OpenSim

User & Developer Guide

Robotics control framework and 3D dynamic simulation tool

Prepared by
Dr. David Jung

Version 0.4.1

Copyright 2003-2004 Oak Ridge National Laboratory
Licensed under the GNU Free Documentation License

Last modified August 11, 2004

Contents

I User Guide....4

1 Simulation....5

1.1 Application programs....5
1.1.1 viewenv....6
1.1.2 poscontrol....7

1.2 File Formats....8
1.2.1 An XML primer....8
1.2.2 Units....9
1.2.3 Manipulator files....9
1.2.4 Platform files....10
1.2.5 Robot files....11
1.2.6 Environment files....12
1.2.7 Tool files....14

2 Inverse Kinematics On Redundant systems (IKOR)....15

2.1 Application programs....15
2.1.1 ikortestrunner....15
2.1.2 ikortestviewer....16

2.2 Specifying off-line test cases....17
2.2.1 IKOR Test files....17

II Developer Guide....20

1 OpenSim Architecture....20

2 Tutorial....23

2.1 Introduction....23
2.1.1 Controllers, Controllables and ControlInterfaces....25
2.1.2 A ControlInterface for our robot's manipulator....27
2.1.3 Instantiating a joint position controller....29

2.2 The base module....30
2.2.1 Basic Types....30
2.2.2 Common types....31
2.2.3 Debugging aids....36
2.2.4 Platform abstractions....36
2.2.5 Serialization and Externalization....36
2.2.6 Utility classes....37

2.2.7 Unit tests..37
2.3 Creating a simple simulation..37
2.3.1 Setup..42
2.3.2 Describing a Robot..42
2.3.3 Viewer setup..44
2.3.4 The main loop..45
2.4 Creating a simulation from specification files..45
3 Guide..50
3.1 Inverse Kinematics..50
3.1.1 Using IKORController..50
3.1.2 The Full-Space Parameterization approach..52
3.2 Obstacle Avoidance..54
4 Reference..55
III Appendix..56
1 Build & installation..56
1.1 Tools..56
2 Coding convention notes..56
2.1 Coding Style..56
2.2 Files..58
3 GNU Free Documentation License..59
4 Document History..66

I User Guide

Introduction

OpenSim is primarily a programming library and set of tools for the robotics researcher to develop robot control code that can be either executed on robotic hardware or used to control real-time or off-line robot simulations.

The target user is the 'average' robotics researcher who needs to generate robotic control code to conduct experiments and test ideas – in so far as an 'average' robotics researcher can meaningfully be defined. Robotics research is conducted by practitioners from an unusually broad set of disciplines with diverse backgrounds–ranging from engineers (e.g. mechanical, control, electrical/electronic) through scientists from computer science, biology (biomechanics, neuroscience, cognitive science etc.) all the way to immunologists! However, the vast majority are currently from computer science, computer engineering and related engineering backgrounds. Consequently, use of the library as a programming framework and some of its tools assumes moderate competence in C++ object-oriented programming. However, the distribution includes some command-line and graphical tools that may be accessible by non-programmers.

The OpenSim system consists of thousands of classes and several programs which can be categorized roughly as follows:

- *Platform support* – classes that provide abstraction over low-level operating-system and other platform resources; such as files, timers etc.–to enhance portability of code over desktop and embedded systems.
- *Robot control framework*[1] – these classes provide an abstraction that is intended to be useful for creating robot control software, without imposing any particular paradigm or methodology on the designer.
- *Robot control components* – these are classes implemented within the framework that provide some common control components to use as peers with the developer's own code (if desired). For example, a PID controller.
- *Simulation system* – this is a library of components that can be optionally used to create 3D dynamic or static simulations of robots (fixed or mobile, single or multiple robot systems in various environments).
- *Tools* – a set of programs that utilize the other libraries that aid in development, visualization and debugging. Some of these could possibly be used by non-programmers. For example, the

[1]The term framework in this context refers to an object-oriented software framework–a standard, generic software foundation that provides a set of classes that support solving domain-specific problems in a vertical application domain; using inheritance and delegation to extend the framework. See "*How to make Software Reuse Work for You*", by Doug Schmidt, C++ Report, January 1999.

viewenv program reads a description of an environment containing robots, simulates them and provides 3D visualization of the run.

This part of the manual documents the tools and how to invoke and use them. For a tutorial introduction and reference to the programmatic framework and component API, refer to the the developer guide–part II of the manual.

1 Simulation

1.1 Application programs

The OpenSim distribution contains a number of command-line and graphical applications for running simulations.

- **viewenv** – (view environment) is a 3D graphical application that can read in environment specifications and simulate them statically or dynamically. It also allows querying robots for supported control interfaces and for some standard interfaces provides graphical controls (for example, to control joint angles, or run an inverse-kinematics controller to control end-effector positions).
- **poscontrol** – (position control) is a 3D graphical application that can read in environment specifications and simulate them dynamically. It has user controls for manipulating the positions of robot variables directly[2].

[2]The functionality of `poscontrol` has been mostly subsumed by the `viewenv` program. Hence, it may be removed in future.

1.1.1 viewenv

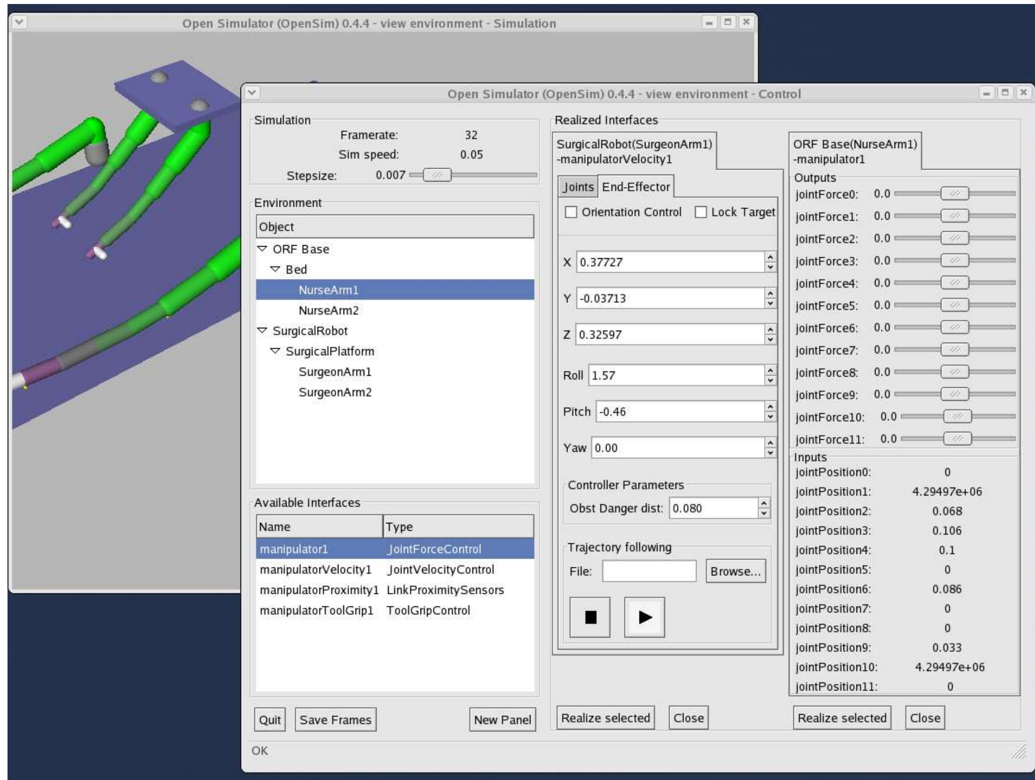

Figure 1 - labeled viewenv screenshot

The **viewenv** program is for visualizing and simulating virtual environments as specified in environment specification files (XML – see section 1.2.6 Environment files). It is invoked via the command-line with the following usage (where '[...]' denotes optional parameters and '|' denotes or):

`viewenv [-env` <*env_spec_file*>`] [-static] [-debug | -nodebug] [-debugoutput | -nodebugoutput] [-debuggfx | -nodebuggfx]`

The full path or relative path from the resource directory of the environment specification file is supplied with the `-env` option. Once the environment has been loaded it will be simulated dynamically (i.e. with Newtonian physics, approximate Coulomb friction and earth gravity) unless the optional `-static` switch is specified. Various amounts of debugging information can be output to the terminal by using the `-debugoutput` switch (or disabled with `-nodebugoutput`). Also, graphical aids for debugging may be displayed by using `-debuggfx`. The `-debug` switch enables both debug output and debug graphics. What is actually output and displayed depends on the implementation of the components being used. For example, the simulated manipulator implementation may display cylindrical proximity sensor range zones when simulating the manipulator link proximity sensors (if any are in use).

Here is an example invocation:

```
viewenv -env defaultenv.xml
```

This will create an environment containing a single fixed platform manipulator (called TitanII) and a stack of boxes (obstacles). Two windows are opened, a visualization window that shows a 3D view of the environment and a control window. At the top left of the control window in the 'simulation' section are shown various statistics about the running simulation, such as the rate at which frames are being rendered (in Hz), the simulation speed relative to real-time (e.g. 1.0 is real-time, 0.5 is half as fast as real-time), and the current simulation step-size (in seconds).

The section labeled 1 in the figure, is a representation of the items that are present in the environment. Each robot is listed along with other items, such as obstacles. The robot entries can be expanded by selecting the expand arrow to their left to show any component parts, such as the platform and any manipulators attached to it. The section labeled 2 lists the avaliable interfaces for controlling the selected robot. Although interfaces are actually associated with a robot, not with its component parts, to simplify the interface, viewenv filters the interfaces listed so that only manipulator specific interfaces are listed when a manipulator object is currently selected in the environment and only platform or robot specific interfaces are listed with the robot is selected.

To actually use a particular interface to control the running simulation, it must be realized. The section labeled 3 in the figure is where the graphical controls associated with realized interfaces are displayed. By default only a single panel is available, but more can be added to the right of the window by clicking the 'New Panel' button. To realize an interface, select it from the 'available interfaces' list and press the 'Realize selected' button at the bottom of the panel in which you want the controls displayed.

Multiple interfaces may be realized in a single panel, in which case they can be selected via the labeled tabs at the panel's top. They are labeled *<robot_name>*(*<component_name>*)-*<interface_name>*. For example, *SurgicalRobot(SurgeonArm1)-manipulatorVelocity1*. What is actually displayed in the control panel depends on the type of the interface that is realized. For some specific interface types special controls are provided. If the interface type is unknown (for example, because you've implemented a `ControlInterface` in your own robot code), a generic control panel will be displayed that is constructed by querying the interface for the number and names of any inputs and outputs it provides. For example, in the figure the `manipulatorVelocity1` interface was realized, which has the type `JointVelocityControl`. This interface type has a special control panel which has two tabs. One allows control of the joint positions and one (the one shown selected) provides control of the end-effector position and orientation. This is achieved by the control panel through instantiating controllers–a PID controller and an inverse kinematics controller, respectively.

1.1.2 poscontrol

This application is currently undocumented and may be deprecated in future.

1.2 File Formats

Most input and output files are XML documents. If you are familiar with the XML format, you may skip the next section. In the sections to follow which detail each particular file format, the description will document the top-level element associated with that type of XML file. Often, these elements can be embedded directly into other files where needed; or referred to indirectly via a `link` attribute. Each element description will detail the element's attributes and their values, the possible element content and which elements can be nested within it.

1.2.1 An XML primer

The eXtensible Markup Language (XML) is an extensible format for specifying information in a structured and portable way. XML files are typically just text files. Specifically they are Unicode text files, commonly in the Unicode UTF-8 encoding, of which part of ASCII is a subset. Consequently, typical XML files in English look like straight ASCII text files. Here is an example XML file.

```
<?xml version="1.0" encoding="UTF-8" standalone="no" ?>
<ikortest name="nhtest">
  <environment link="nhenv.xml"/>
  <testrobot>NonHolonomicRobot,NHTestArm</testrobot>

  <test name="test1">
    <initialconfig> 0  -1  0 40 40 40 40 40 40 40 40</initialconfig>
    <jointweights>  1   1  1  1  1  1  1  1  1  1  1</jointweights>
    <path frame="eebase" timeinterval="0:1:0.005"
          link="xback_path.xml"/>
    <solution solnmethod="fullspace" optmethod="lagrangian"
              criteria="leastnorm" orientationcontrol="false"/>
    <constraints>
      <jointlimit/>
    </constraints>
  </test>

</ikortest>
```

Example 1.2.1 - Simple XML file

In this case, it is an IKOR test specification (detailed in section 2.2). All XML files start with `<?xml ... ?>`, however, we'll not concern ourselves with this here.

Notice that the file is comprised of a nested set of named *elements*, introduced via a start tag (e.g. `<constraints>`) and terminated with an end tag (e.g. `</constraints>`). The names of the elements, their meaning and which elements may be nested within which others, is completely application defined. As a shorthand, elements that would be empty can be written like the `<jointlimit/>` element above–which is short for `<jointlimit></jointlimit>`.

Elements can also have attributes. These are specified inside the start tag, such as the `test` element above, which has a `name` attribute with value `test1`. The attribute value must be quoted. The order of attributes is not important.

Although in general white space (spaces, tabs, newlines etc.) can be important in XML, it is mostly ignored except within some special elements. In the example above, the space at the beginning of the lines is ignored; as is the space between elements. However, the spaces separating the numeric values with the `jointweights` element are significant (for example). It is also possible to place comments, or comment out elements using the notation: `<!-- this is comment text -->`.

> **Caution**
>
> Most of the OpenSim classes that externalize XML files are written to ignore any elements they don't handle – so be careful of misspelling element names as they may be sliently ignored.

1.2.2 Units

Many of the elements for defining inputs to OpenSim make used of numeric values. These are usually interpreted as floating point values. These may appear alone or as part of larger components, such as vectors/points or other kinds of elements. By default, if the value represents a distance, then meters are assumed. If the value represents an angle, then radians are assumed. However, values may be scaled to different units by suffixing them immediately with either '`in`' for inches, or '`deg`' for degrees. Note that this is performed in the externalizer, not the class responsible for the element being defined. Hence, is is possible to erroneously place '`in`' after a value that will be interpreted as an angle and the externalization code will happily scale the value by 0.0254 before passing it to the class in question (the scaling factor from inches to meters) – so be careful.

1.2.3 Manipulator files

```
<?xml version="1.0" encoding="UTF-8" standalone="no" ?>
<manipulator name="BuggyManip" type="serial">
    <kinematicchain type="DH">
     <!--D-H parameters describing the serial Kinematic Chain-->
     <!--       type ,   alpha ,         a ,         d ,   theta , minlimit , maxlimit  -->
     <link>revolute ,  90     ,   0.6      ,   0        ,   0     , -160.0000, 160.0000 </link>
     <link>revolute ,   0     ,   0.6      ,   0        ,   0     ,  -10.0000, 160.0000 </link>
     <link>revolute ,  90     ,   0.7      ,   0        ,   0     ,  -90.0000, 170.0000 </link>
     <link>revolute , -90     ,   0.15     ,   0        ,   0     , -120.0000, 120.0000 </link>
     <link>revolute ,   0     ,   0.15     ,   0        ,   0     , -120.0000, 120.0000 </link>
     <link>revolute ,   0     ,   0.1      ,   0        ,   0     , -120.0000, 120.0000 </link>
    </kinematicchain>
</manipulator>
```

Example 1.2.2 - A manipulator description

- `manipulator` – This element describes a complete robot manipulator. It is the top level element externalized by the `ManipulatorDescription` class.
  - *Attributes*
    - `name` – A name for the manipulator (default `manipulator`).
    - `type` – Either `serial` or `parallel` (default `serial`). The value `parallel` is currently unused.
  - *Elements*

- kinematicchain – This element specifies the kinematic structure of the manipulator. It is the externalization of the KinematicChain class.
  - *Attributes*
    - type – Either DH or mixed (default DH). This determines which link types are allowed. If DH then only types revolute and prismatic are allowed. Otherwise the additional types translating and transform are allowed.
  - *Elements*
    - link – Describes the kinematics of a single link via a sequence of comma separated values. The first value is a string that determines the link type and format of the values that follow. If the type is revolute or prismatic then the link is a DH type link–Denavit-Hartenberg. The following four values then correspond to the DH parameters α (alpha), a, d and (theta). DH type links have a single degree-of-freedom. The final two values correspond to the joint limits–specified in degrees[3] for the revolute joint. If omitted the joint is unlimited. e.g.

