Abstract

This white paper describes Energid Technologies’ kinematic control software, which is part of the Actin™ toolkit. Actin provides generic control for kinematically redundant serial and bifurcating manipulators. It uses a patented configurable augmented Jacobian control technique that is implemented within a software framework that allows the easy creation of supervised and traded control systems using C++. Through Extensible Markup Language (XML) configuration, Energid’s system allows the accommodation of even the most complex mechanisms, multiple end-effector constraints, different optimization criteria, and control-system exchange at runtime.
Robot Kinematic Control

1 Introduction

A roboticist wanting to control a manipulator faces the problem that direct motor control is only available for the joints, while it is the hand that does useful work. Velocity control is the study of how to move the joints to properly move the hand. It includes many techniques, such as pseudoinverse control [1], weighted pseudoinverse control [2], augmented Jacobian techniques [3], [4], extended Jacobian techniques [5], [6], and projection methods [7], [8]. Velocity control contrasts with position control, which is the study of how to position the joints to properly place the hand. Velocity and position control are also often called, respectively, inverse velocity kinematics and inverse position kinematics. Either can be used to indirectly solve for the other through differentiation or integration.

Using velocity control as the core capability is easier than using position control. Direct position control requires solving simultaneous nonlinear equations in the joint variables. For complex arms there is usually no closed-form position-control solution. Velocity control, on the other hand, is linear (in the joint rates), and if a solution exists, it is usually in closed form. The significance of this is increased when working with arms having many (more than six, say) joints. For these types of arms, the position control solution requires establishing meaningful constraints, and these constraints complicate the solution. But velocity control can easily provide solutions that optimize complex and useful criteria, and it can be integrated to indirectly give position control.

Six degrees of freedom are required to arbitrarily place the hand of a serial manipulator, and any serial manipulator with more than six joints is kinematically redundant. Generally, such a manipulator can place a hand or tool at a desired position and orientation (or pose) within its workspace in an unlimited number of ways. Similarly, a bifurcating (i.e., branching), two-handed robot with more than 12 degrees of freedom can place its two hands at specified poses within its workspace in an unlimited number of ways. Energid’s Actin toolkit focuses on just these types of kinematic problems.

This white paper is underpinned by Energid’s belief that the time is right for applying complex robotic systems to real-world problems. Faster computers and novel algorithms create new opportunities. And the algorithmic gains contribute at least equally, if not more, than hardware gains to the expansion of capability. (How, for example, can one compare the instantaneous leap of the Fast Fourier Transform algorithm to the evolutionary growth of computer hardware?) The conceptual and computational framework Energid has developed enables the exploitation of new hardware and software.

2 Philosophy

2.1.1 Conceptual Unification

Energid’s framework unifies many conceptual techniques under one method. This allows engineers to more easily understand alternative algorithms. It is easier to understand a single, general technique, even if it is abstract, than to read 10 or 20 specific papers on different
techniques by different authors and understand them all. The framework provides just this facility.

2.1.2 Computational Unification

The Actin framework unifies the method of computation and testing of velocity-control techniques. This means that an engineer is able to implement many control techniques simply by editing an XML configuration file.

2.1.3 Robustness

A singularity occurs when a manipulator cannot achieve arbitrary motion of the hand. (The term motion here means linear and angular velocity.) A human, for example, experiences a kinematic singularity when an arm is fully outstretched and velocity of the hand away from the body can no longer be achieved. Unless the phenomenon is addressed, near a singularity joint rates become high and unwieldy.

2.1.4 A Common Language

A key innovation provided through Energid’s software is an XML-based data-exchange language for controlling kinematic manipulators. A common velocity-control description language makes manipulator control easy to implement and test on all types of platforms.

3 Components

3.1.1 Kinematic Control

Actin is primarily a toolkit for the kinematic control of robotic manipulators. Using velocity control as the core technique, it 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 physical description. Manipulators with any number of links, any number of bifurcations (branches), nonstandard joint types, and nonstandard end-effector types are supported.

