EVALUATING THE OPEN PHYSICS ABSTRACTION LAYER PROGRAMMING LIBRARY FOR USE IN UNDERGRADUATE INSTRUCTION

by

Ryan Gardner

A capstone project report submitted to the faculty of

Brigham Young University

in partial fulfillment of the requirements for the degree of

Bachelor of Science

Department of Physics and Astronomy

Brigham Young University

April 2007

ABSTRACT

EVALUATING THE OPEN PHYSICS ABSTRACTION LAYER PROGRAMMING LIBRARY FOR USE IN UNDERGRADUATE INSTRUCTION

Ryan Gardner

Department of Physics and Astronomy

Bachelor of Science

The Open Physics Abstraction Layer (OPAL) is an open-source software project that provides a library of physics routines to use in other programs. In this project, the efficacy of using OPAL to perform physics simulations was examined and found lacking. Among its major deficiencies are the inability to export numerical data, and the lack of commonly-used physics elements such as springs, dampers, or linear oscillators. To address these issues, a data-logging framework was created and directions are provided to guide a future developer.

ACKNOWLEDGEMENTS

This project could not have been possible without the help of the OPAL development community, my advisor Dr. Lawrence Rees, or the capstone coordinator, Dr. Branton Campbell.

Table of Contents

List of Figures

Introduction

BACKGROUND

A quality physics demonstration can provide a student with a strong mental picture of how a given system works, and help train his intuition so he can navigate his way through difficult problems. The demonstrations used most often in teaching mechanical systems are currently "real life" demonstrations involving physical objects. With access to an easy-to-use physics simulation program, simulated 3-dimensional computer simulations could be used just as well.

Sometimes just being able to "see it" – either in real life or in a simulation - can help a student grasp a concept. A good software package for physics demonstrations would reduce the need for real life demonstrations. This would enable professors to construct demonstrations easily, and allow students to build their own demonstrations. Software demonstrations could be used in homework assignments, tests, and for independent study courses.

The ability to type a few keystrokes and change some aspect of a system also provides a way to study the effect of one variable (say, the length of a pendulum or the mass of a pulley) has on the rest of the system.

THE BRAINS BEHIND THE SYSTEM

The "brains" to such a versatile physics system is already available. A group of volunteer developers from around the world actively develop and maintain a project known as the Open Dynamics Engine (abbreviated from here on as ODE). The ODE group describes the engine as "an open source, high performance library for simulating rigid body dynamics. "

To develop a simulation using the Open Dynamics Engine, however, requires a significant amount of time and programming knowledge. Fortunately, another open-source group has spent a great deal of time building what is known as the Open Physics Abstraction Layer (referred to as OPAL). OPAL provides a set of higher-level methods and instructions to make developing a physics simulation easier. Most notably, the ability to enter the description of the physics in an XML file, and not in the code itself.

```
<?xml version="1.0" encoding="UTF-8" standalone="yes" ?>
<OpalBlueprint>
  <Solid>
    <Name value="Pendulum" />
    <Static value="false" />
    <Shape type="box">
      <Dimensions x="2" y="2" z="98" />
      <Material hardness="0.8" friction="0"
       bounciness="0.0" density="0.2" />
   </Shape>
   <Offset>
      <Transform type="translate"
       x="0" y="0" z="49" />
    </Offset>
  </Solid>
  <Joint>
    <Name value="Pendulum Hinge" />
    <Type value="universal" />
    <References solid0="null" solid1="Pendulum" />
    <Anchor x="0" y="0" z="100" />
    <Axis>
      <AxisNum value="0" />
      <Direction x="1" y="0" z="0" />
      <LimitsEnabled value="false" />
   </Axis>
    <Axis>
      <AxisNum value="1" />
      <Direction x="0" y="0" z="1" />
      <LimitsEnabled value="false" />
    </Axis>
  </Joint>
</OpalBlueprint>
```

Figure 1-1: An XML OPAL Blueprint file describing a pendulum

Not only is the XML format convenient to enter simulations initially – but changes to the parameters of the system can be made very quickly and without recompiling the simulator code. For instance, to make the pendulum 20 meters long instead of 49 meters long, would require changing two numbers in the XML file and then re-running the simulator. Were the simulation programmed into the simulator, the change process would involve recompiling and relinking.

The XML-format becomes an even more appealing way of entering physical simulations when you use a graphical editor. One OPAL user has created a visual XML editor that allow scenes to be created in a 3D environment instead of a text editor. This freely-available editor simplifies the most difficult parts of making a simulation - creating the physical elements, joining them together, and positioning them in a virtual space.

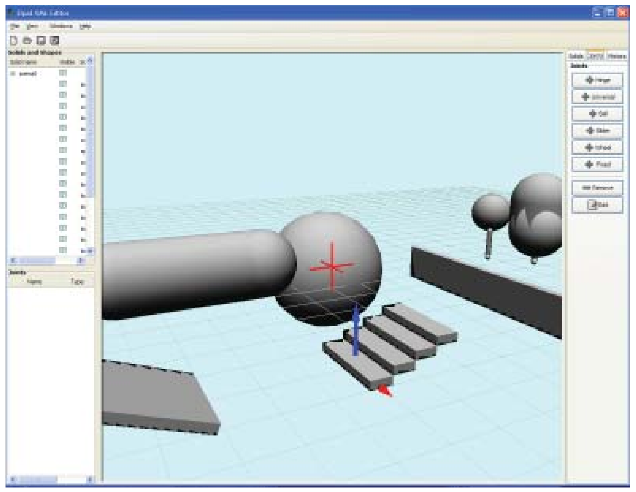

Figure 1-2: A screenshot of the scene editing program for OPAL

Deficiencies of OPAL for use in Physics Simulations

The original goal of this project was to evaluate the accuracy of the methods that OPAL uses to simulate mechanical systems. To accomplish this, the plan was to simulate several systems and compare the results of the simulated motion to the exact solution of the same system. In the early stages of creating simulations inside the OPAL environment, it became clear that OPAL is not designed for simulating the kind of mechanical systems that undergraduate physicists study. It has several areas that are currently deficient.

