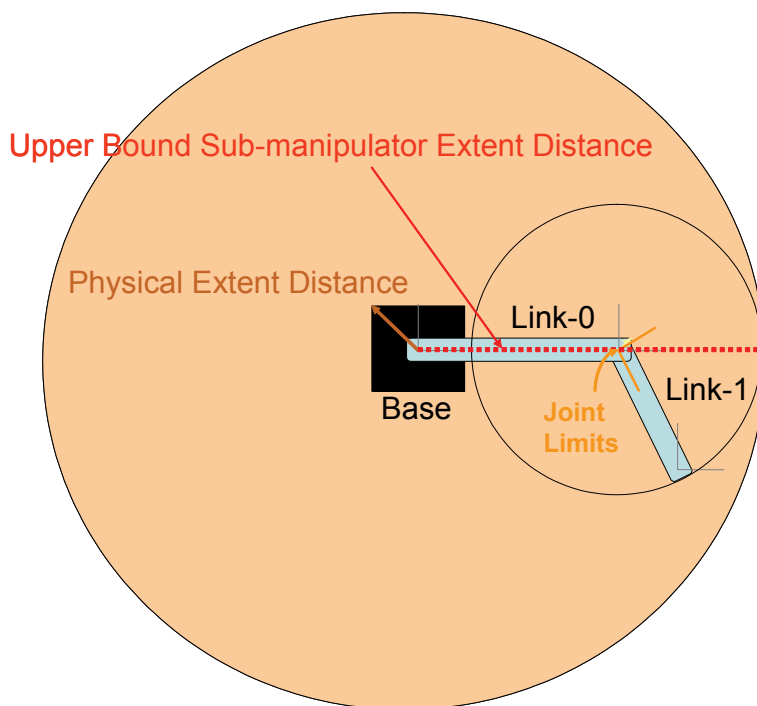
1. subDist = 0
2. **for** each childLink  $l \in M$
3.     **subDist** = **MAX**(subDist, *l.upperBoundSubmanipulatorExtentDistance()*)
4. extDist = *upperBoundExtentDistance()*
5. physicalExtentDist = *maxPhysicalExtentDistance()*
6. **return** (extDist + **MAX**(physicalExtentDist,subDist))

**Listing 2-** Building the Absolute Bounding Sphere

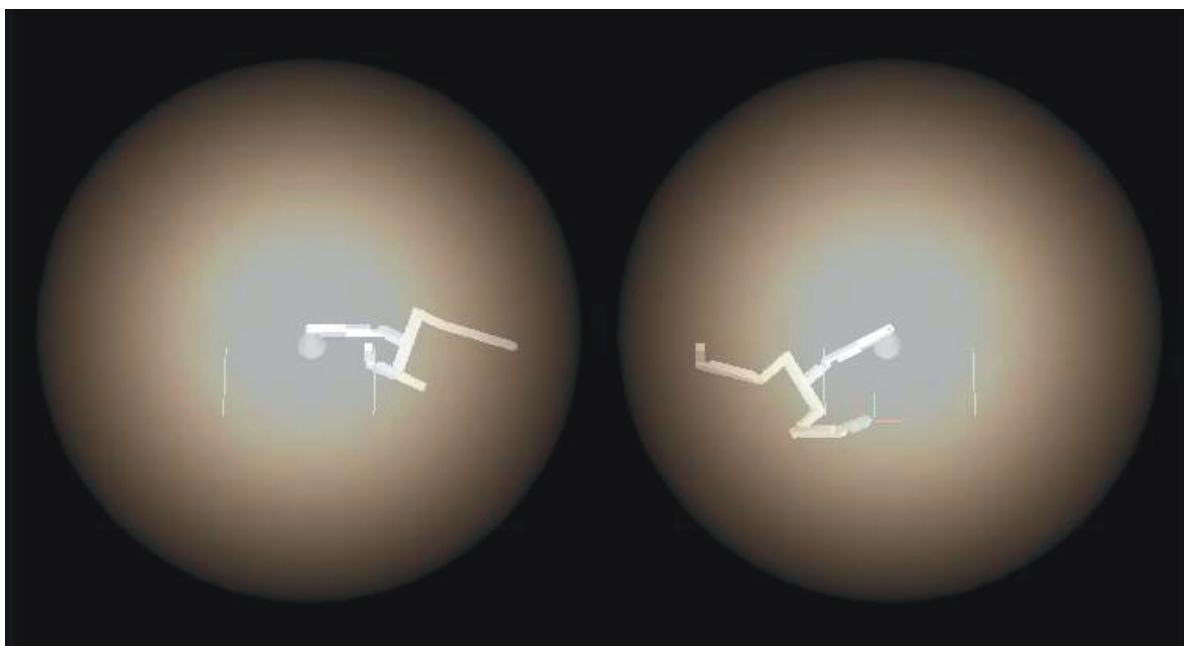
Where *upperBoundExtentDistance* returns the maximum distance from the current DH frame to the next DH. The joint limits and joint types are considered when determining this. And



*maxPhysicalExtentDistance* returns the distance to the polygon in the current physical extent furthest from the DH frame.



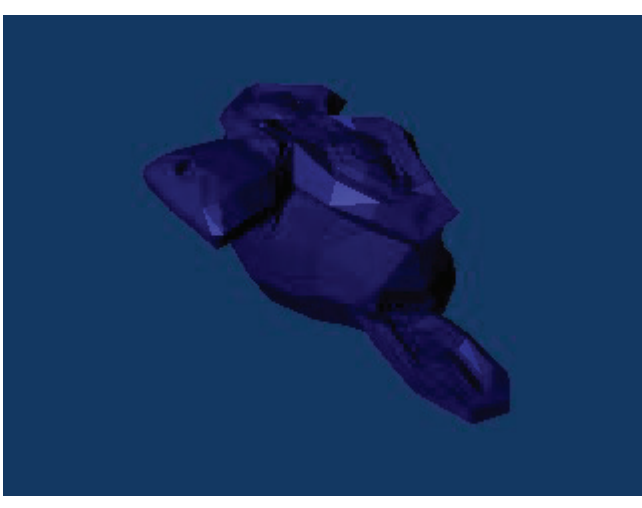
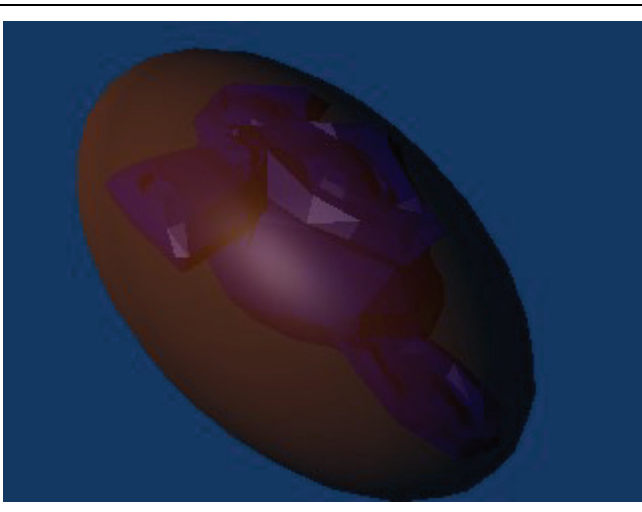
**Figure 13-5:** An illustration of the Bounding Sphere calculation for a two-link manipulator.


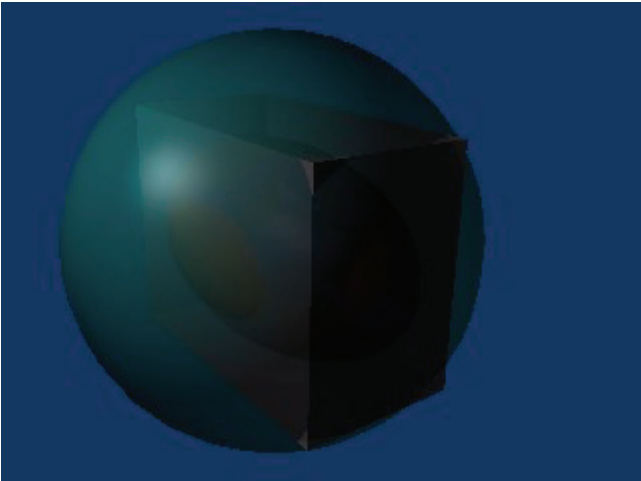


**Figure 13-6:** Absolute Bounding Spheres for the 12-link manipulators.

### 13.9.2 Bounding Volume Hierarchy

Every object that inherits from the *EcShape* base class contains a member variable that points to a low fidelity representation of itself. In the event that there is no low-fidelity representation, the variable points to *EcNULL*. Including this reference in each *EcShape* object allows for the construction of a hierarchical representation of a physical extent. Typically, the bottom of the hierarchy will contain the actual physical extent (usually a polyhedron). Levels above this will generally be less expensive to compute distance and intersection calculations with, but will be a less accurate representation of the actual extent. An example hierarchy is shown in Table 13-4. Here the lowest level is represented by a polyhedron. The low-fidelity approximation of this is an ellipsoid, which, in turn, is approximated by an oriented bounding box, which is approximated by a sphere at the top of the hierarchy.

<p><b>Level 0: Polyhedron</b></p> <p>At the lowest level, a physical extent is typically represented as a polyhedron. Nothing precludes the lowest level physical extent from being represented by a simpler shape primitive however.</p>	
<p><b>Level 1: Ellipsoid</b></p> <p>The low fidelity representation of the polyhedron is represented by an Ellipsoid. The ellipsoid provides a fairly tight fit to the polyhedron and will return a smoothly differentiable distance function. This simplifies obstacle avoidance. Computing distance to the ellipsoid will typically be faster than computing the distance to a complex polyhedron, but it does involve solving a 6<sup>th</sup> order polynomial.</p>	

<p><b>Level 2: Oriented Bounding Box</b></p> <p>The next lowest fidelity representation is an oriented bounding box. The bounding box provides fast distance and intersection checks, but it usually does not fit the data quite as nicely as an ellipsoid. It also is not smoothly differentiable.</p>	
<p><b>Level 3: Sphere</b></p> <p>The lowest fidelity representation is a sphere. The sphere provides an extremely fast and smoothly differentiable approximation.</p>	

**Table 13-4:** An example bounding volume hierarchy.

The reason for creating a bounding volume hierarchy is to minimize the computational complexity of distance and intersection operations. This allows the toolkit to defer more burdensome point-polygon checks until the latest possible moment. There are two items that need consideration with this approach: 1) traversing the hierarchy (i.e. deciding when to go from a lower fidelity model to a higher fidelity model or vice-versa), and 2) offline creation of the BVH.

### **13.9.3 Traversing the hierarchy**

Traversing the bounding volume hierarchy is performed by first imposing minimum and maximum distances restriction on physical extent queries. Since our concern is local collision avoidance, no check is necessary when the distance of objects are separated by a significant amount. For manipulator-environment and manipulator-manipulator collision avoidance this is somewhat mitigated by space partitioning techniques. But the framework should not rely solely on this, as partitions can be very large and manipulator links for sophisticated systems could be separated by large distances, requiring manipulator self-collision optimization as well.

The first time one shape queries another, the top of the bounding volume hierarchy is used. If the distance between the two shapes exceeds a minimum distance threshold, then the query is not

considered further. Otherwise, a maximum query distance threshold is checked (usually set to zero). If the distance computed is below the maximum distance threshold, then the next level of fidelity is used. As two objects approach, this continues until both objects are at the lowest level representation. When two objects at the lowest level intersect, a collision occurs and a penetration distance is calculated.

#### **13.9.4 Example: Adding a Bounding Volume Hierarchy to a shape**

There are two ways in which bounding volumes can be added to represent the physical extent of a link at varying levels of fidelity. Shapes can be explicitly added using the pointers returned by the *EcShape* method *lowerFidelityRepresentation* or implicitly for most shapes using the *addBoundingVolume* method. The *addBoundingVolume* method will attempt to add a bounding volume to the end of the hierarchy by using an estimation algorithm to determine the proper size. The method will return *EcFalse* if unsuccessful. The code below shows how to add two bounding volumes over a polyhedron.

```

// create a sphere as shape 1
EcSphere sphere = EcSphere::testObject();

// set the center
sphere.setCenter(EcVector(0,10,0));
// set the radius
sphere.setRadius(2.0);

// create a capsule as shape2
EcCapsule capsule = EcCapsule::testObject();

// set the line segment
capsule.setLineSegment(EcLineSegment(EcVector(0,0,0),EcVector(5,0,0));
// set the radius
capsule.setRadius(1.0);

// create a query data descriptor
EcShapeQueryData queryData;

// set values in the query descriptor

// set the pointer to shape 1
queryData.queryDescriptor().setShape1Pointer(&sphere);
// set the pointer to shape 2
queryData.queryDescriptor().setShape2Pointer(&capsule);
// set the compute distance flag to true
queryData.queryDescriptor().setComputeDistance(EcTrue);

// the transformation for transform the primary frame of shape 2
// to the primary frame of shape 1;
EcCoordinateSystemTransformation xf
    =EcCoordinateSystemTransformation::nullObject();

// set the transform, in this case, both shapes are assumed to be
// in the same reference frame
queryData.queryDescriptor().setXform(xf);

// set the precision level for the bounding volume hierarchy.
// here we use 0, meaning that we want the distance returned to be
// the distance to the shapes at the base of the hierarchy. In this
// example there are no bounding volumes. Level zero consists of the
// sphere and the capsule.
queryData.queryDescriptor().setDistanceQueryPrecision(0);

```

**Text Box 13-9:** Adding a bounding volume hierarchy.

## 13.10 Penetration Depth Calculation

Computation of penetration depth (PD) between two objects is critical to the performance of impact dynamic simulation. With a simple spring-damper model being used to compute the impact force, penetration depth computation dominates the overall computation time of the impact dynamic simulation. In general, penetration depth for every type of convex object pair can be computed using the expanding polytope (EP) algorithm, which is an iterative method based on the Gilbert-Johnson-Keerthi (GJK) algorithm for distance calculation. Although the EP algorithm is generic and applicable to any convex object, it is significantly slower for simple primitives than closed-form solutions. This section describes in detail the closed-form penetration depth computations for

selected pairs of simple primitives. Beside the penetration depth, other pieces of information about penetration must also be computed. These include the proximity vector and support (or witness) points.

PD calculations of some shape pairs are complicated with many special cases, some of which are unlikely to occur. Consider how PD is used in dynamic simulations. PD is typically used to compute a repulsive force that pushes the two objects away from each other. This means that most of the time, PD calculations will involve “shallow” penetration cases. Therefore, the focus of closed-form PD solutions will be on shallow penetration cases, which tend to be simpler. The less likely and more complicated deep penetrations will still be handled the EP algorithm.

### 13.10.1 Sphere-Sphere

Computing the penetration depth between two intersecting spheres is straightforward. The distance between the two spheres  $A$  and  $B$  is simply the distance between their centers  $\mathbf{c}_A$  and  $\mathbf{c}_B$  minus their radii  $\rho_A$  and  $\rho_B$ . If this distance is negative, then the two spheres intersect and the negative distance is the penetration depth.

$$PD = d(\mathbf{c}_A, \mathbf{c}_B) - \rho_A - \rho_B \quad (13-6)$$

Let the vector between the two centers  $\mathbf{v} = \mathbf{c}_A - \mathbf{c}_B$ . The proximity vector is just the unit vector of  $\mathbf{v}$ . The witness points (or support points) for a nonzero distance are

$$\mathbf{p}_A = \mathbf{c}_A - \rho_A \frac{\mathbf{v}}{\|\mathbf{v}\|} \quad (13-7)$$

and

$$\mathbf{p}_B = \mathbf{c}_B + \rho_B \frac{\mathbf{v}}{\|\mathbf{v}\|}. \quad (13-8)$$

### 13.10.2 Capsule-Sphere

The distance between a capsule  $A$  and a sphere  $B$  is simply the distance between the line segment of the capsule  $\mathbf{l}_A$  and the center of the sphere  $\mathbf{c}_B$  minus their radii  $\rho_A$  and  $\rho_B$ . If this distance is negative, then the capsule and the sphere intersect and the negative distance is the penetration depth.

$$PD = d(\mathbf{l}_A, \mathbf{c}_B) - \rho_A - \rho_B \quad (13-9)$$

Let the point  $\mathbf{c}_A$  be the point on  $\mathbf{l}_A$  closest to  $\mathbf{c}_B$ . Then, the proximity vector and the two witness points are the same as the sphere-sphere case above.

### 13.10.3 Capsule-Capsule

The distance between the two capsules  $A$  and  $B$  is simply the distance between their line segments  $\mathbf{l}_A$  and  $\mathbf{l}_B$  minus their radii  $\rho_A$  and  $\rho_B$ . If this distance is negative, then the two capsules intersect and the negative distance is the penetration depth.

$$PD = d(\mathbf{l}_A, \mathbf{l}_B) - \rho_A - \rho_B \quad (13-10)$$

Let the point  $\mathbf{c}_A$  be the point on  $\mathbf{l}_A$  closest to  $\mathbf{l}_B$  and the point  $\mathbf{c}_B$  be the point on  $\mathbf{l}_B$  closest to  $\mathbf{l}_A$ . Then, the proximity vector and the two support points are the same as the sphere-sphere case above.

#### 13.10.4 **Box-Sphere**

The distance between a box  $A$  and a sphere  $B$  is simply the distance between the box and the center of the sphere  $\mathbf{c}_B$  minus the sphere's radius  $\rho_B$  if  $\mathbf{c}_B$  is outside of the box. If this distance is negative, then the box and the sphere intersect and the negative distance is the penetration depth.

$$PD = d(\text{box}_A, \mathbf{c}_B) - \rho_B \quad (13-11)$$

Let the point  $\mathbf{c}_A$  be the point on the box  $A$  closest to  $\mathbf{c}_B$ . Then, the proximity vector and the two support points are the same as the sphere-sphere case above. If  $\mathbf{c}_B$  is inside the box, it is considered a deep penetration and GJK will be used.

#### 13.10.5 **Box-Capsule**

The distance between a box  $A$  and a capsule  $B$  is simply the distance between the box and the line segment of the capsule  $\mathbf{l}_B$  minus the capsule's radius  $\rho_B$  if  $\mathbf{l}_B$  does not intersect the box. If this distance is negative, then the box and the sphere intersect and the negative distance is the penetration depth.

$$PD = d(\text{box}_A, \mathbf{l}_B) - \rho_B \quad (13-12)$$

Let the point  $\mathbf{c}_A$  be the point on the box  $A$  closest to  $\mathbf{l}_B$  and the point  $\mathbf{c}_B$  be the point on  $\mathbf{l}_B$  closest to the box  $A$ . Then, the proximity vector and the two support points are the same as the sphere-sphere case above. If  $\mathbf{l}_B$  intersects the box, it is considered a deep penetration and GJK will be used.

#### 13.10.6 **Lozenge-Sphere**

The distance between a lozenge  $A$  and a sphere  $B$  is simply the distance between the rectangle of the capsule  $\mathbf{r}_A$  and the center of the sphere  $\mathbf{c}_B$  minus their radii  $\rho_A$  and  $\rho_B$ . If this distance is negative, then the lozenge and the sphere intersect and the negative distance is the penetration depth.

$$PD = d(\mathbf{r}_A, \mathbf{c}_B) - \rho_A - \rho_B \quad (13-13)$$

Let the point  $\mathbf{c}_A$  be the point on  $\mathbf{r}_A$  closest to  $\mathbf{c}_B$ . Then, the proximity vector and the two support points are the same as the sphere-sphere case above.

#### 13.10.7 **Lozenge-Capsule**

The distance between a lozenge  $A$  and a capsule  $B$  is simply the distance between the rectangle of the capsule  $\mathbf{r}_A$  and the line segment of the capsule  $\mathbf{l}_B$  minus their radii  $\rho_A$  and  $\rho_B$  if  $\mathbf{l}_B$  does not intersect  $\mathbf{r}_A$ . If this distance is negative, then the box and the sphere intersect and the negative distance is the penetration depth.

$$PD = d(\mathbf{r}_A, \mathbf{l}_B) - \rho_A - \rho_B \quad (13-14)$$

Let the point  $\mathbf{c}_A$  be the point on  $\mathbf{r}_A$  closest to  $\mathbf{l}_B$  and the point  $\mathbf{c}_B$  be the point on  $\mathbf{l}_B$  closest to  $\mathbf{r}_A$ . Then, the proximity vector and the two support points are the same as the sphere-sphere case above. If  $\mathbf{l}_B$  intersects  $\mathbf{r}_A$ , it is considered a deep penetration and GJK will be used.

### 13.10.8 Lozenge-Lozenge

The distance between two lozenges  $A$  and  $B$  is simply the distance between their rectangles  $\mathbf{r}_A$  and  $\mathbf{r}_B$  minus their radii  $\rho_A$  and  $\rho_B$  if  $\mathbf{r}_A$  does not intersect  $\mathbf{r}_B$ . If this distance is negative, then the box and the sphere intersect and the negative distance is the penetration depth.

$$PD = d(\mathbf{r}_A, \mathbf{r}_B) - \rho_A - \rho_B \quad (13-15)$$

Let the point  $\mathbf{c}_A$  be the point on  $\mathbf{r}_A$  closest to  $\mathbf{r}_B$  and the point  $\mathbf{c}_B$  be the point on  $\mathbf{r}_B$  closest to  $\mathbf{r}_A$ . Then, the proximity vector and the two support points are the same as the sphere-sphere case above. If  $\mathbf{r}_A$  intersects  $\mathbf{r}_B$ , it is considered a deep penetration and GJK will be used.

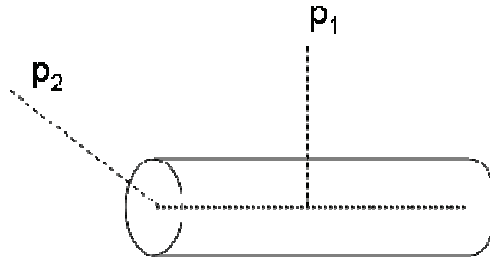
### 13.10.9 Cylinder-Sphere

The distance between a cylinder  $A$  and a sphere  $B$  is simply the distance between the cylinder and the center of the sphere  $\mathbf{c}_B$  minus the radius of the sphere  $\rho_B$  if  $\mathbf{c}_B$  is not inside the cylinder.

$$PD = d(\text{cylinder}_A, \mathbf{c}_B) - \rho_B \quad (13-16)$$

Let the point  $\mathbf{c}_A$  be the point on the cylinder  $A$  closest to  $\mathbf{c}_B$ . Then, the proximity vector and the two support points are the same as the sphere-sphere case above. If  $\mathbf{c}_B$  is inside the cylinder, it is considered a deep penetration and GJK will be used.

The main effort here, thus involves determining the distance between a cylinder and a point. There are basically two cases: (1) the point is close to the side of the cylinder and (2) the point is close to one of the cylinder's circular end caps. To determine which case, we simply determine where on the cylinder's line segment is closest to the point.



**Figure 13-7:** Two cases of distance calculations between a point and a cylinder.  $p_1$  is close to the side of the cylinder while  $p_2$  is close to an end cap.

In the first case, the distance is simply the distance from the point to the cylinder's line segment minus the cylinder's radius  $\rho_A$ .

$$d(\text{cylinder}_A, \mathbf{p}) = d(\mathbf{l}_A, \mathbf{p}) - \rho_A \quad (13-17)$$

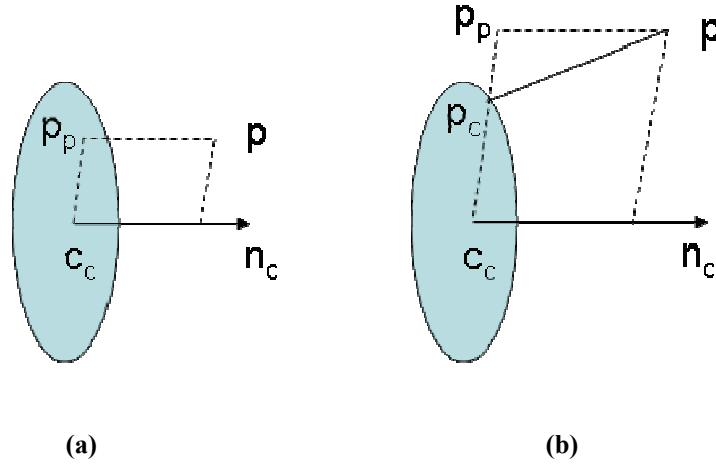
Let the point on the line segment closest to  $\mathbf{p}$  be  $\mathbf{c}_A$ . Let the vector pointing from  $\mathbf{p}$  to  $\mathbf{c}_A$  be  $\mathbf{v} = \mathbf{c}_A - \mathbf{p}$ . The witness point on the cylinder is

$$\mathbf{p}_A = \mathbf{c}_A - \rho_A \frac{\mathbf{v}}{\|\mathbf{v}\|} \quad (13-18)$$



In the second case, it is the distance from the point to the circle that represents the end cap. Let  $\mathbf{c}_C$ ,  $\mathbf{n}_C$ , and  $r_C$  be the center, the normal vector, and the radius of the circle. Note that the radius of the circle  $r_C$  is identical to the radius of the cylinder  $\rho_A$ . Let the projection of point  $\mathbf{p}$  onto the plane of the circle be  $\mathbf{p}_P$ , which can be expressed as:

$$\mathbf{p}_P = \mathbf{p} - (\mathbf{n}_C \cdot (\mathbf{p} - \mathbf{c}_C))\mathbf{n}_C \quad (13-19)$$



**Figure 13-8:** Distance between a point and a circle: (a) the point is projected onto inside the circle and (b) the point is projected outside the circle.

There are two cases as shown in Figure 13-8.

(1) The projection point  $\mathbf{p}_P$  is inside the circle. This is true if  $d(\mathbf{p}_P, \mathbf{c}_C) \leq r_C$ . In this case, the point on the circle closest to  $\mathbf{p}$  (the witness point) is simply the projection point  $\mathbf{p}_P$  and the distance between the point and the circle is the distance between  $\mathbf{p}$  and the plane that supports the circle, which can be expressed as

$$d(\text{circle}, \mathbf{p}) = \mathbf{n}_C \cdot (\mathbf{p} - \mathbf{c}_C). \quad (13-20)$$

(2) The projection point  $\mathbf{p}_P$  is outside the circle, i.e.  $d(\mathbf{p}_P, \mathbf{c}_C) > r_C$ . In this case, the witness point is located at the intersection of the line  $\mathbf{p}_P - \mathbf{c}_C$  and the edge of the circle (shown as the point  $\mathbf{p}_C$  in Figure 13-8(b)).

$$\mathbf{p}_C = \mathbf{c}_C + \frac{r_C}{\|\mathbf{p}_P - \mathbf{c}_C\|}(\mathbf{p}_P - \mathbf{c}_C) \quad (13-21)$$

The distance between the point and the circle is now merely the distance between the two points  $\mathbf{p}$  and  $\mathbf{p}_C$ .

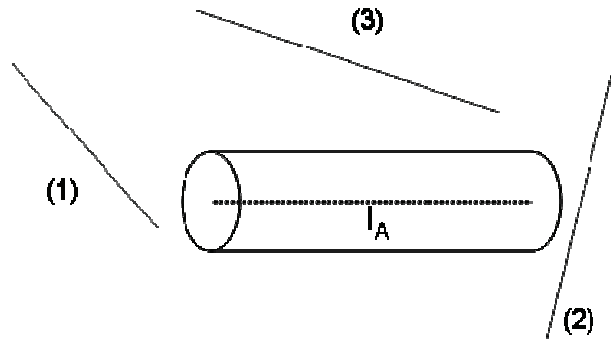
### 13.10.10 Cylinder-Capsule

The distance between a cylinder  $A$  and a capsule  $B$  is simply the distance between the cylinder and the line segment of the capsule  $\mathbf{l}_B$  minus the radius of the capsule  $\rho_B$  if  $\mathbf{l}_B$  does not intersect the cylinder.

$$PD = d(\text{cylinder}_A, \mathbf{l}_B) - \rho_B \quad (13-22)$$

Let the point  $\mathbf{c}_A$  be the point on the cylinder  $A$  closest to  $\mathbf{l}_B$  and the point  $\mathbf{c}_B$  be the point on  $\mathbf{l}_B$  closest to the cylinder  $A$ . Then, the proximity vector and the two support points are the same as the sphere-sphere case above. If  $\mathbf{l}_B$  intersects the cylinder, it is considered a deep penetration and GJK will be used.

The difficulty is in determining the distance and the witness points between a cylinder  $A$  and a line segment  $\mathbf{l}_B$ . There are three cases as shown and described in Figure 13-9.



**Figure 13-9:** Three cases of distance calculations between a line segment and a cylinder. (1) An endpoint of the line segment is closest to the cylinder. (2) The mid-section of the line segment (not endpoint) is closest to an end cap. (3) The mid-section of the line segment is closest to the side of the cylinder.

Now let's examine these cases in details.

*(1) An endpoint of the line segment is closest to the cylinder*

In this case, the problem degenerates to the distance between a point and a cylinder. First, find out which of the two endpoints is closest to the cylinder and then find the distance between that endpoint and the cylinder using the procedure described earlier.

*(2) The mid-section of the line segment is closest to an end cap of the cylinder*

In this case, we first need to determine which end cap is closer to the line segment. Then, the distance between the line segment and the cylinder is just the distance between the line segment and a circle that represents that end cap. This is more complicated than the other two cases and will be detailed later.

*(3) The mid-section of the line segment is closest to the side of the cylinder*

This case is straightforward since the distance between  $\mathbf{l}_B$  and the cylinder  $A$  is just the distance between  $\mathbf{l}_B$  and the cylinder's line segment  $\mathbf{l}_A$  minus the cylinder's radius  $\rho_A$ . Denote the closest points between these two line segments as  $\mathbf{c}_A$  and  $\mathbf{c}_B$ . Let the vector pointing from  $\mathbf{c}_B$  to  $\mathbf{c}_A$  be  $\mathbf{v} = \mathbf{c}_A - \mathbf{c}_B$ . The witness point on the cylinder  $A$  is

$$\mathbf{p}_A = \mathbf{c}_A - \rho_A \frac{\mathbf{v}}{\|\mathbf{v}\|} \quad (13-23)$$

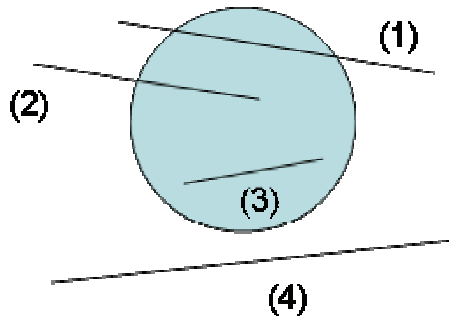
Now we detail the calculation of the distance between a line segment  $\mathbf{l}$  and a circle  $C$ . Let  $\mathbf{c}_C$ ,  $\mathbf{n}_C$ , and  $r_C$  be the center, the normal vector, and the radius of the circle. Note that the radius of the circle  $r_C$  is identical to the radius of the cylinder  $\rho_A$ . We need to consider whether the line segment is parallel to the circle's plane.

**Scenario 1: Line segment is parallel to the circle's plane**

In this scenario, the line segment is projected onto the plane. The projected line segment will either intersect or not intersect the circle. Lines (1)-(3) in Figure 13-10 show three possibilities of how the projected line segment can intersect the circle. In these cases, the distance between the line segment and the circle is just the distance between the line segment and the circle's plane, which in turn is the distance between one of the line segment's endpoint and the plane since the line segment is parallel to the plane. Let  $\mathbf{p}_0$  be the first endpoint of the line segment  $\mathbf{l}$ , then the distance between  $\mathbf{l}$  and the circle is

$$d(\text{circle}, \mathbf{l}) = d(\text{plane}, \mathbf{p}_0). \quad (13-24)$$

Because of the parallel, the witness points on both the line segment and the circle are arbitrary. One sensible choice for the witness point on the line segment is the middle of its section that lies inside the circle. We need to find where the line segment intersects the circle by solving a quadratic equation.



**Figure 13-10:** Top view of four different cases when the line segment is parallel to the circle. (1) – (3) The projection of the line segment intersects the circle. (4) The projection lies completely outside the circle.

Let  $\mathbf{ep}_0$  and  $\mathbf{ep}_1$  be the endpoints of the projected line segment and  $\mathbf{b}=\mathbf{ep}_1-\mathbf{ep}_0$ . Let  $\mathbf{q}=\mathbf{ep}_0 + t\mathbf{b}$  be a point on the projected line segment, where  $t$  is line parameter between 0 and 1. We need to solve the following equation for  $t$ .

$$\begin{aligned} \|\mathbf{q} - \mathbf{c}_C\|^2 &= r_C^2 \\ \|\mathbf{ep}_0 + t\mathbf{b} - \mathbf{c}_C\|^2 &= r_C^2 \end{aligned} \quad (13-25)$$

Let  $\mathbf{x}=\mathbf{ep}_0-\mathbf{c}_C$ . The above equation becomes

$$\begin{aligned} \|\mathbf{b}\|^2 t^2 + 2\mathbf{x} \cdot \mathbf{b}t + \|\mathbf{x}\|^2 - r_C^2 &= 0 \\ at^2 + bt + c &= 0 \end{aligned} \quad (13-26)$$

This is just a standard quadratic equation. The solutions to this equation are

$$\begin{aligned} t_0 &= \frac{-b - \sqrt{b^2 - 4ac}}{2a} \\ t_1 &= \frac{-b + \sqrt{b^2 - 4ac}}{2a} \end{aligned} \quad (13-27)$$

It has two solutions because a line can intersect a circle at two points. Note that  $t_1$  is always greater than or equal to  $t_0$  since  $\sqrt{b^2 - 4ac}$  and  $a$  are always positive. Let  $t_w$  be the t-value of the witness point on the line segment.

Case (1): Both endpoints are outside the circle ( $t_0 > 0$  and  $t_1 < 0$ ). In this case,  $t_w = (t_0 + t_1)/2$ .

Case (2): One endpoint is inside and the other is outside the circle (either  $t_0 < 0$  and  $t_1 < 1$  or  $t_0 > 0$  and  $t_1 > 1$ ). In this case,  $t_w = t_1/2$  or  $(t_0 + 1)/2$ .

Case (3): Both endpoints are inside the circle ( $t_0 < 0$  and  $t_1 > 1$ ). In this case,  $t_w = 0.5$ .

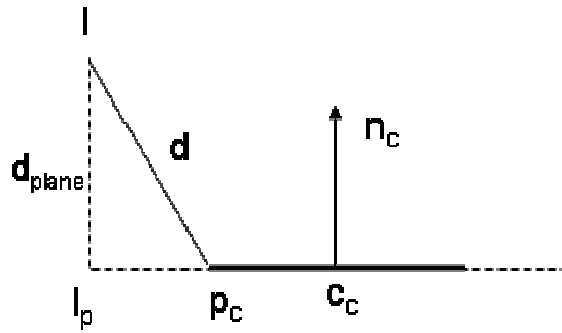
Let the endpoints of the line segment be  $\mathbf{p}_0$  and  $\mathbf{p}_1$ , the witness point on the line segment can then be expressed as

$$\mathbf{p}_l = \mathbf{p}_0 + t_w(\mathbf{p}_1 - \mathbf{p}_0) \quad (13-28)$$

In all three cases, the witness point on the circle is just the projection of the witness point on the line segment onto the circle's plane.

For case (4) where the projection of the line segment lies completely outside the circle, consider Figure 13-11. Let  $\mathbf{l}$  denote the line segment and  $\mathbf{l}_p$  the projection of the line segment. Note here that they both are seen as points since we are looking the side view. Let the distance from the line segment to the circle's plane be  $d_{plane}$  and the distance between  $\mathbf{l}_p$  and  $\mathbf{c}_C$  be  $d_{cp}$ . Then the distance between the line segment and the circle is

$$d(circle, \mathbf{l}) = \sqrt{d_{plane}^2 + (d_{cp} - r_C)^2} . \quad (13-29)$$



**Figure 13-11:** Side view of case (4) where the projection of the line segment lies completely outside the circle. The dashed line is the circle's plane and the thick solid line represents the side of the circle.

Let  $\mathbf{v}$  be the vector from  $\mathbf{c}_c$  to  $\mathbf{l}_p$ . The witness point on the circle  $\mathbf{p}_c$  is

$$\mathbf{p}_c = \mathbf{c}_c + \frac{r_c}{\|\mathbf{v}\|} \mathbf{v} \quad (13-30)$$

### Scenario 2: Line segment is not parallel to the circle's plane

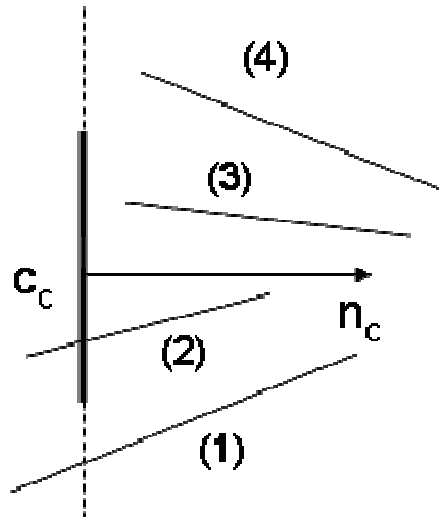
We first need to determine which of the four possibilities are shown in Figure 13-12 is the case. To achieve this, we need to find where the line segment intersects the circle's plane by solving a linear equation. Let the endpoints of the line segment be  $\mathbf{p}_0$  and  $\mathbf{p}_1$  and  $\mathbf{b} = \mathbf{p}_1 - \mathbf{p}_0$ . Let  $\mathbf{q} = \mathbf{p}_0 + t\mathbf{b}$  be a point on the line segment, where  $t$  is line parameter between 0 and 1. Solving the following equation for  $t$

$$\begin{aligned} (\mathbf{q} - \mathbf{c}_c) \cdot \mathbf{n}_c &= 0 \\ (\mathbf{p}_0 + t\mathbf{b} - \mathbf{c}_c) \cdot \mathbf{n}_c &= 0 \end{aligned} \quad (13-31)$$

yields  $t_i$ , which is the  $t$  value at the intersection point).

$$t_i = \frac{(\mathbf{c}_c - \mathbf{p}_0) \cdot \mathbf{n}_c}{\mathbf{b} \cdot \mathbf{n}_c}. \quad (13-32)$$

If  $0 \leq t_i \leq 1$ , then the line segment intersects the plane (cases (1) and (2) in Figure 13-12). Otherwise, the extension of the line segment intersects the plane (cases (3) and (4) in Figure 13-12). The intersection point can be found as  $\mathbf{q}_i = \mathbf{p}_0 + t_i \mathbf{b}$ . If the magnitude of the vector  $\mathbf{q}_i - \mathbf{c}_c$  is less than  $r_c$ , then the intersection point lies inside the circle (cases (2) and (3)).



**Figure 13-12:** Side view of four different cases when the line segment is not parallel to the circle. The dashed line is the circle's plane and the thick solid line represents the side of the circle. (1) The line segment intersects the plane outside the circle. (2) The line segment intersects the plane inside the circle. (3) The extension of the line segment intersects the plane inside the circle. (4) The extension of the line segment intersects the plane outside the circle.

Cases (1) and (4):

Let  $\mathbf{v}$  be the vector from  $\mathbf{c}_c$  to  $\mathbf{q}_i$ . The witness point on the circle  $\mathbf{p}_c$  is

$$\mathbf{p}_c = \mathbf{c}_c + \frac{r_c}{\|\mathbf{v}\|} \mathbf{v} \quad (13-33)$$

The distance between the line segment and the circle is just the distance between the witness point  $\mathbf{p}_c$  and the line segment.

Case (2):

Since the line segment intersects the plane inside the circle, the distance between the line segment and the circle is zero and the witness points on both the circle and the line segment is the intersection point  $\mathbf{q}_i$ .

Case (3):

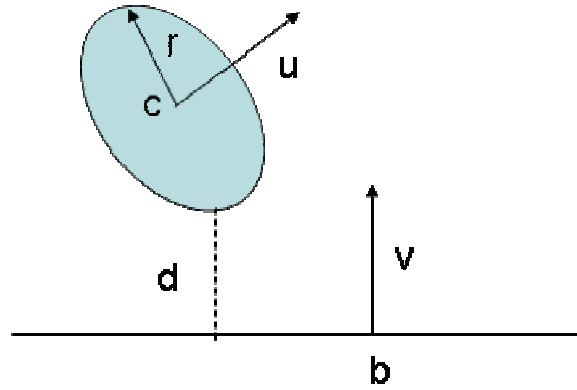
In this case, one of the endpoints of the line segment is closest to the circle. If  $t_i < 0$ , then the closest endpoint is the first one. If  $t_i > 1$ , then it is the second endpoint. Either way, this case degenerates to finding the distance between a point and a circle, which has already been discussed above.

### 13.10.11 Cylinder-Half Space

A half space bisects the 3D space into two halves and is represented by a bisecting plane. The normal vector of the plane points out of the half space. To find PD between a cylinder and a half space, we need to first find a distance between a circle and a plane.

Let  $r$ ,  $\mathbf{c}$ , and  $\mathbf{u}$  be the radius, the center, and the normal vector of the circle, respectively. Let  $\mathbf{b}$  and  $\mathbf{v}$  be the base point and the normal vector of the plane. The distance between the circle and the plane is given by

$$d(\text{circle}, \text{plane}) = (\mathbf{c} - \mathbf{b}) \cdot \mathbf{v} - r \|\mathbf{u} \times \mathbf{v}\|. \quad (13-34)$$



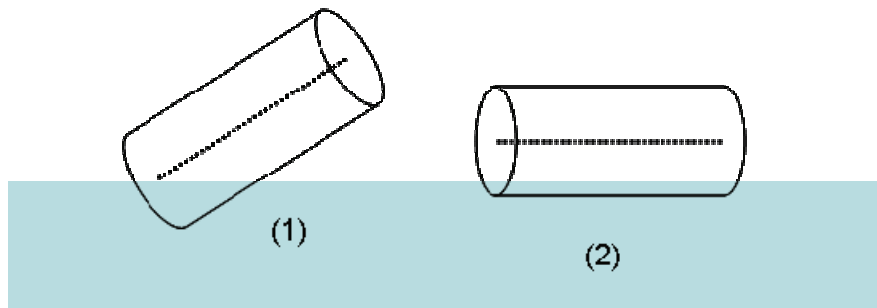
**Figure 13-13:** Distance between a circle and a plane.

If the circle and the plane are not parallel ( $\|\mathbf{u} \times \mathbf{v}\| > 0$ ), the witness point on the circle  $\mathbf{p}_C$  is given by

$$\mathbf{p}_C = \mathbf{c} + \frac{r}{\|\mathbf{u} \times \mathbf{v}\|} (\mathbf{u} \times \mathbf{v}) \quad (13-35)$$

If the circle and the plane are parallel, then the center of the circle is chosen as the witness point. In either case, the witness point on the plane is just the projection of  $\mathbf{p}_C$  onto the plane.

Now consider the distance between cylinder and a half space. We start by computing the distance from one end cap of the cylinder to the plan of the half space. Then, use equation (30) to determine the witness point on the other end cap and compute the distance from that point to the plane. Then, comparing the two distances will yield either case (1) or (2) in Figure 13-14. In case (1) in which one end cap is closer to the half space, the minimum distance is chosen with the corresponding witness points. In case (2), the two distances are equal. The witness point on the cylinder is then chosen to be the mid-point on the side of the cylinder.



**Figure 13-14:** Two cases of a cylinder penetrating a half space. (1) One end cap is closer to the half space. (2) The side of the cylinder is parallel to the half space.

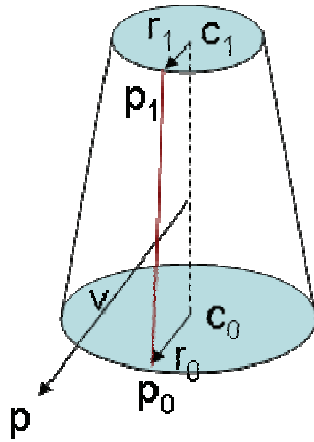
### 13.10.12 Cone-Sphere

Before diving into details of the cone-sphere pair, let's discuss similarities and differences between the cone and the cylinder. The cone in Actin is actually a frustum and is similar to the cylinder. The

only difference, which makes the cone more complicated, is that the side of the cone is not parallel to its line segment. As far as the two circular end caps, the cone is identical to the cylinder except that the two end caps do not have to have the same radius. However, the process of finding the PD between the end caps of the cone to other shapes is exactly the same as that of the cylinder. Therefore, for cone discussion, we will only focus on the PD calculation from the side (lateral surface) of the cone to other shapes.

The distance between the lateral surface of a cone and a sphere is merely the distance between the lateral surface and the center of the sphere minus the radius of the sphere if the center is not inside the cone. If the center is inside the cone, then this will be considered a deep penetration and GJK will be used.

Thus, let's consider the distance between a point  $\mathbf{p}$  and the lateral surface of a cone whose end caps are circles with centers  $\mathbf{c}_0$  and  $\mathbf{c}_1$  and radii  $r_0$  and  $r_1$  (see Figure 13-15).



**Figure 13-15:** Distance between the lateral surface of a cone and a point.

First, find the vector  $\mathbf{v}$  pointing to the cone's line segment to point  $\mathbf{p}$ . This is accomplished by a method that computes the distance between a point and a line segment. This method also provides the information about the point on the line segment closest to  $\mathbf{p}$ , in terms of a  $t$ -value.

$$\mathbf{v} = \mathbf{p} - (\mathbf{c}_0 + t(\mathbf{c}_1 - \mathbf{c}_0)) \quad (13-36)$$

If the magnitude of  $\mathbf{v}$  is less than the radius of the circle at formed by the plane along  $\mathbf{v}$  and perpendicular to the cone's line segment intersecting the cone,  $\mathbf{p}$  is inside the cone. This radius is given by  $r = r_0 + t(r_1 - r_0)$ . If  $\mathbf{p}$  is inside the cone, then the distance between  $\mathbf{p}$  and the cone is zero.

$\mathbf{v}$  now can be used to find a line segment on the lateral surface closest to  $\mathbf{p}$ . This line segment will have endpoints  $\mathbf{p}_0$  and  $\mathbf{p}_1$  as shown Figure 13-15, which are given by

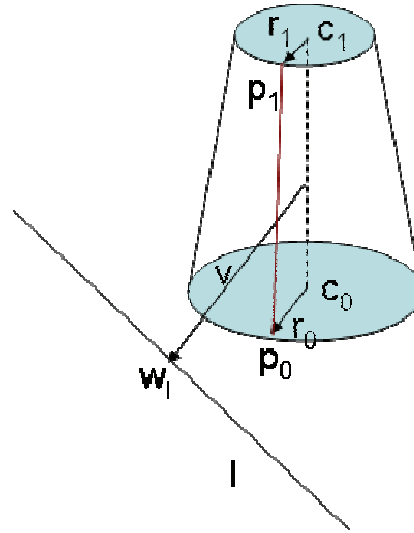
$$\mathbf{p}_i = \mathbf{c}_i + \frac{r_i}{\|\mathbf{v}\|} \mathbf{v}, \quad i=0,1 \quad (13-37)$$

If  $\mathbf{p}$  is not inside the cone, given the line segment on the lateral surface closest to  $\mathbf{p}$ , the distance between the cone and  $\mathbf{p}$  is simply the distance from that line segment to  $\mathbf{p}$ . The point on the line segment is also the witness point on the cone.



### 13.10.13 Cone-Capsule

The distance between the lateral surface of a cone and a capsule is merely the distance between the lateral surface and the line segment of the capsule minus the radius of the capsule if the line segment does not intersect the cone. If the line segment intersects the cone, then this will be considered a deep penetration and GJK will be used.



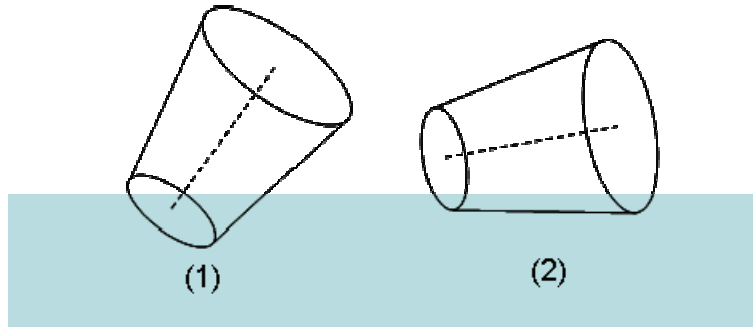
**Figure 13-16:** Distance between the lateral surface of a cone and a line segment.

To compute the distance between a line segment  $I$  and the lateral surface of a cone, we first need to compute the witness points between  $I$  and the cone's line segment (see Figure 13-16). Let  $w_I$  denote the witness point on  $I$ . Then, we compute the line segment on the lateral surface closest to  $w_I$ , using the procedure previously detailed in the cone-sphere pair. If  $w_I$  is inside the cone, then  $I$  intersects the cone and the distance between  $I$  and the cone is zero.

Let  $I_l$  denote the line segment on the lateral surface. If  $I$  does not intersect the cone, the distance between the cone and  $I$  is simply the distance from  $I_l$  to  $I$ . The point on  $I_l$  closest to  $I$  is also the witness point on the cone.

### 13.10.14 Cone-Half Space

Consider Figure 13-17, due to the nature of the half space, the computation of PD between a cone and a half space and that between a cylinder and a half space is almost identical. The only difference is that for a cylinder, we can choose either end cap as the starting point since both have the same radius. For a cone, one end cap could be a point (if the radius is zero). Thus, we should use the end cap with a larger radius as the starting point. Later steps are exactly the same as those for cylinder-half space.



**Figure 13-17:** Two cases of a cone penetrating a half space. (1) One end cap is closer to the half space. (2) The side of the cone is parallel to the half space.

### 13.10.15 Implementations

Previously, the PD computation is handled inside *EcShape::computePenetrationDistanceAndSupportPoints* method, which simply used the GJK algorithm to compute both PD and support (witness) points. Subclasses of *EcShape* should override this method to implement closed-form solutions for PD and support points.

```

/// compute the penetration distance and support points
virtual EcBoolean computePenetrationDistanceAndSupportPoints
(
    const EcShape& otherShape,
    const EcCoordinateSystemTransformation& xform,
    EcVector& supportPointA,
    EcVector& supportPointB,
    EcReal& distance,
    EcVector& proxVec
) const;

```

**Text Box 13-10:** The method for computing PD and related information in *EcShape*.

Argument	Description
otherShape	The other shape with which this shape intersects.
xform	Transforms the other shape to this shape's frame.
supportPointA	Upon return, the support point on this shape in local coordinates.
supportPointB	Upon return, the support point on the other shape in local coordinates.
distance	Upon return, the penetration distance (negative or zero).
proxVec	Upon return, the proximity vector, which is a vector in the direction of impact (from the other shape to this shape).

**Text Box 13-11:** Arguments to *computePenetrationDistanceAndSupportPoints* method.

To prevent the duplication of code for the same shape pair (e.g. capsule-cone and cone-capsule), we follow the following established convention. Based on the enumerations defined in *EcShape* and shown in Text Box 13-12, the computation of PD and support points between a shape pair is only

implemented in the class with the larger enumeration value. For example, PD between the cone-capsule pair is implemented in *EcCone*, not *EcCapsule*, since CONE comes after CAPSULE and thus has a larger enumeration value. Before a call to compute PD, the two shapes will be checked for their enumeration values and, if necessary, the order will be switched to ensure that *computePenetrationDistanceAndSupportPoints* is never called from the shape with smaller enumeration value.

```

/// enumerations for supported geometry
typedef enum
{
    POINT_POLYGON,      /// A point polygon representation
    POLY_PHYSICAL,     /// A point physical extent
    TRI_PHYSICAL,      /// A triangular physical extent
    SPHERE,            /// A sphere defined by a radius and center
    CAPSULE,           /// A capsule defined by a line segment and a radius
    ELLIPSOID,         /// An ellipsoid
    TETRAHEDRON,      /// A tetrahedron defined by a point and three edges
    OBB,               /// An oriented bounding box
    LOZENGE,          /// A lozenge defined by a 3D rectangle and a radius
    HALFSPACE,        /// Half-space defined by a plane and normal vector
    GRID,             /// A grid shape made up of bricks
    CYLINDER,         /// A Cylinder defined by a line segment and radius
    CONE,             /// A Cone defined by a line segment and 2 radii
    UNION,            /// composite shape that is a union of other shapes
    INTERSECTION,     /// Intersection of two other shapes
} EcShapeValues ;

```

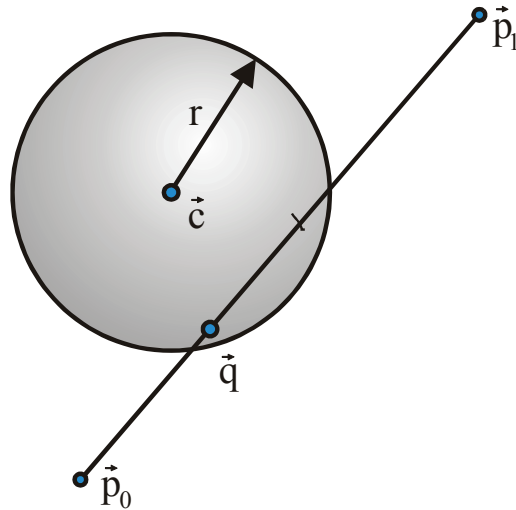
**Text Box 13-12:** Enumerations for different shapes.

## 13.11 Line Segment Intersection

The sections below describe the line-segment shape intersection formulas. The sections are broken down by shape.

### 13.11.1 Sphere

An example sphere is shown in the figure below. It has two parameters, a 3D center point and a radius. The line-segment parameters are also shown, comprising two 3D endpoints.



**Figure 13-18:** Sphere intersection. The sphere parameters of center and radius are shown, as well as the line-segment parameters of two endpoints. The first intersection point is identified by  $\vec{q}$ .

