



LIGO Laboratory / LIGO Scientific Collaboration

LIGO- T020205-v2

ADVANCED LIGO

28 April 2014

Models of the
Advanced LIGO Suspensions
in Mathematica™

Mark Barton

Distribution of this document:
DCC

This is an internal working note
of the LIGO Project.

California Institute of Technology
LIGO Project – MS 18-34
1200 E. California Blvd.
Pasadena, CA 91125
Phone (626) 395-2129
Fax (626) 304-9834
E-mail: info@ligo.caltech.edu

Massachusetts Institute of Technology
LIGO Project – NW17-161
175 Albany St
Cambridge, MA 02139
Phone (617) 253-4824
Fax (617) 253-7014
E-mail: info@ligo.mit.edu

LIGO Hanford Observatory
P.O. Box 1970
Mail Stop S9-02
Richland WA 99352
Phone 509-372-8106
Fax 509-372-8137

LIGO Livingston Observatory
P.O. Box 940
Livingston, LA 70754
Phone 225-686-3100
Fax 225-686-7189

<http://www.ligo.caltech.edu/>

Table of Contents

1	<i>Introduction</i>	5
1.1	Purpose and Scope	5
1.2	References	5
1.3	Version history	5
2	<i>Overview</i>	5
3	<i>Theory</i>	6
3.1	Method of calculation	6
3.1.1	Basic Normal Mode Calculation.....	6
3.1.2	Categories of Coordinates and the Master Potential Matrix.....	7
3.1.3	The Effective Potential Matrix	8
3.1.4	The Equation Of Motion Matrix and the Coupling Matrix	8
3.1.5	The Coupling Matrix	9
3.1.6	Damping.....	9
3.1.7	Dissipation dilution.....	10
3.2	Potential terms for standard elements	11
3.2.1	Wires.....	11
3.2.2	Springs	12
3.3	Dynamics	13
3.4	Dynamics via frequency dependent matrices	13
3.5	Dynamics via state-space formalism	13
4	<i>Illustration of concepts with a toy model</i>	15
4.1	Stiffness and coupling matrices	15
4.2	The mass matrix	16
4.3	Basic transfer functions	17
4.4	The state-space matrices	17
5	<i>Implementation in software</i>	18
5.1	Directory structure	18
5.2	Phases of the calculation	19
5.3	Utility functions	19
5.4	Model specific utilities	20
6	<i>Models of Particular Interest</i>	20
6.1	Triple Model	20
6.2	Quad Model	21
6.3	“Lite” and “Xtra-Lite” Versions of the Triple and Quad Models	21

6.4	“Lateral” Versions	22
6.5	“V5” Violin Mode Versions	22
7	<i>Exploring the supplied cases</i>	22
7.1	Computing cases from scratch versus using precomputed results.....	22
7.2	Stages of the calculation	22
7.3	Available listing and plotting commands.....	24
7.3.1	Frequencies	24
7.3.2	Mode shape tables.....	24
7.3.3	Mode shape plots	25
7.3.4	Force transfer function plots.....	26
7.3.5	Displacement transfer function plots	26
7.3.6	Thermal noise plots.....	26
7.4	Default numerical values.....	26
7.5	Creating new cases of the existing models.....	27
7.6	Using overrides.....	28
8	<i>The definition of the model</i>	28
8.1	Name, Description and Version.....	29
8.2	Preliminaries	29
8.3	The model definition proper	29
8.3.1	The master variable list: <code>allvars</code>	29
8.3.2	The list of “floats”.....	30
8.3.3	The parameter list: <code>allparams</code>	30
8.3.4	Coordinate lists for rigid bodies and points on them.....	30
8.3.5	Items with gravitational potential energy: <code>gravlist</code>	31
8.3.6	Wires: <code>wirelist</code>	31
8.3.7	Springs: <code>springlist</code>	33
8.3.8	The kinetic energy: <code>kinetic</code>	34
8.3.9	Values of constants: <code>defaultvalues</code>	34
8.3.10	Approximation to equilibrium position: <code>startpos</code>	35
8.3.11	Extra potential terms for debugging: <code>preeqterm</code> and <code>posteqterm</code>	36
8.4	Model dependent diagnostic utilities.....	36
8.4.1	Angular velocity transformation matrices: <code>e2s</code> , <code>e2b</code> and <code>e2ni</code>	36
8.4.2	Plotting routines: <code>eigenplot[]</code>	37
8.4.3	Listing routines: <code>pretty[]</code>	38
8.4.4	Input and output vectors	38
9	<i>The calculation of the model</i>	38
9.1	The <code>Calculate[]</code> function	38
9.2	Stage 0A – normal modes without wire bending elasticity	39
9.2.1	Numerical Substitutions: <code>constval</code>	39

9.2.2	Numerical start position for minimization of the potential: <code>startval</code>	39
9.2.3	Coordinates that don't participate in the minimization: <code>nonoptcoords</code> and <code>nonoptval</code>	39
9.2.4	Variables that participate in the minimization: <code>optcoords</code>	39
9.2.5	The base list of potential terms for the minimization: <code>potentialtermlist</code>	39
9.2.6	Additional wire terms for the minimization: <code>potentialtermlistWBnr</code> , <code>potentialtermlistWTnr</code> , <code>potentialtermlistWEnr</code>	40
9.2.7	The full expression for the potential: <code>potential</code>	40
9.2.8	The numeric potential: <code>potentialNN</code>	40
9.2.9	The minimization: <code>potential0</code> and <code>optval</code>	40
9.2.10	Velocity variables: <code>velocities</code>	40
9.2.11	The kinetic energy matrix: <code>kineticN</code> and <code>kineticmatrix</code>	40
9.2.12	The potential used for the normal mode calculation: <code>potentialtermlist0</code>	40
9.2.13	The Stage 0A damping-specific potential matrices: <code>potentialmatrices0T</code>	41
9.2.14	The Stage 0 potential matrix: <code>potentialmatrix0</code>	41
9.2.15	The Stage 0 normal modes: <code>eigenvalues0</code> , <code>eigenvectors0</code> and <code>Hz0</code>	41
9.3	Stage 0B – damping without wire bending elasticity	41
9.3.1	Potential matrices without tension: <code>potentialmatrices0NT</code>	41
9.3.2	The equations of motion function and the coupling function: <code>eom0</code> and <code>coupling0</code> 41	
9.4	Preparation for Stages 1 and 2	41
9.4.1	Wire angles: <code>relaxval</code>	41
9.4.2	Additional potential term lists: <code>potentialtermlistWB</code> and <code>potentialtermlistWE</code>	42
9.5	Stage 1A – normal modes and damping with wire bending elasticity	42
9.6	Stage 1B – normal modes and damping with wire torsional elasticity	42
9.7	Stage 2 – normal modes and damping with additional wire stretch due to bending...	43
9.8	Exporting state-space matrices	43
9.8.1	Building state-space matrices with model results	43
9.8.2	Exporting state-space matrices	43
10	Appendix	43
10.1	Download locations	43
10.2	Directory structure	44
10.2.1	Toolkit Directory	44
10.2.2	Model/Case/Calculation Directories	45
10.2.3	Detail of directory structure for typical model	45
10.2.4	Directory structure differences from older versions	47
10.2.5	Flow of control between code in toolkit models, cases and calculations	48

1 Introduction

1.1 Purpose and Scope

This document explains the physical assumptions, internal structure and usage of the models of the Advanced LIGO suspensions written in Mathematica by Mark Barton (“the Mathematica model” as opposed to “the Matlab model” of Calum Torrie, Ken Strain et al.).

1.2 References

Ultralow frequency oscillator using a pendulum with crossed suspension wires. M.A. Barton and K. Kuroda, Rev. Sci. Instrum. 65(12): 3775, 1994.

A low-frequency vibration isolation table using multiple crossed-wire suspensions. M.A. Barton et al., Rev. Sci. Instrum. 67(11): 3994, November 1996

[P000039](#)-00: Matthew Husman, PhD Thesis, 1999, University of Glasgow.

[T050255](#)-07: Investigation into blade torsion, blade lateral flexibility, and the effect they have on blade and wire performance, I. Wilmut et al.

[T050172](#)-00: Modifications to Quad Controls Prototype to fix the “d’s”, C.I. Torrie.

[T050257](#)-01: Quad Pendulum Controls Prototype - Identification of Modes, M.A. Barton

[T020011](#)-00: Suspension model comparisons, M.A. Barton.

[T070101](#)-00: Dissipation dilution, M.A. Barton.

[T080188](#)-v1: Models of the Advanced LIGO Suspensions in MATLAB

1.3 Version history

1/15/03: Pre-rev-00 draft. Discusses quad model v 2.0.3.

2/9/03: Rev. 00.

3/2/2005: Rev. 01. Discusses quad and triple models 3.3 and toolkit version 2.0.

9/13/2006: Rev. 02 draft. Added theory of transfer functions and state-space matrix export. Added illustration of concepts using a toy model.

6/4/08: Rev. 02 final. Added References section. Mentioned quad lateral model, violin mode models, Mathematica v.6 changes. Added diagram for toy model section. Later added to new DCC as T020205-v1.

7/8/12ff: -v2 draft. Major rearrangement. Stuff on SVN-compatible directory structure.

2 Overview

This document describes a toolkit of utilities in Mathematica for setting up models of mass-wire-spring systems, plus triple, quad and other pendulum models based on it. Throughout this document and the toolkit code, a “model” is a specification of particular arrangement of toolkit elements (i.e., masses, wires, and springs). A model definition will include a default set of

numerical values mostly for debugging, but these are intended to be overridden in whole or part. A “case” of a model goes on to specify a particular set of numeric values of actual interest.

The toolkit evolved out of a calculation of the properties of a device called the X-pendulum developed as a possible low-frequency vibration isolator for the Japanese TAMA project (Barton et al., 1994, 1996). At the time of version -00 of this document, the toolkit was merely a set of Mathematica functions that had to be copied into each model description file. As the number of models increased, the task of keeping the various copies synchronized became onerous, so the functions from v2.5 of the quad model were encapsulated as a Mathematica package, `PendUtil.nb`. The version of the toolkit described in this document is v5.9.

The basic physical objects which the toolkit allows for are:

- (i) Rigid bodies with up to 6 degrees of freedom
- (ii) Massless wires with longitudinal, transverse and torsional elasticity
- (iii) Springs described by a 6x6 matrix of elastic constants and a 6x1 vector of preload forces

All sources of elasticity can have arbitrary frequency dependent damping.

A number of models of interest to LIGO have been created, including one for a generic GEO-style triple suspension (as for the HLTS, HSTS and BSFM suspensions) and one for a quad suspension. Additional models can be defined fairly easily by using existing ones as templates.

The toolkit and a selection of models have been published on the SUS group SVN repository.

3 Theory

3.1 Method of calculation

3.1.1 Basic Normal Mode Calculation

The primary calculation is based on the method of normal modes as described in Goldstein, “Classical Mechanics”. Conceptually such a calculation has the following steps:

- (i) Express the potential energy of the system in terms of the coordinates:

$$E_p = E_p(x_1, \dots, x_n) = E_p(\mathbf{x})$$

- (ii) Express the kinetic energy of the system in terms of the coordinates and coordinate velocities:

$$E_K = E_K(x_1, \dots, x_n, \dot{x}_1, \dots, \dot{x}_n)$$

- (iii) Minimize the potential energy to find the equilibrium values of the coordinates.

$$\mathbf{x}_{eq} = (x_{1(eq)}, \dots, x_{n(eq)})^T$$

- (iv) Differentiate the potential energy of the system w.r.t. pairs of coordinates at equilibrium to create a matrix of second derivatives, a.k.a., the potential energy matrix or the stiffness matrix.

$$\mathbf{K}: K_{ij} = \left. \frac{\partial E_p}{\partial x_i \partial x_j} \right|_{\mathbf{x}=\mathbf{x}_{eq}}$$

$$E_p = E_p(\mathbf{x}_{eq}) + \frac{1}{2}(\mathbf{x} - \mathbf{x}_{eq})^T \mathbf{K}(\mathbf{x} - \mathbf{x}_{eq})$$

- (v) Differentiate the kinetic energy of the system w.r.t. pairs of coordinate velocities at equilibrium to create a matrix of second derivatives, a.k.a., the kinetic energy matrix or the mass matrix.

$$\mathbf{M}: M_{ij} = \left. \frac{\partial E_K}{\partial \dot{x}_i \partial \dot{x}_j} \right|_{\substack{\dot{\mathbf{x}}=0 \\ \mathbf{x}=\mathbf{x}_{eq}}}$$

$$E_K = \frac{1}{2} \dot{\mathbf{x}}^T \mathbf{M} \dot{\mathbf{x}}$$

- (vi) Assume a sinusoidal solution, putting the equation of motion in the form

$$\mathbf{K} \mathbf{e}_i = \omega_i^2 \mathbf{M} \mathbf{e}_i$$

- (vii) Do a simultaneous diagonalization of the stiffness and mass matrices to obtain the eigenfrequencies $f_i = \omega_i/2\pi$ and eigenmodes \mathbf{e}_i :

$$\mathbf{x}_i(t) = \mathbf{x}_{eq} + \mathbf{e}_i e^{\omega_i t}$$

However for a practical calculation step (iv) needs to be considerably elaborated, partly for efficiency and partly to support additional calculations such as transfer functions and thermal noise estimates.

3.1.2 Categories of Coordinates and the Master Potential Matrix

The coordinates used in the normal mode analysis need to be independent and to describe the position and orientation of all the masses that move during a normal mode oscillation. However two other classes of coordinates are of interest.

First there are the position/orientation coordinates of any junctions directly between elastic elements with no associated mass. For the purposes of the toolkit, these have been named *floats*, because they float between mass positions under the influence of the adjacent elastic elements.

$$\mathbf{q} = (q_1, \dots, q_m)^T$$

It is convenient to use such junctions and such coordinates in the specification of some problems, but the coordinates are not independent for the purposes of the normal mode analysis and must be suppressed by solving the equations for force equilibrium at the junction in terms of the true normal mode coordinates.

Second, there are the position/orientation coordinates of the structure and any other points linked to the pendulum proper by elastic elements.

$$\mathbf{s} = (s_1, \dots, s_l)^T$$

These are fixed at nominal positions

$$\mathbf{s}_{nom} = (s_{1(nom)}, \dots, s_{l(nom)})^T$$

for the purposes of the normal mode analysis but are taken as movable for the purposes of calculating a transfer function from displacement.