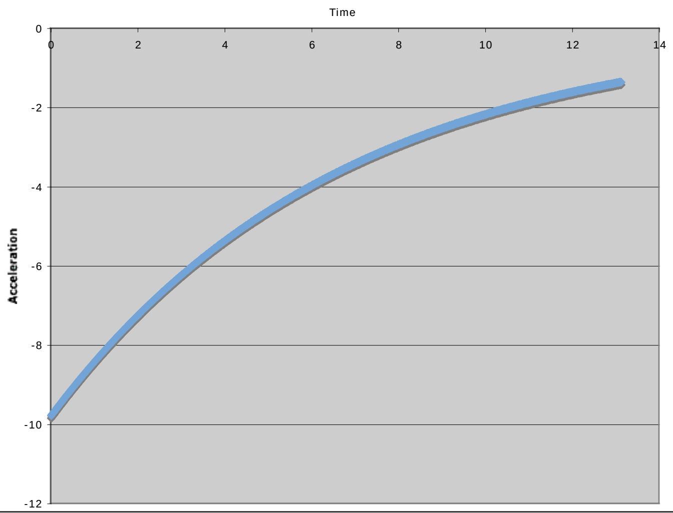

Figure 1-3: Acceleration vs. time for an object falling with linear damping unspecified (by default it is on)

1. LACK OF EASY BUILDING BLOCKS

The building-blocks of basic mechanical systems problems include springs, pulleys, pendulums, wheels, carts, oscillating motors, and inclined planes. Of these elements, only a few of them are easy to define in the XML-format (or source code). The rest of them involve constructing things that simulate these basic elements out of other existing elements inside OPAL.

2. PARADIGM REGARDING DEFAULT VALUES

OPAL also has certain default assumptions about the simulation you are creating - and some of these assumptions run counter to the thinking of a physicist. A falling object, for instance, will be acted upon by what OPAL refers to as “linear damping” - a rough simulation of air resistance. This value can be turned off, but if it is not explicitly turned off, it will cause objects to fall with non-constant acceleration.

3. NO BUILT-IN METHOD TO OUTPUT PHYSICAL DATA

Even if it were easy to construct simulations in OPAL, and those simulations behaved as expected, the simulation would be of limited use to the physicist. OPAL is mostly used to produce visual simulations of physical systems. Although a person firing a cannonball in a simulated environment may not care to plot the trajectory of that ball

- a physicist studying such a system most certainly would. As OPAL exists now, there is no easy way to extract position data for each time step. Later on, I describe how I programmatically have fixed this deficiency - but the lack of a built-in method is troubling.

4. NO METHOD TO OUTPUT JOINT-RELATED DATA

In addition to the data regarding an objects position, velocity, and other attributes - information about the joints joining these objects is important to be able to analyze the system. In the case of a pendulum, for instance, position data is much less convenient to work with than the angle of the joint. In the case of a double or triple pendulum, the position data alone would be a nightmare to analyze, whereas the joint-angle data would be very simple.

Overcoming Deficiencies

The gaps between what OPAL provides already, and what a physicist would need to easily create simulations is hardly insurmountable. The scope of this project is to bridge the gaps where possible, and provide the footings for future work to bridge other gaps.

There are two possible places to address these issues. The first place is within OPAL itself. This is the best place to address issues directly related to functionality that OPAL provides the programmer, or functions that should be handled within OPAL.

The second area to overcome the issue is within the program that relies on OPAL. This is the best place to tackle problems that are outside of the scope of what OPAL project is attempting to provide, and can be solved by calling methods already contained in OPAL.

ADDRESSING ISSUES WITHIN OPAL

The open-source nature of OPAL means two things. Its existence means that there is a group of programmers who are interested enough in having a physics abstraction layer that they have built one. It is true that everyone who wants to can contribute to the source code of OPAL project, but before new code is accepted into the project, it must be approved by one of the primary developers.

Many open-source projects operate under the mantra “patches welcome” - meaning that they are willing to accept changes to their source code to fix bugs or improve functionality, and will then review these changes. Many times, in open-source development, patches that are submitted are not accepted into the project. Certainly additions that consist of low-quality code, introduce new bugs, or do nothing useful are rejected almost immediately. When a code change, or patch, would make sweeping changes to the architecture of the current project, regardless of how well written, it will usually be rejected.

In the case of OPAL, many of the changes that would be required to make it work the way a physicist wants are the kind of changes that require more than a simple one-line fix here or there. Most of them require some significant consideration on the part of the developers as to where and how to implement the changes. To deal with these kind of changes, the best way to help the main developers is to provide detailed information describing situations in which a user may want a certain functionality, and then developing a requirements document.

Admittedly, this kind of approach relies heavily on the willingness of volunteer developers. Clarifying what is wanted, why it is wanted, and providing a compelling reason to make such a change can greatly increase the likelihood of a programmer to want to make the change. Should no programmers on OPAL project want to make these changes, it is possible that in the future another physicist will be able to start work on making these changes based on the significant effort I have already done in identifying key areas for improvement.

ADDRESSING ISSUES AT THE PROGRAM LEVEL

To address problems at the program level is relatively easy for a programmer. To do so in such a way as to also provide some assistance to future developers who may be facing a similar problem involves some additional planning.

In the case of OPAL, outputting data about the objects being simulated is best solved at the application level. OPAL provides a set of routines that can be used to extract data, and a mechanism to assign a certain function in a program to be called at the end of every time step in the simulation.

To contribute to the ease of future users who may want to output detailed logging data of their simulations, I created a rather robust data logging module that allows the programmer to specify what specific data he wishes to have output, and what file to output the data into. This data-logging module is more thoroughly described in the "Methods" section. The basic idea behind it is that anyone who uses OPAL in their program can include two files into their project, write three lines of code into their application, and have the ability to have detailed data output to a file of their choosing.

```
void OpalDemo::createPostEventListener(void) {
    std::string outputFile = "/path/to/output/file";
    int flags = 0 | LOG_POSITION \
                  | LOG_EULER \
                  | LOG_GLOBAL_ANGULAR_VELOCITY \
                  | LOG_LOCAL_LINEAR_VELOCITY \
                  | JOINT_LOG_VELOCITY  \
                  | JOINT_LOG_ANGLES \
                  | JOINT_LOG_DISTANCE;

    // This next line creates a new instance of the
    // DataLogger object with the variables set above
    // and a reference to a vector of PhysicalEntity
    // objects defined elsewhere
   mDataLogger = new DataLogger(mPhysicalEntityList,
mSimulator->getStepSize(),outputFile,flags);

    // The next line registers this data logger as
    // the post-step event handler for OPAL
    // simulator named "mSimulator"
    mSimulator->addPostStepEventHandler(mDataLogger);
}
```

Figure 2-1: Example C++ code illustrating how easily the DataLogger is accessed and used by a programmer

Methods

DATA LOGGING MODULE

To extract data from the simulator, I wrote a data logging module in C++ that fulfills the following requirements:

1. It allows the programmer to specify in a simple way which of the 8 or 9 possible data elements he wants output for each time step.