3.1.2 XML Configuration

Components are configurable using the Extensible Markup Language, and a developer can easily connect 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 XML schemas. So a control system designed and built with Actin will automatically have a corresponding XML language that can be used with other commercial software products.

3.1.3 Toolkit Implementation

Behavior of the control system can be changed through XML configuration, subclassing, and dynamic link libraries. Existing components are fully parameterizable through XML descriptions that can be changed using Energid’s editing tools, XML editing tools, or even text editors. Virtually all components within the Actin Toolkit are implemented as virtual classes.
that can be extended through subclassing. And the control-system database is implemented using containers and variable vectors that can load new components through dynamic link libraries.

Figure 1 Actin provides generic software for selecting joint positions and velocities that give desired end-effector positions and velocities. This is an illustration of Actin-generated control for the RRC K-1207i.

4 Kinematic Control

4.1.1 Core Algorithmic Framework
The core velocity framework in Actin is based on the manipulator Jacobian equation:

\[ V = J(q) \dot{q} \] (1)

where \( V \) 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); \( q \) is the \( n \)-length vector of joint positions (with \( \dot{q} \) being its time derivative); and \( J \) is the \( m \times n \) manipulator Jacobian, a function of \( q \). (For spatial arms with a single end effector, \( V \) 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.)

For any physical manipulator that is not self-connecting, a manipulator Jacobian can be defined to make equation (1) true. When the manipulator is kinematically redundant, the dimension of \( \mathbf{V} \) is less than the dimension of \( \mathbf{q} \) \((m<n)\), and (1) is underconstrained when \( \mathbf{V} \) is specified. By using \( \mathbf{V} \) to represent relative motion, (1) can support self-connecting mechanisms by setting the relative motion to zero.

The velocity control question then is the following: given a desired hand motion \( \mathbf{V} \), what are the joint rates \( \mathbf{\dot{q}} \) that best achieve this motion? To answer this, the framework is built on that described in [9], which uses a scalar \( \alpha \), a matrix function \( \mathbf{W}(\mathbf{q}) \), and a scalar function \( f(\mathbf{q}) \) to solve for \( \mathbf{\dot{q}} \) given \( \mathbf{V} \) through the following formula:

\[
\mathbf{\dot{q}} = \left[ \mathbf{N}_J^T \mathbf{W} \right]^{-1} \left[ \mathbf{V} - \alpha \mathbf{N}_J^T \nabla f \right],
\]

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, \( \mathbf{JN}_J = 0 \), 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.

Actin goes beyond the formulation in (2), however, to create a more general framework. Instead of insisting on the use of the gradient of a function, it uses a general column vector \( \mathbf{F}(\mathbf{q}) \). Not all vector functions are gradients. This minor, but important, modification yields the following formula:

\[
\mathbf{\dot{q}} = \left[ \mathbf{N}_J^T \mathbf{W} \right]^{-1} \left[ \mathbf{V} - \alpha \mathbf{N}_J^T \mathbf{F} \right].
\]

4.1.2 End-Effectector Descriptions

The array \( \mathbf{V} \), as used above, represents the motion of all the manipulator’s end effectors. A special class holds the description of an end-effectector set, which contains any number of any type of end effectors. A point end effector, for example, gives three degrees of constraint, and a frame gives six. Each end effector is described using a string identifier for the link to which it is rigidly attached as well as an offset description. For forming kinematic entities—such as Jacobians—the end effectors are concatenated.

4.1.3 Local Position Control

In Actin, velocity control is used to define local position control through the following procedure: A measure of the position of the end-effector (3D position for a spatial point end effector, for example) and the desired position are used to construct an end-effector velocity that, if followed, will give alignment.