To cope with these complications it is convenient to consider a master potential matrix \mathbf{P} with second derivatives with respect to all three classes of coordinates:

$$E_P = E_P(\mathbf{x}_{eq}, \mathbf{q}_{eq}, \mathbf{s}_{nom}) + \frac{1}{2} \begin{pmatrix} \mathbf{x}^T - \mathbf{x}_{eq}^T & \mathbf{q}^T - \mathbf{q}_{eq}^T & \mathbf{s}^T - \mathbf{s}_{nom}^T \end{pmatrix} \mathbf{P} \begin{pmatrix} \mathbf{x} - \mathbf{x}_{eq} \\ \mathbf{q} - \mathbf{q}_{eq} \\ \mathbf{s} - \mathbf{s}_{nom} \end{pmatrix}$$

The master potential matrix has a block structure with many useful submatrices:

$$\mathbf{P} = \begin{pmatrix} \mathbf{K} & \mathbf{C}_{XQ} & \mathbf{C}_{XS} \\ \mathbf{C}_{QX} & \mathbf{Q} & \mathbf{C}_{QS} \\ \mathbf{C}_{SX} & \mathbf{C}_{SQ} & \mathbf{S} \end{pmatrix} \quad (\mathbf{C}_{XQ} = \mathbf{C}_{QX}^T \text{ etc})$$

\mathbf{K} , \mathbf{Q} and \mathbf{S} specify the interactions among a class of coordinates with the other two types held fixed. \mathbf{C}_{XQ} and the like specify the cross-coupling from one class to another.

3.1.3 The Effective Potential Matrix

If there are a non-zero number of float coordinates, then the \mathbf{K} submatrix of \mathbf{P} is no longer the appropriate one to use in the normal mode analysis, because it contains partial derivatives taken with \mathbf{q} constant. Rather, we need an effective value, \mathbf{K}_{eff} , evaluated with the q_i at their dynamic equilibrium positions at all time. With the structure at its nominal position, $\mathbf{s} = \mathbf{s}_{nom}$, the equilibrium positions are

$$\mathbf{q} = \mathbf{q}_{eq} - \mathbf{Q}^{-1} \mathbf{C}_{QX} (\mathbf{x} - \mathbf{x}_{eq})$$

so that

$$\mathbf{K}_{eff} = \mathbf{K} - \mathbf{C}_{XQ} \mathbf{Q}^{-1} \mathbf{C}_{QX}$$

3.1.4 The Equation Of Motion Matrix and the Coupling Matrix

To calculate a transfer function from displacement input of the structure, we need the matrix that gives the coupling of displacements of the \mathbf{s} coordinates to generalized force inputs at the \mathbf{x} coordinates. If there are no floating coordinates, this is simply the submatrix \mathbf{C}_{XS} . However as with the normal mode analysis, there are additional terms due to the variation in \mathbf{q} :

$$f_{xs} = \mathbf{C}_{XS(eff)} (\mathbf{s} - \mathbf{s}_{nom}) = (\mathbf{C}_{XS} - \mathbf{C}_{XQ} \mathbf{Q}^{-1} \mathbf{C}_{QS}) (\mathbf{s} - \mathbf{s}_{nom})$$

If we also allow for other generalized forces \mathbf{f}_x acting directly on \mathbf{x} , the equation of motion during a transfer function test is

$$\mathbf{K}_{eff} (\mathbf{x} - \mathbf{x}_{eq}) + \mathbf{M} \dot{\mathbf{x}} = \mathbf{f}_x + \mathbf{C}_{XS(eff)} (\mathbf{s} - \mathbf{s}_{nom})$$

In the frequency domain this becomes

$$\mathbf{K}_{eff}(\mathbf{x} - \mathbf{x}_{eq}) - (2\pi f)^2 \mathbf{M}(\mathbf{x} - \mathbf{x}_{eq}) = \mathbf{f}_x + \mathbf{C}_{XS(eff)}(\mathbf{s} - \mathbf{s}_{nom})$$

where f is frequency in Hz. Numerically, this can be solved for \mathbf{x} for a sequence of different values of f to give force-input or displacement-input transfer functions as a function of frequency. The left hand side of the above equation can be conveniently rewritten in terms of an equation of motion matrix

$$\mathbf{E} = \mathbf{K}_{eff} - (2\pi f)^2 \mathbf{M}$$

3.1.5 The Coupling Matrix

When models are put in state-space form for use in Simulink or E2E, it may be useful to couple one model to another, such as and in particular a SUS pendulum model to a model of the corresponding SEI platform. This requires calculating the forces on the support, which are given by a sum of two terms. The component generated as the pendulum moves but the structure remains constant is given by the transpose of the coupling matrix:

$$f_{sx} = \mathbf{C}_{SX(eff)}(\mathbf{x} - \mathbf{x}_{eq}) = (\mathbf{C}_{SX} - \mathbf{C}_{SQ} \mathbf{Q}^{-1} \mathbf{C}_{QX})(\mathbf{x} - \mathbf{x}_{eq}) = \mathbf{C}_{XS(eff)}^T(\mathbf{x} - \mathbf{x}_{eq})$$

This would be incorporated in the state-space C matrix. The component generated as the pendulum remains constant but the structure moves is

$$f_{ss} = \mathbf{S}_{(eff)}(\mathbf{s} - \mathbf{s}_{nom}) = (\mathbf{S} - \mathbf{C}_{SQ} \mathbf{Q}^{-1} \mathbf{C}_{QS})(\mathbf{s} - \mathbf{s}_{nom})$$

This would be incorporated in the state-space D matrix. Together the two terms give the total force for arbitrary relative motion of the support and pendulum.

3.1.6 Damping

Lossiness in elastic components can be represented by a complex elastic constant:

$$k \rightarrow k_0(\mathcal{E}'(\omega) + i\mathcal{E}''(\omega)) \quad (\text{where } \omega = 2\pi f)$$

where the real and imaginary multipliers \mathcal{E}' and \mathcal{E}'' are related by the Kramers-Krönig relations:

$$\mathcal{E}'(\omega) - 1 = \frac{2}{\pi} PV \int_{-\infty}^{\infty} \frac{\mathcal{E}''(x)}{x - \omega} dx$$

$$\mathcal{E}''(\omega) = -\frac{2}{\pi} PV \int_{-\infty}^{\infty} \frac{\mathcal{E}'(x) - 1}{x - \omega} dx$$

Provided the losses are small, the frequency dependence of the real part can be neglected and the imaginary part can be identified as a frequency dependent loss angle:

$$k \rightarrow k_0(1 + i\phi(f))$$

Two issues need to be handled to apply this in a general way in the context of the toolkit. First, different elastic components will have damping multipliers of different magnitude and frequency dependence. This is handled by breaking up the total potential into individual terms and processing them independently. If dissipation dilution is not applicable (but see next section):

$$\mathbf{P} = \sum \mathbf{P}_i(\varepsilon'_i(f) + i\varepsilon''_i(f)) \dots \dots \dots (i=1 \dots \text{number of terms; no dissipation dilution})$$

This has a number of advantages:

- It is efficient to generate the \mathbf{P}_i because each typically depends on only a few variables. (This permits an optimization in which the variables used in a particular term are determined by a preliminary test, with only those variables then iterated over. This turns out to be *much* quicker than letting Mathematica discover for itself in the course of doing the partial derivatives that most terms are independent of most variables.)
- It is efficient to organize the \mathbf{P}_i (which are numeric) separately from the ε'_i and ε''_i (which are symbolic) and combine them as late as possible in the calculation. This also allows different values of ε'_i and ε''_i to be tried without a costly recomputation of the \mathbf{P}_i .
- It is easy to allow the potential terms for wire bending and torsion to be optional. This is important because they are very costly to compute, but typically small in magnitude and significant only for the calculation of thermal noise.
- It also facilitates the handling of dissipation dilution (see below).

3.1.7 Dissipation dilution

Naively implementing the calculation for \mathbf{P} as a sum over the \mathbf{P}_i with damping multipliers will typically overestimate the damping (and the thermal noise) because it does not allow for dissipation dilution. Dissipation dilution arises because there are two quite different ways of creating a restoring force on an object using a lossy spring ([T070101-00](#)). The first is to harness the spring directly to the object. This produces first-order changes in the length of the spring when the object undergoes an oscillation. Thus the dissipated energy is also first-order in the amplitude. The second method is to use the spring to generate a static force and to use some mechanical device to change the mechanical advantage between the static force and the force on the object. A violin string with longitudinal but not bending loss is the canonical example of this case. Longitudinal tension is used to create a lateral restoring force. First-order lateral excursions of the string produce only a second-order change in length and thus only a second-order loss in energy, which for small amplitudes is negligible.

A pendulum, of the sort the toolkit is intended to simulate, is an interesting variation on the second case. As a matter of physical reality, there are two independent reasons why a pendulum with a lossy fibre should have low net loss. The first is often remarked on: that the potential energy generated during lateral excursions of the bob is stored not in extension of the fibre but gravitationally. The second is that the restoring force on the bob is the lateral component of the tension in the fibre. A pendulum would still be low-loss even if the tension were provided not by gravity but by a second wire connecting the bob to a mechanical ground below, creating a system similar to a violin string, just with a mass in the middle.

For more on this point, and dissipation dilution more generally, see [T070101-00](#). The upshot is that since the low-loss component of the restoring force is produced by a static force coupled through a variable mechanical advantage, the easy way to isolate it is to evaluate the second derivatives twice, once with the full potential and once with all the tensions and static forces arbitrarily set to zero:

$$\mathbf{P} = \sum (\mathbf{P}_i|_{tension_off} (\boldsymbol{\varepsilon}'_i(f) + i\boldsymbol{\varepsilon}''_i(f))) + \sum (\mathbf{P}_i|_{tension_on} - \mathbf{P}_i|_{tension_off})$$

Any restoring force that persists with the tension off involves length changes of the associated spring that are first-order in oscillation amplitude, and so produces an energy loss that is first-order in the stored energy.

Because the wire bending potential terms are small and costly to compute (see Section 3.2.1), they are not computed twice. However since the contribution to static tension due to wire bending is typically trivial, this is normally a satisfactory approximation.

3.2 Potential terms for standard elements

3.2.1 Wires

The treatment of the wires was based on the conceptual framework used by Matthew Husman in [P000039-00](#). However the Maple code written by Matt was not adapted - rather, the calculation was reimplemented from scratch in Mathematica.

A wire is considered to join a point given in local coordinates on one object to a similar point on a second object. Optionally the user can specify the attachment angles at which the wire emerges from the objects as two three-component vectors. If a single dummy symbol is given instead of an vector, an angle which makes the wire straight at equilibrium is automatically calculated and substituted. Alternatively, if “0” is specified, the wire is treated as if freely hinged at the attachment point.

The potential energy of each wire is broken up into a number of separate terms, each with its own damping function. The first reflects increase of the straight-line distance between the end-points:

$$E_{P(wire_longitudinal)} = \frac{1}{2} k_w (l(t) - l_0)^2$$

The user is responsible for supplying a value for the net elastic constant of the wire k_w , either directly or in terms of other quantities such as the Young’s modulus of the wire. To implement the tension-off condition required for calculation of dissipation dilution, the unstretched length l_0 is set to equal the equilibrium length.

The torsional energy is a single term for the purposes of the damping but has two sub-terms, one proportional to the unloaded torsional rigidity and one proportional to the tension:

$$E_{P(wire_torsional)} = \frac{1}{2} \left(\frac{TJ}{A} + GJ_e \right) \Delta\theta_T^2$$

where T is the tension, J is the polar moment of area, G is the torsional elastic modulus, J_e is a torsional stiffness geometric factor of the same dimensions as J , and $\Delta\theta_T$ is the net twist.

The energy of lateral bending is calculated by assuming the wire is a massless beam under tension.

$$E_{P(wire_bending)} = \frac{YM_1}{2} \int_0^L \left(\frac{d^2y}{dl^2} \right)^2 dl + \frac{YM_2}{2} \int_0^L \left(\frac{d^2z}{dl^2} \right)^2 dl$$

where Y is the Young's modulus, M_i are the axial moments of area along the principal axes of the wire, and $y(l)$ and $z(l)$ are the deviations from a straight line in the respective principal axis directions as a function of distance l along the wire. The shape of the beam can be calculated analytically and has been integrated to give a closed form expression for the energy, which is too complicated to reproduce here but depends on the tension, and the amount of bending along these axes at the end-points relative to the straight line between them.

Finally there is a term representing the energy due to the small additional amount of longitudinal stretch relative to the straight line between the endpoints) due to lateral bending.

$$E_{P(\text{wire_extra_stretch})} = \frac{T}{2} \int_0^L \left(\frac{dy}{dl} \right)^2 dl$$

where T is the tension. This has similarly been integrated to give a (very long) closed-form expression for the energy.

Some straightforward but lengthy geometry is required to relate the above energy terms to the positions and orientations of the masses. To avoid errors, all of this geometry was derived symbolically in Mathematica. For efficiency in later calculating the partial derivatives, the geometry is linearized about the equilibrium position. By default, to avoid circularity, the equilibrium position is computed from a version of the potential that omits the wire bending and extra-extension terms. In the common case where the pendulum is stable without the extra wire terms, and the user is happy to accept the automatically calculated attachment angles that conform to this preliminary equilibrium position (i.e., the way wire clamps are normally designed), the true equilibrium position is the same, so that no additional approximation has been introduced.

However if the pendulum is not stable without the wire bending and extra-extension terms, or the user wants to explore the effect of wire clamps that don't attempt to match the natural angle of the wire, the wire terms can be including from the beginning. In this case, again to avoid circularity, any unspecified wire attachment angles are calculated to conform to the starting position used for numerically finding the equilibrium position, rather than the equilibrium position itself.

3.2.2 Springs

A spring is similar to a wire in that it connects two points on two masses via two attachment angles. However a spring is taken to be a zero-length element, so the reference state is with the two points superimposed (and the attachment vectors aligned). The spring then exerts restoring forces and torques on the masses in 6 DOFs according to a tensor of elastic constants and a vector of pre-load forces:

$$E_{P(\text{spring})} = \frac{1}{2} (\Delta x \quad \Delta y \quad \Delta z \quad \Delta\theta_Y \quad \Delta\theta_P \quad \Delta\theta_R) \mathbf{K}_S \begin{pmatrix} \Delta x \\ \Delta y \\ \Delta z \\ \Delta\theta_Y \\ \Delta\theta_P \\ \Delta\theta_R \end{pmatrix} + \mathbf{f}_S^T \begin{pmatrix} \Delta x \\ \Delta y \\ \Delta z \\ \Delta\theta_Y \\ \Delta\theta_P \\ \Delta\theta_R \end{pmatrix}$$

where Δx etc are the differences in x , y , z , yaw, pitch and roll in the local coordinate system of the spring, relative to the reference state.