Given the figure above, define

$$\vec{d} = \vec{p}_1 - \vec{p}_0. \quad (13-38)$$

Then, to calculate the intersection of a line segment and a sphere, let the closest intersection point,  $\vec{q}$ , be defined as

$$\vec{q} = \vec{p}_0 + t\vec{d}, \quad (13-39)$$

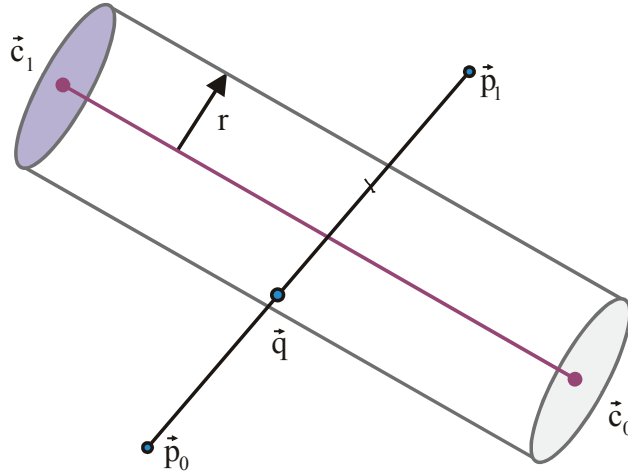
The value for  $t$  can be found by solving the following:

$$\|\vec{p}_0 + t\vec{d} - \vec{c}\|^2 = r^2, \quad (13-40)$$

which gives a quadratic equation with an easy solution. If the resulting value is either not real or not in the range of  $[0,1]$ , then the line segment does not intersect the sphere. There are often two solutions—the one with the smallest  $t$ -value is selected as the closest.

### 13.11.2 Cylinder

Parameters for a cylinder are shown below. These include a line segment and a radius.



**Figure 13-19:** Cylinder intersection. The cylinder is defined by a line segment, having endpoints  $\vec{c}_0$  and  $\vec{c}_1$ , and a radius.

As before, define

$$\vec{d} = \vec{p}_1 - \vec{p}_0. \quad (13-41)$$

Also define

$$\vec{b} = \vec{c}_1 - \vec{c}_0. \quad (13-42)$$

To calculate the intersection of a line segment and a sphere, let the closest intersection point,  $\vec{q}$ , be defined as

$$\vec{q} = \vec{p}_0 + t\vec{d}, \quad (13-43)$$

The distance,  $D$ , between  $\vec{q}$  and the line defined by the central axis of the cylinder satisfies

$$D^2 = \frac{\|\vec{b} \times (\vec{q} - \vec{c}_0)\|^2}{\|\vec{b}\|^2}. \quad (13-44)$$

Setting this distance to  $r$  gives the following:

$$r^2 \|\vec{b}\|^2 = \|\vec{b} \times (\vec{p}_0 + t\vec{d} - \vec{c}_0)\|^2, \quad (13-45)$$

or

$$r^2 \|\vec{b}\|^2 = \|\vec{b} \times (\vec{p}_0 - \vec{c}_0) + \vec{b} \times (t\vec{d})\|^2, \quad (13-46)$$

giving

$$\|(\vec{b} \times \vec{d})\|^2 t^2 + 2 * (\vec{b} \times (\vec{p}_0 - \vec{c}_0)) \cdot (\vec{b} \times \vec{d}) t + \|(\vec{b} \times (\vec{p}_0 - \vec{c}_0))\|^2 - r^2 \|\vec{b}\|^2 = 0. \quad (13-47)$$

This forms a quadratic equation with an easy solution. The result must be validated against the constraints that  $t$  must be real and it must be in the range  $[0,1]$ . It must also be validated against the limits of cylinder. When there are two results, the one with the smallest  $t$ -value is selected as the closest.

Then, a test must also be performed against the cylinder caps. This can be done by defining  $\vec{q}$  as above and solving the following for the first cap using

$$(\vec{p}_0 + t\vec{d} - \vec{c}_0) \cdot \vec{b} = 0, \quad (13-48)$$

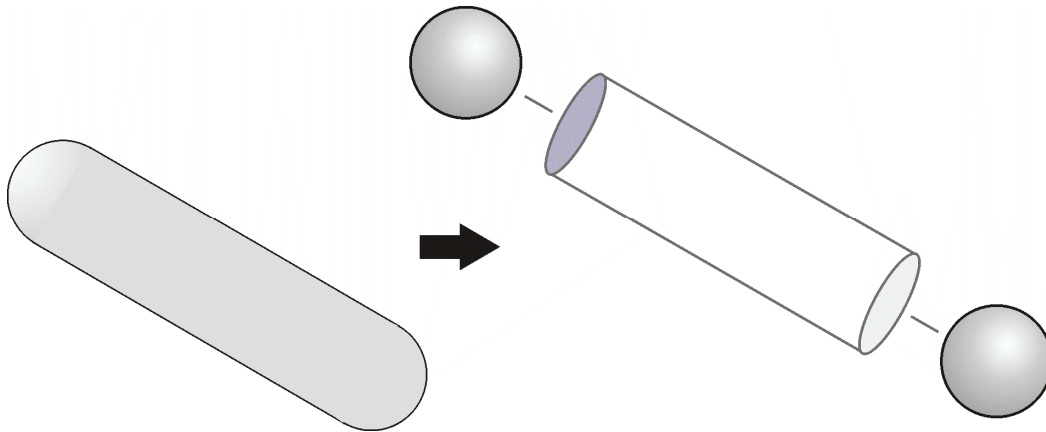
and solving for the second cap using

$$(\vec{p}_0 + t\vec{d} - \vec{c}_1) \cdot \vec{b} = 0, \quad (13-49)$$

The points so found must be tested against the constraint that  $t$  be contained in  $[0,1]$  and against the radius of the cylinder through  $\|\vec{p}_0 + t\vec{d} - \vec{c}_0\|^2 < r^2$  and  $\|\vec{p}_0 + t\vec{d} - \vec{c}_1\|^2 < r^2$ .

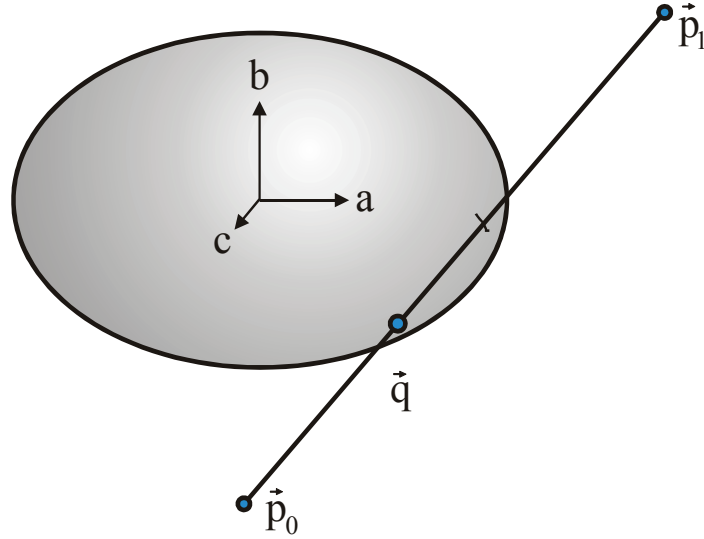
### 13.11.3 Capsule

To establish the intersection between a line segment and a capsule, the capsule is decomposed into a cylinder and two spheres, as shown below. Intersection tests are then performed with these decomposed parts, using the techniques discussed above.



**Figure 13-20:** Capsule intersection. The capsule is decomposed into a cylinder and two spheres, with each tested using the techniques described above.

### 13.11.4 Ellipsoid



**Figure 13-21:** Ellipsoid intersection.

An ellipsoid, as shown above, is represented through the following equation expressed in an explicit offset frame:

$$\frac{x^2}{a^2} + \frac{y^2}{b^2} + \frac{z^2}{c^2} = 1. \quad (13-50)$$

For this, the line segment used for intersection is first represented in the explicit offset frame. So, for this discussion, the all vector quantities are represented in the central frame of the ellipsoid, rather than world coordinates. With this, let

$$\vec{d} = \vec{p}_1 - \vec{p}_0. \quad (13-51)$$

And, as before, let the closes intersection point,  $\vec{q}$ , be defined as

$$\vec{q} = \vec{p}_0 + t\vec{d}, \quad (13-52)$$

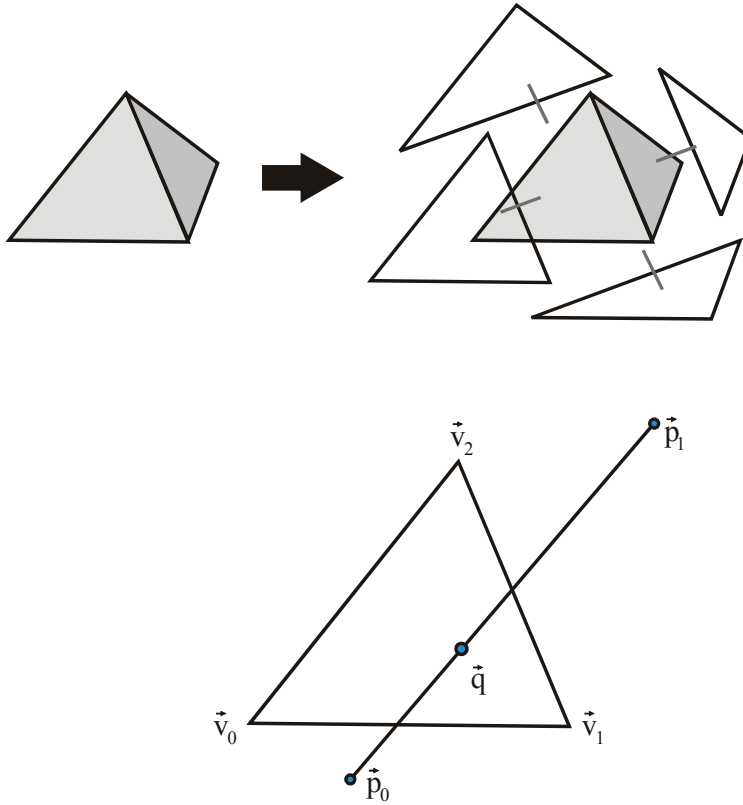
The value for t can be found by solving the following:

$$\frac{(p_{0,x} + t d_x)^2}{a^2} + \frac{(p_{0,y} + t d_y)^2}{b^2} + \frac{(p_{0,z} + t d_z)^2}{c^2} = 1, \quad (13-53)$$

which gives a quadratic equation with an easy solution. As before,  $t$  must be real, and it must be in the range  $[0,1]$ , and when there are two solutions, the one with the smallest t-value is selected as closest.

### 13.11.5 Tetrahedron

Intersection with a tetrahedron is evaluated by decomposing the tetrahedron into four 3D triangles, as illustrated below.



**Figure 13-22:** Tetrahedron intersection is performed using the four triangular faces.

As before, let

$$\vec{d} = \vec{p}_1 - \vec{p}_0. \quad (13-54)$$

And let the point where the line through the line segment intersects the plane of the triangle be defined through the following:

$$\vec{q} = \vec{p}_0 + t\vec{d}. \quad (13-55)$$

Define

$$\vec{e}_0 = \vec{v}_1 - \vec{v}_0, \quad (13-56)$$

$$\vec{e}_1 = \vec{v}_2 - \vec{v}_0, \quad (13-57)$$

and

$$\vec{e}_3 = \vec{v}_1 - \vec{v}_2. \quad (13-58)$$

Let the triangle normal be defined as



$$\vec{n} = \vec{e}_0 \times \vec{e}_1. \quad (13-59)$$

Assuming the triangle is not collinear,  $t$  can be found using

$$(\vec{p}_0 + t\vec{d} - \vec{v}_0) \cdot \vec{n} = 0. \quad (13-60)$$

This gives a linear equation that can easily be solved for  $t$  to give  $\vec{q}$ . The value of  $t$  must be in the range  $[0,1]$ .

To ascertain if  $\vec{q}$  is inside the triangle, the following three checks are made:

$$(\vec{q} - \vec{v}_0) \times \vec{e}_1 \cdot \vec{n} \geq 0 \quad (13-61)$$

$$(\vec{q} - \vec{v}_1) \times \vec{e}_0 \cdot \vec{n} \leq 0 \quad (13-62)$$

$$(\vec{q} - \vec{v}_2) \times \vec{e}_2 \cdot \vec{n} \geq 0 \quad (13-63)$$

These checks ensure sequentially that the point is on the correct side of each edge.

### 13.11.6 Half Space

A half space is defined as one side of a 3D plane. Its only surface is the plane defining it. The intersection parameters are shown in the figure below.

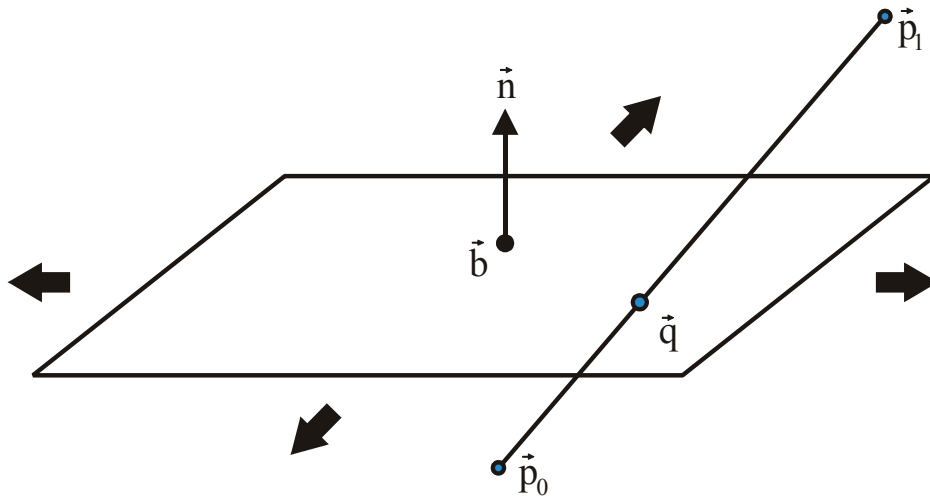


Figure 13-23: Half space intersection.

As before, let

$$\vec{d} = \vec{p}_1 - \vec{p}_0. \quad (13-64)$$

And let the point where the line through the line segment intersects the plane be defined through

$$\vec{q} = \vec{p}_0 + t\vec{d}. \quad (13-65)$$

Then at the intersection point,  $t$  satisfies

$$(\vec{p}_0 + t\vec{d} - \vec{b}) \cdot \vec{n} = 0, \quad (13-66)$$

giving a linear equation that is easily solved. The value of  $t$  must be in the range  $[0,1]$  for an intersection to have occurred.

### 13.11.7 Oriented Box

An oriented box is illustrated in the figure below.

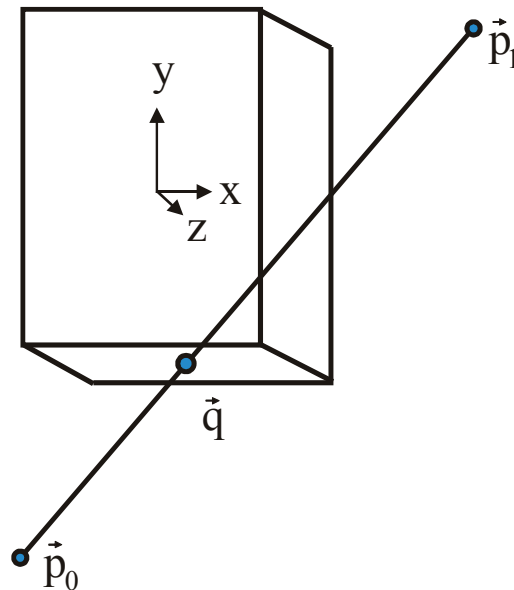


Figure 13-24: Oriented box intersection.

The oriented box, as shown is represented through the following bounds on the x-, y-, and z-components of 3D points, as represented in an explicit offset frame:

$$\begin{aligned} -a &\leq x \leq a, \\ -b &\leq y \leq b, \\ -c &\leq z \leq c. \end{aligned} \quad (13-67)$$

For this, the line segment used for intersection is first represented in the explicit offset frame. So, for this discussion, the all vector quantities are represented in the central frame of the box, rather than world coordinates. With this, let, as before,

$$\vec{d} = \vec{p}_1 - \vec{p}_0. \quad (13-68)$$

And, as with previous shapes, let the intersection point,  $\vec{q}$ , be defined as

$$\vec{q} = \vec{p}_0 + t\vec{d}, \quad (13-69)$$

The intersection points for the face planes perpendicular to each dimension can be calculated from

$$\begin{aligned} a &= p_{0x} + t_{0x}d_x \\ -a &= p_{1x} + t_{1x}d_x \end{aligned} \quad (13-70)$$

$$\begin{aligned} b &= p_{0y} + t_{0y}d_y \\ -b &= p_{1y} + t_{1y}d_y \end{aligned} \quad (13-71)$$

and

$$\begin{aligned} c &= p_{0z} + t_{0z}d_z \\ -c &= p_{1z} + t_{1z}d_z \end{aligned} \quad (13-72)$$

If any component of  $d$  is zero, a special case arises for that dimension. Otherwise, the  $t$ -values can be solved through the linear equations given. The result for each  $t$ -value takes the form of the following (shown only for the  $x$ -dimension):

$$\begin{aligned} t_{0x} &= (a - p_{0x})/d_x, t_{1x} = (-a - p_{1x})/d_x, & d_x &\neq 0 & (13-73) \\ \text{inclusive} & & d_x &= 0 \cap -a \leq p_{0x} \leq a \\ \text{exclusive} & & & \text{otherwise} \end{aligned}$$

Here, “inclusive” means all  $t$ -values are acceptable as it pertains to this dimension, and “exclusive” means no  $t$ -value is acceptable as it pertains to this dimension. The  $y$ - and  $z$ -components are treated similarly. Let

$$t_{\text{entry}} = \max(\min(t_{0x}, t_{1x}), \min(t_{0y}, t_{1y}), \min(t_{0z}, t_{1z})) \quad (13-74)$$

and

$$t_{\text{exit}} = \min(\max(t_{0x}, t_{1x}), \max(t_{0y}, t_{1y}), \max(t_{0z}, t_{1z})). \quad (13-75)$$

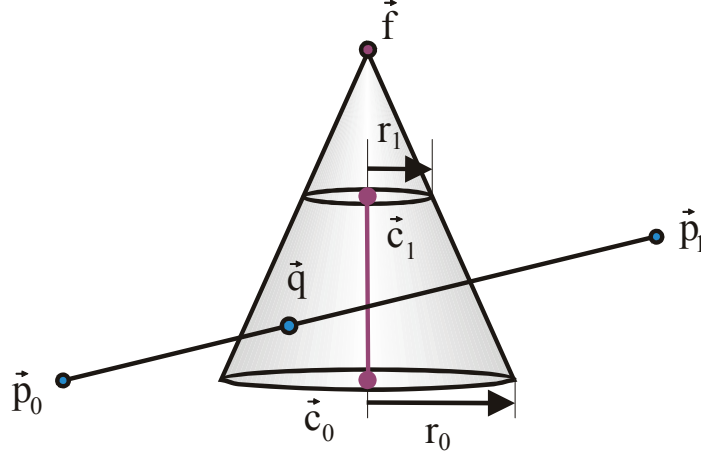
With these values, if

$$t_{\text{entry}} < t_{\text{exit}} \quad (13-76)$$

and  $0 \leq t_{\text{entry}} \leq 1$ , then  $t_{\text{entry}}$  gives the impact point. Otherwise, the line segment does not impact the box.

### 13.11.8 Cone Frustum

A cone frustum is the volume between two circular caps inside a conical surface. This is shown in the figure below.



**Figure 13-25:** Cone frustum intersection. The frustum lies between the two circular regions.

Given the figure above, define

$$\vec{d} = \vec{p}_1 - \vec{p}_0 \quad (13-77)$$

and

$$\vec{b} = \vec{c}_1 - \vec{c}_0. \quad (13-78)$$

As before, let the closest intersection point,  $\vec{q}$ , be defined as

$$\vec{q} = \vec{p}_0 + t\vec{d}, \quad (13-79)$$

If  $r_0 = r_1$ , the cone frustum can be treated as a cylinder. And if  $r_0 < r_1$ , the cone frustum can be flipped to give  $r_0 > r_1$ . With  $r_0 > r_1$ , the point  $\vec{f}$ , as shown in the figure, exists, and the point  $\vec{q}$  lies on the surface of the cone with point at  $\vec{f}$  if

$$\frac{-(\vec{q} - \vec{f}) \cdot \vec{b}}{\|\vec{q} - \vec{f}\| \|\vec{b}\|} = \frac{\|\vec{b}\|}{\sqrt{\vec{b} \cdot \vec{b} + (r_0 - r_1)^2}}. \quad (13-80)$$

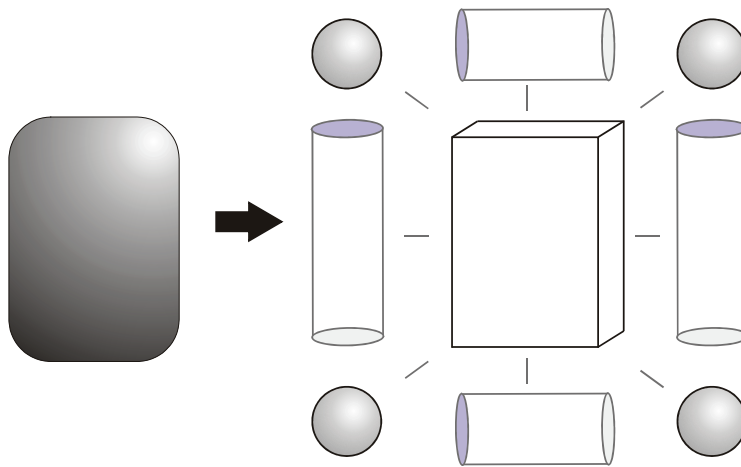
Squaring and substituting the value of  $\vec{q}$  gives the requirement in terms of  $t$  that the point lie on the double cone centered at  $\vec{f}$ :

$$((\vec{p}_0 + t\vec{d} - \vec{f}) \cdot \vec{b})^2 (\vec{b} \cdot \vec{b} + (r_0 - r_1)^2) = \|\vec{b}\|^4 \|\vec{p}_0 + t\vec{d} - \vec{f}\|^2. \quad (13-81)$$

This can be used to solve for  $\vec{q}$ , which can then be tested through an inner product with  $\vec{b}$  to see if it is inside the frustum. As always, the value of  $t$  must be in the range  $[0,1]$ .

### 13.11.9 Lozenge

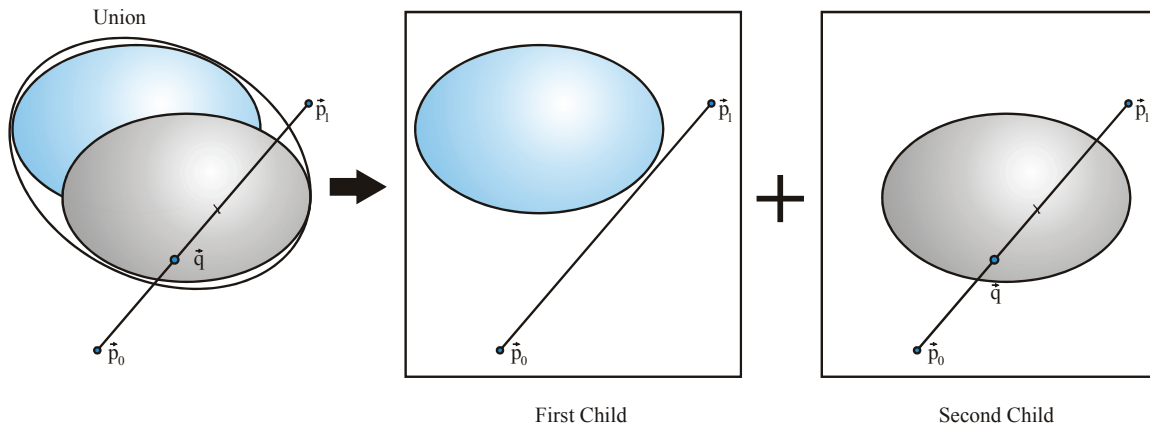
To establish the intersection between a line segment and a lozenge—which is defined as all points within a specified distance from a 3D rectangle—the lozenge is decomposed into an oriented box, four cylinders and four spheres, as shown below. Intersection tests are then performed with these decomposed parts, using the techniques discussed above (tests against the cylindrical caps are excluded).



**Figure 13-26:** Lozenge intersection. The lozenge is decomposed into 9 primitive shapes, each of which is tested for intersection using the techniques described above.

### 13.11.10 Union

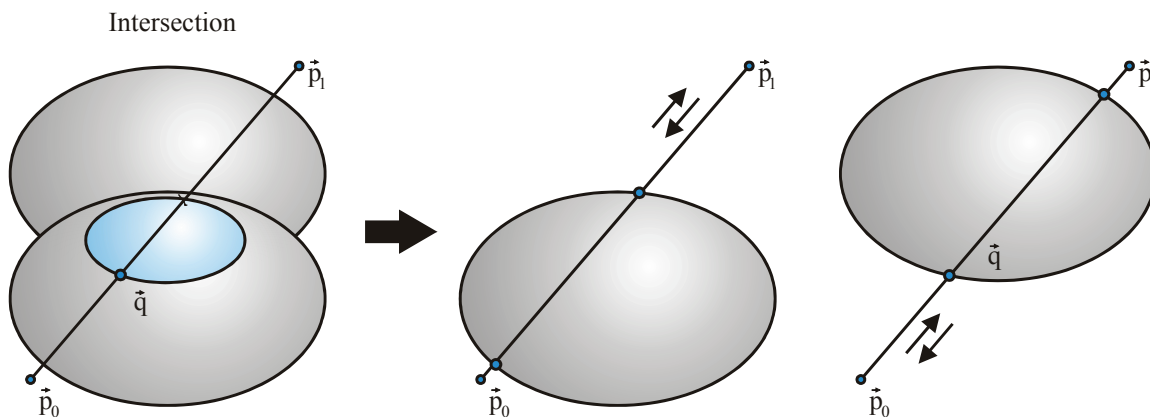
Energid's shape descriptions include combination types. One of these is the Union type, which has two child elements. To find the line-segment intersection of the Union, we just serially intersect with the two child shapes and take the closest point. This is illustrated in the figure below.



**Figure 13-27:** Union-of-shapes intersection. The intersection point is the closest intersection point for a child shape.

### 13.11.11 Intersection

Another combination shape is the Intersection. Finding the line-segment intersection with a shape Intersection is more involved than for a Union. The implementation assumes the child shapes are convex. With the convexity assumption, the intersection point, if it exists, must be the first intersection point of one of the two child shapes. Which (if either) of the two shapes give the intersection point is found by intersecting the line segment in both forward and reverse directions for both child shapes and reasoning with these results. The process is illustrated in the figure below.



**Figure 13-28:** Intersection-of-shapes intersection. The intersection point for the shape Intersection is first intersection for a child shape that is contained within the other shape.

## 14 Force Control and Grasping

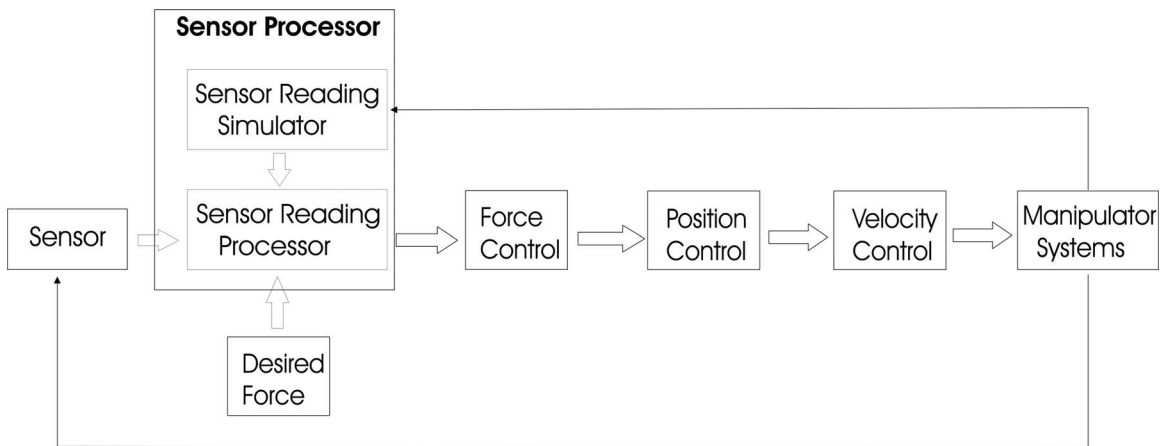
### 14.1 Force Control

This section describes force control, which is essential for most manipulation tasks such as grasping. Our approach leverages the surface-property descriptions and position control developed during the prior effort to build network-exchangeable force-control methods that are configured using XML.

#### 14.1.1 Design

Free-space operation will be used when the hand is away from the target object, using the local position control system that already exists in the toolkit. When the manipulator is in contact with the target object, compliance-based control is used. Compliance-based control is implemented using the position controller. Proportional-integral control is applied to set desired forces, and feed-forward compensation is added to the control law, with parameters based on the flexible surface properties of both the sensor attached to the end effector and the contacted surface.

**Figure 14-1** shows a flow chart of the force control system. The sensor processor is designed to work with both real hardware sensors and simulated sensors. The sensor reading simulator is used to estimate sensor readings during simulation based on proximity measures between the manipulator and the environment. The sensor reading processor can take the hardware sensor readings or the simulated sensor readings as inputs. The actual force that this sensor experienced is calculated from the sensor reading and compared against the desired force for that sensor. The output of this module is the difference between the desired force and the measured force and that is feed into the force control module.



**Figure 14-1:** Flow chart of the force control system.

A compliance-based approach is used for the force control module, which provides an especially robust and practical implementation. Force is controlled through the following proportional-plus-integral control law:

$$\mathbf{V}_F = \mathbf{D}(k_p(\mathbf{F}_d - \mathbf{F}_m) + k_i \int (\mathbf{F}_d - \mathbf{F}_m) dt), \quad (14-1)$$

where  $\mathbf{D}$  is a feed-forward compensator matrix,  $\mathbf{V}_F$  is the frame velocity formed by concatenating linear with angular velocity,  $\mathbf{F}_d$  is the desired general force,  $\mathbf{F}_m$  is the measured general force, and  $k_p$  and  $k_i$  are user-defined gains. For noisy force sensors, a greater weighting toward integral control will give smoother solutions, with less steady-state error. For more accurate force sensors, a heavier weighting toward position control will give a quicker response.

Note the velocity  $\mathbf{V}_F$  calculated in (14-1) is used to calculate a new desired position as the input to the position control module. This method will allow one end effector to operate with force control while another in free-space mode operates using local position control. With this approach, reconfiguration of the arm to optimize secondary criteria is also possible and easily implemented using our velocity-control algorithms. This will allow force to be controlled while reconfiguring the arm for optimal strength, for example. By allowing force control in self-contact, this approach will support force control in cooperating end effectors as well.

## 14.1.2 Implementation

### 14.1.2.1 Sensor Processor

The toolkit supports a variety of sensors for both simulation and hardware interface. A base class *EcBaseSensorProcessor* can be subclassed to make new sensors. The main method supported in this class is *computeForceDelta*. This takes the sensor reading from either the simulator or an actual sensor, calculates the proper force and compares against the desired force. Two types of sensor processors will be implemented and are discussed in detail as the following.

#### 14.1.2.1.1 Touch Sensor

A touch sensor processor *EcTouchSensorProcessor* has been added to the code base. The most rudimentary type of this kind of sensor is based on micro switches, which detect simple contact. For more sophisticated force control, a touch sensor with continuous reading is also modeled. This sensor is attached to a link, with a known location and direction specified with respect to the primary frame of the link (refer to Figure 14-2). The sensor is represented by a union of convex shapes as part of the link to which it is attached. Our proximity calculation routine is capable of reporting the distance query to the individual shape level.

With this approach, the interaction with parts of the manipulator other than the sensor will not be detected for force control purpose. Only when the sensor intersects the environment will there be a reading. The sensor reading is a function of the surface properties of the sensor and that of the object contacted. Assuming that the interaction between the sensor and the environment by a spring model can be approximated, with  $k_e$  and  $k_s$  being the spring constants of the environment and the sensor respectively, the penetration distance (minimum linear movement to separate two overlapping objects) is  $d$ , then the deformation of the sensor,  $d_s$ , is

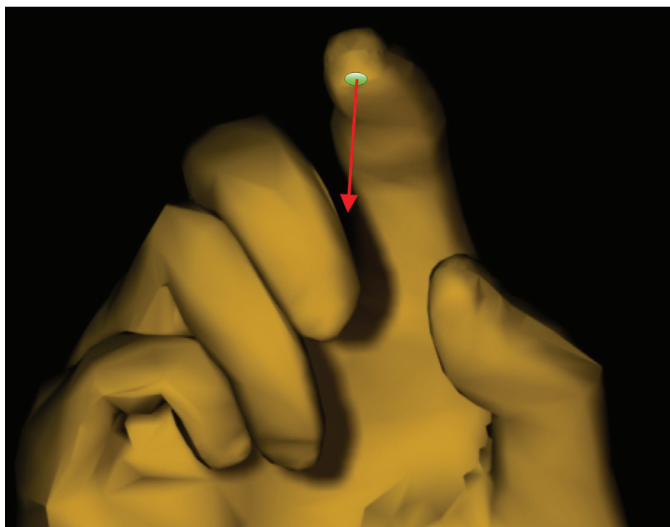
$$d_s = \frac{k_e d}{(k_e + k_s)}, \quad (14-2)$$

and the interactive force between the two is



$$\|\mathbf{f}_m\| = \|\mathbf{f}_e\| = \frac{k_e k_s d}{(k_e + k_s)}, \quad (14-3)$$

where  $\mathbf{f}_m$  is the force experienced by the sensor.



**Figure 14-2:** A touch sensor attached to one fingertip. The location and direction of the sensor is specified with respect to the primary frame of the finger.

The direction of the force is calculated from the specified sensor direction with respect to the primary frame of the attached link and the current system state information. The desired force is a general force with both a linear force  $\mathbf{f}_d$  and a moment  $\mathbf{n}_d$ , combined with a point of application. This covers the most general case. For the implementation of the touch sensor, only the magnitude of the force in the general force term is used. This is useful for the grasping task where the magnitude of the contact force is of interest. In this case, the force difference  $\mathbf{f}_d - \mathbf{f}_m$  is calculated from  $\|\mathbf{f}_d\| - \|\mathbf{f}_m\|$  and the sensor direction.

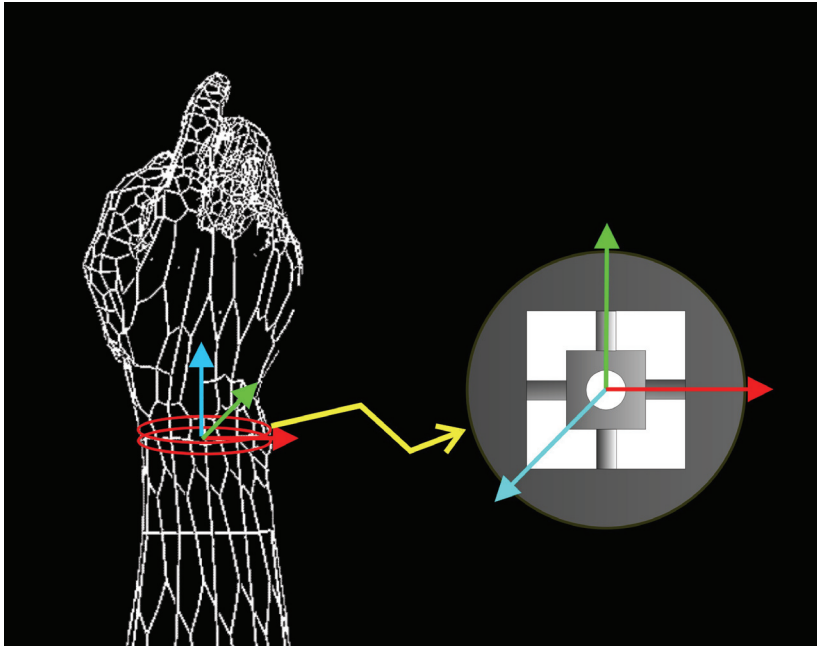
#### ***14.1.2.1.2 Wrist-Force Sensor***

The second kind of force sensor processor is for a wrist-force sensor as illustrated in Figure 14-3. This kind of sensor measures the force (three components) and moment (three components) exerted by the end effector on the environment. For simulation, the sensor readings are calculated from penetration distance calculated between the shapes used to construct the manipulator and the environment. Note that, unlike with the touch sensor, all the forces acting on the end effector need to be considered. Our system can calculate the penetration distance, direction of penetration, and location of contact between the physical extents within the system. For each collision event, the amount of force is calculated as in equation (14-3). The direction and the location of the force with respect to the primary frame of the associated link are provided by the simulated system. If the location of application and the direction of the force with respect to the sensor frame are given by  $\mathbf{p}$  and  $\mathbf{n}$ , then the measured force is given by

$$\mathbf{f}_m = \frac{k_e k_s d}{(k_e + k_s)} \mathbf{n}, \quad (14-4)$$

and the moment is given by

$$\mathbf{n}_m = \mathbf{f}_m \times \mathbf{p}. \quad (14-5)$$



**Figure 14-3:** A wrist force/torque sensor measures the composite of all forces applied to the outboard link.

If it is assumed that the contact surface of the collided shapes do not move relatively, then this would be another source of torque. Let the end effector and the environment each have an angular spring constant  $c_s$  and  $c_e$ , then the torque generated on the contact surface would be

$$\|\mathbf{n}_m\| = \|\mathbf{n}_e\| = \frac{c_e c_s}{(c_e + c_s)} \theta, \quad (14-6)$$

where  $\theta$  is the angular displacement deviates from the beginning of the contact. The force  $\mathbf{F}_m$  measured from this sensor is a combination of all the collision incidents using (14-4), (14-5), and (14-6).

The desired force  $\mathbf{F}_d$  for this case is a general force specified by six components. The force difference is given by  $\mathbf{F}_d - \mathbf{F}_m$  directly.

### 14.1.2.2 Force Control

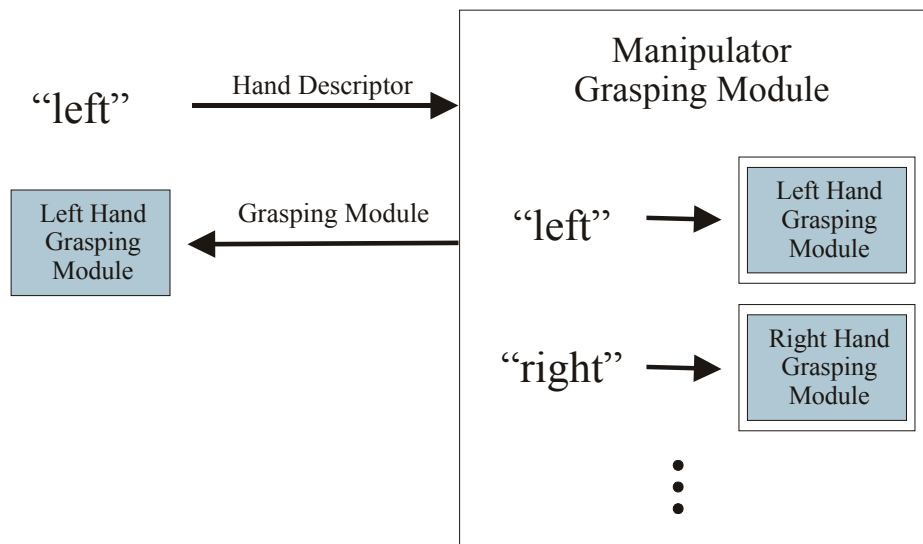
*EcForceControlSystem* contains all the sensors in the system, as well as the desired force for each sensor. It will take the force difference for each sensor and process it according to (14-1). Note that the velocity (linear and angular) calculated from (14-1) is for the sensor. The corresponding velocity for the end effector is also needed. The force control system builds on the position control system. When there is no interaction between the manipulator and the environment, the system is driven directly by the position control system. Once the manipulator is in contact with the target object, the force control system becomes active. It will first calculate the end effector velocity by using equation (14-1) and the transformation between the sensor and the end effector. Then it will calculate the new desired position for the end effector and provide that to the position control system.

## 14.2 Grasping

The Actin™ toolkit provides tools for implementing the kinematic and force components of grasping. It combines innovative algorithms linked by the central theme that a decision tree is used to find the best combination of algorithms for each grasp. The decision tree is configurable through XML and uses basic properties of the object to be grasped to quickly identify the best grasping method. Within this tree, prototype algorithms use precalculated initial grasps that can be hand-tuned for shape families. Within the framework, these initial grasps can be iteratively improved using properties of the object.

### 14.2.1 Interface

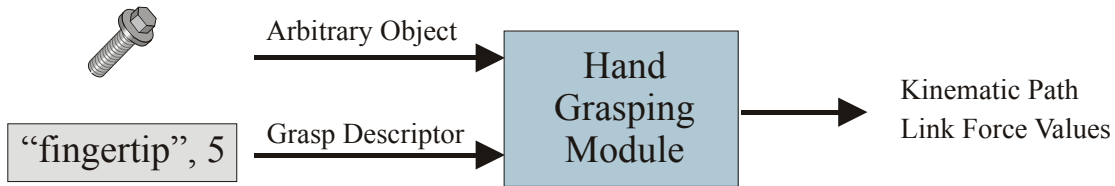
Since a manipulator can have multiple hands, the manipulator grasping module, class *EcManipulatorGraspingModule*, is organized into a collection of grasping modules, each for a single hand. This collection is accessed using a string descriptor of the hand (such as “left” or “right”) through a string-based map. The manipulator grasping module is illustrated in the figure below.



**Figure 14-4:** Multiple hand grasping modules will be organized into a manipulator grasping module. The input to the grasping module is a hand identifier (such as “left” or “right”). A separate manipulator is used for each manipulator in the system.

The single-hand modules shown in Figure 14-4 are subclassed from *EcBaseHandGraspingModule*. Each calculates 1) a spatial path for the grasping surfaces (such as fingertips) to position the hand for

grasping and 2) a set of forces for the grasping surfaces to consummate the grasp. The input to the module is an arbitrary object, described using the Energid CAD format. The object description may include information on the environment of the object. The hand grasping module is illustrated in the figure below.



**Figure 14-5:** The input to each grasping module (as shown in Figure 14-4) includes an object to be grasped, a string describing the grasp family, and an index identifying a unique grasp. The output is 1) a kinematic end-effector path to place the hand in contact with the object without detrimentally disturbing it and 2) a set of link forces to consummate the grasp. Note that, in this approach, knowledge of the hand is part of the hand grasping module.

The hand-grasping module takes as input the object to be grasped and a grasp descriptor. The descriptor has a string grasp-family identifier and an integer index. Each unique index returns a different grasp. Each hand grasping module will first calculate the grasp, then provide information on the grasp as a function of a parameter,  $\gamma$ , that varies from zero to one. When  $\gamma = 0$ , the hand is placed at the start of the grasping trajectory. When  $\gamma = 1$ , it is placed at the end of the grasping trajectory, just in contact with the object to be grasped.

In code, the interface to the hand grasping module will be through the following methods:

Methods	Description
<pre>void calculateGrasp (   const EcGraspDescriptor&amp; descriptor,   const EcStatedSystem&amp; objectStatedsystem,   EcU32 objectIndex,   const EcStatedSystem&amp; manipStatedSystem,   EcU32 manipIndex );</pre>	This calculates the grasp and saves it internally for access.
<pre>const EcEndEffectorSet&amp; endEffectorSet (   EcReal gamma,   EcBoolean&amp; isNew );</pre>	Returns the end effectors to be used for the grasp motion, parameterized by $\gamma$ . Sets the isNew flag if the end-effector set is new.
<pre>const EcCoordinateSystemTransformation... Vector&amp; endEffectorPositions (   EcReal gamma );</pre>	Returns the end-effector positions to be used for the grasp motion, parameterized by $\gamma$ .
<pre>const EcEndEffectorSet&amp; endEffectorSetForForceControl ( );</pre>	Returns an end-effector set to be used during force control. This allows hybrid force and positioning control to be used together for the grasp.
<pre>const EcManipulatorEndEffectorPlacement&amp; endEffectorPositionsForForceControl</pre>	Returns the end-effector positions to be

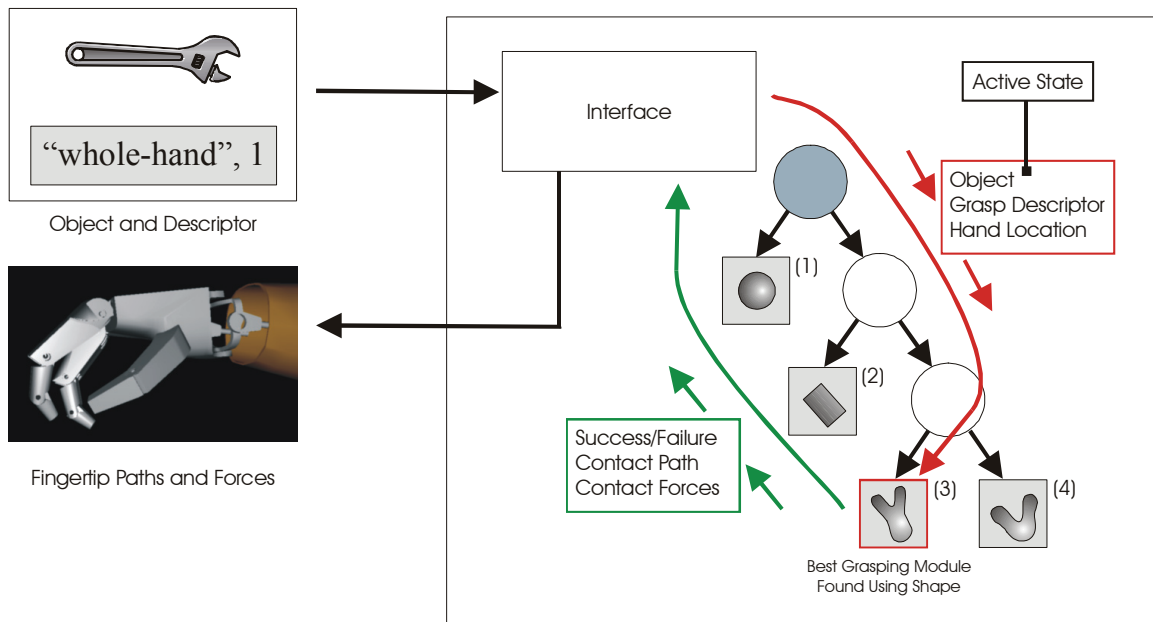
( );	used during force control.
void getGraspForces ( EcGraspForce& graspForce );	Returns the links and desired forces required for the grasp. These forces are applied after the hand is in place (i.e., when $\gamma = 1$ ).

**Table 14-1:** The class interface for the hand grasping module.

### 14.2.2 Decision Tree

The Actin™ grasping approach embraces the variety of grasping algorithms through the use of a configurable decision tree that can grow and change. Branching nodes in the tree will redirect program control to the best algorithm for the shape of the object to be grasped. This tree, which is configured using XML, will allow the addition of new algorithms. It can grow to have hundreds or thousands of algorithms, each tailored to grasping one family of shapes. The decision tree is illustrated in Figure 14-6.

The description of the object to be grasped includes the object geometry, location, and environment. To represent this information, the same data structure is used as represents the robotic manipulators and environment for the simulation and control capability. A pointer to an *EcStatedSystem* object is passed down the tree.

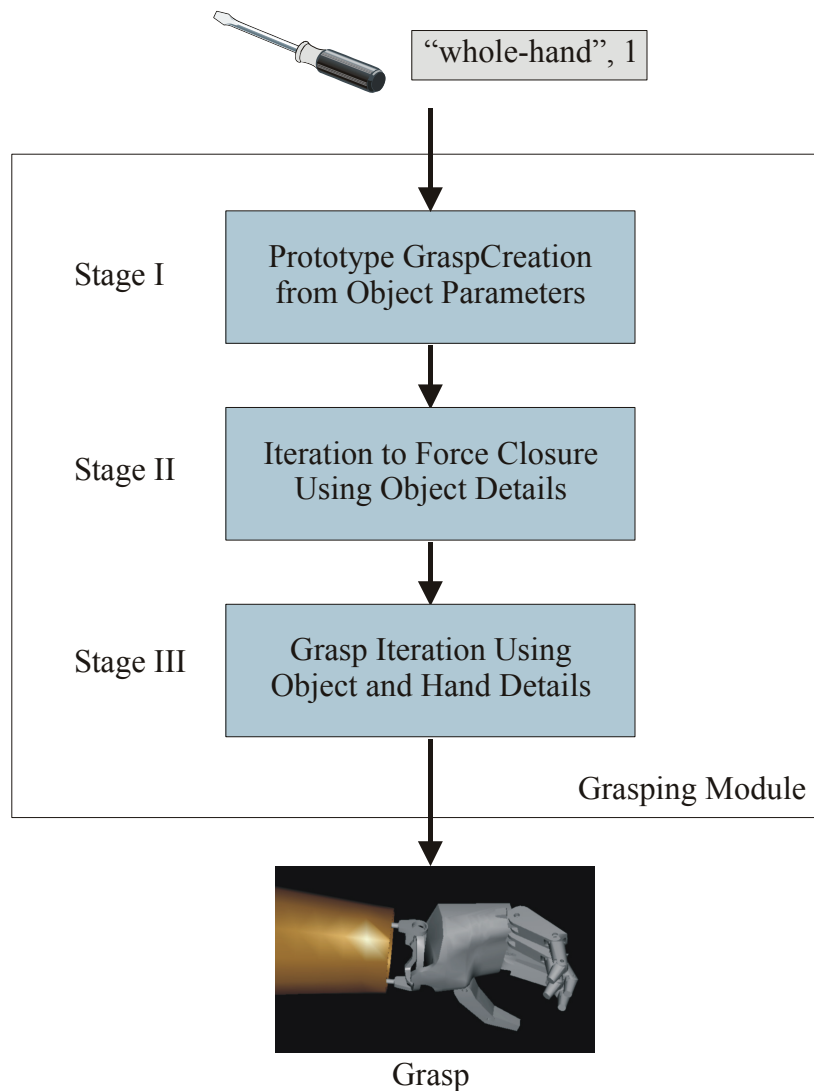


**Figure 14-6:** The best grasping algorithm to implement the grasping module in Figure 14-5 is found using a decision tree that matches the shape and surface properties of the object to be grasped. Each leaf node in the tree is an algorithm, whose implementation is flexible (limited only by the interface structure and C++). As shown, algorithm (1) would apply for sphere-like objects, (2) would apply for cylinder-like objects, and (3) and (4) would apply for different families of forking objects. The object to be grasped is not generally a pure shape, but a best match is found in the tree. A decision tree in this form provides potential for unlimited growth—new algorithms for new shapes can be

added without disturbing existing algorithms. The active state will be used for dynamic programming—object properties will only be calculated once.

### 14.2.3 Organization of Each Grasping Algorithm

Through the toolkit approach, virtually any algorithm can be used within the decision tree framework shown in Figure 14-6. One specific type of algorithm exists in the toolkit. The approach is based on refinement of a pre-established grasp. Each pre-established grasp is tailored to the matching shape found in the decision tree, and objects resembling that shape will be graspable through the refinement process. Our emphasis in this design is to establish a framework that can be used to support a variety of methods that will be successful with a broad spectrum of grasping scenarios. This framework, which uses three stages, is illustrated in the figure below. The input to Stage I is the object to be grasped and a descriptor. The output of Stage I and both the input and output of Stages II and III are grasps.



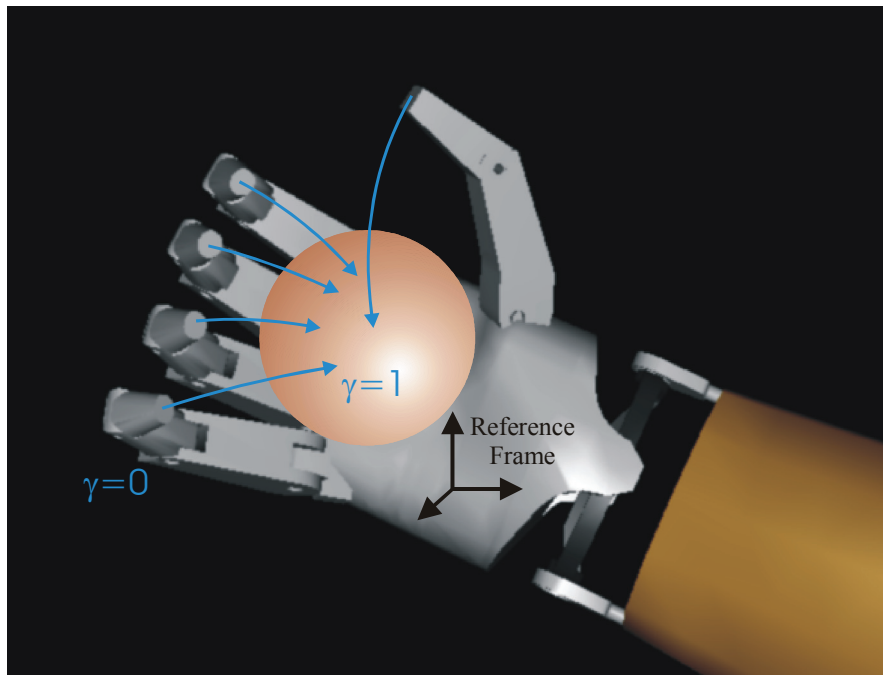
**Figure 14-7:** Prototype algorithms for use within the tree shown in Figure 14-6 are implemented using a three-stage process. First, a prototype grasp is created from object parameters. This grasp may be precalculated, captured from real grasps, created offline through simulation, or created

manually through human reasoning. This initial grasp can be refined to provide a force-closure solution using the details of the object to be grasped, including shape and surface properties. Finally, the model of the hand can be included to refine the solution through simulation.