As an illustrative example, for point end effectors, the desired velocity of the point can simply be a scalar gain times the difference in position. That is, if \( \mathbf{p}_a \) is the actual position and \( \mathbf{p}_d \) is the
desired position, then the desired velocity, $\dot{v}_d$, can be set by the following, where $k_i$ is a positive gain:

$$\dot{v}_d = k_i \cdot (\ddot{p}_d - \ddot{p}_a).$$  \hspace{1cm} (4)

### 4.1.4 Simulation

Actin’s simulation capability includes multiple, full manipulator articulated dynamics models and impact dynamics simulation. Special algorithms for both large and small systems are provided. These simulations can be used to validate the kinematic control, or they can be embedded in the kinematic controller to give new types of control.

### 4.2 Implementation

#### 4.2.1 Physical Manipulator Description

The 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 is decomposed into any number of manipulators, each of which is represented through any number of links in a tree structure. The state is decomposed into a velocity and a position state, manipulator-by-manipulator. Figure 2 illustrates this organization. Separating system and state allows easy logging, check pointing, and storage of the dynamic information.

![Diagram: Stated System](image)

**Figure 2** The representation of the manipulators and the environment is organized into a *system* and a *state*. The state changes time step to time step, while the system remains static.
Every manipulator description is stored as a link tree, where any link can have any number of child links. One base link provides reference information. As part of the design, this link can be any rigid body in the manipulator. All links have a unique string label and integer identifier. Link integer identifiers use depth-first ordering. This organization is illustrated in Figure 3.

![Figure 3](image)

**Figure 3** The links in the system are organized into a tree. One link, the base link, plays a special role as a reference. Children have access to the parent.

The location of the base of each manipulator is specified through a sequence of transformations. The entire system is represented with respect to a universal reference frame as part of the environmental specification. Then, each manipulator has a static base frame whose location is specified as part of the manipulator system. The dynamic location of the base frame with respect to the static base frame is specified as part of the state, and changes with each time step. This is illustrated in Figure 4.

![Figure 4](image)

**Figure 4** Location of a manipulator’s base frame. The dynamic base frame location for each manipulator is identified through a sequence of transformations.

### 4.2.2 Link Description

Each link in the tree describing a manipulator holds the following information: 1) kinematic data, 2) mass properties, 3) actuation parameters, 4) physical extent, 5) surface properties, and 6) bounding volume.

The kinematic data, describes the connections between links. Energid’s Actin toolkit supports Paul's (Koivo’s) Denavit-Hartenberg notation [10], with the sequence z-rotation, z-translation, x-
rotation, x-translation, and it also supports Craig’s notation [11]: x-rotation, x-translation, z-rotation, z-translation, and general transformations. The implementation supports prismatic, rotational, and screw-type joints. It supports a separate frame for describing the physical extent, end effectors, and mass properties. This is illustrated below.

**Figure 5** The location of the joint axis with respect to the proximal frame can be specified using a number of description methods, including Paul’s or Craig’s Denavit-Hartenberg notation. A special, primary frame is used for link data.

In this formulation, the link conceptually contains the joint. The distal frame of a link is rigidly attached to the proximal frame of a child. This provides a more flexible way to represent kinematic properties. It allows multiple formalisms (such as Paul or Craig) to be used, and it supports new types of joints and flexibility properties.

The link’s mass properties include the 10 unique scalars needed for dynamics calculation through their inclusion in the scalar mass, the vector first moment of inertia, and the $3 \times 3$ symmetric second moment of inertia.

The link’s physical extent is stored using a triangles, polygons, 3D shapes, or unions of any of these. This approach is flexible and general. One powerful component of Actin is the surface-property specification method. The specification of surface properties allows improved rendering, simulation, and intelligent reasoning.

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 a 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 Figure 6.
Each link holds a string-to-surface-property map. Polygons, triangles, and shapes specify their surface property using a string reference. Each surface property maintains string-string, string-float, and string-integer maps, which are configurable by the user through XML. They can represent virtually any properties, such as flexibility or fragility. The example shown here is color.

Every link also maintains a hierarchy of bounding volumes that are used for fast distance and intersection calculation.

### 4.2.3 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 represent the repulsive force or torque that is proportional to the incursion within a specified zone of the hard stop.