3.3 Dynamics

Two different ways of representing the dynamics of the system are useful. For use within Mathematica, it is convenient to use frequency-dependent matrices. These allow the full complex-multiplier description of the damping to be retained. For export to Matlab and E2E, the state-space formalism is more convenient, although.

3.4 Dynamics via frequency dependent matrices

The dynamics of the pendulum proper can be represented by a frequency-dependent equation of motion matrix

$$\mathbf{E} = \mathbf{K}_{eff} - (2\pi f)^2 \mathbf{M}$$

This is implemented as a matrix-valued pure function of a single nameless variable which represents frequency. The frequency domain transfer function from force/torque on the pendulum to displacements of the pendulum is

$$\mathbf{x} = \mathbf{E}^{-1} \mathbf{f}_x$$

This can be solved numerically for a series of values of frequency f . For purposes of actual computation, taking the inverse and multiplying is best avoided in favour of an equivalent special purpose routine such as Mathematica's `LinearSolve[]` .)

The transfer function from structure displacement is then derived by transforming to a force first:

$$\mathbf{x} = \mathbf{E}^{-1} \mathbf{C}_{XS(eff)} \mathbf{s}$$

3.5 Dynamics via state-space formalism

For time domain simulation, the state-space formalism is more convenient. The toolkit doesn't attempt to do such simulation itself but it does allow for export of state-space matrices in formats suitable for use in either Matlab/Simulink or E2E. The form of state-space formalism convenient for mechanical problems takes the state of the pendulum to consist of both the positions/orientations and the linear/angular velocities in a double-length vector:

$$\begin{pmatrix} \mathbf{x} - \mathbf{x}_{eq} \\ \dot{\mathbf{x}} \end{pmatrix}$$

We assume a standard set of inputs and outputs convenient for E2E:

$$inputs = \begin{pmatrix} \mathbf{s} - \mathbf{s}_{nom} \\ \mathbf{f}_x \end{pmatrix}$$

i.e., displacement of the structure and force on the mass, and

$$outputs = \begin{pmatrix} \mathbf{x} - \mathbf{x}_{eq} \\ \mathbf{f}_s \end{pmatrix}$$

i.e., the position of the mass and force back on the structure. Then the state-space matrices are

$$\mathbf{A} = \begin{pmatrix} \mathbf{0} & \mathbf{I} \\ \text{Re}(-\mathbf{M}^{-1}\mathbf{K}_{eff}) & \frac{\text{Im}(-\mathbf{M}^{-1}\mathbf{K}_{eff})}{2\pi f} \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ \frac{-k_1 k_1}{m(k_1 + k_1)} & 0 \end{pmatrix}$$

$$\mathbf{B} = \begin{pmatrix} \mathbf{0} & \mathbf{0} \\ \text{Re}(-\mathbf{M}^{-1}\mathbf{C}_{XS(eff)}) & \mathbf{M}^{-1} \end{pmatrix} = \begin{pmatrix} 0 & 0 \\ \frac{-k_1 k_1}{m(k_1 + k_1)} & \frac{1}{m} \end{pmatrix}$$

$$\mathbf{C} = \begin{pmatrix} \mathbf{I} & \mathbf{0} \\ \text{Re}(-\mathbf{C}_{SX(eff)}) & \mathbf{0} \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ \frac{k_1 k_2}{k_1 + k_2} & 0 \end{pmatrix}$$

$$\mathbf{D} = \begin{pmatrix} \mathbf{0} & \mathbf{0} \\ -\mathbf{S}_{eff} & \mathbf{0} \end{pmatrix} = \begin{pmatrix} 0 & 0 \\ \frac{-k_1 k_2}{k_1 + k_2} & 0 \end{pmatrix}$$

and the state space equations are

$$\begin{pmatrix} \dot{\mathbf{x}} \\ \ddot{\mathbf{x}} \end{pmatrix} = \begin{pmatrix} \mathbf{0} & \mathbf{I} \\ \text{Re}(-\mathbf{M}^{-1}\mathbf{K}_{eff}) & \frac{\text{Im}(-\mathbf{M}^{-1}\mathbf{K}_{eff})}{2\pi f} \end{pmatrix} \begin{pmatrix} \mathbf{x} - \mathbf{x}_{eq} \\ \dot{\mathbf{x}} \end{pmatrix} + \begin{pmatrix} \mathbf{0} & \mathbf{0} \\ \text{Re}(-\mathbf{M}^{-1}\mathbf{C}_{XS(eff)}) & \mathbf{M}^{-1} \end{pmatrix} \begin{pmatrix} \mathbf{s} - \mathbf{s}_{nom} \\ \mathbf{f}_x \end{pmatrix}$$

and

$$\begin{pmatrix} \mathbf{x} - \mathbf{x}_{eq} \\ \mathbf{f}_s \end{pmatrix} = \begin{pmatrix} \mathbf{I} & \mathbf{0} \\ \text{Re}(-\mathbf{C}_{SX(eff)}) & \mathbf{0} \end{pmatrix} \begin{pmatrix} \mathbf{x} - \mathbf{x}_{eq} \\ \dot{\mathbf{x}} \end{pmatrix} + \begin{pmatrix} \mathbf{0} & \mathbf{0} \\ -\mathbf{S}_{eff} & \mathbf{0} \end{pmatrix} \begin{pmatrix} \mathbf{s} - \mathbf{s}_{nom} \\ \mathbf{f}_x \end{pmatrix}$$

The transfer function in Laplace transform format is

$$\begin{pmatrix} \mathbf{x} - \mathbf{x}_{eq} \\ \mathbf{f}_s \end{pmatrix} = (\mathbf{C}(s\mathbf{I} - \mathbf{A})^{-1}\mathbf{B} + \mathbf{D}) \begin{pmatrix} \mathbf{s} - \mathbf{s}_{nom} \\ \mathbf{f}_x \end{pmatrix}$$

where $s = i\omega = 2\pi if$.

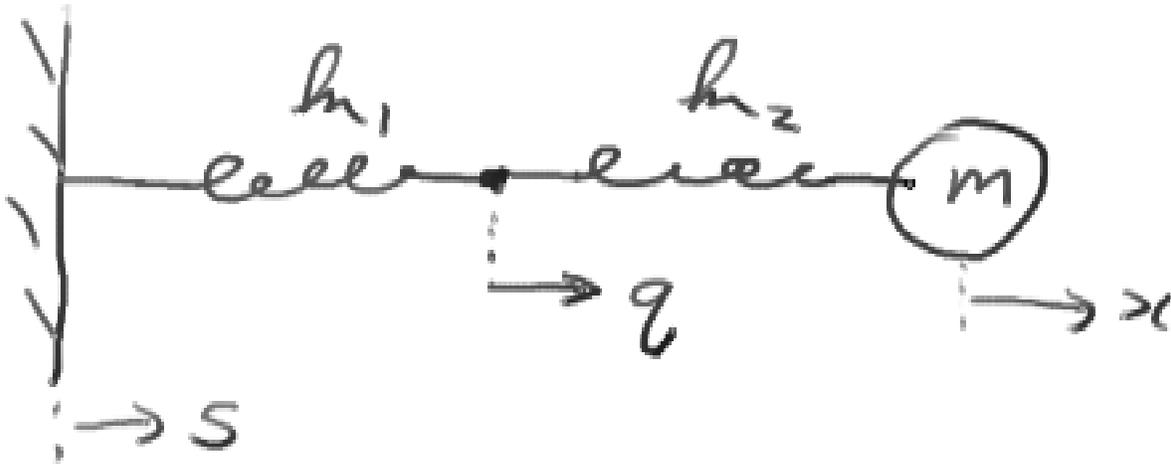
Note that for numerical time domain simulation, the matrix coefficients need to be real and frequency-independent. The above matrices are real, but not necessarily frequency independent. The real parts of the stiffness matrices are typically only weakly dependent on frequency, so it is normally a good approximation to substitute the values at DC or some arbitrary frequency near the eigenfrequencies of most interest. However the submatrix

$$\frac{\text{Im}(-\mathbf{M}^{-1}\mathbf{K}_{\text{eff}})}{2\pi f}$$

in the lower right of the \mathbf{A} matrix is only frequency independent if all sources of damping are proportional to velocity, in which case the imaginary part of the matrix in the numerator will be proportional to f . This will cancel with the f in the denominator leaving a fully numeric matrix. Structural damping, on the other hand will have a frequency independent numerator but a frequency dependent quotient and thus cannot be represented in this format.

4 Illustration of concepts with a toy model

Consider a mass m free to move in the x direction, connected to mechanical ground by two massless springs, k_1 and k_2 , in series. Let the x -coordinate of the mass be x , the coordinate of the junction between the two springs be q , and the coordinate of the ground be s . In the terminology of the toolkit, x is a variable, q is a float, and s is a parameter.



4.1 Stiffness and coupling matrices

The potential energy is

$$E_p = \frac{1}{2}k_1(q-s)^2 + \frac{1}{2}k_2(x-q)^2$$

The master stiffness matrix is

$$\mathbf{P} = \begin{pmatrix} \mathbf{K} & \mathbf{C}_{xQ} & \mathbf{C}_{xS} \\ \mathbf{C}_{Qx} & \mathbf{Q} & \mathbf{C}_{QS} \\ \mathbf{C}_{Sx} & \mathbf{C}_{SQ} & \mathbf{S} \end{pmatrix} = \begin{pmatrix} k_2 & -k_2 & 0 \\ -k_2 & k_1 + k_2 & -k_1 \\ 0 & -k_1 & k_1 \end{pmatrix}$$

The float stiffness matrix \mathbf{Q} gives the elasticity on the junction with both the structure and mass fixed:

$$\mathbf{Q} = (k_1 + k_2)$$

As might be hoped, the magnitude is the sum of the individual stiffnesses, as for springs added in parallel. The positive sign here says that when junction is displaced a small amount in the +x direction, it encounters a potential with a positive gradient, i.e., there is a restoring force on it in the -x direction.

This sign convention is prototypical for all the stiffness and coupling matrices: multiplying a stiffness matrix by a displacement vector gives a potential gradient vector, and adding a negative sign gives a force/torque vector.

The raw (mass) stiffness matrix \mathbf{K} is equal to the spring constant of the nearest spring only (k_2), because the partial derivatives are taken with q constant. The effective stiffness matrix that applies when q is allowed to “float” to its time-dependent neutral position between s and x is

$$\mathbf{K}_{eff} = \mathbf{K} - \mathbf{C}_{xQ} \mathbf{Q}^{-1} \mathbf{C}_{Qx} = (k_2) - \left(\frac{k_2^2}{k_1 + k_2} \right) = \left(\frac{k_1 k_2}{k_1 + k_2} \right)$$

As might be hoped, this is the usual formula for the stiffness of two springs added in series.

The raw coupling matrix from the structure to the mass (\mathbf{C}_{XS}) is zero, because the two points are not connected directly. However the effective value incorporates the action of the two springs:

$$\mathbf{C}_{XS(eff)} = \mathbf{C}_{XS} - \mathbf{C}_{xQ} \mathbf{Q}^{-1} \mathbf{C}_{QS} = (0) - \left(\frac{(-k_2)(-k_1)}{k_1 + k_2} \right) = \left(\frac{-k_1 k_2}{k_1 + k_2} \right)$$

The negative sign here says that when the structure moves in the +x direction, the mass encounters a potential with a negative gradient, i.e., there is a force on it in the +x direction.

The structure stiffness matrix is similar to the (mass) stiffness matrix and gives the restoring force on the structure with the mass fixed but the spring junction floating. The raw value reflects only the nearest spring (k_1), but the effective value reflects the series sum of the two stiffnesses.

$$\mathbf{S}_{eff} = \mathbf{S} - \mathbf{C}_{sQ} \mathbf{Q}^{-1} \mathbf{C}_{Qs} = (k_1) - \left(\frac{k_1^2}{k_1 + k_2} \right) = \left(\frac{k_1 k_2}{k_1 + k_2} \right)$$

4.2 The mass matrix

The kinetic energy is

$$E_K = \frac{1}{2} m v^2$$

so the mass matrix is just

$$\mathbf{M} = (m)$$

4.3 Basic transfer functions

The dynamics of the pendulum proper can be represented by a frequency-dependent equation of motion matrix

$$\mathbf{E} = \mathbf{K}_{eff} - (2\pi f)^2 \mathbf{M} = \begin{pmatrix} k_1 k_2 \\ k_1 + k_2 - (2\pi f)^2 m \end{pmatrix}$$

4.4 The state-space matrices

We assume the standard set of inputs and outputs convenient for time domain simulation using the LIGO E2E simulation package:

$$inputs = \begin{pmatrix} \mathbf{s} - \mathbf{s}_{nom} \\ \mathbf{f}_x \end{pmatrix} = \begin{pmatrix} s \\ f_x \end{pmatrix}$$

i.e., displacement of the structure and force on the mass, and

$$outputs = \begin{pmatrix} \mathbf{x} - \mathbf{x}_{eq} \\ \mathbf{f}_s \end{pmatrix} = \begin{pmatrix} x \\ f_s \end{pmatrix}$$

i.e., the position of the mass and force back on the structure. Then the state-space matrices are

$$\mathbf{A} = \begin{pmatrix} \mathbf{0} & \mathbf{I} \\ \text{Re}(-\mathbf{M}^{-1}\mathbf{K}_{eff}) & \frac{\text{Im}(-\mathbf{M}^{-1}\mathbf{K}_{eff})}{2\pi f} \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ \frac{-k_1 k_1}{m(k_1 + k_1)} & 0 \end{pmatrix}$$

$$\mathbf{B} = \begin{pmatrix} \mathbf{0} & \mathbf{0} \\ \text{Re}(-\mathbf{M}^{-1}\mathbf{C}_{XS(eff)}) & \mathbf{M}^{-1} \end{pmatrix} = \begin{pmatrix} 0 & 0 \\ \frac{-k_1 k_1}{m(k_1 + k_1)} & \frac{1}{m} \end{pmatrix}$$

$$\mathbf{C} = \begin{pmatrix} \mathbf{I} & \mathbf{0} \\ \text{Re}(-\mathbf{C}_{SX(eff)}) & \mathbf{0} \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ \frac{k_1 k_2}{k_1 + k_2} & 0 \end{pmatrix}$$

$$\mathbf{D} = \begin{pmatrix} \mathbf{0} & \mathbf{0} \\ -\mathbf{S}_{eff} & \mathbf{0} \end{pmatrix} = \begin{pmatrix} 0 & 0 \\ \frac{-k_1 k_2}{k_1 + k_2} & 0 \end{pmatrix}$$

and the state space equations are

$$\begin{pmatrix} \dot{x} \\ \ddot{x} \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ \frac{-k_1 k_1}{m(k_1 + k_1)} & 0 \end{pmatrix} \begin{pmatrix} x \\ \dot{x} \end{pmatrix} + \begin{pmatrix} 0 & 0 \\ \frac{-k_1 k_1}{m(k_1 + k_1)} & \frac{1}{m} \end{pmatrix} \begin{pmatrix} s \\ f_x \end{pmatrix}$$

and

$$\begin{pmatrix} x \\ f_s \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ \frac{k_1 k_2}{k_1 + k_2} & 0 \end{pmatrix} \begin{pmatrix} x \\ \dot{x} \end{pmatrix} + \begin{pmatrix} 0 & 0 \\ \frac{-k_1 k_2}{k_1 + k_2} & 0 \end{pmatrix} \begin{pmatrix} s \\ f_x \end{pmatrix}$$

5 Implementation in software

The following sections will be easier to follow with reference to a local installation of the toolkit and at least one model. See Section 10.1ff for more detail on downloading and installing the software.

