LASER INTERFEROMETER GRAVITATIONAL WAVE OBSERVATORY

LIGO Laboratory / LIGO Scientific Collaboration

LIGO- T020205-00-D | *ADVANCED LIGO* | 9 Feb 2003

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 Introduction ... 4

1.1 Purpose and Scope ... 4

1.2 References ... 4

2 Overview ... 4

2.1 Toolkit ... 4

2.2 Physical systems modelled ... 4

2.2.1 Triple Model ... 5

2.2.2 Quad Model ... 5

2.3 Method of calculation ... 8

2.4 Utility functions ... 9

2.5 Model specific utilities ... 9

3 Installation ... 9

3.1.1 Directory structure overview ... 9

3.1.2 Registering paths in the code ... 10

4 Exploring the supplied cases ... 10

4.1 Computing cases from scratch versus using precomputed results ... 11

4.2 Stages of the calculation ... 11

4.3 Available listing and plotting commands ... 13

4.3.1 Frequencies ... 13

4.3.2 Mode shape tables ... 13

4.3.3 Mode shape plots ... 13

4.3.4 Force transfer function plots ... 14

4.3.5 Displacement transfer function plots ... 14

4.3.6 Thermal noise plots ... 15

4.4 Default numerical values ... 15

4.5 Creating new cases of the existing models ... 16

4.6 Using overrides ... 16

5 The definition of the model ... 17

5.1 Version History ... 17

5.2 Dependencies ... 17

5.3 The model definition proper ... 18

5.3.1 The master variable list: `allvars` ... 18

5.3.2 The parameter list: `allparams` ... 18

5.3.3 Coordinate lists for rigid bodies and points on them ... 19

5.3.4 Items with gravitational potential energy: `gravlist` ... 19

5.3.5 Wires: `wirelist` ... 19

5.3.6 Springs: springlist 21
5.3.7 The kinetic energy: kinetic 22
5.3.8 Values of constants: defaultvalues 22
5.3.9 Approximation to equilibrium position: startpos 24
5.4 Model dependent diagnostic utilities 24
5.4.1 Angular velocity transformation matrices: e2s, e2b and e2ni 24
5.4.2 Plotting routines: eigenplot[] 25
5.4.3 Listing routines: pretty[] 25
5.4.4 Input and output vectors 25
6 The calculation of the model 25
6.1 The Calculate[] function 26
6.2 Stage 0A – normal modes without wire bending elasticity 26
6.2.1 Numerical Substitutions: constval 26
6.2.2 Numerical start position for minimization of the potential: startval 27
6.2.3 Variables that participate in the minimization: optvars 27
6.2.4 Variables that don't participate in the minimization: nonoptvars and nonoptval 27
6.2.5 The list of potential terms to be used for the minimization: potentialtermlist 27
6.2.6 The full expression for the potential: potential 27
6.2.7 The numeric potential: potentialNN 27
6.2.8 The minimization: potential0 and optval 27
6.2.9 Velocity variables: velocities 27
6.2.10 The kinetic energy matrix: kineticN and kineticmatrix 28
6.2.11 The potential used for the normal mode calculation: potentialtermlist0 28
6.2.12 The Stage 0A damping-specific potential matrices: potentialmatrices0T... 28
6.2.13 The Stage 0 potential matrix: potentialmatrix0 28
6.2.14 The Stage 0 normal modes: eigenvalues0, eigenvectors0 and Hz0 28
6.3 Stage 0B – damping without wire bending elasticity 28
6.3.1 Potential matrices without tension: potentialmatrices0NT 28
6.3.2 Coupling matrices: couplingmatrices0T, couplingmatrices0NT, couplingmatrices0 29
6.3.3 The equations of motion function and the coupling function: eom0 and coupling0 29
6.4 Preparation for Stages 1 and 2 29
6.4.1 Wire angles: relaxval 29
6.4.2 Additional potential term lists: potentialtermlistWB and potentialtermlistWE 29
6.5 Stage 1 – normal modes and damping with wire bending elasticity 30
6.6 Stage 2 – normal modes and damping with additional wire stretch due to bending 30

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

"Suspension Model Comparisons", by Mark Barton (T020011-00). Detailed comparison of the results from the Mathematica models discussed in this document and the equivalent Matlab models of Calum Torrie, Ken Strain et al.