### 4.2.4 End Effectors

Just as it is possible to exchange control systems (to minimize kinetic energy or avoid obstacles, for example), it is possible to exchange end-effector descriptions at run time using Actin. Most end effectors are rigidly attached to some link on the manipulator, and they can be attached in any way. Point end effectors, for example, can be attached with any offset, and frame end effectors can be attached with any offset and rotation. Some end-effectors are not attached to a specific link—examples include center-of-mass constraint and spatial momentum constraint. End-effector types are listed below:

- 2D positioning
- 3D position
- 3D orientation
- 3D position and orientation
- 3D center of mass
- Linear joint-value constraints
- Spatial momentum constraint
All of these can be used as 1) hard constraints or 2) soft constraints that are balanced with other optimization criteria.

4.2.5 **Velocity Control System**

Large, fielded control systems are typically complex, with many patches and logical paths added by the development team. These fielded control systems transcend the logically encapsulated equations found in textbooks and academic papers. Actin embraces this phenomenon by organizing the control system and all supporting mathematics into a flexible logical tree that can be represented in XML.

This logical tree can better represent additions and modifications that otherwise might be hard coded into the control system. It also provides a method of organization that supports dynamic programming—the storage of subproblem solutions to prevent duplicate calculation. The logical tree is built from containers and expressions. These families of objects are described through XML and are connected together in code to produce unique functions. Containers and elements are described below.

4.2.5.1 **Containers**

A container performs three tasks. It 1) holds a single expression element, which is the root of a tree, 2) is able to create any type of expression element that can lie in the tree beneath it, and 3) provides access to the state of the manipulator system. A single container is the interface between the expression elements and the rest of the code.

4.2.5.2 **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 types of expression elements: branch and leaf elements. Branch elements have children that are expression elements. Leaf elements have no children. The control expression tree is formed using branch elements terminated by leaf elements.

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.
Figure 7 For building a control system, the container holds the root element in the expression tree, which is composed of branch (white) and leaf (gray) elements. All elements hold a pointer to the container (only two are illustrated).

The basic expression element types are listed in the tables below.

<table>
<thead>
<tr>
<th>Type</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Plus</td>
<td>Addition</td>
</tr>
<tr>
<td>Minus</td>
<td>Subtraction</td>
</tr>
<tr>
<td>Times</td>
<td>Multiplication</td>
</tr>
<tr>
<td>Element Inverse</td>
<td>1/x</td>
</tr>
<tr>
<td>Negative</td>
<td>-x</td>
</tr>
<tr>
<td>Sine</td>
<td>sin(x)</td>
</tr>
<tr>
<td>Cosine</td>
<td>cos(x)</td>
</tr>
<tr>
<td>Transpose</td>
<td>(·)T – matrix transpose</td>
</tr>
</tbody>
</table>

Table 1 Basic Branch Elements, their C++ class, and their meaning.
<table>
<thead>
<tr>
<th>Type</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Constant</td>
<td>Constant floating-point value</td>
</tr>
<tr>
<td>Diagonal Matrix</td>
<td>Matrix with arbitrary diagonal</td>
</tr>
<tr>
<td>General Column</td>
<td>Column with arbitrary values</td>
</tr>
<tr>
<td>Identity Matrix</td>
<td>The identity matrix</td>
</tr>
<tr>
<td>Joint Value</td>
<td>State joint value</td>
</tr>
<tr>
<td>Single Element Column</td>
<td>Column with one nonzero value</td>
</tr>
<tr>
<td>Single Element Matrix</td>
<td>Matrix with one nonzero value</td>
</tr>
</tbody>
</table>

**Table 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 of manipulator configuration. In addition, the table below gives leaf elements to directly implement specific types of control.