#### 14.2.4 *Prototype Grasp Creation*

Stage I in Figure 14-7 creates a prototype grasp that may be prerecorded, captured from a human grasp, or calculated using an algorithm. This module solves the hardest grasping subproblem, which is global in nature: how, roughly, should hand be moved and forces applied given the entire space of all possible paths and forces. By tailoring this stage to each shape family, human intelligence can be used to solve the most difficult aspects of grasping.

The grasping execution will have two stages, a kinematic positioning stage and a force application stage. For kinematic positioning, end-effector paths will be parameterized by a value that varies from zero to one. This will take the hand from a completely open position to a completely closed position around the object to be grasped. This is illustrated in the figure below. It applies both to whole-hand and fingertip grasps. Note the paths are defined using end effectors rather than joints. This allows the end effectors to be placed using the position controller, exploiting all the tools (such as collision avoidance) that it provides.



**Figure 14-8:** The prototype grasp will be defined through a set of end-effector paths parameterized by  $\gamma \in [0,1]$ . When  $\gamma = 0$ , the locations of the end effectors are at the start of the grasp, giving an open hand, and when  $\gamma = 1$ , they are at the end of the grasp, closed, ready to apply force. The end-effector paths may be represented in any reference frame—in the system coordinate frame or in the primary frame of another link on the manipulator (such as the palm).

After the fingers are placed, forces are applied using the force control module. The forces are represented as point forces on the links contacting the grasped object.

## 15 Data Capture

### 15.1 Path Saving and Following

The toolkit contains a simulation that enables a user to create arm trajectories that support complex tasks. These trajectories can be saved and replayed in powerful ways to support tasks such as controlling remote manipulators. There are two path saving and replaying techniques: 1) state path and 2) guide frame path.

#### 15.1.1 State Path

The state path saving and replaying approach is contained in the *EcStatePath* class. This approach stores all of the manipulator joint angles and makes them available for playback. Table 15-1 shows the methods available for this capability. *EcStatePath* subclasses from *EcXmlCompoundType* which enables the user to read and write the recorded data to and from an XML file.

Method	Description
add	Records the state for the current time.
getState	Retrieves state for the current time.
reset	Clears the record.
getSize	Returns the size of the record.
minTimeBetweenSample	Get the minimum storage rate.
setMinTimeBetweenSample	Set the minimum storage rate.

**Table 15-1:** Listing of primary methods available to the developer for the path recording and playback using the state method. Check the code documentation for a complete list and description.

Text Box 15-1 shows code for setting up a trajectory, Text Box 15-2 shows an example for recording a trajectory, and Text Box 15-3 shows an example for playback.



```

// Position control system needed for test
EcPositionControlSystem positionControl =
    EcPositionControlSystem::testObject();

// Position controller needs pointer to stated system
EcStatedSystem sSystem=EcStatedSystem::testObject();
positionControl.setStatedSystem(&sSystem);

// create storage for dynamic state and path saving
EcManipulatorSystemState dynamicState;
EcStatePath statePath;
EcGuideFramePath guideFramePath;

// instantiate a renderer
EcRenderWindow renderer;

// set the size of the window
const EcU32 size = 320;
renderer.setWindowSize(2*size, size);

// create a visualizable stated system object for rendering
EcVisualizableStatedSystem visStatedSystem;
visStatedSystem.setStatedSystem(sSystem);
visStatedSystem.setVisualizationParameters
    (EcVisualizationParameters::testObject());
renderer.setVisualizableStatedSystem(visStatedSystem);

// get the current offset in system coordinates
const EcU32 manipIndex = 0;
const EcU32 endEffectorIndex = 0;

EcCoordinateSystemTransformation initialPose =
    positionControl.actualPlacement(manipIndex,endEffectorIndex);

// create circular path around the initial position
EcCoordinateSystemTransformation finalPose=initialPose;

// execution parameters
EcU32 steps=500;
EcReal simRunTime = 5.0;
EcReal simTimeStep = simRunTime/steps;
EcReal radius=0.3;
EcU32 loops=3;
EcOrientation orient(0,0,0,1);
EcReal startingTime=positionControl.time();

```

**Text Box 15-1:** Example code for setting up a trajectory. This is example section #1 in the path following example code.

```

// move to the desired pose, and render the position every time step
for (EcU32 ii=0;ii<steps;++ii)
{
    // get the current time
    EcReal currentTime=simTimeStep*ii;

    // set the pose for manipulator 0 and end effector 0
    EcCoordinateSystemTransformation pose;
    pose.setOrientation(orient);
    EcReal angle=Ec2Pi*loops*currentTime/simRunTime;
    EcVector offset=radius*EcVector(cos(angle),sin(angle),0);
    pose.setTranslation(finalPose.translation()+offset);
    positionControl.setDesiredPlacement(manipIndex,endEffectorIndex,pose);

    // calculate the state at current time
    positionControl.calculateState(currentTime+startingTime,dynamicState);

    // store state and guide frame path
    statePath.add(dynamicState);
    guideFramePath.add
        (positionControl.desiredPlacementVector(), currentTime+startingTime);

    // set state for rendering
    visStatedSystem.setState(dynamicState);
    renderer.setState(dynamicState);

    // view the system
    renderer.renderScene();
}

```

**Text Box 15-2:** Example code for creating and recording a trajectory. This is section #2 in the path following example code.

The state path is recorded using the *add* method. The *add* method takes an *EcManipulatorSystemState* reference and stores the state only if a minimum time since the last sample has passed. Note that the time stamp for the state is contained within the state.

```

for(ii=0;ii<steps;++ii)
{
    // get the current time
    EcReal currentTime=simTimeStep*ii;

    // get the stored state for the current time
    statePath.getState(currentTime+startingTime,dynamicState);

    // set state for rendering
    visStatedSystem.setState(dynamicState);
    renderer.setState(dynamicState);

    // view the system
    renderer.renderScene();
}

```

**Text Box 15-3:** Example code for replaying a trajectory using the state. This is section #3 in the path following example code.

The state path is replayed using the *getState* method. The *getState* method takes three arguments:

- 1) Time – input
- 2) State – output
- 3) Interpolation method – input. Currently, linear interpolation is the only option, which is the default.

To conserve the stored data size, the minimum time between samples is defaulted to 0.1 seconds, and can be overridden with *setMinTimeBetweenSample*. Each record is stored with a time stamp. During play back, the nearest two samples to the system clock are selected and the data are interpolated. The data contains a translation and orientation component. For translation, linear interpolation is applied, which yields a constant translation speed between the two samples. For orientation, interpolation between two quaternions is applied such that a constant angular velocity is maintained between the two samples. This approach will lead to continuity in position but not in velocity and acceleration.

### 15.1.2 Guide Frame Path

The guide frame path saving and replaying approach is contained in the *EcGuideFramePath* class. The approach stores all of the end effector placement data and makes them available for playback. Table 15-2 shows the methods available for this capability. *EcGuideFramePath* subclasses from *EcXmlCompoundType* which enables the user to read and write the recorded data to and from an XML file.

Method	Description
add	Records the state for the current time.
getFrameVec	Retrieves end effector placement for the current time.
reset	Clears the record.
getSize	Returns the size of the record.
minTimeBetweenSample	Get the minimum storage rate.
setMinTimeBetweenSample	Set the minimum storage rate.

**Table 15-2:** Listing of primary methods available to the developer for the path recording and playback using the guide frame method. Check the code documentation for a complete list and description.

Text Box 15-2 shows an example for recording a trajectory and Text Box 15-4 shows an example for playback.

The guide frame path is recorded using the *add* method. The *add* method takes two parameters: *EcManipulatorEndEffectorPlacementVector* and time, and stores the end effector data only if a minimum time since the last sample has passed.

```

// reset the state for next test
sSystem=EcStatedSystem::testObject();
positionControl.setStatedSystem(&sSystem);

// create a placement vector for retrieving from storage
EcManipulatorEndEffectorPlacementVector placementVector;

for(ii=0;ii<steps;++ii)
{
    // get the current time
    EcReal currentTime=simTimeStep*ii;

    // get the stored placement vector for the current time
    guideFramePath.getFrameVec(currentTime+startingTime,placementVector);

    // set the placement command and calculate the state
    positionControl.setDesiredPlacementVector(placementVector);
    positionControl.calculateState(currentTime+startingTime,dynamicState);

    // set state for rendering
    visStatedSystem.setState(dynamicState);
    renderer.setState(dynamicState);

    // view the system
    renderer.renderScene();
}

```

**Text Box 15-4:** Example code for replaying a trajectory using the end effector placements. This code is captured in the path following example code.

The guide frame path is replayed using the *getFrameVec* method. The *getFrameVec* method takes three arguments:

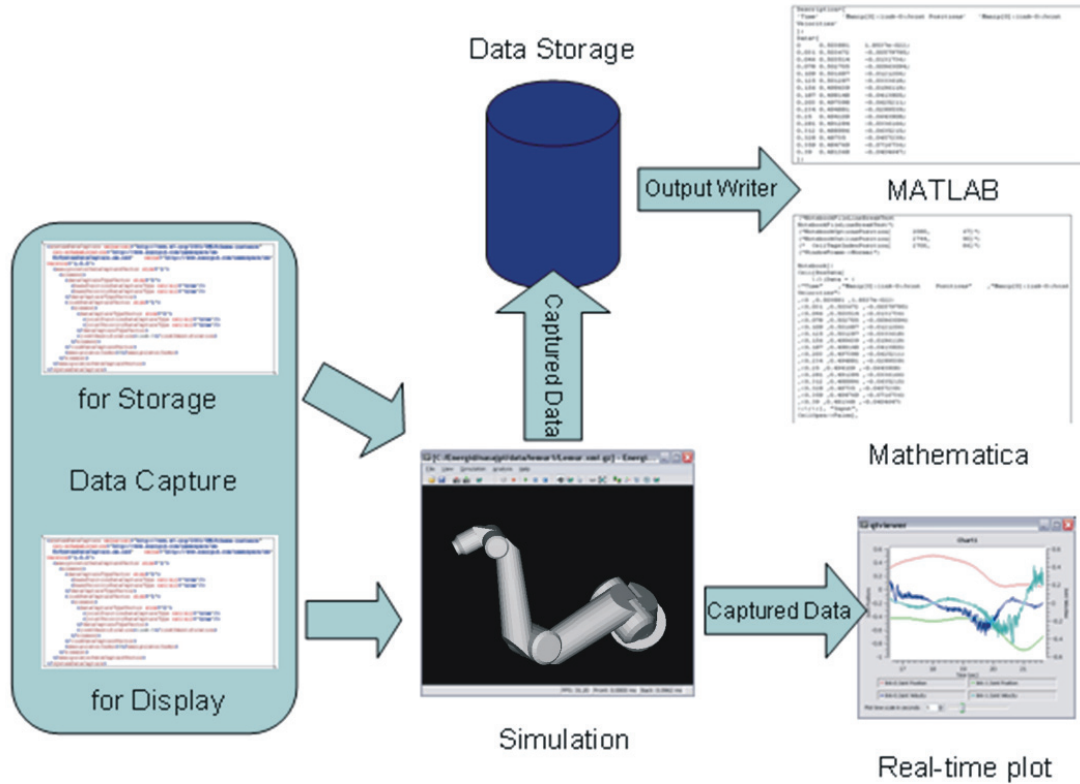
- 1) Time – input
- 2) End effector placement – output
- 3) Interpolation method – input. Currently, linear interpolation and nearest neighbor are available options. Linear interpolation is the default.

## 15.2 Storage and Display of Simulation Data

It is not unusual that the user of ActinViewer wants to be able to store the simulation data or display it in a plot. ActinViewer provides support that allows the user to do both of those tasks easily and efficiently. This section describes the back-end infrastructure required to support capturing of simulation data and shows example of how to programmatically configure the data capture.

### 15.2.1 Design of Data Capture

In our design, there are three main components in capturing and saving simulation information (excluding displaying it in a plot). The first is a collection of data capture classes systematically organized to capture the every bit of information that the user desires for the whole system. The information captured by these data capture classes can then be stored in a data storage class, ready to be saved to files. Finally, the output writer classes save the information stored in the storage to files in the formats chosen by the user. Figure 15-1 shows the schematic of the process of storing simulation data.



**Figure 15-1:** Schematic of data capture, data storage, and output writers.

A new library called “SimulationAnalysis” was created to contain all the implementations related to data capturing.

### 15.2.1.1 Data Component

*EcBaseSystemDataComponent* is an abstract base class for *EcSystemDataCapture* and *EcSystemDataStorage*. *EcBaseSystemDataComponent* itself is derived from *EcXmlCompoundType* so it can be read from and written to XML format. *EcSystemDataCapture* provides the descriptions and *EcSystemDataStorage* the numeric values of the captured data. These two classes will be discussed in detail in the following sections. *EcSystemDataVector* is an XML vector container for instances of classes that are derived from *EcBaseSystemDataComponent*. When instances of *EcSystemDataCapture* and *EcSystemDataStorage* are added to the same instance of *EcSystemDataVector*, the descriptions and the numeric values of the captured data can be written to a file in the same manner. All the classes derived from *EcBaseSystemDataComponent* must implement the methods listed in the table below

Method	Description
--------	-------------

type	Returns either GENERAL_CAPTURE for descriptions or GENERAL_STORAGE for numeric values.
getStringElements	Returns a vector of strings that contains either the descriptions or the numeric values of the captured data.

**Table 15-3:** Methods in EcBaseSystemDataComponent.

### 15.2.1.2 Data Capture

Two main classes for configuring what information to be captured are *EcSystemDataCapture* and *EcDataCaptureType*. In a nutshell, *EcSystemDataCapture* is simply a collection of instances of *EcDataCaptureType* (which performs the actual data capturing action) organized in such a way to replicate how manipulators and links are constructed in a system. Through *EcSystemDataCapture*, the user can add or remove a data capture type (an instance of *EcDataCaptureType*) to or from any manipulator or link. A data capture type can be either manipulator-level, link-level, or end-effector level. Examples of manipulator-level data types are the base position and base velocity of the manipulator. Link-level data types include joint positions and link forces. End-effector placements are examples end-effector level data types. Some of the important methods of *EcSystemDataCapture* are listed in the table below.

Method	Description
addEndEffectorDataCaptureType	Adds an end-effector data capture type.
addLinkDataCaptureType	Adds a link-level data capture type.
addManipulatorDataCaptureType	Adds a manipulator-level data capture type.
captureData	Captures all the data configured by the data capture types that are included.
isDataCaptureTypeIncluded	Returns true if the given data capture type is included or false otherwise.
removeEndEffectorDataCaptureType	Removes an end-effector level data capture type.
removeLinkDataCaptureType	Removes a link-level data capture type.
removeManipulatorDataCaptureType	Removes a manipulator-level data capture type.
label	Returns the label primarily used for displaying purposes.
setLabel	Sets the label.
allocateStorage	Allocates appropriate amount of memory for the data storage.

storeData	Stores the captured data in the data storage.
getStringElements	Returns a vector of strings that contain the descriptions of captured data elements.
type	Returns GENERAL_CAPTURE to indicate that this component is a data capture.

**Table 15-4:** Methods in *EcSystemDataCapture*.

*EcDataCaptureType* itself is an abstract base class that provides a common interface for all data capture types. Classes for capturing a specific piece of information, e.g. joint position or joint torque, must be derived from *EcDataCaptureType* and implement the methods described in the table below.

Method	Description
captureData	Captures the specific simulation data.
Data	Returns the data captured in captureData.
dataSize	Returns size of the data, e.g. a joint position has a size of 1 while a force has a size of 6 (3 for linear force and 3 for moment).
description	Returns the description of the data capture type, e.g. “Joint Position,” for displaying purposes.
disableFlags	Returns the disable flags, primarily used for data capture types with size greater than 1. The disable flags notify which data elements should be ignored.
setDisableFlags	Sets the disable flags. See disableFlags above.
type	Returns an enumeration that indicates if the data capture type is of manipulator-level, link-level, or end-effector level data type. The enumerations are MANIPULATOR_DATA_TYPE, LINK_DATA_TYPE, and END_EFFECTOR_DATA_TYPE.
label	Returns the description of each element of the captured data. For example, the first element of base position is “Translation X.”
token	Returns a unique token of this class. It is used in many situations, e.g. to create a new instance of this class or to identify an instance of this class.

**Table 15-5:** Methods in *EcDataCaptureType*.

Currently, the concrete classes in Table 15-6 have been implemented to capture several types of useful information. Other data capture types can be added by deriving from *EcDataCaptureType*.

<b>Class</b>	<b>Description</b>
<i>EcBaseAccelerationDataCaptureType</i>	Captures base acceleration of a manipulator. This is a 6x1 vector.
<i>EcBasePositionDataCaptureType</i>	Captures base position of a manipulator. This is a 7x1 vector.
<i>EcBaseVelocityDataCaptureType</i>	Captures base velocity of a manipulator. This is a 6x1 vector.
<i>EcControlTorqueDataCaptureType</i>	Captures control torque in each link of a manipulator. This is a scalar.
<i>EcEndEffectorAccelerationDataCaptureType</i>	Captures actual acceleration of an end-effector. Only valid in dynamic simulation.
<i>EcEndEffectorPlacementDataCaptureType</i>	Captures actual placement of an end-effector.
<i>EcEndEffectorVelocityDataCaptureType</i>	Captures actual velocity of an end-effector.
<i>EcExternalForceDataCaptureType</i>	Captures external force applied to each link including the base of a manipulator. This is a 6x1 vector.
<i>EcJointAccelerationDataCaptureType</i>	Captures joint acceleration in each link of a manipulator. This is a scalar.
<i>EcJointPositionDataCaptureType</i>	Captures joint position in each link of a manipulator. This is a scalar.
<i>EcJointTorqueDataCaptureType</i>	Captures joint torque in each link of a manipulator. This is a scalar.
<i>EcJointVelocityDataCaptureType</i>	Captures joint velocity in each link of a manipulator. This is a scalar.
<i>EcReachTargetDataCaptureType</i>	Captures whether or not an end-effector successfully reaches the desired location.
<i>EcStructuralForceDataCaptureType</i>	Captures external force applied to each link including the base of a manipulator. This is a 6x1 vector.

**Table 15-6:** Currently implemented data capture types.



*EcDisplaySystemDataCapture* is a convenient class that encapsulates a string and two instances of *EcSystemDataCapture* and is used primarily for displaying (plotting) purposes. It is designed to support two-Y-axes data plots. The string is used for the plot title and the two instances of *EcSystemDtaCapture* are used for the left and right Y-axes of the plot.

Method	Description
title	Returns the title of the plot.
setTitle	Sets the title of the plot.
leftSystemDataCapture	Returns the system data capture for the left Y-axis.
setLeftSystemDataCapture	Sets the system data capture for the left Y-axis.
leftSystemDataCapture	Returns the system data capture for the left Y-axis.
setLeftSystemDataCapture	Sets the system data capture for the left Y-axis.

**Table 15-7:** Methods in *EcDisplaySystemDataCapture*.

### 15.2.1.3 Data Storage

The data storage is where the captured data are stored before being written to files. The class *EcSystemDataStorage* contains all the captured data in a collection of real numbers and the simulation time stamp at the moment the data are captured. A vector of *EcSystemDataStorage* instances then represents a time history of the captured data. Table 15-8 lists the methods of *EcSystemDataStorage*. Note, however, that the user will generally not have to deal directly with *EcSystemDataStorage*.

Method	Description
manipualtorDataVector	Returns the manipulator data vector
setManipulatorDataVector	Sets the manipulator data vector
setSpecificManipulatorLevelData	Sets specific manipulator-level dta
setSpecificLinkLevelData	Sets specific link-level dta
setSpecificEndEffectorLevelData	Sets specific end-effector dta
time	Returns simulation time
setTime	Sets simulation time
pathPointIndex	Returns the index of the point in the path
setPathPointIndex	Sets the index of the point in the path

getStringElements	Returns a vector of strings that contain the data (converted from numbers) in this storage. Used for saving data to file.
type	Returns GENERAL_STORAGE to indicate that this component is a data storage.

**Table 15-8:** Methods in *EcSystemDataStorage*.

### 15.2.1.4 Output Writer

Once the data are stored in the data storage, it can later be written to files in different formats through classes that are derived from *EcBaseOutputWriter*. *EcBaseOutputWriter* is an abstract base class with the pure virtual methods listed in Table 15-9. Like the data capture type, the user can easily add an output writer that writes to a new file format by just deriving a class from *EcBaseOutputWriter*. Currently, four file formats – namely Mathematica, MATLAB, comma-delimited text, and XML – are supported.

Method	Description
fileExtension	Returns the file extension of the file format, e.g. “.m” for MATLAB.
formatDescription	Returns a description of the format, e.g. “MATLAB”, for displaying purposes.
initialize	Performs any necessary action before writing data to file.
finalize	Performs any necessary action after writing data to file.
token	Returns a unique token of this class. It is used in many situations, e.g. to create a new instance of this class or to identify an instance of this class.
writeComponentOpening	Writes any necessary element before writing a data component to file.
writeDataComponent	Writes a data component stored in an <i>EcSystemDataStorage</i> object to file.
writeComponentClosing	Writes any necessary element after writing a data component to file.

**Table 15-9:** Methods in *EcBaseOutputWriter*.

Class	Description
<i>EcMathematicaOutputWriter</i>	Writes the simulation outputs in the Mathematica format

	(.nb).
<i>EcMatlabOutputWriter</i>	Writes the simulation outputs in the MATLAB format (.m).
<i>EcTextOutputWriter</i>	Writes the simulation outputs in the comma-delimited text format (.txt). This can be loaded into Excel.
<i>EcXmlOutputWriter</i>	Writes the simulation outputs in the XML format (.xml).

**Table 15-10:** Currently implemented output writers.

There is a helper class *EcSystemStoredData* that provides a convenient way to write captured data to files, especially if one wishes to save the capture data in multiple formats all at once. It encapsulates a data storage, a vector of output writers, and file output streams so the user does not need to set them up.

Method	Description
setOutputWriterVector	Sets the vector of output writers.
allocateStorage	Allocates enough memory for the internal data storage.
beginSaveToFile	Opens a file or files that will be used to write the captured data and prepares the file(s) for saving.
finishSaveToFile	Finalizes the saving process and close the file(s).
saveCapturedData	Saves a set of captured data by appending it to the open file(s).
isSaving	Returns true if the saving process is ongoing.

**Table 15-11:** Methods in *EcSystemStoredData*.

## 15.2.2 Configuration Example

This section will show an example of how to programmatically configure the data capturing mechanism to store or plot the desired data from a simulation. Perhaps, an easier to perform the same configuration is through GUI, which is covered in the ActinViewer chapter.

### 15.2.2.1 Data Capture

The text box below shows the code snippet for configuring the data capture to capture the following information:

1. Base position of the first manipulator in the system (index 0).
2. Base velocity of the first manipulator.
3. Joint position of all the links in the first manipulator.
4. Joint velocity of the link labeled “link-3” in the first manipulator.
5. Placement of the first end-effector in the first manipulator.

```

// read the simulation from file
EcSystemSimulation simulation;
simulation.readFromFile("SimulationFile.xml");

// get the reference to the first manipulator
const EcIndividualManipulator&
    manip0=simulation.statedSystem().system().manipulators()[0];

// configure the data capture
EcSystemDataCapture sysDataCapture;
sysDataCapture.addManipulatorDataCaptureType(0,
    EcSimAnalysis::EcBasePositionDataCaptureTokenType);
sysDataCapture.addManipulatorDataCaptureType(0,
    EcSimAnalysis::EcBaseVelocityDataCaptureTokenType);
sysDataCapture.addLinkDataCaptureType(0,
    EcSimAnalysis::EcJointPositionDataCaptureTokenType, manip0);
sysDataCapture.addLinkDataCaptureType(0, "link-3",
    EcSimAnalysis::EcJointVelocityDataCaptureTokenType);
sysDataCapture.addEndEffectorDataCaptureType(0, 0,
    EcSimAnalysis::EcEndEffectorPlacementDataCaptureTokenType);

```

**Text Box 15-5:** Code snippet for configuring the desired data capture.

### 15.2.2.2 Writing Output to Files

Now, let's say we want to save the captured data into a file in a comma-separated text format. The text box below shows how to do this programmatically.

```

// create and then allocate the memory for a data storage
EcSystemDataStorage dataStorage;
simulation.storageSystemDataCapture().allocateStorage(dataStorage);

// create a vector for system data
EcSystemDataVector dataVector;
// add the system data capture as the first element of the vector
// this will provide the descriptions for the captured data
dataVector.pushBack(simulation.storageSystemDataCapture());

// simulation loop
const EcU32 numSteps=100;
const EcReal timeStep=0.01;
for(EcU32 ii=0; ii<numSteps; ++ii)
{
    EcReal time=ii*timeStep;

    // run the simulation
    // ...

    // afterwards, capture the data
    simulation.captureStorageData();
    dataStorage.setTime(time);
    // store the data in the storage
    simulation.storageSystemDataCapture().storeData(dataStorage);
    // add the data storage to the data vector
    dataVector.pushBack(dataStorage);
}

// output the data to a text file
EcTextOutputWriter textWriter;
const EcString txtFilename="output"+textWriter.fileExtension();
std::ofstream fout(txtFilename.c_str());

```

```

// initialize the stream
textWriter.initialize(dataVector,fout);
// write the data to the stream
for(EcU32 ii=0; ii<dataVector.size(); ++ii)
{
    textWriter.writeComponentOpening(dataVector[ii], fout);
    textWriter.writeDataComponent(dataVector[ii], fout);
    textWriter.writeComponentClosing(dataVector[ii], fout);
}
// finalize the stream
textWriter.finalize(dataVector, fout);

```

**Text Box 15-6:** Code snippet for writing captured data to file.

Alternatively, one can use the helper class *EcSystemStoredData* to write the captured data to files as illustrated in the text box below. In this example, the captured data is written in two different formats, namely Matlab and text formats.

```

// output the data to files in Matlab and comma-separated text formats
EcMatlabOutputWriter matlabWriter;
EcTextOutputWriter  textWriter;
EcOutputWriterVector outputWriterVector;
outputWriterVector.pushBack(matlabWriter);
outputWriterVector.pushBack(textWriter);

// use a convenient helper class EcSystemStoredData
EcSystemStoredData storedData;
// set the output writers and allocate memory for storage
storedData.setOutputWriterVector(outputWriterVector);
storedData.allocateStorage(sysDataCapture);

// set the file name without extension. File extension (.m, .txt) will be
automatically added
const EcString filename("output");
// initialize
storedData.beginSaveToFile(filename);
// simulation loop
const EcU32 numSteps=100;
const EcReal timeStep=0.01;
for(EcU32 ii=0; ii<numSteps; ++ii)
{
    EcReal time=ii*timeStep;

    // run the simulation
    // ...

    // afterwards, capture the data
    simulation.captureStorageData();

    // save the data to file
    storedData.saveCapturedData(time, simulation.storageSystemDataCapture());
}
// finalize
storedData.finishSaveToFile();

```

**Text Box 15-7:** An alternative method for writing captured data to file.

Note that the first example above stores all the history of captured data in a vector before writing it all in a file while the second example writes the captured data for each simulation time instance to files. The former will generally be faster since file I/O is slower than memory read/write. However, it will use more memory since it needs to store the entire history of the captured data in the simulation.

### 15.2.2.3 Output Examples

This section presents examples of outputs in different file formats. To simplify the outputs, let's configure the data capture to capture only the joint position and velocity of "link-0" of the first manipulator (index 0). The text box below shows an example of XML describing an *EcSystemDataCapture* object that has been configured as such.

```
<systemDataCapture xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.energid.com/namespace/sm
  sysDataCapture.sm.xsd" xmlns="http://www.energid.com/namespace/sm"
  version="1.0.0">
  <manipulatorDataCaptureVector size="1">
    <element>
      <dataCaptureTypeVector size="0"/>
      <endEffectorDataCaptureVector size="0"/>
      <linkDataCaptureVector size="1">
        <element>
          <dataCaptureTypeVector size="2">
            <jointPositionDataCaptureType>
              <dataLabels size="1">
                <element>Joint Position</element>
              </dataLabels>
              <disableFlags size="0"/>
            </jointPositionDataCaptureType>
            <jointVelocityDataCaptureType>
              <dataLabels size="1">
                <element>Joint Velocity</element>
              </dataLabels>
              <disableFlags size="0"/>
            </jointVelocityDataCaptureType>
          </dataCaptureTypeVector>
          <linkIdentification>link-0</linkIdentification>
        </element>
      </linkDataCaptureVector>
      <manipulatorIndex>0</manipulatorIndex>
    </element>
  </manipulatorDataCaptureVector>
  <systemDataCaptureLabel></systemDataCaptureLabel>
</systemDataCapture>
```

**Text Box 15-8:** Example of XML describing an *EcSystemDataCapture* instance.

The text boxes below show the same captured data but in different formats. Note that since XML is very verbose, the captured information of only one time stamp at 0 is shown.

```
(*NotebookFileLineBreakTest
NotebookFileLineBreakTest*)
(*NotebookOptionsPosition[      2080,      67]*)
(*NotebookOutlinePosition[      2744,      90]*)
(* CellTagsIndexPosition[      2700,      86]*)
(*WindowFrame->Normal*)

Notebook[{
Cell[BoxData[
  \(\ (Data = {
{"Time" , "Manip[0]:link-0:Joint Positions" , "Manip[0]:link-0:Joint Velocities"}
, {0 , 0.503881 , 1.8537e-022}
, {0.031 , 0.503672 , -0.00579795}
, {0.046 , 0.503514 , -0.0131736}
, {0.078 , 0.502705 , -0.00963094}
, {0.109 , 0.501687 , -0.0121206}
, {0.125 , 0.501287 , -0.0333618}
, {0.156 , 0.499639 , -0.0196119}
```

```

, {0.187 , 0.498148 , -0.0413905}
, {0.203 , 0.497398 , -0.0625211}
, {0.234 , 0.494881 , -0.0299539}
, {0.25 , 0.494109 , -0.0643908}
, {0.281 , 0.491284 , -0.0336166}
, {0.312 , 0.488996 , -0.0635215}
, {0.328 , 0.48735 , -0.0457239}
, {0.359 , 0.484769 , -0.0716736}
, {0.39 , 0.481368 , -0.0404647}
};\}\}, "Input",
CellOpen->False],

Cell[BoxData[
  \ (Do[Data[\([1, ii]\)] =
    StringReplace[
      Data[\([1,
        ii]\)], {"\<:\>" -> "\<\>", "\< \>" \[Rule] "\<\>", "\<- \>" -> \
"\<\>", "\<[\>" -> "\<\>", "\<]\>" -> "\<\>"}]\ ;
    ToExpression[
      Data[\([1, ii]\)] <> "\<=\>" <> "\<Table[Data[[jj,\>" <>
        ToString[ii] <> "\<]],{jj,2,Length[Data]]\>"], {ii, 1,
          Length[Data[\([1]\)]}\}], "Input",
      CellOpen->False],

Cell[BoxData[
  \ (variables = Data[\([1]\)]\)], "Input"
],
FrontEndVersion->"5.2 for Microsoft Windows",
ScreenRectangle->{{0, 1280}, {0, 911}},
WindowSize->{532, 740},
WindowMargins->{{0, Automatic}, {Automatic, 0}},
ShowSelection->True
]

```

**Text Box 15-9:** Example of simulation outputs in Mathematica format.

```

Description=[
'Time' 'Manip[0]:link-0:Joint Positions' 'Manip[0]:link-0:Joint Velocities'
];
Data=[
0      0.503881      1.8537e-022;
0.031  0.503672      -0.00579795;
0.046  0.503514      -0.0131736;
0.078  0.502705      -0.00963094;
0.109  0.501687      -0.0121206;
0.125  0.501287      -0.0333618;
0.156  0.499639      -0.0196119;
0.187  0.498148      -0.0413905;
0.203  0.497398      -0.0625211;
0.234  0.494881      -0.0299539;
0.25   0.494109      -0.0643908;
0.281  0.491284      -0.0336166;
0.312  0.488996      -0.0635215;
0.328  0.48735       -0.0457239;
0.359  0.484769      -0.0716736;
0.39   0.481368      -0.0404647;
];

```

**Text Box 15-10:** Example of simulation outputs in MATLAB format.

```

Time,Manip[0]:link-0:Joint Positions,Manip[0]:link-0:Joint Velocities
0,0.503881,1.8537e-022
0.031,0.503672,-0.00579795
0.046,0.503514,-0.0131736
0.078,0.502705,-0.00963094
0.109,0.501687,-0.0121206
0.125,0.501287,-0.0333618
0.156,0.499639,-0.0196119
0.187,0.498148,-0.0413905
0.203,0.497398,-0.0625211
0.234,0.494881,-0.0299539
0.25,0.494109,-0.0643908
0.281,0.491284,-0.0336166
0.312,0.488996,-0.0635215
0.328,0.48735,-0.0457239
0.359,0.484769,-0.0716736
0.39,0.481368,-0.0404647

```

**Text Box 15-11:** Example of simulation outputs in comma-delimited text format.

```

<?xml version="1.0" encoding="ISO-8859-1"?>
<default xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.energid.com/namespace/cr
  capturedDataStream.cr.xsd" xmlns="http://www.energid.com/namespace/cr" size="0">
<sm:generalCapture xmlns:sm="http://www.energid.com/namespace/sm">
  <sm:manipulatorDataCaptureVector size="1">
    <sm:element>
      <sm:dataCaptureTypeVector size="0"/>
      <sm:linkDataCaptureVector size="1">
        <sm:element>
          <sm:dataCaptureTypeVector size="2">
            <sm:jointPositionDataCaptureType>
              <sm:disableFlags size="0"/>
            </sm:jointPositionDataCaptureType>
            <sm:jointVelocityDataCaptureType>
              <sm:disableFlags size="0"/>
            </sm:jointVelocityDataCaptureType>
          </sm:dataCaptureTypeVector>
          <sm:linkIdentification>link-0</sm:linkIdentification>
        </sm:element>
      </sm:linkDataCaptureVector>
      <sm:manipulatorIndex>0</sm:manipulatorIndex>
    </sm:element>
  </sm:manipulatorDataCaptureVector>
  <sm:systemDataCaptureLabel>Storage</sm:systemDataCaptureLabel>
</sm:generalCapture>
<sm:generalStorage xmlns:sm="http://www.energid.com/namespace/sm">
  <sm:manipulatorDataVector size="1">
    <sm:element>
      <sm:linkLevelData size="1">
        <sm:element size="2">
          <sm:element size="1">
            <group>0.50388100000000002</group>
          </sm:element>
          <sm:element size="1">
            <group>1.8537000038131529e-022</group>
          </sm:element>
        </sm:element>
      </sm:linkLevelData>
      <sm:manipulatorLevelData size="0"/>
    </sm:element>
  </sm:manipulatorDataVector>

```



```
<sm:time>0</sm:time>
</sm:generalStorage>
</default>
```

**Text Box 15-12:** Example of simulation outputs in XML format.

### 15.2.2.4 Display in Plots

Shown below is a code snippet for configuring a display data capture to capture the following information:

1. Set the title of the plot to “Plot 1.”
2. For the left Y-axis of the plot
  - a. Set the label to “Joint Positions.”
  - b. Joint position of the link labeled “link-0” in the first manipulator.
  - c. Joint position of the link labeled “link-1” in the first manipulator.
3. For the right Y-axis of the plot
  - a. Set the label to “Joint Velocities.”
  - b. Joint velocity of the link labeled “link-0” in the first manipulator.
  - c. Joint velocity of the link labeled “link-1” in the first manipulator.

```
// set up the data capture for the left Y-axis
EcSystemDataCapture leftDataCapture;
leftDataCapture.setLabel("Joint Positions");
leftDataCapture.addLinkDataCaptureType(0, "link-0",
    EcSimAnalysis::EcJointPositionDataCaptureTypeToken);
leftDataCapture.addLinkDataCaptureType(0, "link-1",
    EcSimAnalysis::EcJointPositionDataCaptureTypeToken);

// set up the data capture for the right Y-axis
EcSystemDataCapture rightDataCapture;
rightDataCapture.setLabel("Joint Velocities");
rightDataCapture.addLinkDataCaptureType(0, "link-0",
    EcSimAnalysis::EcJointVelocityDataCaptureTypeToken);
rightDataCapture.addLinkDataCaptureType(0, "link-1",
    EcSimAnalysis::EcJointVelocityDataCaptureTypeToken);

// set up the display data capture
EcDisplaySystemDataCapture dispDataCapture;
dispDataCapture.setTitle("Plot 1");
dispDataCapture.setLeftSystemDataCapture(leftDataCapture);
dispDataCapture.setRightSystemDataCapture(rightDataCapture);
```

**Text Box 15-13:** Code snippet for configuring data capturing for plotting purposes.

Assuming that we have a class that can generate a graphical plot that takes an instance of *EcXmlRealVector* as a data point, the following code snippet shows how to prepare the captured data so that it is ready to be sent to the plotting class.

```
// set up the data storages for left and right Y-axis data
EcSystemDataStorage leftDataStorage;
EcSystemDataStorage rightDataStorage;

// allocate the memory
dispDataCapture.leftSystemDataCapture().
    allocateStorage(leftDataStorage);
dispDataCapture.rightSystemDataCapture().
    allocateStorage(rightDataStorage);
```

```

// vectors of numbers to cache the captured data for plotting
EcXmlRealVector leftData, rightData;

// simulation loop
const EcU32 numSteps=100;
const EcReal timeStep=0.01;
for(EcU32 ii=0; ii<numSteps; ++ii)
{
    EcReal time=ii*timeStep;

    // run the simulation
    // ...

    // afterwards, capture the data
    dispDataCapture.leftSystemDataCapture().captureData(simulation);
    dispDataCapture.rightSystemDataCapture().captureData(simulation);

    // store the data in the storage
    dispDataCapture.leftSystemDataCapture().storeData(leftDataStorage);
    dispDataCapture.rightSystemDataCapture().storeData(rightDataStorage);

    // obtain the captured data in the vector forms
    leftDataStorage.getRealElements(leftData);
    rightDataStorage.getRealElements(rightData);

    // then we can add leftData and rightData to the plot
}

```

**Text Box 15-14:** Code snippet for preparing the data from the data capture for plotting.

## 16 Studies

### 16.1 Parametric and Monte Carlo Studies

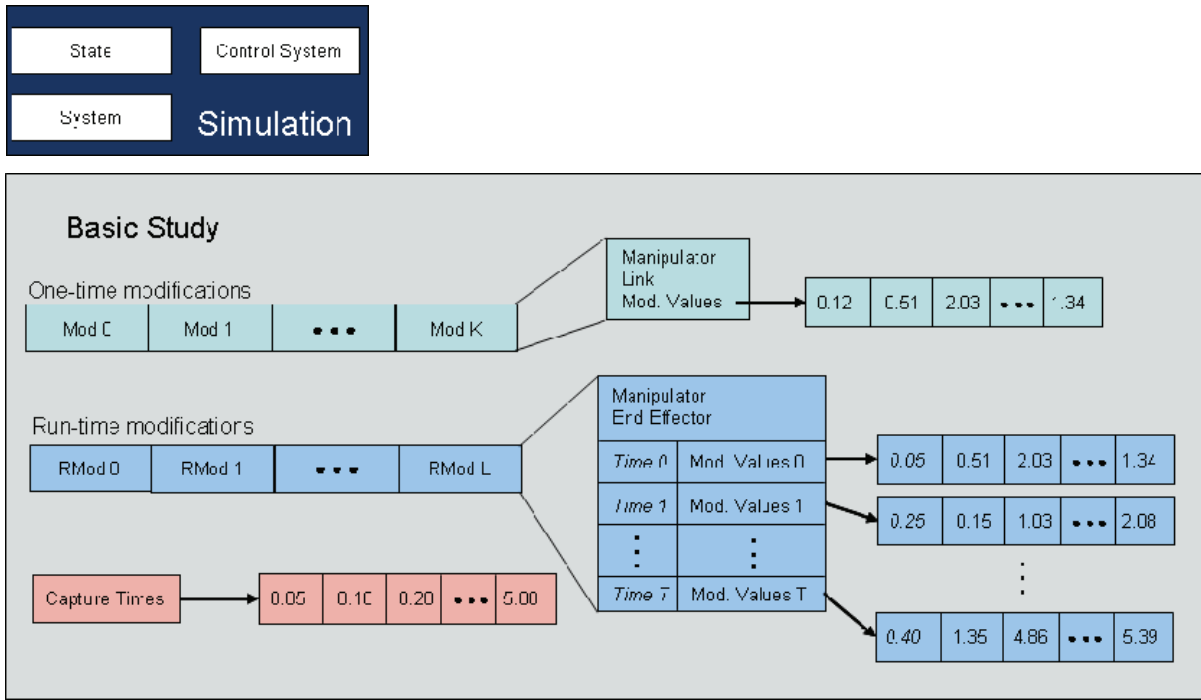
#### 16.1.1 Design

In our design, parametric and Monte Carlo studies are essentially the same thing except that, for Monte Carlo, randomness is introduced by using random modification classes as will be discussed later. There are three study classes – namely *EcBasicStudy*, *EcSimpleStudy*, and *EcComprehensiveStudy* – differing from one another in terms of complexity, with *EcBasicStudy* being the simplest and *EcComprehensiveStudy* the most complex. Depending on the need of the desired study, one can choose to use one type of study over the others.

Figure 16-1 shows the diagram of a basic study. A basic study represents a single simulation run. In order to run a basic study, a *simulation* – which contains the state (positions and velocities), the system (physical manipulators), and control system – is first passed through a sequence of ‘one-time’ modifications, each of which modifies a single entry of the simulation (either the system, the state, or the control system). Each modification is a collection of data sets that include 1) the manipulator index, 2) the minor index (link or end-effector index), and 3) a vector of new values for the parameters to be modified. For example, one modification object may change the joint position of “link-0” of manipulator 0. Another may change the mass property of “link-5” of manipulator 2.

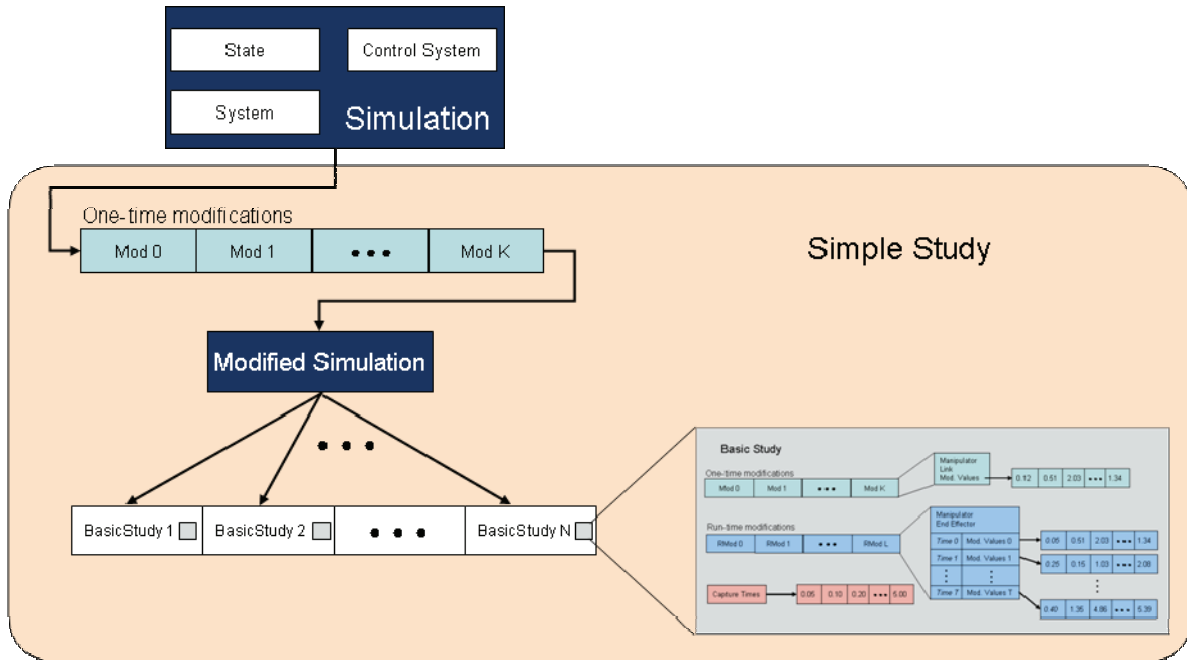
‘Run-time’ modifications are similar to ‘one-time’ modifications in the sense that they too modify some part of the simulation. However, instead of doing it only once, they continually make

modifications throughout the simulation timespan at the specified time instances. The capture times specify the time instances at which the simulation output should be captured (to later be written to file or analyzed).



**Figure 16-1:** The diagram of a basic study.

Up another level is a simple study, diagramed in Figure 16-2, which is essentially a collection of basic studies that are executed in series in a single thread. Before entering each of the basic studies, a simulation can be modified through a set of one-time modifications. These modifications will affect all the basic studies, as shown in Figure 16-2.



**Figure 16-2:** The diagram of a simple study.

At the highest level is a comprehensive study, diagramed in Figure 16-3. A comprehensive is basically a collection of simple studies that can be executed in different threads. Note that the number of simple studies does not have to equal the number of threads. If the number of threads is less than the number of simple studies, the remaining studies will wait in line and start their runs only after the first batch of studies have finished. For example, if the number of threads is 2 and the number of simple studies is 5. Then, only two simple studies will be run simultaneously. After, the first two studies finish, the next two will then be run. Finally, the fifth study will be executed after the third and the fourth study complete their runs.

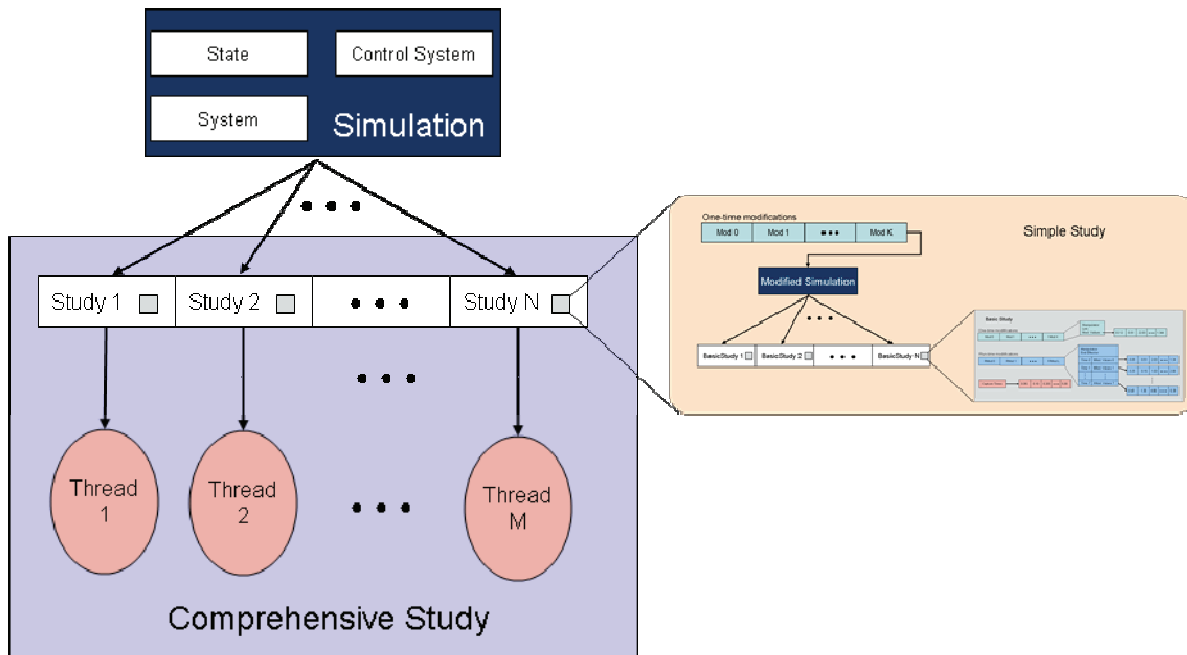
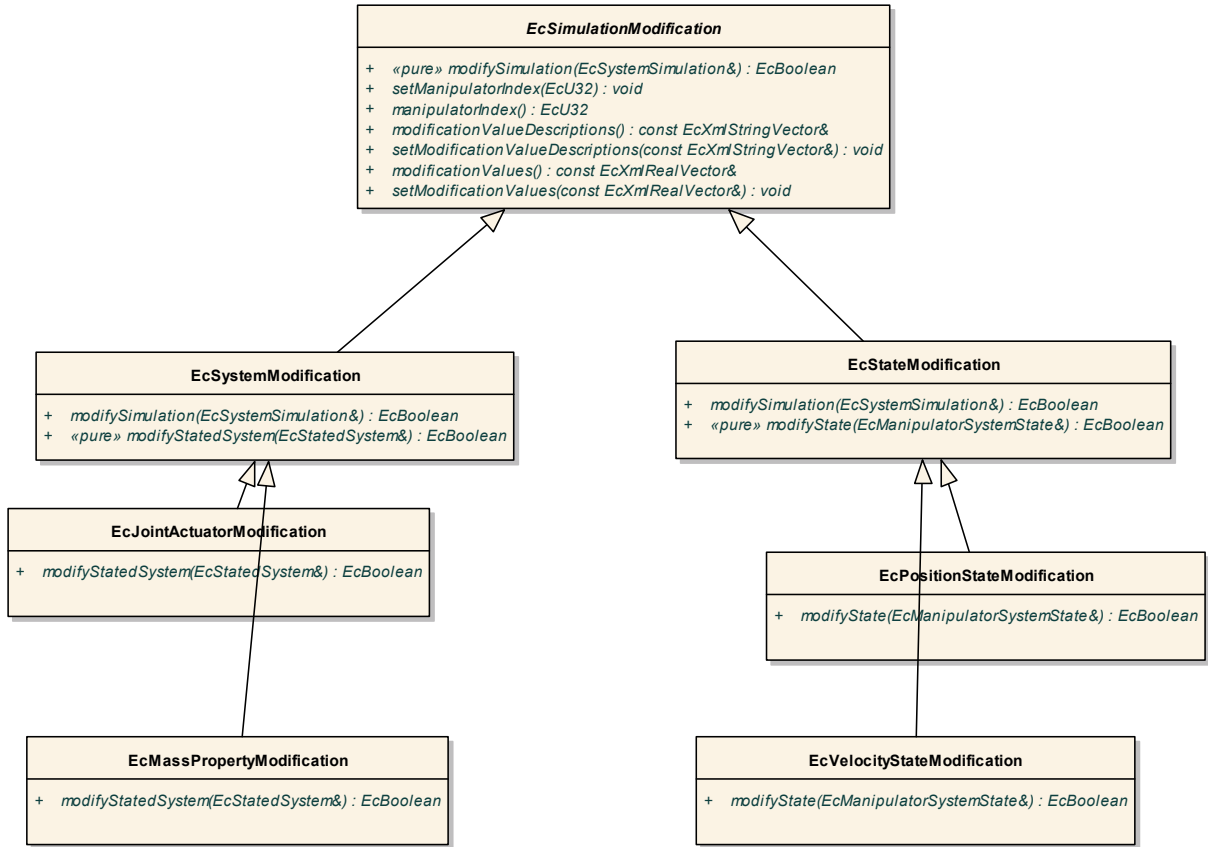


Figure 16-3: The diagram of a comprehensive study.

### 16.1.2 Implementation

The classes to support parametric and Monte Carlo studies can be broadly categorized into two types: modification classes and study classes. As the name suggests, modification classes are responsible for modifying properties or parameters of some part of a simulation. Modification classes are divided into two categories: one-time modification and run-time modification. Study classes in general use modification classes to modify the parameters as desired and then run studies based on the modified simulation. The outputs of these studies can be written in several formats including XML, MATLAB, Mathematica, and Excel. This section describes the details of modification and study classes.

Figure 16-4 shows a class diagram for ‘one-time’ modification classes. These modifications are used to make desired changes to the simulation prior to the study starts a new run. The top three classes in the diagram – *EcSimulationModification*, *EcSystemModification*, and *EcStateModification* – are abstract. *EcSimulationModification* is the base class of all ‘one-time’ modification classes. The main method in *EcSimulationModification* is *modifySimulation*, which is a pure virtual function and takes an *EcSystemSimulation* as an argument. As the names imply, *EcSystemModification* and *EcStateModification* are used to modify the system parameters and the state, respectively. The content of *EcSimulationModification* class is described in Table 16-1.



**Figure 16-4:** Class diagram of 'one-time' modifications.

Method	Description
<i>manipulatorIndex</i> / <i>setManipulatorIndex</i>	Gets/sets a manipulator in the system through its index.
<i>minorIndex</i> / <i>setMinorIndex</i>	For the chosen manipulator, gets/sets either a link or an end-effector through its index.
<i>modifySimulation</i>	Pure virtual method. Derived class must implement this to modify the simulation according to its own rules.

**Table 16-1:** Methods in *EcSimulationModification*.

There are currently four concrete modification classes:

1. *EcJointActuatorModification* is derived from *EcSystemModification* and is responsible for applying changes to a joint actuator, e.g. maximum joint torques, viscous friction, etc.
2. *EcMassPropertyModification* is derived from *EcSystemModification* and is responsible for applying changes to the mass property of a link, i.e. mass, first moment of inertia, and second moment of inertia.

3. *EcPositionStateModification* is derived from *EcStateModification* and is responsible for applying changes to a joint position. If the specified link is the base link, the modification values are for the position and orientation (through a quaternion) of the base.
4. *EcVelocityStateModification* is derived from *EcStateModification* and is responsible for applying changes to a joint velocity. If the specified link is the base link, the modification values are for the linear and angular velocities of the base.

Other types of modifications can be added by deriving from either *EcSystemModification*, *EcStateModification*, or even directly from *EcSimulationModification*.

Similar to ‘one-time’ modification, all ‘run-time’ modification classes are derived from *EcRunTimeSimulationModification*. Run-time modifications serve the same purpose as the one-time counterpart except that they can be applied multiple times during the run instead of once prior to the run. Currently, only *EcDesiredPlacementPathModification* is derived from *EcRunTimeSimulationModification*. It is used to modify the desired placements along the path of an end-effector.