2. The code is written in a modular way that allows for easy maintenance in the future, should new data formats be added or parts of OPAL code change

3. Whenever possible, the code uses methods that are easy to maintain.

4. It exports the data in a format that is easily imported into data analysis programs.

The code is organized as follows. The programmer ties into the DataLogger class and instantiates the object. Once the DataLogger object is created, he assigns it to be the post event-step listener. The following code is all a developer needs to include in order to implement the data logger in his project:

The LOG_POSITION, LOG_EULER, LOG_LOCAL_LINEAR_VELOCITY, and other similarly named variables mentioned above are constants that are defined in a file called movementLoggerConstants.h. This file defines the flag variables to be equal to unique powers of two. Using binary logic to "OR" them together uses a single integer-worth of memory to store "on" or "off" states for the 20 different possible types of data that can be turned off or on.

When the DataLogger is instantiated, it creates an empty vector of DataLoggingModule objects and passes a reference to this vector to the DataLoggingModuleFactory, along with the flags indicating which logging modules to construct. The DataLoggingModuleFactory loops over the relevant powers of two and sees if a given flag is set. If the flag is set, it will create the corresponding DataLoggingModule for each physical object in the simulation.

After the DataLoggingFactory has created all of the logging modules, the DataLogger is finished being constructed. The next time that the DataLogger is called is at the end of the first time step - which is the zero-second time step. At this point, the DataLogger iterates over all of the DataLoggingModules stored in its vector of modules. For each module, it outputs any relevant header data, each field in the file separated by a tab.

After each subsequent time step, the DataLogger writes the current time to the file, and then iterates over all of the DataLoggingModules and calls each module's OutputData method. Each module outputs its data to the file, again separated by tabs.

The DataLogger does not need to know what kind of data needs to be output in order to print the local velocity of an object, or the current position of an object, or any of the other data elements. All of the different DataLogging modules are subclasses of the main virtual class DataLoggingModule. All of the DataLogging modules have the same "OutputData" method to return a string that gets printed to the file. Each module implements the business logic behind printing out the relevant data. At the bottom of this page are two examples of the OutputData()

```
std::string DataLoggingModuleMatrix44r::outputMa-
trix44r() {
  std::ostringstream s1;
  opal::Matrix44r dataMatrix44r = getDataMatrix44r();
  s1.precision(MOVEMENT_LOG_OUTPUT_PRECISION);

  for (int j=0;j<16;j++) {
     s1 << dataMatrix44r[j] << "\t";
  }

  return s1.str();

}
std::string DataLoggingModuleMatrix44r::OutputData() {
  return outputMatrix44r();
}
```

Figure 2-2a: The output method that prints out the values stored in a 4x4 matrix

```
std::string DataLoggingModuleVec3r::outputVec3r() {
    std::ostringstream s1;
    opal::Vec3r dataVec3r = getDataVec3r();
    s1.precision(MOVEMENT_LOG_OUTPUT_PRECISION);
    s1 << dataVec3r.x << "\t" << dataVec3r.y \
       << "\t"<< dataVec3r.z << "\t";
    return s1.str();
}

std::string DataLoggingModuleVec3r::OutputData() {
       return outputVec3r();
}
```

Figure 2-2b: The output method that generates the output used in the module responsible for printing data stored in three-dimensional vecors

methods and the functions that generate the string that is printed to the file.

The DataLogging Module that I wrote provides OPAL users with a robust way to specify what data users are interested in monitoring, and then an easy way to have the program output the data for them to analyze.

HOW OPAL ENABLES THE DATA LOGGING MODULE

OPAL has several functions that it provides which enables the creating of such a data logging module. The main object that runs the simulator is known as the "Simulator." It has two public functions:

```
Solid* opal::Simulator::getSolid( unsigned int a) const
```

and

```
unsigned int opal::Simulator::getNumSolids() const
```

By knowing the number of solids that exist in the simulation, and being able to retrieve a solid object via an index number, it is possible to extract data from all of the solid objects.

OPAL LACKS METHODS TO EXTRACT THE JOINTS

Unlike the solid objects, the joints in OPAL simulation are kept private to the simulator. There is no "getNumJoints()" method, nor is there a "getJoint(unsigned int a)" method in the current version of OPAL.

In an attempt to enable a "JointLoggingModule", I added these functions in to the Simulator class and recompiled OPAL library. I wrote the necessary code to log the data for the joints, but when I tried to use it, I quickly realized that the code involving joints inside of OPAL is not ready for that kind of data extraction.

OVER-RELIANCE ON ASSERTIONS MAKES DEALING WITH JOINTS TROUBLESOME

In programming, it is possible to write code to handle situations in which you believe the program should never enter. Consider this trivial example of a function that would add the sales tax to a subtotal to return a total:

```
int calculate total ( int salestax, int subtotal ) {
    assert(salestax > 0);
    return (salestax + subtotal);
}
```

When executing the code, if the above method is ever called with *salestax* less than or equal to zero, then the assertion will fail and the program will halt execution.

Assertions are useful in some situations, but they are very rigid in one thing - when an assertion fails, the entire program stops.

In OPAL code, the Joint objects can have data for several axes, or on only one axis depending upon how the joint is instantiated. When a joint that has data for only one axis is requested to return data on an axis that has no data, an assertion is called and the program stops. When attempting to retrieve information about a given axis being enabled, an assertion is called and the program exits. When attempting to determine if a given axis can rotate, and you check an axis number that doesn't exist, an assertion is called and the program exists.

This behavior is not necessarily bad programming. The methods in OPAL::Joint class are meant to be used internally by objects that have knowledge about the joints themselves. A more robust way of handling the error, however, is to have a certain value that is reserved for error status. For instance -32768 could be returned, and then the calling function can check for this value and realize it had specified invalid input. No currently functioning program would be affected by this kind of change, because any program now that is hitting the assertions is being immediately halted anyway.

There are 17 places in OPAL::Joint code that have assertions. To provide for the ability to easily pull data from joints, those assertions should be converted to return error code values.

Results and Discussion

In its current state, adapting OPAL to work for physics demonstrations is difficult but possible. The amount of time and effort required to do so, however, makes other kinds of demonstrations (i.e. a physical demonstration, or a simple animation in a program such as Maple) more cost-effective when preparation time is taken into consideration.

There are several areas that need improvement before a suitable mechanics simulator can be built on top of OPAL. Of these areas, the biggest one - exporting data into a format that can be analyzed computationaly - was overcome as a result of this project. After this obstacle is overcome, there are two more that are best addressed by the current development community of OPAL. These are default values and physicist friendly building blocks.

Suggested Improvements to OPAL

CREATE PHYSICIST-FRIENDLY BUILDING BLOCKS

It is possible to string some together some of the building blocks currently in OPAL to create the main building blocks that are necessary for physics simulations; however, it is not easy.