      ```
      <link>revolute, 90, 0.6, 0, 0, -160.0000, 160.0000</link>
      ```

      If the link type is translating then the three values following the type represent a direction vector along which the 1-D.O.F. joint translates. The last two values are also joint limits. A transform link is static (0-D.O.F).

1.2.4 Platform files

```
<platform mobile="true" holonomic="false" name="buggy">
  <dimensions>(  4.5700,   2.1600,   0.7000)</dimensions>
  <originoffset>( 1.2, 0, 0.35)</originoffset>
</platform>
```

Example 1.2.3 - A platform description

- platform – This element describes a robot platform. It is the top level element externalized by the PlatformDescription class.
  - *Attributes*
    - name – A name for the platform (default platform).
    - mobile – Either true or false (default false). Indicates if this is a mobile platform (e.g. has wheels or legs), or a stationary platform.
    - Holonomic – Either true or false (default false). This is only applicable for mobile platforms. It specifies whether the platform is capable of holonomic motion or not.
  - *Elements*
    - dimensions – If present, indicates that the geometric shape of the platform is a simple box with the given dimensions. The origin of the box is at its centroid. The dimensions are specified as a 3 element vector (x,y,z). e.g. (4.5700, 2.1600, 0.7000).
    - originoffset – Specifies an offset between the origin of the platform geometry and the origin of the robot/platform. Defaults to 0–i.e. (0, 0, 0).
    - params – Applies only for non-holonomic mobile platforms. It specifies the distance between the platform origin and the back drive axle (L) and the distance between the back drive axle and the front drive wheel(s) (W). They are specified as attributes, like so: <params L="2.9" W="3.7"/>

---

[3] As degrees are assumed by default, don't suffix with 'deg' or the value will be scaled by the degree-to-radian scaling factor twice.

1.2.5 Robot files

```
<?xml version="1.0" encoding="UTF-8" standalone="no" ?>
<robot name="Buggy">
    <!--description of a robot (with a [mobile] platform and one or more
manipulators-->
    <platform mobile="true" holonomic="false" name="buggy">
      <dimensions>(  4.5700,   2.1600,   0.7000)</dimensions>
      <originoffset>( 1.2, 0, 0.35)</originoffset>
    </platform>
    <manipulator name="BuggyManip" link="buggymanip.xml">
      <offset>(  0.2000,   0.2000,   0.0000)</offset>
    </manipulator>
</robot>
```

Example 1.2.4 - A robot description

- `robot` – This element describes a robot in terms of its components – a platform and optionally some number of attached manipulators. It is the top-level element externalized by the `RobotDescription` class. Derived classes, such as `SimulatedRobotDescription` may add further specific information (for example, simulation specific parameters).
  - *Attributes*
    - `name` – A name for the robot (default `robot`).
  - *Elements*
    - `platform` – this element describes the robot platform. It must adhere to the format detailed in section 1.2.4 above. There must be exactly one platform for a robot.
    - `manipulator` – one or more of these elements describe each of the manipulators attached to the robot. The content must adhere to the format described in section 1.2.3 above. In addition the following attributes and elements may be specified.
      - *Attributes*
        - `link` – a file which contains a manipulator description (i.e. its top-level element is manipulator). This allows the description of the manipulator to be stored in a separate file. In this case the usual content of the element may be omitted. The file name may be absolute or relative to a resource directory.
        - `name` – if a `name` attribute is specified in addition to the `link` attribute, this name will override any name specified in an attribute of the `manipulator` element in the file referenced by the `link` attribute.
      - *Elements*
        - `offset` – a 3 element vector that specifies an offset from the robot/platform coordinate origin to the mount point of the manipulator.

1.2.6 Environment files

```
<?xml version="1.0" encoding="UTF-8" standalone="no" ?>
<environment type="basic">

    <!--robots   (description, position, and orientation (Quat) )-->
    <robot name="TitanII" link="titan2.xml">
      <position>(  0.0000,   0.0000,   0.100001)</position>
      <orientation>(  0.0000,  0.0000,  0.0000,  1.0000 )</orientation>
    </robot>


    <!--tools -->
    <tool name="ExtensionTool" link="extensiontool.xml">
      <position>(   0.0000,   -1.0000,   0.050001)</position>
      <orientation>(  0.0000,  0.0000,  0.0000,  1.0000 )</orientation>
    </tool>


    <!--obstacles (position, orientation (Quat) and description)-->
    <obstacle name="obstacle0" type="box">
      <position>(  0.5000,   1.0000,   0.250001)</position>
      <orientation>(  0.0000,  0.0000,  0.0872,  0.9962 )</orientation>
      <dimensions>(  0.4000,   0.4000,   0.4000)</dimensions>
      <material name="plastic" type="simple">
        <density>1</density>
        <basecolor>(  0.7000,  0.4000,  0.8000 )</basecolor>
      </material>
    </obstacle>

    <obstacle name="obstacle1" type="box">
      <position>(  0.5000,   1.0000,   0.6210)</position>
      <orientation>(  0.0000,  0.0000,  0.1305,  0.9914 )</orientation>
      <dimensions>(  0.3400,   0.3400,   0.3400)</dimensions>
      <material name="plastic" type="simple">
        <density>1</density>
        <basecolor>(  0.7000,  0.4000,  0.8000 )</basecolor>
      </material>
    </obstacle>

</environment>
```

Example 1.2.5 - An environment description

- `environment` – This element describes a virtual environment for the purpose of simulation. It specifies robots, tools, obstacles and other parameters that define the environment. It is the top-level element externalized by the `SimulatedBasicEnvironment` class.
  - *Attributes*
    - `type` – specified the type of environment specification contained within the element. This attributes exists to allow different styles of environment descriptions in the future. Currently the only supported type is `basic`.
  - *Elements*
    - `robot` – any number of robots may be contained in an environment. The content of this element should adhere to the format described in section 1.2.5 above, with the following optional additional attributes and elements.
      - *Attributes*
        - `link` – the link attribute is used to specify a robot file to use as an alternative description of the robot. In this case the usual content of this element may be omitted. The file name may have an

absolute path or be relative to a resource directory. The referenced file must be a robot file (i.e. a file whose top-level element is `robot`).

        - `name` – the name of the robot. If this attribute is specified with the `link` attribute, then the name will override any name specified as an attribute to the `robot` element contained within the file referenced by the `link` attribute.
    - *Elements*
        - `position` – this element specifies the initial Cartesian position of the robot with respect to the global environment coordinate origin. The vertical axis is the Z-axis whose 0 point is on the ground plane with the +ve half-axis pointing up. The position is specified as a 3 element vector.
        - `orientation` – this element specifies the initial orientation of the robot with respect to the global environment coordinate frame. It is specified as either a 3 or 4 element vector. If it has 3 elements it is interpreted as roll, pitch and yaw angles (EulerRPY) in radians (by default); if 4 elements it is interpreted as a quaternion (x,y,z,w). For detail, see the description of the `Orient` class from the `base` module.
- `tool` – there may be any number of tools present in an environment. The content of this element should adhere to the format described in section 1.2.7, with the following optional additional attributes and elements.
    - *Attributes*
        - `link` – the `link` attribute is used to reference an alternative description of the tool, rather than including the description in-line. It is used analogously to the `link` attribute of the `robot` element.
    - *Elements*
        - `position` – this element specifies the initial position of the tool in the environment. It is analogous to the `position` sub-element of the `robot` element described above.
        - `orientation` – this element specifies the initial orientation of the tool in the environment. It is analogous to the `orientation` sub-element to the `robot` element described above.
- `obstacle` – there can be any number of obstacles in the environment. Obstacles are passive objects that cannot be controlled – although they may move in response to applied forces, such as when coming into contact with a robot or other moving objects.
    - *Attributes*
        - `name` – a name for the obstacle.
        - `type` – the type of obstacle. Currently this is limited to either `box` or `sphere`.
    - *Elements*
        - `position` – the position in global Cartesian coordinates of the obstacle in the environment. This is a 3 element vector (default units are meters).
        - `orientation` – the orientation with respect to the global frame of the environment. This can be a 3 or 4 element vector (see the description of the analogous element within the `robot` element above).
        - `dimensions` – only applicable for type `box`. This element gives the dimensions of the box along the x, y and z axes respectively. The origin is at the center of the box (centered for each axis).
        - `radius` – only applicable for type `sphere`. This element contains a single number for the radius of the sphere. The origin is at the center of the sphere.
        - `material` – this element defines the material properties of the obstacle body. It is the top-level element externalized by the `physics::Material` class. Some of the enclosed elements are only used for visualization and may be omitted if visualization of the environment is not required (for example the `basecolor` element which specifies a surface appearance property).
            - *Attributes*
                - `name` – a string name for the material.

    - `type` – a type for the material. Currently only `simple`.
- *Elements*
    - `density` – this element contains a single numerical value that defined the density of the material (Kg/m^3). For example, the density of water is 1.0. The default is 1.0 if omitted.
    - `basecolor` – this element is a 3 element vector that specifies the base surface color of the material. The elements are interpreted as Red, Green and Blue components respectively and must be in the range [0...1]. This element may be omitted if visualization is not required (in which case it will default to green).
    - `surfaceappearance` – this element specifies the appearance of the surface of the material.
        - *Attributes*
            - `type` – the appearance type. Currently only `image` (also the default).
            - `image` – an image source to use for texturing the surface of the material. This can be a file with an absolute path or a path relative to a resource directory (e.g. "`image/dirt.jpg`").

1.2.7 Tool files

Not yet documented. Tools are essentially like manipulators, but without an offset or transform to the first joint.

2 Inverse Kinematics On Redundant systems (IKOR)

The IKOR (Inverse Kinematics Of Redundant-manipulators) library module was designed to tackle two shortcomings of existing techniques and systems for redundant manipulator control. Firstly, IKOR embodies a new technique that decouples the computation of all possible motions from the criteria used to resolve the redundancy by narrowing those possibilities down to a single motion to be executed. This allows the criteria to be dynamically selected appropriately for the current task. Also, and perhaps more importantly, the redundancy resolution optimization allows motion constraints to be specified in a standard way such that various different constraints can be dynamically applied.

2.1 Application programs

The OpenSim distribution contains some command-line applications for exercising the IKOR code.

- **ikortestrunner** – is used to run off-line tests of the inverse kinematics code, using either static or dynamic simulations (without a graphical display). All the parameters of the tests are specified via a test specification file – which is detailed in section 2.2.
- **ikortestviewer**[4] – will read in the test specification and results files output by **ikortestrunner** and display the results graphically.

2.1.1 ikortestrunner

The **ikortestrunner** has the following usage:

`ikortestrunner` <*test_spec_filename*>

The test specification file includes all the information necessary to instantiate and run an off-line (i.e. not real-time) test of various IKOR components. Some of the required information may be supplied directly within the test specification file or may be specified via references to other files. The format of the test specification files is detailed in section 2.2.

The file must provide a specification of an environment, which must in turn specify at least one robot, which has at least a single manipulator. The robot may have a fixed or mobile platform. If mobile, the platform can be either holonomic or non-holonomic. Currently, only the first robot in the environment and its first manipulator are used for running the tests. In addition, the environment may contain obstacles. Any information about the environment and robots not required for a non-graphical, non-real-time simulation will be ignored (for example, aspects of the appearance of a robot arm not related to its geometry).

---

[4]Note that at the time of writing, the **ikortestviewer** program has not been ported from the previously used GUI framework (GLOW & osgGLOW) to the currently used Gnome/GTK/gtkmm framework (as used by **viewenv**).

Each IKOR test consists of one or more sub-tests. Each sub-test consists of a separate trajectory segment that the manipulator end-effector is required to follow. Each test also specifies information about which solutions algorithms will be used for the inverse-kinematics computations, possibly limits on some free variables and possibly various constraints on the system. An example of the output from a test run is shown below.

```
Loading test specification from file '/home/jungd/unix/dev/OpenSim/resources/data/test/nhtest.xml'.
Testing robot:NonHolonomicRobot with manipulator:NHTestArm
Executing test: test1
robot::sim::IKORTester::executeTest -- initial q=[0,-
1,0,0.698132,0.698132,0.698132,0.698132,0.698132,0.698132,0.698132,0.698132]  x=
[0.213408,0.0126854,2.80563]
robot::sim::IKORTester::executeTest --   final q=[-12.5441,-
1.93549,3.1362,0.860632,0.458453,0.196663,0.170966,-0.035325,-0.0302477,0.000811575,-0.00114126]
x=[-14.7865,-4.98724,2.80563] (199 steps)
robot::sim::IKORTest::Test::saveResult -- Saving joint trajectory file
'/home/jungd/unix/dev/OpenSim/opensim/test1_jtraj.xml'.
robot::sim::IKORTest::saveResults -- Saving complete test specification and results to file
'/home/jungd/unix/dev/OpenSim/resources/data/test/nhtest_results.xml'.
Exiting.
```

Listing 2.1.1 - Output of ikortestrunner

Note that the output file, in this case `nhtest_results.xml`, is typically written into the same directory as the input test specification file. The name is derived from the test name–which is part of the specification.

2.1.2 ikortestviewer

The ikortestviewer is for visualization of the output files generated by **ikortestrunner** in a line-diagram format. The figure below shows an example of the displayed visualization. A graphical control window is also opened that allows control over which trajectory segments and what interval of each segment is to be displayed. It has the following usage:

`ikortestviewer` <*test_spec_filename*>

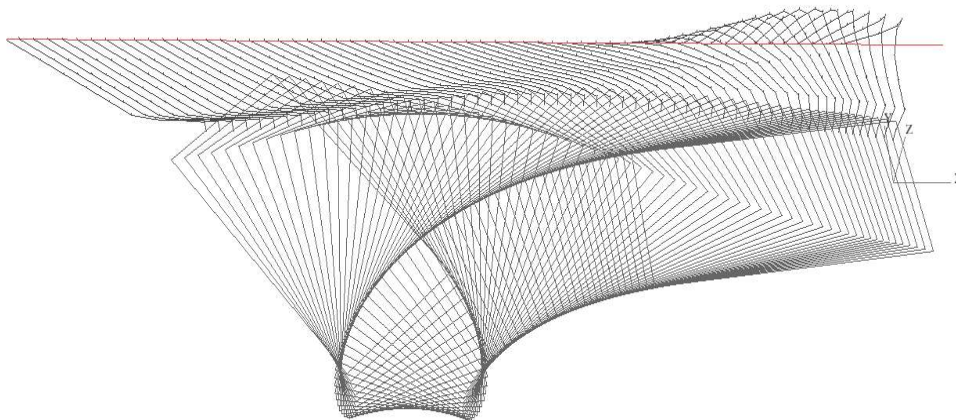

Figure 2 - Visualization of `ikortestrunner` output via `ikortestviewer` showing the configuration trajectory of a manipulator mounted on a non-holonomic mobile platform

2.2 Specifying off-line test cases

IKOR Test specification files are written as XML documents[5]. If you are not familiar with the XML format you should read section 1.2.1 - which is a short XML primer.