"Development of Suspensions for the GEO600 Gravitational Wave Detector" by Calum Torrie (PhD Thesis, University of Glasgow, 2000). Description of the design of the GEO suspensions, with particular reference to an analytical model of them in Matlab.

"Suspension and Control for Interferometric Gravitational Wave Detectors", Matthew Husman (PhD Thesis, University of Glasgow, 2000). Description of the design and control of the GEO suspensions, with particular reference to an analytic model of them in Maple.

2 Overview

2.1 Toolkit

The triple and quad pendulum models described later in this document are based on a common toolkit of utilities for setting up models of mass-wire-spring systems which 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. 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 and bending 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.

2.2 Physical systems modelled

Throughout this document, a "model" is a specification of particular arrangement of toolkit elements (i.e., masses, wires, and springs) but not the numerical values of the various element properties. A "case" of a model is the model with a particular set of numeric values. Two major models of interest to LIGO have been defined: one for a generic GEO triple suspension and one for a quad suspension. Additional models can be defined fairly easily by someone familiar with the toolkit.

2.2.1 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 (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 "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.

Each blade spring is represented by a small mass element connected by a spring element to the object the blade spring is mounted on. Together with the three main masses, this makes 9 rigid bodies with 54 DOFs, plus 6 spring elements and 10 wire elements. A schematic diagram with the principal geometric and mechanical parameters is given in Figure 1. As far as possible, the names of parameters have been made identical to the corresponding ones in the Matlab model of Torrie et al.

2.2.2 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. A schematic with the main geometrical and mechanical parameters is givien in Figure 2. As far as possible, the names of parameters have been made identical to the corresponding ones in the Matlab model of Torrie et al.

As with the triple, each blade spring is represented by a small mass element connected by a spring element to the object the blade spring is mounted on. Together with the four main masses, this makes 10 rigid bodies with 60 DOFs, plus 6 spring elements and 14 wire elements.

Note a key difference in the quad relative to the triple: the triple has four springs on the upper mass, one for each of the wires leading down to the intermediate mass. The quad has only two springs on each of the "new" and "upper" masses, with two wires attached to the tip of each.