<table>
<thead>
<tr>
<th>Type</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Accuracy Measure Gradient</td>
<td>The gradient of the error sensitivity for a single joint</td>
</tr>
<tr>
<td>Collision Avoidance</td>
<td>A column of joint rates that can be used to drive the arm away from self-collision</td>
</tr>
<tr>
<td>Column Table Function</td>
<td>The identity matrix</td>
</tr>
<tr>
<td>Joint Limit Avoidance</td>
<td>Joint rates that can be used to drive away from joint limits</td>
</tr>
<tr>
<td>Joint Torque Squared Gradient</td>
<td>The gradient of the torque squared on a single joint</td>
</tr>
<tr>
<td>Mass Matrix</td>
<td>The manipulator inertia matrix</td>
</tr>
<tr>
<td>Obstacle Avoidance</td>
<td>A column of joint rates that can be used to drive the arm away from environmental obstacles</td>
</tr>
<tr>
<td>Potential Energy Gradient</td>
<td>Gravity-based potential energy gradient</td>
</tr>
<tr>
<td>Singularity Avoidance</td>
<td>The gradient of a measure of singularity proximity</td>
</tr>
</tbody>
</table>

**Table 3** Extended Leaf Elements, their C++ class, and their meaning.

A special expression element in the tree defines the core control system. The core control system requires information on the manipulator system, end effectors, and desired end-effector motion that it accesses through the top-level container. The matrix, vector, and scalar functions that are used in equation (3) are described using the expression tree, as illustrated in Figure 8.
Filters can be used to limit a measure of the joint rates or measures of end-effector error. Multiple filter elements can be daisy-chained to give flexible filtering. The form of the filters is shown in Figure 9.

**Figure 8** The core velocity control system. It has three children in the control tree—one each for the matrix, vector, and scalar used in equation (3). This is used with system information provided through the top-level container to calculate joint rates.

**Figure 9** Filters can be used to limit joint rates and end-effector error. In this example, the weighting matrix \( W \) is used to form a metric on the joint rates.

# 5 XML

The Actin toolkit provides extensive support for reading and writing control systems using XML. The toolkit contains XML objects that enable a developer to provide read and write support for any class. These XML objects can also be member variables in other XML objects. Through this approach, complex class structures can have XML read and write capability.
5.1 Organization

All XML objects build on a common base class. Through inheritance, XML read and write support is available to any subclass, and many rudimentary types are simply defined through template classes. These support real, integer, and Boolean values, as well as vectors, maps, sets, and pairs. Enumerations are also supported, where integers are represented through a fixed set of strings.

The rudimentary types are used to build more complex types through repetitive composition. The more complex XML data is organized by 1) compound types, 2) container types, and 3) variable vector types.

In Actin, compound class types have member variables that are statically defined, but whose values can be changed at run time (e.g., through a configuration file). Containers hold a single member that can be changed from one class to another at run time, and variable vector types hold multiple members that can be changed from one class to another at run time.

An illustration of the use of a container type is shown in the figure below. Its use parallels the use of the variable vector type.

![Diagram](image)

**Figure 10** Each container type provides an interface. There may be multiple classes that can meet this interface—illustrated as A, B, or C. These classes may be built into Actin or may be defined using dynamic link libraries that are written without recompilation of the control system. The specific object to be used is specified though XML.

5.2 Reading and Writing

XML data can be written as plain text or compressed text, through files, URLs (read only), TCP streams, or general streams. The XML files can also contain XLinks which redirect the XML reader to a URL for downloading XML fragments.

5.2.1 Files and URLs

Functions exist for reading and writing XML files. Depending on the file name, these functions can read and write in plain or compressed text. If the file name ends with “.gz” (the typical gzip
suffix), then the file is inflated upon reading and compressed upon writing. If the file name starts with “http://”, then the XML is read from a URL.

### 5.2.2 Streams

The XML reader and writer use the standard template library istream and ostream. A developer can create new streams based on istream and ostream for XML reading and writing. Special methods for reading and writing that can take any stream are provided for this purpose. Special TCP streams are available to provide capability to read and write to a TCP socket.