2.2.1 IKOR Test files

```
<?xml version="1.0" encoding="UTF-8" standalone="no" ?>
<ikortest name="nhtest">
  <environment link="nhenv.xml"/>
  <testrobot>NonHolonomicRobot,NHTestArm</testrobot>

  <test name="test1">
    <initialconfig> 0  -1  0 40 40 40 40 40 40 40 40</initialconfig>
    <jointweights>  1   1  1  1  1  1  1  1  1  1  1</jointweights>
    <path frame="eebase" timeinterval="0:1:0.005" link="xback_path.xml"/>
    <solution solnmethod="fullspace" optmethod="lagrangian"
              criteria="leastnorm" orientationcontrol="false"/>
    <constraints>
      <jointlimit/>
    </constraints>
  </test>

  <display>
    <obstacles/>
    <axes/>
    <camera alpha="-250" theta="13.5" d="9" target="(-0.18, 0.1, 0.7)"/>
  </display>

</ikortest>
```

Example 2.2.1 - An IKORTest test specification

- ikortest – This element describes an off-line IKOR test case. It is the top level element externalized by the robot::sim::IKORTest class.
  - *Attributes*
    - name – A name for the tests (default ikortest).
  - *Elements*
    - environment – This element describes the simulated environment in which the tests are to be conducted. It specifies the robot and its platform and manipulators; obstacles and tools etc. Its format is detailed in section 1.2.6. Note that the environment element supports the link attribute for specifying an external file that describes the environment.
    - testrobot – This indicates the name or index of the robot and which one of its (possibly many) manipulators are to be tested. If omitted the default is the first robot specified in the environment and its first manipulator. The names or index numbers are given separated by a comma. If names are used, they must match exactly the names given in the robot or manipulator specifications.
      e.g.
      <testrobot>NonHolonomicRobot,2</testrobot>
      This indicates the 3rd manipulator of the robot named NonHolonomicRobot in the environment description. Note that the indices are 0-based.

---

[5]Currently, there is no explicit Document Type Definition (DTD) defined for the input format. The elements recognized and their behavior are determined by the input externalization of the robot::sim::IKORTest class and are described here. A DTD may be defined in future.

- `test` – A single test specification. There can be any number of these within the `ikortest` element. Each test will use the final state of environment (robot and manipulator) from the previous test (if any), unless the `initialconfig` element is used to override it.
  - *Attributes*
    - `name` – A name for the test (default `test`).
  - *Elements*
    - `initialconfig` – A space separated sequence of initial values for the robot variables corresponding to each degree-of-freedom. For example, if the robot is not mobile, the values will represent the joint positions of the manipulator. The interpretation of each value depends on the type of joint it corresponds to. For example, for a revolue joint the value will represent an angle (in degrees). If omitted, the initial configuration will be the last configuration of the previous test, or 0 for the first test.
    - `jointweights` – A space separated vector of weight values for the robot variables corresponding to each degree-of-freedom. These will become the diagonal entries of the diagonal weight matrix B. This is often used to adjust for different units – for example to weight the platform position variables differently to manipulator joint angles. If omitted the default is 1 (i.e. B will be the identity).
    - `attachedtool` – If present, this element specifies the name of a tool that will be considered to be attached to the end-effector for this test. The name string must match exactly the name of a tool described in the environment description. The variables corresponding to any degrees-of-freedom of an articulated tool are appended to the end of the configuration vector.
    - `path|trajectory` – Either a path or a trajectory can be specified here. Recall that a path comprises position and orientation components and a trajectory additionally has a time component. If the test requires a path and a trajectory is specified the time component is ignored. Conversely, if a trajectory is required but a path is specified, the time components are generated such that the path spans 1 second, unless overridden via the `timeinterval` attribute. The `link` element accepts the following attributes.
      - *Attributes*
        - `frame` – Optional. The coordinate frame in which the path is specified. One of `ee`, `eebase`, `base`, `mount`, `platform` or `world`. This overrides any specification in the path or trajectory file if the `link` attribute is used.
        - `timeinterval` – Optional. Specifies the time interval over which the path is to be executed (if required by this solution method) – or overrides it in the case where a trajectory is specified. It is a string of one of the forms *`duration`*, or *`start-time`*`:`*`end-time`* , or *`start-time`*`:`*`end-time`*`:`*`sample-period`* . Cannot use used with `maxdx`.
        - `maxdx` – Maximum value of |dx| for any step of the trajectory computation (optional). Either the string `default` or a real value. Cannot use used with `timeinterval`.
    - `solution` – This element (a sub-element of `test`) specifies which solution method is to be used for the inverse kinematics computation. It is an empty element with the options determined via the attributes described below.
      - *Attributes*
        - `solnmathod` – Solution method (optional). Either `pseudoinv` (the default) or `fullspace`.
        - `optmethod` – Optimization method (optional). Either `pseudoinv`, `lagrangian`, `bangbang` or `simplex`. If omitted, defaults to `pseudoinv` if the solution method is `pseudoinv` or `lagrangian` if it is `fullspace`.
        - `criteria` – Optimization criteria (optional). Either `leastnorm` (the default) or `leastflow` (incompatible with `pseudoinv` solution method, and currently unsupported).

        - `orientationcontrol` – Either `true` or `false`. If omitted, defaults to `true`. Determines if the inverse kinematics solution will include the orientation components of the end-effector, or only the position components.
    - `constraints` – This element specifies the constraint types that will be handled by the optimization method (if appropriate). If omitted, the default is unconstrained. Each constraint type is specified via the child elements detailed below. Note that most are empty elements.
        - *Elements*
            - `jointlimit` – Activate Joint limit constraints for any joint that specified limits in the platform, manipulator or tool (if attached) descriptions.
            - `obstacle` – Constrains links of the manipulator to maintain a distance from obstacles (and other object/manipulators/robots etc.) that is larger than a specified 'danger' distance. Note that the manipulator must have proximity sensors for this to have any effect (as specified in the manipulator description).
            - `acceleration` – reserved for future use - not currently implemented.
            - `eeimpact` – reserved for future use - not currently implemented.
    - `display` – An optional empty element used to specify some test specific viewing options for the **ikortestviewer** program. Is it typically present as a result of using the save feature of **ikortestviewer**. The `startIndex` attribute specifies the time step index at which the result trajectory steps will begin to be displayed and the `endIndex` the last time step index displayed.
    - `result` – This element contains the results of running the test with the **ikortestrunner** program. Each line contains a set of comma separated vectors, each with space separated components. The vectors are `time`, `q`, `x`, `dx`, and `dq`. The `completed` attribute indicates if the test was complete, or if it was aborted before the end-effector traveled the entire target trajectory (for example, due to a solution error).
- `display` – This (sub-element of `ikortest`) is an optional element used to specify the viewing options (such as camera parameters, what is shown etc.) for the **ikortestviewer** program. Is it typically present as a result of using the save feature of **ikortestviewer**.
    - *Elements*
        - `obstacles` – An empty element, that if present indicates that any obstacles specified in the environment will be shown.
        - `axes` – An empty element, that if present indicates that the world frame axes will be shown (with X,Y & Z labels)
        - `eepath` – An empty element, that if present indicates that the end-effector target trajectories will be shown.
        - `stepmod` – An integer number `n`, that specifies that only every n^{th} time step will be shown (rather than all the steps as is the default).
        - `platform` – An empty element, that if present indicates that the platform outline will be shown (currently the outline of the bounding square at the origin in Z).
        - `camera` – An empty element, that if present, specifies the 3D camera viewing parameters via its attributes.

II Developer Guide

Introduction

OpenSim is primarily a programming library and set of tools for the robotics researcher to develop robot control code that can be either executed on robotic hardware or used to control real-time or off-line robot simulations.

This guide provides an architectural overview, a tutorial style introduction and a reference to the framework API for robotic control code developers.

1 OpenSim Architecture

The OpenSim system is divided into several *modules*. When built for a desktop environment, each module is linked as a shared library[6]. Each module corresponds to a C++ namespace (and some modules have sub-modules as nested namespaces). The source directory hierarchy corresponds to the namespaces. The main modules are as follows (also refer to Figure 3):

- `base` – contains classes that provide an abstraction over operating-system facilities, such as files, timers etc., and also many utility classes for I/O, reading and writing structured formatted files (e.g. XML files), events handers, smart-pointers, arrays, vectors, matrices, miscellaneous math routines (e.g. SVD), and many other (non-robotics) related facilities. All other modules depend on the base module – but it only depends itself directly on operating-system APIs.
- `gfx` – classes to support graphics used by the simulation module. Currently, provides some basic geometric objects, like lines, segments and triangles etc. Most of the visualization is currently handled via the Open Scene Graph (OSG) library[7].
- `physics` – the physics module provides a set of classes for simulating sets of arbitrarily shaped rigid bodies acting under the laws of Newtonian mechanics and interacting via various constraints–such as friction, joints (hinge, universal, slider etc.). An abstract interface is defined to allow alternative implementations of physics and collision code to be plugged-in. Currently, the only implementation utilized is the Open Dynamics Engine (ODE)[8] for physics solving. The implementation can optionally supply and instantiate OSG objects for visualization (if OpenSim is built with OSG).
- `robot` – this is the namespace in which all robotics related interfaces and components are implemented. It includes interfaces for low-level interaction with robot hardware (either real or

---

[6]The terms *library* and *module* are used loosely and interchangeably throughout the remainder of the documentation and the source code comments. So the use of 'library' doesn't necessarily imply a shared dynamically loaded library – although that would typically be the case for a desktop build of OpenSim.

[7]A modern Open Source scene-graph library written in C++. Refer to http://www.openscenegraph.org.

[8]ODE is an open source implementation of a Newtonian solver using both an LCP and an iterative solution method. Refer to http://opende.sourceforge.net.

simulated), interfaces for controllers, convenient classes for describing robots, platforms and manipulators at various levels of detail (for example, to describe geometry to the simulation system or just to record kinematic configurations of manipulators). All the description classes provide externalization to read and write standard extensible XML specifications of robot platforms and manipulators. It only depends on the `base` module.

- `robot::control` – this module contains some useful control components for developers to utilize or leverage when writing their own controller code. For example, a `ManipulatorPIDPositionController` class.
- `robot::control::kinematics` – this module contains classes related to kinematics and inverse kinematics. For example, it includes an implementation of a solver for IK of redundant manipulators using a Lagrangian optimizer and a Jacobian generator class (parts of IKOR).
- `robot::sim` – this module implements simulation of multi-robot environments. It includes interfaces for describing environments and uses the `physics` and `gfx` modules to realize dynamic simulations. This is achieved by representing robots & manipulators as collections of constrained rigid bodies for the physics solver. It also contains classes to support specifying off-line test-cases and their simulation results for repeated testing at a later time.
- `apps` – this is the module that contains all the tool program code that utilizes the other modules. For example, the **viewenv** program which can read an environment specification (including robots, manipulators, obstacles etc.) and simulate it with a 3D visualization.

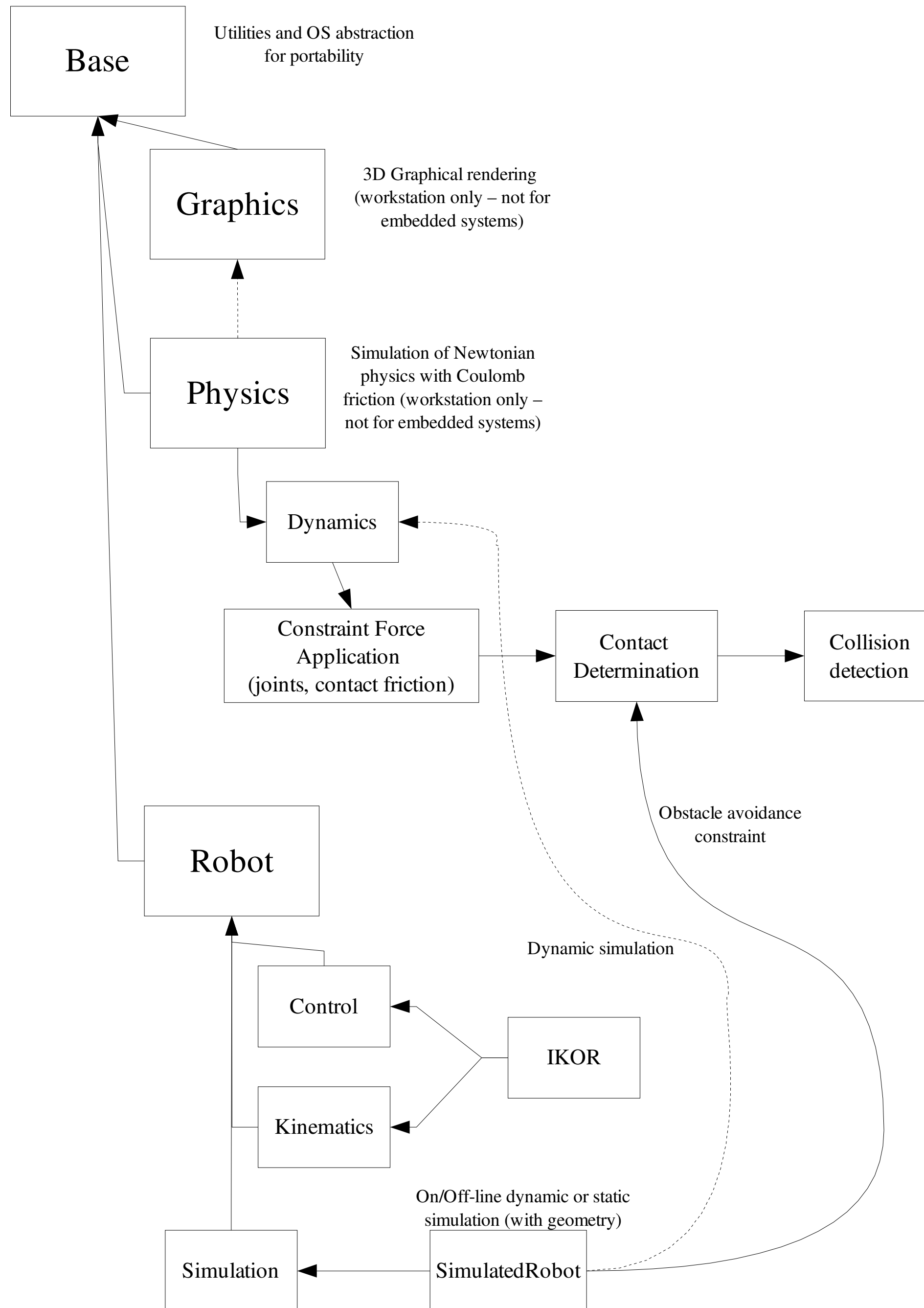

Figure 3 - overview of module dependencies

In order to become proficient at writing code within the OpenSim framework, first an understanding of the coding conventions and many of the classes in the `base` module is necessary. However, before going into great details about the base module API, a tutorial introduction will be beneficial.

2 Tutorial

This section provides an introduction to OpenSim in a tutorial style. First an introduction to some important classes is given by way of discussion and code snippets. As all the OpenSim modules are supported by the classes and types in the `base` module, at least an overview familiarity with many of the classes provided in the base module is necessary before any sense can be made of even simple robot control and simulation programs. Hence, the second section provides a base module overview. The sections following that are a walk-through style tutorial for some robot programs and visualizations for which complete source-code is listed and discussed. The source code is supplied in the OpenSim distribution and may be compiled and run.

2.1 Introduction

This section provides an introduction to some of the classes in the robot module. It is not intended that all the details of the supporting classes be understood at this point, but rather the aim is to give the flavor of what developing under OpenSim is like and what can be achieved. The code presented here is sometimes simplified and consists of code snippets; and hence cannot always be compiled as a complete working application.