To create something that behaves like a spring, for instance, a user could specify a "Slider" joint that slides along one axis and then give that joint a "Bounciness" attribute. One might think that the "Bounciness" is just another word for "Spring constant" - but it is not. The "bounciness" setting on a joint determines the restitution force that a joint exerts once it hits it's limit.

Here is a list of building blocks that a physicist would be interested in, and what parameters matter:

1. Linear Spring

Type:	Joint
Parameters:	Spring constant
	Size when fully compressed
	Maximum stretched size
	Stretched distance
Behavior:	The linear spring would allow an object to move in one dimension with the constraint of the spring attached to it.

Setting up the spring would involve joining two objects   with a spring joint, and then setting the spring constant and the current distance that the spring was stretched.

Setting up a simple system with a mass attached to the ceiling by a spring would be an ideal test case for verifying the basic functionality of the spring.

2. Linear Dashpot

Type:	Joint
Parameters:	Stroke (amount of displacement)
	Damping coefficient
Behavior:	The linear dashpot applies a force proportional to velocity, but in the opposite direction.

A test case for the linear dashpot would be to set up a mass attached to a ceiling with both a spring and a dashpot, and observe the resulting motion. Exporting the data and analyzing it versus known exact solutions for such systems would help verify that the calculations in creating the dashpot were correct

3. Linear Oscillator

Type:	Motor
Parameters:	Frequency
	Amplitude
	Oscillator Pattern (Sine, Cosine, Square, Sawtooth)
	Current Displacement
Behavior:	The linear oscillator would attach to an object and would apply a certain displacement to it depending upon the time. The linear oscillator acts irrespective of the mass of the other object, and will move the object into the position dictated by the frequency and current time step.

A test case for this would be a linear oscillator attached to a block moving on a frictionless floor. Exporting the position data, it should plot exactly like a sine wave position data for the attached object should exactly match the driving function of the linear oscillator

CREATE PHYSICIST-FRIENDLY DEFAULT SETTINGS

Currently, the default settings are stored in a the namespace of OPAL::defaults. Below are the definitions that should be included in an OPAL::defaults that is "physicist friendly"

For solids:

```
const bool     enabled = true
const bool     sleeping = true
const real     sleepiness = (real)0.5
const bool     isStatic = false
const real     linearDamping = (real)0
const real     angularDamping = (real)0
```

For the ODE simulator:

```
const real     autoDisableLinearMin = 0
const real     autoDisableLinearMax = (real)0.2
const real     autoDisableAngularMin = 0
const real     autoDisableAngularMax = (real)0.2
const int      autoDisableStepsMin = (int)1
const int      autoDisableStepsMax = (int)60
const real     autoDisableTimeMin = 0
const real     autoDisableTimeMax = (real)0.4
const real     minMassRatio = (real)0.001
const real     minERP = (real)0.1
const real     maxERP = (real)0.9
const real     globalCFM = (real)1e-5
const real     jointFudgeFactor = (real)0.1
const real     maxFriction = (real)1000.0
const real     surfaceLayer = (real)0.000
const int      maxRaycastContacts = 10
```

With those relatively minor changes made to OPAL itself, it would be possible for a physicist to create a program that would read in OPAL blueprint XML files and then run physics simulations. Using other existing 3D graphics libraries, such as Ogre 3D, it would be easy to display 3D animations of physical systems being run in real time.

References

The source code for OPAL is freely available from the official OPAL website. In addition, documentation that refers to specific methods within this code was indespensible in preparing this report.

OPAL Source Code Documentation

Various Authors, (May 2006), Generated Source Code Documentation for OPAL 0.4.0, retrieved from SourceForge hosted website.
http://opal.sourceforge.net/api/0.4.0/index.html

Various Authors, (May 2006), OPAL Documentation Wiki, retrieved from the OPAL wiki hosted at Louisville University.
http://ox.slug.louisville.edu/~o0lozi01/opal_wiki/index.php/Main_Page