5.1 Directory structure

To minimize code duplication, the Mathematica code has been divided up logically and by directory structure into four classes: (i) the toolkit, (ii) models, (iii) cases, (iv) calculations.

The toolkit comprises the most generic code and is structured as a number of Mathematica “packages” in a toolkit directory, which needs to be on the Mathematica search path (`$Path`). See Section 10.2.1 for more on how these packages are installed and added to `$Path`.

A model is a specification of a particular type of suspension as a particular combination of elements supported by the toolkit. For example, the `QuadLite2Lateral` model has the structure of an aLIGO quad suspension, with 4 masses, 6 blades (each with lateral as well as vertical compliance), and 14 wires. A model definition notebook has a default set of numerical parameters, but the focus is on the generic structure. The directory where the model definition notebook lives is called the model directory.

A case is subordinate to a model. It is defined by a set of numerical parameters, which can be specified from scratch, or as overrides to the default set. The numerical parameters are specified in a case definition notebook, which lives in a case directory one or more levels below the model directory.

A calculation is subordinate to a case. It was separated out from the idea of a case because when sharing models and cases via an SVN, different users will want different versions of the calculation, with, for example, different plots or other analysis. A calculation is defined by a calculation notebook, which lives in a calculation directory within the case directory.

Execution starts in the calculation notebook, and a search is made upward in the directory structure to find first the case definition and then the model definition. See Section 10.2.5 for more detail on the flow of control.

5.2 Phases of the calculation

Because the calculation for the wire bending potential terms referred to in Section 3.2.1 above is time consuming, the calculation is divided into five stages, 0A, 0B, 1A, 1B, and 2, adding in extra terms gradually. Stage0A is quick to calculate and the results may be a usable approximation for some purposes.

Stage0A first compiles a list of the potential terms for gravity, springs, and the basic longitudinal stretching of wires. Each list element is a pair consisting of a totally symbolic expression for the potential term plus a damping tag symbol that represents the appropriate type of damping for that term. The damping tag allows the correct complex damping function to be applied later. By default, Stage 0A does not include potential terms for bending of wires, or the additional longitudinal stretch from bending, but these can be turned on if desired.

The individual terms are then combined into a single expression relevant at zero frequency by multiplying by the real part of the respective complex damping functions at DC, summing, and then substituting numerical values for all quantities except the “variables” (i.e., \mathbf{x}) and the “floats” (i.e., \mathbf{q}).

The equilibrium position is found by numerically minimizing the potential function, starting from a user-supplied approximation that is a required part of the specification of the model.

Each term separately then has numerical values substituted for all quantities except the “variables” (i.e., \mathbf{x}), the “floats” (i.e., \mathbf{q}) and (this time) the parameters (i.e., \mathbf{s}). Partial derivatives are taken with respect to all the variables, floats and parameters that appear in the term, at the equilibrium point previously found. This gives a matrix with the same structure as \mathbf{P} , but sparse, with contributions only for a single term with a single damping type. These per-term stiffness matrices are collected in a list, tagged with their damping types.

Stage 1A adds terms for torsion of wires. Stage 1B adds terms for bending of the wires near the attachment points. Stage 2 adds terms for the slight additional longitudinal stretching of the wires due to bending away from the straight line assumed in Stage 0.

5.3 Utility functions

To support the above calculations the toolkit provides a large number of utility functions which are common to both the triple and quad models:

- (i) geometry of points on rigid bodies subject to translations and rotations
- (ii) geometry of angular velocity and infinitesimal rotations
- (iii) manipulating variables, parameters and velocities
- (iv) manipulating lists of substitutions
- (v) properties of wires (length, stretching and bending potential energy etc)
- (vi) properties of springs

- (vii) creating the total potential for the system
- (viii) creating matrices of elastic constants (a.k.a., potential energy matrices)
- (ix) creating matrices of mass/MOI coefficients
- (x) processing and pretty-printing eigenvectors
- (xi) outputting useful intermediate results
- (xii) converting to state-space matrix format
- (xiii) computing transfer functions
- (xiv) computing thermal noise

5.4 Model specific utilities

As well as the specification of the model proper, the user needs to supply some utility functions that depend on the model in ways that are too indirect to allow them to take the model specification as a parameter or to be automatically generated. These include:

- (i) routines for drawing 3D pictures of the mode shapes
- (ii) vectors that specify force and displacement inputs of interest (e.g., “unit x-direction displacement input at the structure”)
- (iii) vectors that specify outputs of interest (e.g., “unit x-direction displacement output at the upper left OSEM position”)

6 Models of Particular Interest

6.1 Triple Model

The triple model is a three-stage pendulum with the structure of the usual GEO triple. From the ground it consists of

- (i) A support structure (taken to be immovable except for the purpose of transfer functions).
- (ii) Two “upper” blade springs.
- (iii) Two “upper” wires, one per upper blade spring.
- (iv) The “upper” mass, a.k.a., mass 1.
- (v) Two “lower” blade springs.
- (vi) Four “intermediate” wires, one per lower blade spring.
- (vii) The “intermediate” mass, a.k.a., mass 2.
- (viii) Four “lower” wires (or fibres).
- (ix) The optic, a.k.a., mass 3.

There have been three different versions of the triple model with slightly different modeling of the blade springs. See the discussion of the “lite” and “xtra-lite” families of models below. These were removed in two stages.

6.2 Quad Model

The quad model is a four stage pendulum with the structure of the conceptual design for the test mass suspension in Advanced LIGO. From the ground it consists of

- (i) A support structure (not modelled as movable except for the purpose of transfer functions).
- (ii) Two “upper” blade springs.
- (iii) Two “upper” wires, one per upper blade spring.
- (iv) The “top” mass, a.k.a. mass 0 or mass N (“N” = new, relative to the triple masses)
- (v) Two “intermediate” blade springs.
- (vi) Four wires, two per intermediate blade spring.
- (vii) The “upper” mass, a.k.a., mass 1.
- (viii) Two “lower” blade springs.
- (ix) Four “intermediate” wires, two per lower blade spring.
- (x) The “intermediate” mass, a.k.a., mass 2.
- (xi) Four “lower” wires (or fibres).
- (xii) The optic, a.k.a., mass 3.

Note a key difference in the quad relative to the triple: the quad has only two blades at levels below the top (intermediate and lower), with two wires attached to the tip of each blade spring, whereas with the triple there are four blades at the lower level, with one wire per spring throughout.

6.3 “Lite” and “Xtra-Lite” Versions of the Triple and Quad Models

In the earliest versions of both the triple and quad models, each blade spring was represented by a 6-DOF mass element connected to its base (the structure or one of the hanging masses) by a 6-DOF spring. This was purely for ease of implementation – particularly with early versions of the toolkit it was simpler to represent approximate one-dimensional motion with stiff potentials than geometric constraints. This increased the size of the matrices that had to be handled, but was not a problem as long as only frequency domain results were of interest. However when output from the models came to be used for time domain simulation, the calculation would bog down because high-frequency eigenmodes involving the blade tips required very fine time steps to resolve. Thus for each model (triple and quad) two simplified versions were created.

The “lite” version of each model invokes the appropriate geometry to constrain each blade tip to move with the object it was mounted to in five DOFs, leaving one independent DOF. This reduces the number of DOFs (and the size of the matrices) to 24 for the triple and 30 for the quad. However the frequency of the eigenmode of a blade tip in its working direction is still quite high (because the blade is compliant but the wire attached to it is not). Thus the lite models are still marginal for time domain simulations.

The “xtra-lite” versions go one step further and eliminate the blade tip objects entirely. The junction between the spring and the wire still has a coordinate associated with it, but it is a float coordinate with no associated mass or MOI. Since it has no inertia, it can be assumed to track the

point of force/torque equilibrium between the blade and wire as the other objects move, which allows it to be suppressed from the normal mode analysis.

6.4 “Lateral” Versions

In the course of hanging the controls prototype build of the quad suspension, it was discovered that lateral compliance of the blades in the upper masses has a significant effect on the fundamental pitch frequency, and can easily push it unstable. See T050255-07, T050172-00, and T050257-01. To allow for this, extra DOFs representing the lateral positions of the blades were added. The quad xtra-lite lateral model is now the preferred model for the quad suspension. It turns out that for the triple suspensions in aLIGO, the lateral compliance effect is not significant, so for a long time the plain xtra-lite version was used. However at the urging of Jeff Kissel a TripleLite2Lateral model was prepared and brought into service.

6.5 “V5” Violin Mode Versions

Several quad and triple versions have been extended by dividing the lowest wires into six segments and adding mass beads between the segments to approximate the distributed mass. This creates 5 violin modes in each wire and each direction of wire displacement (longitudinal/transverse to the optic axis). The first 3 of these are tolerably harmonic and can be used to study violin mode thermal noise and coupling effects. It would be a fairly simple matter of cut and paste to create models with additional beads (V10 or V20 etc),

7 Exploring the supplied cases

Each model comes with a number of predefined cases, and a good way to start becoming familiar with the software is to load one of them and try the plotting and analysis functions. Look in the case subdirectory (e.g., subdirectory `default` for the case with default values) and open the calculation notebook, which will have a name like `ASUS3ModelCalcDefault.nb`.

7.1 Computing cases from scratch versus using precomputed results

Because the calculation time can be quite long, the calculation notebook for each case is set up to save intermediate results in the corresponding `precomputed` subdirectory. If the switch `useprecomputed` defined at the beginning of the calculation notebook is set to `True`, the archived results are read back in instead of being recomputed from scratch. Each of the supplied cases comes with such a set of results. To load them, first check that the `precomputed` switch is indeed set to `True` in the Switches section and evaluate the whole notebook. All the key results are then in the workspace and various additional plots and analyses can be performed, following the existing examples.

7.2 Stages of the calculation

Because some parts of the calculation are time consuming and not always of interest, the calculation is divided up into stages as follows.

- Stage 0A: The potential terms for gravity, spring elements, and longitudinal wire-stretching are generated and a minimization is done to find the equilibrium position. (As of v2.5 of the toolkit, the terms for wire bending can be included in the equilibrium as well.) The kinetic energy is processed to produce the mass matrix, the potential terms are processed to produce stiffness

matrices and a preliminary normal mode calculation is done. The key results are `eigenvalues0A` (a list of eigenvalues in descending order of magnitude), `eigenvectors0A` (a list of eigenvectors in the corresponding order), and `Hz0A` (the eigenvalues converted to frequencies in Hz). This stage takes only a few minutes and gives a good approximation to the mode shapes and frequencies. However the stiffness matrices generated in this step do not correctly take into account dissipation dilution. Thus no damping, transfer function or thermal noise information is calculated as it would be invalid. Also the mode shapes and frequencies may be slightly inaccurate if one of the damping functions has a real part that is not 1 at zero frequency.

- Stage 0B: The potential terms for spring elements and longitudinal wire-stretching are reprocessed with the static forces arbitrarily set to zero. For wires, the unstretched length is temporarily set to the stretched length implied by the equilibrium position, and for springs, the preload is set to zero. The stiffness that is still present under these conditions is due to first order length changes in the wires and springs, and so contributes to damping and thermal noise. The stiffness that disappears relative to Stage 0A was due to the static force acting through a variable mechanical advantage¹. It is added to the gravity stiffness matrix (which then acquires a new interpretation as the stiffness matrix for lossless restoring forces). The resulting damping calculations are of limited interest because they don't include the wire bending elasticity but at least they validly reflect the losses from the sources of elasticity they do include. Key results for this stage are:
 - (i) `eom0`: the equation of motion function, which describes the dynamics of the pendulum. It is a function which accepts a numerical value of frequency and returns a complex matrix which converts a vector of generalized force inputs to a vector of generalized displacement outputs.
 - (ii) `coupling0`: the coupling function, which describes the coupling between the support and the pendulum. It accepts a numerical value of frequency and returns a complex matrix

¹ The canonical example of a restoring force due to a variable mechanical advantage is a simple one-wire pendulum. It is commonly said that a pendulum exhibits dissipation dilution because the restoring force is gravitational, but this is wrong. The gravitational force is always directly downwards and can't push anything sideways – the restoring force is actually the sideways component of the tension in the wire. Another way of describing this is to say that there is a mechanical advantage through which the tension is coupled to sideways force on the mass, and that this changes with the angle of the wire. Since there are no first order changes in tension and thus length, the wire can be quite lossy as a longitudinal spring without degrading the performance of the pendulum for small amplitude oscillations. The only way the wire can contribute to loss is through the damping associated with the small restoring force the wire contributes directly by resisting bending at the flexure point.