Consider the following simple hypothetical situation. Suppose that you have a robot that has its own on-board computer with an operating system and C++ compiler on which you can compile the OpenSim `base` and `robot` modules. The robot has a 7 joint manipulator and a vendor supplied software library that allows the joint velocities to be commanded. You would like to use the OpenSim framework to ultimately developer high-level controllers, but for now you just want to use an existing PID controller from the OpenSim `robot::control` module to be able to have simple position control over the joints.

The first things you'll need to do is to wrap the vendor supplied functions for commanding the joint velocities with a `ControlInterface` derived class. This will allow the `robot::control::ManipulatorPIDPositionController` class to interface with the manipulator.

Before we get to that step, we need a basic OpenSim styled 'main' program:

```
// include headers for modules we depend on
#include <base/base>
#include <robot/control/control>

// now the specific classes we need
#incoude <base/Time>
#include <base/Application>
#incuude <robot/ControlInterface>
#include <robot/Controllable>
#include <robot/control/ManipulatorPIDPositionController>

// bring some class names into scope to save typing
```

```
using base::Application;   // saves us from having to type base::Application
using robot::control::ManipulatorPIDPositionController;

int main(int argc, char *argv[])
{
  // every OpenSim program needs to declare/instantiate a single
  //  Application object.
  Application app("/resources","/cache");

  // ... main control loop goes here ...

  return 0;
}
```

First, headers for each module on which we will depend are included – in this case `base` and `robot::control`. Although each module will also include the header for modules upon which it depends itself, we include `base` here for readability – even though `robot::control` would include it for us anyway. In OpenSim each class is implementation in a separate file – one file for the header[9] and one `.cpp` file for the implementation. Hence, we also include the headers for each class we need. These will also include other class headers on which they depend, but it will help readability if you specifically include the classes you use.

Notice that the `Application` constructor takes two arguments. These are paths to locations for loading resource files and saving cache files (more on these later). On a desktop system, these will correspond to directories; on an embedded system they could be something else (such as memory buffers, network resources, in fact anything that implements the virtual file-system interfaces defined in `base`).

We could have used the C++ statements:

```
using base;
using robot;
using robot::control;
```

This would have pulled in all the classes defined in the respective namespaces. That saves a lot of typing, and may be preferable for quick prototyping programs. For best long term control of name collisions, it is best to individually list the classes we want to use each with its own `using` statement (or to specifically qualify the name where used).

Before we can write the main control loop, we need something to control. So first, lets implement a class to represent our robot. In a separate header file (& optionally `.cpp` file), we might write:

```
#include <base/base>
#include <robot/robot>  // including the module

#include <robot/Robot>  // and the abstract class Robot

using robot::Robot;

```

[9] Currently, OpenSim follows the C++ convention of not having extensions on header files. However, this may change in future to more easily accommodate operating systems that force mixing of file names and file meta-data – such as the common Microsoft Windows practice of using three character name extensions to indicate file content type.

```
class MyRobot : public Robot
{
public:
  MyRobot()
  {
    // call any vendor robot init functions
    vendor_init(); // for example
    ...
  }

  // standard method we have to implement from base::Object
  virtual String className() const { return String("MyRobot");}

  // These two methods must be implemented (they are abstract in class Robot)

  // This method enumerates the available ControlInterface names and types.
  virtual array<std::pair<String,String> > controlInterfaces() const
  {
    // return an empty array of ControlInterface <name,type> pairs, as we
    //  don't provide any yet!
    return array<std::pair<String,String>();
  }

  /// get a ControlInterface by name
  virtual ref<ControlInterface> getControlInterface(String interfaceName="")
                                            throw(std::invalid_argument)
  {
    throw std::invalid_argument("control interface not available");
  }

};
```

Now we have a class that represents our robot, which doesn't do anything useful. Before we delve into the two methods above, a short overview of the `ControlInterface`, `Controller` and `Controllable` classes is in order.

2.1.1 `Controllers`, `Controllables` and `ControlInterfaces`

The `robot` module defines two interfaces (abstract classes) called `Controller` and `Controllable`. These are intended to represent, not surprisingly, objects that can be controlled (a manipulator, for example) and objects that do the controlling (a motion controller or path planner, for example), respectively. Note that any class can be both a `Controller` and `Controllable`–which is typically the case for many classes; save those that either interface directly with the actual hardware or the most top-level controller. A `Controller` interacts with a `Controllable` by way of a `ControlInterface`–also a standard interface. A `ControlInterface` provides a number of real valued inputs and outputs[10]. For example, a `ControlInterface` for the velocity control of the joints of a manipulator might have one output for each joint velocity and possibly one input to read back each joint encoder position.

[10] It is possible for a `ControlInterface` to have 0 inputs or 0 outputs. For example, some sensors may be represented by a `ControlInterface` that provides only inputs, no outputs.

TODO: Control system figure here

A `Controllable` is simply any class that can provide one or more `ControlInterface` objects. A `Controller` is just any class that embodies a control loop–that typically calls the `setOutput()` and `getInput()` method of one or more `ControlInterfaces`.

```
class ControlInterface : public base::ReferencedObject, public base::Named
{
public:
  ControlInterface();
  ControlInterface(const String& name, const String& type);

  const String& getType() const;

  virtual Int           inputSize() const;
  virtual String        inputName(Int i) const;
  virtual Real          getInput(Int i) const;
  virtual const Vector& getInputs() const;

  virtual Int    outputSize() const;
  virtual String outputName(Int i) const;
  virtual void   setOutput(Int i, Real value);
  virtual void   setOutputs(const Vector& values);
};

class Controllable : virtual public base::ReferencedObject
{
public:
  /// Provide ControlInterface for named interface (or default interface)
  /// throws std::invalid_argument if the interface name is unknown.
  virtual ref<ControlInterface> getControlInterface(String interfaceName="")
                                       throw(std::invalid_argument);
};

class Controller : virtual public base::ReferencedObject
{
public:

  /** Provide ControlInterface through which Controller may control.
      Can be called multiple times to pass multiple ControlInterfaces.
      Unknown ControlInterface types will be ignored. */
  virtual void setControlInterface(ref<ControlInterface> controlInterface);

  /** Query if the Controller has been passed all the ControlInterfaces it
      needs via setControlInterface() */
  virtual bool isConnected() const;


  /** Execute an iteration of the control loop.
      returns true if it wants to quit the loop - however this may
      be ignored by the user/caller. */
  virtual bool iterate(const base::Time& time);
};
```

Listing 2.1.1 - Signatures for `ControlInterface`, `Controller` and `Controllable` classes

Each `ControlInterface` has a string name and a string type. Although these are free-form and can be anything in your own implementations, they are intended to indicate something about the type of interface represented and a specific name for each one the Controllable provides. For example, the

SimulatedRobot class implemented in robot::sim that we'll use later, provides a ControlInterface of type "JointForceControl" for each manipulator mounted on the (simulated) robot, accessed via the names "manipulator*N*" (where *N* is the manipulator index). These names and types are meant to provide some loose run-time checking that only appropriate connections between Controllers and Controllables are made. The names and types also show in the graphical tools for running simulations. You may have noticed that the ControlInterface also provides names for individual inputs and outputs. This mainly used for debugging simulations and may be ignored when implementing ControlInterfaces for embedded use.

Note

The Input/Output terminology for ControlInterfaces can sometimes be confusing. It is intuitive from the client/user of a ControlInterface–i.e. the outputs are those values the client writes to a ControlInterface in order to command a Controllable object; while the inputs are values that the client reads back from Controllable via a ControlInterface. When actually implementing a ControlInterface subclass, this can seem backward, as the outputs are the values passed to the object and the inputs are those values it must supply to the client/user.

2.1.2 A ControlInterface for our robot's manipulator

The Robot class inherits from Controllable, and hence must implement the two methods: controlInterfaces() - that provides the list of the ControlInterfaces it can provide; and getControlInterface(*name*) - that gets a particular ControlInterface instance, given its string name.

Looking at the documentation (or header) for ManipulatorPIDPositionController, we can see that it requires a ControlInterface of type "JointVelocityControl". In turn, as a Controllable, it provides an interface of type "JointPositionControl" through which we can command the joint positions but invoking the setOutput() method. Let is implement a ControlInterface derived class called MyVelocityInterface and update the relevant methods of our MyRobot class to return it. Note that, as any other code that uses our interface class doesn't need to know its actual type (just that it is a subclass of ControlInterface), we can hide the definition as a nested class of MyRobot. If the implementation gets to be too large, we can always move it into its own header and .cpp files later.

```
class MyRobot : public Robot
{
public:
  MyRobot()
  {
    // call any vendor robot init functions
```

```
    vendor_init(); // for example
    ...
  }

  // standard method we have to implement from base::Object
  virtual String className() const { return String("MyRobot");}


protected:  // hide the definition of our interface
  class MyVelocityInterface : public BasicControlInterface
  {
     MyVelocityInterface()
       : BasicControlInterface("myVel", "JointVelocityControl",// name, type
                               7, 7) // 7 inputs & 7 outputs
     {}

     // standard method we have to implement from base::Object
     virtual String className() const { return String("MyVelocityInterface");}

     // only really important when writing a simulated Robot
     virtual String inputName(Int i) const { return "position"; }
     virtual String outputName(Int i) const { return "velocity"; }

     // read the joint encoder value as pass it back to the client/Controller
     virtual Real   getInput(Int i) const
     {
       return vendor_get_joint_encoder(i);
     }

     // take the client/Controller's velocity value for a particular
     //  joint 'i' and set it using our robot's vendor provided function
     virtual void   setOutput(Int i, Real value)
     {
       vendor_set_joint_velocity(i, value);
     }

  }; // end of MyVelocityInterface


public:
  // These two methods must be implemented (they are abstract in class Robot)

  // This method enumerates the available ControlInterface names and types.
  virtual array<std::pair<String,String> > controlInterfaces() const
  {
    // return an empty array of 1 ControlInterface <name,type> pair
    array<std::pair<String,String> interfaces;
    interfaces.push_back(std::make_pair<String, String>("myVel",
                                                       "JointVelocityControl");
    return interfaces;
  }

  /// get a ControlInterface by name
  virtual ref<ControlInterface> getControlInterface(String interfaceName="")
                                              throw(std::invalid_argument)
  {
    // if either MyVel or the default is requested, return our interface
    if ((interfaceName=="") || (interfaceName=="myVel"))
      return ref<ControlInterface>( NewObj MyVelocityInterface() );

    throw std::invalid_argument("control interface not available");
```

```
  }

};
```

That's it. Now we have a Robot class–a `Controllable`–that can provide a "`JointVelocityControl`" type interface that can be used to command its manipulator joint velocities and read back encoder values. We could have chosen any strings for the name and type of our `ControlInterface`, however as our interface is compatible with that documented in the `ManipulatorPIDPositionController` header we should use the same type name. Now that our robot has been integrated into part of the OpenSim framework, we can use any of the preexisting OpenSim Controller components that use the same interface type. Also, if we write our own `Controller` that commands an interface of type "`JointVelocityControl`", we can also use it to control a simulated robot – as the `robot::sim` module contains a `SimulatedRobot` implementation that can provide an interface of that type.

One detail to notice in our implementation, is that rather than our interface class inheriting directly from `ControlInterface`, we've used a convenience class called `BasicControlInterface`. This simply implements some of the required abstract methods of `ControlInterface` for us. For example, it implements `setOutputs(`*Vector*`)` as a loop that calls our `setOutput()` implementation for each joint in turn. If there is a more efficient way to command multiple joint velocities simultaneously using the vendor supplied functions for your robot, you may want to override the implementation of `setOutputs` yourself.

2.1.3 Instantiating a joint position controller

Now we are ready to fill out our main program to instantiate a joint-position controller, connect it to our joint-velocity interface and command a particular joint configuration of the manipulator. Back to our main function:

```
int main(int argc, char *argv[])
{
  // every OpenSim program needs to declare/instantiate a single
  //  Application object.
  Application app("/resources","/cache");

  // create an instance of our Robot (can only be one as it represents a
  //  real physical robot!)
  ref<MyRobot> myRobot( NewObj MyRobot() );

  // create a joint-velocity ControlInterface
  ref<ControlInterface> velInterface( myRobot->getControlInterface("myVel") );

  // create a preexisting ManipulatorPIDPositionController, which is a
  //  controller that uses simple PID control to maintain commanded joint
  //  position configurations by commanding velocities
  // (it is both a Controller and a Controllable)

  ref<ManipulatorPIDPositionController> controller(
                              NewObj ManipulatorPIDPositionController(chain) )
```

```
  // tell the controller which interface to command
  controller->setControlInterface( velInterface ); // i.e. ours

  // set its parameters
  controller->setCoeffs(Kp, Ki, Kd);

  // Now, obtain a joint-position control interface through which we
  //  will command the controller (no need to supply a name as there is
  //  only one - the default)
  ref<ControlInterface> posInterface( controller->getControlInterface() );

  // command the joint configuration we want (this could of course be
  //  updated in the control loop - but we just want a fixed configuration
  //  for simplicity in this example)
  posInterface->setOutputs( configurationVector );

  // finally, we're ready for the control loop.
  //  just keep calling iterate()
  for(;;) { // i.e. for ever

    // might want to read sensors, do other stuff here...

    // only one controller in this example
    controller->iterate( Time::now() );
  }

  return 0;
}
```

That concludes our example of how to control joint-positions of our hypothetical robot with 7-dof manipulator. Many details were omitted for brevity and will be covered in the following sections of the manual. For example, you may have noticed that the PID controller class requires a `chain` argument upon construction. This is an instance of the `KinematicChain` class that is used to describe the kinematics of serial manipulators. The controller needs such a description so that it knows which joints are revolute for intelligently handling when angular 'position' values wrap around. Many details of supporting classes, such as the smart-pointer `ref<>` class, vectors etc. and error handling were also omitted.

That should give you a flavor of what developing with and building on the OpenSim framework is like. The following sections of the manual give a more in-depth tutorial on many of the classes found in the core modules. Following that, a walk-through tutorial for creating dynamic simulations and 3D visualizations is presented.

2.2 The `base` module

2.2.1 Basic Types

The base module provides a set of typedef aliases for various types that are designed to be used by OpenSim and extensions to make it easy to switch their concrete types for specific operating-system and/or CPU platforms. For example, it would allow you to change you entire code base from using

double precision to single precision floating point values (or anything you wanted if you implement a class with operator overloading and value semantics that behaves like the built-in floating point types), just by changing the typedef in the base/base header.

The types are as follows:

- `Int` – unsigned integers (typically `unsigned int`)
- `SInt` – signed integers (typically `int`)
- `LInt` – long unsigned integers (typically `long unsigned int`)
- `Real` – floating point values (typically `double`)
- `String` – character strings (typically `std::string`)
- `Byte` – an 8bit quantity (typically `unsigned char`)

2.2.2 Common types

Storage management

The OpenSim code almost exclusively manages heap storage via reference counting. Instead of using raw C-style pointers (using the `*` operator), instead the code uses a smart pointer class, `ref`, which behaves in a similar manner to regular pointers, but allows referenced objects to count how many pointers refer to them. Once the reference count of an object instance falls to 0, it is automatically deleted. This is commonly termed *garbage collection*. It avoids having to think about who 'owns' object instances and hence where they should be `deleted` (as they don't need to be explicitly `deleted` at all)–and also avoids memory leaks.

The advantage of a reference counted implementation of garbage collection is that object deletion is deterministic and under full control of the developer–which is important for real-time systems. Disadvantages are that all classes must inherit the `ReferencedObject` class (so classes from other libraries developed independently from OpenSim cannot be managed with `ref<>` pointers); that an extra `int _refCount` field is added to every object (as a consequence of inheriting from the `Referenced` class); and that reference cycles will not be collected. These disadvantages could be avoided by other garbage collection techniques, such as incremental and/or generational garbage collection[11], but these techniques are not time deterministic.