OPAL XML Editor

Marzena Gasidło, (February 2005), Ocelot's Jungle: OpalXMLEditor, retrieved from Marzena Gasidlo's website.
http://www.ocelotsjungle.republika.pl/index.html

```
/*
 *  movementLoggerConstants.h
 *
 *  Created by Ryan Gardner on 11/4/06.
 *
 */

#ifndef MOVEMENT_LOGGER_CONSTANTS_H
#define MOVEMENT_LOGGER_CONSTANTS_H


const int MOVEMENT_LOG_OUTPUT_PRECISION = 9; // number of significant figures to output
////////////////////////////
// Log Constants - the following bit flags represent certain constants that can be logged using the MovementLogger
///////////////////////////

// Logs the position of this Solid in global coordinates. [ 2 ^ 0 ]
const int LOG_POSITION = 0x1;

// Logs the euler angles of the Solid's orientation.  [ 2 ^ 1 ]
const int LOG_EULER = 0x2;

// Logs the Solid's inertia tensor as a 4x4 matrix. This will be the identity matrix if the Solid is static. [
2 ^ 2 ]
const int LOG_INERTIA_TENSOR = 0x4;

// Logs the Solid's angular velocity in global coordinates  [ 2 ^ 3 ]
const int LOG_GLOBAL_ANGULAR_VELOCITY = 0x8;

// Logs the Solid's angular velocity in local coordinates.  [ 2 ^ 4 ]
const int LOG_LOCAL_ANGULAR_VELOCITY = 0x10;

// Logs the Solid's linear velocity in local coordinates.   [ 2 ^ 5 ]
const int LOG_LOCAL_LINEAR_VELOCITY = 0x20;

// Logs the Solid's linear velocity in global coordinates.  [ 2 ^ 6 ]
const int LOG_GLOBAL_LINEAR_VELOCITY = 0x40;

// a quaternion representing the Solid's orientation.   [ 2 ^ 7 ]
const int LOG_QUATERNION_DATA = 0x80;

// The Solid's transform relative to the global origin.   [ 2 ^ 8 ]
const int LOG_SOLID_TRANSFORM = 0x100;

// This will force the movement logger to output data for static solids as well  [ 2 ^ 9 ]
const int LOG_STATIC_SOLIDS = 0x200;

// [ 2 ^ 10 ]
const int LOG_LINEAR_DAMPING = 0x400;

// [ 2 ^ 11 ]
const int LOG_ANGULAR_DAMPING = 0x800;

//  Reserved [ 2 ^ 12 ]
// = 0x1000;

//// Joints related constants


// Joint accumulated damage [ 2 ^ 13 ]
const int JOINT_LOG_ACCUMULATED_DAMAGE = 0x2000;
// Joint anchor position [ 2 ^ 14 ]
const int JOINT_LOG_ANCHOR_POS = 0x4000;
// Joint distance [ 2 ^ 15 ]
const int JOINT_LOG_DISTANCE = 0x8000;
```

```
// joint velocities [ 2 ^ 16 ]
const int JOINT_LOG_VELOCITY = 0x10000;
// joint angles [ 2 ^ 17 ]
const int JOINT_LOG_ANGLES = 0x20000;
// joint anchor [ 2 ^ 18 ]
const int JOINT_LOG_ANCHOR = 0x40000;
// joint lower limit [ 2 ^ 19 ]
const int JOINT_LOG_LOW_LIMIT = 0x80000;
// joint higher limit [ 2 ^ 20 ]
const int JOINT_LOG_HIGH_LIMIT = 0x100000;

// the highest power of 2 stored in a movement constant flag
// - used for iterators over the constants
const int K_LOWEST_SOLID_EXPONENT = 0;
const int K_HIGHEST_SOLID_EXPONENT = 11;
const int K_LOWEST_JOINT_EXPONENT = 12;
const int K_HIGHEST_JOINT_EXPONENT = 20;
const int K_HIGHEST_CONST_EXPONENT = 20;

// ints describing the types of data.
const int K_VEC3R = 0;
const int K_POINT3R = 1;
const int K_MATRIX44R = 2;
const int K_REAL = 3;
const int K_UNIMPLEMENTED = 4;
const int K_CONSTANT = 5;


struct movementConstant {
    std::string MeasurementName;
    int typeOfData;
    movementConstant (std::string Name, int type) : MeasurementName(Name), typeOfData(type) {}
};

const movementConstant DATA_TYPES[21] = {
/*  0 */    movementConstant("Position", K_POINT3R),
/*  1 */    movementConstant("Euler Angle", K_VEC3R),
/*  2 */    movementConstant("Inertia Tensor", K_MATRIX44R),
/*  3 */    movementConstant("Global Angular Velocity", K_VEC3R),
/*  4 */    movementConstant("Local Angular Velocity", K_VEC3R),
/*  5 */    movementConstant("Local Linear Velocity", K_VEC3R),
/*  6 */    movementConstant("Global Linear Velocity", K_VEC3R),
/*  7 */    movementConstant("Quaternion Data", K_UNIMPLEMENTED),
/*  8 */    movementConstant("Solid Transform", K_MATRIX44R),
/*  9 */    movementConstant("Static Solids", K_CONSTANT),
/* 10 */    movementConstant("Linear Damping", K_REAL),
/* 11 */    movementConstant("Angular Damping", K_REAL),
/* 12 */    movementConstant("Reserved For Future Use", K_UNIMPLEMENTED),
/* 13 */    movementConstant("Accumulated Damage", K_REAL),
/* 14 */    movementConstant("Anchor Position", K_POINT3R),
/* 15 */    movementConstant("Distances", K_REAL),
/* 16 */    movementConstant("Velocities", K_REAL),
/* 17 */    movementConstant("Angles", K_REAL),
/* 18 */    movementConstant("Anchor", K_REAL),
/* 19 */    movementConstant("Low Limits", K_REAL),
/* 20 */    movementConstant("High Limits", K_REAL) };

#endif
```

```
/*
 *  DataLoggingModule.h
 *
 *  Created by Ryan Gardner on 11/11/06.
 *
 */
#ifndef DATALOGGING_MODULE_H
#define DATALOGGING_MODULE_H

#include <opal/Logger.h>
#include <opal/PostStepEventHandler.h>
#include <opal/Solid.h>
#include <opal/Point3r.h>
#include <opal/OpalMath.h>
#include <opal/Vec3r.h>
#include “PhysicalEntity.h”
#include <sstream>
#include “MovementLogger.h”
#include “movementLoggerConstants.h”

class DataLoggingModule {
       protected:
              int type; // the bit that represents this things type
              std::string moduleName;
       public:
              virtual std::string HeaderOutput()=0;
              virtual std::string OutputData()=0;
              virtual ~DataLoggingModule();
};


class DataLoggingFactory {
       protected:
               // the flags that represent which solid objects to make - these are stored for debugging purpos-
es
               int solid_flagz;
               // the flags that represent which joint objects to make - same as above. for dev only.
               int joint_flagz;

         // methods to add a module, or multiple modules to be tracked
         void AddModules(int sflags, int jflags);
         void AddSolidModules(int sflags);
         void AddJointModules(int jflags);

         // A list of physical entities. This is needed for it to create the modules.
        std::vector<PhysicalEntity*>& mPhysicalEntityList; //  reference to a list of physical entities...

        std::vector<opal::Joint*> mJointList; // a vector of joints

    public:
        // a list of physical entities...
        std::vector<DataLoggingModule*> mModuleList;

              DataLoggingFactory(int sflags, int jflags, std::vector<PhysicalEntity*>& physList,  std::
vector<opal::Joint*> jointList):
               solid_flagz(sflags),
               joint_flagz(jflags),
               mPhysicalEntityList(physList),
               mJointList(jointList)
        {
            AddModules(sflags, jflags);
        };
              // returns a pointer to our internal list
              std::vector<DataLoggingModule*>* getDataLoggignModules() const;
};
```

```
//
// OPAL has a class “PostStepEventHandler” that provides an interface to handle
// events after everytime step. In this case, we are going to be writing out data
// at every time step.
//
class DataLogger : public opal::PostStepEventHandler {
    protected:
        DataLoggingFactory mDataLoggingModules;

        // track the current time in the simulator
        pal::real mCurrentTime;
         // stores the previous step size
         opal::real mLastStepSize;
       // store the size of the timeStep here
         opal::real mStepSize;

    public:
        DataLogger(std::vector<PhysicalEntity*>& physList, std::vector<opal::Joint*> jointList, opal::real 
StepSize, const std::string outputFile, const int sFlags, const int jFlags);