#### 5.2.3 XLink

XLink provides a hyperlink capability to XML using an attribute-based syntax. Actin supports the HTML-like absolute and relative unidirectional hyperlinks. The XML standard defines an XLink namespace for this task, which is used by Actin.

### 5.3 Schema 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 containing multiple namespaces can be combined to define a larger class of XML documents. Through the capabilities available through a schema, flexible XML vocabularies can be defined that mirror the complex data structures defined in the toolkit. Though 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.

![Diagram of XML validation process](image)

**Figure 11** The XML validation process. XML messages are validated with external tools using the schema. This can be combined with internal validation that Actin performs.

The Actin toolkit can auto-generate a schema when it writes an 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.
6 Toolkit Implementation

6.1.1 Organization

Actin is implemented using approximately 500 C++ classes, with about half dedicated to kinematic control (the others provide dynamic simulation, networking, visualization, sensor interfacing, machine vision, and other supporting functionality). These are implemented in toolkit form and are designed to be subclassed by developers with access only to the libraries and header files. Just a few steps allow the developer to create a new control system using the full power of C++, yet the developer does not have to reimplement any of the code that already exists in the toolkit.

6.1.2 Self-Testing Code

All concrete classes in Actin implement self-test methods. These are virtual functions that run multiple tests on the class. They provide unit testing as well as interface testing among the member objects. This test is repeatable and does not depend on the state of the instantiated object at the time the self-test is called. (The self-test method is virtual instead of static to allow developers to take advantage of polymorphism when debugging.)

7 Examples

Some examples of Actin being used to control complex robotic mechanisms are shown here. Actin includes software for 3D rendering of robotic mechanisms. This code was used to create the synthetic scenes shown. The flexible surface properties that are integral to the Actin software are used for rendering by defining color and specular properties.

7.1.1 PA-10

The Mitsubishi PA-10 is a popular commercially available kinematically redundant arm, with seven degrees of freedom. Actin has been used to control this arm with active joint-limit and collision avoidance. Actin can also be used to control more-complex systems that are constructed with the PA-10 as a component. The figures below show examples of Actin working with a PA-10.
Figure 12 Actin is shown controlling a Mitsubishi PA-10 at Johnson Space Center. Joint-limit control with positioning/orienting control is in effect. With seven controllable degrees of freedom, this system has two degrees of redundancy when positioning and orienting the end effector.
Figure 13 Actin is shown controlling a Mitsubishi PA-10 connected to an extra actuator that allows it to move along a rail. Collision avoidance control is in effect. With eight controllable degrees of freedom, this system has two degrees of redundancy when positioning and orienting the end effector.

7.1.2 Robonaut

NASA Johnson Space Center has created an anthropomorphic robot called Robonaut. The image below shows Actin controlling a 57 degree-of-freedom (three in the waist, two in the neck, seven in each arm, and 19 in each hand) model of Robonaut.
Figure 14 Actin is shown controlling an articulated model of NASA’s Robonaut. Coordinated control, which optimally uses all joints together, is being applied. With a fixed base, this model has 57 degrees of freedom. Not all degrees of freedom in the model are independent in the physical system, and these dependencies are accommodated using linear constraint end effectors.

7.1.3 Satellite Tug

The U.S. Navy is developing a robotic satellite tug that will have three seven-degree of freedom manipulators. The figure below shows Actin being used to control a prototype mock-up of the system.
Figure 15 Actin is shown controlling a model of a satellite tug. Each of the three arms has seven degrees of freedom, and the satellite base moves freely. Thus, this system has 27 degrees of freedom.

8 About Energid

Energid Technologies is an advanced technology company providing engineering products and services for robotic, machine-vision, training, virtual reality, and simulation applications. Whether you need a specialized training system, a sophisticated object recognition system, or complex robotic software for space exploration, Energid can provide the technologies and services needed.

If your project could use Energid’s expertise, please contact us at (888) 547-4100 or visit us at www.energid.com and we will be happy to discuss how we can work together to make your project a success.
Technical Bibliography