As `ref<>` is a template class, it allows pointer type-checking like regular pointers, but the syntax is slightly different.

```
// declare a class that will use by-reference semantics
```

[11] These techniques are common among inherently garbage collected languages, such as Java and C#.

```
class MyRefClass : public ReferencedObject
{
  MyRefClass() ...
  ...

  void method() ...
};

// create an instance
ref<MyRefClass> myRefClass( NewObj MyRefClass() );

// call method, just like a C pointer
myRedClass->method();

// another pointer to the same object
ref<MyRefClass> prefd; // initially null (0)
prefd = myRefClass;
prefd->nethod(); // call same method on same instance

if (prefd) {
  printf("prefd is non-null");
}
```

Much the same semantics as C pointers is supported. Reference pointers can be assigned according to the rules of pointer compatibility (e.g. A pointer `ref<`*Sub*`>` can be assigned to a `ref<`*Super*`>` where *Sub* inherits *Super*; the `->` operator will perform virtual lookup as usual). Some C pointer manipulations are explicitly disallowed for type safety, such as pointer arithmetic, assigning a reference pointer from a C pointer etc. For example:

```
ref<MyRefClass> r;
r = new MyRefClass(); // compile error.
```

This will not work as the only time a C pointer can be 'assigned' to a reference pointer is during initialization, at which time it is assumed that you've created a new instance of the object. It is still possible to abuse this however:

```
MyRefClass myrefClass;
ref<MyRefClass> r(&myrefClass);
```

This will compile fine, but will probably cause a crash or segmentation fault at run-time (if you're lucky!)[12]. It allocates an object on the local stack, then gives the address of it to the reference pointer. This is a mistake as the object will be `delete`d when its reference count drops to 0 (for example, when the `ref<> r` goes out of local scope), but the object was not allocated from the heap using `new`.

Problems can be avoided by following some simple conventions:

- Divide your classes into two logical types: those that have *reference* semantics and those with *value* semantics.
  (examples from OpenSim: `Vector`, `Matrix` & `Expression` have value semantics, while `Robot` has reference semantics)
- Make all reference types inherit from `ReferencedObject`

---

[12]For debug builds the `Referenced()` constructor explicitly checks that the instance isn't on the stack and throws a `std::runtime_error` exception if it is.

- Always instantiate reference type instances like this:
  `ref<MyReferenceClass> refptr( NewObj MyReferenceClass(arg0, arg1...) );`
  - Note that `NewObj` is just a macro for `new` that can be redefined to add allocation tracking for debugging/profiling etc.
  - Sometimes you'll also see this:
    ```
    ref<MyReferenceClass> refptr =
              ref<MyReferenceClass>( NewObj MyReferenceClass(arg0...) );
    ```
    Most compilers transform this into the constructor form above rather than calling the default 0 argument constructor or `ref<>` first followed by its assignment operator.
- Don't use `delete` (that's why we're using automatic garbage collection after all). That is:
  `ref<MyRefClass> r( NewObj MyRefClass() ); delete &(*r);`
  will cause the object to be `delete`d twice!
  - Note that, if you don't defined an overridden virtual destructor for your class, the above statements won't even compile as the `Referenced` destructor is protected (for this reason) and can only be called by `Referenced` itself (via `delete` when its reference count goes to 0).
- Never place reference types on the stack (i.e. As local variables) or as struct/class fields (use `ref<>`s instead).
- Never pass by value, always pass a `ref<>` (which is just as efficient as passing a C pointer on most compilers):
  `ref<ConvexPolyhedra> convexHull(ref<const Polyhedra> p) {...}`
- If you'd like to make shallow copies of the objects, implement the copy constructor, inherit `base::Cloneable` and implement the `clone()` method in terms of the copy constructor; if not, write a `protected` copy constructor to prohibit copies.

- Make all value semantic types inherit from `Object`[13] (not `ReferencedObject`).
  - Make the object semantics behave like the built-in types (int etc.), by implementing both the copy constructor **and** `operator=()`.
  - If the object size is equal to or smaller than that of an `int`, pass it by value. If it is larger, pass it using a C++ `const` reference (which has the same semantics, but avoids the copy):
    `Complex sum(const Complex& c1, const Complex& c1) { return c1+c2; }`
  - Keep in mind that `virtual` methods incur an overhead both at call time and for the storage of each object (as each object must include a pointer to the class's virtual function table). So, virtual methods are best avoided for classes meant to have value semantics.

[13]This is not strictly necessary if for some reason you can't, don't want to.

The array class

This is a dynamically sizable array class that has optional index range checking and automatic increasing capacity. It can be used like the std::vector class but provides some additional features.

- the indexing operator[] is range checked for debug builds, but not for release builds for efficiency.
- You can use the at(*index*) methods for indexing when you want range checking at runtime (even in release builds). In addition, the array size will be automatically 'grown' when the index is larger than the current size.
- In addition to the array(*size*) constructor, an array(*size*, *initial capacity*) constructor is also provided so that excessive repeated reallocation to increase the capacity can be avoided. Note that the capacity is independent of the size (size <= capacity).
- Some std::list style accessors are available.

The reflist class

The reflist<*T*> template class implements a linked list of ref<*T*>s. It behaves essentially like std::list< ref<*T*> >. It provides explicit differentiation between a list of ref<*T*> and a list of ref<const *T*>. In addition to the begin() and end() iterators it also provides const_begin() and const_end().

Attribute interface classes

The base module also contains a number of simple abstract classes meant to be used as interfaces to indicate that classes have some particular 'attribute'. We've already seen the Referenced class that indicates that a class that inherits it has the attribute of having a reference count and being garbage collected. Some others are:

- Named – instances of the object have a String name. Provides a public String getName() method and a protected setName(String) method that can be exposed by subclasses if desired.
- Cloneable – indicates that a class supports making a clone (copy) of itself (a shallow copy in terms of contained reference pointers) through the clone() method. This is especially useful because it is a virtual method; so a copy can be made even if the concrete type of the object is unknown (which it would have to be in order to make a copy by calling the copy constructor).
- Object – the root of (almost) all OpenSim objects. Currently just requires concrete subclasses to implement the className() method to return a String class name. This is used for debugging and serialization.
- Hashable – indicates that objects of the class can generate a hash value (an array of Bytes) from their value through a call to hashCode(). This can be used, for example, for hash table implementations of dictionary/map classes.

- `Serializable` – means that objects of the class implement the `serialize(Serializer&)` method and hence can be written to and restored from a serialization stream. See section 2.2.5 for more details.
- `Externalizable` – means that objects of the class implement the `externalize (Externalizer&, ...)` method and can read and write their state from/to an externalization stream, possibly in multiple formats. For example, many OpenSim classes can be externalized to human readable XML formats. See section 2.2.5 for more details.
- `Simulatable` – classes that inherit from `Simulatable` implement the `preSimulate()` and `simulateForSimTime(Time&)` methods, indicating that they can be 'simulated'. To implement your own `Simulatable` simply provide implementations of these methods and either attach your instance to the `Universe` directly via its `addSimulatable()` method, or call the `simulateForSimTime()` method yourself from within that same method of another `Simulatable`.

<u>Math related classes</u>

- `Math` – a class that embodies most of the usual mathematics functions found in platform math libraries. They are available as static methods. Some of the methods include: `sqr`, `sqrt`, `cube`, `abs`, `sign`, `pow` (power), `random`, `isNAN` (is Non-A-Number), `sin`, `cos`, `tan`, `asin`, `acos`, `atan`, `atan2`, `degToRad()`, `radToDeg()`, `equals(r1,r2,eps)` (i.e. equal to within *eps* tolerance), `zeroIfNeighbour`, `minimum`, `maximum` (2 & 3 argument versions), `bound(&v, lower, upper)`, `normalizeAngle`, `normalizeAngle2PI`, `inverse(Matrix)`, `nullSpace(Matrix, Int& nullSpaceRank, Real& k2)`, and `pseudoInverse (Matrix)`.
- `consts` – this is actually a nested namespace within base that holds a number of useful constants. They are used with the notation `consts::Pi`, `consts::SqrtHalf` etc. Some of the constants include `Pi`, `TwoPi`, `Infinity` (=`maxReal`), `minReal`, `maxReal`, `minSInt`, `maxSInt`, `minInt`, `maxInt`, `epsilon` ($1x10^{-10}$), `epsilon2` ($1x10^{-20}$), `inchesPerMeter` and `metersPerInch`.
- `Vector` – a mathematical vector of arbitrary dimension. Constructed with the dimension as argument (e.g. `Vector(3)` for a 3D vector). Has overloaded `operator()` so it can be index (0-based) via the notation `v(i)`. Methods like `size()`, `resize(newdim)`, `norm()`, `magnitude ()` (sqrt of norm), `negate()`, iterators (begin & end), etc.
  - Note that there are also `Vector2` and `Vector3` classes that are specialized for fixed 2&3D vectors. However, these will probably be deprecated in future in favor of `Vector`. There are conversions to/from `Vector3` via the functions `toVector3(Vector)` and `fromVector3 (Vector3)`.

  - The `Vector` header also supplies some utility types and functions for manipulating `Vectors`. All the usual mathematical operators are overloaded, there are functions for `dot(v1,v2)` (dot product), `Range(start, end)`, `vectorRange(Vector, Range)` (e.g. `vectorRange(v, Range(0,2)) = v2` would assign the elements of vector *v2* to the subset of elements of *v* in the range *[0..2)*) and `zeroVector(dim)`.

- `Matrix` – a mathematical matrix or arbitrary rows and columns. Constructed with the number of rows and columns as arguments (e.g. Matrix(2,3) for a 2x3 matrix). Has overloaded operator() so it can be indexed (0-based) via the notation m(r,c). Methods like rows(), cols(), resize(rows,cols), negate(), and transpose().

  - Note that there are also `Matrix3` and `Matrix4` classes that are specialized for fixed 3x3 and 4x4 matrices. However, these will probably be deprecated in future in favor of `Matrix`. There are conversions to/from `Matrix3` and `Matrix4` via the functions `toMatrix4(Matrix)`, `fromMatrix4(Matrix4)`, `toMatrix3(Matrix)` and `fromMatrix3(Matrix3)`.

  - The `Matrix` header also supplies some utility types and functions for `Matrix` manipulations. All the usual mathematical operators are overloaded and functions are provided for `matrixRow(Matrix, r)`, `matrixColumn(Matrix, c)`, `matrixRange(Matrix, rowrange, colrange)`, `transpose(Matrix)`, `zeroMatrix(r,c)`, `identityMatrix(r,c)`, etc. For example, `matrixRange(m, Range(0,2), Range(0,2)) = identityMatrix(2,2)` would set the upper left 2x2 sub-matrix of m to the 2x2 identity).

- `Orient` – the `Orient` class represents an orientation in 3D space. It makes using an orientation possible without knowledge of the underlying representation used and allows easy conversion between many different possible representations. The current representation schemes supported include 3x3 transformation matrices, Euler parameters (which use quaternion arithmetic), Cardan-Bryant (all combinations of anti-cyclic rotations about the 3 principle axes, e.g. XYZ, XZY, etc.), Euler angles (all combinations of cyclic rotations about

2.2.3 Debugging aids

Exception, Debugln, etc. Assert

2.2.4 Platform abstractions

VEntry & friends, Time, PathName, Application

2.2.5 Serialization and Externalization

2.2.6 Utility classes

Trajectory, Path, Orient, Transform, point, EventListener, Expression, Serializer, Externalizer, Externalizable, Dimension3, Cache, MD5, Universe, World, SVD

2.2.7 Unit tests

TODO: write

2.3 Creating a simple simulation

This section gives a walk-through tutorial on how to create a simple dynamic, multi-robot simulation and visualize it in 3D. This will give you an idea of how the various classes come together to create simulations so that you can better understand how to go about extending OpenSim with your own implementations of controllers, simulated sensors, robots, manipulators and other components. If you just want to instantiate a simulation using existing components you can just specify it using the *environment* XML file format and view it with the `viewenv` program (see the User Guide part of the manual for details).

We start with a main program similar to the one presented in the introduction. This time, we'll assume the program will be built on a desktop/workstation platform with the 3D graphics, graphical user interface (GUI) and file I/O available. Below is a screen-shot of the end result.