Member	Description
<i>manipulatorIndex / setManipulatorIndex</i>	Gets/sets a manipulator in the system through its index.
<i>minorIndex / setMinorIndex</i>	For the chosen manipulator, gets/sets either a link or an end-effector through its index.
<i>modifySimulation</i>	Pure virtual method. Derived class must implement this to modify the simulation at specified times according to its own rules.

**Table 16-2:** Methods in *EcRunTimeSimulationModification*.

If the vector of time instances and the vector of modification values are of size 1 and that time is 0, then run-time modifications behave exactly like ‘one-time’ modifications. It may be tempting to dismiss the use of one-time modifications in favor of the run-time counterpart. However, since there are overheads, in terms of performance and resources, associated with run-time modifications, one-time modifications are recommended for use with any system or state that is meant to be modified only once.

*EcSimulationModificationVector* and *EcRunTimeSimulationModificationVector* are vector classes that contain instances of classes derived from *EcSimulationModification* and *EcRunTimeSimulationModification*, respectively. They are used in the study classes described below.

As far as study classes, *EcBaseStudy* is the abstract base class for all study classes. Three study classes are implemented for the three studies diagrammed in Figure 16-1 to Figure 16-3. They are *EcBasicStudy*, *EcSimpleStudy*, and *EcComprehensiveStudy*. The main method which must be implemented by all classes derived from *EcBaseStudy* is *startSimulations*, which takes an instance of *EcSystemSimulation* as an argument. Important methods of *EcBaseStudy* as well as their descriptions are listed in Table 16-3.

Method	Description
<i>numThreadsForSimulations / setNumThreadsForSimulations</i>	Gets/sets the maximum number of threads to be created for simulation runs.
<i>simulationTimes / setSimulationTimes</i>	Gets/sets all the simulation time instances for the end-effector paths.
<i>endEffectorPaths / setEndEffectorPaths</i>	Gets/sets all the desired end-effector placements at the time instances specified in simulationTimes.
<i>randomNumberSeed / setRandomNumberSeed</i>	Gets/sets the seed for random number generation. Used for Monte Carlo simulations.
<i>startSimulations</i>	Pure virtual method. Must be implemented by all derived classes. Start the simulations for the study.

**Table 16-3:** Methods in *EcBaseStudy*.

The methods specific to *EcBasicStudy* class are listed in Table 16-4. The simulation run is started by calling *startSimulations* method which takes an *EcSystemSimulation* object as an argument. A series of one-time modifications will then be applied prior to the run and a series of run-time modifications during the run.

Method	Description
<i>modificationVector / setModificationVector</i>	Gets/sets the ‘one-time’ modifications that should be applied before starting a new run.
<i>runTimeModificationVector / setRunTimeModificationVector</i>	Gets/sets all the ‘run-time’ modifications that should be applied during the run.
<i>captureTimes / setCaptureTimes</i>	Gets/sets all the capture times for the run. Capture times are used to signify that the data should be captured at those time instances.
<i>simulationTimeStep / setSimulationTimeStep</i>	Gets/sets the simulation time step.
<i>dataStorage</i>	Gets the simulation output data captured at the times specified in captureTimes.
<i>startSimulations</i>	Start the simulation for the study.

**Table 16-4:** Methods in *EcBasicStudy*.

*EcSimpleStudy* is up one level from *EcBasicStudy* in the study hierarchy. It contains a vector of *EcBasicStudy* instances as well as a vector of one-time modifications common to all basic studies. The methods in *EcSimpleStudy* are listed in Table 16-5.



Member	Description
<i>commonModificationVector</i> / <i>setCommonModificationVector</i>	Gets/sets the one-time modification vector that is common to all basic studies.
<i>basicStudies</i> / <i>setBasicStudies</i>	Gets/sets the series of basic studies.
<i>runIndividualPaths</i> / <i>setRunIndividualPaths</i>	Gets/sets whether or not the individual basic studies should use their own paths when running the simulations.
<i>startSimulations</i>	Start the simulations for all the basic studies. These basic studies are run in series in a single thread.

**Table 16-5:** Methods in *EcSimpleStudy*.

At the top of the study hierarchy is *EcComprehensiveStudy*, which contains a vector of *EcSimpleStudy* instances. Table 16-6 lists the methods specific to *EcComprehensiveStudy*.

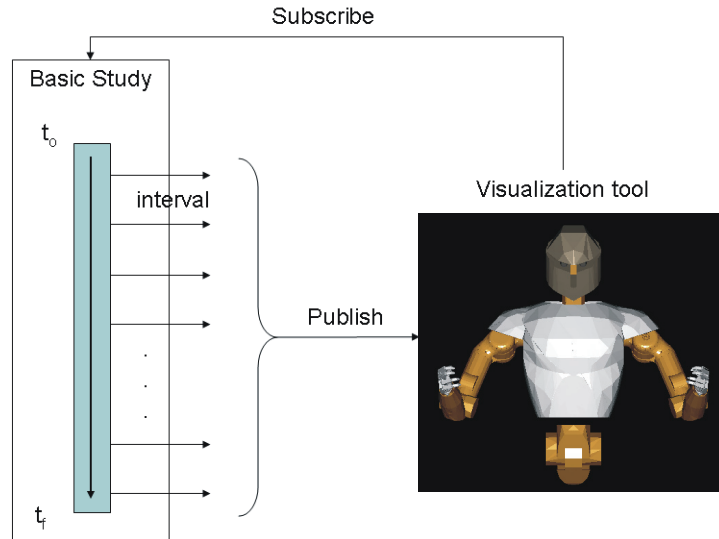
Member	Description
<i>simpleStudies</i> / <i>setSimpleStudies</i>	Gets/sets the vector of simple studies.
<i>runIndividualPaths</i> / <i>setRunIndividualPaths</i>	Gets/sets whether or not the individual simple studies should use their own paths when running the simulations.
<i>startSimulations</i>	Start the simulations for all the simple studies. These studies are run in parallel in multiple threads.

**Table 16-6:** Methods in *EcComprehensiveStudy*.

## 16.2 Simulation Visualization

While a Monte Carlo simulation or a parametric study is running, it would be insightful to be able to visualize what is happening. Support for visualization must be embedded in the simulation loop. This is accomplished through a callback mechanism through subscribe-publish model.

Due to the hierarchical structure presented in the previous section, actual simulation loops are only implemented in *EcBasicStudy*, while instances of *EcSimpleStudy* and *EcComprehensiveStudy* delegate the simulation work to the basic studies. Because of this, the visualization support needs only be added to the simulation loop in *EcBasicStudy*.



**Figure 16-5:** Simulation visualization with subscribe-publish model.

Figure 16-5 Illustrates how the callback mechanism works. The visualization tool subscribes to a basic study that it wants to receive the simulation data. This model allows any number of subscribers to the same study. During the simulation, in this case of Robonaut, the data is published and all the subscribers then receive the simulation data every interval. The *EcStateCallback* class was created to manage the callback mechanism. Its methods are listed in Table 16-7. Table 16-8 shows the additional members related to the callback in *EcBasicStudy*. Note that, since publishing the data incurs some overhead, *EcBasicStudy* has an option not to publish any data if there is no subscriber wanting to receive the data.

Method	Description
<i>subscribe</i>	Subscribes to this callback.
<i>unsubscribe</i>	Unsubscribe from this callback.
<i>publish</i>	Publishes the new data to be consumed by the subscriber.
<i>numSubscribers</i>	Returns the total number of subscribers for this callback.

**Table 16-7:** Methods of *EcStateCallback*.

Method	Description
<i>callbackIndex / setCallbackIndex</i>	Gets/sets the callback index. Used for identification in case of multiple callbacks.
<i>callbackIntervalInMs / setCallbackIntervalInMs</i>	Gets/sets interval in millisecond (simulation time, not wall clock time) for which the simulation data is published.
<i>subscribe</i>	Subscribes to the callback in this study.
<i>unsubscribe</i>	Unsubscribes from the callback in this study.

**Table 16-8:** Additional methods in *EcBasicStudy* to handle callbacks.

Also available is a program called “studyTool” that executes a comprehensive study. It provides a rendering window that subscribes to the callbacks in the sub-studies of the comprehensive study. Running a study is then a two-step process. The first step is to programmatically create a study and save it in a file. Coding examples of this process will be given later. Then, you can run “studyTool” on the study file to execute the study.

## 16.3 Randomization and Monte Carlo Simulation

For Monte Carlo simulation, it is usually desirable and convenient to randomize properties of the whole system. This is accomplished with two new modification classes. The first class, *EcSystemRandomModification*, is intended to randomly modify the system properties while the other, *EcStateRandomModification*, randomizes the initial state. These two classes can be used just like any other modification class. However, instead of modifying the properties and/or the state of just a single link of a single manipulator, they are designed to randomly modify the properties and/or the state of all links of all manipulators.

### 16.3.1 Randomization of System Properties

Figure 16-6 shows a diagram of classes that are involved in randomizing system properties. *EcSystemRandomModification* is derived from *EcSystemModification*, which was previously detailed. Like all modification classes, the main function of *EcSystemRandomModification* is to modify the simulation that is passed on to it according to its rule, which is to randomize all the properties of all links of all manipulators in the system. *EcSystemRandomModification* relies on another class – *EcSystemRandomVariation* – on how to randomize the system properties. Since manipulators are identified by indices and links by string labels, *EcSystemRandomVariation* contains a vector of maps of strings and instances of *EcLinkPropertyRandomVariation*. Each instance of *EcLinkPropertyRandomVariation* corresponds to a link in a manipulator and contains a set of instances of variation classes, each of which knows how to randomize a specific property of a link. Currently, the following four variation classes have been implemented:

1. *EcLinkKinematicsVariation*: This randomizes the link kinematics by adjusting the precursor.
2. *EcMassPropertyVariation*: This randomizes the mass properties (mass and first and second moments of inertia).
3. *EcJointActuatorRandomVariation*: This randomizes mechanical properties of actuators such as inertia, viscous friction, etc.

4. *EcSurfacePropertyVariation*: This randomizes the damper and spring constants of the surface.

*EcSystemRandomVariation* also contains an instance of *EcLinkPropertyRandomVariation* that is used as the default variation for all the links not contained in the vector of maps. This is very convenient because we need not specify the variations for all the links. For example, to have one variation apply to all the links, we can just specify the default variation and leave the vector of maps empty.

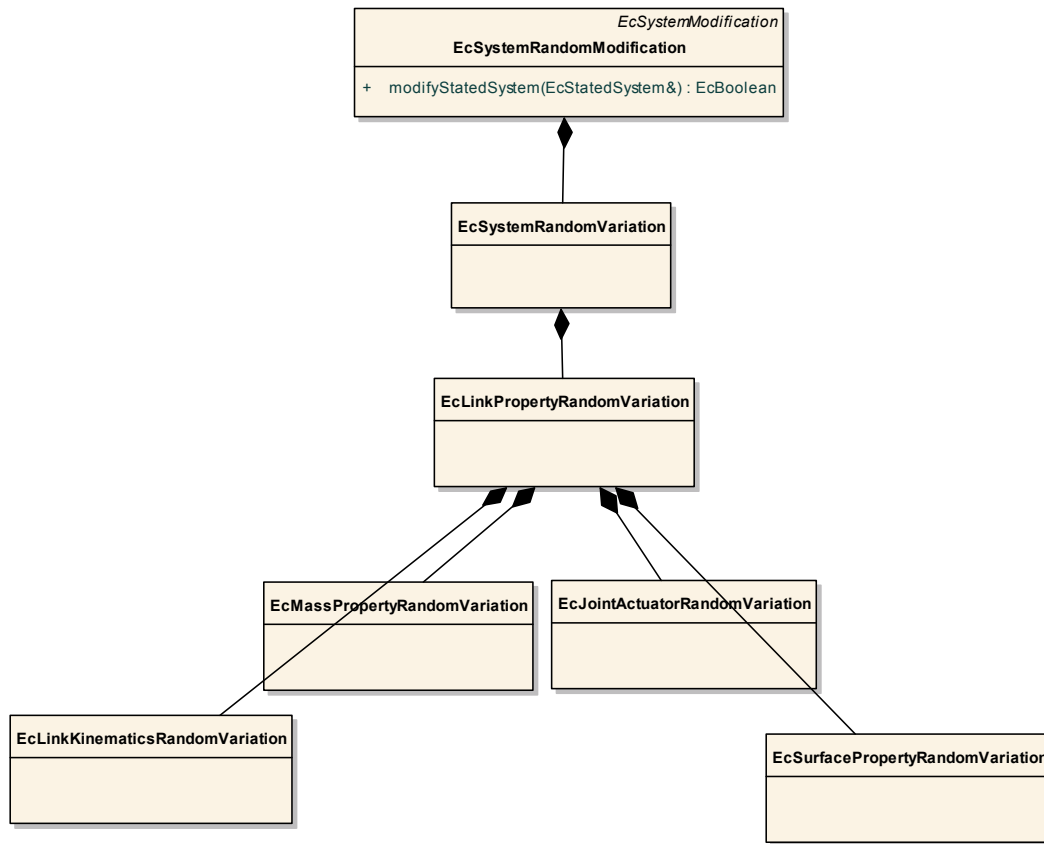


Figure 16-6: Class diagram for randomization of system properties.

### 16.3.2 Randomization of State Variables

Functionally, *EcStateRandomModification* is to the state what *EcSystemRandomModification* is to the system. It is used to randomize all the state variables of all manipulators in the system. As with *EcSystemRandomModification*, *EcStateRandomModification* also relies on another class – *EcStateRandomVariation* – to provide the means to randomize the state variables. *EcStateRandomVariation* contains a vector of *EcPositionStateRandomVariation* instances and a vector of *EcVelocityStateRandomVariation* instances. An instance of *EcPositionStateRandomVariation* represents random variations of position state variables of a single manipulator. The same is also true for *EcVelocityStateRandomVariation* but for the velocity state variables. The relationships of all classes related to randomization of state are depicted in Figure 16-7.

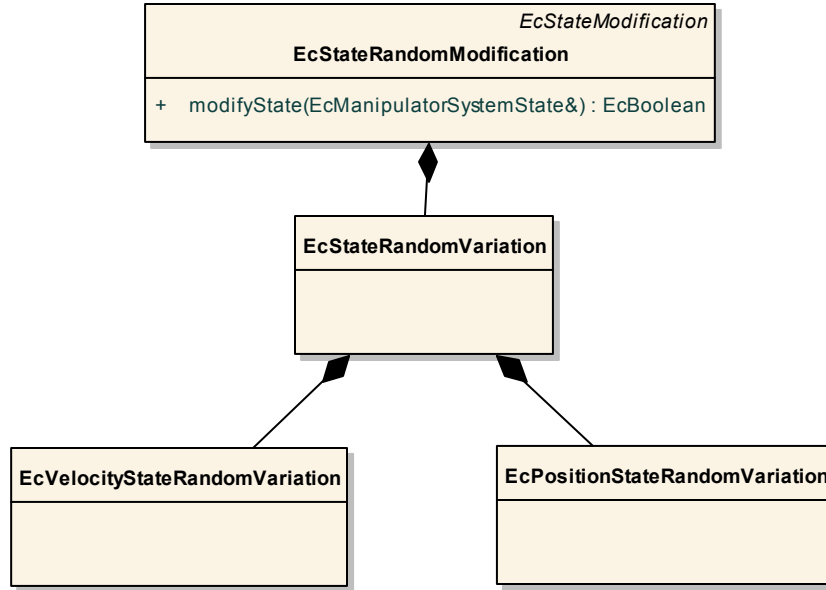


Figure 16-7: Class diagram for randomization of state.

## 16.4 Mass Properties Randomization

Randomization of most properties is straightforward assuming that the standard deviations (one-sigma values) of those properties are given. However, these one-sigma values are not always readily available for a robot under study. Mass property information is an example of this. Robot dynamic behavior changes directly and significantly with the mass properties, yet it is difficult to estimate how the mass properties change in the field. In fact, it is often challenging even to get nominal data for the second moment of inertia—difficulty in obtaining data on variation from the nominal values is much greater.

Mass properties consist of a mass (one scalar), a first moment of inertia (three scalars), and a second moment of inertia (six scalars). These values are interrelated—increasing mass tends to increase the elements of the first and second moments. This implies that the standard deviations of the elements of the first and second moments of inertia are related to the standard deviation of the mass. What a robot modeler would hope for in the best case would be a way to estimate the variation of the first and second moments of inertia as a function of a known variation in mass, as variation in mass is easy to estimate with a scale and a set of the objects to be modeled. A mathematical approach to doing this is exactly what we worked out in the last quarter, which is presented below.

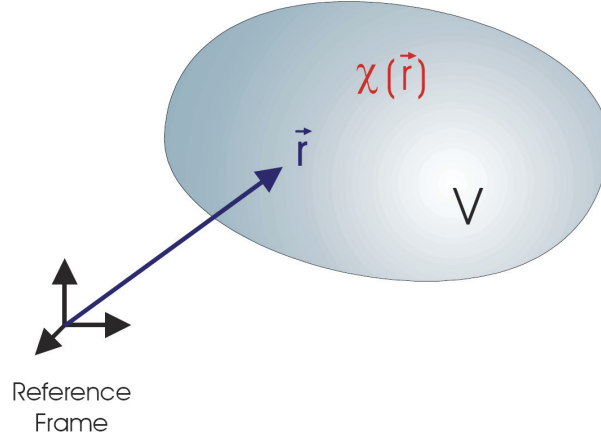
### 16.4.1 Background

In the general sense (not restricted to mass properties), let the integration of a zero-mean random variable  $\chi$  over a 3D volume  $V$  be established as follows:

$$X = \int_V \chi dV \quad (16-1)$$

Here,  $\vec{r}$  is a 3D point within the volume  $V$ .

This integral is not rigorously defined here—the constraint imposed on it is that it must be consistent with discrete approximation. Let the standard deviation of  $X$  be labeled  $\sigma_X$ . These properties are illustrated in the figure below.



**Figure 16-8:** An illustration of the properties used to define equation (16-1).

Then, let some function of 3D position  $\vec{r}$  over the volume  $V$  be  $f(\vec{r})$ , and let the random variable  $F$  be defined as follows:

$$F = \int_V f(\vec{r}) \chi(\vec{r}) dV \quad (16-2)$$

It is proposed that, with reasonable constraints on  $f$  and  $\chi$  as a function of  $\vec{r}$ , the standard deviation of  $F$  is given by the following:

$$\sigma_F = \sigma_X \sqrt{\frac{1}{V} \int_V f^2(\vec{r}) dV} \quad (16-3)$$

The rationale for this is the following:

Equation (16-1) can be approximated as a summation over  $N$  subvolumes forming  $V$ :

$$X \approx \hat{X} = \sum_{i=1}^N \chi(\vec{r}_i) \Delta V \quad (16-4)$$

Assuming each sample of  $\chi$  is independent with standard deviation  $\sigma_\chi$ ,

$$\sigma_{\hat{X}}^2 = \sum_{i=1}^N \sigma_\chi^2 \Delta V^2 = N \sigma_\chi^2 \Delta V^2 = \frac{\sigma_\chi^2 V^2}{N} \quad (16-5)$$

Similarly,

$$F \approx \hat{F} = \sum_{i=1}^N f(\vec{r}_i) \chi(\vec{r}_i) \Delta V \quad (16-6)$$

$$\sigma_{\hat{F}}^2 = \sum_{i=1}^N \sigma_{\chi}^2 f^2(\vec{r}_i) \Delta V^2 = \sigma_{\chi}^2 \Delta V \sum_{i=1}^N f^2(\vec{r}_i) \Delta V = \frac{\sigma_{\chi}^2 V}{N} \sum_{i=1}^N f^2(\vec{r}_i) \Delta V \approx \frac{\sigma_{\chi}^2}{V} \int_V f^2(\vec{r}) dV \quad (16-7)$$

This leads directly to the formula in (16-3).

### 16.4.2 Mass

The mass of rigid body defined over volume  $V$  is given by

$$m = \int_V \rho dV, \quad (16-8)$$

where  $\rho$  is the mass density for the differential volume  $dV$ . To describe the variability of the density, let it be represented as

$$\rho = (1 + \chi) \bar{\rho}, \quad (16-9)$$

where  $\bar{\rho}$  is a nominal value and, again,  $\chi$  has zero mean. Then the mass becomes

$$m = \int_V (1 + \chi) \bar{\rho} dV = \int_V \bar{\rho} dV + \int_V \chi \bar{\rho} dV \quad (16-10)$$

Let

$$\bar{m} = \int_V \bar{\rho} dV \quad (16-11)$$

be the nominal mass, and

$$\tilde{m} = \int_V \chi \bar{\rho} dV, \quad (16-12)$$

be the random variation, so that

$$m = \bar{m} + \tilde{m}. \quad (16-13)$$

Because  $\bar{m}$  is constant, the formula in (16-3) gives

$$\sigma_m = \sigma_x \sqrt{\frac{1}{V} \int_V \bar{\rho}^2(\vec{r}) dV} \quad (16-14)$$

In general, we will assume that we know  $\sigma_m$  for the rigid body being modeled (not  $\sigma_x$ ).

### 16.4.3 First Moment

The first moment is given by

$$\vec{h} = \int_V \vec{r} \rho dV \quad (16-15)$$

This gives

$$\vec{h} = \int_V \vec{r} \bar{\rho} dV + \int_V \vec{r} \chi \bar{\rho} dV \quad (16-16)$$

Let  $\bar{h}$  be the nominal component,

$$\bar{h} = \int_V \vec{r} \bar{\rho} dV. \quad (16-17)$$

And let  $\tilde{h}$  be the random variation,

$$\tilde{h} = \int_V \vec{r} \chi \bar{\rho} dV. \quad (16-18)$$

Representing  $\vec{r}$  as  $\vec{r} = [x \ y \ z]^T$ , and substituting into (16-3) gives

$$\vec{\sigma}_{\bar{h}} = \begin{bmatrix} \sigma_{h_x} \\ \sigma_{h_y} \\ \sigma_{h_z} \end{bmatrix} = \sigma_x \begin{bmatrix} \sqrt{\frac{1}{V} \int_V x^2 \bar{\rho}^2 dV} \\ \sqrt{\frac{1}{V} \int_V y^2 \bar{\rho}^2 dV} \\ \sqrt{\frac{1}{V} \int_V z^2 \bar{\rho}^2 dV} \end{bmatrix} \quad (16-19)$$

Substituting in (16-14) gives



$$\bar{\sigma}_{\bar{h}} = \sigma_m \left[ \begin{array}{c} \sqrt{\int_V x^2 \bar{\rho}^2 dV} / \sqrt{\int_V \bar{\rho}^2 dV} \\ \sqrt{\int_V y^2 \bar{\rho}^2 dV} / \sqrt{\int_V \bar{\rho}^2 dV} \\ \sqrt{\int_V z^2 \bar{\rho}^2 dV} / \sqrt{\int_V \bar{\rho}^2 dV} \end{array} \right] \quad (16-20)$$

This gives a formula for the standard deviation of the first moment of inertia of a rigid body as a function of the standard deviation of its mass. A useful approximation results if the nominal mass density  $\bar{\rho}$  is assumed constant over the volume. In this case, the standard deviation of first moment takes on a special form:

$$\bar{\sigma}_{\bar{h}} = \sigma_m \left[ \begin{array}{c} \sqrt{\bar{\rho} \int_V x^2 \bar{\rho} dV} / \sqrt{\bar{\rho} \int_V \bar{\rho} dV} \\ \sqrt{\bar{\rho} \int_V y^2 \bar{\rho} dV} / \sqrt{\bar{\rho} \int_V \bar{\rho} dV} \\ \sqrt{\bar{\rho} \int_V z^2 \bar{\rho} dV} / \sqrt{\bar{\rho} \int_V \bar{\rho} dV} \end{array} \right]. \quad (16-21)$$

This gives

$$\bar{\sigma}_{\bar{h}} = \frac{\sigma_m}{\sqrt{m}} \left[ \begin{array}{c} \sqrt{\int_V x^2 \bar{\rho} dV} \\ \sqrt{\int_V y^2 \bar{\rho} dV} \\ \sqrt{\int_V z^2 \bar{\rho} dV} \end{array} \right], \quad (16-22)$$

which can be expressed as

$$\bar{\sigma}_{\bar{h}} = \frac{\sigma_m}{\sqrt{2m}} \left[ \begin{array}{c} \sqrt{I_{yy} + I_{zz} - I_{xx}} \\ \sqrt{I_{xx} + I_{zz} - I_{yy}} \\ \sqrt{I_{xx} + I_{yy} - I_{zz}} \end{array} \right], \quad (16-23)$$

where  $I_{xx}$ ,  $I_{yy}$ , and  $I_{zz}$  are the diagonal elements of the standard moment of inertia tensor, given by

$$\begin{aligned}
I_{xx} &= \int_V (y^2 + z^2) \bar{\rho} dV \\
I_{yy} &= \int_V (x^2 + z^2) \bar{\rho} dV \\
I_{zz} &= \int_V (x^2 + y^2) \bar{\rho} dV
\end{aligned}
\tag{16-24}$$

The formula in (16-23) gives the standard deviation of the first moment of a rigid body as a function of the standard deviation of the mass and ordinary mass properties.

#### 16.4.4 Second Moment

The diagonal elements of the inertia matrix are defined as above. In addition, the off-diagonal elements are defined as follows:

$$\begin{aligned}
I_{xy} &= -\int_V xy \bar{\rho} dV \\
I_{xz} &= -\int_V xz \bar{\rho} dV \\
I_{yz} &= -\int_V yz \bar{\rho} dV
\end{aligned}
\tag{16-25}$$

It follows from (16-3) and an argument similar to that used for the first moment of inertia that the standard deviations for the six scalars defining the second moment of inertia are given by the following:

$$\sigma_{I_{xx}} = \sigma_m \sqrt{\int_V (y^2 + z^2)^2 \bar{\rho}^2 dV} / \sqrt{\int_V \bar{\rho}^2 dV}
\tag{16-26}$$

$$\sigma_{I_{yy}} = \sigma_m \sqrt{\int_V (x^2 + z^2)^2 \bar{\rho}^2 dV} / \sqrt{\int_V \bar{\rho}^2 dV}
\tag{16-27}$$

$$\sigma_{I_{zz}} = \sigma_m \sqrt{\int_V (x^2 + y^2)^2 \bar{\rho}^2 dV} / \sqrt{\int_V \bar{\rho}^2 dV}
\tag{16-28}$$

$$\sigma_{I_{xy}} = \sigma_m \sqrt{\int_V x^2 y^2 \bar{\rho}^2 dV} / \sqrt{\int_V \bar{\rho}^2 dV}
\tag{16-29}$$

$$\sigma_{Ixz} = \sigma_m \sqrt{\int_V x^2 z^2 \bar{\rho}^2 dV} / \sqrt{\int_V \bar{\rho}^2 dV} \quad (16-30)$$

$$\sigma_{Iyz} = \sigma_m \sqrt{\int_V y^2 z^2 \bar{\rho}^2 dV} / \sqrt{\int_V \bar{\rho}^2 dV} \quad (16-31)$$

If the nominal mass density  $\bar{\rho}$  is constant over the volume, these take the following simplified form:

$$\begin{aligned} \sigma_{Ixx} &= \frac{\sigma_m}{\sqrt{m}} \sqrt{\int_V (y^2 + z^2)^2 \bar{\rho} dV} & (16-32) \\ &= \frac{\sigma_m}{\sqrt{V}} \sqrt{\int_V (y^2 + z^2)^2 dV} \\ &= \frac{\sigma_m}{\sqrt{V}} \sqrt{\int_V (y^4 + 2y^2 z^2 + z^4) dV} \end{aligned}$$

$$\begin{aligned} \sigma_{Iyy} &= \frac{\sigma_m}{\sqrt{m}} \sqrt{\int_V (x^2 + z^2)^2 \bar{\rho} dV} & (16-33) \\ &= \frac{\sigma_m}{\sqrt{V}} \sqrt{\int_V (x^4 + 2x^2 z^2 + z^4) dV} \end{aligned}$$

$$\begin{aligned} \sigma_{Izz} &= \frac{\sigma_m}{\sqrt{m}} \sqrt{\int_V (x^2 + y^2)^2 \bar{\rho} dV} & (16-34) \\ &= \frac{\sigma_m}{\sqrt{V}} \sqrt{\int_V (x^4 + 2x^2 y^2 + y^4) dV} \end{aligned}$$

$$\sigma_{Ixy} = \frac{\sigma_m}{\sqrt{m}} \sqrt{\int_V x^2 y^2 \bar{\rho} dV} = \frac{\sigma_m}{\sqrt{V}} \sqrt{\int_V x^2 y^2 dV} \quad (16-35)$$

$$\sigma_{kxz} = \frac{\sigma_m}{\sqrt{V}} \sqrt{\int_V x^2 z^2 dV} \quad (16-36)$$

$$\sigma_{lyz} = \frac{\sigma_m}{\sqrt{V}} \sqrt{\int_V y^2 z^2 dV} \quad (16-37)$$

### 16.4.5 Validity of Randomized Mass Properties

After random perturbation, the mass properties must meet requirements for validity. Mass can take on any positive value. The first moment of inertia can take on any real values. But the second moment of inertia has complex constraints. The second moment of inertia must be positive semi-definite, and no diagonal entry can exceed the sum of the other two. These constraints are represented by the following equations:

$$I_{xx} \geq 0 \quad (16-38)$$

$$I_{xx} I_{yy} \geq I_{xy}^2 \quad (16-39)$$

$$2I_{xy} I_{xz} I_{yz} + I_{xx} I_{yy} I_{zz} \geq I_{yz}^2 I_{xx} + I_{xz}^2 I_{yy} + I_{xy}^2 I_{zz} \quad (16-40)$$

$$I_{xx} + I_{yy} \geq I_{zz} \quad (16-41)$$

$$I_{xx} + I_{zz} \geq I_{yy} \quad (16-42)$$

$$I_{yy} + I_{zz} \geq I_{xx} \quad (16-43)$$

### 16.4.6 Ellipsoid Volume

To be able to compute the standard deviations of the elements of the second moment of inertia, we assume the volume of an ellipsoid. We then need solutions to volume integrals over a 3-dimensional ellipsoid. According to Sykora [33], the volume integrals over n-dimensional ellipsoids have a closed-form expression of

$$W_n(p, a) = \left( \prod_{k=1}^n a_k^{2p_k+1} \right) \frac{2}{2p+n} \frac{\prod_{k=1}^n \Gamma(p_k + 1/2)}{\Gamma(p + n/2)} \quad (16-44)$$

where

$$W_n(p, a) = \int_{V(a)} E(r, p) d\tau \quad (16-45)$$

$$E(r, p) \equiv \prod_{k=1}^n (x_k^2)^{p_k} \quad (16-46)$$

$\Gamma(x)$  is the gamma function;  $p = \sum_{k=1}^n p_k$  where  $p_k$  is non-negative real exponent of the  $k$ -component;  $r \equiv \{x_1, x_2, \dots, x_n\}$  where  $r$  is the position vector in the  $n$ -dimensional Euclidean space,  $x$ 's are its Cartesian coordinates;  $a_k$  is the length of the semi-axis in the  $x_k$  direction.

In our case, we are dealing with the 3-dimensional ellipsoid, so  $n=3$ . Also, for the 3-dimension case, we represent  $x_1, x_2$ , and  $x_3$  with  $x, y$ , and  $z$ , respectively and  $a_1, a_2$ , and  $a_3$  with  $a, b$ , and  $c$ .

Since we need the gamma function, we will note here that  $\Gamma(1/2) = \sqrt{\pi}$ . By using the property  $\Gamma(x+1) = x\Gamma(x)$ , we can then obtain  $\Gamma(3/2) = \frac{1}{2}\sqrt{\pi}$ ,  $\Gamma(5/2) = \frac{3}{4}\sqrt{\pi}$ , and  $\Gamma(7/2) = \frac{15}{8}\sqrt{\pi}$ .

From (16-44), by setting  $p_1=2, p_2=0$ , and  $p_3=0$ . we can solve for  $\int_V x^4 dV$  as

$$\begin{aligned} \int_V x^4 dV &= a^5 bc \frac{2}{7} \frac{\Gamma(5/2)\Gamma^2(1/2)}{\Gamma(7/2)} \\ &= \frac{12\pi}{105} a^5 bc \end{aligned} \quad (16-47)$$

Similarly,

$$\begin{aligned} \int_V y^4 dV &= ab^5 c \frac{2}{7} \frac{\Gamma(5/2)\Gamma^2(1/2)}{\Gamma(7/2)} \\ &= \frac{12\pi}{105} ab^5 c \end{aligned} \quad (16-48)$$

and

$$\begin{aligned}\int_V z^4 dV &= abc^5 \frac{2 \Gamma(5/2)\Gamma^2(1/2)}{7 \Gamma(7/2)} \\ &= \frac{12\pi}{105} abc^5\end{aligned}\tag{16-49}$$

From (16-44), by setting  $p_1=1, p_2=1$ , and  $p_3=0$ . we can solve for  $\int_V x^2 y^2 dV$  as

$$\begin{aligned}\int_V x^2 y^2 dV &= a^3 b^3 c \frac{2 \Gamma^2(3/2)\Gamma(1/2)}{7 \Gamma(7/2)} \\ &= \frac{4\pi}{105} a^3 b^3 c\end{aligned}\tag{16-50}$$

Similarly,

$$\begin{aligned}\int_V y^2 z^2 dV &= ab^3 c^3 \frac{2 \Gamma^2(3/2)\Gamma(1/2)}{7 \Gamma(7/2)} \\ &= \frac{4\pi}{105} ab^3 c^3\end{aligned}\tag{16-51}$$

and

$$\begin{aligned}\int_V x^2 z^2 dV &= a^3 bc^3 \frac{2 \Gamma^2(3/2)\Gamma(1/2)}{7 \Gamma(7/2)} \\ &= \frac{4\pi}{105} a^3 bc^3\end{aligned}\tag{16-52}$$

Substituting (16-48), (16-49), and (16-51) and  $V = \frac{4}{3} \pi abc$  into (16-32) gives

$$\begin{aligned}\sigma_{xx} &= \frac{\sigma_m}{\sqrt{V}} \sqrt{\frac{12\pi}{105} ab^5 c + \frac{8\pi}{105} ab^3 c^3 + \frac{12\pi}{105} abc^5} \\ &= \frac{2\sigma_m \sqrt{\pi(3ab^5 c + 2ab^3 c^3 + 3abc^5)}}{\sqrt{105V}} \\ &= \sigma_m \sqrt{\frac{1}{35} (3b^4 + 2b^2 c^2 + 3c^4)}\end{aligned}\tag{16-53}$$

Similarly,

$$\sigma_{lyy} = \sigma_m \sqrt{\frac{1}{35} (3a^4 + 2a^2c^2 + 3c^4)} \quad (16-54)$$

$$\sigma_{lzz} = \sigma_m \sqrt{\frac{1}{35} (3a^4 + 2a^2b^2 + 3b^4)} \quad (16-55)$$

$$\sigma_{lxy} = \frac{\sigma_m ab}{\sqrt{35}} \quad (16-56)$$

$$\sigma_{lzx} = \frac{\sigma_m ac}{\sqrt{35}} \quad (16-57)$$

$$\sigma_{lyz} = \frac{\sigma_m bc}{\sqrt{35}} \quad (16-58)$$

### 16.4.7 Ellipsoids from Mass Properties

Given a set of mass properties (mass, first moment, and second moment), there is only one ellipsoid that can give those properties. This is the ellipsoid centered at the center of mass and aligned with the principal axes of inertia with ellipsoid semiaxes that give the principal moments of inertia for a density that gives the mass.

Calculating this ellipsoid is a multistep process. First, the center of the ellipsoid is established as the center of mass. That is,

$$\bar{c} = \frac{1}{m} \vec{h} \quad (16-59)$$

where  $\bar{c}$  is the center of the ellipsoid. The mass properties are then represented at the center of mass using the methods built into the class *EcRigidBodyMass*. Next, the principal frame of the ellipsoid is selected as the principal frame of the second moment of inertia using eigenvalue decomposition on the  $3 \times 3$  matrix representing the second moment of inertia. This frame is defined through the rotation matrix established as follows:

$$R = [e_0 \quad e_1 \quad e_2] \quad (16-60)$$

where  $e_0$ ,  $e_1$ , and  $e_2$  are the normalized orthogonal eigenvectors of the  $3 \times 3$  symmetric matrix representing the second moment of inertia. (Symmetric matrices always have eigenvectors that satisfy these requirements.)

The mass properties are then represented in this frame, again using the methods built into the class *EcRigidBodyMass*.

In this frame, the  $3 \times 3$  second moment of inertia matrix has zero off-diagonal terms,  $I_{xy}$ ,  $I_{xz}$ , and  $I_{yz}$ . The diagonal terms,  $I_{xx}$ ,  $I_{yy}$ , and  $I_{zz}$  are used to calculate the semiaxes of the ellipsoid A, B, and C using the following formula:

$$\begin{bmatrix} A^2 \\ B^2 \\ C^2 \end{bmatrix} = \frac{5}{2m} \begin{bmatrix} -1 & 1 & 1 \\ 1 & -1 & 1 \\ 1 & 1 & -1 \end{bmatrix} \begin{bmatrix} I_{xx} \\ I_{yy} \\ I_{zz} \end{bmatrix} \quad (16-61)$$

With these values, the surface of the ellipsoid is defined as all points  $\{x,y,z\}$  satisfying

$$\frac{(x-c_x)^2}{A^2} + \frac{(y-c_y)^2}{B^2} + \frac{(z-c_z)^2}{C^2} = 1 \quad (16-62)$$

in the frame centered at the center of mass and aligned with the principal axes of inertia.

## 16.5 Coding Examples

This section will show you how to programmatically create and save parametric and Monte Carlo studies so that it can later be run by the studyTool executable.

### 16.5.1 Parametric Study

In this set of examples, we will create a parametric study in which we will apply the following modifications to the first manipulator (index 0) of the system:

- Move the base position by 0.5 m in the Y direction.
- Change the joint position of link 2 to 0.25 rad.
- Add offset to the length of link 0 by 0.01 m in the X axis.
- Add 1.5 kg to the mass of link 1.

```
// the simulation
EcSystemSimulation simulation;
simulation.readFromFile("simulation.xml");

// get convenient references
const EcStatedSystem& statedSystem=simulation.statedSystem();
const EcPositionState& posState=statedSystem.state().positionStates()[0];
const EcIndividualManipulator& manip=statedSystem.system().manipulators()[0];

// move the base position of manip0 by 0.5 m in Y direction
const EcCoordinateSystemTransformation origBasePos=posState.coordSysXForm();
EcPositionStateModification basePosMod;
basePosMod.setManipulatorIndex(0);
basePosMod.setLinkIndex(EcIndividualManipulator::BASEINDEX);
EcCoordinateSystemTransformation newBasePos(origBasePos);
```



```

newBasePos.setTranslationY(origBasePos.translation().y()+0.5);
basePosMod.setBasePosition(newBasePos);

// change the joint position of link2 of manip0 to 0.25 rad
EcPositionStateModification jointPosMod;
jointPosMod.setManipulatorIndex(0);
jointPosMod.setLinkIndex(2);
jointPosMod.setJointPosition(0.25);

// add offset of 0.01 m in X axis to length of link0 of manip0
EcLinkKinematicsModification linkKinMod;
linkKinMod.setManipulatorIndex(0);
linkKinMod.setLinkIndex(0);
EcCoordinateSystemTransformation offset;
offset.setTranslationX(0.01);
linkKinMod.setOffset(offset);

// add 1.5 kg to mass of link1 of manip0
const EcManipulatorLink* link1=manip.linkByIndex(1);
const EcRigidBodyMassProperties& origMassProp=link1->massProperties();
EcMassPropertyModification massPropMod;
massPropMod.setManipulatorIndex(0);
massPropMod.setLinkIndex(1);
EcRigidBodyMassProperties newMassProp;
newMassProp.set(
    origMassProp.mass()+EcNonNegReal(1.5),
    origMassProp.firstMoment(),
    origMassProp.secondMoment());
massPropMod.setMassProperties(newMassProp);

// add all the modifications to a vector
EcSimulationModificationVector modVector;
modVector.pushBack(basePosMod);
modVector.pushBack(jointPosMod);
modVector.pushBack(linkKinMod);
modVector.pushBack(massPropMod);

// create a basic study and set the modification vector in the study
EcBasicStudy basicStudy1;
basicStudy1.setModificationVector(modVector);

```

**Text Box 16-1:** Code snippet for creating a modification vector and a basic study.

Now, say, we want to add another study with almost the same modification vector, except that we do not need the last modification (mass properties) and we want these two studies to be run in series (one after another).

```

// remove the last modification (mass properties)
modVector.erase(3);
// create another basic study and set the modification vector in the study
EcBasicStudy basicStudy2;
basicStudy2.setModificationVector(modVector);

// create a vector of basic studies and add the two studies
EcBasicStudyVector basicStudies;
basicStudies.pushBack(basicStudy1);
basicStudies.pushBack(basicStudy2);

// create a simple study and set the vector of basic studies
EcSimpleStudy simpleStudy1;
simpleStudy1.setBasicStudies(basicStudies);

```

**Text Box 16-2:** Code snippet for adding basic studies to a simple study.

Now we also want another study but without the last two modifications (link length and mass properties). However, this time we want this study to be run in parallel with the first two studies.

```
// remove the last modification (link length)
modVector.erase(2);
// create another basic study and set the modification vector in the study
EcBasicStudy basicStudy3;
basicStudy3.setModificationVector(modVector);

// clear the study vector and add the last study
basicStudies.clear();
basicStudies.pushBack(basicStudy3);

// create another simple study and set the vector of basic studies
EcSimpleStudy simpleStudy2;
simpleStudy2.setBasicStudies(basicStudies);

// create a vector of simple study and add the two simple studies
EcSimpleStudyVector simpleStudies;
simpleStudies.pushBack(simpleStudy1);
simpleStudies.pushBack(simpleStudy2);

// create a comprehensive study and set the vector of simple studies
EcComprehensiveStudy compStudy;
compStudy.setSimpleStudies(simpleStudies);
compStudy.setNumThreadsForSimulations(2);

// save the study to file
compStudy.writeToFile("comprehensiveStudy.xml",
EcSimStudy::EcComprehensiveStudyToken);
```

**Text Box 16-3:** Code snippet for creating a comprehensive study.

Although the above code snippet shows you how to save the comprehensive to a file, note that to be of use for the studyTool program, we must not only save the study but also the simulation.

```
// create the simulation times and EE paths
EcXmlRealVector simTimes;
EcManipulatorEndEffectorPlacementVectorVector eePaths;

// ... populate the times and paths here

// set the times and paths
compStudy.setSimulationTimes(simTimes);
compStudy.setEndEffectorPaths(eePaths);

// set the simulation and study and write it to file
EcSimulationAndStudy simAndStudy;
simAndStudy.setSystemSimulation(simulation);
simAndStudy.setComprehensiveStudy(compStudy);
simAndStudy.writeToFile("simAndStudy.xml", EcSim::EcSimulationAndStudyToken);
```

**Text Box 16-4:** Code snippet for saving the study to be run by studyTool executable.

## 16.5.2 Monte Carlo Study

This set of examples will show how to create a Monte Carlo study. We would like to randomize the mass properties and link kinematics of all links as follow:

- The default standard deviation for mass is 0.1 kg.
- The default standard deviation for link length is 0.02 m
- The default standard deviation for link orientation is 0.5 deg

- The standard deviation for the mass of “link-1” of manipulator 0 is 0.0 (no randomness)

```

// the default random variations
// for mass, set the std deviation to 0.1 kg
EcMassPropertyRandomVariation defMassRandVar;
defMassRandVar.setMassOneSigma(0.1);
// for link kinematics, set the std dev for translation to 0.02 m and for
// orientation to 0.5 deg
EcLinkKinematicsRandomVariation defLinkKinRandVar;
defLinkKinRandVar.setTranslationOneSigma(0.02);
defLinkKinRandVar.setAngleOneSigma(0.5*EcDEG2RAD);
// set both to the default link random variations
EcLinkPropertyRandomVariation defLinkRandVar;
defLinkRandVar.setMassPropertyVariation(defMassRandVar);
defLinkRandVar.setLinkKinematicsVariation(defLinkKinRandVar);

// create a system random variation and set the default
EcSystemRandomVariation sysRandVar;
sysRandVar.setDefaultLinkPropertyRandomVariation(defLinkRandVar);

// a new set of random variations
EcMassPropertyRandomVariation massRandVar;
massRandVar.setMassOneSigma(0.0);
EcLinkPropertyRandomVariation linkRandVar;
linkRandVar.setMassPropertyVariation(massPropMod);
linkRandVar.setLinkKinematicsVariation(defLinkRandVar);
// set this link random variation for link-1 of manipulator0
sysRandVar.addLinkPropertyRandomVariation(0, "link-1", linkRandVar);

// create a system random modification and set the system random variation to
// modification
EcSystemRandomModification sysRandMod;
sysRandMod.setSystemRandomVariation(sysRandVar);

// create a modification vector and add the random modification
EcSimulationModificationVector modVector;
modVector.pushBack(sysRandMod);

// create a basic study and set the modification vector in the study
EcBasicStudy basicStudy1;
basicStudy1.setModificationVector(modVector);
basicStudy1.setRandomNumberSeed(1157);

```

**Text Box 16-5:** Code snippet for creating a Monte Carlo study.

## 17 Rendering

### 17.1 Overview

To support simulation and control, Actin provides many robot rendering functions. OpenGL was chosen as the low-level graphics library for this because of its capabilities and its being supported across most computing platforms. This section covers window creation and usage as well as user interaction within the window. The description of rendering within Actin has been separated into three sections. The first section gives an overview of the abstract *EcWindow* class which provides the foundation for all other windows. Platform-specific details are also located in this section. Following that is a discussion of general 2D rendering windows, typically used for image processing

tasks or displaying images to the user. The last section covers 3D rendering using OpenSceneGraph. It also includes a subsection on user input handling.

## 17.2 EcWindow: Base Rendering Window Class

A consistent approach to working with rendering windows is needed to provide an easy to use interface for development. The abstract class *EcWindow* provides the basic methods needed for tasks common to all types of windows. Rendering windows rely on the underlying operating system to create, initialize, display, update and clean up. This collection of OS-specific commands are implemented within *EcWindow* in a general manner, with the actual low-level calls being issued through an implementation class *EcWindow::Impl*, discussed in section 17.2.1.

Method	Description
getWindowSize setWindowSize	The interior, rendering portion of the window, not including border.
getWindowPos setWindowPos	The top-left corner of the rendering window.
getWindowSizeAndPos setWindowSizeAndPos	Convenience method on both size and position attributes.
getName setName	The title bar located on top of the window.
getBackgroundColor setBackgroundColor	Color for the rendering portion of the window, given as an <i>EcColor</i> .
getTopMost setTopMost	Attribute defining whether the window should stay on top (un-obscured) of other windows.
getHideWindow setHideWindow	Attribute defining whether the window should be visible on the screen.
getShare setShare	OpenGL allows for sharing of resources across windows. Used to keep from duplicating identical data, especially if the dataset is large.
[static] getImplType [static] setImplType	Default implementation used for window creation.
getImpl setImpl	OS-specific handle to low level details.
swapBuffers	OpenGL drawing is typically performed on a back buffer and the user visible (displayed) buffer is the front. This method will swap the two buffers making the current OpenGL drawing operations visible.
makeCurrent	Make the selected window be the target for any subsequent OpenGL operations.
closeWindow	Close and un-initialize an existing window.
[static] getWindow	General accessor to locate the current window in use.

<code>init</code>	Initialize the window with the set attributes and opens the window.
<code>setImageBuffer</code>	Display a 2D image in the window. It will resize the window based on the incoming image.
<code>renderScene</code>	Perform drawing operations and refresh window.
<code>renderSceneAndCaptureImage</code>	Similar to <code>renderScene</code> , but also capture the contents of the drawing buffer into an <i>EcImage</i> .

**Table 17-1:** *EcWindow* methods.

These methods provide the foundation for all of the other rendering classes that are built upon *EcWindow*. In addition, all of the operating system-specific code is handled in an implementation class, *EcWindow::Impl*.

### 17.2.1 *EcWindow::Impl*

Windowing code by its very nature is system-dependent. A good part of *EcWindow* is composed of pass-through methods to the implementation layer, *EcWindow::Impl*. This abstract class provides a common working set of methods for each implementation. There are currently three subclasses defined: 1) *EcWindow::Impl::Win* for Microsoft Windows platforms (XP and Vista), 2) *EcWindow::Impl::X11* for Linux and Unix derivatives, and 3) *EcWindow::Impl::Qt* for interfacing with Trolltech's GUI libraries. The *Qt* class, while not defining a set of methods for an Operating System, provides a translation of methods from the Actin toolkit to Trolltech's API. Subsequent windowing classes require the use of Qt for a consistent user interface design. By keeping the platform-specific details out of the windowing code, the framework is set to enable the inclusion of additional classes with minimal impact.

If new platform types are required (Apple OSX for example), they can be easily integrated by providing the OS-specific commands derived from *EcWindow::Impl*. The implementation used by default is set to native (*Win* or *X11* currently). This can be changed globally the following command:

```

static void setImplType
(
    const ImplType type
);

where ImplType is defined as

enum ImplType
{
    ImplWin32,    ///< Win32 standard window implementation.
    ImplX11,     ///< X11 window implementation.
    ImplQt,      ///< Trolltech Qt window implementation.
    ImplDefault  ///< Platform default.
};

```

**Text Box 17-1:** Static method used to set the global default implementation for all windows.

This will set the default for all subsequent window creation calls to the specified type.

Changing the implementation on a per-window basis is done using the `setImpl` method within *EcWindow* and the `newImpl` creator method in *EcWindow::Impl* and sub-classes.

An example of how to create a Qt-based window is shown below:

```
// Create a 2D rendering window object.
EcSimpleWindow myQtWindow;
// Force it to use Qt under the covers.
myQtWindow.setImpl(EcWindow::Impl::Qt::newImpl());

// Now we can init and use.
myQtWindow.setImageBuffer(image);
```

**Text Box 17-2:** Example of how to set the implementation type on a per-window basis.

This will take any *EcWindow*-derived object and make it use the specified implementation. Note the implementation type must be set before window initialization.

## 17.3 2D Rendering Windows

2D rendering windows are useful for image processing calculations or for displaying individual images, such as an *EcImage*. They provide a 1:1 mapping between image and screen pixels. Two subclasses of *EcWindow* are available, *EcSimpleWindow* and *EcFBOWindow*. Both use OpenGL as the rendering library.

### 17.3.1 *EcSimpleWindow*

*EcSimpleWindow* provides two main methods, a static method for displaying a popup window for a short time, and a persistent method.

```

/// Give the ability to create an image and relinquish control to user.
/// \param[in] image      EcImage to display.
/// \param[in] duration  Length of time to keep window up.
/// \param[in] title     EcString to put in titlebar.
/// \param[in] posX     Location to place window in X.
/// \param[in] posY     Location to place window in Y.
/// \return      EcBoolean Success or failure of command.
static EcBoolean displayImage
(
    const EcImage &image,
    const EcReal duration = 2.0,
    const EcString& title = "EcSimpleWindow",
    const EcU32 posX = 0,
    const EcU32 posY = 0
);

/// Pass an image to the rendering window to display.
/// \param[in] image      EcImage to display.
/// \param[in] swap      Whether to refresh screen or not.
/// \return      EcBoolean Success or failure of command.
virtual EcBoolean setImageBuffer
(
    const EcImage &image,
    const EcBoolean swap = EcTrue
);

```

**Text Box 17-3:** New methods defined for *EcSimpleWindow*.

A common use for *EcSimpleWindow* is to display an *EcImage*, whether it is temporary or persistent. Image sequences can also be displayed by repetitive calls to `setImageBuffer`.

**17.3.2 EcFBOWindow**

*EcFBOWindow* builds on top of *EcSimpleWindow* by providing FrameBufferObject (FBO) support. FBO is an OpenGL concept that allows for off-screen rendering as well as being able to perform floating-point calculations directly within the rendering pipeline. This is particularly useful when using the Graphics Processing Unit (GPU) to accelerate image processing calculations. There are a number of GPU-based algorithms located in the sensor-imageTransform project. Table 17-2 lists the new methods specific to FBO rendering.

Method	Description
getOffScreenRendering setOffScreenRendering	When off-screen rendering is true, the window will not be visible on the display, but is still available for rendering.
getKeepBound setKeepBound	When multiple rendering operations are to be performed in succession, it is desirable to keep the FBO bound to the rendering pipeline to

	improve efficiency.
setBufferParams	Specific requirements on the FBO are set by passing in the desired <i>EcTextureParameters</i> . A typical case would be to create a floating-point buffer for performing GPU calculations.

**Table 17-2:** *EcFBOWindow* methods.

*EcFBOWindow* is primarily used for image processing involving floating-point calculations. An example of how to configure a floating-point buffer is shown below.

```

/// Create our processing window.
EcFBOWindow floatWindow;
EcTextureParameters floatParams;
/// Specify that we want to use a floating-point buffer.
floatParams.setPixelFormat(EcTextureParameters::Float32RGB);
floatWindow.setBufferParams(floatParams);
/// Float buffers are only available off-screen.
floatWindow.setOffScreenRendering(EcTrue);

... /// Render and process

```

**Text Box 17-4:** Example of how to configure a floating-point buffer for *EcFBOWindow*.

## 17.4 3D Rendering Windows

The classes of 3D rendering windows utilize a scene graph based data structure to efficiently process and display the scene. The OpenSceneGraph toolkit is used to fulfill this need.

### 17.4.1 *EcSGWindow*

The *EcSGWindow* class provides the base class for all scene graph-based 3D rendering. It provides the necessary infrastructure to allow user input. This class is typically used as a secondary window into an existing scene since it doesn't provide direct capability for loading in scene data.