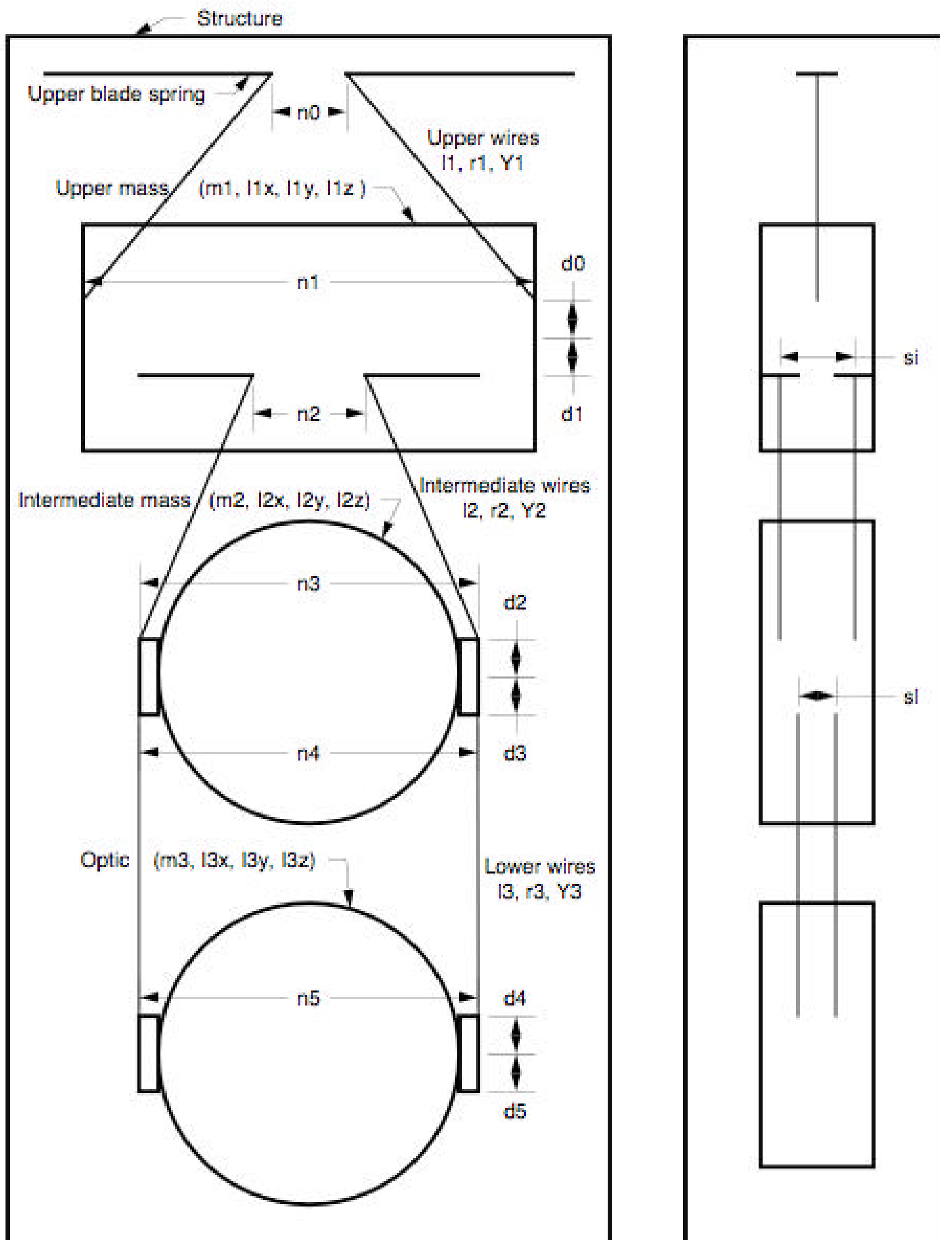


Figure 1: Triple pendulum with principal geometric and mechanical parameters

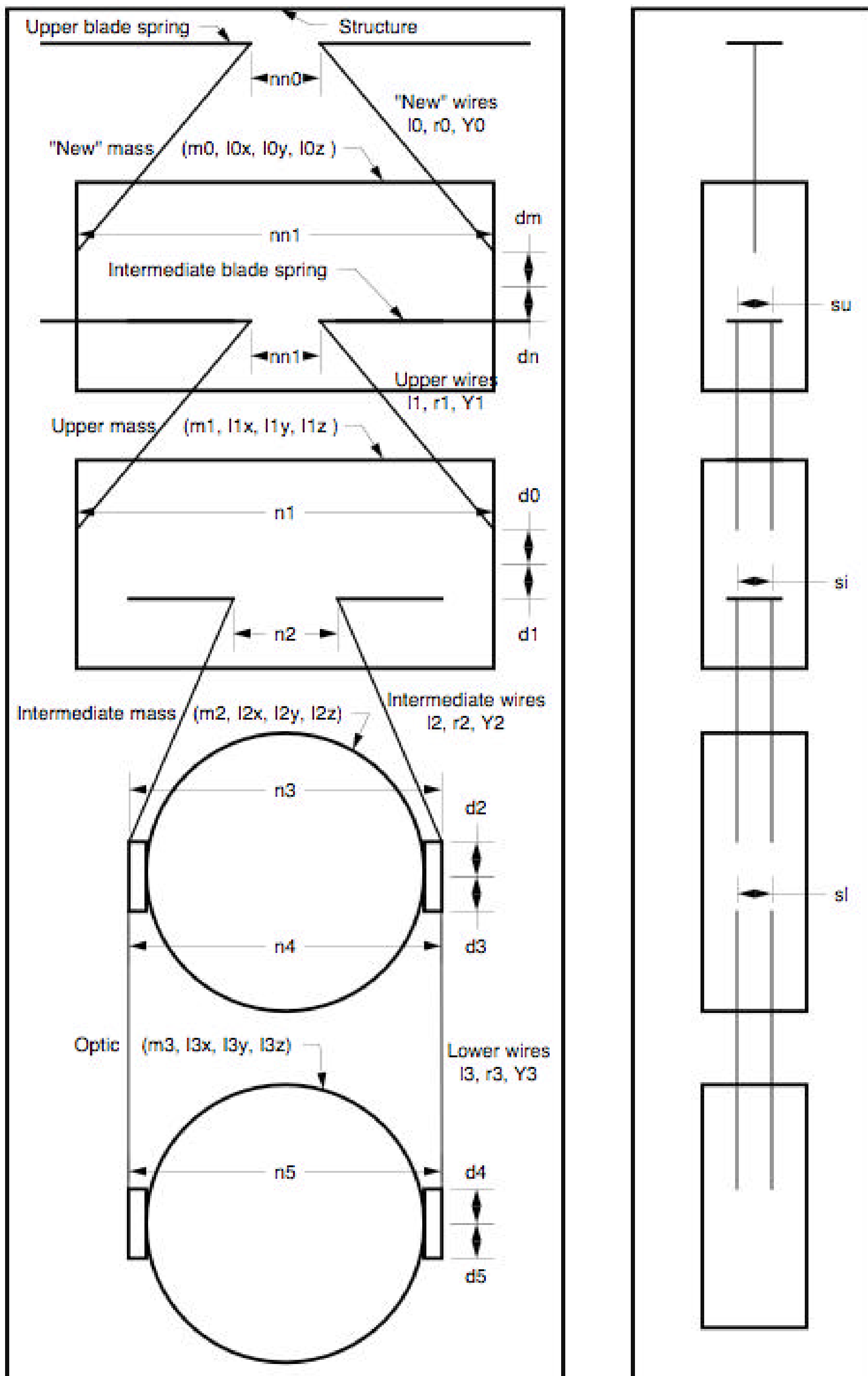


Figure 2: Quad pendulum with principal geometric and mechanical parameters

2.3 Method of calculation

The calculation is based on the method of normal modes as described in say Goldstein, "Classical Mechanics". The simplest version of such a calculation has the following steps:

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

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

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

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

(v) 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.

(vi) Do a simultaneous diagonalization of the stiffness and mass matrices to obtain the eigenfrequencies and eigenmode shapes.

The straightforward way of doing step (v) above would be to create a single expression for the potential, with symbols for the coordinates and numerical values for all constants, and simply take the partial derivatives w.r.t. all coordinates. However this turns out to be poor for three reasons:

(i) Most terms in the potential are independent of most coordinates, and so the potential matrix is sparse. It is very inefficient to let Mathematica discover this for itself in the course of doing the partial derivatives. It is much quicker to take the potential a term at a time, make a list of the coordinates that occur in each term using a special purpose function, and iterate over only those coordinates.

(ii) Different terms in the potential typically have different damping associated with them and merging them makes it impossible to keep track of this.

(iii) The potential terms for wire bending are small in magnitude but very complex and slow to process. Since they are typically only significant for the calculation of thermal noise it is desirable to make them optional.

To avoid these problems, the potential is processed term by term to produce a list of stiffness matrices, which are then combined in different ways according to the application. The theory of normal modes neglects damping, so for calculating the modes, the individual stiffness matrices are simply added. However for transfer functions and thermal noise, the matrices are first multiplied by a user-specified complex damping function for the corresponding damping type.