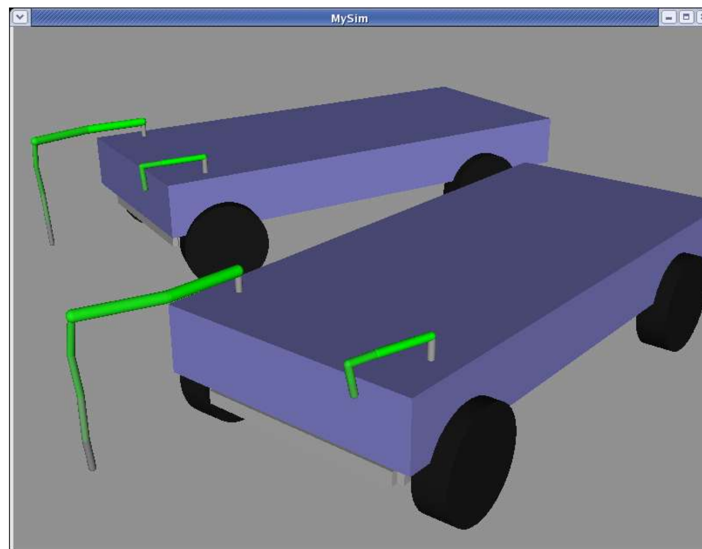

Figure 4 - Screen-shot of simple 3D simulation tutorial

The source code for the program is included with the distribution as `sim_tut1.cpp` in the `apps` directory. It is reproduced below for easy reference. Parts of the listing are labeled numerically and will be referred to in the text as 'part *n*'.

```
#include <base/base>
#include <robot/robot>
#include <robot/sim/sim>

#include <base/Application>
#include <base/Universe>
#include <base/Math>
#include <base/Dimension3>
#include <gfx/TrackballManipulator>

using base::Application;                                        ◄1
using base::Universe;
using base::Vector3;
using base::Matrix4;
using base::Dimension3;
using base::Point3;
using base::Orient;


#include <robot/RobotDescription>
#include <robot/sim/SimulatedBasicEnvironment>

using robot::RobotDescription;
using robot::PlatformDescription;
using robot::ManipulatorDescription;
using robot::KinematicChain;
using robot::sim::SimulatedBasicEnvironment;


// Producer / OSG for visualization
#include <osgProducer/Viewer>



int main(int argc, char *argv[])
{
  try {

    // Assume that the user has define an OPENSIM_HOME environment variable
    // that points to the top of their OpenSim installation
    // We'll assume that the resource and cache directories are relative
    //  to that

    char *homeenv = getenv("OPENSIM_HOME");
    Assertm(homeenv!=0, "environment variable OPENSIM_HOME defined");   ◄2
    String home(homeenv);

    // create singleton app
    Application app(home+"/resources",home+"/cache");      ◄3
    app.displayHeader("MySim");

    // make a universe in which everything is simulated      ◄4
    ref<Universe> universe = app.universe();

    // make use of the robot::sim::SimulatedBasicEnvironment to provide a simple
    // environment (with a ground, gravity etc.) in which to simulate our robots
    ref<SimulatedBasicEnvironment> env(NewObj SimulatedBasicEnvironment(
                                             universe->filesystem(),    ◄5
                                             universe->cache()
                                          )
                                   );

    // before we can add any Robots to the environment, we need to describe them.
    // We do that 'manually' here via code rather than loading a description from a file
    //  for instruction.

    //
    // Describe a wheeled robot with two manipulators
```

```
ref<RobotDescription> robotDesc( env->newRobotDescription() ); // initially has no
                                                          // platform or manipulators

// wheeled platform
ref<PlatformDescription> platformDesc( robotDesc->newPlatformDescription() );
platformDesc->set("MyBuggyPlatform",     // name
                  Dimension3(4.5,2,0.5), // dimensions (m)
                  Vector3(1.2,0,0.25),   // origin offset          ◄6
                  true,                  // is mobile
                  false,                 // non-holonomic
                  2.5, 3                 // axle offset constants (L, W)
                 );

// kinematic chain of first manipulator
KinematicChain chain1; // initially empty chain (no links)
// add 4 links using Denavit-Hartenberg (DH) notation - alpha, a, d, theta
                                   // make an alias for Revolute to save typing
const KinematicChain::Link::LinkType& Revolute = KinematicChain::Link::Revolute;

chain1 += KinematicChain::Link( Revolute, 0,   0.4,  0, 0 );
chain1 += KinematicChain::Link( Revolute, 1.5, 0.2,  0, 0 );        ◄7
chain1 += KinematicChain::Link( Revolute, -1,  0.12, 0, 0 );
chain1 += KinematicChain::Link( Revolute, 0,   0.1,  0, 0 );

// describe a manipulator using chain1
ref<ManipulatorDescription> manipDesc1( robotDesc->newManipulatorDescription() );

Matrix4 m; m.setIdentity();
m.setColumn(4, Vector3(0,0,0.15)); // translation up Z by 0.15

manipDesc1->set("SmallManip", // name
                m,            // transform from platform to first joint  ◄8
                chain1);      // kinematic chain


// next, describe another manipulator, with more links this time
KinematicChain chain2; // initially empty chain (no links)
// add 6 links using Denavit-Hartenberg (DH) notation - alpha, a, d, theta
chain2 += KinematicChain::Link( Revolute, 0,   0.6,   0, 0 );
chain2 += KinematicChain::Link( Revolute, 1.3, 0.6,   0, 0 );
chain2 += KinematicChain::Link( Revolute, 1,   0.3,   0, 0 );      ◄9
chain2 += KinematicChain::Link( Revolute, -2,  0.3,   0, 0 );
chain2 += KinematicChain::Link( Revolute, 0,   0.3,   0, 0 );
chain2 += KinematicChain::Link( Revolute, 3,   0.22,  0, 0 );

ref<ManipulatorDescription> manipDesc2( robotDesc->newManipulatorDescription() );

manipDesc2->set("BigManip", // name
                m,          // transform from platform to first joint   ◄10
                chain2);    // kinematic chain

// now we've described the platform & manipulators,
//  we can use these to desscribe the robot

// first, put the manipulators in an array
array<ref<const ManipulatorDescription> > manipDescrs(2);
manipDescrs[0] = manipDesc1;
manipDescrs[1] = manipDesc2;
// and an array of the mount point offsets relative to the platform ◄11
array<Vector3> manipOffsets(2);
manipOffsets[0] = Vector3(0.6,0.8,0.0);
manipOffsets[1] = Vector3(0.6,-0.8,0.0);

// Now place the platform and manipulator descriptions in our robot description
robotDesc->set("MyBuggy",    // robot name
               platformDesc, // platform
               manipDescrs,  // manipulators     ◄12
```

```
                manipOffsets
               );


// OK, now we have described our robot, lets add two instance to the environment
env->addRobot(robotDesc, Point3(0,0,1.4), Orient(0.01,0.01,0) );            ◄13
env->addRobot(robotDesc, Point3(0.5,4,1.4), Orient(0.01,0.01,0.3) );


// now tell the universe about our environment
universe->addWorld(env);         // tell universe to visualize it       ◄14
universe->addSimulatable(env);   // tell universe to simulate it




//
// Next, if we want to visualize the simulation in 3D we need to setup
//  a window to render into using OpenProducer / OSG
// construct the viewer.
//  (this is fairly boiler-plate code - refer to the Producer & OSG docs)
Producer::RenderSurface *rs = new Producer::RenderSurface;
rs->setWindowName("MySim");
rs->setWindowRectangle(0,400,800,600); // set window pos/size       ◄15
Producer::Camera *camera = new Producer::Camera; // a camera
camera->setRenderSurface(rs);
Producer::CameraConfig *cfg = new Producer::CameraConfig;
cfg->addCamera("Camera",camera);

osgProducer::Viewer viewer(cfg);

// set up the viewer with sensible default event handlers.
viewer.setUpViewer(osgProducer::Viewer::STATE_MANIPULATOR |
                   osgProducer::Viewer::HEAD_LIGHT_SOURCE |
                   osgProducer::Viewer::STATS_MANIPULATOR |
                   osgProducer::Viewer::VIEWER_MANIPULATOR |            ◄16
                   osgProducer::Viewer::ESCAPE_SETS_DONE );

// a camera manipulator to allow us to move the view around via the mouse
gfx::TrackballManipulator* cm = new gfx::TrackballManipulator();
viewer.selectCameraManipulator( viewer.addCameraManipulator(cm) );


// Now, ask the environment to create a scene (an OSG Visual)
osg::Group* sceneRoot = NewObj osg::Group;       // an empty OSG group node◄17
sceneRoot->addChild( env->createOSGVisual() );
                                 //  now with the whole env visualization as a child
// and tell the viewer of our scene
viewer.setSceneData(sceneRoot);

// open the window
viewer.realize();

// move the camera to somewhere sensible so we can see something
cm->setModelScale(2);
                      // eye,             center,             up
cm->computePosition(osg::Vec3(9,2,6), osg::Vec3(0,2,0), osg::Vec3(0,0,1));




// Finally, we need a main loop
//  here we step the simulation and update the view

universe->preSimulate(); // initialization step  ◄18

// this simple main loop doesn't have any frame-rate control or other waiting,
//  so it will just render frames as fast as possible (100% CPU utilization!)
```

```
    while(!viewer.done()) {

      // wait for draw/cull/update traversals to finish before changing the world state
      viewer.sync();

      // step the simulation (for 100th of a sec - simulation time, not real-time)
      universe->simulateForSimTime(1.0/100.0);

      // update the scene by traversing it
      viewer.update();                                                  ◄19

      // fire off the traversals (e.g. draw)
      viewer.frame();

    } // end main loop (when ESC hit)


    viewer.sync();

  } catch (std::exception& e) {
    Consoleln("caught: " << String(e.what())); // display error info
  }

  Consoleln("Exiting.");

  return 0;
}
```

2.3.1 Setup

In the first part (1) of the program, we include the header files for the various classes we wish to use. Some classes may not be explicitly listed if we know that the header we're including will include them in turn. For example, the `robot/robot` header includes many of the basic types commonly used by the robot API (`Vector3` and `Orient`, for example). Here we also bring the classes we will be using into the current *namespace*. As discussed in the tutorial introduction, only explicitly listing the types we want in this way is good software engineering practice because it avoids polluting the current namespace with unwanted names that might clash with names we want to use later. However, sometime when we're just prototyping or quickly need to write short programs we're not so concerned such issues. In that case much of the verbosity of includes and using statements may be avoided by creating a common header file that includes all the headers you routinely use and including that in all your applications. Also, you can bring all symbols in a given namespace into the current namespace with using statements like the following:

```
using base;
using robot;
using robot::sim;
```

Next, comes the main function of the program. First it performs some initialization, then describes some robot parts – like platforms and manipulators. These descriptions are then used to instantiate two virtual robots that are added to a simulated environment. Finally the visualization of the environment is setup and a main loop to continuously step the simulation in small increments is entered.

Parts 2 and 3 create a single instance of the `Application` class and pass it the locations of our resources and cache (as discussed in section 2.1). Part 4 creates a new empty `Universe`. The universe is a container for a set of `Worlds` and `Simulatables`. A world represents the visualization of something (in our case a simulated environment containing robots). A *simulatable* is anything that maintains some simulation state and typically needs to be stepped during a simulation. The implementation of the whole environment with robots is a `Simulatable` that in turn encapsulates other simulatables, such as the rigid-body physics model, collision detection, controllers and the like.

Next, part 5 of the program uses a class that is provided in the `robot::sim` library, which can manage and visualize a simple simulated environment consisting of multiple robots, objects and a flat 'ground' plane with gravity (the Earth's acceleration by default). Initially the environment is empty–consisting of only the ground. Once we've create some robots we can add them to this environment.

2.3.2 Describing a Robot

In order to instantiate some virtual robots, we first need to obtain or create a description of a robot and its parts. Although the `robot` and `robot::sim` libraries provide facilities for specifying robot parts, robots, sensors and whole environments via files which can be read into the program, what these

represent will be made clearer if we construct our robot programmatically–which is what this tutorial program does beginning with part 6.

At the beginning of part 6 a new `RobotDescription` is created ready to hold the various robot parts. We start by describing the platform, which is the main body of the robot–the part to which the coordinate frame of the robot is fixed.

As an aside, you may have noticed a pattern in the way instances of the description classes are created. In particular, they are each created via *factory* methods[14]. So, the `RobotDescription` was instantiated using the `newRobotDescription()` method of `Environment`; while the `PlatformDescription` was instantiated via the `RobotDescription`'s `newPlatformDescription()` method (and `newManipulatorDescription()` for the manipulator). Why use these factory methods instead of simply instantiating them using a statement like `NewObj RobotDescription()`? The factory method is the preferred way because within the libraries there may be a whole family of different classes that inherit from `RobotDescription` (for example). The basic robot description is very generic in nature. However, sometimes you (or the library) need to describe properties of a robot or part that only pertain to a specific use. For example, the `robot::sim` library uses a class called (you guessed it) `SimulatedRobotDescription`. This adds properties that are only relevant for simulations to the description. The `robot::RobotDescription` class is no place for such specialized properties–for example, you may want to build OpenSim with the `robot` library for a computer embedded on a physical robot and wouldn't want to include the `robot::sim` library at all. In summary, the preferred way to create the description classes is via appropriate factory methods. In that way the actual concrete class used to implement the description is hidden from you as you just manipulate references to the generic `RobotDescription`, `PlatformDescription` etc. (although the specific classes can be obtained if you need access to non-generic properties).

Now, back to the program. Part 6 continues by setting the various properties of the platform description, including its dimensions and the wheel axle parameters L and W (more on these in the reference section). Next, in part 7, a manipulator description is created to describe a jointed serial manipulator that will be attached. First, however, the serial chain needs to be described. This is achieved using the `KinematicChain` class. The `KinematicChain` class has value semantics, so it doesn't need to be referenced using the `ref` smart-pointer. When assigning or passing instances to functions or methods, logically a copy will be made (although the implementation will avoid actual copying where possible for efficiency). It allows various manipulations of a chain of links, including adding new links to the end of the chain via the overloaded addition operator. The `Link` class itself also has value semantics and is a nested class of `KinematicChain`. There are numerous types of links and ways to specify them, but only the Denavit-Hartenberg notation is used in the program, which creates 4 new `Links` and adds them to the end of the chain. Next, a new `ManipulatorDescription` is created

---

[14]You can find a detailed explanation of the creational *factory pattern* in any good book on software design patterns near a bookshelf (or on-line library) near you. For example, the classic Gang of Four (GOF) book, "***Design Patterns**, Elements of Reusable Object-Oriented Software*", Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides.

through a factory method. Its properties are set in part 8; a name, the kinematic chain just constructed and a homogeneous 4x4 transformation matrix that specifies the transformation from the 'mount point' to the origin of the first joint of the manipulator. In this case, the transformation is just a translation of 0.15m up the Z-axis to lift the joint above the mount point on the top of the platform. Parts 9 and 10 are similar, but they create a second manipulator description, this time with more links and the name 'BigManip'.

Finally, in parts 11 and 12 the program uses the platform and manipulator descriptions to describe a robot. Since the RobotDescription object was already created near part 6, we just need to set its properties. The manipulator descriptions are supplied as an array, hence part 11 creates a two element array and assigns our manipulator descriptions as its elements. In addition, the robot needs to know where the manipulators are mounted in relation to the platform coordinate origin. These 'mount point' offsets are also supplied in an array (of Vector3s).

Now that a Robot has been described, via descriptions of its constituent parts, the program instantiates two in different positions, in the simulated environment (part 13). For each robot added to the environment a description, position and orientation is supplied.

2.3.3 Viewer setup

In order to actually visualize the simulated environment, we need to setup a window and somehow arrange for the environment to render itself. This is achieved in parts 15-17 of the program using the *Producer* and *Open Scene Graph* (OSG) libraries. Most of this code is boiler-plate code that can be copy&pasted or abstracted for reuse. The respective library documentation can be consulted for detail if required.

To summarize, first a window is created via the *Producer* library's RenderSurface class. A Camera is created, to define which part of the environment we want to be able see, and attached to the render surface. Next the OSG plug-in library for interfacing to Producer–osgProducer–is utilized to create a Viewer–which will manage asking the scene-graph to render itself onto the render surface and also manipulation of the camera in response to mouse movement (with the help of the TrackBallManipulator class).

In part 17, the simulated environment is asked to create an OSGVisual–which is an OSG scene-graph containing all the geometry representing the objects in the environment (i.e. two robots). The visual is added to a new empty scene-graph 'sceneRoot' and the viewer informed of the whole graph so that it can manage rendering it. Finally, the window is opened via viewer.realize() and the camera set to a reasonable initial configuration.

2.3.4 The main loop

Part 18 of the program calls the `Universe`'s `preSimulate()` method, which calls `preSimulate()` on all `Simulatables` attached. Typically a `Simulatable` uses this method to initialize itself prior to calls to `simulateForSimTime()` which actually 'step' the simulation.

Part 19 encompasses the main loop of the program. It is a while loop that exits when `viewer.done()` returns true (e.g. when the `ESC` key it pressed). If all the program needed to do was to render the environment repeatedly, it could just call viewer.frame() over and over in the loop. In fact, just calling it once would be enough as the environment would remain static. However, in addition to rendering the environment the program also needs to update the state of the simulation and also update the scene-graph to be rendered. The usual sequence of events dictated by Producer/OSG for updating and rendering a scene-graph is to call `viewer.sync()`, `viewer.update()` and `viewer.frame()` each time through the loop. The call to `sync()` is needed because OSG is multi-threaded and it doesn't make sense to request a frame to be rendered before the previous one has been finished. The `update()` call updates the state of the scene-graph itself. In this program the scene-graph is modified by changes to the configurations of the objects in the environment being simulated. These are updated by a call to the `Universe::simulateForSimTime()` method. Hence, it is logical to place this call just before the call to `update()`. The `simulateForSimTime()` calls the corresponding method on all simulatables attached to the universe–in this case just our `SimulatedBasicEnvironment` instance `env`.

So, the program loop calls `sync()`, `simulateForSimTime()`, `update()` and `frame()` in that order. Note that this will render frames as fast as the CPU and graphics hardware can manage. Depending on the complexity of the environment, this may be much faster that the update rate of your screen. Hence, although adequate for this example, in practice it would be better to place a delay in the loop to slow down the rendering to a reasonable rate (e.g. 30-100Hz).

Now, if you build and run the program, you'll see a window like that shown in Figure 4. If you are quick you may see the robots initially fall to the ground, as they were positioned slightly above it. The manipulators will just fall down like rag-roll arms as they are not controlled (no torque applied to the joints, nor dampening specified). You can move the camera around using the mouse with various button combinations.

In this section we've seen how to create a simulation and visualize it by programmatically describing the components of the robots we wanted to simulated and linking them together. If this seems like an arduous way to specify simulations then the next section will explain the easy alternative–specifying the simulation in a file and just loading the file at run-time to instantiate the simulation.