Method	Description
getNode setNode	Works with the top-level scene graph node.
getViewer	Retrieves a pointer to internal OSG class.
getInputHandler setInputHandler	Mechanism for providing user input event handling.



getDrawCallback setDrawCallback	Mechanism for providing custom drawing routines.
------------------------------------	--

**Table 17-3:** *EcSGWindow* methods.

#### 17.4.1.1 *EcSGWindow::DrawCallback*

This class provides internal support for replacing the drawing routines. Three separate methods are defined – `preRender`, `render`, and `postRender`. By default the `preRender` and `postRender` do nothing. These two methods are typically the only ones overridden to define new behavior. The `render` method is set to call `preRender`, then the default `osgViewer::Viewer::frame` method and finally `postRender`.

#### 17.4.1.2 *EcSGWindow::ImageCaptureDrawCallback*

This class is a specific implementation of the *EcSGWindow::DrawCallback* class designed to perform window capturing. It overrides the `preRender` method to initialize and setup the capture location as well as the `postRender` methods to collect the results. It is used internally for the `renderSceneAndCaptureImage` method.

#### 17.4.2 *EcRenderWindow*

*EcRenderWindow* is the first standalone implementation based off of *EcSGWindow*. It is a self contained class used primarily for loading and rendering an *EcVisualizableStatedSystem*. It is used to display discrete states of an *EcStatedSystem*. The methods used to set and retrieve system information is given in Table 17-4.

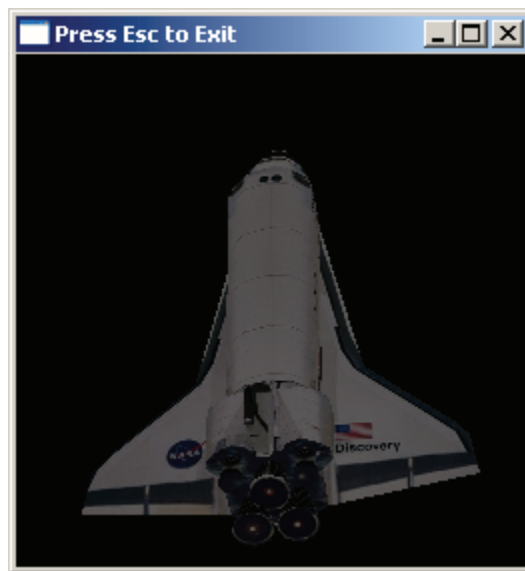
Method	Description
<code>setVisualizableStatedSystem</code>	Loads an <i>EcStatedSystem</i> as well as the <i>EcVisualizationParameters</i> . After a successful call, the scene is ready to be rendered.
<code>statedSystem</code> <code>setStatedSystem</code>	Operates only on the <i>EcStatedSystem</i> .
<code>state</code> <code>setState</code>	Operates only on the <i>EcState</i> .

**Table 17-4:** *EcRenderWindow* methods.

## 17.5 Qt-Based Classes

### 17.5.1 *EcSGWidgetQt*

*EcSGWidgetQt* is based off of *QGLWidget*. The main purpose of this class is to encapsulate the combined use of a Qt Widget with the *EcWindow* class. It also provides the connection of mouse and keyboard events from Qt to OpenSceneGraph. These user events are translated and passed through to be handled using one of the input handlers, described in section 17.6 User Input. This class is typically coupled inside of a different window class such as *EcRenderWindow* or subclassed such as in *EcBaseViewerMainWidget*. One additional method that is located in *EcSGWidgetQt* is `renderUntilEscaped`. What this method does is take the existing scene graph data and allow the user to rotate the model until the Escape key is pressed. An example is shown in Figure 17-1.



**Figure 17-1:** Example of *EcSGWidgetQt* contained within an *EcRenderWindow*. A *EcVisualizableStatedSystem* is loaded and then `renderUntilEscaped` is called to generate an interactive window that the user can manipulate.

### 17.5.2 *EcCamera*

*EcCamera* is a specific class that combines the features of both an *EcSGWindow* as well as *EcSGWidgetQt*. It provides a secondary view off another window that can be manipulated easily with mouse and keyboard input. This class takes advantage of OpenGL context sharing. It has a creator method, `newCamera`, for instantiating a new object.

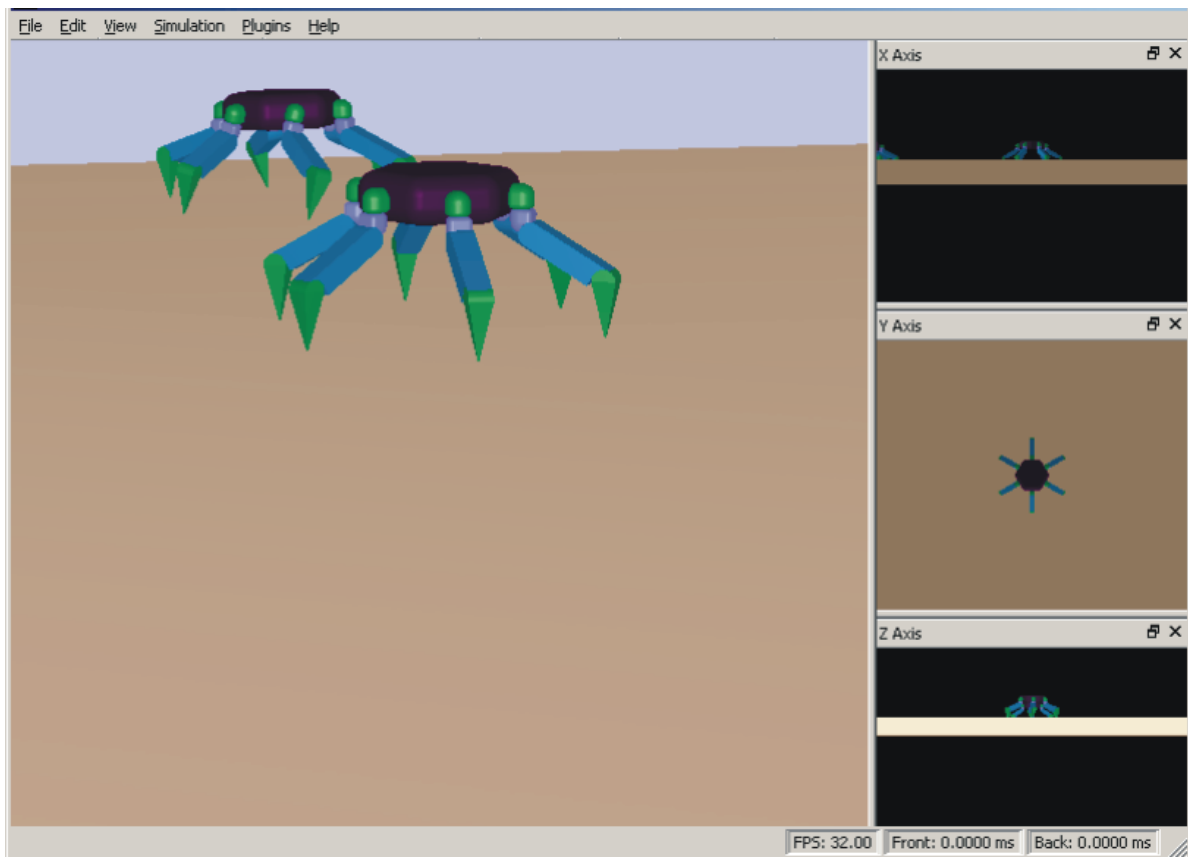
```

/// Creator method for instantiating and setting up camera.
/// \param[in] pWin Pointer to rendering window to share scene with.
/// \param[in] parent Pointer to parent QWidget.
static Camera* newCamera
(
    const EcSGWindow* pWin,
    QWidget* parent = EcNULL
);

```

**Text Box 17-5:** Creator method for instantiating new *EcCamera* objects.

Three instances of the *EcCamera* class are shown below, each one representing a view along one of the major axes. Each sub-window is contained within a Qt *QDockWidget* to attach the window within the confines of the actinViewer window.



**Figure 17-2:** Use of *EcCamera* to create three different views (along X, Y, Z axis) of the main scene.

### 17.5.3 *EcBaseViewerMainWidget*

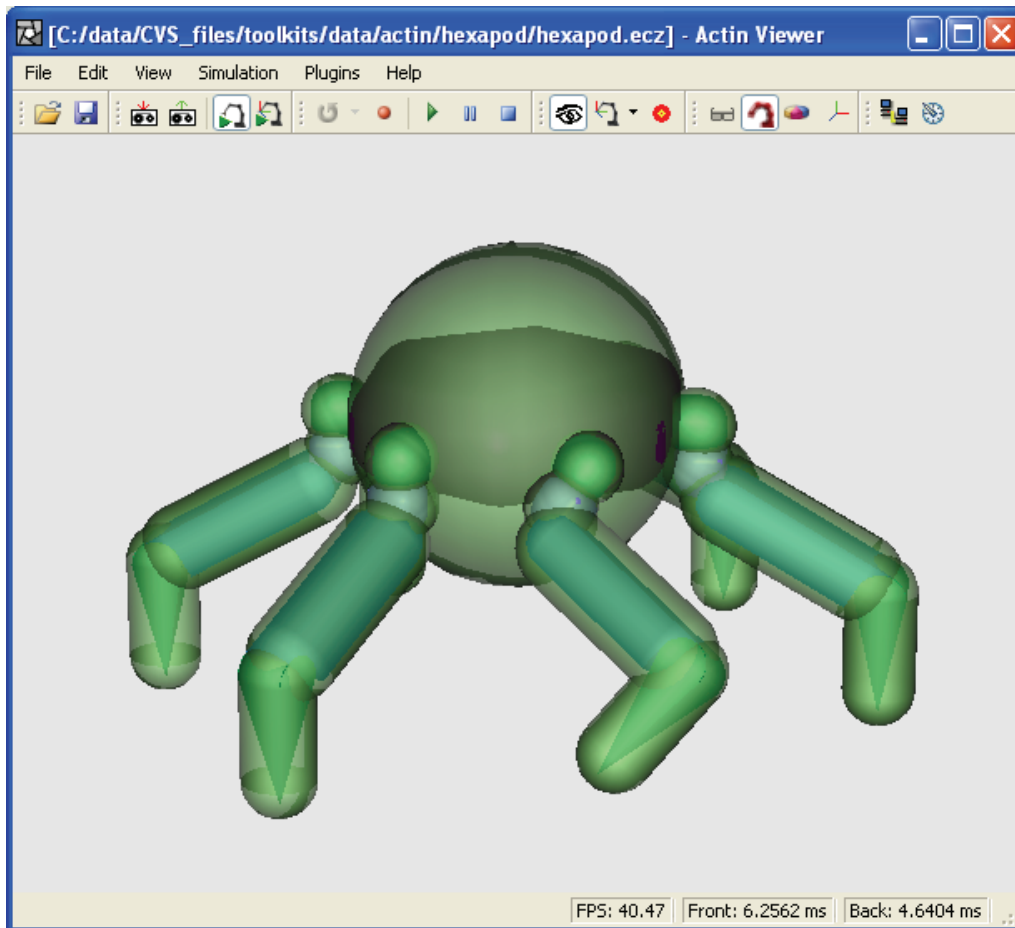
This is a consolidated class that combines the use of an *EcRenderWindow* with *EcSGWidgetQt*. It provides capability for using an *EcSystemSimulation* and provides the main rendering widget for use within an *EcBaseViewerMainWindow*. In addition to methods for loading and accessing simulation classes, a number of Qt signals are defined. These signals are triggered during certain events, such as file loading, simulation time step, or file closing. The *EcPlugin*-derived classes that contain GUI objects use these signals to interact with the viewer application. Table 17-5 summarizes the signals emitted within this class.

Signal	Description
performanceDataReady	Emitted after all rendering is complete. Can be used to update any secondary windows every frame.
signalStartSimulation	Emitted when the user starts running a simulation.
signalStopSimulation	Emitted when the user stops running a simulation.
signalLoadScene	Emitted when the user loads in a new file. Used to initialize variables and state.
signalDeleteContents	Emitted when a file is replaced or when the application is closing. Used to clean up any memory used.

Table 17-5: Signals used within *EcBaseViewerMainWidget*.

### 17.5.4 *EcBaseViewerMainWindow*

Based off of the Qt widget class *QMainWindow*, *EcBaseViewerMainWindow* provides the primary user interface window for application-level windows. It provides a consistent framework that is configurable and used for all viewers. It includes a menubar at the top, status bar at the bottom and an *EcBaseViewerMainWidget* used as the central rendering window. Different types of toolbars are also available, each providing convenient access to common tasks. The *actinViewer* application is one example of how this class is used.



**Figure 17-3:** Example of makeup of *EcBaseViewerMainWindow* class. Shown here is the actinViewer application.

## 17.6 User Input

OpenSceneGraph provides a general event queue that buffers up all input events that can be later processed by an event handler. The *EcSGWidgetQt* class, described in Section 17.5.1 provides the mechanism for taking user input and placing it within the OSG event queue.

### 17.6.1 *EcSGBaseInputHandler*

*EcSGBaseInputHandler* is the foundation class that provides the translation of coordinate systems between Actin and OpenSceneGraph. Actin computes viewer information through the use of *EcPovParameters*. This class takes those parameter values and provides the drawing routine with the correct inverse matrix transformation to set the camera eyepoint. No direct user input is used within this class; all processing is done through the use of *EcPovParameters*.

### 17.6.2 *EcDefaultInputHandler*

This class provides the interaction of mouse and keyboard input that is used with *EcBaseViewerMainWindow*. This class uses the current viewing mode along with keyboard modifiers to specify the type of operation desired. It then computes new information that gets fed back to the rendering window. Currently there are three viewing modes handled: eyepoint, center of interest, and guide frame mode.



## 18 Plugin Interfaces

### 18.1 Control Plugins

The Actin™ Toolkit provides three mechanisms for giving the user control:

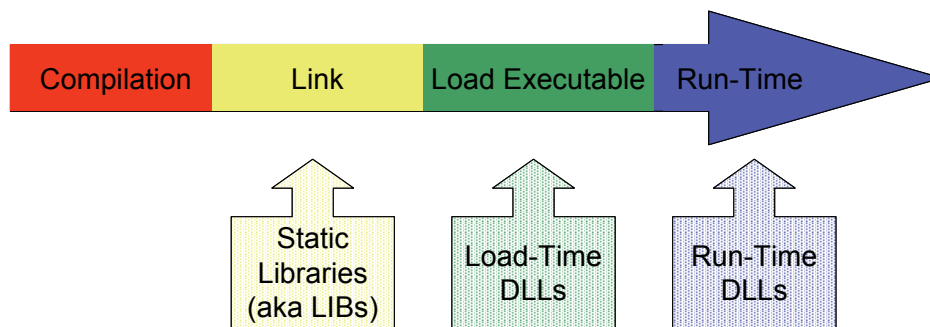
- 1) The XML configuration file gives the user tremendous flexibility in controlling the toolkit. Virtually all of the parameters of the toolkit are set through the configuration file. Also, the configuration file provides the mechanism for polymorphically defining the containers such as *EcBaseExpressionTreeContainer*.
- 2) Inheritance enables the user to use the full power of C++ to configure the toolkit. Almost all of the classes in the toolkit were constructed to support inheritance. Most methods are declared as virtual which enables the developer to override the methods.
- 3) The classes derived from *EcXmlBaseVariableCompoundType* (i.e., *EcBaseExpressionTreeContainer*, *EcXmlVariableElementType*, and *EcXmlVariableVectorType*) enable the user to load a dynamic library (e.g., DLL) through its XML description. New algorithms can be added to the expression tree containers through the plugin interface.

Items 1 and 2 have been addressed throughout this guide. Item 3 will be explained in this section.

The expression tree for the control algorithms provides a useful example for illustrating the power of the plugin interface. The control expression tree has many expression elements that a user can apply to build a control system. For example, plus '+', minus '-', assignment '=', equality '==', and many others are built into the toolkit. A user, however, may want other expression elements for a specific design that are not supported directly in the software toolkit. No matter how many expressions are added, more can be envisioned.

The plugin API enables the user to add new expression elements using C++ without having to recompile the software toolkit or other code built from it. This saves the developer from having to subclass a sequence of classes in the toolkit. With the plugin API, the user has a more flexible tool for inventing control algorithms.

The Microsoft Windows environment provides two types of libraries for plugin support: load-time and run-time dynamic link libraries (DLLs). Load-time DLLs are linked to the software toolkit when the toolkit executable is loaded for execution. Run-time DLLs are linked when the executable requests the linkage. The timing of library linkage is illustrated in Figure 18-1. Versions of UNIX can be configured to operate similarly with ".so" files.



**Figure 18-1:** Timing of library linkage.

The toolkit uses the run-time approach, because it provides more flexibility to the user. For example, the run-time dynamic library can be selectable through the software toolkit XML user interface (see Text Box 18-1). Also, if the user does not need any extra expression elements (i.e., no dynamic library exists), the software toolkit can bypass the linkage altogether. Both of these features are easy to support through a run-time dynamic library.

```
<addLibraries>
  <group>DLL1 DLL2 DLL3</group>
</addLibraries>
```

**Text Box 18-1:** XML configuration of dynamic libraries. Any *EcXmlBaseVariableCompoundType* object can use this syntax to load dynamic libraries. This example loads three.

To link properly to the software toolkit, each dynamic library needs to provide accessor functions for getting information about the user-defined expression elements. The software toolkit creates (i.e., instantiates) expression elements when directed by the user through the XML user interface. To enable this capability, the XML infrastructure needs a token (string name) and a creator function (see Text Box 6-16).

Each token and creator function is defined in each expression element class, and is registered by the expression element container class for use by the XML user interface. To enable the run-time dynamic library support, the library needs to provide access to a token and creator function for each user-defined expression element. The software toolkit provides a dynamic-library framework for this access. To add a new expression element, the user needs to perform three tasks:

- 1) Create the new expression element. The current dynamic library available with the software toolkit provides a modulus ‘%’ expression example, which is contained in “ecExpModulus.cpp” and “ecExpModulus.h.” The user can use these files as a starting point for creating a new expression element. For example, the two primary changes needed to convert the modulus expression to a new expression are 1) performing a search and replace on “Modulus” throughout the file and 2) updating the “value” method to perform the purpose of the new expression element.
- 2) Register the class token and creator functions in “ecExpAccessors.cpp.” The example logic for the modulus expression is easy to duplicate. For example, the current code for the creator registration is:

```
switch (index)
{
  case 0:
    return EcExpressionModulus::creator;
  default:
    return 0;
}
```

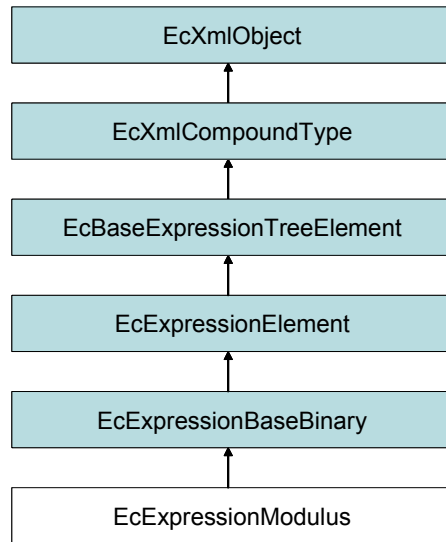
**Text Box 18-2:** Illustration of logic for selecting a creator method. New “case” branches can be added for new expression elements.

A new expression element can be registered by adding another case (e.g., case 1) to the switch.

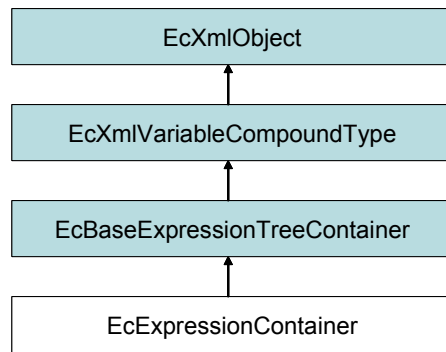


- 3) Update the MS Visual Studio project with the new expression element implementation files and recompile.

The class diagram for the modulus operator is shown in Figure 18-2. All of the code for this dynamic library is available to the user. For completeness, Figure 18-3 shows the class diagram for the expression tree container that enables the loading of the library.



**Figure 18-2:** Example expression element.



**Figure 18-3:** Expression tree container.

Although this section uses an example pertinent to the control expression tree, the ideas are applicable to any container derived from *EcXmlBaseVariableCompoundType*. See Section 6.2.4.1 for more information on *EcXmlBaseVariableCompoundType*.

## 18.2 Interface Plugins

### 18.2.1 *EcPlugin*

Extensibility within Actin is typically desired without having to change the basic framework. The *EcPlugin* class has been designed to provide hooks into the simulation and user interface framework to customize and tailor behavior to suit different needs. External developers can write a Plugin based off of this class without any inherent knowledge of the Actin toolkit.

Method	Description
<code>pluginName</code>	Required: A unique name is required to be defined for lookup and comparison.
<code>setJointCommands</code>	This method gets called each time step with the current time as well as joint information.
<code>getJointState</code>	Called once to initialize simulation with data from <i>EcPlugin</i> -derived class.
<code>init</code>	Any Plugin that creates or modifies GUI components should override this method to initialize their Qt components.
<code>reset</code>	Called when a new file is loaded. Any memory cleanup or re-init should be done here.

**Table 18-1:** *EcPlugin* methods. Aside from `pluginName`, the default for each command is to simply return `EcTrue`. `pluginName` returns a `const char*` to a unique name for the class.

### 18.2.2 *EcPluginManager*

Plugins are shared libraries loaded in at runtime to provide additional functionality on an as-needed basis. *EcPluginManager* is a static class that handles the loading and unloading of *EcPlugin*-based libraries. It keeps track of all currently loaded libraries so that duplicate entries are not created.

### 18.2.3 *Viewer Plugins*

Viewer plugins are available to provide capability to fill a wide variety of needs. They are self-contained modules that can be loaded in as needed. Only the components desired need to be loaded, saving application memory and keeping the viewer from being unnecessarily cluttered with unused functionality. All of them are based upon the *EcPlugin* class (see section 18.2.1 *EcPlugin*)

#### 18.2.3.1 *EcActuatorPlugin*

For use in robot design, Actin includes an actuator-related plugin to assist robot designers when trying to size motors and gearheads for a given manipulator. For each joint actuator, the user can select a motor and chain gearheads in series to be used in the actuator from a database through GUI. This allows the designer to quickly and easily change the actuator components of any actuator and rerun the simulation to see if the new components meet the requirements.

#### 18.2.3.2 *EcCameraPlugin*

Actin provides camera and other raster-data sensors through a plugin. This is implemented through the class *EcCameraPlugin*

### 18.2.3.3 EcCentroidPlugin

Plotting of the center of mass and the zero moment point (ZMP) in Actin Viewer is enabled through the plugin interface.

### 18.2.3.4 EcDataPlotPlugin

Data plotting in the Actin Viewer is also provided through the plugin interface. In our design, there are three main components in capturing and saving simulation information (excluding displaying it in a plot). The first is a collection of data capture classes systematically organized to capture the every bit of information that the user desires for the whole system. The information captured by these data capture classes can then be stored in a data storage class, ready to be saved to files. Finally, the output writer classes save the information stored in the storage to files in the formats chosen by the user.

### 18.2.3.5 EcLogPlugin

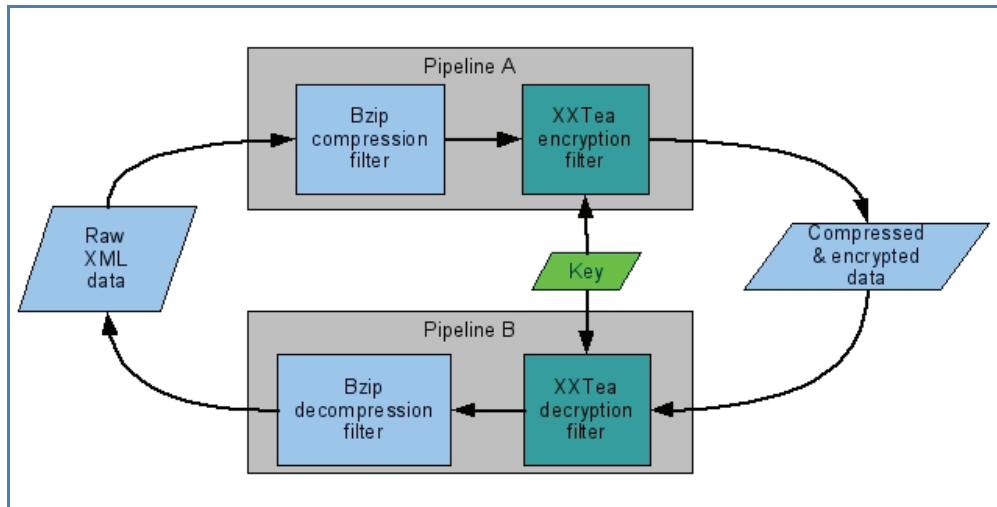
This *EcPlugin* provides a way to view status messages and logging when running the viewer. Most times it is not necessary, but it can be useful if a problem is encountered or when detailed information is desired. By default it will attach itself to the bottom section of the viewer, but can be detached as a floating window as well. It can be turned on or off from the View menu. An example of a detached log window is shown below.



**Figure 18-4:** Floating EcLogPlugin window. It is useful to use this Plugin to determine if there are any problems with simulation files. Here a file gets loaded, but a reference to an image file cannot be found.

## 19 Filters

The efficiency of network operations (especially wireless networks) can be dramatically improved using data compression. Encryption is also prudent when operating in an unknown networking environment. To provide efficient and secure operations, a generic infrastructure is provided that implements the concept of a filtering pipeline. A pipeline is also known as the design pattern “pipes and filters” [34], and consists of a chain of processing elements arranged so that the output of each element is the input to the next. Each element is a filter, in that it analyzes and transforms (i.e. it “*filters*”) its input data stream to produce its output stream. A filtering pipeline may operate on any standard C++ stream, including files on disc and in-memory streams such as strings. Energid’s implementation leverages the work in the Boost libraries- a set of open source C++ libraries that are well respected, heavily tested, and continuously improved.



**Figure 19-1:** Two filtering pipelines

A filtering pipeline allows the user to select an ordered series of one or more transformations to be performed on a stream of data. For example, process A may wish to take a stream of raw XML data, compress it using the bzip2 algorithm then encrypt it using XXTEA and its key. The flow of data through the pipelines is illustrated in Figure 19-1, showing A's pipeline that streams data through the compression filter and encryption filter resulting in the final stream. Process B inverts A's pipeline to retrieve the original raw data.

Three compression (bzip2, zlib, gzip), one encryption (TEA), and one encoding filter (base64) are provided. The list of classes that implement filters and filtering streams are shown in Figure 19-2, and are described in the following sections.

<b>Ec::EcBase64Filter</b>	<i>The base64 filter</i>
<b>Ec::EcFilters</b>	<i>A complete filter specification</i>
<b>Ec::EcFilters::EcFilterArg</b>	<i>Hold the args to a single filter</i>
<b>Ec::EcFilterStream</b>	<i>The high-level class invoking a given set of filters on an input stream</i>
<b>Ec::EcTeaBase</b>	<i>The base class for the TEA cipher</i>
<b>Ec::EcTeaFilter</b>	<i>The XXTea cipher</i>
<b>EcTeaManipulators</b>	<i>Utility class for tea cipher</i>
<b>Ec::EcXXTea</b>	<i>The XXTea cipher</i>

**Figure 19-2:** Class list.

## 19.1 Filter Enumeration

The currently supported filters are listed in the FILTER enumeration found in the `Ec::EcFilters.h` header file. The filters are listed in Figure 19-3.

```
enum FILTER
{
    RAW
    , BZIP2_COMPRESS, BZIP2_DECOMPRESS
    , ZLIB_COMPRESS, ZLIB_DECOMPRESS
    , GZIP_COMPRESS, GZIP_DECOMPRESS
    , BASE64_ENCODE, BASE64_DECODE
    , RSA_ENCRYPT, RSA_DECRYPT
    , TEA_ENCRYPT, TEA_DECRYPT
};
```

**Figure 19-3:** Filters enumeration.

New filters may be added as necessary to the library. Currently, the RSA filters are just placeholders and their attempted use will fail.

## 19.2 Filter Arguments Class

Some filters may require arguments. Encryption filters typically need one or two keys, for example. The *EcFilterArg* class stores arguments to exactly one filter in the filtering pipeline. The class reference is shown in Figure 19-4. The current implementation is specific to the available filters and their arguments.

Public Member Functions	
	<b>EcFilterArg</b> (const <b>FILTER</b> &aFilter) <i>ctor</i>
	<b>EcFilterArg</b> ( <b>FILTER</b> aFilter, const <b>KEY_TYPE</b> &aPublicKey) <i>ctor</i>
	<b>EcFilterArg</b> ( <b>FILTER</b> aFilter, const <b>KEY_TYPE</b> &aPublicKey, const <b>KEY_TYPE</b> &aPrivateKey) <i>ctor</i>
virtual	<b>~EcFilterArg</b> () <i>dtor</i>
	<b>EcFilterArg</b> (const <b>EcFilterArg</b> &rhs) <i>copy ctor</i>
<b>EcFilterArg</b> &	<b>operator=</b> (const <b>EcFilterArg</b> &rhs) <i>assignment operator</i>
Public Attributes	
<b>KEY_TYPE</b>	<b>m_PublicKey</b> > <i>the filter specifier</i>
<b>KEY_TYPE</b>	<b>m_PrivateKey</b> > <i>public key used if filter specifier is a cipher</i>

**Figure 19-4:** *EcFilterArg* class reference.

### 19.3 Filters Class

EcFilters is a convenience class for holding the ordered list of filters (a “filter spec”) to be applied to an f input stream. It has methods for adding each available filter and any filter-specific data that is required. See Figure 19-5 below. The user would create an instance of EcFilters, and call its various methods *in order* to append the filters of choice. For example, calling addBzip2Compress() would add the bzip compression filter to the end of the current filter spec. Of note in the class reference for EcFilters is the convenience method “invert”. This method creates a new filter spec to transform data back into its original form. As example, the invert method would convert the filter spec containing “bzip2, TeaEncrypt” into the inverse filter spec “TeaDecrypt,bzip2Decompress”.

Public Types	
enum	<b>FILTER</b> <i>filter enumeration</i>
Public Member Functions	
	<b>EcFilters</b> () <i>ctor</i>
	<b>EcFilters</b> (const FILTER_ARGS &newArgs) <i>ctor</i>
void	<b>addRaw</b> () <i>addRaw</i>
void	<b>addBzip2Compress</b> () <i>addBzip2Compress</i>
void	<b>addBzip2Decompress</b> () <i>addBzip2Decompression</i>
void	<b>addZlibCompress</b> () <i>addZlibCompress</i>
void	<b>addZlibDecompress</b> () <i>addZlibDecompress</i>
void	<b>addGzipCompress</b> () <i>addGzipCompress</i>
void	<b>addGzipDecompress</b> () <i>addGzipDecompress</i>
void	<b>addBase64Encode</b> () <i>addBase64Encode</i>
void	<b>addBase64Decode</b> () <i>addBase64Decode</i>
void	<b>addTeaEncrypt</b> (const KEY_TYPE &aPublicKey) <i>addTeaEncrypt</i>
void	<b>addTeaDecrypt</b> (const KEY_TYPE &aPublicKey) <i>addTeaDecrypt</i>
void	<b>addRsaEncrypt</b> (const KEY_TYPE &aPublicKey, const KEY_TYPE &aPrivateKey) <i>addRsaEncrypt</i>
void	<b>addRsaDecrypt</b> (const KEY_TYPE &aPublicKey, const KEY_TYPE &aPrivateKey) <i>addRsaDecrypt</i>
FILTER_ARGS	<b>invert</b> () const <i>invert</i>
const FILTER_ARGS &	<b>getFilterArgs</b> () const <i>getFilterArgs</i>
EcBoolean	<b>empty</b> () const <i>determine if filters have been defined</i>

**Figure 19-5:** *EcFilters* class reference.

## 19.4 Filter Stream Class

This generic filtering stream pipeline infrastructure has the added benefit of allowing new filters and streams to be readily added in the future; possibilities include UDP, TCP, and HTTP socket streams. As built, the filter stream class performs filter operations on various combinations of in streams, buffers and strings on input or output. The complete *EcFilterStream* class is detailed in Figure 19-6.

Public Types	
typedef boost::iostreams::filtering_ostream	<b>outFilterStream</b>
Public Member Functions	
	<b>EcFilterStream</b> () <i>ctor</i>
virtual	<b>~EcFilterStream</b> () <i>dtor</i>
	<b>EcFilterStream</b> (const <b>EcFilterStream</b> &rhs) <i>copy ctor</i>
<b>EcFilterStream</b> &	<b>operator=</b> (const <b>EcFilterStream</b> &rhs) <i>assignment operator</i>
virtual EcBoolean	<b>filterInput</b> (const <b>EcFilters</b> &filters, std::istream &in, std::ostream &out) const <i>filterInput</i>
virtual EcBoolean	<b>filterInput</b> (const <b>EcFilters</b> &filters, const EcString &in, EcString &out) const <i>filterInput</i>
virtual EcBoolean	<b>filterInput</b> (const <b>EcFilters</b> &filters, std::istream &in, EcString &out) const <i>filterInput</i>
virtual EcBoolean	<b>filterInput</b> (const <b>EcFilters</b> &filters, const EcString &in, std::ostream &out) const <i>filterInput</i>
virtual EcBoolean	<b>filterOutput</b> (const <b>EcFilters</b> &filters, std::istream &in, std::ostream &out) const <i>filterOutput</i>
virtual EcBoolean	<b>filterOutput</b> (const <b>EcFilters</b> &filters, const EcString &in, EcString &out) const <i>filterOutput</i>
virtual EcBoolean	<b>filterOutput</b> (const <b>EcFilters</b> &filters, std::istream &in, EcString &out) const <i>filterOutput</i>
virtual EcBoolean	<b>filterOutput</b> (const <b>EcFilters</b> &filters, const EcString &in, std::ostream &out) const <i>filterOutput</i>
virtual EcBoolean	<b>stream2Stream</b> (const <b>EcFilters</b> &inFilters, const <b>EcFilters</b> &outFilters, std::istream &in, std::ostream &out) const <i>stream2Stream</i>
virtual EcBoolean	<b>buffer2Buffer</b> (const <b>EcFilters</b> &inFilters, const <b>EcFilters</b> &outFilters, const EcString &in, EcString &out) const <i>buffer2Buffer</i>
virtual EcBoolean	<b>stream2Buffer</b> (const <b>EcFilters</b> &inFilters, const <b>EcFilters</b> &outFilters, std::istream &in, EcString &out) const <i>stream2Buffer</i>
virtual EcBoolean	<b>buffer2Stream</b> (const <b>EcFilters</b> &inFilters, const <b>EcFilters</b> &outFilters, const EcString &in, std::ostream &out) const <i>buffer2Stream</i>
virtual EcBoolean	<b>getWriterStream</b> (const <b>EcFilters</b> &filters, std::ostream &os, <b>outFilterStream</b> &theOutStream) const <i>getWriterStream</i>
virtual EcBoolean	<b>getReaderStream</b> (const <b>EcFilters</b> &filters, std::istream &is, inFilterStream &theInStream) const <i>getReaderStream</i>

**Figure 19-6:** *EcFilterStream* class reference.

The filterInput methods apply the filter spec to an input stream and write the filtered sequence to the output stream.

The process of creating and using a filtering stream is straightforward and consists of a few simple steps as shown in the example in Figure 19-7.



```

EcFilters myFilters;
myFilters.addBzip2Compress();
myFilters.addTeaEncrypt(key);
std::ifstream myInputFile("rawFile.xml",std::ios_base::in|std::ios_base::binary);
std::ofstream myOutputFile("filteredOutput.bz2.tea",std::ios_base::out|std::ios_base::binary);
EcFilterStream myFilterStream;
myFilterStream.filterInput(myFilters,myInputFile,myOutputFile);
myInputFile.close();
myOutputFile.close();

```

**Figure 19-7:** Sample code creating a filtering stream with bzip2 compression and TEA encryption.

## 19.5 Filter Inheritance

Some filters in the Energid toolkits (base64 and XXTEA) derive from the abstract base class “aggregate\_filter“ in the Boost Iostreams library. The aggregate\_filter synopsis [35] is shown in Figure 19-8. Note the private virtual, abstract method “do\_filter”. The base64 and XXTEA filters must inherit from the aggregate\_filter class and provide a concrete implementation of this method to fulfill the interface obligation.

```

namespace boost { namespace iostreams {
#include <vector>
template< typename Ch,
          typename Tr = std::char_traits<Ch>,
          typename Alloc = std::allocator<Ch> >
class aggregate_filter {
public:
    typedef Ch char_type;
    typedef implementation_defined category;
    typedef std::vector<Char, Alloc> vector_type;
    virtual ~aggregate_filter();
    ...
private:
    virtual void do_filter(const vector_type& src, vector_type& dest) = 0;
};
} } // End namespace boost::io

```

**Figure 19-8:** aggregate\_filter class reference.

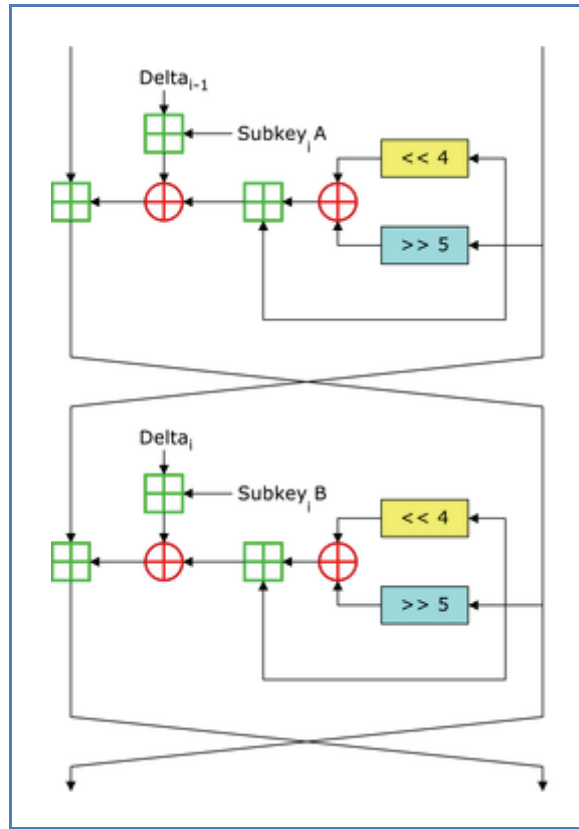
In the following sections, implementation details are provided for the TEA and base64 filters.

## 19.6 TEA Data Encryption Filter Details

Encrypting data before storing it or sending it out over the internet is a critical feature. For real-time performance with minimal impact on computing resources including memory and CPU, a corrected derivative of the Tiny Encryption Algorithm (“TEA”) known as XXTEA has been implemented [36]and made available as another filter in the stream pipeline.

XXTEA operates on data in 64-bit chunks using 128 bits of key divided into two sub-keys. In XXTEA, the number of rounds is calculated as a function of the length of the clear text, with a

minimum of six rounds (determined by the cryptanalysis). XXTEA is a Feistel block cipher like the accepted Data Encryption Standard, with the structural property that encryption and decryption operations are very similar [37]. XXTEA combines multiple rounds of repeated operations including shifts, XORs, and additions as shown in Figure 19-9.



**Figure 19-9:** XXTEA circuit diagram showing two Feistel Rounds [38].

The class *EcTeaFilter* implements the high-level interface used in the filtering streams, and defines the aforementioned `do_filter` method. This class holds a pointer to an instance of the base class *EcTeaBase*, which is refined by the derived class *EcXXTea*. The *EcXXTea* class is the concrete implementation of the corrected TEA algorithm, and provides separate methods for encryption and decryption. The class reference for the XXTEA algorithm is shown in Figure 19-10.

Public Member Functions	
	<b>EcXXTea</b> () <i>default ctor</i>
	<b>EcXXTea</b> (const unsigned char *newKey, const EcU32 newKeyLen) <i>ctor with key</i>
virtual	<b>~EcXXTea</b> () <i>dtor</i>
	<b>EcXXTea</b> (const <b>EcXXTea</b> &rhs) <i>copy ctor</i>
<b>EcXXTea</b> &	<b>operator=</b> (const <b>EcXXTea</b> &rhs) <i>assignment operator</i>
virtual EcBoolean	<b>encrypt</b> (const charVector &src, charVector &dest) <i>encrypt</i>
virtual EcBoolean	<b>decrypt</b> (const charVector &src, charVector &dest) <i>decrypt</i>
<b>EcXXTea</b> *	<b>clone</b> () const <i>copy using copy ctor</i>
<b>EcXXTea</b> *	<b>create</b> () const <i>copy using default ctor</i>
Protected Member Functions	
virtual EcBoolean	<b>crypt_pre</b> () <i>crypt_pre</i>
virtual EcBoolean	<b>crypt_post</b> () <i>crypt_post</i>
virtual EcBoolean	<b>generateSubKey</b> () <i>generateSubKey</i>
virtual EcBoolean	<b>crypt</b> (const charVector &in, charVector &out, const EcBoolean doEncrypt) const <i>crypt</i>
virtual EcBoolean	<b>cryptAll</b> (const charVector &src, charVector &dest, const EcBoolean doEncrypt) <i>cryptAll</i>
Protected Attributes	
ULONGVEC	<b>m_SubKey</b> <i>hold the sub key required by algorithm</i>
<b>EcTeaManipulators</b>	<b>m_Manip</b> <i>utility class for converting between strings and vectors</i>

**Figure 19-10:** The *EcXXTea* class reference

The XXTEA algorithm requires padding for the last block of input data. The PKS padding method has been implemented to provide this functionality. The XXTEA algorithm also requires conversions between the user's character arrays and the algorithm's vectors of floating point. The padding and conversion methods are provided in the utility class *EcTeaManipulators*.

## 19.7 Base 64 Encoding Filter Details

For integration with Energid's GUI program "Actin Viewer" and the Skype network transport, a new filter was necessary, since the Skype API requires that all network messaging be free of embedded null (0x00) characters. The base64 encoding scheme meets this requirement nicely.

The base64 encoding method, as defined in RFC 1421 [39], uses a 64-character alphabet consisting of [a-z,A-Z,0-9,+,/]. This alphabet results in a "printable encoding" scheme. Base64 transforms an arbitrary sequence of 8-bit bytes into 7-bit ASCII text characters. Each three bytes of the original data are divided into four 6-bit blocks that are represented by four 7-bit ASCII characters mapped to the alphabet above. The encoded data is typically 33% larger than the original since 3 bytes are converted into 4. The encoding algorithm is outlined in Figure 19-11.

1. Divide the original data into blocks of 3 bytes
2. Divide the 24 bits in each 3-byte block into 4 groups of 6 bits
3. Map each group of 6 bits into 1 ASCII character in the bas64 alphabet
4. If the last 3-byte block has only 1 byte of original data, pad 2 bytes of zero. After encoding it normally as in step 3, replace the last 2 characters with 2 equal signs "==".
5. If the last 3-byte block has 2 bytes of original data, pad 1 bytes with zero, encode it normally, then replace the last byte with 1 equal sign "=".

**Figure 19-11:** Base64 Encoding Algorithm

The class reference for the Base64 filter is shown in Figure 19-12. Of particular note are the methods `do_filter`, `encode` and `decode`. The concrete method `do_filter` method calls `encode` or `decode` depending on the value of the class variable `m_doEncode`. The `encode()` method converts raw text into base64 and the `decode()` method converts base64 back into raw text.

Public Member Functions	
	<b>EcBase64Filter</b> (const EcBoolean encode=true) <i>default ctor</i>
	<b>EcBase64Filter</b> (const <b>EcBase64Filter</b> &rhs) <i>copy ctor</i>
<b>EcBase64Filter</b> &	<b>operator=</b> (const <b>EcBase64Filter</b> &rhs) <i>assignment operator</i>
virtual	<b>~EcBase64Filter</b> () <i>dtor</i>
virtual EcBoolean	<b>encode</b> (const vector_type &src, vector_type &dest) const <i>encode</i>
virtual EcBoolean	<b>decode</b> (const vector_type &src, vector_type &dest) const <i>decode</i>
Protected Member Functions	
virtual void	<b>do_filter</b> (const vector_type &src, vector_type &dest) <i>do_filter</i>

**Figure 19-12:** *EcBase64Filter* class reference

## 20 Network Operation

Energid provides a set of classes that support TCP (“Transmission Control Protocol”), UDP (“User Datagram Protocol”) and Skype network operations. Network operations allows interactive engineering collaboration, supports engineers wanting to run validation software on fast remote computers while working on a desktop, enables interactive demonstrations to managers or other non-engineers, supports parallelization of simulation runs, and allows teleoperation and control of untethered, autonomous robots.

TCP and UDP support is part of the toolkit. A UDP datagram is an unreliable message limited to 65535 bytes (1400 bytes for Skype UDP). It suffers much better than TCP under unreliable networks since ACKs and retransmits are not involved. UDP datagrams are ideal for small command and control messages of sufficient frequency that missing several in a series is non-critical. For reliable messaging, TCP packets are required. TCP packets are limited to 65535 bytes and may suffer transmission delays across unreliable networks due to packet collisions, window resizing, and timeout/resend cycles.

As discussed in the section on filtering streams, efficient and secure operations are important for network operations. Filtering streams provide compression and encryption options to the user and are integrated with the TCP and UDP classes.

Another impediment to integrated network operation over a WAN (such as the public Internet) is the existence of firewalls and Network Address Translator (NAT) routers that stand between software

components. Energid provides the unique capability to punch through firewalls and NATs to discover software components through its Skype peer-to-peer transport.

## 20.1 Low-level Sockets

### 20.1.1 *Hierarchy*

At the lowest level, sockets provide the TCP and UDP functionality for network operations. Energid's toolkit uses simple inheritance for sockets, as shown in Figure 20-1.

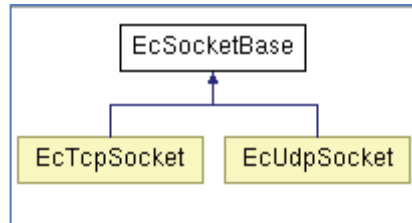


Figure 20-1: Low-level socket hierarchy.

### 20.1.2 *Base Socket Class*

The class `ecSocketBase` is the base class for low-level sockets. The class reference is shown in Figure 20-2. The class is not intended for public consumption, but is provided here for completeness and for insight into its derived classes that are discussed later.

Public Member Functions	
	<b>EcSocketBase</b> (const EcInt32 type, const EcInt32 protocol) <i>constructor from a type and protocol</i>
	<b>EcSocketBase</b> (const EcInt32 socketID) <i>constructor from a socket description</i>
virtual	<b>~EcSocketBase</b> () <i>destructor</i>
	<b>EcSocketBase</b> (const <b>EcSocketBase</b> &rhs) <i>copy constructor</i>
<b>EcSocketBase</b> &	<b>operator=</b> (const <b>EcSocketBase</b> &rhs) <i>assignment operator</i>
EcBoolean	<b>operator==</b> (const <b>EcSocketBase</b> &orig) const <i>equality operator</i>
virtual EcBoolean	<b>bind</b> (const EcU16 localPort) <i>bind to local port</i>
virtual EcString	<b>peerAddress</b> () <i>get the address of the peer (foreign)</i>
virtual EcU16	<b>peerPort</b> () <i>get the port of the peer (foreign)</i>
virtual EcString	<b>localAddress</b> () <i>get function for local address</i>
virtual EcU16	<b>localPort</b> ()
virtual EcInt32	<b>socketID</b> () const <i>get the socket ID</i>
virtual EcBoolean	<b>setRecvTimeout</b> (const EcInt32 timeout) <i>set the timeout for receive in msec.</i>
virtual EcBoolean	<b>setSendTimeout</b> (const EcInt32 timeout) <i>set the send time out in msec.</i>
virtual EcBoolean	<b>setSocketToWait</b> (const EcBoolean wait)
virtual EcU32	<b>getNumPendingBytes</b> () <i>Return the # of pending bytes to read.</i>
virtual EcU32	<b>getLastError</b> () const <i>Return the last socket error that has occurred, or 0 for no errors.</i>

**Figure 20-2:** *EcSocketBase* class reference.

Of interest in the class is the anonymous enum for socket error codes. At this level in the hierarchy, several standard socket error codes have been combined to simplify and abstract the details. They are:

- **ECSOCK\_ERROR\_GENERIC**  
Likely an illegal parameter or calling functions out of order
- **ECSOCK\_ERROR\_NO\_BROADCAST\_ACCESS**  
Trying to broadcast without enabling broadcasting
- **ECSOCK\_ERROR\_NOT\_CONN**  
Socket is not currently connected
- **ECSOCK\_WOULD\_BLOCK**  
Operation would block, or it has timed out

- `ECSOCK_HOST_UNREACHABLE`  
Cannot communicate with the specified remote address
- `ECSOCK_CONN_LOST`  
The socket connection has been interrupted
- `ECSOCK_NETWORK_DOWN`  
The network is down
- `ECSOCK_ADDR_IN_USE`  
The specified address is already in use

Also of note is the requirement that every socket message contain a header of information that serves two purposes: uniquely identifies the message and explicitly states the number of bytes in the data payload of the message.

```

// Prepare data
static const EcU32 TEST_DATA_SIZE = 10000;
EcReal doubleArray[TEST_DATA_SIZE];
EcU32 ii;
for(ii=0;ii<TEST_DATA_SIZE; ++ii)
{
    doubleArray[ii] = ii;
}

// Put size in header
EcSocketHeader sHeader;

```

### 20.1.3 *Tcp Socket*

The class reference for Transmission Control Protocol sockets (derived from `ecSocketBase`) is shown in Figure 20-3. The TCP protocol provides reliable, ordered delivery of the data payload. This form of communication is typically used by the toolkit when sending XML configuration data.

Tcp sockets follow the standard sequences for behaving as clients or servers.



Public Member Functions	
	<b>EcTcpSocket</b> () <i>constructor</i>
	<b>EcTcpSocket</b> (const EcString &peerAddress, EcU16 peerPort, EcBoolean &ifConnect) <i>constructor which creates connects to peer address and port</i>
	<b>EcTcpSocket</b> (EcInt32 socketID) <i>constructor with a previously opened socket ID</i>
virtual	<b>~EcTcpSocket</b> () <i>destructor</i>
	<b>EcTcpSocket</b> (const <b>EcTcpSocket</b> &rhs) <i>copy constructor</i>
	<b>EcTcpSocket</b> & <b>operator=</b> (const <b>EcTcpSocket</b> &rhs) <i>assignment operator</i>
virtual EcBoolean	<b>listen</b> (const EcU16 localPort, const EcInt32 queueLength=1) <i>Listen for incoming connection.</i>
virtual <b>EcTcpSocket</b> *	<b>accept</b> (EcBoolean waitFlag=EcTrue)
virtual EcBoolean	<b>connect</b> (const EcString &peerAddress, const EcU16 peerPort) <i>try to connect with the peer address and port</i>
virtual EcInt32	<b>send</b> (const <b>EcSocketHeader</b> &sHeader, const EcInt8 *message) <i>send message (header and data)</i>
virtual EcInt32	<b>send</b> (const EcInt8 *buffer, const EcInt32 bufferSize) <i>send message (header or data)</i>
virtual EcInt32	<b>receive</b> ( <b>EcSocketHeader</b> &sHeader, EcInt8 *&message)
virtual EcInt32	<b>receive</b> ( <b>EcSocketHeader</b> &sHeader, EcInt8 *&message, const EcInt32 socketLimit)
virtual EcInt32	<b>rcv</b> (EcInt8 *buffer, EcInt32 bufferSize, const EcInt32 msgFlag=0)
virtual EcBoolean	<b>selfTest</b> () <i>self test</i>

**Figure 20-3:** *EcTcpSocket* class reference.

For the client, the process generally involves the following events:

- 1) Create a socket
- 2) Connect to server
- 3) Send/Receive data
- 4) Close socket

For the server, the process generally involves:

- 1) Create a socket
- 2) Bind to a port
- 3) Listen for a connection request from client
- 4) Accept connection
- 5) Send/Receive data
- 6) Close socket.

These events are illustrated in Figure 20-4.

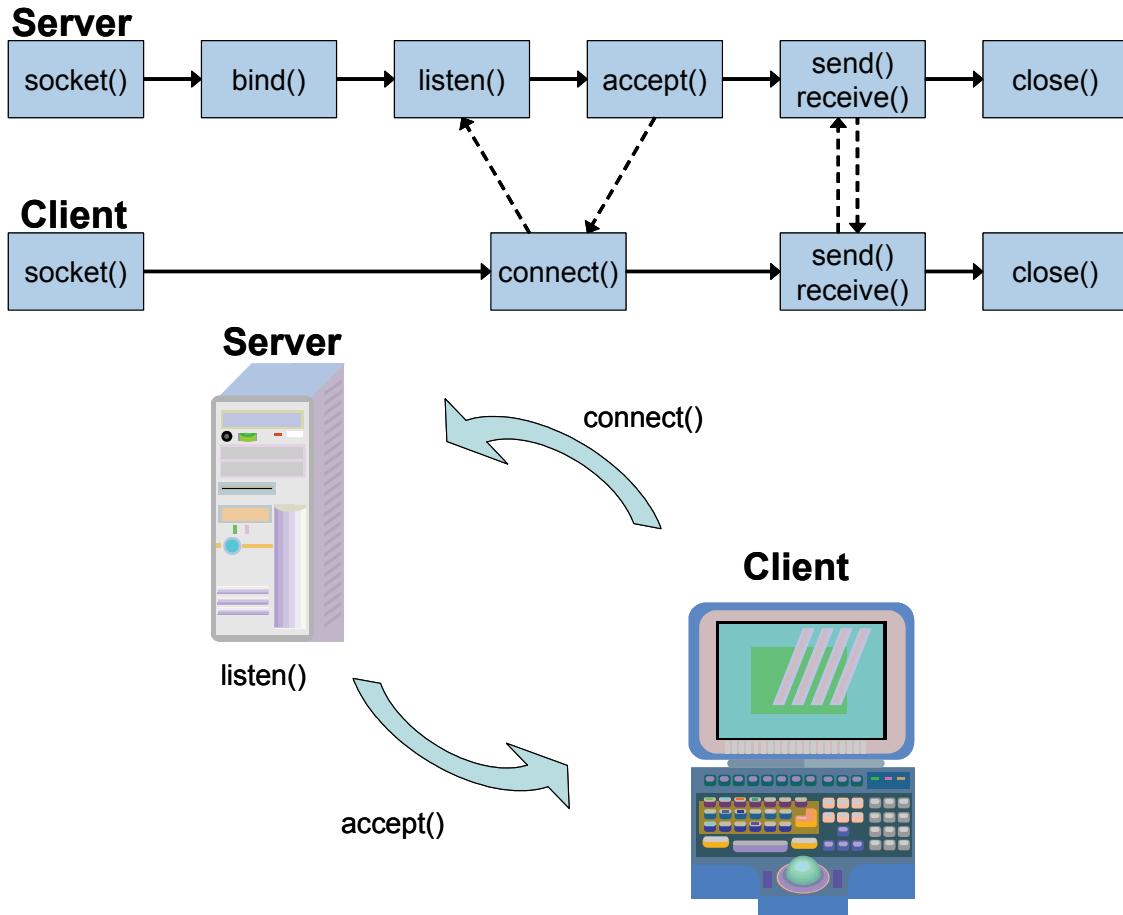


Figure 20-4: TCP client and server events.