The fact that gravity is indeed lossless turns out to be a red herring. Any other method of putting a static tension on the wire that doesn't involve first order length changes will give comparable dissipation dilution. For example, a bead on the middle of a taut wire is effectively a pendulum in which gravity has been replaced by a second wire, and will also have good dissipation dilution. And in the same way, violin modes of taut wires show dissipation dilution because the restoring force is the sideways component of a tension.

which converts a vector of displacement inputs at the support to a vector of generalized forces at the pendulum.

- Stage 1A: The potential terms for torsion of the wires are processed and added in. The results are as for Stage 0, except that the symbol names end in “1”: `eigenvalues1A`, `eigenvectors1A`, `Hz1A`, `eom1A`, etc.
- Stage 1B: The potential terms for bending of the wires are processed and added in. This stage typically takes a few hours (hence the desirability of precomputed results!). The results are as for Stage 0, except that the symbol names end in “1”: `eigenvalues1`, `eigenvectors1`, `Hz1`, `eom1`, etc. Thermal noise plots from these results are good approximations.
- Stage 2: The potential terms for the slight additional longitudinal stretch due to bending of the wires are added in. The results are as for Stage 0, except that the symbols end in “2”: `eigenvalues2`, `eigenvectors2`, etc. This stage takes another few hours and is rarely worth doing because it is such a small effect.

The preceding order of calculation is enforced using the `Calculate[]` function. The function call `Calculate[symbol]` will compute (or recall from the precomputed directory) all results on which *symbol* depends and then *symbol* itself. The function call `Calculate[stage]` where *stage* is one of `Stage0A`, `Stage0B`, `Stage1` or `Stage2` will compute all the results for the given stage. Such calls have been placed strategically through the sample case notebooks to ensure that the necessary results are loaded before the example analysis commands are evaluated. More detail about the `Calculate[]` function is given in the section on the calculation of the model below.

7.3 Available listing and plotting commands

7.3.1 Frequencies

The frequencies are available in the variable `Hz0A` (or `Hz0` or `Hz1` or `Hz2`). Since the interesting low-frequency modes are last, it is convenient to use Mathematica’s syntax for counting from the end of an array: `Hz0A[[-2]]` is the second lowest frequency.

7.3.2 Mode shape tables

Eigenvectors can be found in `eigenvectors0A` (or `...0` or `...1` or `...2`). The order corresponds to that in `Hz0A`, so `eigenvectors0A[[-2]]` is the second lowest frequency eigenvector. Within each vector the order of the coefficients corresponds to that in `allvars - allvars[[-6]]` is `x3` (the x-coordinate of the optic) so `eigenvectors0A[[-2]][[-6]]` is the amplitude of `x3`.

Eigenvectors can also be printed in a neat table format using the function `pretty[]`. It is commonly helpful to use `Chop[]` to throw away insignificant values and to premultiply by the matrix `e2ni` to convert the eigenvector into laboratory yaw, pitch and roll coordinates:

```
pretty[Chop[e2ni.eigenvectors0A[[-2]], 10^-4]]
```

gives something like

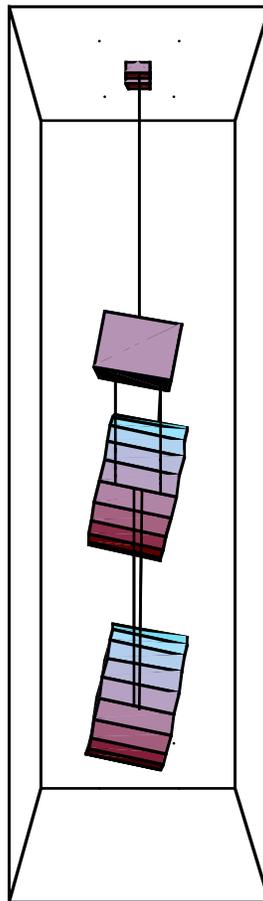
	x	y	z	yaw	pitch	roll
UL blade	0	0	0	0	0	0
UR blade	0	0	0	0	0	0
Mass U	-0.00134996	0	0	0	0.367617	0
LLF blade	-0.00150905	0	0.0118717	0	0.367617	0
LLB blade	-0.00150905	0	-0.0118717	0	0.367617	0
LRF blade	-0.00150905	0	0.0118717	0	0.367617	0
LRB blade	-0.00150905	0	-0.0118717	0	0.367617	0
Mass I	-0.00248634	0	0	0	0.395892	0
optic	-0.00446495	0	0	0	0.408604	0

7.3.3 Mode shape plots

You can get a 3D rendering of a normal mode with the `eigenplot[eigenvector, amplitude, {viewpoint}]` function. (The xtra-lite triple and quad models and any other models that define a non-zero number of float coordinates will have an extra `floatmatrix` argument in the final position. The amplitude parameter needs to be chosen by trial and error so that the mode shape is obvious but not distorted – values between 0.1 and 0.5 are commonly good. The viewpoint should be chosen to For example,

```
eigenplot[eigenvectors0A[[-2]], .5, {0, -2, 0}]
```

gives something like the following



7.3.4 Force transfer function plots

To calculate and plot transfer functions from generalized force on the pendulum to displacement, the following functions can be used:

```
calcTFf[eom, ivec, ovec, f]
plotTFf[eom, ivec, ovec, f1, f2]
```

where *eom* is *eom0* or the like, *ivec* is a vector of generalized force inputs, *ovec* is a vector of displacement outputs, and *f*, *f1* and *f2* are numeric frequencies. Force input basis vectors can be constructed conveniently using `makeinputvector[symbol]`, which returns a vector representing unit force or torque applied to the pendulum coordinate specified by *symbol*. (For example, `makeinputvector[yaw3]` is unit torque in yaw on the optic.) Similarly output basis vectors can be constructed using `makeoutputvector[symbol]`.

7.3.5 Displacement transfer function plots

To calculate and plot transfer functions from displacement of the support to displacement of the pendulum the following functions can be used:

```
calcTF[eom, ivec, ovec, f]
plotTF[eom, ivec, ovec, f1, f2]
```

where *eom* is one of the equation-of-motion functions (*eom0A* or the like), *cfn* is one of the coupling functions (*coupling0A* etc) *ivec* is a vector of generalized displacement inputs, *ovec* is a vector of displacement outputs, and *f*, *f1* and *f2* are numeric frequencies. Displacement input basis vectors can be constructed conveniently using `makeinputvector[symbol]`, which returns a vector representing unit force or torque applied to the support coordinate specified by *symbol*. (For example, `makeinputvector[x00]` is unit x displacement of the support.)

7.3.6 Thermal noise plots

To calculate and plot thermal noise the following functions can be used:

```
noise2[eomfn, iovec, f]
plotTN[eomfn, iovec, f1, f2, scale]
```

where *eomfn* is an equation-of-motion function, *iovec* is an input/output vector, *f*, *f1* and *f2* are numeric frequencies and *scale* is a scale factor. The input/output vector can be made with either `makeinputvector[]` or `makeoutputvector[]` (which actually do exactly the same thing). Its interpretation is a little bit subtle: it can be thought of as representing a ideal frictionless mechanism which causes a point to move as the specified linear combination of the coordinates. The thermal noise reported is the thermal noise of that notional output point of the mechanism. The temperature assumed accords with whatever substitution is given for the symbol *temperature* in `defaultvalues` or `overrides`.

7.4 Default numerical values

Each model comes with a set of default values for the properties of the various elements. These are given as Mathematica substitution rules in a list called `defaultvalues`, which for the triple model looks like this:

```
defaultvalues = {
```

```

g -> 9.8
ux -> 0.1,
uy -> 0.3,
uz -> 0.07,
den1 -> 2700,
m1 -> den1*ux*uy*uz,
...
};

```

The full, commented version of this list can be inspected in the model definition file. The calculation of the model using the default values can be found in the subdirectory `default`.

Often however the user will want to try variations on the default parameters. To promote good version control, the calculation of the model is done in a separate file from the definition. The recommended procedure is to leave the model definition notebook alone and specify variations in the calculation notebook by supplying a list of additional substitutions which override those from `defaultvalues`. The new substitutions should simply be added to the list `overrides` which is defined near the front of the calculation notebook. The next sections describe the mechanics of the process.

7.5 Creating new cases of the existing models

To create a new case with new numeric values for the properties of the pendulum, proceed as follows:

- (i) Duplicate one of the existing case subdirectories, such as `default`. Give the new subdirectory a name reflecting the combination of properties to be tried, such as `heavieroptic`.
- (ii) Rename the calculation notebook in the new subdirectory to end with some fragment of the subdirectory name, e.g., `ASUS3ModelCalCHO.nb`. (The exact name doesn't matter, but making it unique is helpful if you ever have several different notebooks open.)
- (iii) Delete all the files in the `precomputed` subdirectory within the new case directory.
- (iv) Find the assignment to `modelcase` in the Switches section of the calculation notebook and edit the RHS to be a string exactly the same as the case subdirectory name, e.g., `"heavieroptic"`.
- (v) Find the assignment to `overrides` in the Switches section and edit it so that it contains whatever overriding substitutions are required to implement the new case, e.g., `m3-> 50` will change the optic mass to 50 kg. (More detailed instructions are given below.) Add conspicuous comments there and at the top of the file explaining the change.
- (vi) Find the assignment to `precomputed` in the Switches section and edit the RHS to `False`.
- (vii) Evaluate the whole notebook.
- (viii) Edit the RHS of the assignment to `precomputed` back to `True`.

7.6 Using overrides

Note that although the ultimate goal is to give numeric values for all properties, you are not constrained to use only substitutions with numeric right-hand sides in `overrides`. After the old substitutions in `defaultvalues` and the new ones you provide in `overrides` are combined, they are put through a recursive process where substitutions are applied to the right-hand sides of each other until everything is (hopefully) numeric. (The resulting numeric substitution list is called `constval`.) All you need ensure is that any symbols that occur on the RHS of your substitutions also have substitutions defined for them in `overrides` or `defaultvalues`, and that any such recursive chains of substitutions end in numeric values after a finite number of steps. Thus

```
a->b, b->1
```

is good, but

```
a->b, b->a
```

will cause an infinite loop, and

```
a->b, b->c
```

will cause a failure when a numeric value for `a`, `b` or `c` is first required. Order is not important, so

```
b->1, a->b
```

is also good.

Note that the dependence is only one-way. Thus for example, by default, the upper mass `m1` and MOIs (`I1x`, `I1y` and `I1z`) are calculated in terms of the density (`den1`) and the length breadth and height (`ux`, `uy` and `uz`) using the usual expressions for a block. The formulae for mass and MOI can be overridden with specific numbers, say from a CAD program, and these new values will take effect in the normal mode calculation. However the connection with the density and size symbols is then broken so unless you give additional overrides they will retain their original, possibly inconsistent values.

You can also give overrides of the form

```
symbol -> scale[factor]
```

or

```
symbol -> increment[difference]
```

and instead of the substitution from `defaultvalues` being entirely replaced it will be scaled or incremented by *factor* or *difference* respectively. For example,

```
m3->scale[2.0]
```

will double the mass of the optic.

8 The definition of the model

To make more ambitious changes that can't be implemented with overrides (such as asymmetric models) or to make entirely new models requires a more detailed knowledge of the definition and calculation code. This section gives a walkthrough of the code in the definition notebook, with particular emphasis on the items that the model writer needs to supply to define a new model.

8.1 Name, Description and Version

The model is identified by assigning appropriate values to the following:

modelname	Official name of model, e.g., "QuadLite2Lateral"
modelcomment	Description, e.g., "GEO-style quad pendulum with massless blades with z and x DOFs"
modelversion	Numeric version, e.g., 5.9.

Code in the calculation notebooks uses these values in captions and headings. Code in the case definition notebooks checks the values of these symbols to ensure that the correct model has been loaded. For each new version, a list of changes since the last version should be added as a text cell.

8.2 Preliminaries

The Preliminaries section loads the toolkit package and any other packages that may be necessary. Most models will require all of the following:

- `PendUtil`` (the pendulum toolkit utilities)
- `RotationsXYZ`` (custom package for rotations in yaw, pitch and roll)
- `StatusWindow`` (custom package to display text messages in a status window)
- `IFOModel`` (parameters from Bench)

Most existing models also load the following additional packages if the version of Mathematica is 5 or before, but this is such ancient history that new models need not:

- `MyShapes`` (custom package similar to `Graphics`Shapes`` but with rotation bug fixed)
- `Graphics`Graphics`` (standard package for general graphics)
- `Graphics`Polyhedra`` (standard package for drawing polyhedra)
- `LinearAlgebra`MatrixManipulation`` (standard package for matrix manipulation)
-

8.3 The model definition proper

To define a model, a user needs to supply definitions for the symbols outlined in the next sections. Examples are from the triple model, `ASUS3ModelDefn.nb` but the quad model is very similar.

8.3.1 The master variable list: `allvars`

The symbol `allvars` corresponds to the vector \mathbf{x} in the theory section above. It should be a list of all the "variables" in the special sense of the coordinates defining the state of the pendulum for the purpose of the normal mode calculation. That order of the variables is arbitrary but should be logical and easy to remember. The variable list for the triple model is

```
allvars = {
  xul, yul, zul, yawul, pitchul, rollul,
  xur, yur, zur, yawur, pitchur, rollur,
```

```

x1,y1,z1,yaw1,pitch1,roll1,
x1lf,y1lf,z1lf,yaw1lf,pitch1lf,roll1lf
x1lb,y1lb,z1lb,yaw1lb,pitch1lb,roll1lb,
x1rf,y1rf,z1rf,yaw1rf,pitch1rf,roll1rf,
x1rb,y1rb,z1rb,yaw1rb,pitch1rb,roll1rb,
x2,y2,z2,yaw2,pitch2,roll2,
x3,y3,z3,yaw3,pitch3,roll3
};

```

8.3.2 The list of “floats”

The symbol `allfloats` corresponds to the vector \mathbf{q} in the theory section above. It should be list of the coordinates of any junctions between elastic elements that have no mass or MOI associated with them. If it is not defined it defaults to the empty list. There are no floats defined in the full triple or quad models – to see an example, look at one of the “xtralite” models.