Because of the calculation for the wire bending potential terms referred to in (iii) above, the calculation is divided into three stages, 0, 1, and 2. Stage 0 calculates all the potential terms for gravity, springs, and the stretching of wires. Stage 1 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. The wire bending potential terms are calculated using the approach described in Matthew Husman's PhD thesis but reimplemented from scratch in Mathematica. The wire is treated as a beam under tension, and the contributions to the potential energy in the static equilibrium state are used (thus violin modes are excluded).

2.4 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

2.5 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”)

3 Installation

3.1.1 Directory structure overview

The most up-to-date version of the triple and quad models and the required support files can be downloaded from http://www.ligo.caltech.edu/~mbarton/SUSmodels/indexMB.html. This document describes v3.0 of each model. Version history information can be found at the above website and in the code.

The support files are three custom Mathematica packages: RotationsXYZ.*, MyShapes.* and StatusWindow.*. Each package comes in .nb and .m versions. The .nb file is the source and

contains the comments. The .m file is the version actually loaded. It is automatically regenerated whenever the corresponding .nb file is edited. Download both versions to a convenient location that can be reserved for Mathematica packages e.g., ~/MathLib on a Unix system. The package directory needs to be registered in the model files as explained below.

The model proper is distributed as an archive in .tar, .zip, or .sit format. This expands to a directory, typically called `current`, containing the model definition file, typically called `ASUS3ModelDefn.nb` or the like. This directory is called the model directory, and can be renamed and/or placed anywhere convenient provided the path to it is registered in the model files as explained below. The model directory contains subdirectories containing the results for different cases. The case subdirectories need to remain in the model directory. They can be renamed or duplicated to create new cases, provided the subdirectory name is registered in the model files as explained below.