### 20.1.4 Udp Socket

The class `ecUdpSocket` is a straight-forward implementation of the User Datagram Protocol. Data sent according to this protocol is known as a datagram. The UDP protocol does not guarantee that a datagram will be delivered in any particular order, and does not guarantee that it will be delivered at all, i.e., the protocol is unreliable. The advantage of UDP over TCP is the overhead efficiency of avoiding the delivery guarantee and state if each packet. This form of communication is typically used for higher-rate data such as streaming video. The received data can be corrupt or out of order. Corrupted data is typically dropped. Out-of-order data is typically handled by preferring “newer” data identified using an embedded timestamp. For the UDP client, the process generally involves the following events:

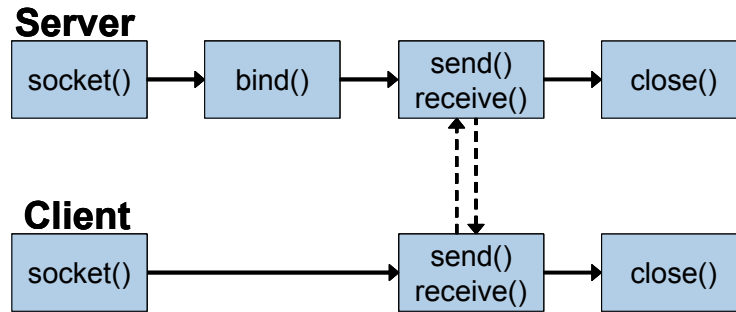
- 1) Create a socket
- 2) Send/Receive data
- 3) Close socket

For the server, the process generally involves:

- 1) Create a socket

- 2) Bind to a port
- 3) Send/Receive data
- 4) Close socket

These events are illustrated in Figure 20-5.



**Figure 20-5:** UDP client/server events.

The class reference is shown in Figure 20-6.

Public Member Functions	
	<b>EcUdpSocket</b> () <i>constructor</i>
virtual	<b>EcUdpSocket</b> (EcU16 localPort) <b>~EcUdpSocket</b> () <i>destructor</i>
	<b>EcUdpSocket</b> (const <b>EcUdpSocket</b> &rhs) <i>copy constructor</i>
	<b>EcUdpSocket</b> & <b>operator=</b> (const <b>EcUdpSocket</b> &rhs) <i>assignment operator</i>
virtual void	<b>setToBroadcast</b> () <i>set this to be a broadcast socket</i>
virtual EcInt32	<b>send</b> (const <b>EcSocketHeader</b> sHeader, const EcInt8 *message, const EcString &peerAddress, const EcU16 peerPort) <i>send message (header and data)</i>
virtual EcInt32	<b>sendTo</b> (const EcInt8 *buffer, const EcInt32 bufferSize, const EcString &peerAddress, const EcU16 peerPort) <i>send message (header or data)</i>
virtual EcInt32	<b>receive</b> ( <b>EcSocketHeader</b> &sHeader, EcInt8 *&message, EcString &peerAddress, EcU16 &peerPort)
virtual EcInt32	<b>recvFrom</b> (EcInt8 *buffer, EcInt32 bufferSize, EcString &peerAddress, EcU16 &peerPort, const EcInt32 msgFlag=0) <i>receive message (header or data)</i>

**Figure 20-6:** UDP class reference.

## 20.2 Mid-Level Classes

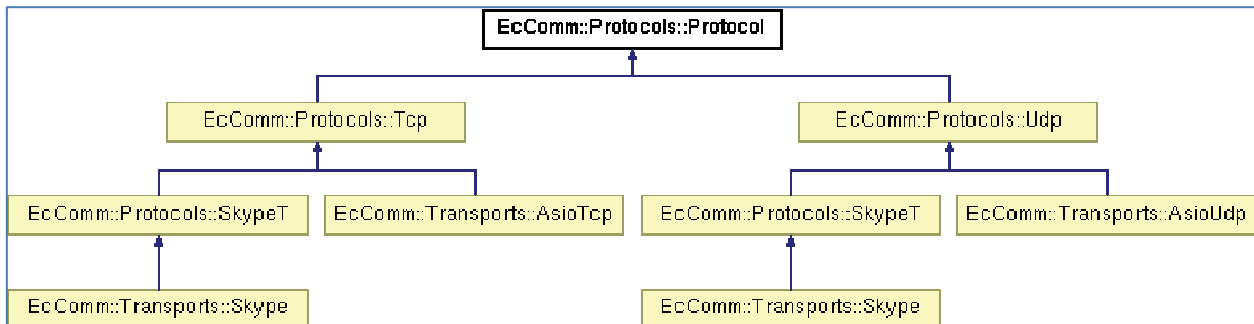
Just above the low-level socket classes is a layer supporting higher-level, typically communications functionality in terms of client-server roles, abstracted transport mechanisms, encryption and compression. The classes of interest are listed in Figure 20-7 with their brief descriptions.

<b>EcComm::Transports::AsioTcp</b>	<i>Tcp implementation using boost ASIO</i>
<b>EcComm::Transports::AsioUdp</b>	<i>Tcp implementation using boost ASIO</i>
<b>EcComm::CommFactory</b>	<i>Create an instance of a transport</i>
<b>CommFactory</b>	<i>Create an instance of an underlying skype implementation (Linux or Windows)</i>
<b>ecDeque</b>	<i>Container for efficient caching of data</i>
<b>EcComm::Protocols::Protocol</b>	<i>Generic abstract base class for a protocol</i>
<b>EcComm::Transports::Skype</b>	<i>Implementation of the skype protocol interface</i>
<b>EcComm::Transports::SkypeImplBase</b>	<b>Skype</b> <i>underlying os-specific implementation abstract base class</i>
<b>EcComm::Protocols::SkypeT</b>	<b>Protocol</b> <i>defining the Skype INTERFACE (both TCP and UDP)</i>
<b>EcComm::Transports::SkypeWinImpl</b>	<i>Concrete implementation for the Windows-specific portion of the skype transport</i>
<b>EcComm::Transports::SkypeX11Impl</b>	<i>Concrete implementation for the Linux-specific portion of the skype transport</i>
<b>EcComm::Protocols::Tcp</b>	<b>Protocol</b> <i>specifying the TCP interface</i>
<b>EcComm::Protocols::Udp</b>	<b>Protocol</b> <i>specifying the UDP interface</i>

**Figure 20-7:** Mid-level class descriptions

### 20.2.1 Hierarchy

The standard methods for interfacing via the TCP and UDP protocols have been implemented as shown in Figure 20-8. The standard methods for connection management, reading and writing are provided. At the top of the illustration, an abstract base class “Protocol” provides methods and data common to both TCP and UDP. In the middle are separate derived classes providing interface methods unique to each protocol. A new protocol called “Skype” was created, which is a hybrid that provides both TCP and UDP-like functionality simultaneously. This is implemented via diamond inheritance and is shown near the bottom of the illustration.



**Figure 20-8:** Mid-level hierarchy.

The TCP implementation is currently implemented by reusing existing ecTcpSocket code. In the future this will be replaced with the boost ASIO networking implementation that should provide improved operations in asynchronous, multi-threaded environments.

### 20.2.2 Protocols

The protocol classes are shown together in UML form in Figure 19-9. Of interest are the data structures and methods for applying filters such as compression and encryption. The user will specify

the filters to use in the communications via the methods “setReadFilters” and “setWriteFilters”. These methods provide asymmetrical filtering such that each end point in the communication may choose to use different or unique sequences of filters.

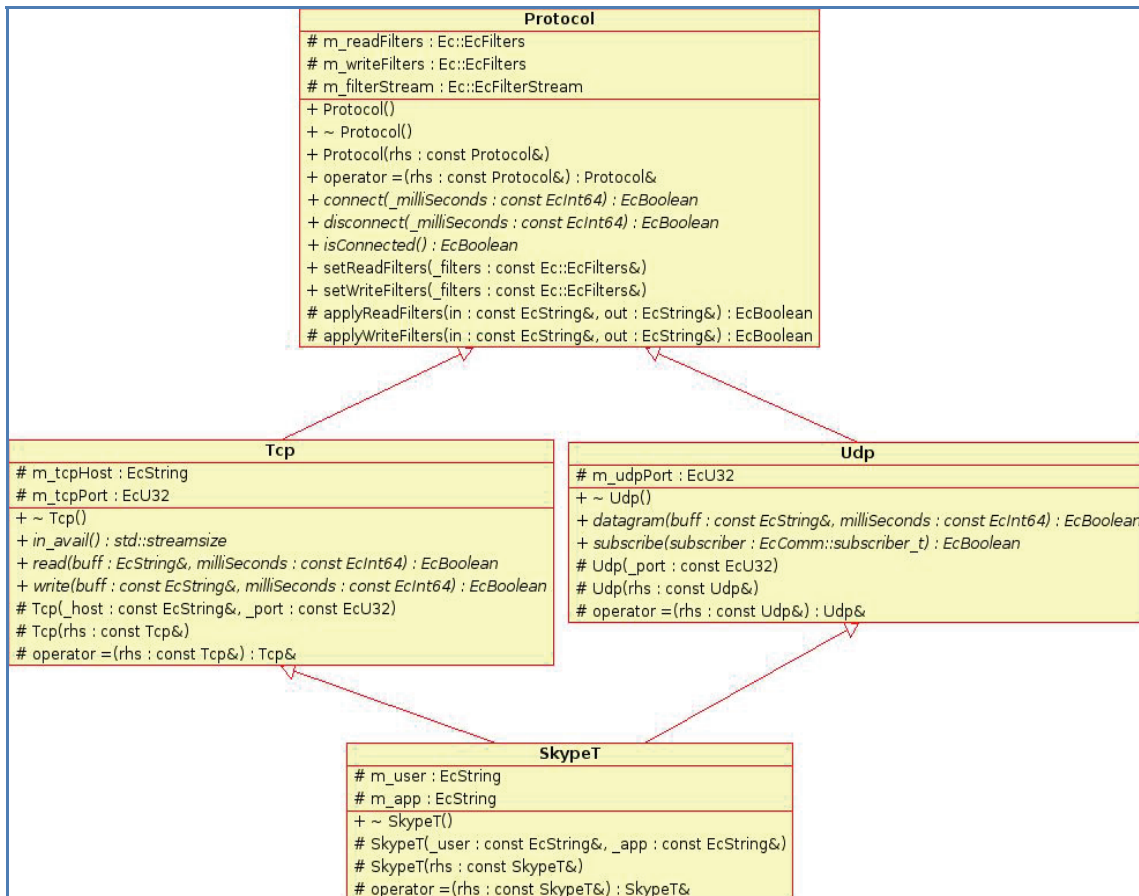


Figure 20-9: Protocol hierarchy

### 20.2.2.1 Protocols::Protocol

Public Member Functions	
	<b>Protocol</b> () <i>default ctor</i>
virtual	<b>~Protocol</b> () <i>dtor</i>
	<b>Protocol</b> (const <b>Protocol</b> &rhs) <i>copy ctor</i>
<b>Protocol</b> &	<b>operator=</b> (const <b>Protocol</b> &rhs) <i>assignment operator</i>
virtual EcBoolean	<b>connect</b> (const EcInt64 _milliseconds=0)=0 <i>connect</i>
virtual EcBoolean	<b>disconnect</b> (const EcInt64 _milliseconds=0)=0 <i>disconnect</i>
virtual EcBoolean	<b>isConnected</b> () const =0 <i>isConnected</i>
virtual void	<b>setReadFilters</b> (const Ec::EcFilters &_filters) <i>setReadFilters</i>
virtual void	<b>setWriteFilters</b> (const Ec::EcFilters &_filters) <i>setWriteFilters</i>
Protected Member Functions	
virtual EcBoolean	<b>applyReadFilters</b> (const EcString &in, EcString &out) const <i>applyReadFilters</i>
virtual EcBoolean	<b>applyWriteFilters</b> (const EcString &in, EcString &out) const <i>applyWriteFilters</i>
Protected Attributes	
Ec::EcFilters	<b>m_writeFilters</b> > <i>the filter spec for data read into the filter</i>
Ec::EcFilterStream	<b>m_filterStream</b> > <i>the filter spec for data written out from the filter</i>

**Figure 20-10:** Class reference for Protocol

As mentioned, the interesting methods are those for setting the filters for reading and writing. In addition to the standard methods as seen in this base class, the protocols support on-the fly data filtering that compresses, decompresses, encrypts and decrypts the stream (see the class reference in Figure 20-10).

As example, to create a sequence of filters that compress and encrypt the outbound data on the protocol, and expect the inbound data to be the same such that the inverse filter operations must be applied, see the code snippet in Figure 20-11. The example code also works the UDP protocol class.

```
// create an instance of the filters
Ec::EcFilters writeFilters;
//base64 encoding will be the first filter operation on outbound data
writeFilters.addBase64Encode();
// zlib compression will be performed after base64
writeFilters.addZlibCompress();

// set the write filters assuming we have a valid instance of
// EcComm::Protocols::Tcp* myTcpClientPtr;
myTcpClientPtr->setWriteFilters(writeFilters);

// we are expecting the other endpoint to use the same filtering
// operations, so set the read filters as the inverse operation of the write
myTcpClientPtr->setReadFilters(writeFilters.invert());
```

**Figure 20-11:** Setting filters for the protocol.

A timeout value may be specified for each of the input/output operations connect, disconnect, read and write for TCP, and datagram for UDP. These timeout units are milliseconds, and the value “-1” meaning is to wait forever for the operation to complete. Beware that infinite timeouts may have undesired consequences should communications be interrupted.

### 20.2.2.2 Protocols::Tcp

The Tcp protocol class is an abstract base class, requiring a concrete implementation as provided in EcComm::Transports::AsioTcp. This class basically adds and exposes the methods for reading and writing to the Protocol base class. The class reference is shown in Figure 20-12.

Public Member Functions	
virtual	<b>~Tcp</b> () <i>dtor</i>
virtual std::streamsize	<b>in_avail</b> ()=0 <i>in_avail</i>
virtual EcBoolean	<b>read</b> (EcString &buff, const EcInt64 milliseconds=-1)=0 <i>read</i>
virtual EcBoolean	<b>write</b> (const EcString &buff, const EcInt64 milliseconds=-1)=0 <i>write</i>
Protected Member Functions	
	<b>Tcp</b> (const EcString &_host="127.0.0.1", const EcU32 _port=9999) <i>ctor</i>
	<b>Tcp</b> (const <b>Tcp</b> &rhs) <i>copy ctor</i>
<b>Tcp</b> &	<b>operator=</b> (const <b>Tcp</b> &rhs) <i>assignment operator</i>
Protected Attributes	
EcU32	<b>m_tcpPort</b> > dotted-quad ip address of remote host to connect to (if client)

**Figure 20-12:** Class reference for Protocols::Tcp.

### 20.2.2.3 Protocols::SkypeT

Network operations can be performed at several levels, from the lowest socket operations to a high-level communications factory that supports communications across various network topographies and through various firewalls and NATs using the peer-to-peer (P2P) protocol Skype.

P2P is a preferred architecture since it is decentralized and without the bottleneck problems associated with traditional client-server networks. P2P connects an ad-hoc association of nodes in which each node provides both client and server functionality in such fashion as to be robust to peer dropouts, bandwidth shortage and network latencies. Skype implements P2P and has proved quite capable across firewall and Network Address Translation (NAT) problems.

Skype provides an Application Programming Interface (API) that allows its functionality to be embedded within a third-party application [40]. Energid’s networking embeds this functionality. The Skype API provides “application to application” messaging in the form of UDP (“User Datagram Protocol”) and TCP (“Transmission Control Protocol”) messages. The ability to choose the transport mechanism is key to efficient real-time control, given how poorly TCP, designed for wired networks) behaves over unreliable networks such as 802.11 wireless.



Currently, Skype requires the API to connect through a Skype client running on both host computers. This inconvenience is being corrected in a forthcoming release of the API. Skype is a proprietary messaging protocol, though the performance benefit may dominate these concerns.

The `Protocols::SkypeT` class is an abstract base class for the Skype protocol. It has a concrete implementation in `EcComm::Transports::Skype` discussed later. It inherits from both `Protocols::Tcp` and `Protocols::Udp` since the Skype API provides both protocols also. The class reference is shown in Figure 20-13.

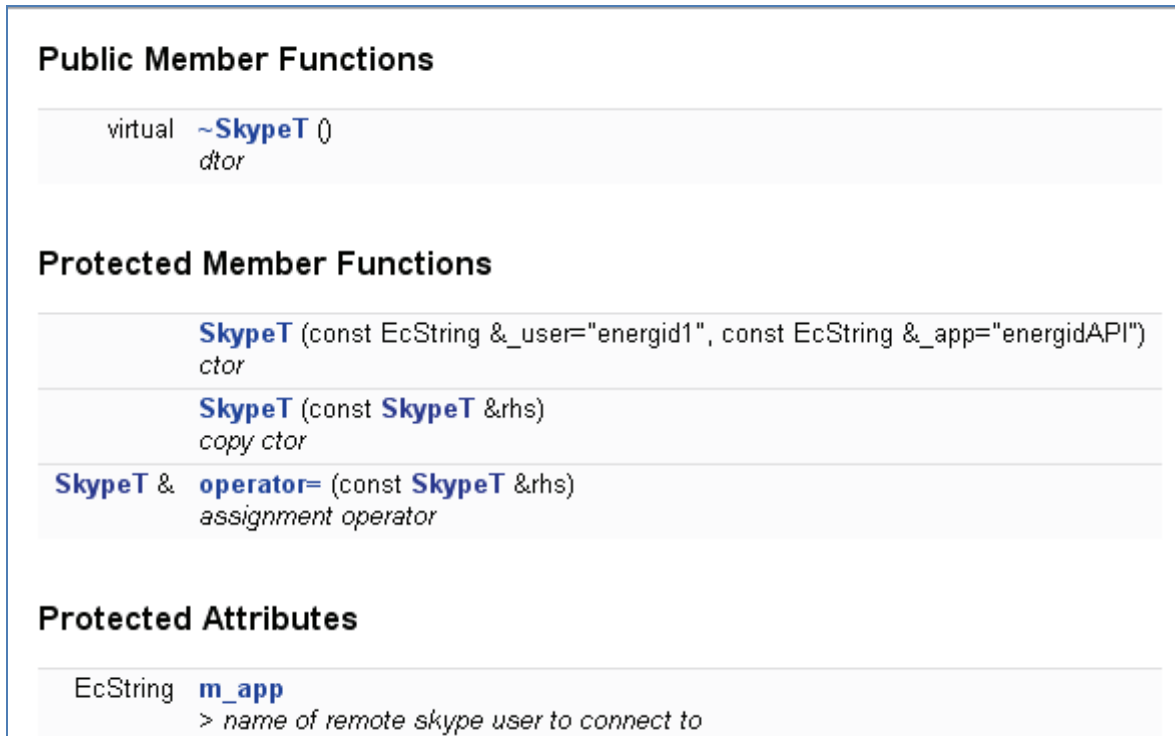


Figure 20-13: Class reference for `Protocols::SkypeT`.

### 20.2.3 Transports

The transport classes provide concrete implementations of the protocols mentioned previously. The transport classes are each discussed in the following sections. The transports add the mechanisms required to support filtering, on top of the low-level socket implementations.

#### 20.2.3.1 AsioTcp

`AsioTcp` is the concrete implementation of the TCP protocol provided in the Energid toolkit. Although the name suggests that the implementation takes advantage of asynchronous input/output, this is not yet the case: the implementation relies on `EcTcpSocket` discussed previously.

Of special note is the constructor flag “beServer”. If set to true, then the connection will behave as a server such that it will bind to a local port and listen for and accept incoming connections. If the flag is false (the default), then the connection will be attempted to the host computer and port. The class reference is shown in Figure 20-14.

The low-level implementation is hidden, replaced with higher-level methods and some convenience methods. The methods and their descriptions are given in Figure 20-15. A complete example is given in Figure 20-23.

Public Member Functions	
	<b>AsioTcp</b> (const EcString &_host="127.0.0.1", const EcU32 _port=9999, const EcBoolean beServer=EcFalse) <i>ctor</i>
virtual	<b>~AsioTcp</b> () <i>dtor</i>
virtual EcBoolean	<b>connect</b> (const EcInt64 _milliSeconds=-1) <i>connect</i>
virtual EcBoolean	<b>disconnect</b> (const EcInt64 _milliSeconds=0) <i>disconnect</i>
virtual EcBoolean	<b>isConnected</b> () const <i>isConnected</i>
virtual std::streamsize	<b>in_avail</b> () <i>in_avail</i>
virtual EcBoolean	<b>read</b> (EcString &buff, const EcInt64 milliSeconds=-1) <i>read</i>
virtual EcBoolean	<b>write</b> (const EcString &buff, const EcInt64 milliSeconds=-1) <i>write</i>

**Figure 20-14:** Class Reference for AsioTcp.

Method	Description
Connect	For the server: bind, listen and accept connections from a client. For the client: initiate connection on remove host and port
Disconnect	Terminate the connection
isConnected	Return true if a connection is established, false otherwise
In_avail	Return the number bytes currently available for reading from the socket
read	Read from the socket
write	Write to the socket

**Figure 20-15:** Primary methods in AsioTcp.

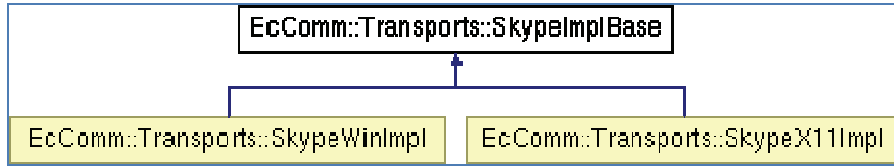
### 20.2.3.2 Skype

The Skype transport provides simultaneous UDP and TCP functionality via the official Skype Application-To-Application API. As a transport, the Skype API is consistent between Windows and Linux. However, at the interface level, the implementations for the two operating systems are quite different.

The windows implementation makes use of the WMCOPYDATA methodology, requiring an *invisible* window for the transport to send and receive events and data from the API. There are two methods in Linux for communicating with the Skype API: the DBUS method and X-Event method.

Energid’s implementation us based on the X-Event method, since the DBUS method burdens the user with extra requirements including additional libraries and correct system configuration.

Though the lowest level implementations are different for Windows and Linux, a major portion of required functionality was implemented in a base class “EcComm::Transports::SkypeImplBase” that the operating system-specific implementations derived from. See Figure 20-16.



**Figure 20-16:** Inheritance diagram for Skype implementation.

Figure 20-17 shows the UML for the above classes and the class reference is given in Figure 20-18. The unique methods are “datagram” and “subscribe”. The datagram method is used for sending a UDP datagram in the form of a string. A timeout value (in milliseconds) may also be specified. The subscribe method is the primary means to receive UDP datagrams and requires that a callback method be created with a specific method signature: `void (const std::string& )`. An example of a callback method and how it is registered is given in Figure 20-19.

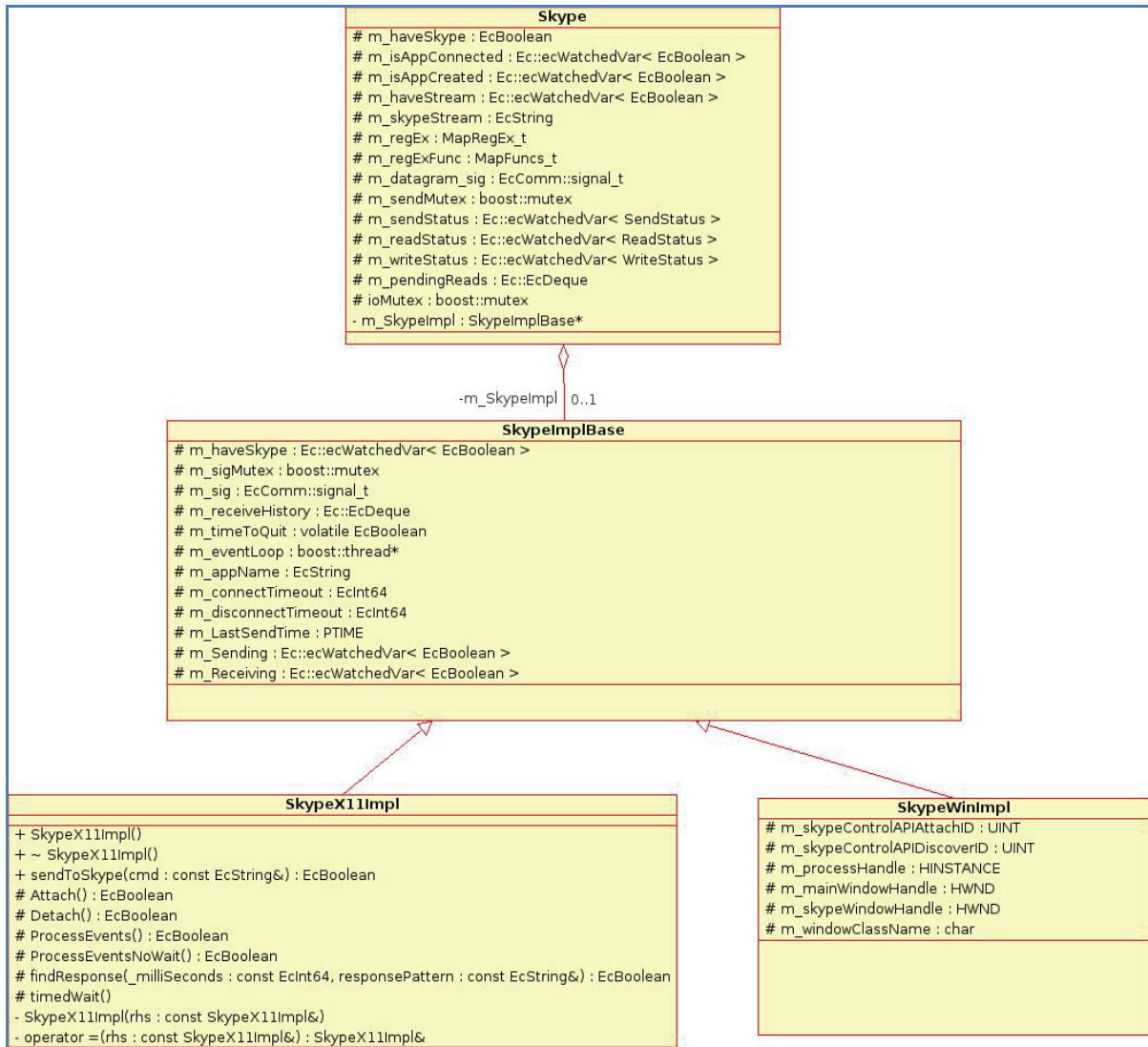


Figure 20-17: UML for the Skype transport implementation.

## Public Member Functions

	<b>Skype</b> (const EcString &_user="energidUSER1", const EcString &_app="energeAPI") <i>ctor</i>
virtual	<b>~Skype</b> () <i>dtor</i>
virtual EcBoolean	<b>connect</b> (const EcInt64 _milliseconds=-1) <i>connect</i>
virtual EcBoolean	<b>disconnect</b> (const EcInt64 _milliseconds=0) <i>disconnect</i>
virtual EcBoolean	<b>isConnected</b> () const <i>isConnected</i>
virtual std::streamsize	<b>in_avail</b> () <i>in_avail</i>
virtual EcBoolean	<b>read</b> (EcString &buff, const EcInt64 milliseconds=-1) <i>read</i>
virtual EcBoolean	<b>write</b> (const EcString &buff, const EcInt64 milliseconds=-1) <i>write</i>
virtual EcBoolean	<b>datagram</b> (const EcString &buff, const EcInt64 milliseconds) <i>datagram</i>
virtual EcBoolean	<b>subscribe</b> (EcComm::subscriber_t subscriber) <i>subscribe</i>

**Figure 20-18:** Class reference for the Skype transport.

```

void echoUdpCallback(const EcString buff)
{
    std::cerr<<"Received UDP echo: len="<<buff.size()<<"data=["<<buff<<']'<<std::endl;
}

int main()
{
    EcString theSkypeUser="mySkypeUserName";
    EcString theSkypeApp="energidAPI";
    Udp* myUdp = CommFactory::createSkype(SKYPE,theSkypeUser,theSkypeApp);
    if ( ! myUdp->connect(theConnectTimeout) )
    {
        EcERROR("connect failed\n");
        return 1;
    }
    if ( ! myUdp->subscribe(echoUdpCallback) )
    {
        EcERROR("subscribe udp callback failed\n");
        return 1;
    }
    if ( ! myUdp->datagram(os.str(),-1) )
    {
        EcERROR("udp datagram send failed\n");
        return 1;
    }
    if (! myUdp->disconnect(theDisconnectTimeout) )
    {
        EcERROR("disconnect failed\n");
        return 1;
    }
    return 0;
}

```

**Figure 20-19:** Code sample for Skype UDP.

## 20.3 CommFactory Utility Class

In addition to the low- and mid-level classes described previously, a utility factory class is available to provide simpler instantiation of a transport. The factory class returns a pointer to an instance of a desired protocol and transport, properly parameterized. Figure 20-20 shows the salient methods in the factory class- the three public member functions for creating a transport instance.

Static Public Member Functions	
static <b>EcComm::Protocols::Tcp</b> *	<b>createTcp</b> (const TRANSPORT &t, const EcString &_host, const EcU32 _port, const EcBoolean beServer=EcFalse) <i>createTcp</i>
static <b>EcComm::Protocols::Udp</b> *	<b>createUdp</b> (const TRANSPORT &t, const EcU32 _port) <i>createUdp</i>
static <b>EcComm::Protocols::SkypeT</b> *	<b>createSkype</b> (const TRANSPORT &t, const EcString &_user, const EcString &_app) <i>createSkype</i>

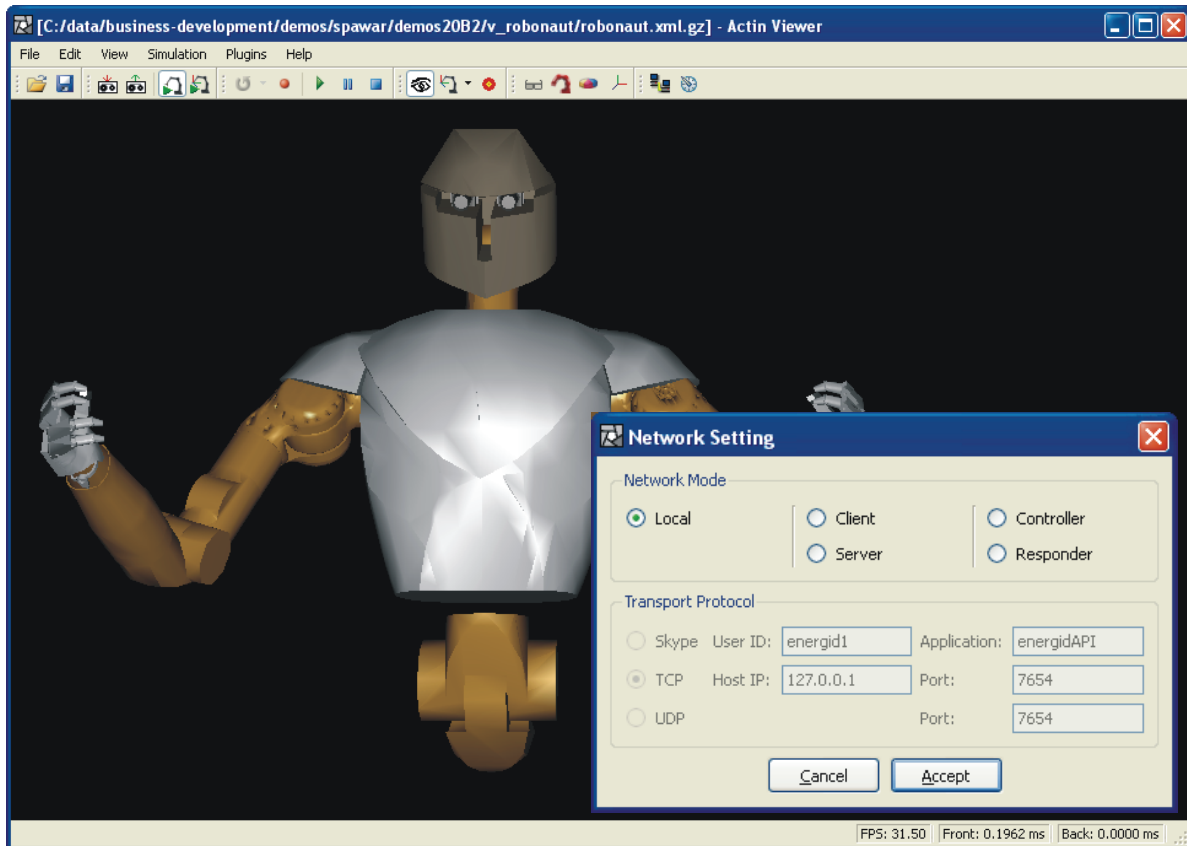
**Figure 20-20:** CommFactory methods.

Please refer to Figure 20-19 for sample code showing the creation of a Skype transport using the commFactory, and Figure 20-23 showing creation of a TCP transport.

## 20.4 Complete Examples

### 20.4.1 Viewer

Filtering streams, compression and base64 encoding have been integrated with the primary tool for visualization and study- Energid's ActinViewer. Figure 20-21 shows the concept as implemented, with data flow from the viewer, through the filters and transport, across the internet, to a live robot on the other end. Shown in Figure 20-22 is the dialog box for configuring the new network options, allowing for the specification of the various parameters in an easy-to-use graphical form.



**Figure 20-21:** Command and Control using Skype and the Actin Viewer.

The Actin Viewer, now with Skype's ability to bust through firewalls and Network Address Translations "NATs" was demonstrated at the 2007 RoboBusiness conference in Boston, Massachusetts. From the conference, the viewer was used to control a real robot arm physically located at Energid's office in India- some 7000 miles distant.

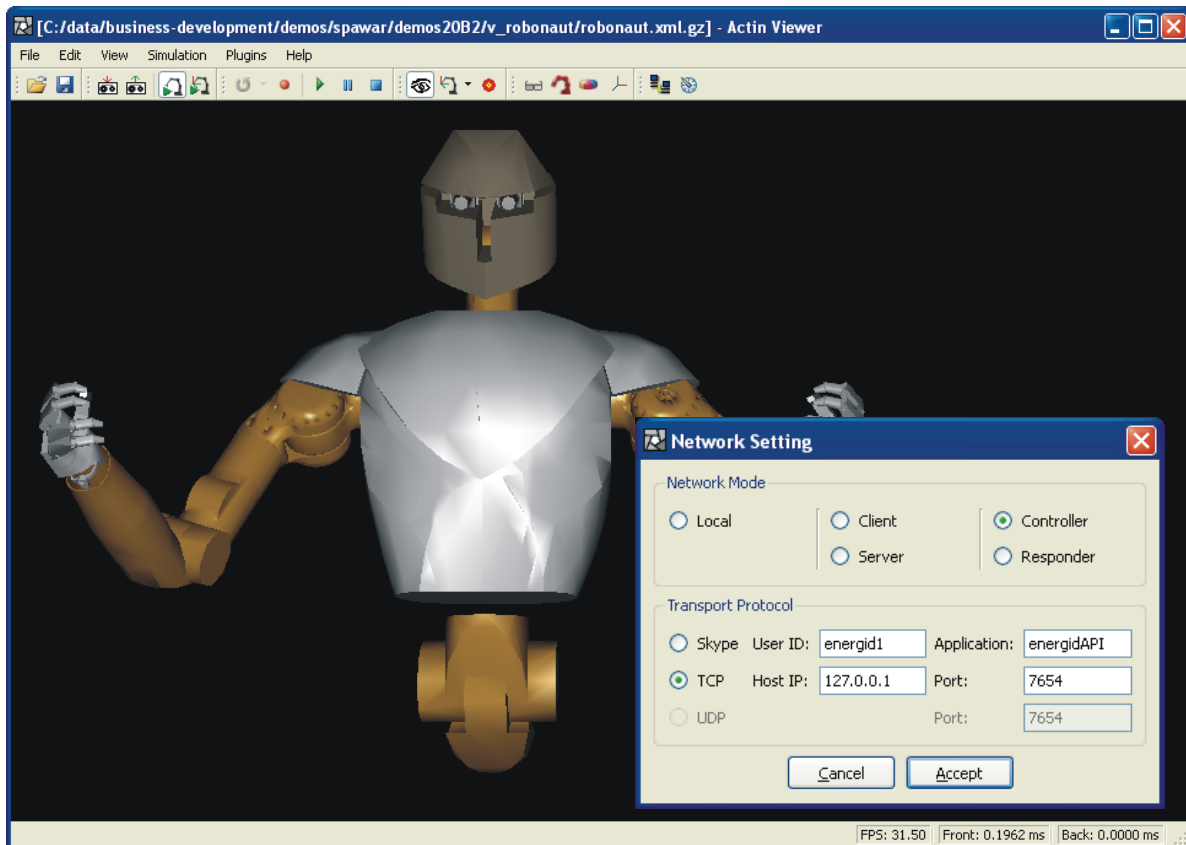


Figure 20-22: Actin Viewer with dialog for network configuration.

#### 20.4.2 *Tcp Client and Server with base64 encoding and bzip compression*

Figure 20-23 shows a complete example using the TCP protocol along with base64 encoding and compression. The server waits forever for a client to connect, then waits forever to receive a message from the client (“Johnny”), then writes a message back to the client that contains the text of the client message along with some additional text (“Hello Johnny. My name is Frankie”).



```
#include <boost/thread/thread.hpp>
#include <boost/bind.hpp>
#include <transport/ecCommFactory.h>

class ecTcpExample
{
public:
    ecTcpExample()
    { }

    virtual ~ecTcpExample()
    { }

    virtual void helloServer()
    {
        myTcpServerPtr.reset(EcComm::CommFactory::createTcp(
            EcComm::ASIO, "127.0.0.1", 9876, true));
        myTcpServerPtr->setWriteFilters(writeFilters);
        myTcpServerPtr->setReadFilters(writeFilters.invert());
        if ( myTcpServerPtr->connect(-1) )
        {
```

```

        EcString buff;
        if ( myTcpServerPtr->read(buff,-1) )
        {
            EcPRINT("Server received client name: %s\n",buff.c_str());
            EcString msg="Hello " + buff+". My name is Frankie";
            myTcpServerPtr->write(msg,-1);
        }
    }

    virtual void helloClient()
    {
        myTcpClientPtr.reset
(EcComm::CommFactory::createTcp(EcComm::ASIO,"127.0.0.1",9876,false));
        myTcpClientPtr->setWriteFilters(writeFilters);
        myTcpClientPtr->setReadFilters(writeFilters.invert());
        if ( myTcpClientPtr->connect(-1) )
        {
            if ( myTcpClientPtr->write("Johnny",-1) )
            {
                EcString buff;
                if ( myTcpClientPtr->read(buff,-1) )
                {
                    EcPRINT("Client received message: %s\n",buff.c_str());
                }
            }
        }
    }

    virtual void run()
    {
        writeFilters.addBase64Encode();
        writeFilters.addZlibCompress();

        boost::thread serverThread(boost::bind( &ecTcpExample::helloServer,this));
        boost::thread clientThread(boost::bind( &ecTcpExample::helloClient,this));
        clientThread.join();
        serverThread.join();
    }

protected:
    boost::shared_ptr<EcComm::Protocols::Tcp> myTcpServerPtr;
    boost::shared_ptr<EcComm::Protocols::Tcp> myTcpClientPtr;
    Ec::EcFilters      writeFilters;
};

#endif

int main()
{
    ecTcpExample myExample;
    myExample.run();
    return 0;
}

```

**Figure 20-23:** Complete TCP Client-Server example with encoding and compression.

### 20.4.3 Low-level UDP Example

Figure 20-24 gives the header file for a UDP server class.

```
#include <boost/thread/condition.hpp>
#include <boost/thread/mutex.hpp>

class EcUdpServerThread
{
public:
    /// constructor
    EcUdpServerThread( boost::mutex& mutex, boost::condition& condition);

    // thread initial function
    void operator()
        ();

private:
    boost::mutex&      m_Mutex;
    boost::condition& m_Condition;
};
```

**Figure 20-24:** ecUdpServerThread.h.

The code to run the example is given in Figure 19-25, and Figure 19-26 shows the implantation details of the server thread class.

```

#include <boost/scoped_ptr.hpp>
#include <boost/thread/thread.hpp>
#include <foundCore/ecMacros.h>
#include "ecUdpServerThread.h"
#include <socket/ecUdpSocket.h>
static const EcU32 EC_TEST_DATA_SIZE = 100000;
int main(argc, char**argv)
{
    // For this example, a server is created. mutex and condition help
    // coordinate events between the client and server.
    boost::mutex mutex;
    boost::condition condition;
    boost::scoped_ptr<boost::thread> pServerThread;
    {
        // Acquire lock before starting server
        boost::mutex::scoped_lock lock(mutex);
        // start the server thread
        pServerThread.reset(new boost::thread(EcUdpServerThread(mutex, condition)));

        // wait until server is ready so the server won't miss data
        // this unlocks the mutex, which is relocked when signaled by the server
        condition.wait(lock);
    }

    {
        // This lock is required to make sure the server does not close the socket
        // before all the data is received here at the client
        boost::mutex::scoped_lock lock(mutex);
        // send data to server and then receive it back prepare data
        EcReal doubleArray[EC_TEST_DATA_SIZE];
        for(EcU32 ii = 0; ii < EC_TEST_DATA_SIZE; ++ii)
        {
            doubleArray[ii] = ii;
        }
        // Put size in header
        EcSocketHeader sHeader;
        EcU32 size = EC_TEST_DATA_SIZE * sizeof(EcReal);
        sHeader.size = size;
        sHeader.ID = 0;
        EcString peerAddress = EcTestIp; // this is local IP address
        EcU16 peerPort = EcTestPort; // port
        // Send the data as a string to the server
        EcUdpSocket socket;
        EcInt32 sent = socket.send(sHeader, (EcInt8*)&doubleArray,
            peerAddress, peerPort);
        if(static_cast<EcInt64>(sent) == static_cast<EcInt64>(size))
        {
            // let the receive timeout (2 secs), otherwise it will block indefinitely.
            socket.setRecvTimeout(5000);
            EcInt8* buffer = EcNULL;
            EcInt32 received = socket.receive(sHeader, buffer, peerAddress, peerPort);
            if(static_cast<EcInt64>(received) != static_cast<EcInt64>(size))
            {
                EcWARN("EcNetworkOperationExample::run: clientToServer failed.\n");
            }
            // Destructor closes the socket.Delete buffer if it is not empty
            EcARRAYDELETE(buffer);
        }
        else
        {
            EcWARN("EcNetworkOperationExample::run: clientToServer failed.\n");
        }
    }
    // wait until server is done
    pServerThread->join();
    return 0;
}

```

**Figure 20-25:** Code to run the UDP example

```

#include "ecUdpServerThread.h"
#include <socket/ecUdpSocket.h>
#include <foundCore/ecMacros.h>

EcUdpServerThread::EcUdpServerThread(boost::mutex& mutex, boost::condition& condition)
:m_Mutex(mutex), m_Condition(condition)
{}

void EcUdpServerThread::operator() ()
{
    // create new socket, and binds to port
    EcUdpSocket socket(EcTestPort);
    // If this lock occurs before the client wait call, the server will pause because
    // a lock was previously called by the client. If this lock occurs after the client
    // wait call, the server will continue as normal.
    {
        boost::mutex::scoped_lock lock(m_Mutex);
    }
    // Server and client sockets are now ready,so release client to start sending
    m_Condition.notify_all();
    // receive and return data from/to client. Peer port and address will be set when
    // done receiving data. This will enable the data to be sent back.
    EcU16    sourcePort;
    EcString sourceAddress;
    // this buffer will be allocated inside receive(); we do not know the
    // buffer size until we receive the socket header
    EcInt8* buffer = EcNULL;
    // socket header is a struct invented ourself, can be extended for new need.
    EcSocketHeader sHeader;
    // this will let the receive timeout (5 secs), otherwise it will block indefinitely.
    socket.setRecvTimeout(5000);
    EcInt32 received = socket.receive(sHeader, buffer, sourceAddress, sourcePort);
    if (received == -1)
    {
        EcWARN("EcUdpSocket::operator() Error: received timed out. Trying again.\n");
        received = socket.receive(sHeader, buffer, sourceAddress, sourcePort);
        if (received <= 0)
        {
            // trouble receiving
            EcWARN("EcUdpSocket::operator() Error: server receive failed\n");
            EcARRAYDELETE(buffer);
            return;
        }
    }
    else if (received == 0)
    {
        EcWARN("EcUdpSocket::operator() Error: server receive failed\n");
        EcARRAYDELETE(buffer);
        return;
    }
    // send back to client
    EcInt32 sent = socket.send(sHeader, buffer, sourceAddress, sourcePort);
    if (sent != sHeader.size)
    {
        EcWARN("EcUdpSocket::operator(): failed on send()\n");
    }
    EcARRAYDELETE(buffer);
    // Locked first by client.
    // Wait to delete until client gets all of its data.
    {
        boost::mutex::scoped_lock lock(m_Mutex);
    }
}

```

**Figure 20-26:** ecUdpServerThread.cpp

## 21 Models and Other Format Loading

The primary method of loading data into the toolkit is through XML configuration files. The Actin™ toolkit provides several converters that enable the user to load in other formats. These formats can be transformed into an *EcSystemSimulation* or *EcVisualizableStatedSystem*. Once loaded, the objects can be used or written out into an XML configuration file. New converters can also be added through a dynamic library.

### 21.1 Overview of Converters

Table 21-1 contains a description of the converters supported in the Actin™ toolkit.

Format	Class	Description
3DS	<i>EcSystem3dsLoader</i>	Standard format for describing 3-dimensional scenes with one or more objects.
ASE	<i>EcSystemAseLoader</i>	Standard format for describing 3-dimensional scenes with one or more objects.
CFG	<i>EcSystemCfgLoader</i>	NASA JSC format for RoboSim. A CFG file includes VEC files to build 3-dimensional scenes. For example, this format is used for describing Robonaut.
PNTTP	<i>EcSystemPntpLoader</i>	Energid format. Point polygon files (.pntp) contain a list of 3 files for describing 3-dimensional objects: point polygons (.pp), system (.system), and state (.state).
VEC	<i>EcSystemVecLoader</i>	NASA JSC format for Robosim. A VEC file describes an object that can be combined with other objects through CFG files.

**Table 21-1:** List of format converters.

The converters are contained and called from *EcSystemAllLoader*, which also contains a dynamic-library technique for adding new converters. Currently, *EcSystemAllLoader* is only created and called from the GUI. Through the GUI, the user can use the “File->Open” command to open new files. If *EcSystemAllLoader* does not recognize the file extension, it calls the library to get new converters. The current converter dynamic library contains an example for how to add a “sphere-type” converter.

All of the converters build upon *EcBaseSystemLoader*. Table 21-2 shows the primary methods of *EcBaseSystemLoader* which is the interface for each new and existing converter.

Method	Description
loadSimulationFromFile	Create a simulation given a simulation reference

	and filename.
loadVisualizableStatedSystemFromFile	Create a visualizable stated system given a reference and filename.

**Table 21-2:** Listing of primary methods available through the *EcBaseSystemLoader* class. Each file converter including *EcSystemAllLoader* contains an overloaded version. The overloaded version of *EcSystemAllLoader* contains calls to all the other converters and the dynamic library. Check the code documentation for a complete list and description.

Although the interface for *EcSystemAllLoader* is currently only called by the GUI, it can be used by new code as the developer dictates. Text Box 21-1 shows an *EcSystemAllLoader* example taken from the self test code.

```

pSimulation = new EcSystemSimulation;
loadFile = "dllTest.sphere";

// Load the CFG file from the DLL
// The file extension is "testcfg" to force the loader to
// call the DLL.
if( loader.loadSimulationFromFile(*pSimulation,loadFile) )
{
    // If successfully loaded, write out to an XML file
    filename=EcString("./")+
        EcSelfTestDirectory+
        EcString("/")+
        EcString("sphere.xml");
    pSimulation->writeToFile(filename,EcSim::EcSimulationToken);
}

EcDELETE(pSimulation);

```

**Text Box 21-1:** *EcSystemAllLoader* example. In this example, a “sphere-type” file is loaded to a simulation and then written out to an XML file.

### 21.1.1 Point Polygon Format

The point polygon format is an Energid format that is described in more detail here. There are two point polygon format examples for the PUMA and RRC K-1207i in the data directory. The primary point polygon file contains filenames for three other files that describe the point polygons, system and state. The text box below shows an example. Each of these files consists of raw data in space delimited ASCII format.

```

# Point Polygon Data File
pp: puma6.pp

# System Data File
sys: puma6.sys

# System Data File
state: puma6.state

```

**Text Box 21-2:** Point Polygon File Example.

A conversion tool, *EcConversionTool*, is called within the point polygon converter to read the system, state, and point polygons, and do the conversion to a simulation or stated system.

The system data file contains the static parameters that define the system. The file starts with the number of degrees of freedom followed by the DH parameters, centers of mass in DH frame, masses, inertias at center of mass expressed in DH frame, gravity vector, viscous friction coefficients, motor inertias, gear ratios, lower joint limits, and upper joint limits. Except for the gravity vector, each data item is repeated for each degree of freedom. A description of the data can be placed at the bottom of the file.

The state data file contains initializations for the dynamic parameters that define the state. The file contains joint positions and rates for each degree of freedom. A description of the data can be placed at the bottom of the file.

The point polygon data file contains sets of points and polygons. The number of sets is equivalent to the number of degrees of freedom plus the base. The number of degrees of freedom is determined by the system file. The first set in the point polygon file is for the base. Each set of data starts with two numbers defining the number of points and polygons. The point and polygon data follows. Each polygon collection starts with a number of indices which enables the reader to iterate through the data.

## 21.2 Description for Adding New Converters

The toolkit provides a base class for building new converters and it provides dynamic-library source code for integrating the new converters into the developer's application. Source code for the example "sphere-type" format is contained with the dynamic-library interface; the developer can use these files as an example for developing new converters.

Once a new converter class is created, it needs to be added to the converter dynamic library. The converter DLL is organized similarly to the plugin interface described in Section 18. The dynamic library contains two functions that provide the interface to the application: *getSimulation* and *getVisualizableStatedSystem*. These two functions need to be updated with each new converter. Text Box 21-3 shows an example of the contents of *getSimulation*.

```
if(filename.find(EcString(".sphere")) != EcString::npos)
{
    EcSystemSphereLoader systemSphereLoader;
    return systemSphereLoader.
    loadVisualizableStatedSystemFromFile(visStatedSystem, filename);
}
else
{
    EcWARN("ecConvertApi::getVisualizableStatedSystem(): "
           "unknown file extension in %s\n", filename.c_str());
    return EcFalse;
}
```

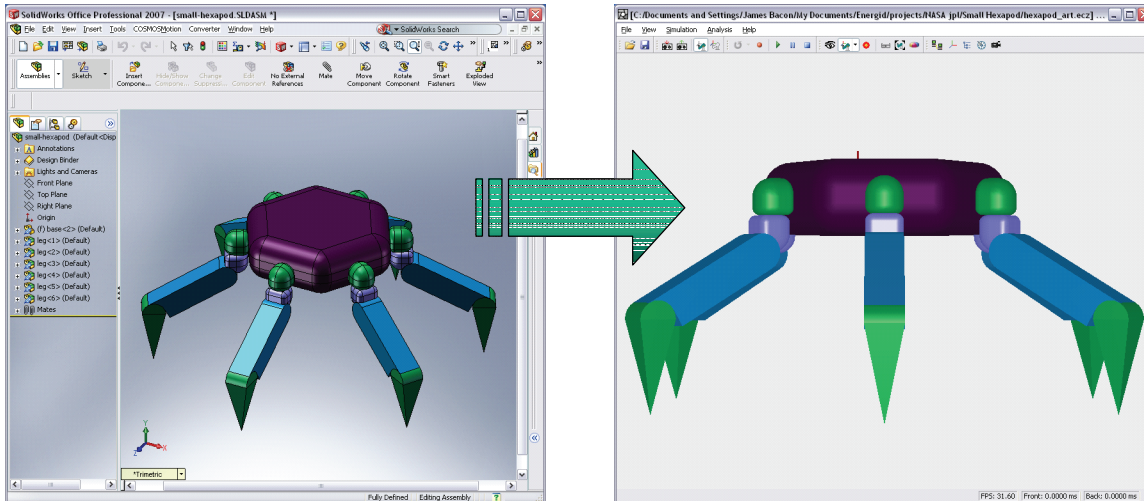
**Text Box 21-3:** Example code for *getSimulation*.

New converters can be added to the dynamic-library interface through new "else if" statements. Through this process, new converters can be added by the developer to the application.