        void OPAL_CALL handlePostStepEvent();

       void setStepSize(opal::real newStepSize);

    private:
        inline void updateCurrentTime(void);
               void outputHeaderData(void);
        void outputData(void);

        // We use the loggerImpl to go along with the standard way of doing things in OPAL package
               opal::loggerImpl::Logger mLogStream;
};

// The following classes all provide functionality to print out a different kind of data
// Which class is created for a given data element is determined by an entry in the movementLoggerConstants.h
// file that links the
//
class DataLoggingModuleVec3r : public DataLoggingModule {
       protected:
               std::string moduleName; // tracks the name of this module
        opal::Solid* dataSource; // the source of the data
        opal::Joint* jointSource; // the source of the joint data
        int vec3rToLog; // a flag indicating which vec3r this module logs
        opal::Vec3r getDataVec3r(); // a method that returns the vec3r for this module
      public:
               DataLoggingModuleVec3r(std::string name, opal::Solid* dataSrc, opal::Joint* jointSrc, int 
vec3rToLog);
               std::string HeaderOutput();
               std::string OutputData();
               std::string outputVec3r();
};

class DataLoggingModulePoint3r : public DataLoggingModule {
       protected:
               std::string moduleName; // tracks the name of this module
        opal::Solid* dataSource; // the source of the data
        opal::Joint* jointSource; // the source of the joint data
        int point3rToLog; // a flag indicating which vec3r this module logs
        opal::Point3r getDataPoint3r(); // a method that returns the vec3r for this module
      public:
               DataLoggingModulePoint3r(std::string name, opal::Solid* dataSrc, opal::Joint* jointSrc, int 
vec3rToLog);
               std::string HeaderOutput();
               std::string OutputData();
               std::string outputPoint3r();
};

class DataLoggingModuleMatrix44r : public DataLoggingModule {
       protected:
               std::string moduleName; // tracks the name of this module
        opal::Solid* dataSource; // the source of the data
        opal::Joint* jointSource; // the source of the joint data
        int matrix44rToLog; // a flag indicating which Matrix44r this module logs
        opal::Matrix44r getDataMatrix44r(); // a method that returns the Matrix44r for this module
      public:
               DataLoggingModuleMatrix44r(std::string name, opal::Solid* dataSrc, opal::Joint* jointSrc, int 
Matrix44rToLog);
               std::string HeaderOutput();
               std::string OutputData();
               std::string outputMatrix44r();
};

class DataLoggingModuleReal : public DataLoggingModule {
       protected:
               std::string moduleName; // tracks the name of this module
               opal::Solid* dataSource; // the source of the data
               opal::Joint* jointSource; // the source of the joint data
               int realToLog; // a flag indicating which real this module logs
               opal::real getDataReal(); // a method that returns the Real for this module
               void getMultiAxisDataReal(opal::real* rval); // a method that returns the Real for this module
               bool multiAxisReporting; // a bool value that determines if this will output multiple-axis data
      public:
               DataLoggingModuleReal(std::string name, opal::Solid* dataSrc, opal::Joint* jointSrc, int realToLog);
               std::string HeaderOutput();
               std::string OutputData();
               std::string outputReal();
};

// The following method is commented out because it was not needed, and not
// therefore fully implemented.
//class DataLoggingModuleQuaternion : public DataLoggingModule {
//
//     std::string outputQuaternion(opal::Quaternion &toPrint) {
//             std::ostringstream s1;
//             s1.precision(MOVEMENT_LOG_OUTPUT_PRECISION);
//             s1 << toPrint.length() << “\t”;
//             s1 << toPrint.getRoll() << “\t”;
//             s1 << toPrint.getYaw() << “\t”;
//             s1 << toPrint.getPitch() << “\t”;
//
//             return s1;
//     }
//
//};



#endif
```

```
/*
 *  DataLoggingModule.cpp
 *
 *  Created by Ryan Gardner on 11/11/06.
 *
 */

#include “DataLoggingModule.h”


// When the DataLogger object is created, it creates two output streams - a main one
// that the simulation data will be written to, and a second one that is for debugging data.
// Unless there is an error, the debugging data stream has nothing written to it, and nothing
// gets written to the disk.
DataLogger::DataLogger(std::vector<PhysicalEntity*>& physList, std::vector<opal::Joint*> jointList, opal::real
StepSize, std::string outputFile, int sflags, int jflags) :
                                     mStepSize(StepSize),
                                     mLastStepSize(StepSize),
                                     mCurrentTime(0),
                    mDataLoggingModules(sflags, jflags, physList, jointList)
{

    // set up a file for the debug.
    std::string debugFileStr = outputFile + “debug”;

    std::ofstream* outFile;
    outFile = new std::ofstream(outputFile.c_str(), std::ios::app);

    mLogStream = opal::loggerImpl::Logger();
    mLogStream.setStream(“movementLog”, outFile, “”);
}


// This method provides the basic functionality of outputting the header data to a named stream
// The header row in the file will identify what is in that column.
// To output the data, it iterates over the list of data logging modules, and calls each
// module’s OutputHeader method
void DataLogger::outputHeaderData() {
    mLogStream.stream(“movementLog”) << “Time\t”;

    for( std::vector<DataLoggingModule*>::iterator dM = mDataLoggingModules.mModuleList.begin();
         dM != mDataLoggingModules.mModuleList.end(); dM++ )
    {
        DataLoggingModule* dataModule = *dM;
        mLogStream.stream(“movementLog”) << (dataModule->HeaderOutput());
    }

    mLogStream.stream(“movementLog”) << “\n”;
}

// This method provides the basic functionality of outputting the data to a named stream
// To output the data, it iterates over the list of data logging modules, and calls each
// module’s OutputData method
void DataLogger::outputData() {
    mLogStream.stream(“movementLog”) << mCurrentTime << “\t”;

    for( std::vector<DataLoggingModule*>::iterator dM =  mDataLoggingModules.mModuleList.begin();
         dM != mDataLoggingModules.mModuleList.end(); dM++ )
    {
        DataLoggingModule* dataModule = *dM;
        mLogStream.stream(“movementLog”) << (dataModule->OutputData());
    }

    mLogStream.stream(“movementLog”) << “\n”;
}
```

```
// This is the method that is called at the end of each time step in the simulator
// When this method is called, it first checks to see if it is the first time through the loop
// - and if so, it will output the header data.
// If it isn't, then it will update the current time, and then output the data.
void OPAL_CALL DataLogger::handlePostStepEvent() {
    if (mCurrentTime == 0 || mCurrentTime < StepSize) {
        outputHeaderData();
    }
    updateCurrentTime();
    outputData();
}

// Some simulations may wish to adjust the step size, but doing so will mean that