Each case subdirectory (e.g., `default`) contains a calculation notebook with a name like `ASUS3ModelCalcDefault.nb`, and a subdirectory called `precomputed` with archived intermediate results for that case. The subdirectory name needs to be registered in the calculation notebook as explained in the next section.

3.1.2 Registering paths in the code

As explained above, there are three directories of interest that need to be registered: the custom package directory, the model directory, and the case subdirectory for the current case. These have to be set in each case calculation file. Therefore, upon downloading and unpacking the model files, repeat the following for each case:

1. Go into the case subdirectory and look for a notebook with a name like `ASUS3ModelCalc`*`XXX`*`.nb`. (The *XXX* part will normally reflect the case subdirectory name, e.g., `default -> ASUS3ModelCalcDefault.nb`). Open the notebook in Mathematica.
2. Check the assignment to `modelcase` in the Switches section near the beginning. It should correspond exactly to the name of the case subdirectory (e.g., "`default`").
3. Check the assignment to `modeldirectory` in the Switches section. Check that there is a case in the `Switch[]` statement corresponding to the value of `$System` for your version of Mathematica, and if not, create one. The value returned for that case should be a string giving an absolute path to the model directory. The path string should be in the particular standard syntax for your OS (e.g., slash-delimited for Unix, colon-delimited for Mac OS 9, etc).
4. Check the assignment to `modelsupportdirectory` in the Switches section. Check that there is a case in the `Switch[]` statement corresponding to the value of `$System` for your version of Mathematica, and if not, create one. The value returned for that case should be a string giving an absolute path to the directory where you have placed the custom packages `RotationsXYZ.*`, `MyShapes.*` and `StatusWindow.*`. The path string should be in the particular standard syntax for your OS.

4 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` or `ASUS4ModelCalcDefault.nb`.

4.1 Computing cases from scratch versus using precomputed results

Because the calculation time for the full model is 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.

4.2 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 equilibrium position is found, the kinetic energy is processed to produce the mass matrix, the potential terms for gravity, spring elements, and longitudinal wire-stretching 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 of one of the damping functions has a real part that is not 1 at zero frequncy.
- 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 advantage[1]. It is added to the gravity stiffness matrix (which then acquires

---

[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

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

- Stage 1: 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.

As of version 3.0 of each model the calculation is done using the `Calculate[]` function. The function call `Calculate[`*`stage`*`]` where *`stage`* is one of `Stage0A`, `Stage0B`, `Stage1` or `Stage2` will compute (or recall from `precomputed`) 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.

---

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.

4.3 Available listing and plotting commands

4.3.1 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.

4.3.2 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

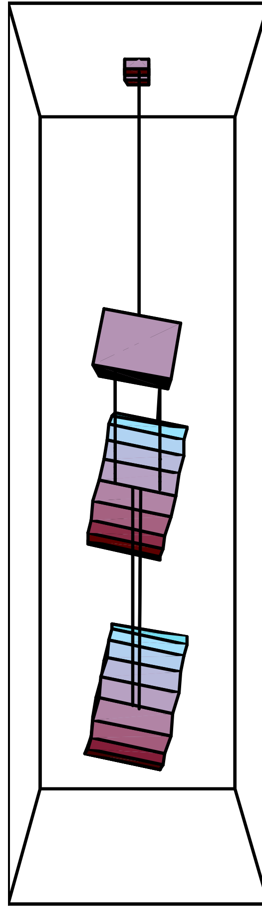
	x	y	z	yaw	pitch	roll
UL blade	0	0	0	0	0	0
UR blade	0	0	0	0	0	0
Mass U	-0.00134996	0	0	0	0.367617	0
LLF blade	-0.00150905	0	0.0118717	0	0.367617	0
LLB blade	-0.00150905	0	-0.0118717	0	0.367617	0
LRF blade	-0.00150905	0	0.0118717	0	0.367617	0
LRB blade	-0.00150905	0	-0.0118717	0	0.367617	0
Mass I	-0.00248634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

4.3.3 Mode shape plots

You can get a 3D rendering of a normal mode with the `eigenplot[`*`eigenvector`*`,` *`amplitude`*`, {`*`viewpoint`*`}]` function. 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



4.3.4 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 `makefinputvector[`*symbol*`]`, which returns a vector representing unit force or torque applied to the pendulum coordinate specified by *symbol*. (For example, `makefinputvector[yaw3]` is unit torque in yaw on the optic.) Similarly output basis vectors can be constructed using `makeoutputvector[`*symbol*`]`.