## 21.3 SolidWorks Plugin Converter



The Actin SolidWorks Converter is a tool for converting SolidWorks models to the Actin XML format. This format can be loaded into the Actin viewer (see Figure 21-1) or utilized by other Actin applications which are created with the Actin libraries and header files. The converter is a SolidWorks add-in that is visible after selecting “Tools/Add-Ins” and checking “convert”. Once loaded, a Converter menu and toolbar options become visible.



**Figure 21-1:** The left picture shows a hexapod model within SolidWorks and the right picture shows the converted hexapod model within the Actin viewer.

The conversion process captures the physical extent and joint information of the model and other SolidWorks data such as color and mass properties. The documentation below describes the model modifications necessary to enable the conversion, and it describes the conversion options.

### 21.3.1 Model Setup

#### 21.3.1.1 Step 1: Label Each Joint

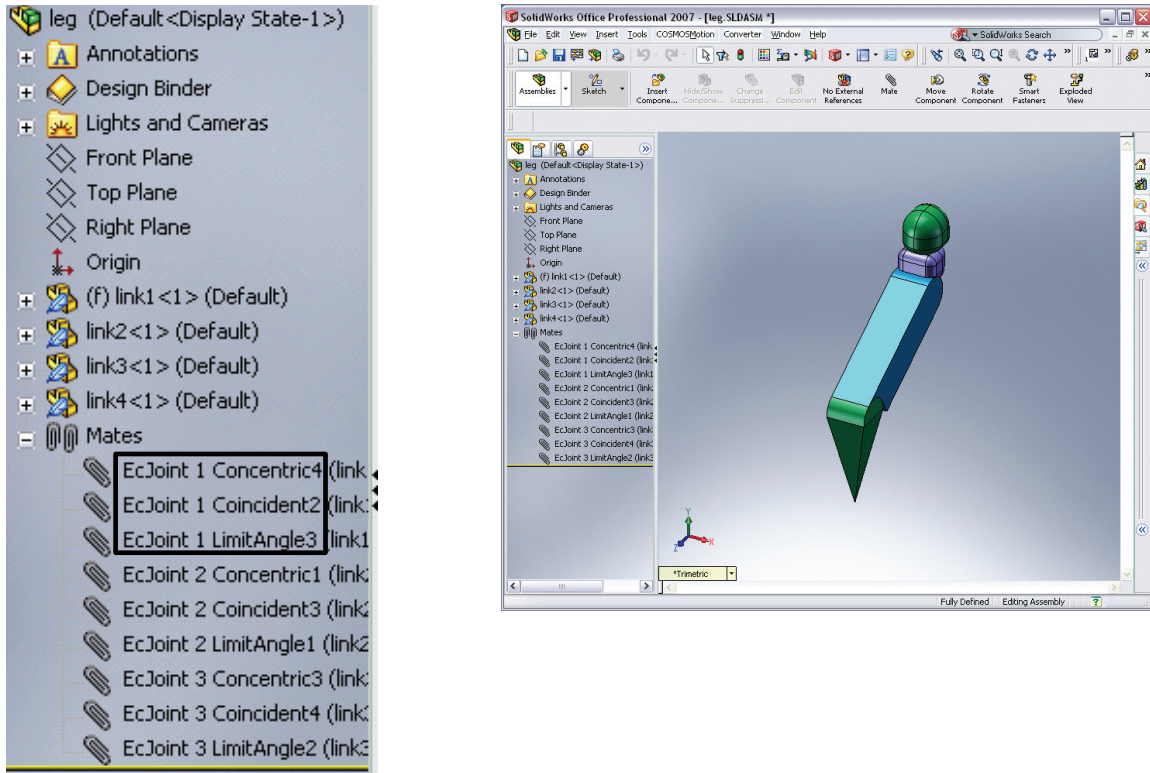
SolidWorks provides many options for mating parts together into assemblies. During the conversion to the Actin format, these mates are analyzed to determine which parts are combined into manipulator links and to determine the joint definitions between the links. To simplify the analysis, mates associated with joints are identified by a prefixed label. This label has the following syntax:

*EcJoint groupTag*

*EcJoint* is a keyword and *groupTag* is a user-define alphanumeric tag that groups the mates associated with a joint. If the *EcJoint* label is present for any mate, the mate description is added to a joint. If the label is not present, the mated parts are combined into a link. More than one mate is necessary for defining a joint. The group tag identifies collections of mates for each joint.

All mates contain information describing which mate entities to connect. The Actin converter only recognizes mate entities that are parts. For example, if a part is mated to an assembly, or an axis or reference plane not associated with a part, the mate is ignored. It is possible that enough information could be lost in this process where a link might artificially fragment. If this happens, typically a couple mates can be redefined to solve the issue. Connecting parts is a requirement because assemblies do not map well to links. Assemblies can represent a subset of a link or contain multiple links. Mapping an assembly to a link would require extra information from the user.

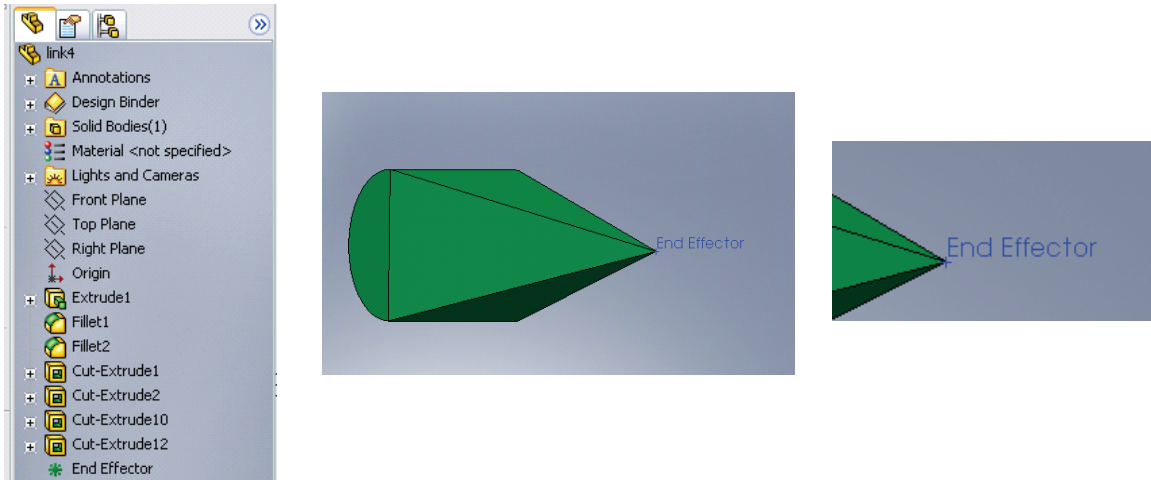
Figure 21-2 shows the mates for one leg. This hexapod has 4 joints per leg. Three are in the leg subassembly and one is in the higher level assembly where the legs are attached to the base. As can be seen, there are three joints in the mate list. Each of the three joints is defined by a coincident, concentric, and an optional angle limit mate. Note: a distance mate can be used in lieu of a coincident mate. If no angle limit mate is defined, the joint limit defaults to -180 to 180 degrees.



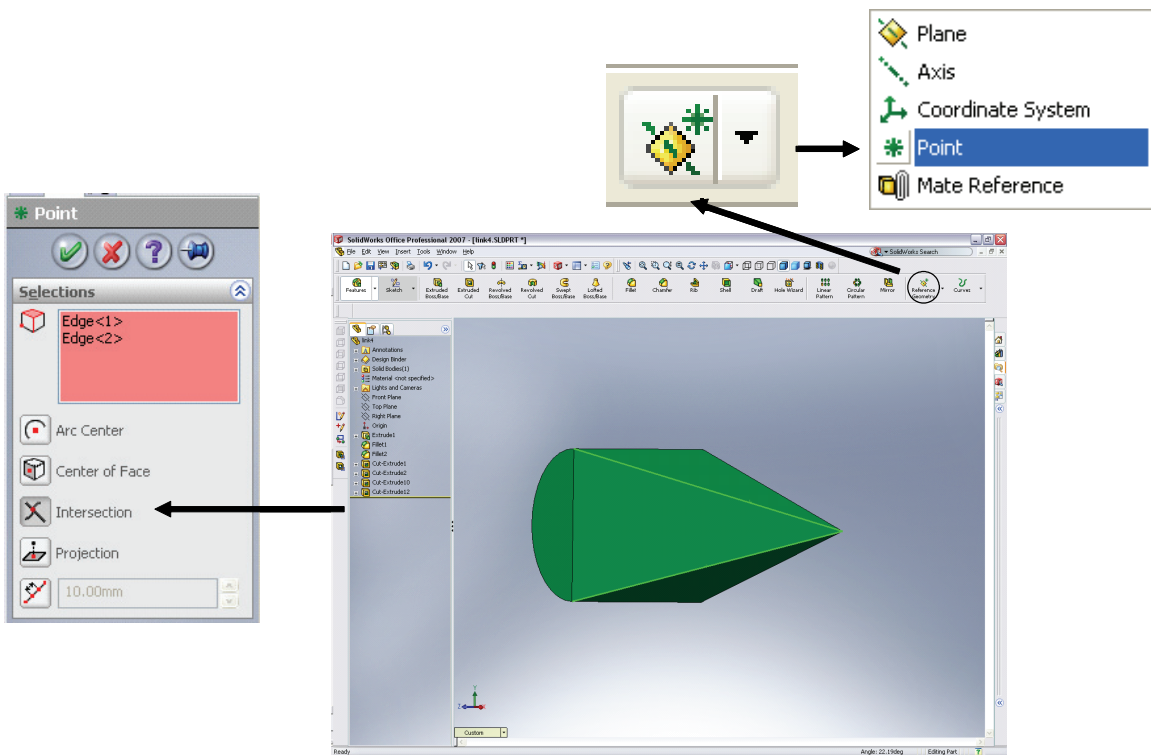
**Figure 21-2:** Mate group. Mates associated with joints are prefixed with the keyword “EcJoint” and a grouping tag.

### 21.3.1.2 Step 2: Defining Reference Frame for End Effector

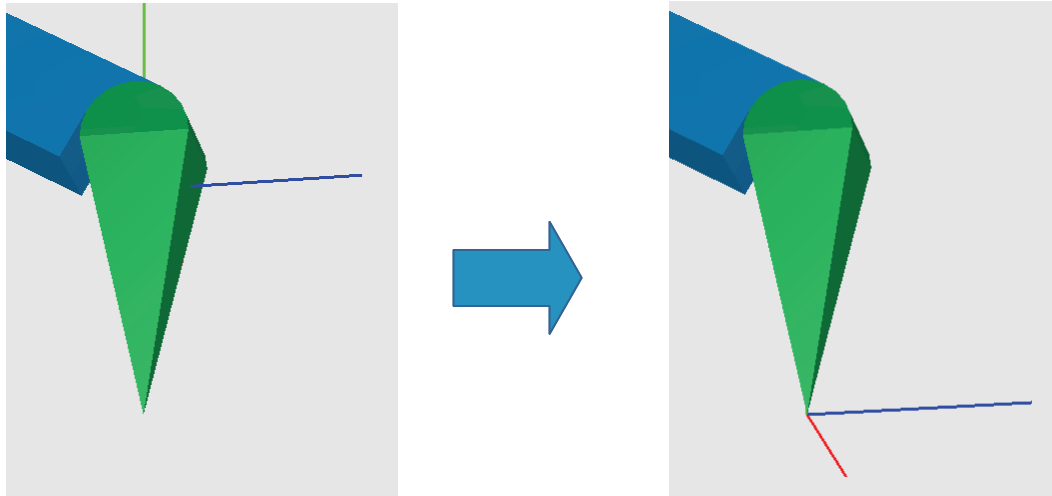
Currently, there is one other model modification which is useful for the conversion process. All other settings are established by the Converter Property Manager which is described below. The placement of the end effector frames are more intuitively set through the model instead of through data in the property manager. The end effector frames are set through a reference point named “End Effector”. Figure 21-3 illustrates where the reference point is placed for the hexapod arm. The feature manager in the left graphic shows the name of the end effector part at the top and the reference point at the bottom. Figure 21-4 illustrates how the reference point is added to the part. SolidWorks provides a flexible capability for adding reference points as described in the figure. Figure 21-5 shows how the guide frame is nicely placed on the tip of the end effector in the location of the reference point. This provides an intuitive approach to placing the end effectors.



**Figure 21-3:** View of the end effector reference point. The Feature Manager (left) shows the named “End Effector” reference point at the bottom of the list. The middle and right figures show the placement of the end effector.



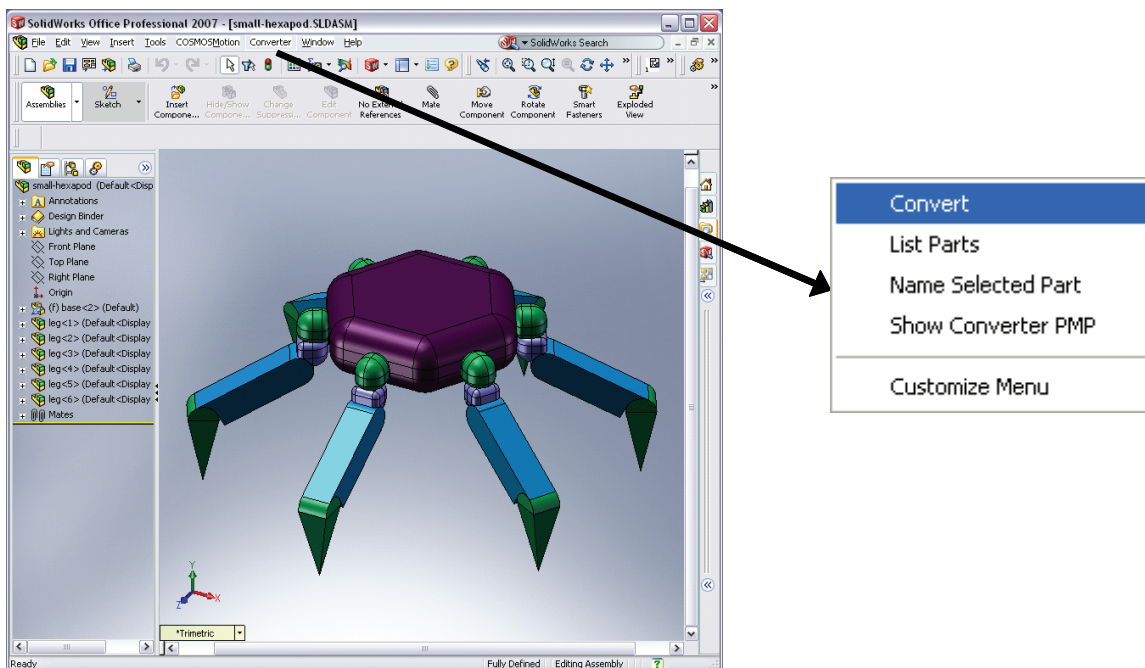
**Figure 21-4:** Illustration of how to add a reference point. Select “Point” from the toolbar at the top, which opens a property manager to the left. For the hexapod peg, select two intersecting lines on the peg. Once complete, rename the reference point in the Feature Manger to “End Effector”. This enables the user to use other reference points for other purposes.



**Figure 21-5:** Illustrates the before and after of the end effector reference point attachment. Notice that the end effector guide frame in the right figure is placed nicely at the tip of the end effector, whereas it was previously in the center of the link.

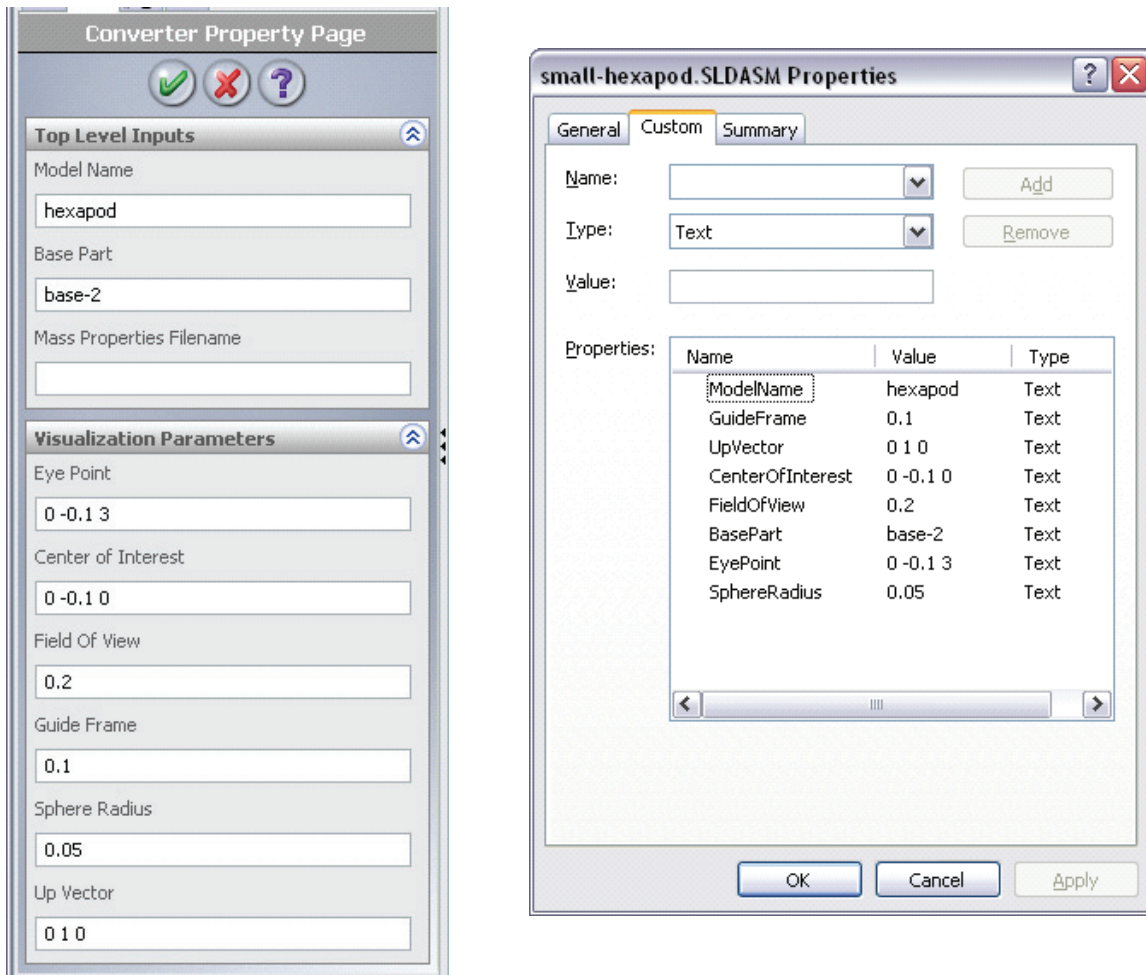
### 21.3.1.3 Step 3: Fill in the Property Manager Page (PMP)

Several parameters can be set through a property manager page. Figure 21-6 illustrates how to get to the Converter Property Manager Page by selecting “Converter/Show Converter PMP” from the menu. (Note that selecting “Converter/Convert” from the menu converts the assembly. The converter will warn the user if the appropriate parameters are not set in the Converter Property Manager Page.)



**Figure 21-6:** By selecting Converter/Show Converter PMP, the user can configure the conversion process.

Figure 21-7 shows the settings available through the Converter Property Manager Page. There are currently two groups of properties: top level inputs and visualization parameters. The Model Name and Base Part are mandatory parameters. The converter will warn the user if these are not set. The visualization parameters, described in Table 21-4 are optional. These parameters define the appearance of the model in the Actin viewer.



**Figure 21-7:** Property Manager Page. The left figure shows a complete PMP. The right figure shows how the data is saved in the file properties page. These parameters can be viewed when SolidWorks is closed.

Property Name	Description
Model Name	Name of model (e.g., hexapod). Currently, this is only used as the base name for the output file (e.g., hexapod.ecz).
Base Part	Name of base part. The base part defines the base link which is needed by the simulation and control system.

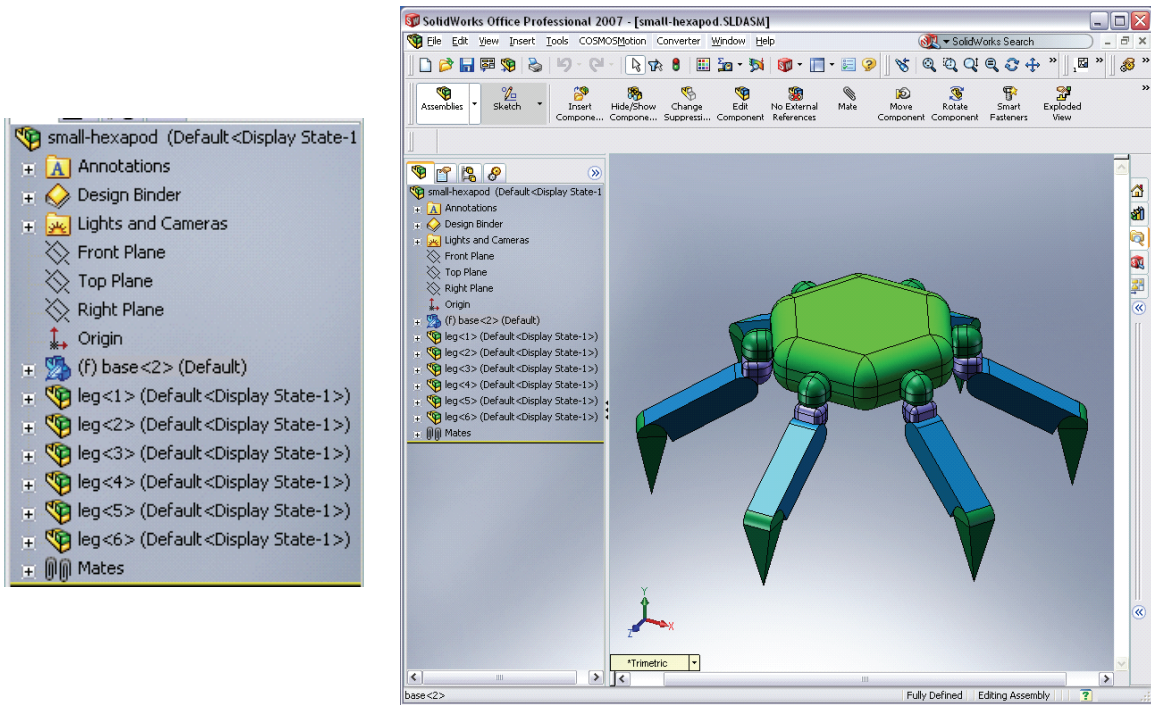
Mass Properties Filename	Excel XML file with mass properties (optional)
--------------------------	--

**Table 21-3:** Group 1 input parameters.

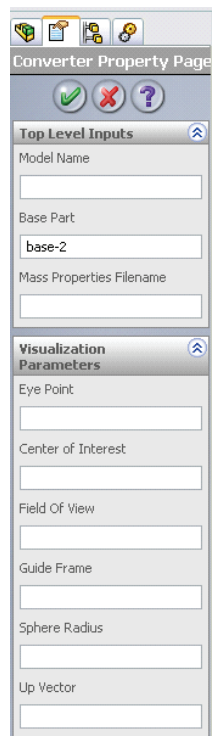
Property Name	Description
Eye Point	Eye point vector which defines the location of the eye point.
Center of Interest	Center of interest vector which defines the point being viewed.
Field of View	Field of view angle in radians.
Guide Frame	Size of guide frame that is placed on end effector.
Sphere Radius	Radius size of center of interest sphere. This center of interest sphere can be moved around in the Actin viewer to viewer different locations.
Up Vector	The up-vector is used in Actin to assist with viewing models.

**Table 21-4:** Group 2 visualization parameters. The conversion process will work without setting these parameters.

The most difficult parameter to set is the base part. The SolidWorks API uses a slightly different part naming syntax than what is shown in the SolidWorks GUI. If the base part is selected through the Feature Manager design tree (see Figure 21-8), the Converter Property Manager Page will extract the name of the selected part and initialize the base part to the selected part (see Figure 21-9). Selecting the base part through the Feature Manager is recommended instead of selection through the main window, because it is difficult to select a whole part in the main window. Typically faces, edges, and vertices (not parts) are selected through the main window. If this process produces unexpected results, the user can test the part selection through the “Converter/Name Selected Part” menu. This menu feature prints the selected part name to the screen. If there is a problem with the selection, this command will provide feedback to the user.



**Figure 21-8:** Illustration of selecting the base part in the Feature Manager. The selected part is highlighted in green in the main window.



**Figure 21-9:** Results of selecting the base part in the Feature Manager prior to opening the Converter Property Page.

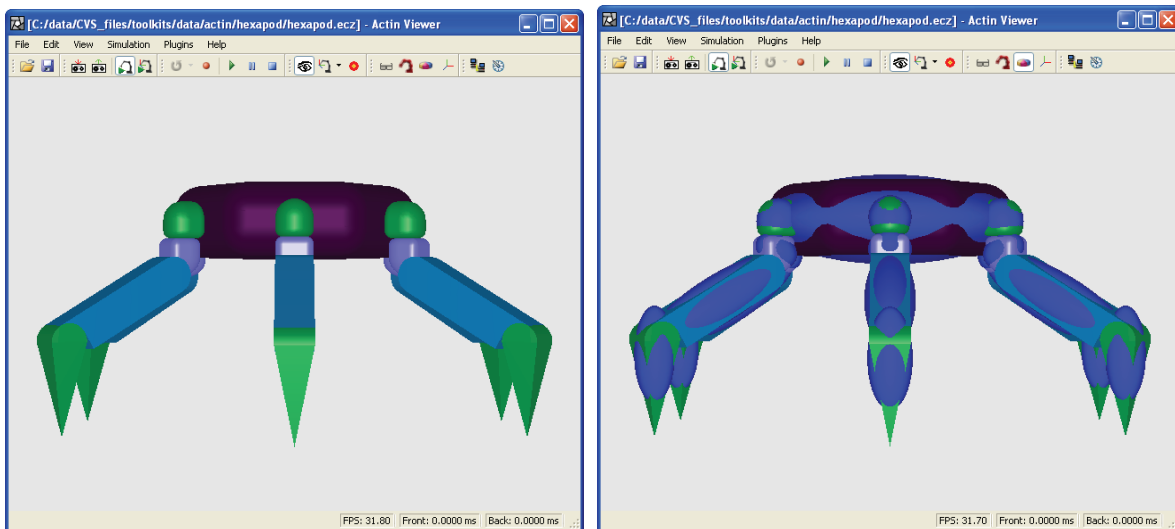
## Mass Properties

There are three methods of defining mass properties through the conversion process: 1) use SolidWorks defined mass properties, 2) set the mass properties for each link through a spreadsheet, and 3) let Actin define the mass properties.

By leaving the mass properties filename blank in the PMP, SolidWorks will define the mass properties. Figure 21-10 shows the SolidWorks mass properties in the Actin viewer.

Option 2 enables use of an Excel XML file by placing the filename in the mass properties field of the PMP. An example Excel sheet is in Figure 21-11.

Option 3 enables use of Actin for setting the mass properties. This is done by placing the keyword “Actin” in the mass properties field. This option was primarily added to overcome problems with SolidWorks models that contain surface bodies (not solid bodies). Surface bodies don’t have mass properties and oftentimes SolidWorks sends bad data for surface bodies to the converter. If the model has surface bodies, it is recommended that Option 2 or 3 be used.



**Figure 21-10:** Mass Properties Illustration. The left figure shows the default model and the right figure shows mass property ellipsoids overlaying each link. This visual approach to viewing the mass properties is a valuable verification tool.

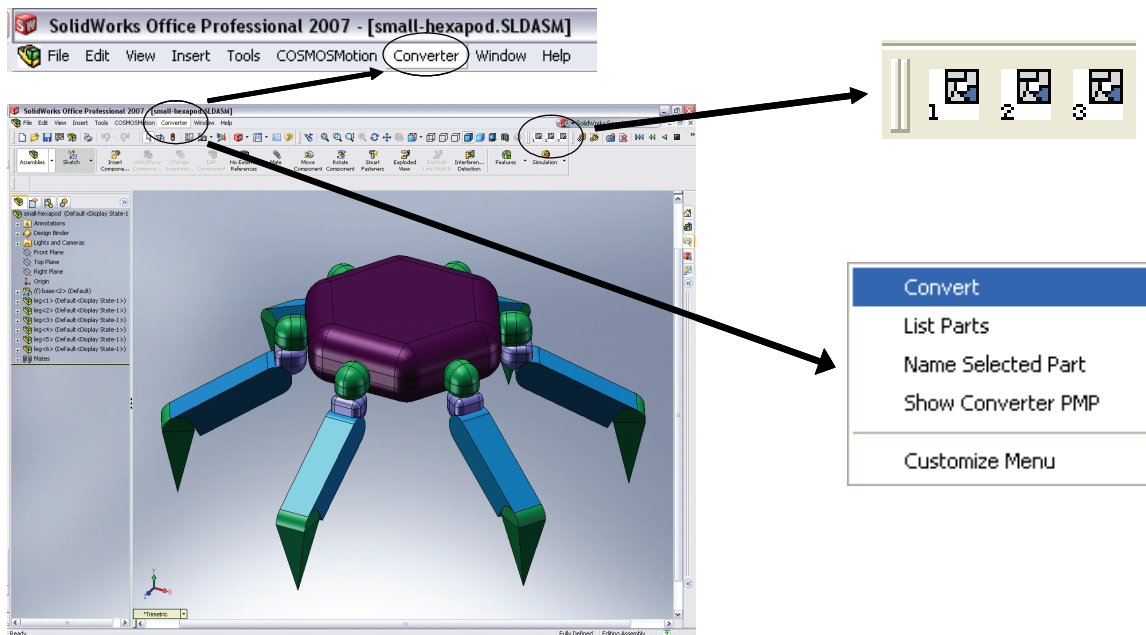


	A	B	C	D	E
1					
2	<b>Name</b>	base			
3					
4	<b>Mass</b>	Kg			1
5		5.32			
6					
7	<b>Center of Mass</b>	m			1
8		-5.18E-02			
9		1.17E-03			
10		-1.22E-03			
11					
12	<b>Moment of Inertia</b>	Kg* <sup>square</sup> m			1
13		3.81E-02	6.19E-04	-2.24E-04	
14		0.000618897	6.03E-02	-2.73E-05	
15		-0.00022378	-2.7312E-05	5.96E-02	
16					

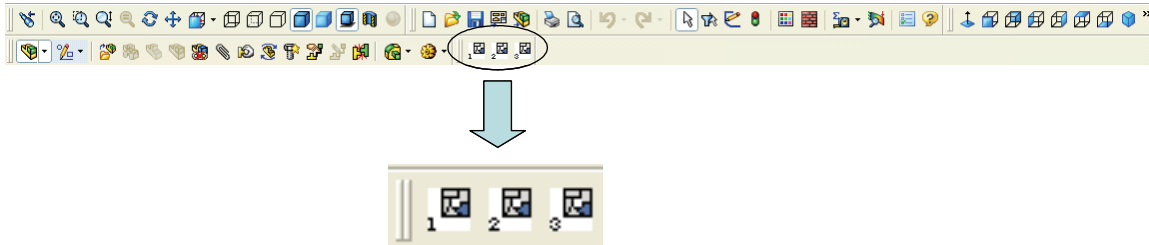
**Figure 21-11: Spreadsheet Illustration.** This spreadsheet illustrates the format for setting mass properties by link. Use “Save As” to convert to an XML file which the SolidWorks converter can use.

### 21.3.1.4 Final Step: Run Converter

The converter menu contains four options: 1) convert the model, 2) capture a part list, 3) identify the selected part, and 4) open the converter property manager page (see Figure 21-12). The first three menu options are also executable through the “Converter” toolbar (see Figure 21-13). The icons numbered 1, 2, and 3 are available for executing the menu items. Table 21-5 provides a description of the Converter menu and toolbar options.






**Figure 21-12: Execute Conversion.** Conversion can be executed through the menu or toolbar.

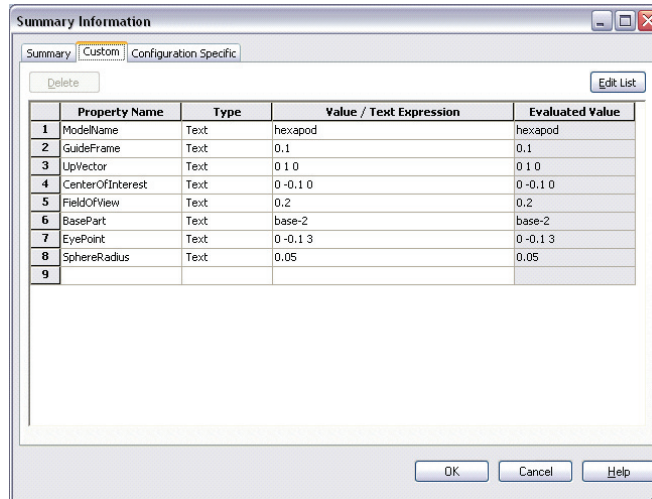


**Figure 21-13:** Illustration of the Converter toolbar.

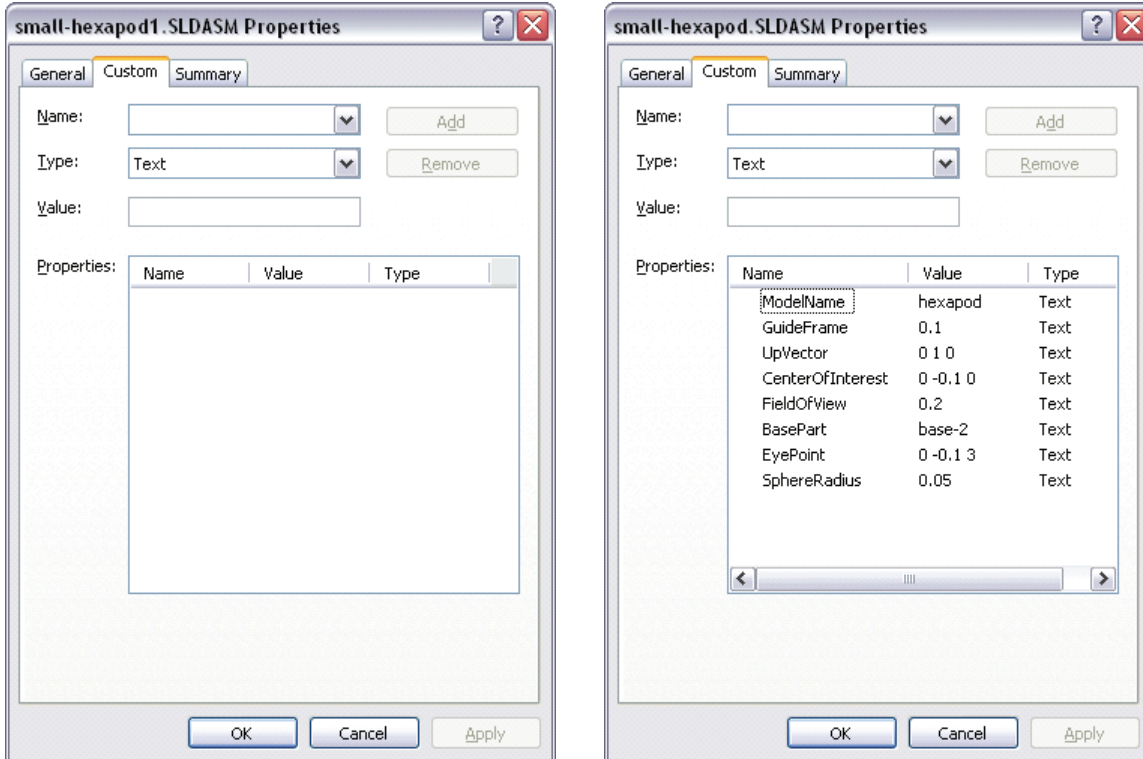
**Table 21-5:** Description of the converter menu and toolbar.

Converter Menu	Converter Toolbar	Description
Convert		Convert model
List Parts		Create part list in EcPartList.txt
Name Selected Part		Show name of selected part
Show Converter PMP	No toolbar option	Show converter property manager page

After the parameters in the Converter Property Manager Page are set and the “okay” button is selected, the parameter values are extracted from the Converter Property Manager Page and are stored in the file properties. The file properties are visible within SolidWorks and outside of SolidWorks. Within SolidWorks, the properties are visible through the “File/Properties” menu (see Figure 21-14). From outside of SolidWorks, the properties are visible by right clicking the file and selecting “properties” (see Figure 21-15). The properties are also editable through these paths, though the Converter Property Manager Page is the preferred approach.



**Figure 21-14:** Custom file properties for the conversion process are available for viewing or editing through SolidWorks by selecting the “File/Properties” menu.



**Figure 21-15:** Custom file properties provide data for the conversion process. The left figure shows a blank properties page and the right figure shows a properties page with conversion properties as set by the Converter Property Manager Page. The properties page can be accessed by right clicking on the file and selecting “Properties”.

### 21.3.1.5 Summary

The conversion time takes approximately 1 second and the file output, hexapod.ecz, is about 150 KB for the current model. Once hexapod.ecz is created, it can be viewed by the Actin viewer and the

manipulator can be commanded to perform various actions. The Actin libraries and headers can also be used to create new applications of which the Actin viewer is just one of them.

## 22 Analysis Tools

### 22.1 Using Simulink to Drive Actin with Desired End Effector Positions

The contents of the Simulink end-effector model are in [InstallPath]\toolkits\software\actinSimulinkInterface\driveActinWithEndEffectors. There are five files in the directory as shown in Figure 22-1. All of these files are text files. Simulink models have the “mdl” extension. To exercise this example, double click driverActinWithEndEffector.mdl.

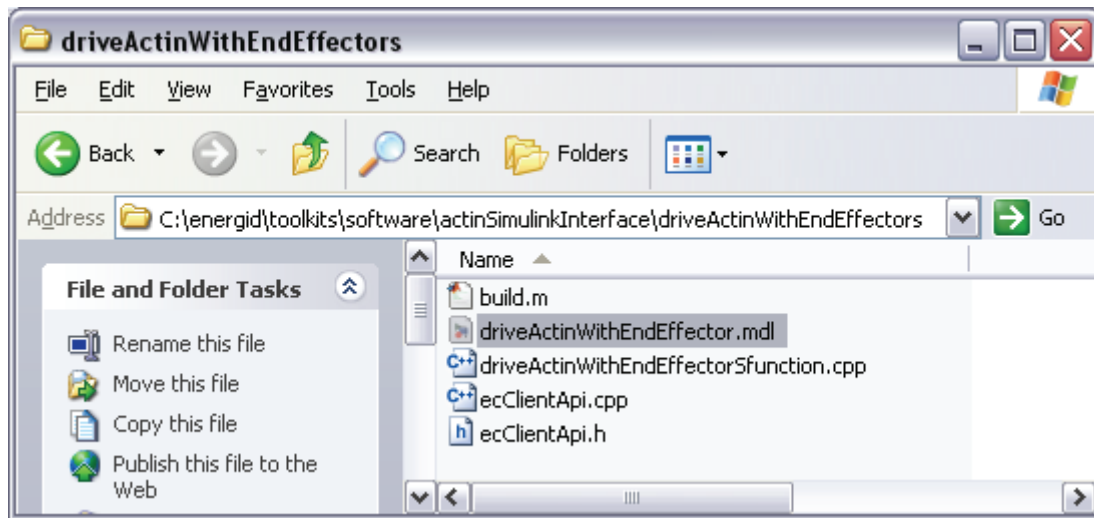


Figure 22-1: Directory content for End Effector simulation.

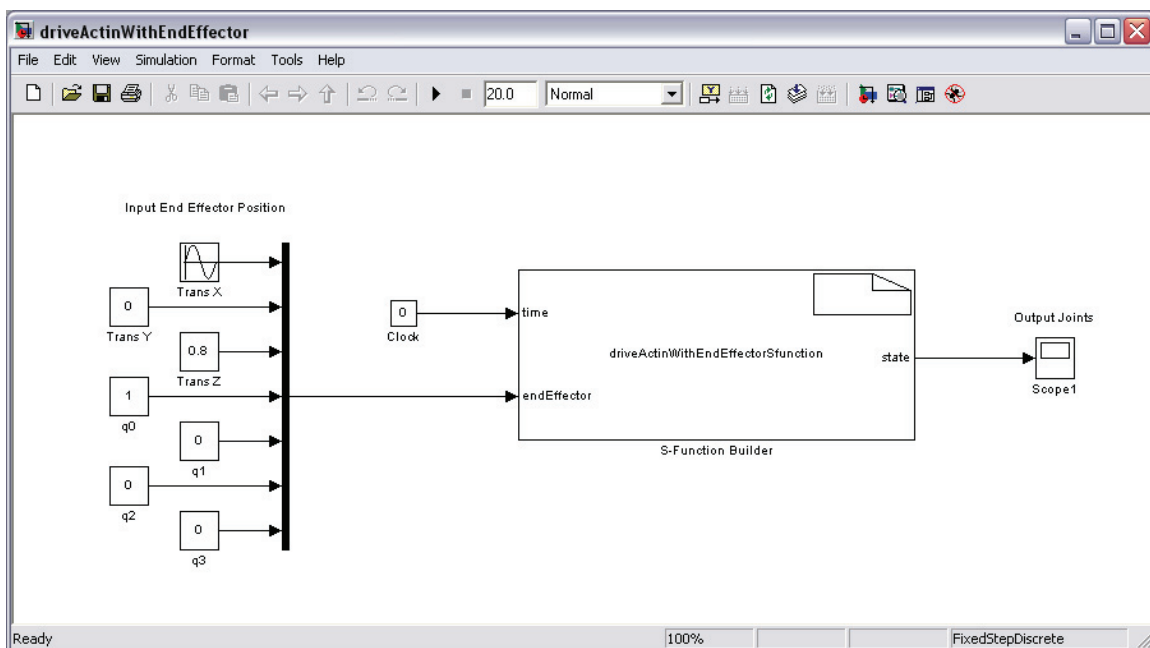
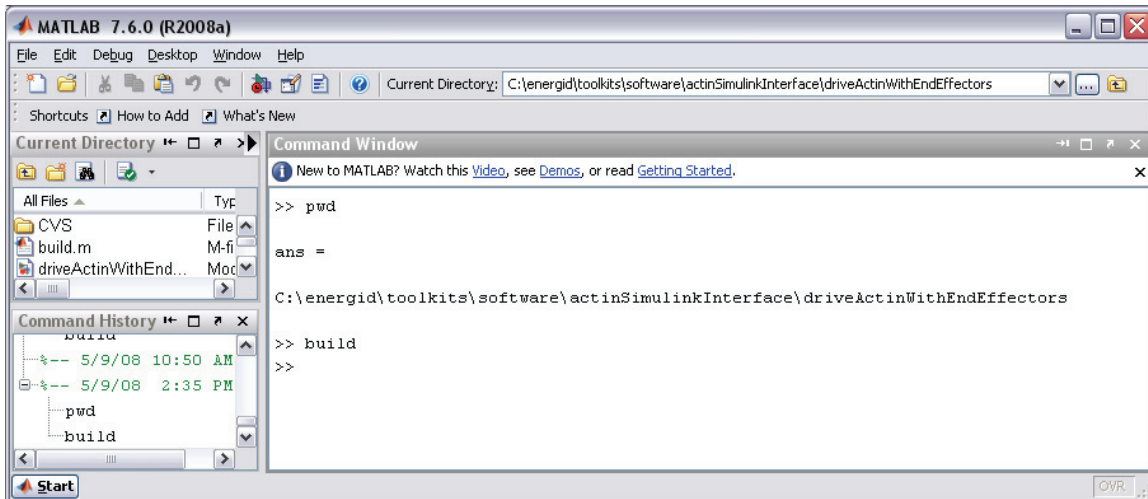


Figure 22-2: Example Simulink model for the End Effector simulation.



**Figure 22-3:** Build model using CMex.

Double clicking the mdl file opens both the model (see Figure 22-2) and a Matlab command window (Figure 22-3). As seen in the mdl file, there are three primary features in the simulation: 1) the S-Function which is the large block in the middle, 2) the S-Function inputs to the left, and 3) the S-Function outputs to the right. The inputs are time and end-effector positions (three translations and four quaternions for orientation). The outputs are a vector of joint angles. The inputs and outputs of the S-Function can be redirected to a much more sophisticated simulation. The current inputs and outputs primarily illustrate capability and provide test signals.

As seen in Figure 22-1, there are three C++ source files: driverActinWithEndEffectorSfunction.cpp, ecClientApi.cpp, and ecClientApi.h. These source files need to be built prior to running the simulation that is visible in the Simulink model (mdl file). Figure 22-3 illustrates how to build the files using the build.m script. Build.m contains the build setting for CMex and calls the CMex compiler (see the text box below).

```

TOOLKITS = getenv('EC_TOOLKITS');
includeDirs = [ ...
    '-I' TOOLKITS 'include' '' ...
    '-I' TOOLKITS '..\external\boost_1_34_1' '' ...
    '-I' TOOLKITS 'software\actin\src\test' '' ...
    '-I' TOOLKITS 'software\foundation\src\test' '' ...
];

flags = '-DWIN32 -DMATLAB_INTERFACE -DBOOST_ALL_NO_LIB -
DEC_STABLE_FOUNDCORE_DYNAMIC_LIBS';

LIBPATH = [TOOLKITS ...
    'lib' ...
];
LIBPATH1 = [TOOLKITS ...
    'external\bzip2-1.0\lib' ...
];
LIBPATH2 = [TOOLKITS ...
    'external\qhull-2003.1\lib' ...
];
LIBPATH3 = [TOOLKITS ...
    'external\zlib-1.2.3\lib' ...
];
libs= [ ...
    LIBPATH 'control.lib' '' ...
    LIBPATH 'convertSimulation.lib' '' ...
    LIBPATH 'convertSystem.lib' '' ...
    LIBPATH 'filterStream.lib' '' ...
    LIBPATH 'foundCore.lib' '' ...
    LIBPATH 'function.lib' '' ...
    LIBPATH 'geometry.lib' '' ...
    LIBPATH 'grasping.lib' '' ...
    LIBPATH 'imageSensor.lib' '' ...
    LIBPATH 'iostreams.lib' '' ...
    LIBPATH 'manipulator.lib' '' ...
    LIBPATH 'matrixUtilities.lib' '' ...
    LIBPATH 'measure.lib' '' ...
    LIBPATH 'simulation.lib' '' ...
    LIBPATH 'simulationAnalysis.lib' '' ...
    LIBPATH 'socket.lib' '' ...
    LIBPATH 'stream.lib' '' ...
    LIBPATH 'visualization.lib' '' ...
    LIBPATH 'vrml97.lib' '' ...
    LIBPATH 'walking.lib' '' ...
    LIBPATH 'xml.lib' '' ...
    LIBPATH1 'bzip2.lib' '' ...
    LIBPATH2 'qhull.lib' '' ...
    LIBPATH3 'zlib.lib' '' ...
];

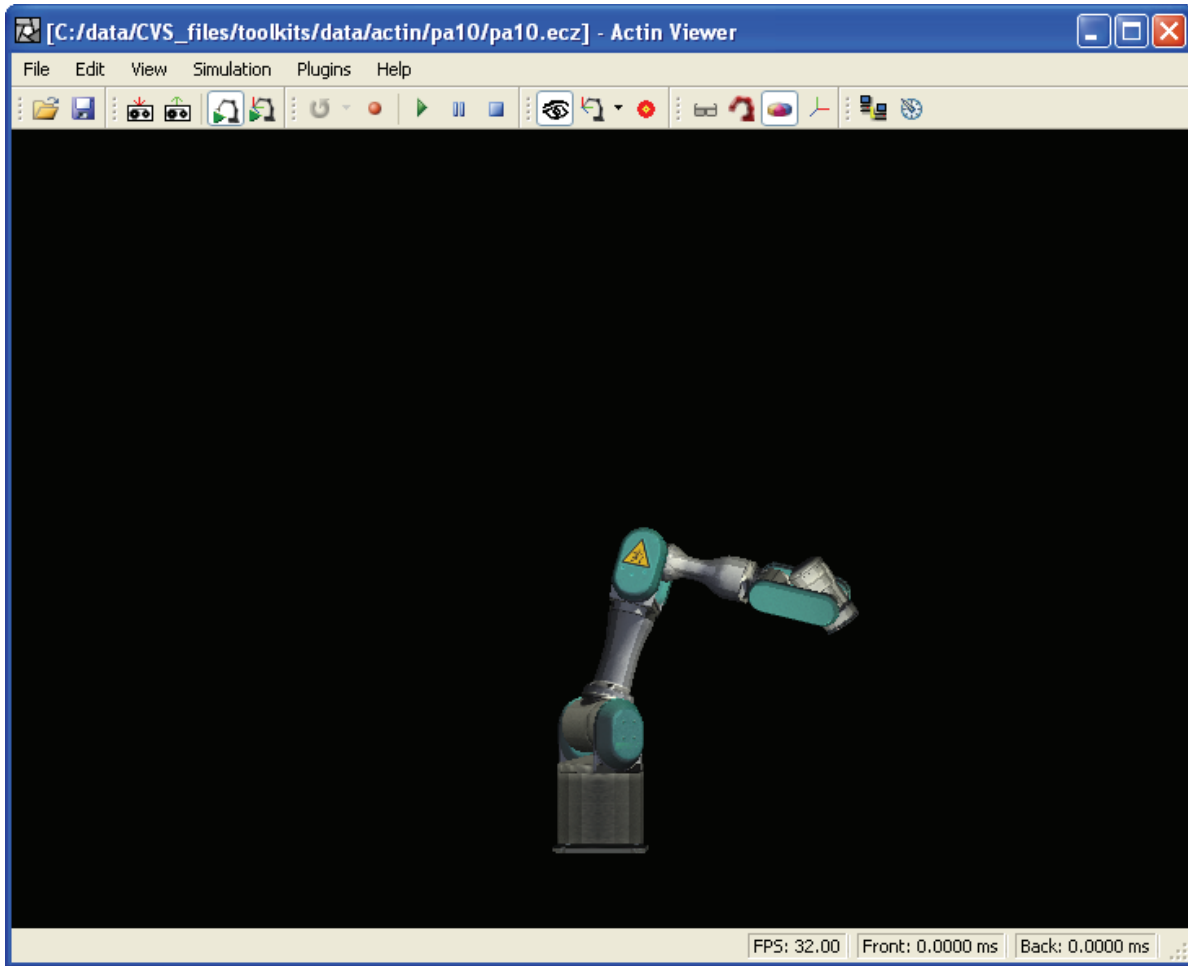
src= ['driveActinWithEndEffectorSfunction.cpp ecClientApi.cpp'];

% build the mex file.
eval(['mex ' flags '' includeDirs '' src '' libs]);

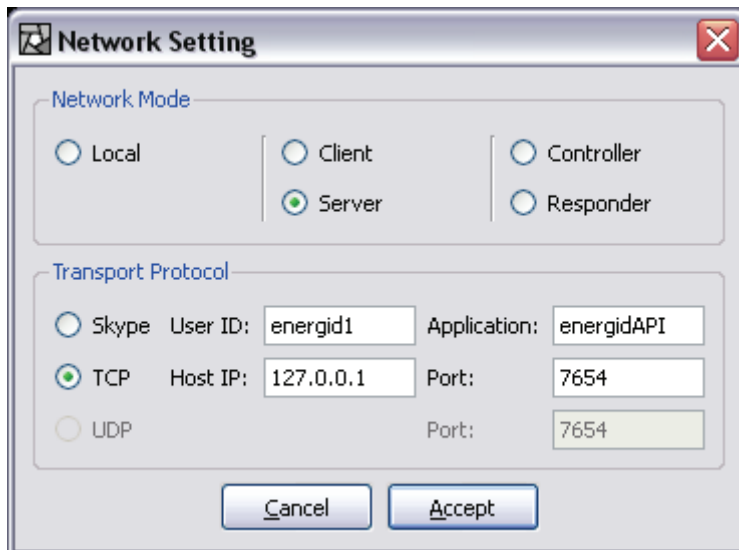
```

**Text Box 22-1:** Contents of build script.

Once the model is built, the simulation can be executed. The S-Function in the Simulink model connects to the Actin Viewer through TCP/IP. Before running the simulation, open the Actin Viewer and load a model (see Figure 22-4). Put the viewer in server mode (see Figure 22-5). Server mode enables the viewer to run the simulation and send out data to other applications through TCP/IP.