8.3.3 The parameter list: `allparams`

The symbol `allparams` corresponds to the vector \mathbf{s} in the theory section above, i.e., a list of all the “parameters”, in the special sense of the coordinates that do not vary for the purposes of the normal mode calculation, but do vary for the purposes of displacement-input transfer functions. For the current models these are the 6 coordinates of the structure, but you can add the coordinates of any other object in the environment the effect of whose displacement on the pendulum you might conceivably want to calculate. The parameter list for the triple model is

```

allparams = {
    x00, y00, z00, yaw00, pitch00, roll00
};

```

8.3.4 Coordinate lists for rigid bodies and points on them

To make things more readable later on, it is convenient to define names for the lists of coordinates that define each rigid body. These need to conform to the format expected by the geometry functions: six coordinates in x/y/z/yaw/pitch/roll order. For example, the optic in the triple model is

```

optic = {x3, y3, z3, yaw3, pitch3, roll3};

```

The items in the list don’t need to be single symbols from `allvars`, they just need to be in terms of such symbols plus constants from `defaultvalues`. For example,

```

constrainedbody = {1 Sin[theta], 1 Cos[theta], 0, 0, 0, 0};

```

could represent a rigid body constrained to move without rotation along a line at a constant angle θ to the x-axis. (The coordinate 1 would have to appear in `allvars`, and the constant `theta` in `defaultvalues`.)

Typically the user will also want to define coordinate lists for static objects. For example, in the triple model the support structure is

```

support = {x00, y00, z00, yaw00, pitch00, roll00};

```

(This happens to be the same list as `allparams`, but is conceptually different in that the order and interpretation of items in `allparams` is not important, whereas here it must be `x/y/z/yaw/pitch/roll`.)

Similarly, points of interest on rigid bodies should be specified as lists of `x`, `y` and `z` local coordinates. For example in the triple pendulum, the left wire-attachment point on the upper mass is

```
massU1={0,-n1,d0};
```

8.3.5 Items with gravitational potential energy: `gravlist`

All the potential terms due to gravity should be grouped in a list called `gravlist`. The following fragment from the triple model initializes the list and adds a term for the potential energy of the optic to it:

```
gravlist = {};
AppendTo[gravlist, m3 g z3];
```

(Building up a list with `AppendTo []` is overkill for something so simple but it's good practice for the more complicated structures to come. There it makes things rather more readable.)

8.3.6 Wires: `wirelist`

The various wires in the model are specified in a list called `wirelist`. Each entry in `wirelist` describes a single wire and has the following format:

```
{
  6-coordinate list defining first mass,
  3-coordinate attachment point for first mass (local coordinates),
  3-coordinate attachment vector for first mass,
  6-coordinate list defining second mass,
  3-coordinate attachment point for second mass (local coordinates),
  3-coordinate attachment vector for second mass,
  Young's modulus,
  unstretched length,
  longitudinal elasticity,
  vector defining principal axis 1,
  moment of area along principal axis 1,
  moment of area along principal axis 2,
  linear elasticity type,
  angular elasticity type,
  torsional elasticity type,
  shear modulus,
  cross sectional area for torsional calculations,
  torsional stiffness geometric factor
}
```

Each wire is presumed to be strung between two rigid bodies, and items 1 and 4 are the coordinate lists specifying the bodies. (Each body can be either massive and have a coordinate list of variables drawn from `allvars` or massless and have a coordinate list drawn from `allfloats`.) Items 2 and 5 are the attachment points in body coordinates. Items 3 and 6 are vectors in local coordinates that specify the angle at which the wires are attached. For example, for the wire in a simple pendulum, the vectors would be $\{0, 0, -1\}$ (i.e., down) at the top and $\{0, 0, 1\}$ (i.e., up) at the bottom. However as a convenience, if you put anything but a three-item list in either of those positions, the later calculation ignores it, and instead works out what the vector would have to be for the wire not to be bent at the end when the pendulum comes to equilibrium.

Items 7 and 8 are the Young's modulus for the wire and the unstretched length.

Item 9 is the elasticity of the wire considered as a longitudinal spring. The reason for having the user work this out manually, rather than taking the cross-sectional area as a parameter, is to allow for the case where the area varies along the length of a fibre.

Items 10-12 relate to the bending of the wire near the endpoints. The wire is allowed to have an oblong cross-section with different rigidities along different axes. Item 10 is a vector in local coordinates specifying the axis along which the rigidity is maximum, Item 11 is the moment of area along that axis, and Item 12 is the moment of area along the axis at right angles.

Items 13, 14 and 15 are identifying symbols that specify which damping functions are to be attributed to the longitudinal, bending and torsional elasticities respectively. (The damping function for each tag should be specified as a substitution in `defaultvalues` – see below. If none is given it defaults to 1 for the real part and 0 for the imaginary part.)

Items 16, 17 and 18 are the shear modulus, the cross-sectional area (for the purposes of the torsion calculation only) and the torsional stiffness geometric factor. For backward compatibility for models defined before torsion was supported, the four torsion-related items (15, 16-18) are optional on a per-wire basis.

In the triple model, initializing `wirelist` and adding a definition for the wire from the top left blade spring to the upper mass looks like this:

```
wirelist = {};
AppendTo[wirelist, {
  bladeUL,
  bladeULa,
  bladeULavec,
  massU,
  massU1,
  massU1vec,
  Y1,
  u11,
  kw1,
  {1, 0, 0},
  M11,
  M12,
```

```

        wireUtype,
        wireUatype
    }];

```

8.3.7 Springs: `springlist`

The elastic elements in the model other than wires are specified in `springlist`. Note that a “spring” element only models elasticity. To model the mass of a physical spring as well requires a separate rigid body element.)

Each entry in `springlist` describes a single spring element and has the following format:

```

{
    coordinate list defining first mass,
    attachment point for first mass (local coordinates),
    attachment angles for first mass (yaw, pitch, roll),
    coordinate list defining second mass,
    attachment point for second mass (local coordinates),
    attachment angles for second mass (yaw, pitch, roll),
    damping type,
    6x6 elasticity matrix,
    1*6 pre-load force/torque vector
}

```

Items 1–6 are similar to Items 1-6 in the list of wire properties and specify the two objects that the spring is attached to. There is one important difference however: for wires, the attachment angle is specified as a vector, whereas for a spring it is specified as a triple of angles: yaw, pitch and roll. (The three-angle formulation is more general and elegant but it turned out to make the computations for the wire-bending elasticity intractable.) The spring is considered to have its own x/y/z coordinate system, and the attachment angles are the amount that one would have to rotate the spring to reach its working position, relative to an initial state with its coordinate system aligned with that of the mass.

Item 7 is a symbol specifying a damping function, as for the wires.

Item 8 is a 6x6 elasticity matrix giving elastic constants with respect to differential displacements in the spring local coordinate system.

Item 9 is a vector giving the amount of preload force or torque in the spring x, y, z, yaw, pitch and roll directions when the attachment points on the two masses are coincident and aligned according to the attachment angles.

The following fragment from the triple model shows the entry for the upper left blade spring being added to the list:

```

springlist = {};
AppendTo[springlist, {
    support,

```

```

bladeULnom,
{0,pitchbul,rollbul},
bladeUL,
COM,
{0,pitchbul,rollbul},
bladeUtype,
DiagonalMatrix[{kbux,kbuy,kbuz,kbyawu,kbpitchu,kbrollu}],
{0,0,bdu*kbuz,0,0,0}
}];

```

8.3.8 The kinetic energy: kinetic

To make the specification of the kinetic energy more readable it is convenient to group the moment-of-inertia constants into tensors, like the following for the optic in the triple model:

```
IM3 = {{I3x, 0, 0}, {0, I3y, 0}, {0, 0, I3z}};
```

The kinetic energy should then be given as a expression in terms of Mathematica total derivatives of the variables (from `allvars`, *not* `allfloats`) with respect to `t` (time), and assigned to the symbol `kinetic`. This is admittedly not very user-friendly and really ought to be automated in some future version. For the moment, just copy the pattern of the following fragment showing the term for the kinetic energy of the upper mass (`massU`) in the triple model:

```

kinetic = (
...
+(1/2) m3 Plus@@(Dt[b2s[optic,COM],t]^2)
+(1/2) omegaB[yaw3, pitch3, roll3].IM3.omegaB[yaw3, pitch3, roll3]
...
);

```

Here, `Plus@@(vector)^2` is a Mathematica shortcut for summing over squares, `Dt[coordinate, t]` is the total time derivative, `b2s[object, point]` (a function from the package `RotationsXYZ``) is the coordinates of a local point on an object, `COM` (defined previously as `{0,0,0}`) is the centre of mass in local coordinates, and `omegaB[]` (also from `RotationsXYZ``) is the angular velocity vector.

8.3.9 Values of constants: defaultvalues

As noted previously, `defaultvalues` is a list of substitutions for properties of the pendulum. Specifically, it should contain substitutions for everything that isn't a "variable" (a coordinate involved in the normal modes). As well as the properties of the pendulum per se, it should also contain some or all of the following special items. First it needs to contain substitutions for the following constants:

- `g` (local gravitational acceleration)
- `temperature` (temperature, for the thermal noise calculations)
- `boltzmann` (Boltzmann's constant, for the thermal noise calculations)

Second, it should contain substitutions for the usual static values of the “parameters” (i.e., the coordinates of the structure or other objects involved only in transfer functions).

Third, it should contain the function call `constraintsubstitutions[]`, which expands to a list of substitutions of the form `kconvariablename -> 0`, one for each variable in `allvars`. Each of the generated constants represents an elastic force tying the corresponding DOF to mechanical ground. Additional potential terms corresponding to these constants are automatically generated in the calculation notebook, but since the constants are zero by default, they normally have no effect. However for debugging purposes, you may find it convenient to override particular constraint elasticities with large values to immobilize particular parts of the pendulum.

Fourth, it may also contain substitutions of the form

```
damping[real,dampingtype] -> (function&)
```

and/or

```
damping[imag,dampingtype] -> (function&)
```

which define the frequency dependence of the elasticity and damping for potential terms of the specified *dampingtype*. The `&` operator is Mathematica’s syntax for a so-called pure function, which is a object that represents a function operation but lacks a name. The arguments of a pure function are represented by `#1`, `#2` etc. The parentheses around the pure function are required here because `&` has a lower precedence than `->`. The functions you supply should accept a value of frequency in Hz (*not* rad/s) and return a multiplier giving the real or imaginary component of the complex elasticity as a fraction of the purely real elasticity defined by the potential. If you don’t supply one or both substitutions for a particular damping type they default to

```
damping[real,dampingtype] -> (1&)
```

and

```
damping[imag,dampingtype] -> (0&)
```

i.e., frequency-independent elasticity with no damping. Unless you’re really paranoid about the Kramers-Kronig relationship or working with very large loss angles you can normally rely on the default for the real part and specify only the imaginary one. Structural damping can be specified by

```
damping[imag,dampingtype] -> (phi&)
```

and thermoelastic damping would be

```
damping[imag,dampingtype] ->
(delta*(2*N[Pi]*#1*tau)/(1+(2*N[Pi]*#1*tau)^2)&)
```

where `delta` and `tau` are defined elsewhere in `defaultvalues`.

8.3.10 Approximation to equilibrium position: `startpos`

The list `startpos` should be a list of substitutions for the variables in `allvars` giving an approximate equilibrium state for the system. This is used as a starting point for finding the exact equilibrium by numerical minimization of the potential. The substitutions can have symbols on the right hand sides provided of course that there are substitutions for those symbols in `defaultvalues`.

8.3.11 Extra potential terms for debugging: `preeqterm`list and `posteqterm`list

If desired `preeqterm`list can be defined in the model case notebook as a list of `{potentialterm, dampingtype}` pairs to modify the default structure of the model. Any such terms are added to the main potential before the equilibrium position is found. The potential expressions in `preeqterm`list will be evaluated twice, once with `tensionoffswitch->False` and once with `tensionoffswitch->True`. If a particular term has a component representing a static tension, it should be zeroed out in the latter case.

If desired `posteqterm`list can be defined in the model case notebook as a list of `{potentialterm, dampingtype}` pairs to modify the default structure of the model. Any such terms are added to the main potential after the equilibrium position is found, as with the constraint forces. Any static forces they represent must cancel for the equilibrium position to be valid. The equilibrium position may be invoked with expressions like `allvars[[i]]/.optval/.nonoptval`. To prevent error messages from `optval` and `nonoptval` being evaluated before they are calculated, `posteqterm`list should normally be defined with the `:=` operator.

The potential expressions in `preeqterm`list and `posteqterm`list will be evaluated twice, once with `tensionoffswitch->False` and once with `tensionoffswitch->True`. If a particular term has a component representing a static tension, it should be zeroed out in the latter case.

8.4 Model dependent diagnostic utilities

The model writer is also responsible for providing custom versions of the utilities described in this section, which are not necessary for the basic recalculation of the model, but are still highly desirable for analyzing the resulting data.