4.3.5 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.)

4.3.6 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 makefinputvector[] 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.

4.4 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 notebooks are set up to encourage leaving the model definition notebook alone and specifying 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.

4.5 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.

4.6 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 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.

5 The definition of the model

To make more ambitious changes that can't be implemented with overrides (such 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.

5.1 Version History

The type and version of the model should be specified by assigning a text description of the model to `modelcomment` and a numeric version number to `modelversion` in the Version history section at the beginning of the file. Code in the calculation notebook 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.

5.2 Dependencies

The Dependencies subsection of the Preliminaries section loads various packages and support files. These include:

- `` Graphics`Graphics` `` (standard package for general graphics)
- `` Graphics`Polyhedra` `` (standard package for drawing polyhedra)

- `LinearAlgebra`MatrixManipulation`` (standard package for matrix manipulation)
- `RotationsXYZ`` (custom package for rotations in yaw, pitch and roll)
- `MyShapes`` (custom package similar to Graphics`Shapes` but with rotation bug fixed)
- `StatusWindow`` (custom package to display text messages in a status window)
- `IFOModel.m` (parameters from Bench)
- `wirefunctions.m` (functions for wire bending)
- PendUtil.m (the pendulum toolkit utilities)

5.3 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.

5.3.1 The master variable list: `allvars`

The symbol `allvars` 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. At least for the current models, the structure is a passive anchor point as far as normal mode motion is concerned, so its variables are not included. 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,
      xllf,yllf,zllf,yawllf,pitchllf,rolllf
      xllb,yllb,zllb,yawllb,pitchllb,rolllb,
      xlrf,ylrf,zlrf,yawlrf,pitchlrf,rolllrf,
      xlrb,ylrb,zlrb,yawlrb,pitchlrb,rolllrb,
      x2,y2,z2,yaw2,pitch2,roll2,
      x3,y3,z3,yaw3,pitch3,roll3
};
```

5.3.2 The parameter list: `allparams`

The symbol `allparams` should be 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
```

```
};
```

5.3.3 Coordinate lists for rigid bodies and points on them

To make things more readable later on, it is highly desirable 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 = {l Sin[theta], l Cos[theta], 0, 0, 0, 0};
```

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

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

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

As well as the movable objects, you should also define coordinate lists for static objects that have wires or springs attached, such as the structure. In the triple model this is

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

5.3.4 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.)

5.3.5 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:

```
{
    coordinate list defining first mass,
    attachment point for first mass (local coordinates),
    attachment vector for first mass,
    coordinate list defining second mass,
    attachment point for second mass (local coordinates),
```

```
        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
}
```

Each wire is presumed to be strung between two rigid bodies, and items 1 and 4 are the coordinate lists specifying the bodies. Items 2 and 5 are the attachment point 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 obvious: 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.

Finally, Items 13 and 14 are identifying symbols that specify which damping functions are to be attributed to the longitudinal and bending elasticities respectively. (The damping function for each tag is specified as a substitution in `defaultvalues` – see below.)

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

```
wirelist = {};
AppendTo[wirelist,{
        bladeUL,
        bladeULa,
        bladeULavec,
        massU,
        massUl,
        massUlvec,
        Y1,
```

```
        ul1,
        kw1,
        {1,0,0},
        M11,
        M12,
        wireUtype,
        wireUatype
}];
```

5.3.6 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,krollu}],
        {0,0,bdu*kbuz,0,0,0}
}];
```

5.3.7 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 coordinates 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`*`]` is the coordinates of a local point on an object, `COM` (equal to `{0,0,0}`) is the centre of mass of any body in local coordinates for that body, and `omegaB[]` is the angular velocity vector.