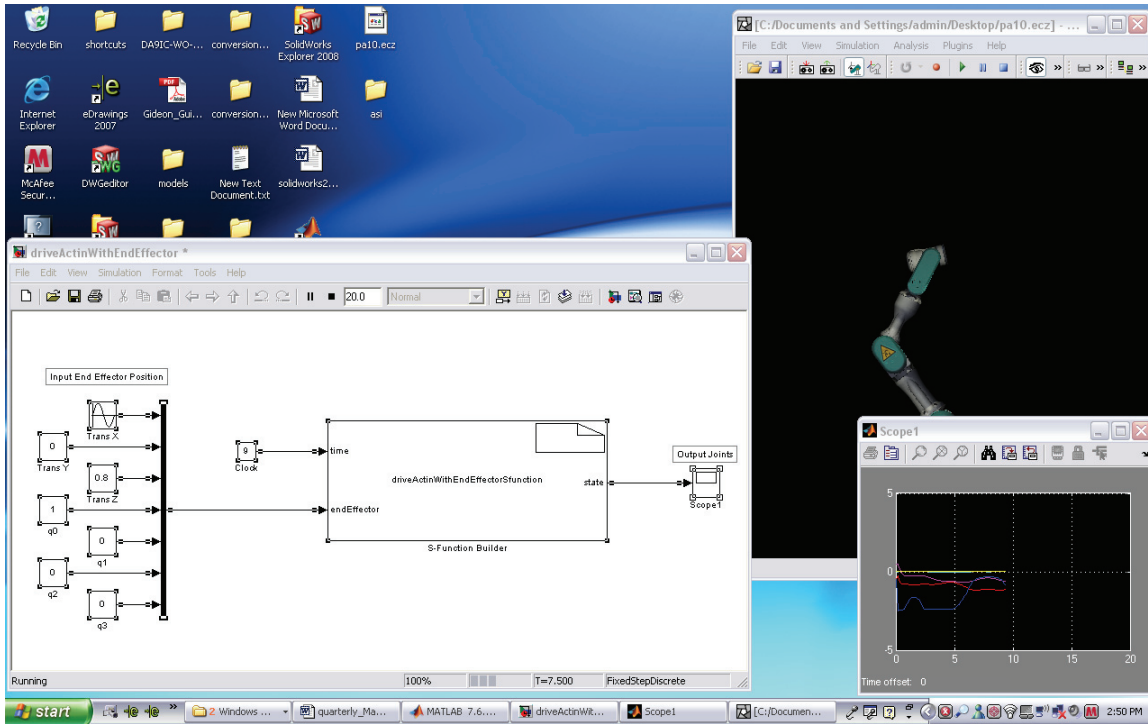


**Figure 22-4:** Model to run with Simulink simulation.



**Figure 22-5:** Enter server mode so that Actin simulation can calculate joint angles from end-effector commands.

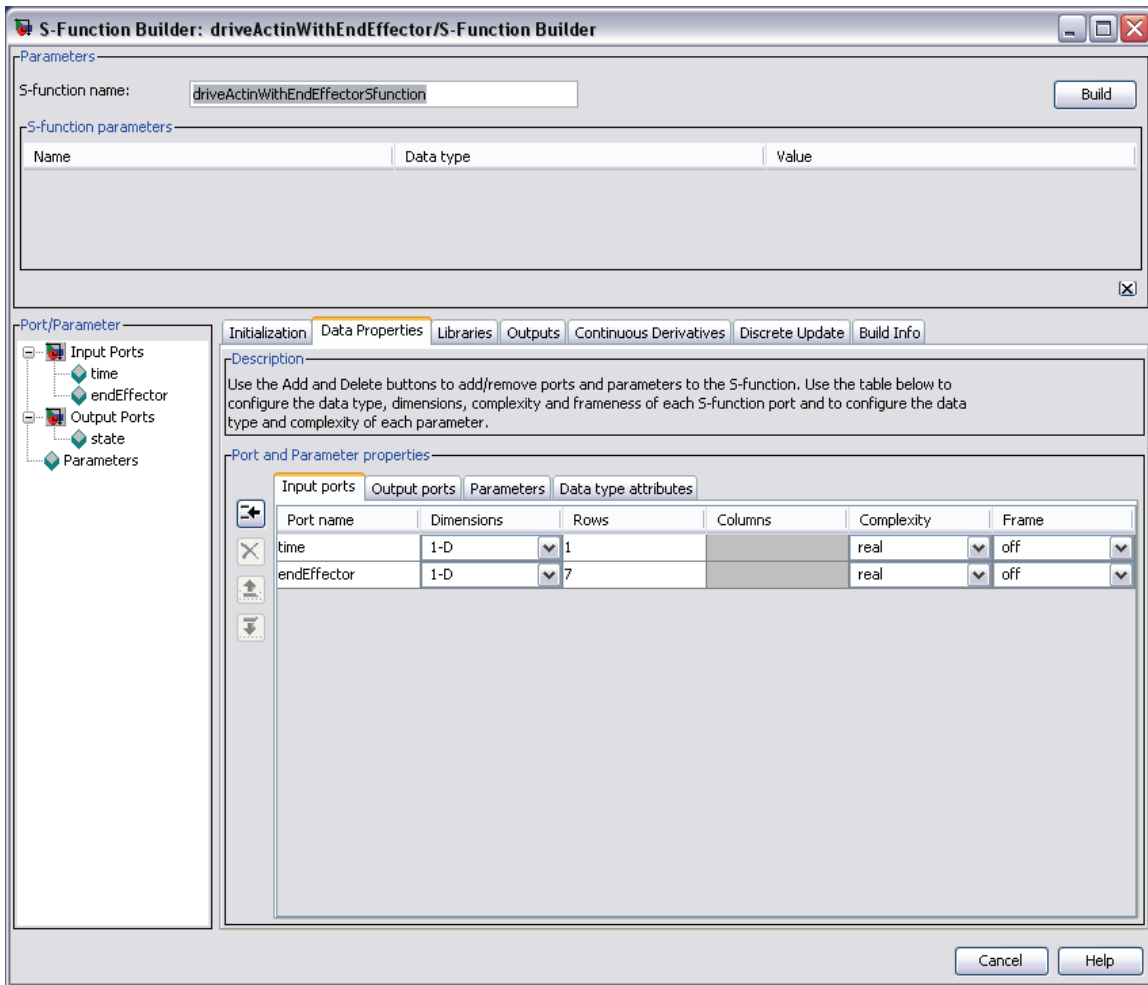
After entering server mode, the viewer waits for Simulink to send commands. In the Simulink window, the user can configure various parameters like execution time, and then tell the simulation to run (see Figure 22-6). This figure also shows a plotting window which gives a time history of the joint angles.



**Figure 22-6: Run Simulation**

S-Functions are powerful and can be very complex. Simulink supplies several tools and techniques to assist with development. Figure 22-7 shows the S-Function builder for auto-generating simple S-Functions. This S-Function builder was used to create the initial code for the current S-Function. The builder always starts from scratch and creates a new C file with the S-Function name. In this example, the C file was renamed with a C++ file extension. Rerunning the builder will not overwrite this cpp file. The edits to the auto-built file were minimal. This is nice because the builder can be reused to make changes and the Actin code interface can be reinserted. This approach is currently helpful to resolve one of the primary issues with S-Functions; that is, that the number of inputs is hardwired. S-Functions have some dynamic capability and Energid is researching this. For example, Actin can receive a vector of end-effector commands from a vector of manipulators. In this Simulink model though, only one end-effector command is used from one manipulator. Many more end-effector inputs can be added to this S-Function as desired by the developer. The builder (Figure 22-7) enables the developer to easily change the input and output interface. Once modified, the build button needs to be pushed, the Actin code needs to be re-integrated with driverActinWithEndEffectorSfunction.cpp, and the Matlab build script needs to be executed again. For an experienced user, this can be done in 10-15 minutes.





**Figure 22-7:** Overview of S-Function builder for simple functions.

Developers experienced with S-Functions can also bypass the builder and edit the file directly. The text box below shows an example of the end-effector inputs. INPUT\_1\_WIDTH shows the hard-wired size of the end-effector input array (currently set to 7).

```

/* Input Port 1 */
#define IN_PORT_1_NAME      endEffector
#define INPUT_1_WIDTH      7
#define INPUT_DIMS_1_COL   1
#define INPUT_1_DTYPE      real_T
#define INPUT_1_COMPLEX    COMPLEX_NO
#define IN_1_FRAME_BASED   FRAME_NO
#define IN_1_DIMS          1-D
#define INPUT_1_FEEDTHROUGH 1
#define IN_1_ISSIGNED      0
#define IN_1_WORDLENGTH    8
#define IN_1_FIXPOINTSCALING 1
#define IN_1_FRACTIONLENGTH 9
#define IN_1_BIAS          0
#define IN_1_SLOPE         0.125

```

**Text Box 22-2:** Section of auto-code that shows settings for end-effector inputs.

In the S-Function code (e.g., driverActinWithEndEffectorSfunction.cpp), there are three primary functions as illustrated in Table 22-1. Though there are lots of functions, many of which are not utilized in this example, these 3 function provide the interface to Actin.

Function	Description
mdlStart	One-time initialization function. This is called once at the beginning of each simulation start.
mdlOutputs	Outputs data once per timestep. Since this S-Function does not calculate states and derivatives, most of the content of the S-Function is called through mdlOutputs.
mdlTerminate	One-time termination function. This is called once at the end of each simulation run.

**Table 22-1:** Primary functions of S-Function.

### 22.1.1.1 Using Simulink to Drive Actin with Joint Angles

The contents of the Simulink joint model are in [InstallPath]\toolkits\software\actinSimulinkInterface\driveActinWithJoints. There are five files in the directory as shown in Figure 22-8. To exercise this example, double click driverActinWithJoints.mdl.

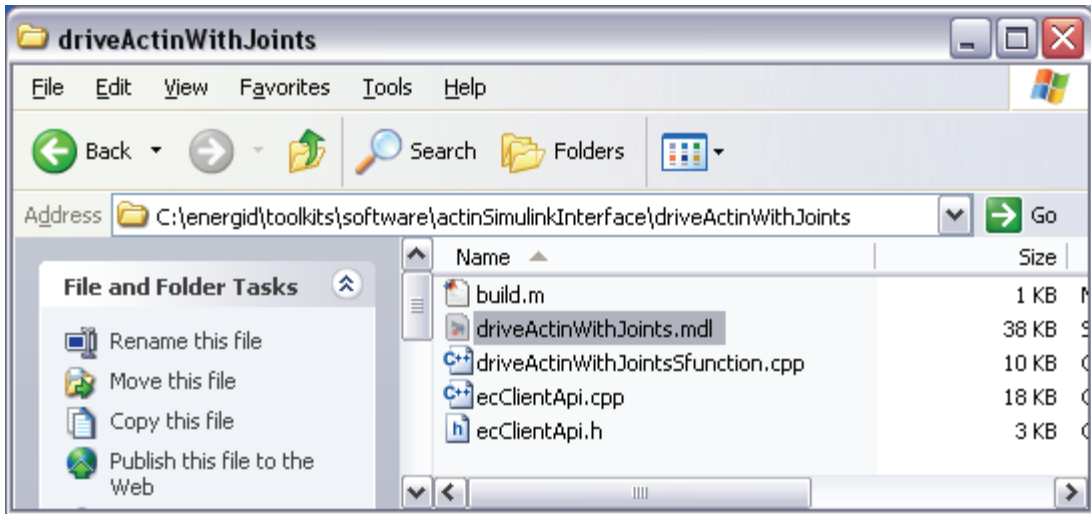


Figure 22-8: Directory content for joint simulation.

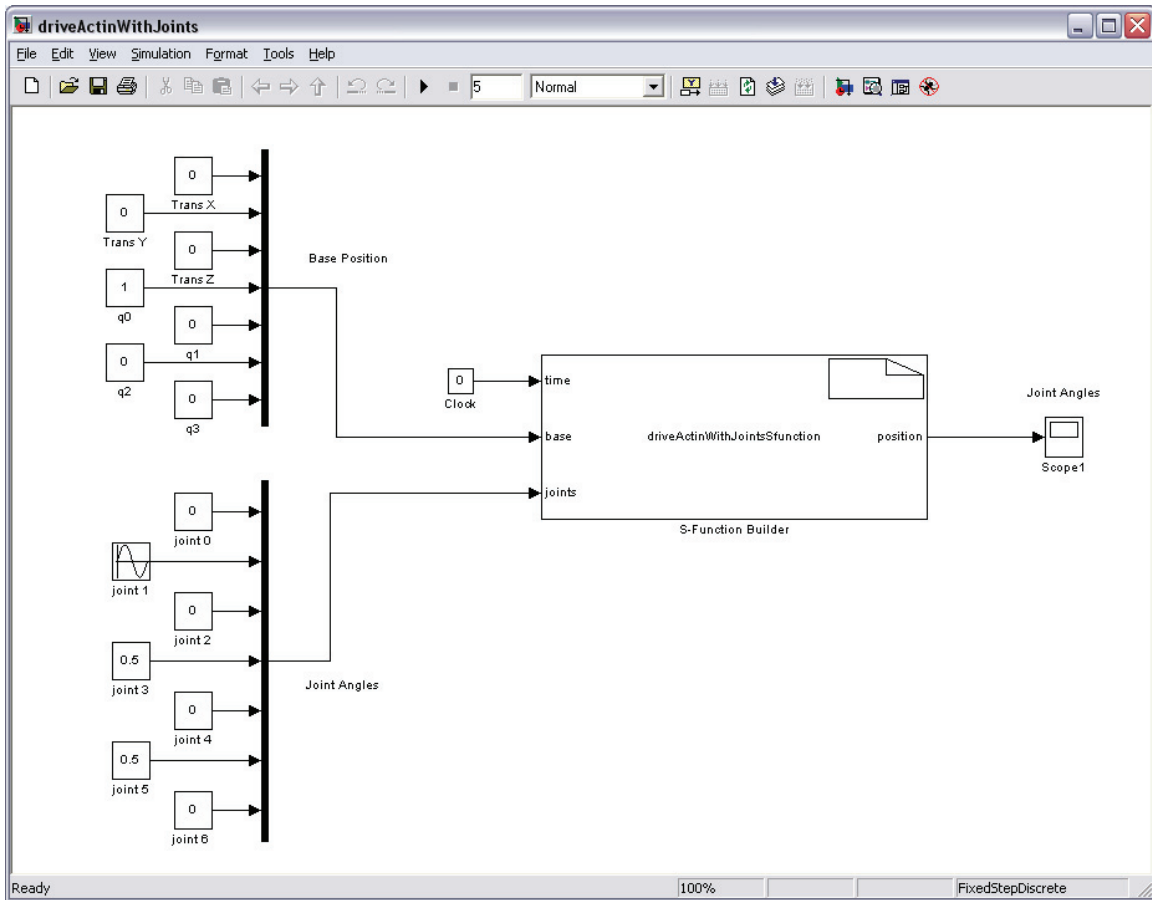
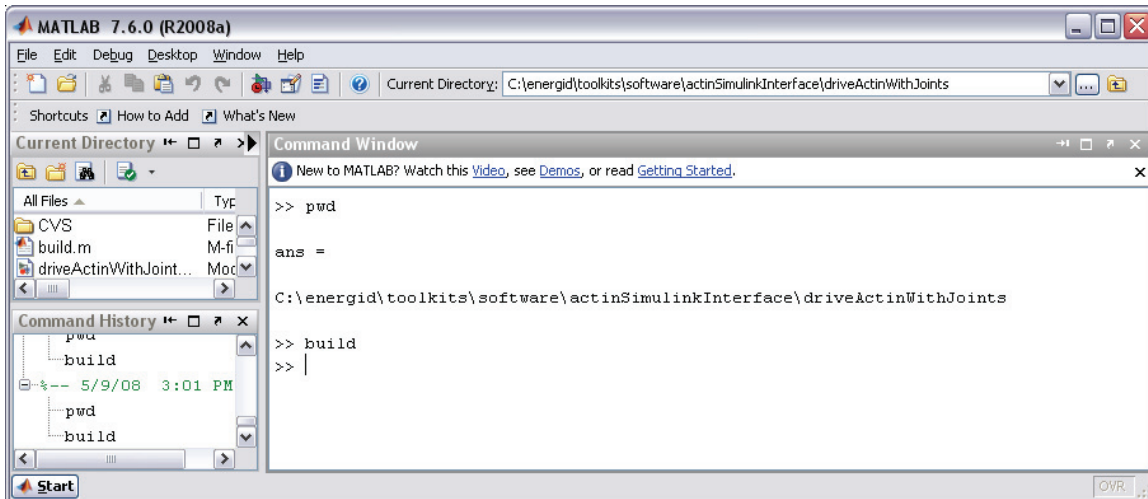


Figure 22-9: Example Simulink model for joint simulation.



**Figure 22-10:** Build model using CMex.

Double clicking the mdl file opens both the model (see Figure 22-9) and a Matlab command window (Figure 22-10). As seen in the mdl file, there are three primary features in the simulation: 1) the S-Function which is the large block in the middle, 2) the S-Function inputs to the left, and 3) the S-Function outputs to the right. The inputs are time, base position, and joint angles. The inputs and outputs of the S-Function can be redirected to a much more sophisticated simulation just as with the end-effector simulation. The current inputs and outputs primarily illustrate capability and provide signals for testing. These can be changed any way the developer wishes. To activate the Simulink simulation, Figure 22-10 illustrates how to build the source files using the build.m script. The text box below shows the contents of the build script. This script has less content than the end-effector build script due to the need for less Actin code.

```

TOOLKITS = getenv('EC_TOOLKITS');
includeDirs = [ ...
    '-I' TOOLKITS 'include' '' ...
    '-I' TOOLKITS '..\external\boost_1_34_1' '' ...
];

flags = '-DWIN32 -DMATLAB_INTERFACE -DBOOST_ALL_NO_LIB -
DEC_STABLE_FOUNDCORE_DYNAMIC_LIBS';

LIBPATH = [TOOLKITS ...
    'lib' ...
];
LIBPATH1 = [TOOLKITS ...
    'external\bzip2-1.0\lib' ...
];
LIBPATH2 = [TOOLKITS ...
    'external\zlib-1.2.3\lib' ...
];
libs= [ ...
    LIBPATH 'foundCore.lib' '' ...
    LIBPATH 'xml.lib' '' ...
    LIBPATH 'filterStream.lib' '' ...
    LIBPATH 'iostreams.lib' '' ...
    LIBPATH 'stream.lib' '' ...
    LIBPATH 'socket.lib' '' ...
    LIBPATH1 'bzip2.lib' '' ...
    LIBPATH2 'zlib.lib' '' ...
];

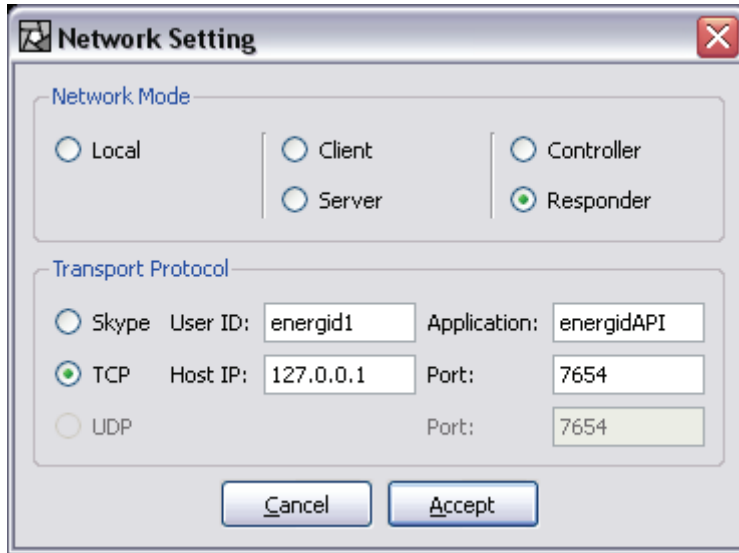
src= ['driveActinWithJointsSfunction.cpp ecClientApi.cpp'];

% build the mex file.
eval(['mex ' flags '' includeDirs '' src '' libs]);

```

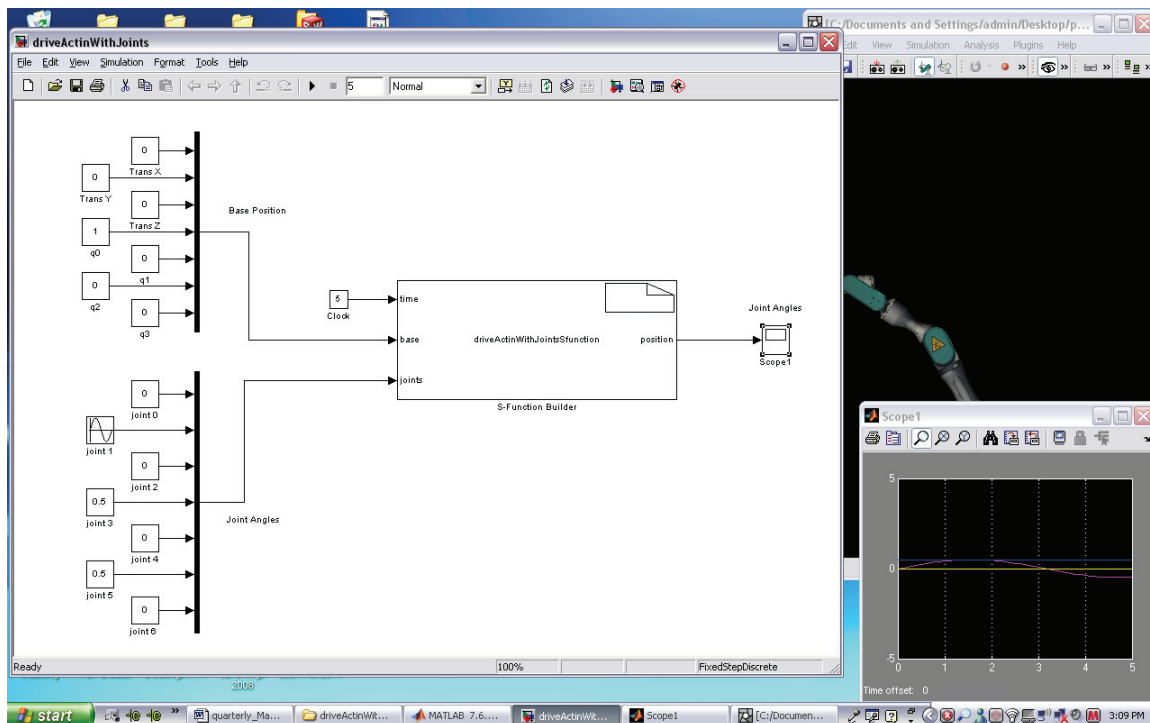
**Text Box 22-3:** Contents of build script.

Once the model is built, the simulation can be executed. The S-Function in the Simulink model connects to the Actin Viewer through TCP/IP. Before running the simulation, open the Actin Viewer and load a model (see Figure 22-4). Put the viewer in responder mode (see Figure 22-11). Note that the end-effector simulation requires server mode. Responder mode is different in that it does not run the Actin simulation. The Actin Viewer takes the base positions and joint angles, updates the state, and renders to result. Responder mode enables the viewer to receive these inputs through TCP/IP from another application like Simulink and render the results.



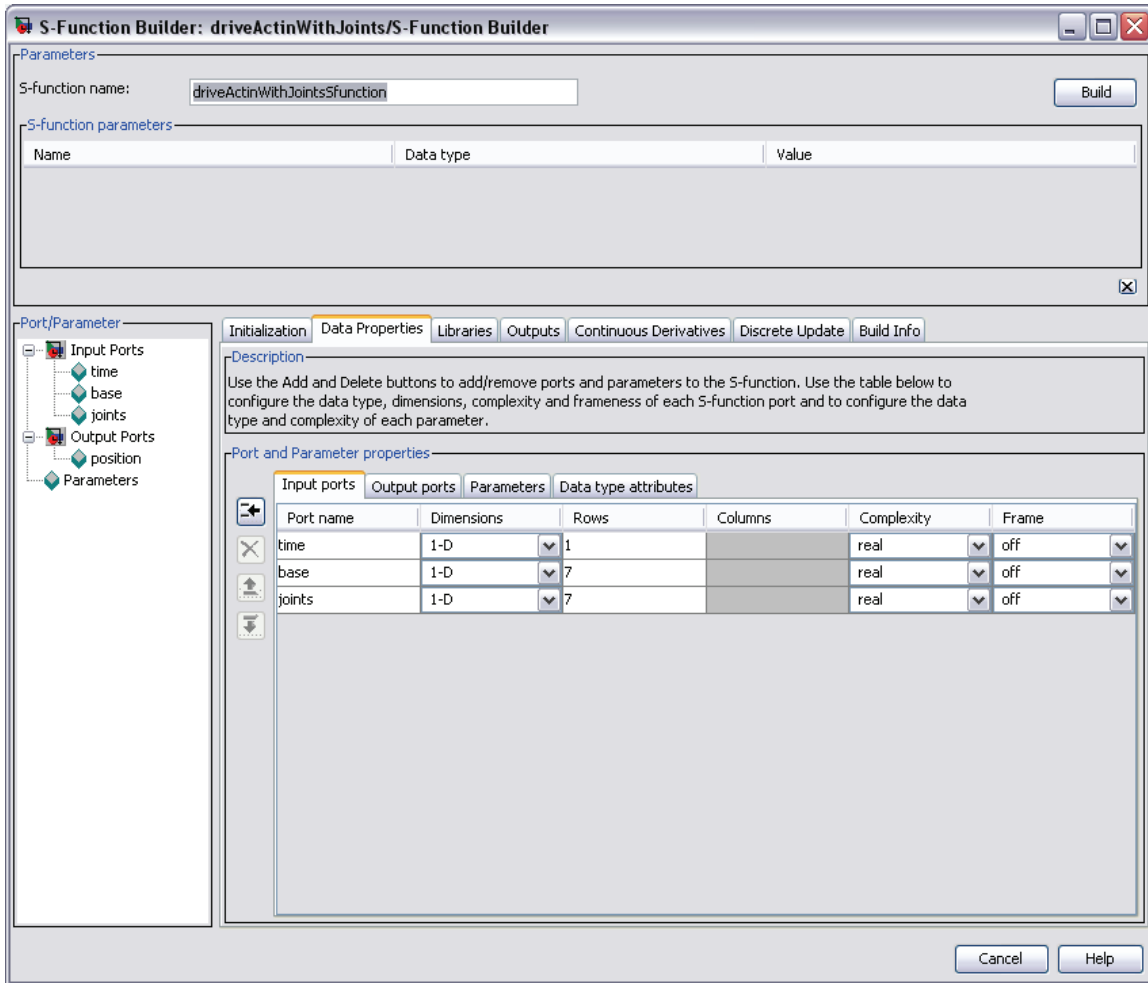
**Figure 22-11:** Enter responder mode so that the Actin viewer can display the rendered model with joint angles that reflect the angles sent from Simulink.

After entering responder mode, the viewer waits for Simulink to send commands. Figure 22-12 shows a running simulation with a model being rendered with dynamic joint angles. This figure also shows a plotting window which gives a time history of the joint angles. Note that the output is a direct feed through. No data is returned from Actin in this example.



**Figure 22-12:** Run Simulation.

As with the end-effector S-Function, the inputs and outputs are hardwired into the auto-generated C code. Figure 22-13 illustrates the input port page of the S-Function builder. The inputs and outputs can be modified through this GUI or the auto-generated source code can be edited.



**Figure 22-13:** Display of S-Function input ports.

## 23 Boost

### 1.1 What is Boost

Boost is a collection of C++ libraries. At the time of this writing, Boost version 1.34.1 contains 73 libraries of varying complexity. Many of the libraries were designed and implemented by industry experts, and all of the libraries are peer-reviewed by industry experts. At least 9 of the Boost libraries have been chosen for inclusion into Technical Report 1, or TR1, which is likely to be included in the next official standard of C++. This is a testament to the utility and quality of the Boost libraries.

## 1.2 Why use Boost

There are many reasons to use Boost libraries. The libraries are designed by experts with a focus on genericity, reusability, and portability. This means that the libraries are applicable to a wide range of problem domains, and they work on a wide range of platforms. The libraries are highly-orthogonal. This allows individual libraries to be used without incurring overhead from unused libraries, and it also implies synergy from combining multiple Boost libraries. Boost provides a window into the future of C++. As previously mentioned, many Boost libraries are already on the track to standardization in C++. The bottom line is that the proper understanding and application of Boost libraries improves productivity and code quality.

## 1.3 Overview of Boost libraries

The sheer number and scope of Boost libraries makes them difficult to summarize. This overview merely scratches the surface of several libraries that are widely used in Energid software. To obtain detailed, up-to-date documentation for the libraries, please visit <http://boost.org/libs/libraries.htm>.

### 1.3.1 Boost.Assign

Boost.Assign is a header-only library that eases the task of populating STL-compliant containers. Container objects, such as vectors and maps typically require multiple phase construction. In order to construct an instance of *EcStringVector* with three string elements, the vector construction must be followed by three calls to `push_back`.

```
EcStringVector v1;
v1.push_back("one");
v1.push_back("two");
v1.push_back("three");
```

This multi-phase construction can be eliminated with the use of `boost::assign::list_of`.

```
const EcStringVector v2 = boost::assign::list_of
    ("one") ("two") ("three");
```

Notice that single-phase construction allows the container object to be declared constant. This is not possible with multi-phase construction.

Boost.Assign can also be used to construct map container objects. The following example demonstrates populating an instance of *EcStringStringMap*.

```
EcStringStringMap m1;
m1["English"] = "Hello";
m1["Spanish"] = "Hola";
```

Using `boost::assign::map_list_of`, this task is greatly simplified.

```
const EcStringStringMap m2 = boost::assign::map_list_of
    ("English", "Hello") ("Spanish", "Hola");
```



### 1.3.2 Boost.Conversion

Boost.Conversion is a header-only library that provides `boost::lexical_cast`. The following example demonstrates using `boost::lexical_cast` to convert non-string types to a string.

```
assert(boost::lexical_cast<EcString>(10) == "10");
assert(boost::lexical_cast<EcString>(3.14159) == "3.14159");
```

The reverse conversion is also handled by `boost::lexical_cast`.

```
assert(boost::lexical_cast<EcInt32>("10") == 10);
assert(boost::lexical_cast<EcReal>("3.14159") == 3.14159);
```

Failed conversions will throw a `boost::bad_lexical_cast` exception.

```
try {
    boost::lexical_cast<EcInt32>("two")
} catch (boost::bad_lexical_cast&) {
    EcERROR("Unable to cast \"two\" to EcInt32!\n");
}
```

### 1.3.3 Boost.Filesystem

Boost.Filesystem provides an interface for portably dealing with file path operations. This library is not a header-only library; therefore, its usage requires linking in the binary “filesystem” library. The following demonstrates basic path operations:

```
// Use namespace alias to reduce line length in this example
namespace bfs = boost::filesystem;
// Construct paths
const bfs::path basePath = bfs::path("base/path",
bfs::native);
bfs::path examplePath = bfs::path(basePath / "dir");
// Append to path
examplePath /= "basename.ext";
// Basic operations
assert(examplePath.branch_path() == "base/path/dir");
assert(examplePath.leaf() == "basename.ext");
```

The library provides numerous convenience functions.

```
assert(bfs::basename(examplePath) == "basename");
assert(bfs::extension(examplePath) == ".ext");
assert(!bfs::exists(examplePath));
```

In order to get the current working directory, `boost::filesystem::current_path` can be used.

```
assert(bfs::is_directory(bfs::current_path()));
```

### 1.3.4 *Boost.Foreach*

Boost.Foreach is a header-only library that provides a simple mechanism for iterating over a container.

```
const EcStringVector strings = boost::assign::list_of
    ("one") ("two") ("three");
BOOST_FOREACH(const EcString& value, strings)
{
    EcPRINT("%s\n", value.c_str());
}
```

### 1.3.5 *Boost.Format*

Boost.Format is a header-only library that provides a type-safe C++ alternative to printf. It allows standard printf-style formatting.

```
const EcString value =
    str(boost::format("Real %.1f, Int %3d") % 3.1 % 10);
assert(value == "Real 3.1, Int 10");
```

It also has a more expressive style for formatting.

```
const EcString value =
    str(boost::format("Real %|1$0.1f|, Int %|2$3d|") % 3.1 %
10);
assert(value == "Real 3.1, Int 10");
```

Boost.Format can repeat arguments.

```
const EcString value =
    str(boost::format("Real %1%, Int %2%, Repeat First %1%")
% 3.1 % 10);
assert(value == "Real 3.1, Int 10, Repeat First 3.1");
```

An exception is thrown if the expected number of arguments do not match the actual number of arguments.

```
try {
    str(boost::format("Real %1%, Int %2%") % 3.1);
} catch (boost::io::too_few_args&) {
    EcERROR("Format is missing an argument!\n");
}
```

### 1.3.6 *Boost.Iostreams*

Boost.Iostreams provides a framework for building complex streams. This library is not a header-only library; therefore, its usage requires linking in the binary “iostreams” library.

With Boost.Iostreams, it is easy to add filters inline to a stream. For instance, an ostream to compress data while placing it into a string buffer can be defined as follows:

```
EcString buffer;
// Create the ostream
boost::iostreams::filtering_ostream out;
out.push(boost::iostreams::bzip2_compressor());
out.push(boost::iostreams::back_inserter(buffer));
```

The stream can now be used as expected, and the data will be compressed in the buffer.

```
// Serialize data to the buffer
out << 3.14 << " " << 10 << " " << EcString("Hello");
out.flush();
```

In order to read the stream, a corresponding istream can be created.

```
// Create the istream
boost::iostreams::filtering_istream in;
in.push(boost::iostreams::bzip2_decompressor());
in.push(boost::make_iterator_range(buffer));
```

Finally, the new istream can be used as expected, and the data will be decompressed when it is read from the buffer.

```
EcReal inR;
EcU32 inI;
EcString inS;
// Deserialize from the buffer
in >> inR >> inI >> inS;
```

### 1.3.7 Boost.Numeric

Boost.Numeric is a header-only library that provides **boost::numeric\_cast**. Normal casting operations, such as `static_cast`, do not detect the loss of range of a numeric type; therefore, casting between numeric types is a traditionally unsafe operation. To remedy this, Boost.Numeric provides **boost::numeric\_cast**, which throws exceptions during unsafe casts. The following example shows safe casts from an integer and double to a `EcU8` value:

```
assert(boost::numeric_cast<EcU8>(42) == EcU8(42));
assert(boost::numeric_cast<EcU8>(3.14) == EcU8(3));
```

If the value is too large to fit in the destination type, an exception is thrown.

```
try {
    boost::numeric_cast<EcU8>(256);
} catch (boost::numeric::positive_overflow&) {
    EcERROR("Unable to cast 256 to EcU8!\n");
}
```

```
}
}
```

Similarly, if a value is too small to fit in the destination type, an exception is thrown.

```
try {
    boost::numeric_cast<EcU8>(-1);
} catch (boost::numeric::positive_overflow&) {
    EcERROR("Unable to cast -1 to EcU8!\n");
}
```

Consistently using `boost::numeric_cast` allows some logic errors to be flagged at runtime. This prevents undefined behavior and reduces the burden of debugging problems that are difficult to track down. An actual example from a color interpolation routine is shown below.

```
// Can you spot the error?
const EcU8  startR  = 255;
const EcU8  stopR   = 0;
const EcReal dR     = boost::numeric_cast<EcReal>(stopR -
startR);
const EcU8  interpR = startR + boost::numeric_cast<EcU8>(dR);
```

The red color is being interpolated between a start value and an end value. Since, a red value can only contain a value between 0 and 255, *EcU8* is used as the type. When the starting red value is larger than the ending red value, *dR* contains a negative number, and a **`boost::numeric::bad_numeric_cast` exception** is thrown in the last line. The remedy to this is shown below:

```
const EcInt16 deltaR = boost::numeric_cast<EcInt16>(dR);
const EcU8  interpR = boost::numeric_cast<EcU8>(startR +
deltaR);
```

### 1.3.8 Boost.Program\_Options

`Boost.Program_Options` provides a framework for parsing command line options. This library is not a header-only library; therefore, its usage requires linking in the binary “`program_options`” library. The following example shows how to specify options:

```
// Use namespace alias to reduce line length in this example
namespace bpo = boost::program_options;

// Create the options description
bpo::options_description desc("Options");

// Add options
desc.add_options()
    // -h or -help flag : No associated value
    ("help,h", "Show this help message")
    // -i or -in-file : Value must be a string, not defaulted
```

```

(
  "in-file,i", bpo::value<EcString>(),
  "Name of the input file"
)
// -o or -out-file : Value must be a string, defaulted
(
  "out-file,o", bpo::value<EcString>()-
>default_value("default.out"),
  "Name of the output file"
)
// -s or -short-int : Value must be an unsigned int,
defaulted
(
  "start-int,s", bpo::value<EcU32>()->default_value(5),
  "The starting integer"
)
// -f or -real-value : Value must be a float, not defaulted
("real-value,f", bpo::value<EcReal>(), "The real value")
;

```

Positional options can also be specified. For instance the following command specifies that the first argument that does not follow an option flag should be treated as a value for “in-file”.

```

// First non-flag argument is in-file value
bpo::positional_options_description posOptions;
posOptions.add("in-file", -1);

```

After creating the options descriptions, the command line arguments can be parsed and stored in a map.

```

const EcInt32 argc = 4;
char* argv[argc] = {"thisProgram", "someInputFile.in", "-f",
"3.14"};
bpo::parsed_options parsed =
bpo::command_line_parser(argc,argv)
  .options(desc)
  .positional(posOptions)
  .run();
bpo::variables_map vm;
bpo::store(parsed, vm);
bpo::notify(vm);

```

Finally, the application logic can work with the stored option values.

```
if (vm.count("help"))
{
    // Print a usage message
    std::cout << desc << std::endl;
}
else if (!vm.count("in-file"))
{
    EcERROR("Missing an input file argument!\n");
}
else
{
    // See if real-value was specified on the command line
    const EcReal realValue =
        (vm.count("real-value")) ? vm["real-value"].as<EcReal>()
: 0.0;
    // First non-flagged argument, thanks to positional options
    assert(vm["in-file"].as<EcString>() == "someInputFile.in");
    // Still has the default value
    assert(vm["out-file"].as<EcString>() == "default.out");
    // Still has the default value
    assert(vm["start-int"].as<EcU32>() == EcU32(5));
    assert(realValue == 3.14);
}
```

### 1.3.9 Boost.Regex

Boost.Regex provides a native C++ regular expression library. This library is not a header-only library; therefore, its usage requires linking in the binary “regex” library. The first step in using the library is to create the regular expression.

```
const boost::regex e("^Hello, ?\\s+(\\S+)$");
```

This expression can then be used to test arbitrary strings for a match.

```
// Good match
assert(boost::regex_match("Hello Bob", e));
// Bad match: begins with a space
assert(!boost::regex_match(" Hello Bob", e));
// Bad match: need at least one whitespace character after
Hello
assert(!boost::regex_match("HelloBob", e));
```

```

// Good match: optional comma
assert(boost::regex_match("Hello, Bob", e));

// Bad match: need at least one whitespace character after
comma
assert(!boost::regex_match("Hello,Bob", e));

// Good match: more than one whitespace is OK
assert(boost::regex_match("Hello   World", e));

```

### 1.3.10 *Boost.Signals*

Boost.Signals provides a powerful signals and slots callback mechanism. This library is not a header-only library; therefore, its usage requires linking in the binary “signals” library. Signals are an ideal way to inform observers when a particular event occurs. The following code creates a signal.

```

// Create a signal that takes a single argument and returns
void
//   The argument is a reference to EcStringVector
boost::signal<void (EcStringVector&)> notificationSignal;

```

The observers must be convertible to functions with the same signature as the signal. In this case, there are two observer functors.

```

struct Observer1
{
    void operator() (EcStringVector& sv) const
    {
        sv.push_back("Observer1");
    }
};

struct Observer2
{
    void operator() (EcStringVector& sv) const
    {
        sv.push_back("Observer2");
    }
};

```

The observers must be registered with the signal.

```

// Connect observer slots to the signal
notificationSignal.connect(Observer1());
notificationSignal.connect(Observer2());

```

```

// Observer1 is added twice, so the observer will be notified
twice
notificationSignal.connect(Observer1());

```

Finally, the signal can be triggered, and the observer functors will be called.

```

// Trigger the signal
EcStringVector sv;
notificationSignal(sv);

```

### 1.3.11 *Boost.Smart\_Ptr*

`Boost.Smart_Ptr` is a header-only library that provides several variants of smart pointers. This library is perhaps the most widely used of all Boost libraries. The primarily-used smart pointers are `boost::scoped_ptr`, `boost::shared_ptr`, and `boost::weak_ptr`. Here is a simple Resource class that is used in the examples.

```

struct Resource
{
    EcU32 value;
};

```

A `boost::scoped_ptr` automatically frees its managed pointer when the pointer is no longer used or when the pointer goes out of scope. Since the scope of an object ends when an exception is thrown, the `boost::scoped_ptr` works correctly in the presence of exceptions. Exception-safe code with pointers is often impossible without the use of a smart pointer, such as `boost::scoped_ptr`.

```

// Begin scope
{
    typedef boost::scoped_ptr<Resource> ResourceScopedPtr;
    ResourceScopedPtr ptr(new Resource);
    // First Resource pointer is automatically freed
    ptr.reset(new Resource);
    // Second Resource pointer is automatically freed at scope
end
}

```

A `boost::scoped_ptr` is noncopyable, and it does not allow sharing of its managed resource. For this reason, it is not possible to create STL containers of such pointers. If it is necessary to share resources, copy smart pointers, or store smart pointers in a STL container, then a `boost::shared_ptr` can be used.

```

// Begin scope
{
    typedef boost::shared_ptr<Resource> ResourceSharedPtr;
    typedef std::vector<ResourceSharedPtr> ResourcePtrVector;

```



```

ResourcePtrVector resourcePtrs;
for (EcU32 i = 0; i < 10; ++i)
{
    ResourceSharedPtr ptr(new Resource);
    resourcePtrs.push_back(ptr);
    ptr->value = i;
    // ptr goes out of scope, and the Resource pointer is
not
    // not freed because it is still used in resourcePtrs
}
// All Resource pointers in resourcePtrs are freed at scope
end
}

```

Using **boost::shared\_ptr**, there can be many references to the same resource. As long as a single **boost::shared\_ptr** is managing a resource, it will not be freed. At times, there is a need to observe the resource in a **boost::shared\_ptr** without actually referencing that resource. The **boost::weak\_ptr** is designed for this task.

```

typedef boost::weak_ptr<Resource> ResourceWeakPtr;
ResourceWeakPtr weakPtr;
// The weak_ptr is not referencing a valid shared_ptr
assert(weakPtr.expired());
// Begin scope
{
    typedef boost::shared_ptr<Resource> ResourceSharedPtr;
    ResourceSharedPtr ptr(new Resource);
    weakPtr = ptr;
    // The weak_ptr is now referencing a valid shared_ptr
    assert(!weakPtr.expired());
    // ptr goes out of scope, and the Resource pointer is freed
}
// The weak_ptr is no longer referencing a valid shared_ptr
assert(weakPtr.expired());

```

### 1.3.12 *Boost.Thread*

Boost.Thread provides a portable interface for creating threaded applications. This library is not a header-only library; therefore, its usage requires linking in the binary “thread” library. The fundamental primitive of the library is **boost::mutex**. This primitive serializes access to a specific section of code. The mutex is usually used to guard resources that are shared

across threads. Normally, a **boost::mutex::scoped\_lock** is used to serialize access to a section of code. The scoped lock waits to acquire a lock on a mutex, and it automatically releases the lock on the mutex when it goes out of scope.

```
// Create a mutex (shared across threads)
boost::mutex m;

// Mutex m must be shared across threads that run the
following code
{
    // Wait until mutex m is unlocked, then acquire the lock
    boost::mutex::scoped_lock lock(m);
    // A lock on mutex m is acquired
    // ... operations that must be serialized
    // lock goes out of scope, and the lock on mutex m is
released
}
```

Sometimes it is desirable to wait for a particular condition prior to processing logic in a thread. This can be accomplished with **boost::condition**.

```
// Create a condition and a mutex (shared across threads)
boost::condition c;
boost::mutex m;

// In one thread
{
    boost::mutex::scoped_lock lock(m);
    // A lock on mutex m is acquired
    // Release the lock on mutex m until the condition c is
triggered
    c.wait(lock);
    // A lock on mutex m is acquired (again)
    EcPRINT("Notified of condition c!\n");
    // ... operations that should run upon notification of
condition c
    // lock goes out of scope, and the lock on mutex m is
released
}

// In another thread
{
    // ... do some processing to determine when to trigger
```

```

condition
    // Trigger the condition
    c.notify_all();
}

```

Unfortunately, if a condition is never triggered, the wait will never return. It is often necessary to place a timeout on the wait. This can be accomplished with **boost::condition::timed\_wait** and a **boost::xtime** object.

```

{
    boost::mutex::scoped_lock lock(m);

    boost::xtime expiration;
    // Get the current time
    boost::xtime_get(&expiration, boost::TIME_UTC);
    // Expire one second from now
    expiration.sec += 1;
    // Wait for condition c with a timeout
    if (c.timed_wait(lock, expiration))
    {
        // A lock on mutex m is acquired
        EcPRINT("Notified of condition c!\n");
    }
    else
    {
        EcPRINT("Timeout during wait for condition c!\n");
    }
    // lock goes out of scope, and the lock on mutex m is
    released
}

```

A thread can be commanded to sleep using **boost::thread::sleep** and a **boost::xtime** object.

```

boost::xtime sleepExpiration;
// Get the current time
boost::xtime_get(&sleepExpiration, boost::TIME_UTC);
// Expire sleep two seconds from now
sleepExpiration.sec += 2;
boost::thread::sleep(sleepExpiration);

```

A thread is created by constructing a **boost::thread** object. The thread constructor takes a thread function as an argument. Here is an example thread functor.

```

class AlarmFunc
{
public:
    AlarmFunc
        (
            EcU8          timeoutSec,
            boost::condition& startCondition
        ) :
        m_StartCondition(startCondition),
        m_Mutex(),
        m_Timeout(timeoutSec)
    {}

    void operator() ()
    {
        EcPRINT("AlarmFunc awaiting start command!\n");
        boost::mutex::scoped_lock lock(m_Mutex);
        m_Condition.wait(lock);
        EcPRINT("AlarmFunc started!\n");
        boost::xtime expiration;
        boost::xtime_get(&expiration, boost::TIME_UTC);
        expiration.sec += m_Timeout;
        boost::thread::sleep(expiration);
        EcPRINT("AlarmFunc exiting!\n");
    }
private:
    boost::condition& m_StartCondition;
    boost::mutex      m_Mutex;
    EcU8              m_Timeout;
};

```

It is often convenient to create a thread pointer in order to delay spawning the thread.

```

typedef boost::scoped_ptr<boost::thread> ThreadPtr;
ThreadPtr threadPtr1;
ThreadPtr threadPtr2;
// Neither thread is currently running
boost::condition startCondition;
AlarmFunc func1(EcU8(2), startCondition);

```

```
AlarmFunc func2(EcU8(4), startCondition);  
// Spawn the threads  
threadPtr1.reset(new boost::thread(func1));  
threadPtr2.reset(new boost::thread(func2));  
  
// Start the alarms  
startCondition.notify_all();  
  
// Wait for the threads to exit  
threadPtr1->join();  
threadPtr2->join();
```

## 24 Bibliography

- [1] *Robotic Manipulators: Mathematics, Programming, and Control*, R.P. Paul, Cambridge, MA: MIT Press, 1981.
- [2] *Introduction to Robotics*, J.J. Craig, Reading, MA: Addison-Wesley, 1989.
- [3] "Animating Rotation with Quaternion Curves," K. Shoemake, SIGGRAPH, vol. 19, no. 3, San Francisco, CA, July 22-26, 1985, pp. 245-254.
- [4] D.H. Eberly, *3D Game Engine Design*, Morgan Kaufmann Publishers, 2000.
- [5] SEDRIS Web Site, [www.sedris.org](http://www.sedris.org).
- [6] "Review of Pseudoinverse Control for Use with Kinematically Redundant Manipulators," C.A. Klein and C.H. Huang, *IEEE Trans. on Sys., Man, and Cybernetics*, vol. SMC-13, pp. 245-250, Mar./Apr. 1983.
- [7] "Robot Manipulability," K.L. Doty, C. Melchiorri, E.M. Schwartz, and C. Bonivento, *IEEE J. Robot. Automat.*, vol. 11, pp. 462-468, June 1995.
- [8] "Task Space Tracking with Redundant Manipulators," O. Egeland, *IEEE J. Robot. Automat.*, vol. RA-3, pp. 471-475, Oct. 1987.
- [9] "Configuration Control of Redundant Manipulators: Theory and Implementation," H. Seraji, *IEEE Trans. Robot. Automat.*, vol. 5, pp. 472-490, Aug. 1989.
- [10] "Kinematic Programming Alternatives for Redundant Manipulators," J. Baillieul, *Proc. 1985 IEEE Int. Conf. Robot. Automat.*, St. Louis, MO, Mar. 25-28, 1985, pp. 722-728.
- [11] "Improved Configuration Control for Redundant Robots," H. Seraji and R. Colbaugh, *J. Robot. Syst.*, vol. 7, no. 6, pp. 897-928, 1990.
- [12] "Optimal Rate Allocation in Kinematically Redundant Manipulators—The Dual Projection Method," M.Z. Huang and H. Varma, *Proc. 1991 IEEE Int. Conf. Robot. and Automat.*, Philadelphia, PA, Apr 24-29, 1988, pp.28-36.
- [13] "An Efficient Gradient Projection Optimization for Manipulators with Multiple Degrees of Redundancy," H. Zghal, R.V. Dubey, and J.A. Euler, *Proc. IEEE 1990 Int. Conf. Robot. and Automat.*, Cincinnati, OH, May 13-18, 1990, pp. 1006-1011.
- [14] "On the Implementation of Velocity Control for Kinematically Redundant Manipulators," J.D. English and A.A. Maciejewski, *IEEE Trans. on Sys., Man, and Cybernetics—Part A: Systems and Humans*, vol. 30, no. 3, May 2000, pp. 233-237.
- [15] "Efficient Dynamic Computer Simulation of Robotic Mechanisms," M.W. Walker and D.E. Orin, *Journal of Dynamic Systems, Measurement, and Control*, vol. 104, Sep. 1982, pp. 205-211.
- [16] "Fault Tolerance for Kinematically Redundant Manipulators: Anticipating Free-Swinging Joint Failures," J.D. English and A.A. Maciejewski, *IEEE Transactions on Robotics and Automation*, vol. 14, pp. 566-575, 1998.
- [17] *Robot Dynamics Algorithms*, R. Featherstone, Kluwer Academic Publishers, 1987.
- [18] "On-Line Computational Scheme for Mechanical Manipulators," J.Y.S. Luh, M.W. Walker, and R.P.C. Paul, *Journal of Dynamic Systems, Measurement, and Control*, vol. 102, pp. 69-76, June 1980.
- [19] "The Minimum Form of Strength in Serial, Parallel and Bifurcating Manipulators," R.O. Ambrose and M.A. Diftler, *Proceeding of the 1998 IEEE International Conference on Robotics and Automation*, Leuven, Belgium, May 1998.
- [20] D. Baraff, "Coping with Friction for Non-Penetrating Rigid Body Simulation," Siggraph '91, Las Vegas, vol. 25, no. 4, July 1991, pp. 31-40.

- [21] K.C. Cheok, H. Hu, and N.K. Loh, "Modeling and Identification of a Class of Servomechanism Systems with Stick-Slip Friction," *Journal of Dynamic Systems, Measurement, and Control*, vol. 110, Sept 1988, pp 324-328
- [22] D. Karnopp, "Computer Simulation of Stick-Slip Friction in Mechanical Dynamic Systems," *Transactions of the ASME*, vol. 107, March 1985, pp. 100-103.
- [23] M.W. Walker and D.E. Orin, "Efficient Dynamic Computer Simulation of Robotic Mechanisms," *Journal of Dynamic Systems, Measurement, and Control*, 104, 205-211, 1982.
- [24] A. Fijany and A.K. Bejczy, "An Efficient Algorithm for Computation of Manipulator Inertia Matrix," *Journal of Robotic Systems*, 7(1), 57-80, 1990.
- [25] R. Featherstone, *Robot Dynamics Algorithms*, Kluwer Academic Publishers, Boston, 1987.
- [26] K.W. Lilly, *Efficient Dynamic Simulation of Robotic Mechanisms*, Kluwer Academic Publishers, Boston, 1993.
- [27] K.C. Cheok, H. Hu, and N.K. Loh (1988), "Modeling and Identification of a Class of Servomechanism Systems with Stick-Slip Friction," *Journal of Dyn. Systems, Meas., and Control*, vol. 110, Sept 1988, pp 324-328
- [28] D. Karnopp (1985) "Computer Simulation of Stick-Slip Friction in Mechanical Dynamic Systems," *Transactions of the ASME*, vol. 107, March 1985, pp. 100-103.
- [29] M.W. Spong (1987), "Modeling and Control of Elastic Joint Robots," *Journal of Dynamic Systems, Measurement, and Control*, vol. 109, pp. 310-319, 1987.
- [30] J-H Yang and L-C Fu (1996), "Nonlinear Adaptive Control for Manipulator System with Gear Backlash," *Proc. 35<sup>th</sup> Conf. Decision and Control*, Kobe, Japan, Dec 1996, pp. 4369-4374.
- [31] S. Queen, K. London, and M. Gonzalez, "Momentum-Based Dynamics for Spacecraft with Chained Revolute Appendages," NASA 2005 Flight Mechanics Symposium, Greenbelt, MD, October 18-20, 2005.
- [32] "On the Implementation of Velocity Control for Kinematically Redundant Manipulators," J.D. English and A.A. Maciejewski, *IEEE Trans. on Sys., Man, and Cybernetics—Part A: Systems and Humans*, vol. 30, no. 3, May 2000, pp. 233-237.
- [33] S. Sykora, "Volume Integrals over n-Dimensional Ellipsoids," *Stan's Library*, v. 1, 2005.  
<http://www.ebyte.it/library/docs/math05a/nDimEllipsoidVolumes05.html>
- [34] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, M. Stal: *Pattern-Oriented Software Architecture – A System of Patterns*, John Wiley & Sons, 1996
- [35] The Boost Iostreams Library. Retrieved 11:30, May 25, 2007, from <http://boost.org/libs/iostreams/doc/index.html>
- [36] Roger M. Needham and David J. Wheeler. "Tea extensions." Technical report, Computer Laboratory, University of Cambridge, October 1997
- [37] Feistel cipher. (2007, February 26). In *Wikipedia, The Free Encyclopedia*. Retrieved 22:18, February 28, 2007, from [http://en.wikipedia.org/w/index.php?title=Feistel\\_cipher&oldid=111030203](http://en.wikipedia.org/w/index.php?title=Feistel_cipher&oldid=111030203)
- [38] XTEA. (2007, January 3). In *Wikipedia, The Free Encyclopedia*. Retrieved 00:47, March 1, 2007, from <http://en.wikipedia.org/w/index.php?title=XTEA&oldid=98176875>

- [39] XTEA. (2007, January 3). In *Wikipedia, The Free Encyclopedia*. Retrieved 00:47, March 1, 2007, from <http://en.wikipedia.org/w/index.php?title=XTEA&oldid=98176875>
- [40] Skype API. Retrieved 06:00, March 1, 2007, from [https://developer.skype.com/Docs/ApiDoc/Skype\\_API\\_reference](https://developer.skype.com/Docs/ApiDoc/Skype_API_reference).