// the simulator data will not have uniform timesteps in the output file. For this reason,
// an informational note is written to the debug stream to help the person analyzing the data
// understand what changed and why.
void DataLogger::setStepSize(opal::real newStepSize) {
       if (newStepSize != mStepSize) {
              mLogStream.stream("movementDebug") << "Step size change. Was:" << mLastStepSize << " is now: "
<< newStepSize << "\n";
       }
       mLastStepSize = mStepSize;
       mStepSize = newStepSize;
}

// This method simply updates the stored current time with a forward-incremented timestep
inline void DataLogger::updateCurrentTime(void) {
       mCurrentTime += mStepSize;
}

// This method will create modules to log all of the data set in the sflags and jflags integers
// At the moment, joint modules do not work due to some problems in how OPAL outputs and access
// joint data being unfriendly to data extraction. When OPAL is updated to allow for more friendly
// access to joint-data from outside programs, this line can be uncommented and joint data will
// be output. (Some testing of this code may be necessary at that point, of course)
void DataLoggingFactory::AddModules(int sflags, int jflags) {
    AddSolidModules(sflags);
    // AddJointModules(jflags);
}

// This method will iterate over the vector of PhysicalEntity objects and will use the bits set in the
// sflags to determine what kind of data logging is wanted.
//
// If the flag is set to log a certain type of data, this method will then perform a basic lookup against
// data stored in the movementConstants.h file to determien what kind of logging module it should create.
//
// If the physical entity object were modified to store a local integer containing solid flags, and
// a method to access this information was provided - this method could be changed to allow for object-level
// data logging relatively easily.
//
//
void DataLoggingFactory::AddSolidModules(int sflags) {

    // step over the solid modules first
    for( std::vector<PhysicalEntity*>::iterator pE = mPhysicalEntityList.begin(); pE != mPhysicalEntityList.
end(); pE++ ) {
        PhysicalEntity* tempEntity = *pE;

        // if the physical entity is static (meaning, it is set to never move),
        // by default, we don't care about tracking its motion. The LOG_STATIC_SOLIDS flag can be set
        // - which will enable the logging of the static data
        if (tempEntity->getStatic() && !(outputFlags & LOG_STATIC_SOLIDS == LOG_STATIC_SOLIDS)) {
          continue;
        }
        std::string physEntityName = tempEntity->getName();
        opal::Solid* tSolid = tempEntity->getSolid();

        // First we add the modules that correspond to the Solids-related logging
        for ( int k=K_LOWEST_SOLID_EXPONENT; k <= K_HIGHEST_SOLID_EXPONENT; k++) {
            // test to see if the given flag is set.
            int curExponent = (1 << k);
            if ( ((curExponent) & sflags) == (curExponent) ) {
                std::ostringstream entityModuleName;
                entityModuleName << physEntityName << " " << DATA_TYPES[k].MeasurementName;
                // if the flag is set, check to see what kind of flag it is
                DataLoggingModule* newModule;

                switch (DATA_TYPES[k].typeOfData) {
                    case K_VEC3R:
                        newModule = new DataLoggingModuleVec3r(entityModuleName.str(), tSolid, 0, curExponent);
                        mModuleList.push_back(newModule);
                        break;
                    case K_POINT3R:
                        newModule = new DataLoggingModulePoint3r(entityModuleName.str(), tSolid, 0, curExponent);
                        mModuleList.push_back(newModule);
                        break;
                    case K_MATRIX44R:
                        newModule = new DataLoggingModuleMatrix44r(entityModuleName.str(), tSolid, 0, curExponent);
                        mModuleList.push_back(newModule);
                        break;
                    case K_REAL:
                        // implement
                        break;
                    case K_UNIMPLEMENTED:
                        // message
                        break;
                    case K_CONSTANT:
                        // implement
                        break;
                }
            }
        }
    }
}


// This method performs an almost identical function to the AddSolidModules funtion above. Joints, however
// currently do not reply to this kind of data extraction very politely. First of all, OPAL doesn't currently
// have a getter function to pull the vector of Joints out of the simulator data. Adding this functionality to
// OPAL involves adding three or four lines of code to one of the files describign the simulator - but once the
// joint data is available, and an attempt is made to pull data out of a joint - it becomes immediately apparent
// that it was not developed with this in mind.
//
//
void DataLoggingFactory::AddJointModules(int jflags) {
    for( std::vector<opal::Joint*>::iterator jE = mJointList.begin(); jE != mJointList.end(); ++jE ) {
        opal::Joint* tempJoint = *jE;

        if (tempJoint == 0) {
            break;
        }
        std::string jointName = tempJoint->getName();

        for (int j=K_LOWEST_JOINT_EXPONENT; j <= K_HIGHEST_JOINT_EXPONENT; j++) {
            int curExponent = (1 << j);
            // if a given flag is set
            if ( ((curExponent) & jflags) == (curExponent) ) {
                std::ostringstream entityModuleName;
                entityModuleName << jointName << " " << DATA_TYPES[j].MeasurementName;
                DataLoggingModule* newModule;

                switch (DATA_TYPES[j].typeOfData) {
                    case K_VEC3R:
                        newModule = new DataLoggingModuleVec3r(entityModuleName.str(), 0, tempJoint, curExpo-
nent);
                        mModuleList.push_back(newModule);
                        break;
                    case K_POINT3R:
                        newModule = new DataLoggingModulePoint3r(entityModuleName.str(), 0, tempJoint, curExpo-
nent);
                        mModuleList.push_back(newModule);
                        break;
                    case K_MATRIX44R:
                        newModule = new DataLoggingModuleMatrix44r(entityModuleName.str(), 0, tempJoint, curEx-
ponent);
                        mModuleList.push_back(newModule);
                        break;
                    case K_REAL:
                        newModule = new DataLoggingModuleReal(entityModuleName.str(), 0, tempJoint, curExpo-
nent);
                        mModuleList.push_back(newModule);
                        break;
                    case K_UNIMPLEMENTED:
                        // message
                        break;
                    case K_CONSTANT:
                        // implement
                        break;
                }

            }

        }

    }
}

// This is the deconstructor function.
DataLoggingModule::~DataLoggingModule() {
}

// The Vect3r constructor handles storng the name, a pointer to a solid to pull data from, and an integer rep-
resenting
// which Vec3r to pull out of the given solid module