5.3.8 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 may contain the function call `constraintsubstitutions[]`, which expands to a list of substitutions of the form `kcon`*`variablename`* `-> 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 substutions 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 in this application because `&` has a lower precedence than `->`. The functions you supply should accept a value of frequency in Hz 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 a substitution for `damping[real,`*`dampingtype`*`]` or `damping[imag,`*`dampingtype`*`]` in `defaultvalues`, they fall back to an even more fundamental default of

```
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 velocity-proportional damping would be

```
damping[imag,dampingtype] -> (2*Pi*b*#/k&)
```

for damping of `b` N/(m/s) relative to elasticity of `k` N/m

5.3.9 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`.

5.4 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.

5.4.1 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, while the three individual rotations implied in (yaw+dyaw, pitch+dpitch, roll+droll) aren't about mutally 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 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[yaw1,pitch1,roll1],
            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[yawlrb,pitchlrb,rolllrb],
            IdentityMatrix[3],
            omegaSNI.omegaIS[yaw2,pitch2,roll2],
            IdentityMatrix[3],
            omegaSNI.omegaIS[yaw3,pitch3,roll3]
      }]/.optval;
```

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

5.4.3 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[]`.

5.4.4 Input and output vectors

To enable basic use of the variour 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.

6 The calculation of the model

6.1 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).

6.2 Stage 0A – normal modes without wire bending elasticity

6.2.1 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`.

6.2.2 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`.

6.2.3 Variables that participate in the minimization: `optvars`

The minimization of the potential should normally be done with respect to all the variables in allvars. However as a aid to debugging, only those in the list `optvars` are used. This is useful in isolating problems in the specification of the wires and springs. For the full calculation `optvars` should be initialized to `allvars`.

6.2.4 Variables that don't participate in the minimization: `nonoptvars` and `nonoptval`

The list `nonoptvars` is set to the complement of optvars. 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 `nonoptvar`.

6.2.5 The list of potential terms to be used for the minimization: `potentialtermlist`

The 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},…}
```

6.2.6 The full expression for the potential: `potential`

The terms in potentialtermlist are 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.

6.2.7 The numeric potential: potentialNN

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

6.2.8 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 `optvars`.

6.2.9 Velocity variables: `velocities`

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

6.2.10 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`.

6.2.11 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.

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

The potential term parts of all the items in `potentialtermlist0` are differentiated to form individual damping-specific potential matrices. Matrices with the same damping tags are combined to form a list `potentialmatrices0T` of the form {{*dampingtype*, *matrix*}, {*dampingtype*, *matrix*},…}. (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.

6.2.13 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.

6.2.14 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`.

6.3 Stage 0B – damping without wire bending elasticity

6.3.1 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`.

6.3.2 Coupling matrices: couplingmatrices0T, couplingmatrices0NT, couplingmatrices0

The coupling matrices are analogous to the potential matrices except that they describe the elasticity from the structure to the pendulum instead of the pendulum internal elasticity. Thus they are calculated in a completely analogous way.

6.3.3 The equations of motion function and the coupling function: eom0 and coupling0

This are calculated from kineticmatrix, potentialmatrices0 and couplingmatrices0. Thus they neglect wire bending elasticity but correctly incorporate dissipation dilution on the remaining sources of damping.

6.4 Preparation for Stages 1 and 2

6.4.1 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.

6.4.2 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 is derived in the notebook WireAnalysis.nb, based on the discussion in Matt Husman's thesis. The two key results of the analysis are exported to the notebook wirefunctions.m which is read in by the model definition file when it is loaded. 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 functions wirepotential1[] and wirepotential2[] defined in the Utilities section of the model notebook 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.

6.5 Stage 1 – normal modes and damping with wire bending elasticity

The Stage 1 calculation processes the entries in `potentialtermlistWB` to produce `potentialmatricesWB` and `couplingmatricesWB`. These are then merged with `potentialmatrices0` and `couplingmatrices0` respectively to produce `potentialmatrices1` and `couplingmatrices1`. In turn, these are processed in the same way as for Stage 0 to produce `potentialmatrix1`, `couplingmatrix1`, `eom1`, `coupling1`, `eigenvalues1`, `eigenvectors1` and `Hz1`.

6.6 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` and `couplingmatricesWE`. These are then merged with `potentialmatrices1` and `couplingmatrices1` respectively to produce `potentialmatrices2` and `couplingmatrices2`. In turn, these are processed in the same way as for Stages 0 and 1 to produce `potentialmatrix2`, `couplingmatrix2`, `eom2`, `coupling2`, `eigenvalues2`, `eigenvectors2` and `Hz2`.