2.4 Creating a simulation from specification files

This second program demonstrates loading an environment specification, including the descriptions of all contained robots and their constituent parts, from an XML file stored in the resource directory. The first

and last parts of the program are identical to the previous one and the middle section that described all the robot components has been completely replaced with a few lines to load the environment description from a file instead (part 1 in the program listing below).

The environment description is stored in the resources directory, under a subdirectory called 'data' and in the file named '`sim_tut2_env.xml`'. This was created by adding a few lines to the program from the previous tutorial to save the environment into a file. You can examine the contents of the file with a text viewer or editor. To understand it you may need to refer to section 1.2 File Formats in the user guide part of the manual. Because it was generated via program output rather than hand written, it is a monolithic description of the whole environment in a single file. In particular, there are two robots in the environment and the descriptions of their manipulators and platforms are embedded in each – even though both are the same (recalling from the previous program, we instantiate two identical robots). Typically, if hand writing environment specifications, the specification would be split over several files to avoid repetition and help maintainability. For example, the platform and manipulators may be specified in separate XML files and just referenced by a robot XML file, which would be referenced in turn by the environment file (twice in this case). Please refer to the program listing.

```
#include <base/base>
#include <robot/robot>
#include <robot/sim/sim>

#include <base/Application>
#include <base/Universe>
#include <base/Math>
#include <base/Dimension3>
#include <gfx/TrackballManipulator>

using base::Application;
using base::Universe;
using base::Vector3;
using base::Matrix4;
using base::Dimension3;
using base::Point3;
using base::Orient;


#include <robot/RobotDescription>
#include <robot/sim/SimulatedBasicEnvironment>

using robot::RobotDescription;
using robot::PlatformDescription;
using robot::ManipulatorDescription;
using robot::KinematicChain;
using robot::sim::SimulatedBasicEnvironment;


// Producer / OSG for visualization
#include <osgProducer/Viewer>



int main(int argc, char *argv[])
{
  try {

    // Assume that the user has define an OPENSIM_HOME environment variable
    // that points to the top of their OpenSim installation
    // We'll assume that the resource and cache directories are relative
    //  to that

    char *homeenv = getenv("OPENSIM_HOME");
    Assertm(homeenv!=0, "environment variable OPENSIM_HOME defined");
    String home(homeenv);

    // create singleton app
    Application app(home+"/resources",home+"/cache");
    app.displayHeader("MySim");

    // make a universe in which everything is simulated
    ref<Universe> universe = app.universe();

    // make use of the robot::sim::SimulatedBasicEnvironment to provide a simple
    // environment (with a ground, gravity etc.) in which to simulate our robots
    ref<SimulatedBasicEnvironment> env(NewObj SimulatedBasicEnvironment(
                                            universe->filesystem(),
                                            universe->cache()
                                         )
                                 );

    // now load the complete description of the environment, with robots and
    //  their parts from the file 'data/sim_tut2_env.xml' in the resource
    //  directory
    ref<base::VFile> envFile( universe->cache()->findFile(
                                    String("data/sim_tut2_env.xml")) );
    env->load(envFile, "xml");
```

◄1

```
// now tell the universe about our environment
universe->addWorld(env);          // tell universe to visualize it
universe->addSimulatable(env);    // tell universe to simulate it


//
// Next, if we want to visualize the simulation in 3D we need to setup
//  a window to render into using OpenProducer / OSG
// construct the viewer.
//  (this is fairly boiler-plate code - refer to the Producer & OSG docs)
Producer::RenderSurface *rs = new Producer::RenderSurface;
rs->setWindowName("MySim");
rs->setWindowRectangle(0,400,800,600); // set window pos/size
Producer::Camera *camera = new Producer::Camera; // a camera
camera->setRenderSurface(rs);
Producer::CameraConfig *cfg = new Producer::CameraConfig;
cfg->addCamera("Camera",camera);

osgProducer::Viewer viewer(cfg);

// set up the viewer with sensible default event handlers.
viewer.setUpViewer(osgProducer::Viewer::STATE_MANIPULATOR |
                   osgProducer::Viewer::HEAD_LIGHT_SOURCE |
                   osgProducer::Viewer::STATS_MANIPULATOR |
                   osgProducer::Viewer::VIEWER_MANIPULATOR |
                   osgProducer::Viewer::ESCAPE_SETS_DONE );

// a camera manipulator to allow us to move the view around via the mouse
gfx::TrackballManipulator* cm = new gfx::TrackballManipulator();
viewer.selectCameraManipulator( viewer.addCameraManipulator(cm) );


// Now, ask the environment to create a scene (an OSG Visual)
osg::Group* sceneRoot = NewObj osg::Group;       // an empty OSG group node
sceneRoot->addChild( env->createOSGVisual() );
                                //  now with the while env visualization as a child
// and tell the viewer of our scene
viewer.setSceneData(sceneRoot);

// open the window
viewer.realize();

// move the camera to somewhere sensible so we can see something
cm->setModelScale(2);
                      // eye,            center,            up
cm->computePosition(osg::Vec3(9,2,6), osg::Vec3(0,2,0), osg::Vec3(0,0,1));



// Finally, we need a main loop
//  here we step the simulation and update the view

universe->preSimulate(); // initialization step

// this simple main loop doesn't have any frame-rate control or other waiting,
//  so it will just render frames as fast as possible (100% CPU utilization!)
while(!viewer.done()) {

  // wait for draw/cull/update traversals to finish before changing the world state
  viewer.sync();

  // step the simulation (for 100th of a sec - simulation time, not real-time)
  universe->simulateForSimTime(1.0/100.0);

  // update the scene by traversing it
  viewer.update();
```

```
      // fire off the traversals (e.g. draw)
      viewer.frame();

    } // end main loop (when ESC hit)


    viewer.sync();

  } catch (std::exception& e) {
    Consoleln("caught: " << String(e.what())); // display error info
  }

  Consoleln("Exiting.");

  return 0;
}
```

The `SimulatedBasicEnvironment` class inherits from `Environment` and hence is an `Externalizable`. This means it supplies the `externalize()` method and it's convenience methods `load()` and `save()`. Here the program uses the `load()` method to load the content of the `sim_tut2_env.xml` file into the `env` instance. The first parameter to load is a `VFile`–which is the abstract interface to all file-like I/O streams. The program uses the `Universe` cache's `findFile()` method to obtain a `VFile` from the supplied file-name, which is the easiest way to get a `VFile` from a file if it is located in the resource path. Although we could have bypassed the cache and used the file-system directly, for some file-formats that require preprocessing upon loading, the cache may be able to speed up the load process by re-using previously preprocessed and cached information. In this case, for the current implementation, nothing is cached, but it is good practice to use the cache interface anyway.

Once the environment has been loaded, the remainder of the program is identical to the program from the last section. If you compile and run this program (which can be found in `apps/sim_tut2.cpp`), you'll get the same result as before.

Although this allows us to examine the environment stored in the specification XML file, we could also have loaded it into the `viewenv` utility (see section 1.1.1 of the User Guide). This utility additionally allows us to perform some basic control of the robots and their manipulators.

3 Guide

3.1 Inverse Kinematics

The tutorial section of this guide explained how to programmatically instantiate a simulated environment containing a robot (or robots) and visualize via a simulation loop. In addition, section 2.1.1 described how a robot, or any actuated device that provides a `ControlInterface`, can be controlled utilizing the `Controller` and `Controllable` interface classes. This section explains how a robot manipulator can be controlled via inverse kinematics by employing the existing inverse kinematics controller classes from the `robot::control::kinematics` module.

3.1.1 Using `IKORController`

The easiest way to utilize the provided inverse kinematics functionality is to use the `IKORController` class. This class in turn uses a number of supporting classes that can be used directly if a lower-level of access is required. The supporting classes will also be described below. The `IKORController` class is named after a research project in which the techniques used by the supporting classes were developed–the IKOR project (Inverse Kinematics On Redundant systems)[15].

The `IKORController` class is both a `Controller` and a `Controllable`. As such, it needs to have a suitable `ControlInterface` passed to it via the `setControlInterface()` method–in this case the `ControlInterface` must implement the "`JointPositionControl`" type. For example, the `SimulatedRobot` class provides this interface type (for example, by calling `SimulatedRobot::getControlInterface("manipulatorPosition0");`). As a `Controllable`, the `IKORController` provides two interface types in turn (via its `getControlInterface()` method)–one for controlling and reading the end-effector position/orientation (whose type is "`EndEffectorPositionControl`") and another convenience interface for reading the positions of each link origin (of type "`LinkOriginPositions`").

The inverse kinematics computations are performed in the class's `iterate(base::Time)` method (a method inherited from `Controller`). Hence, the control program's main loop (or the `iterate()` method of a higher-level `Controller`) should periodically call `iterate()`.

Upon construction, the `IKORController` must be provided with the solution method, a `Robot` class, the index of the manipulator to be controlled and some other options. The solution methods

[15]see "*Resolving Kinematic Redundancy with Constraints Using the FSP (Full Space Parameterization) Approach*", Francois G. Pin & Faithlyn A. Tulloch, Proceedings of the 1996 IEEE International Conference on Robotics and Automation.

and "*Motion Planning for Mobile Manipulators with a Non-Holonomic Constraint Using the FSP (Full Space Parameterization) Method*", Francois G. Pin, Kristi A. Morgansen, Faithlyn A. Tulloch, Charles J. Hacker and Kathryn B. Gower, Journal of Robotic Systems 13(11), 723-736 (1996).

currently supported are `LeastNorm` and `FSPLagrangian`. The `LeastNorm` method uses a Pseudo-inverse computation to provide a solution that gives the least norm joint motion, but doesn't support any constraints. This is achieved using the `LeastNormIKSolver` class.

The `FSPLargrangian` method uses the `IKOR` class, described in the following section, which has support for constraints. By default, the `IKORController` creates constraints for joint-limits according to the limits obtained via the `KinematicChain` (which is in turn retrieved via the `Robot`'s `RobotDescription`). In addition, if the `Robot` instance passed can supply a `ControlInterface` of type "`LinkProximitySensors`" for the manipulator being controlled (through its `getControlInterface()` method), then obstacle avoidance constraints are used during the solution computations by providing link proximity sensor data to the `InverseKinematicsSolver` (`IKOR`) via its `setProximitySensorData()` method. The constructor also has a `platformActive` option, which when true informs the IKOR solver that the first 3 degrees-of-freedom in the kinematic chain represent a non-holonomic platform (*x*,*y*,*theta*) and enables a suitable non-holonomic constraint.

```
// create an IK controller
ref<IKORController> ikorController(NewObj IKORController(
                                      IKORController::FSPLagrangian,
                                      robot, // Robot instance
                                      index, // index of manipulator on robot to control
                                      false, // no platform dofs
                                      orientationControl, // control orientation?
                                      Orient::EulerRPY)// orient representation in state
                                  );

// give the controller the ControlInterface via which it can command jount positions
ikorController->setControlInterface(jointPositionInterface);

// set threshold distance from links, below which obstacle avoidance is triggered
ikorController->setProximityDangerDistance( 0.1 ); // 10cm

// get a ControlInterface via which the end-effector position[/orientation] can be
//  commanded (in this case of type "EndEffectorPositionControl").  No name is necessary
//  as this is the default interface returned.
ref<ControlInterface> eePosInterface( ikorController->getControlInterface() );

         …

// main loop
for(;;) {
         …

  // set new target end-effector pos[/orient]
  //  (vector must be 3-dim if no orientation control, otherwise it
  //   must be of dim 3+(no. orient components)
  //   - for example, if orientation is specified in EulerRPY format, 6-dim)

  eePosInterface->setOutputs(endEffectorTargetVector);

         …

  // iterate the controller
  //  - if IK needs to be computed this iteration, it will be inside this call
  //    (dx = difference between target and current position)

  ikorController->iterate( Time::now() );

         …
}
```

3.1.2 The Full-Space Parameterization approach

The IKOR solver class (used by `IKORController`) implements the Full-Space Parameterization (FSP) approach described in the articles referenced above. The solution is broken down into two parts. Firstly, a `FullSpaceSolver` (by default the `SVDFullSpaceSolver` class) is used to generate a vector-space of possible joint motions that satisfy the requested end-effector motion and incorporate the available redundancy in the manipulator. Any method may be substituted by deriving a class from the `FullSpaceSolver` abstract interface class. For example, the `IKORFullSpaceSolver` class implements the original technique described in the references (which has been superseded by the SVD method). Refer to the figures below for the relationships between the various classes and interfaces. Next, the space of solutions is narrowed to a single solution vector using Lagrangian optimization, resulting in a solution that minimizes a given criteria while also satisfying a set of constraints. By default, the `AnalyticLagrangianFSBetaOptimizer` class is used by IKOR (an analytic Lagrangian in full-space with beta-form constraints algorithm), but any class implementing the `LagrangianOptimizer` abstract interface could be used.

In general an optimization criteria is represented by any class implementing the `Optimizer::Objective` abstract interface and constraints are represented by classes that implement the `Optimizer::Constraints` abstract interface class. However, the `AnalyticLagrangianFSBetaOptimizer` requires criteria and constraints in a particular form, as represented by the `ReferenceOpVectorFormObjective` and `BetaFormConstraints` classes respectively (refer to the referenced articles for detail on the algorithm formulation).

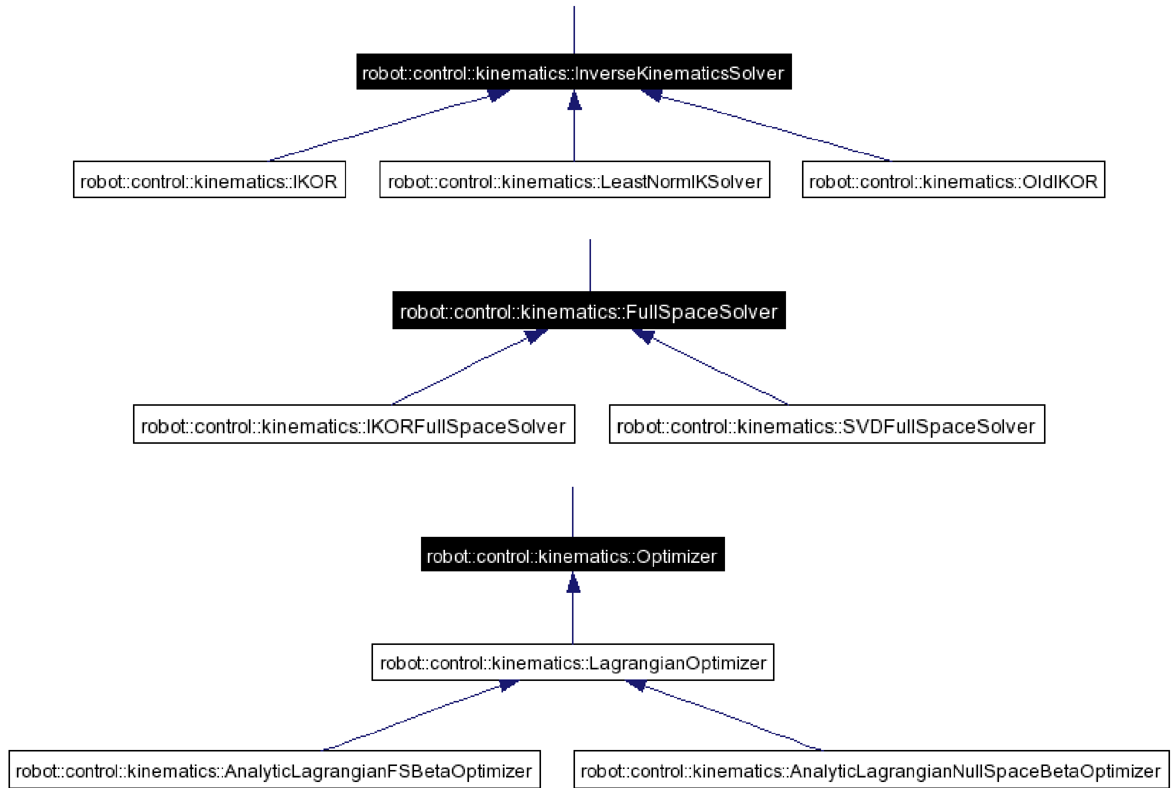

Figure 5 - Inheritance diagram of selected IK solver classes

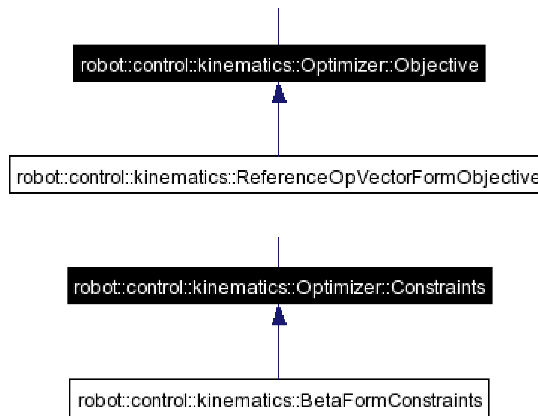

Figure 6 - Inheritance diagram of `Objective` & `Constraint` interfaces used by the `LagrangianOptimizer`

Some examples of constraints that have been formulated in beta-form for use with the FSP beta-form optimizer include joint-limits (`IKOR::JointLimitBetaConstraint`) and a constraint that applies a 'push-away' force to a specific point on a manipulator (`IKOR::PushAwayBetaConstraint`) which is used for obstacle avoidance.