DataLoggingModuleVec3r::DataLoggingModuleVec3r(std::string name, opal::Solid* dSource, opal::Joint* jSource,
int v3rToLog):
    moduleName(name),
    dataSource(dSource),
    jointSource(jSource),
    vec3rToLog(v3rToLog) {
    // All of the member variables are assigned in the calling of this constructor
    // so there is nothing left to do in this method.
}

//
//
//
opal::Vec3r DataLoggingModuleVec3r::getDataVec3r() {
    opal::Vec3r returnVal;
    switch (vec3rToLog) {

        case LOG_EULER:
            returnVal = dataSource->getEulerXYZ();
            break;

        case LOG_LOCAL_LINEAR_VELOCITY:
            returnVal = dataSource->getLocalAngularVel();
            break;

        case LOG_GLOBAL_LINEAR_VELOCITY:
            returnVal = dataSource->getGlobalLinearVel();
            break;

        case LOG_GLOBAL_ANGULAR_VELOCITY:
            returnVal = dataSource->getGlobalAngularVel();
            break;

        case LOG_LOCAL_ANGULAR_VELOCITY:
            returnVal = dataSource->getLocalAngularVel();
            break;

        default: // this is just in here while I'm testing
            returnVal = dataSource->getGlobalLinearVel();
            break;
    }
    return returnVal;
}

std::string DataLoggingModuleVec3r::outputVec3r() {
       std::ostringstream s1;
    opal::Vec3r dataVec3r = getDataVec3r();
       s1.precision(MOVEMENT_LOG_OUTPUT_PRECISION);
       s1 << dataVec3r.x << "\t" << dataVec3r.y << "\t" << dataVec3r.z << "\t";
       return s1.str();
}

std::string DataLoggingModuleVec3r::HeaderOutput() {
    std::ostringstream head;
    head << moduleName << " X" << "\t" << moduleName << " Y" << "\t" << moduleName << " Z" << "\t";
    return head.str();
}

std::string DataLoggingModuleVec3r::OutputData() {
       return outputVec3r();
}

/***************************/
// Point3r output module

DataLoggingModulePoint3r::DataLoggingModulePoint3r(std::string name, opal::Solid* dSource, opal::Joint*
jSource, int p3rtoLog): moduleName(name), dataSource(dSource), jointSource(jSource),point3rToLog(p3rtoLog) {
    std::cout << "the class didn't puke when creating the vec3r\n";
}

opal::Point3r DataLoggingModulePoint3r::getDataPoint3r() {
    opal::Point3r returnVal;
    switch (point3rToLog) {

        case LOG_POSITION:
            returnVal = dataSource->getPosition();
            break;

     /*   case JOINT_LOG_ANCHOR_POS:
            returnVal = dataSource->getLocalAngularVel();
            break;
     */
        default: // this is just in here while I'm testing
            returnVal = dataSource->getPosition();
            break;
    }
    return returnVal;
}

std::string DataLoggingModulePoint3r::outputPoint3r() {
        std::ostringstream s1;
    opal::Point3r dataPoint3r = getDataPoint3r();
        s1.precision(MOVEMENT_LOG_OUTPUT_PRECISION);
        s1 << dataPoint3r.x << "\t" << dataPoint3r.y << "\t" << dataPoint3r.z << "\t";
        return s1.str();
}

std::string DataLoggingModulePoint3r::HeaderOutput() {
    std::ostringstream head;
    head << moduleName << " X" << "\t" << moduleName << " Y" << "\t" << moduleName << " Z" << "\t";
    return head.str();
}

std::string DataLoggingModulePoint3r::OutputData() {
        return outputPoint3r();
}

/********************************/
// Matrix44r output module

DataLoggingModuleMatrix44r::DataLoggingModuleMatrix44r(std::string name, opal::Solid* dSource, opal::Joint*
jSource, int m44rToLog): moduleName(name), dataSource(dSource), jointSource(jSource),matrix44rToLog(m44rToLog)
{
    std::cout << "the class didn't throw errors when creating the Matrix44r\n";
}

opal::Matrix44r DataLoggingModuleMatrix44r::getDataMatrix44r() {
    opal::Matrix44r returnVal;
    switch (matrix44rToLog) {

        case LOG_INERTIA_TENSOR:
            returnVal = dataSource->getInertiaTensor();
            break;

        default: // this is just in here while I'm testing
            returnVal = dataSource->getInertiaTensor();
            break;
    }
    return returnVal;
}

std::string DataLoggingModuleMatrix44r::outputMatrix44r() {
        std::ostringstream s1;
    opal::Matrix44r dataMatrix44r = getDataMatrix44r();
        s1.precision(MOVEMENT_LOG_OUTPUT_PRECISION);

    for (int j=0;j<16;j++) {
            s1 << dataMatrix44r[j] << "\t";
        }

    return s1.str();

}

std::string DataLoggingModuleMatrix44r::HeaderOutput() {
    std::ostringstream head;

    for (int i=0;i<4;i++) {
        for (int j=0;j<4;j++) {
            head << moduleName << “Solid Transform “ << i << j << “\t”;
        }
    }

    return head.str();
}

std::string DataLoggingModuleMatrix44r::OutputData() {
       return outputMatrix44r();
}

/**********************************************/
// Real logging module

DataLoggingModuleReal::DataLoggingModuleReal(std::string name, opal::Solid* dSource, opal::Joint* jSource, int
rToLog): moduleName(name), dataSource(dSource), jointSource(jSource), realToLog(rToLog) {
    // all binary for joints with flags greater than 2^13 that are reals have multiple-axis reporting for them.
    multiAxisReporting = (rToLog > (1 << 13)) ? true : false;
    std::cout << “the class didn’t puke when creating the real\n”;
}

opal::real DataLoggingModuleReal::getDataReal() {
    opal::real returnVal;
    switch (realToLog) {

        case LOG_LINEAR_DAMPING:
            returnVal = dataSource->getLinearDamping();
            break;

        case LOG_ANGULAR_DAMPING:
            returnVal = dataSource->getAngularDamping();
            break;

        default: // this is just in here while I’m testing
            returnVal = dataSource->getLinearDamping();
            break;
    }
    return returnVal;
}

void DataLoggingModuleReal::getMultiAxisDataReal(opal::real* returnVal) {
    switch (realToLog) {
        case JOINT_LOG_DISTANCE:
            for (int i=0;i<3;i++)
                returnVal[i] = jointSource->getDistance(i);
            break;

        case JOINT_LOG_VELOCITY:
            for (int i=0;i<3;i++)
                returnVal[i] = jointSource->getVelocity(i);
            break;

        case JOINT_LOG_LOW_LIMIT:
            for (int i=0;i<3;i++)
                returnVal[i] = jointSource->getLowLimit(i);
            break;

        case JOINT_LOG_HIGH_LIMIT:
            for (int i=0;i<3;i++)
                returnVal[i] = jointSource->getHighLimit(i);
            break;

        default: // this is just in here while I’m testing
            for (int i=0;i<3;i++)
                returnVal[i] = jointSource->getHighLimit(i);
            break;
    }
}

std::string DataLoggingModuleReal::outputReal() {
        std::ostringstream s1;
    s1.precision(MOVEMENT_LOG_OUTPUT_PRECISION);

    if (multiAxisReporting) {
        opal::real dataReal[3];
        getMultiAxisDataReal(dataReal);

        s1 << dataReal[0] << "\t" << dataReal[1] << "\t" << dataReal[2] << "\t";
    }
    else {
        opal::real dataReal = getDataReal();
        s1 << dataReal;
    }

        return s1.str();
}

std::string DataLoggingModuleReal::HeaderOutput() {
    std::ostringstream head;

    if (multiAxisReporting) {
        head << moduleName << " X" << "\t" << moduleName << " Y" << "\t" << moduleName << " Z" << "\t";
    }
    else {
        head << moduleName;
    }

    return head.str();
}

std::string DataLoggingModuleReal::OutputData() {
        return outputReal();
}
```