8.4.1 Angular velocity transformation matrices: `e2s`, `e2b` and `e2ni`

The coefficients corresponding to the angle variables in the raw eigenvectors are not in a very convenient basis. In accordance with the normal mode formalism, they represent infinitesimal amounts by which the yaw, pitch, and roll angles at equilibrium are incremented in normal mode motion. However because of non-linearity and non-commutativity issues, the incremental rotation in going from (yaw, pitch, roll) to (yaw+dyaw, pitch+dpitch, roll+droll) is not the same as (dyaw, dpitch, droll) (except for yaw=pitch=roll=0). Worse, the three individual rotations implied in (yaw+dyaw, pitch+dpitch, roll+droll) aren't about mutually orthogonal axes (again, except when yaw=pitch=roll=0). The toolkit provides transformation matrices to convert the eigenvector coefficients into a variety of more useful forms. "Space" coordinates use a basis of infinitesimal rotations about the global or space x, y and z axes (in that order) and are useful if you want to interpret normal mode motion as an angular velocity. The 3x3 matrix function `omegaIS[]` transforms to that basis:

```
omegaIS[yaw,pitch,roll].{dyaw, dpitch, droll}
```

"Body" coordinates use a basis of infinitesimal rotations about the body x, y and z axes for the object and are useful in conjunction with the MOI tensor (which is implicitly in body coordinates). These can be calculated using `omegaIB[]`:

```
omegaIB[yaw,pitch,roll].{dyaw, dpitch, droll}
```

(Finally non-incremental yaw/pitch/roll coordinates use a basis of infinitesimal rotations in yaw, pitch and roll. Since this amounts to a basis of infinitesimal rotations about the z, y and x axes, for the object they are effectively just the space coordinates permuted:

```
omegaSNI.omegaIS[yaw,pitch,roll].{dyaw, dpitch, droll}
```

where

```
omegaSNI = {{0,0,1},{0,1,0},{1,0,0}}
```

However since these provided utilities only apply to the angle variables of one object at a time, and the toolkit has no way to know which variables in `allvars` are for angles, the user is responsible for creating matrices (`e2b`, `e2s` and `e2ni`) which process a whole eigenvector at once, along the following lines:

```
e2ni := DiagonalBlockMatrix[{
  IdentityMatrix[3],
  omegaSNI.omegaIS[yawul,pitchul,rollul],
  IdentityMatrix[3],
  omegaSNI.omegaIS[yawur,pitchur,rollur],
  IdentityMatrix[3],
  omegaSNI.omegaIS[yawl,pitchl,rolll],
  IdentityMatrix[3],
  omegaSNI.omegaIS[yawllf,pitchllf,rollllf],
  IdentityMatrix[3],
  omegaSNI.omegaIS[yawllb,pitchllb,rollllb],
  IdentityMatrix[3],
  omegaSNI.omegaIS[yawlrf,pitchlrf,rolllrf],
  IdentityMatrix[3],
  omegaSNI.omegaIS[yawlrp,pitchlrp,rolllrp],
  IdentityMatrix[3],
  omegaSNI.omegaIS[yaw2,pitch2,roll2],
  IdentityMatrix[3],
  omegaSNI.omegaIS[yaw3,pitch3,roll3]
}]/.optval;
```

8.4.2 Plotting routines: `eigenplot[]`

To enable drawing 3D pictures of the normal modes there should be a custom version of the routine `eigenplot[eigenvector, amplitude, {viewpoint}]`. The versions in the existing models can be used as a pattern. They use routines from the custom package `MyShapes``, which is similar to the standard package `Graphics`Shapes`` but with a few refinements for precisely this application. The toolkit function `tosub[eigenvector]` is provided to convert the displacements relative to the equilibrium position given by the eigenvector to absolute values in the form of a list of substitutions. Since eigenvectors only directly specify the behaviour of variables, if there are any floats defined and there are objects that need to be plotted that depend on them, `eigenplot[]` will need to have an extra argument `floatmatrix` giving the conversion

between variables and floats. A call to `tosub[eigenvector, floatmatrix]` then generates substitutions for floats as well as variable.

Note that eigenvectors should *not* be processed with `e2ni` before being given to `eigenplot[]`.

8.4.3 Listing routines: `pretty[]`

To present eigenvectors in a useful manner there should be a custom version of the routine `pretty[eigenvector]` to list the coefficients in the eigenvector in a table format with headings. It is commonly convenient to process eigenvectors with `e2ni` before listing them, but this should be left up to the end-user rather than being done automatically in `pretty[]`.

8.4.4 Input and output vectors

To enable basic use of the various transfer function routines there should be a set of input and output vectors for the coordinates likely to be of interest such as those of the structure and the final payload. Toolkit functions `makeinputvector[parameter]`, `makeinputvector[variable]` and `makeoutputvector[variable]` are supplied to make this easy.

9 The calculation of the model

9.1 The `Calculate[]` function

As explained above, the `Calculate[]` function is normally used to compute or load results for a whole stage at once. However it works on any symbol representing one of the standard intermediate or final results. A typical clause in the definition of `Calculate[]` looks like this:

```
Calculate[symbol] := (
    Calculate[dependencies];
    If[!useprecomputed || exceptdamping,
        Status["Computing symbol"];
        code to compute symbol;
        saveprecomputed["symbol.m", symbol];
        Done[],
    (*else*)
        getprecomputed["symbol.m"];
    ];
)
```

Note the following features:

- Before attempting to compute or load the requested symbol, `Calculate[]` calls itself recursively to compute or load all the symbols on which the requested one depends.
- If the switch `useprecomputed` is `True`, `Calculate[]` attempts to load an archived value for the symbol from the `precomputed` subdirectory within the case directory.
- The condition “`|| exceptdamping`” is omitted in the clauses for result symbols that do not depend on the frequency dependence of elasticity, in particular the potential matrices. Thus by

setting both `useprecomputed` and `exceptdamping` to `True`, one can explore the effects of different magnitudes and frequency dependences of damping without wasteful recalculation.

The effect of the recursion is that intermediate results are computed in the order described in the next few sections. When debugging a new model it will be helpful to work through the result symbols in that order, calling `Calculate[]` manually on each one in turn so that stages with errors can be identified. Note that `Calculate[]` has no effect on symbols that have already been loaded. To recalculate a result after an error has been corrected, call `Clear[symbol]`; `Calculate[symbol]`. Alternatively, clear the results for a whole stage at once with `Reset[stage]` (where *stage* is one of `Stage0A` etc).

9.2 Stage 0A – normal modes without wire bending elasticity

9.2.1 Numerical Substitutions: `constval`

The seminumeric substitutions in `defaultvalues` and `overrides` are merged using `Override[]` and then processed using `Recurse[]` to propagate the numeric values throughout, producing a list called `constval`.

9.2.2 Numerical start position for minimization of the potential: `startval`

The starting position `startpos` suggested by the model writer is converted to fully numeric form called `startval` by processing with `constval`.

9.2.3 Coordinates that don't participate in the minimization: `nonoptcoords` and `nonoptval`

The minimization of the potential should normally be done with respect to all the variables and floats in `allvars` and `allfloats`. However as an aid to debugging, all those in the list `nonoptcoords` are excluded. This is useful in isolating problems in the specification of the wires and springs. The variables in it are pegged to the values in `startval` during the minimization. The list `nonoptval` is a subset of `startval` with substitutions for just the variables in `nonoptcoords`.

9.2.4 Variables that participate in the minimization: `optcoords`

The list `optcoords` is set to the complement of `nonoptcoords` with respect to `allvars` and `allfloats`. (For backward compatibility, `nonoptvars` is also accepted but this is deprecated.)

9.2.5 The base list of potential terms for the minimization: `potentialtermlist`

The default set of potential terms that are to be considered in the minimization are assembled into a list together with their damping tags:

```
potentialtermlist = {{term,tag},{term,tag},...}
```

This includes all the terms generated from `gravlist` and `springlist`, the terms for simple longitudinal extension of the wire from `wirelist`, plus any extra terms defined manually in `preeqtermlist`. This was the originally the full list, but as of v2.5 of the toolkit, it can be supplemented with optional extra wire terms as discussed in the next section.

9.2.6 Additional wire terms for the minimization: `potentialtermlistWBnr`, `potentialtermlistWTnr`, `potentialtermlistWEnr`

As of v2.5 of the toolkit, if `wiretermsearly` is defined to be `True` or `EqOnly`, either directly or as a substitution in `constval`, then any wire potential terms for bending or torsion that can be calculated are accumulated in `potentialtermlistWBnr`, `potentialtermlistWTnr` or `potentialtermlistWEnr` (B = bending, T = torsion, E = extra longitudinal extension).

The reason that some may not be able to be calculated is the feature whereby wire attachment angles specified as dummy symbols are calculated after the minimization such that the associated wires are straight at equilibrium (see 9.4.1). Bending terms can only be calculated for endpoints where the wire attachment angles are given explicitly. Torsion terms can only be calculated for wires where the wire attachment angles for both endpoints are given explicitly. Any terms not meeting these requirements are deferred to Stages 1 or 2 as of old.

9.2.7 The full expression for the potential: `potential`

The terms in `potentialtermlist` are combined with any in `potentialtermlistWBnr`, `potentialtermlistWTnr` and `potentialtermlistWEnr`, multiplied by the real part of the damping multiplier evaluated at $f=0$ (`damping[real, dampingtype][0]`) and summed to produce the total potential as a single expression.

9.2.8 The numeric potential: `potentialNN`

The symbolic expression for the potential is processed with `constval` and `nonoptval` to produce `potentialNN`, an expression containing only the coordinates in `optcoords`.

9.2.9 The minimization: `potential0` and `optval`

The minimization gives `potential0`, the value of the potential at the minimum, and `optvals`, a specification of the minimum as a list of substitutions for the variables in `optcoords`.

9.2.10 Velocity variables: `velocities`

A list of velocity names is created by adding “v” to all the variable names in `allvars`.

9.2.11 The kinetic energy matrix: `kineticN` and `kineticmatrix`

The expression `kinetic` for the kinetic energy is given in terms of symbolic constants, variables and total time derivatives of variables. The time derivatives are replaced by velocity variables, and `constval`, `optval` and `nonoptval` are used to remove all other symbols, producing `kineticN`. Partial derivatives with respect to all pairs of variables in `velocities` are then taken to produce `kineticmatrix`.

9.2.12 The potential used for the normal mode calculation: `potentialtermlist0`

Extra items of the form `{constraintterm, constrainttype}` are added to `potentialtermlist` to produce `potentialtermlist0`. The extra terms represent elastic forces tying the various DOFs to the equilibrium position determined in the previous step. Terms defined manually in `posteqtermlist` are also added at this point.

As of toolkit v2.5, if `wiretermsearly` is defined to be `True` (but *not* `EqOnly`), either directly or as a substitution in `constval`, any terms in `potentialtermlistWBnr`, `potentialtermlistWTnr` or `potentialtermlistWEnr` are appended.

9.2.13 The Stage 0A damping-specific potential matrices: `potentialmatrices0T`

The potential term parts of all the items in `potentialtermlist0` are differentiated with respect to the variables, floats and parameters to form individual damping-specific potential matrices corresponding to the T_i . Matrices with the same damping tags are combined. The final list `potentialmatrices0T` has the form $\{\{dampingtype, matrix\}, \{dampingtype, matrix\}, \dots\}$. (For no particularly good reason the tag comes first here rather than second as above.) As explained above, the potential matrices at this stage should be treated with caution because they have not yet been corrected for dissipation dilution.

9.2.14 The Stage 0 potential matrix: `potentialmatrix0`

The individual potential matrices in `potentialmatrices0T` are multiplied by the real part of the complex elasticity multiplier evaluated at $f=0$ (`damping[real, dampingtype][0]`) and summed to produce a net stiffness matrix.

9.2.15 The Stage 0 normal modes: `eigenvalues0`, `eigenvectors0` and `Hz0`

The potential and kinetic energy matrices are simultaneously diagonalized to produce `eigenvalues0` and `eigenvectors0`. The eigenvalues, which are in units of $(\text{rad/s})^2$ are then scaled to units of Hz to produce `Hz0`.

9.3 Stage 0B – damping without wire bending elasticity

9.3.1 Potential matrices without tension: `potentialmatrices0NT`

The entire potential matrix calculation from stage 0A is repeated with the wire tension and spring preload set to zero, producing `potentialmatrices0NT`. This and `potentialmatrices0T` are passed to `dissipationdilution[]` which correlates them and identifies the components of elasticity that produce damping to produce a new list `potentialmatrices0`.

9.3.2 The equations of motion function and the coupling function: `eom0` and `coupling0`

These are calculated from `kineticmatrix` and `potentialmatrices0`. Thus they neglect wire bending elasticity but correctly incorporate dissipation dilution on the remaining sources of damping.

9.4 Preparation for Stages 1 and 2

9.4.1 Wire angles: `relaxval`

As noted above, the wire definitions are allowed to have placeholder symbols for wire attachment angles. The definitions are scanned for such symbols and the optimum attachment angles are calculated taking into account the equilibrium position of the pendulum. The values are assembled into a substitution list `relaxval`.

9.4.2 Additional potential term lists: `potentialtermlistWB` and `potentialtermlistWE`

In preparation for Stages 1 and 2, new potential terms describing the wire bending are generated and stored in the lists `potentialtermlistWB` and `potentialtermlistWE` in the same `{{term,tag},...}` format as for `potentialtermlist0`. The first list, used in Stage 1 describes the potential from pure angular bending of the wire. The second list, used in Stage 2, describes the potential from the additional longitudinal stretch of the wire due to the fact that it has been displaced away from the straight line between the endpoints assumed in Stage 0.

The elasticity theory used in the wire bending phase of the calculation was derived by Mark Barton using Mathematica based on the discussion in Matt Husman's thesis. The function `wirebendingPE[T,l,EE,I1,I2,alpha1,beta1,alpha2,beta2]` computes the bending part of the potential energy for a wire with tension `T`, Young's modulus `EE`, moments of area `I1` and `I2` along its principal axes, which is stretched between two points `l` apart on a line and makes angles `alpha1` and `alpha2` with the line at one end and `beta1` and `beta2` at the other end.

Similarly, `wirebendingdelta[T,l,EE,I1,I2,alpha1,beta1,alpha2,beta2]` computes the extra longitudinal extension of the wire due to the curvature of the wire.

The code for wire torsion was kindly provided by Ben Lee of the UWA Gravity Group.

The functions `wirepotential1[]`, `wirepotential1T[]` and `wirepotential2[]` do the 3D geometry required to apply the above functions to wires at the angles implied by the equilibrium position of the model.

The tags associated with the new terms are taken from the entries in `wirelist`. Those in `potentialtermlistWB` get the tag specified for bending; those in `potentialtermlistWE` get the tag specified for stretching.

9.5 Stage 1A – normal modes and damping with wire bending elasticity

The Stage 1A calculation processes the entries in `potentialtermlistWB` representing wire lateral bending to produce `potentialmatricesWB`. This is then merged with `potentialmatrices0` to produce `potentialmatrices1A`. In turn, this processed in the same way as for Stage 0 to produce `potentialmatrix1A`, `eom1A`, `coupling1A`, `eigenvalues1A`, `eigenvectors1A` and `Hz1A`.

9.6 Stage 1B – normal modes and damping with wire torsional elasticity

The Stage 1B calculation processes the entries in `potentialtermlistWT` representing torsion, to produce `potentialmatricesWT`. This is then merged with `potentialmatrices1A` to produce `potentialmatrices1`. In turn, this processed in the same way as previously to produce `potentialmatrix1`, `eom1`, `coupling1`, `eigenvalues1`, `eigenvectors1` and `Hz1`.

9.7 Stage 2 – normal modes and damping with additional wire stretch due to bending

The Stage 2 calculation processes the entries in `potentialtermlistWE` to produce `potentialmatricesWE`. This is then merged with `potentialmatrices1` to produce `potentialmatrices2`. In turn, this is processed in the same way as for Stages 0 and 1 to produce `potentialmatrix2`, `eom2`, `coupling2`, `eigenvalues2`, `eigenvectors2` and `Hz2`.