3.2 Obstacle Avoidance

In order implement obstacle avoidance by manipulators during inverse kinematic control, information about the proximity of obstacles in relation to the manipulator's links is required. If using the `IKOR` solver directly, such information can be obtained via any method and supplied by a call to the `setProximitySensorData()` on each iteration. This method takes an array of `LinkProximityData` structures (defined in `InverseKinematicsSolver`), one per link in the kinematic chain. Each structure contains the distance to a detected 'object', the direction from the detection point and the distance along the link from the link's origin to the detection point (e.g. proximity sensor position, for example). The *danger distance*, *d*, is also provided, which is the distance below which a constraint becomes active to push the link away from the 'obstacle'.

If using the `IKORController` class instead, obstacle proximity information is automatically obtained from the passed `Robot` object instance. The `Robot::getControlInterface()` method is called with an interface name of "`manipulatorProximity`*N*" (where *N* is is the index of the manipulator in question) to obtain an interface to link proximity sensors[16]. For example, if performing simulation, the `SimulatedRobot` class provides this interface by computing proximity distances using the physics module collision detection infrastructure.

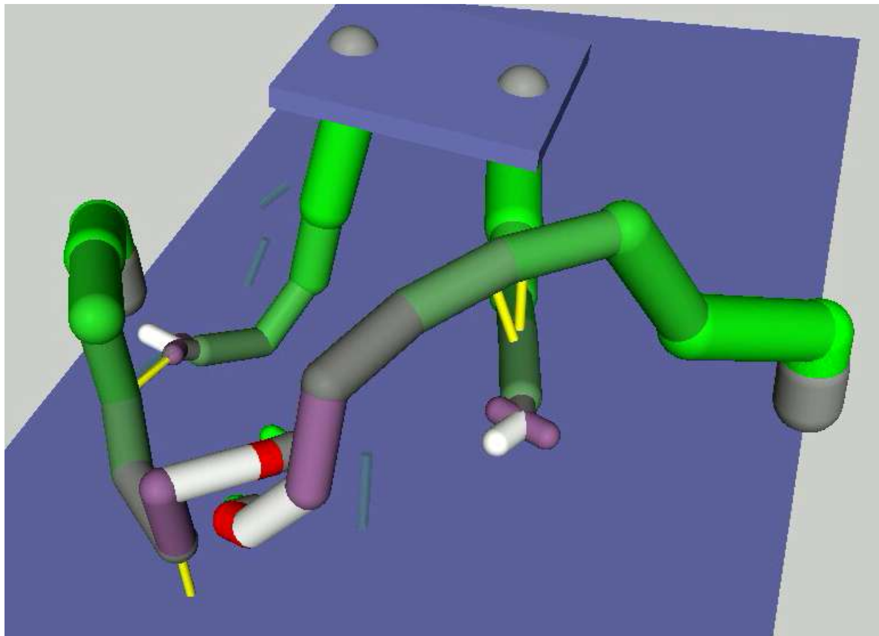

Figure 7 - Manipulators avoiding each other and the platform. Yellow lines indicate shortest vector from a link proximity 'sensor' to the detected obstacle that is within a danger distance threshold.

[16]Since typical proximity sensors (and the `SimulatedRobot` simulated sensors) cannot determine *what* is in proximity to the sensor, the `IKORController` will indiscriminately avoid anything in proximity to the manipulator links–including its own links or platform and other manipulators.

4 Reference

III Appendix

1 Build & installation

Caution
As OpenSim is still very much in development, the installation is still not polished and the instructions here are sketchy and incomplete. Please contact the developer(s) for help.

If you're not a core developer, you'll need to download the tar archive and extract it. Core developers can use *subversion*. The build tool *bjam* is included in the distribution of *boost.build*. Ensure it is in your path. Examine the `linux.env` shell script to ensure it doesn't need to be modified for your environment, then source it. Now you should just be able to type `bjam` from the command line to build the whole system (currently only GNU/Linux and gcc 3.2.x is supported). All build targets are currently placed directories separate from the source tree. You'll have to run them from there, as there are no install targets in the `Jamfiles` yet.

1.1 Tools

Please use a modern ANSI C++ compiler. For example, at least version 3.2.x of the GNU gcc compiler (http://www.gnu.org/software/gcc).

For documentation use the special comments compatible with the *doxygen* automatic document generation program (http://www.stack.nl/~dimitri/doxygen/). It is installed by default on most Linux distributions. There is a `Doxyfile` in the repository. The `Doxyfile` is setup to use *dot* to generate the graphs. You'll need to install *dot* first. Instructions are on the *doxygen* page. You'll also need to adjust the `STRIP_FROM_PATH` variable in the `Doxyfile` to match your installation.

The *boost.build* system is the build system in use (which is based on *jam* see – http://www.boost.org). It is included in the distribution.

2 Coding convention notes

2.1 Coding Style

This is a summary of some of the coding conventions used to develop OpenSim. If you extend OpenSim it would be appreciated if you adhere to these much as feasible.

- Use standard ANSI C++
  - Avoid C library where possible (i.e. no `printf`, etc.)
  - Use namespaces. Don't forget to qualify standard library symbols with std:: (some compilers also put std:: symbols in the global namespace!)
  - Use the new `#include <string>` syntax (i.e. not `string.h`)
  - Don't use C defines or macros unless the benefits are really worth it over inline functions and `const` values. Use 0 instead of `NULL` and `base::minimum()` instead of `MIN` etc. Having said that, please use the macros defined in the `base` header–`Debugln()` and friends for printing, debug output and logging (these may be re-implemented in terms of a better logging system in future).
  - Use STL containers or derive/implement STL like containers. Don't craft your own lists etc. within unrelated code using `next`, `prev` pointers and the like in classes.
- Variable and function/method identifiers should start with lowercase and capitalize the first letter of each word. Please don't use the '`_field`' convention.
- Class names should capitalize the first letter of each word, including the first. For example '`DynamicBodySystemForcer`'.
- Don't use `struct`. Use `class` and `public:` instead.
- Use the following bracing style:

```
try {
     if (condition) {
           doIt();
           doItAgain();
     }
     else {
           theOther();
           checkIt();
     }
} catch (std::exception& e) {
     handleException();
}
```

- Use exceptions exclusively for error handling – don't return special values or 0 pointers if you can avoid it. Declare which exceptions a method throws it should be caught or when performance is critical. If a performance critical method doesn't throw anything use `throw()` in the prototype.
- Use the `const` keyword everywhere that is appropriate. I know that using `const` can sometimes be a pain when interpreting compiler errors, but it is better in the long run.
- Never use the special value 0 (null) of pointers for a special meaning (except perhaps to mean 'uninitialized').

- Use the `base::ref` smart pointers where possible (section 2.2.2 of the Developer Guide documents their usage). Note the use of the `NewObj` macro from `base::MemoryTracer` as an alternative to `new`. This is optional, but will allow objects to be names and traced if the memory tracing facility it enabled (for `DEBUG` builds only). This is also the preferred way to pass pointer arguments and return pointer values.

```
void afunc()
{
  ref<MyReferecedObject> myRefdObject( NewObj MyReferncedObj() );
  myRefdObject->myMethod();
}

ref<MyReferencedObj> aclonefunc(ref<const MyReferencedObj> copyFrom)
{
  // no problem to call base::Cloneable::clone() on a const object.
  //  construct & return new smart-pointer from address of new copy
  return ref<MyReferencedObj>( &copyFrom->clone() );
}

afunc(); // call
// myRedObject has been deleted here, as the local went out of scope.

ref<MyReferencedObj> copy( aclonefunc(objToCopy) );
copy->myMethod(); // call myMethod on closed object
```

- When passing a non-smart pointer-type argument to a method/function or storing it in a member variable, consider using a reference. In function/method arguments, if the pointer should never be zero, use a reference. For member variables, if the pointer needs to be provided to the constructor, use a reference, otherwise you'll have to use a pointer. Try to always pass and return references to and from functions/methods, even if you just have to cast it to a pointer for use or storage. For example:

```
void addForcer(const DynamicBodySystemForcer& forcer)
{ forcers.push_back(&forcer); }
```

-

2.2 Files

TODO: write.

3 GNU Free Documentation License

The OpenSim manual is released under the following license.

```
        GNU Free Documentation License
          Version 1.2, November 2002

 Copyright (C) 2000,2001,2002  Free Software Foundation, Inc.
     59 Temple Place, Suite 330, Boston, MA  02111-1307  USA
 Everyone is permitted to copy and distribute verbatim copies
 of this license document, but changing it is not allowed.


0. PREAMBLE

The purpose of this License is to make a manual, textbook, or other
functional and useful document "free" in the sense of freedom: to
assure everyone the effective freedom to copy and redistribute it,
with or without modifying it, either commercially or noncommercially.
Secondarily, this License preserves for the author and publisher a way
to get credit for their work, while not being considered responsible
for modifications made by others.

This License is a kind of "copyleft", which means that derivative
works of the document must themselves be free in the same sense.  It
complements the GNU General Public License, which is a copyleft
license designed for free software.

We have designed this License in order to use it for manuals for free
software, because free software needs free documentation: a free
program should come with manuals providing the same freedoms that the
software does.  But this License is not limited to software manuals;
it can be used for any textual work, regardless of subject matter or
whether it is published as a printed book.  We recommend this License
principally for works whose purpose is instruction or reference.


1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work, in any medium, that
contains a notice placed by the copyright holder saying it can be
distributed under the terms of this License.  Such a notice grants a
world-wide, royalty-free license, unlimited in duration, to use that
work under the conditions stated herein.  The "Document", below,
refers to any such manual or work.  Any member of the public is a
licensee, and is addressed as "you".  You accept the license if you
copy, modify or distribute the work in a way requiring permission
under copyright law.

A "Modified Version" of the Document means any work containing the
Document or a portion of it, either copied verbatim, or with
modifications and/or translated into another language.

A "Secondary Section" is a named appendix or a front-matter section of
the Document that deals exclusively with the relationship of the
publishers or authors of the Document to the Document's overall subject
(or to related matters) and contains nothing that could fall directly
within that overall subject.  (Thus, if the Document is in part a
textbook of mathematics, a Secondary Section may not explain any
mathematics.)  The relationship could be a matter of historical
```

connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

A section "Entitled XYZ" means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as "Acknowledgements", "Dedications", "Endorsements", or "History".) To "Preserve the Title" of such a section when you modify the Document means that it remains a section "Entitled XYZ" according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution

```
and modification of the Modified Version to whoever possesses a copy
of it.  In addition, you must do these things in the Modified Version:

A. Use in the Title Page (and on the covers, if any) a title distinct
   from that of the Document, and from those of previous versions
   (which should, if there were any, be listed in the History section
   of the Document).  You may use the same title as a previous version
   if the original publisher of that version gives permission.
B. List on the Title Page, as authors, one or more persons or entities
   responsible for authorship of the modifications in the Modified
   Version, together with at least five of the principal authors of the
   Document (all of its principal authors, if it has fewer than five),
   unless they release you from this requirement.
C. State on the Title page the name of the publisher of the
   Modified Version, as the publisher.
D. Preserve all the copyright notices of the Document.
E. Add an appropriate copyright notice for your modifications
   adjacent to the other copyright notices.
F. Include, immediately after the copyright notices, a license notice
   giving the public permission to use the Modified Version under the
   terms of this License, in the form shown in the Addendum below.
G. Preserve in that license notice the full lists of Invariant Sections
   and required Cover Texts given in the Document's license notice.
H. Include an unaltered copy of this License.
I. Preserve the section Entitled "History", Preserve its Title, and add
   to it an item stating at least the title, year, new authors, and
   publisher of the Modified Version as given on the Title Page.  If
   there is no section Entitled "History" in the Document, create one
   stating the title, year, authors, and publisher of the Document as
   given on its Title Page, then add an item describing the Modified
   Version as stated in the previous sentence.
J. Preserve the network location, if any, given in the Document for
   public access to a Transparent copy of the Document, and likewise
   the network locations given in the Document for previous versions
   it was based on.  These may be placed in the "History" section.
   You may omit a network location for a work that was published at
   least four years before the Document itself, or if the original
   publisher of the version it refers to gives permission.
K. For any section Entitled "Acknowledgements" or "Dedications",
   Preserve the Title of the section, and preserve in the section all
   the substance and tone of each of the contributor acknowledgements
   and/or dedications given therein.
L. Preserve all the Invariant Sections of the Document,
   unaltered in their text and in their titles.  Section numbers
   or the equivalent are not considered part of the section titles.
M. Delete any section Entitled "Endorsements".  Such a section
   may not be included in the Modified Version.
N. Do not retitle any existing section to be Entitled "Endorsements"
   or to conflict in title with any Invariant Section.
O. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or
appendices that qualify as Secondary Sections and contain no material
copied from the Document, you may at your option designate some or all
of these sections as invariant.  To do this, add their titles to the
list of Invariant Sections in the Modified Version's license notice.
These titles must be distinct from any other section titles.

You may add a section Entitled "Endorsements", provided it contains
nothing but endorsements of your Modified Version by various
parties--for example, statements of peer review or that the text has
been approved by an organization as the authoritative definition of a
```

standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections Entitled "History" in the various original documents, forming one section Entitled "History"; likewise combine any sections Entitled "Acknowledgements", and any sections Entitled "Dedications". You must delete all sections Entitled "Endorsements".

6. COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an "aggregate" if the copyright resulting from the compilation is not used to limit the legal rights

of the compilation's users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

8. TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

If a section in the Document is Entitled "Acknowledgements", "Dedications", or "History", the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

9. TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided for under this License. Any other attempt to copy, modify, sublicense or distribute the Document is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

10. FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See http://www.gnu.org/copyleft/.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation.

4 Document History

Date	Author	Comment
05/03/03	David Jung	Created initial document, version 0.1 (incomplete)
12/03/04	David Jung	Additions, version 0.2 (incomplete)
19/05/04	David Jung	Additions, version 0.3 (incomplete)
29/07/04	David Jung	Beginnings of Guide section & other additions, version 0.4