9.8 Exporting state-space matrices

9.8.1 Building state-space matrices with model results

The package `PendUtil.nb` defines a number of functions for constructing the A, B, C and D matrices of the state space formalism. There are quite a few variants for different purposes, with different numbers of inputs and outputs but the names typically start with `ssA`, `ssB`, `ssC` and `ssD`.

The usual inputs are a group of displacement inputs corresponding to the structure coordinates in `allparams`, followed by a group of force/torque inputs corresponding to the variables in `allvars`. The standard minimal set of outputs are displacements corresponding to the variables in `allvars`. Variant `ssC` and `ssD` matrix functions add (at the end) outputs for the forces/torques on the structure corresponding to the coordinates in `allparams`. This expanded set is useful for coupling the state spaces of two different models and is illustrated in the diagram on the next page. The block structure of all four matrices is annotated with the physical interpretation of each block and how it is constructed using the quantities described in Section 3.1.

9.8.2 Exporting state-space matrices

The packages `MatlabExport.nb` and `E2EExport.nb` contain utility functions for outputting selected quantities to text files readable by Matlab and the LIGO E2E group's simulation environment `modeler/Alfi`. The Matlab utilities are somewhat more general purpose and allow scalars, vectors and matrices to be written, and even provide limited support for symbolic results. The E2E routines are specifically for state-space matrices.

10 Appendix

10.1 Download locations

Previously, the toolkit and selected models of interest were published at <http://www.ligo.caltech.edu/~e2e/SUSmodels/>. This was time-consuming to maintain and is somewhat out of date. The explanatory content has been moved to the Advanced LIGO wiki at

<https://awiki.ligo-wa.caltech.edu/aLIGO/Suspensions/MathematicaModels>,

and the Mathematica code has been moved to the Advanced LIGO SUS SVN repository. The repository as a whole can be viewed via the web interface at

<http://redoubt.ligo-wa.caltech.edu/websvn/listing.php?reparent=sus&>

or accessed via an svn client at

<https://redoubt.ligo-wa.caltech.edu/svn/sus> .

LIGO.ORG credentials are needed for viewing both the wiki and the SVN, and write privileges granted by David Barker are needed for committing changes to the SVN.

Relative to the repository root, denoted by `^`, the toolkit and models are in subdirectories of

```
^/trunk/Common/MathematicaModels
```

10.2 Directory structure

As noted above, the Mathematica code has been divided up logically and by directory structure into four classes: (i) the toolkit, (ii) models, (iii) cases, (iv) calculations.

This directory structure has been revised and elaborated since the last version of this manual, and the “calculations” class is new, having been separated out from the idea of a case. To manage the extra complexity, the code to work with it has been encapsulated in two simple functions, which should in fact greatly reduce the burden on the user.

The older structure was rather unsatisfactory because paths to two key directories needed to be hard-coded in many different places, but it seemed to be the least bad system achievable under Mathematica v4, for which the toolkit was first developed.

Unfortunately since the hard-coded paths normally need to be different for different users, this made collaboration difficult - one couldn't send another user a case of a model and have it be immediately usable. In aLIGO, the SUS SVN repository became the preferred place to store stuff like models, and the old structure would have been intolerable, because different users would be continually overwriting files for non-substantive changes. Fortunately new features introduced in Mathematica v5 made it possible to do better.

10.2.1 Toolkit Directory

The toolkit comprises the most generic code and is structured as a number of Mathematica “packages” in a toolkit directory, which needs to be on the Mathematica search path (`$Path`). As is standard for packages, each comes as a source notebook (with extension `.nb`) and an automatically generated executable file (with the same filename but extension `.m`). Key packages used by most models include

- `PendUtil`` (the main pendulum specific utilities)
- `RotationsXYZ`` (custom package for rotation matrices in terms of yaw, pitch and roll)
- `MyShapes`` (custom package similar to `Graphics`Shapes`` but with rotation bug fixed and a few extra shapes)
- `StatusWindow`` (custom package to display progress messages in a status window)
- `IFOModel`` (parameters from Bench)
- `MatlabExport`` (export of values, vectors and matrices to Matlab)
- `E2EExport`` (export of values, vectors and matrices to E2E)

Formerly the location of the toolkit could be anywhere, but the path to it had to be hard-coded into each case. We now insist that the toolkit files are stored in a subdirectory of one of the Mathematica `Autoload` directories, either the global `$BaseDirectory/Autoload/` or the user-specific `$UserBaseDirectory/Autoload/`. (These path specifiers are pseudo-code - a hybrid of Mathematica (`$BaseDirectory`) and Unix (`../Autoload/`) syntaxes. To find the actual location, examine the value of the `$BaseDirectory` or `$UserBaseDirectory` symbol in Mathematica, and then complete the path in the appropriate syntax for your OS.)

The significance of these locations is that when the Mathematica kernel is launched, it searches all subdirectories of `Autoload` for files called `init.m` and runs them. The toolkit includes an `init.m` file which adds its enclosing directory to the path, making the rest of the toolkit files available.

The toolkit can be installed by going to the `Autoload` directory and doing a checkout of the SVN directory

```
^/trunk/Common/MathematicaModels/PendulumToolkit
```

By default this will create a directory `../Autoload/PendulumToolkit` but the exact name `PendulumToolkit` is not significant to Mathematica. `PendulumToolkit` can also be a Unix/Mac/Windows symlink to a directory in a more convenient location (but not a Mac OS X alias or a Windows shortcut).

For installations of Mathematica prior to v8, a more complicated setup is required - see the wiki for details.

10.2.2 Model/Case/Calculation Directories

A model is defined by a model definition notebook, which resides in a model directory. All the cases and calculations for that model reside in subdirectories, so that a model and all its addons can be single checkout from the SVN.

A case is defined by a case definition notebook, which resides in a case directory. Case directories can reside in the model directory, or can be grouped into “folders” and “subfolders” below that. (Intermediate directory levels used to group related cases are called folders.) Since a model and all its cases will be one big subtree of the SVN, each user is encouraged to have their own user folder, named with their ligo.org name (`albert.einstein` or the like) to keep cases of personal interest separate from reference cases representing production suspensions.

A calculation is defined by a calculation notebook, which must reside in the top level of the case directory.

Execution starts in the calculation notebook, and a search upwards is made to find first the case definition and then the model definition. The author of a new case is responsible for creating a calculation directory called `stdcalc`. Users are welcome to have their own calculation directory named with their ligo.org name (`albert.einstein` or the like). Such calculation directories are also called user directories (to be distinguished from user folders two or more levels above.)

10.2.3 Detail of directory structure for typical model

The directory structure for the quad model (which is representative) is outlined below:

QuadLite2Lateral/ (The quad model directory, checked out from the SVN.)
 ASUS4L2LateralModelDefn.nb (Model definition notebook for the quad model.)
 ASUS4L2LateralModelDefn.m (Autosave version with executable code from above.)
 default/ (The "case directory" for the default case of the quad model)
 ASUS4XLLateralCaseDefn.nb (The case definition notebook for the default case.)
 ASUS4XLLateralCaseDefn.m (Autosave version with executable code from above.)
 precomputed/ (Precomputed results for the default case.)
 stdcalc/ (Pseudo-user directory with prototype version of calculation.)
 ASUS4XLLateralModelCalcPlots.nb (User calculation notebook for pseudo-user stdcalc with representative collection of plots.)
 ASUS4XLLateralModelCalcExport.nb (User calculation notebook for pseudo-user stdcalc that generates standard set of exported result files.)
 quickcalc/ (Pseudo-user directory with prototype version of quick calculation.)
 ASUS4XLLateralModelCalcPlots.nb (User calculation notebook for pseudo-user quickcalc with representative collection of plots.)
 ASUS4XLLateralModelCalcExport.nb (User calculation notebook for pseudo-user quickcalc that generates standard set of exported result files.)
 albert.einstein (User directory for A. Einstein.)
 ASUS4XLLateralModelCalc.nb (Custom calculation for A. Einstein.)
 anothercase/ (Another case, with different numerical parameters.)
 ASUS4XLLateralCaseDefn.nb (The case definition for anothercase.)
 precomputed/
 stdcalc/
 quickcalc/
 SuchAndSuchCases/ (A folder for a particular type of case.)
 asuchandsuchcase/ (A case in a folder.)
 SoAndSoCases/ (A subfolder for a particular subtype of case.)
 asoandsocase/ (A case in a subfolder.)
 ... (And so on to arbitrary depth.)
 ... (Other cases at the top level. Note however that default is the only top-level case allowed in the SVN.)
 ... (Other models.)

10.2.4 Directory structure differences from older versions

The need to make the toolkit and models SVN-friendly was the impetus to revise the old scheme. The key changes are as follows:

- The `init.m` in `PendulumToolkit` does most of the path set up. It adds `PendulumToolkit` to the Mathematica path (`$Path`) (eliminating the need for the `modelsupportdirectory` variable), and defines functions to make it easy to define `modeldirectory` relative to the calculation notebook.
- In Mathematica v7 and earlier, there is no function by which the `init.m` file in `PendulumToolkit` can tell the name of the directory it is in, which it needs to set the path to include itself. Thus we use a supplementary `init.m` file in `$BaseDirectory/Kernel/` or `$UserBaseDirectory/Kernel/` to provide that information via one hard-coded assignment. Templates for creating one of these files are provided, but only need be installed by users with older versions.
- The `old/current` level of the hierarchy is gone. "The" model directory (the one pointed to by `modeldirectory`) can now have a name reflecting the model. It's easiest to have it be the same as on the SVN (e.g., `QuadLite2Lateral`) but you can check it out under another name if you prefer. Models can be checked out in arbitrary places - different models don't have to be siblings.
- Case directories no longer have to live in the top level of the model directory, but can now be organized into a tree structure of arbitrary depth. (Directories used for the purpose of grouping cases are called *folders* in this and other documentation.) The formal name of a case has been generalized from a single string to a Mathematica list of strings giving the relative path from the model directory to the case directory, e.g., `{"myfolder", "mysubfolder", "mycase"}`.
- A new "user" level has been added at the bottom of the hierarchy. Each person using a case of a model from the SVN is strongly encouraged to have their own calculation notebook in their own user subdirectory, with their preferred set of plots etc. There are two standard pseudo-users, `stdcalc` and `quickcalc`, with template calculations that can be customized. `stdcalc` is the full calculation as of old. `quickcalc` is currently only available for quad and triple model and uses the package `QuickQuadLateral` or `QuickTriple` to do a slightly approximate but quicker version of the calculation.
- The case calculation notebook has been split into a case definition notebook that contains only definitional stuff and lives on the case level, and a user calculation notebook that lives on the user level.
- The case definition notebook now has all cells set to be initialization cells and has an associated autosave `.m` file like the model definition notebook.
- The user calculation notebook is responsible for loading the case definition, which in turn is responsible for loading the model definition. Functions are provided to make this easy.

10.2.5 Flow of control between code in toolkit models, cases and calculations

The following sequence illustrates how everything gets loaded for a case {"myfolder", "mysubfolder", "mycase"} of the quad model when the toolkit has been installed in `$UserBaseDirectory/Autoload/PendulumToolkit`.

When the Mathematica kernel launches, `$UserBaseDirectory/Kernel/init.m` file is discovered and run (assuming it has been created as recommended for Mathematica v7 or earlier). If it is actually Mathematica v7 or earlier that is running, it adds `$UserBaseDirectory/Autoload/PendulumToolkit` to the Mathematica `$Path`.

The `$UserBaseDirectory/Autoload/PendulumToolkit/init.m` file is discovered and run, and for Mathematica v8 or later adds `$UserBaseDirectory/Autoload/PendulumToolkit` to `$Path`.

Regardless of version, `$UserBaseDirectory/Autoload/PendulumToolkit/init.m` also defines a number of useful functions for managing the hierarchy, all based on `notebookdirectory[]` which returns the directory of the notebook from which it is executed (typically a user calculation notebook). The key ones are `loadcasefromuser[]` which attempts to load a case definition, and `loadmodelfromuser[]` which attempts to load a model definition.

The user calculation notebook, `QuadLite2Lateral/myfolder/mysubfolder/mycase/stdcalc/ASUS4XLLateralModelCalc.nb` is responsible for calling `loadcasefromuser["ASUS4XLLateralCaseDefn.m"]` to run the .m version of the case definition notebook in the directory one level above (`QuadLite2Lateral/myfolder/mysubfolder/mycase/ASUS4XLLateralCaseDefn.m`).

The case definition notebook is responsible for setting `modelcase = {"myfolder", "mysubfolder", "mycase"}` and calling `loadmodelfromuser["ASUS4L2LateralModelDefn.m"]` to load the model. `loadmodelfromuser[]` searches upward from the case directory (`QuadLite2Lateral/myfolder/mysubfolder/mycase`) to the model directory (`QuadLite2Lateral`).

Notes

The name of the function `loadmodelfromuser[]` is potentially confusing because it reflects the anticipated situation at runtime rather than anything about the source code. The call to the function will normally appear in the case definition notebook, and will be copied to the associated .m file by the autosave process. The .m file will be run by a call to `loadcasefromuser[]` in the user calculation notebook, so the user calculation notebook will be the current notebook for the execution of both calls. If you evaluate `loadmodelfromuser[]` directly in the case definition notebook with Shift-Return or the like, it will fail. Normally this should not be a problem because there's so little non-trivial code in most case definition files that it can be debugged simply by saving changes and evaluating the `loadcasefromuser[]` line in the user calculation notebook.

If you do ever have a particularly complicated case definition notebook that you would like to debug a cell at a time you can change `loadmodelfromuser[]` to `loadmodelfromcase[]` temporarily. In Mathematica v8 there's a variable `$InputFileName` by which the .m version of the case definition notebook could know where it is, but this has been avoided for backward compatibility.

In older versions of the toolkit, some functions like `saveprecomputed[]` assumed that the Mathematica working directory was set to the case directory, and the case notebook was responsible for setting it. That requirement has been eliminated from the toolkit and the setup for it has been eliminated from the default template cases, but if it matters to your own code you can always reinstate it with `SetDirectory[casedirectory[]]` after the model has been loaded.

The search behaviour of `loadmodelfromuser[]` is a natural generalization of the old system. Formerly the standard code for loading a model would search the case directory and the model directory in that order. That way, a customised model (say with added asymmetry) in the case directory would be given precedence. Now the search goes from the case directory via all intermediate levels to the model directory. If you have a group of cases with the same type of asymmetry, consider taking advantage of this by putting them in a subfolder with a shared custom model definition.