resonance Documentation

Release 0.10.1

Jason K. Moore, Kenneth Lyons

Dec 10, 2017

Contents:

Topical Outline

The course is taught over 20 two hour class periods during a quarter system of 10 weeks of instructions and 1 week for examinations. One of the 20 class periods is reserved for a midterm examination, 2 hours are reserved for exam reviews leaving 36 hours of in class time. The following lists the topics for each of the class periods which correspond to the detailed headers below:

L#	Date	Notebook #
01	W Sep 27	1, 2
02	M Oct 02	3
03	W Oct 04	4
04	M Oct 09	5
05	W Oct 11	6
06	M Oct 16	7
07	W Oct 18	8
08	M Oct 23	9
NA	T Oct 24	Drop Date
09	W Oct 25	10
10	M Oct 30	11
11	W Nov 01	12
12	M Nov 06	Exam
13	W Nov 08	13
NA	F Nov 10	Veterans Day Holiday
14	M Nov 13	14
15	W Nov 15	15
16	M Nov 20	16
17	W Nov 22	17
NA	R Nov 23	Thanksgiving Holiday
NA	F Nov 24	Thanksgiving Holiday
18	M Nov 27	18
19	W Nov 29	19
20	M Dec 04	20
21	W Dec 06	21
NA	T Dec 12	Final Exam @ 6:00 PM

1.1 Analyzing Vibrating Systems

1.1.1 1. Introduction to Jupyter

This notebook introduces students to the Jupyter notebook environment and establishes good practices for creating computational notebooks and scientific python programming.

After the completion of this assignment students will be able to:

- open Jupyter notebooks and operate basic functionality
- fetch assignments, complete exercises, submit work and view the graded work
- solve basic scientific python problems
- create a well formatted and fully executing notebook

1.1.2 2. Introduction to vibrations: Book Balancing on a Cup

This notebook introduces a single degree of freedom vibratory system in which a textbook balances on a cylindrical cup. The system is implemented as a model that students can interact with in order to visualize its free response and compare to the demonstration in the classroom.

After the completion of this assignment students will be able to:

- visualize a system's free response
- estimate the period of a sinusoidal vibration from a time series
- compare a computer simulation result to experimental result
- interactively adjust the book inertia to see the affect on system response
- understand the concept of natural frequency nd its relationship to mass/inertia

1.1.3 3. Measuring a Bicycle Wheel's Inertia

This notebook introduces the concept of using vibratory characteristics to estimate parameters of an existing system. It discusses how vibrations can be measured and how these measurements might relate to parameters of interest, such as the inertia of a bicycle wheel.

After the completion of this assignment students will be able to:

- describe different methods of measuring vibrations
- choose appropriate sensors and sensor placement
- visualize the vibrational measurements
- use curve fitting to estimate the period of oscillation
- understand the concept of natural frequency and its relationship to mass/inertia and stiffness
- state two of the three fundamental characteristics that govern vibration (mass/inertia and stiffness)
- use frequency domain techniques to characterize a system's behavior

1.1.4 4. Clock Pendulum with Air Drag Damping

This notebook introduces the third fundamental characteristic of vibration: energy dissipation through damping. A simple pendulum model is implemented that allows students to vary the damping parameters and visualize the three regimes of linear damping.

After the completion of this assignment students will be able to:

- understand the concept of damped natural frequency and its relationship to mass/inertia, stiffness, and damping
- state the three fundamental characteristics that make a system vibrate
- compute the free response of a linear system with viscous-damping in all three damping regimes
- identify critically damped, underdamped, and overdamped behavior
- determine whether a linear system is over/under/critically damped given its dynamic properties
- understand the difference between underdamping, overdamping, and crticial damping

1.1.5 5. Clock Pendulum with Air Drag and Joint Friction

This notebook builds on the previous one by introducing nonlinear damping through Coulomb friction. Students will be able to work with both a linear and nonlinear version of the same system (pendulum) in order to compare the free response in both cases.

After the completion of this assignment students will be able to:

- identify the function that governs the decay envelope
- compare this non-linear behavior to the linear behavior
- estimate the period of oscillation
- compute the free response of a non-linear system with viscous and coulomb damping

1.1.6 6. Vertical Vibration of a Bus Driver's Seat

This notebook introduces external forcing of a vibratory system, where the external force is modeled as a sinusoidal input to the bottom of a bus driver's seat.

After the completion of this assignment students will be able to:

- excite a system with a sinusoidal input
- understand the difference in transient and steady state solutions
- use autocorrelation to determine period
- relate the frequency response to the time series
- create a frequency response plot
- define resonance and determine the parameters that cause resonance

1.1.7 7. Vertical vibration of a Bus Driver's Seat with a Leaf Spring

This notebook builds on the previous one by replacing the linear spring with a realistic leaf spring.

After the completion of this assignment students will be able to:

- create a force versus displacement curve for a leaf spring

- describe the time response and frequency response of a non-linear system
- show that sinusoidal fitting does not necessarily describe non-linear vibration

1.1.8 8. Bicycle Lateral Vibration

This notebook introduces a simple lean and steer bicycle model as an example of a system with multiple degrees of freedom. Coupling and modes are discussed from a data analysis perspective.

After the completion of this assignment students will be able to:

- get a sense of the coupling of input to output through frequency response plots
- simulate a 2 DoF vibratory model
- identify a MDoF system and see effects of coupling through time and frequency domain
- determine if a general 2 DoF is stable
- sweep through input frequencies to discover modal frequencies

1.1.9 9. Simulating a building during an earthquake

This notebook uses a lumped parameter multi-story building model as a many-degree-of-freedom system with all oscillatory modes.

After the completion of this assignment students will be able to:

- examine time domain and frequency coupling with MDoF
- sweeping through frequencies to discover modal frequencies
- visualize the system's response at modal frequencies to see mode shapes

1.2 Modeling Vibrating Systems

1.2.1 10. Modeling the Bicycle Wheel Inertia Measurement System

This notebook walks through modeling two different test rigs for determining the vibrational characteristics of a bicycle wheel. After coming up with a simple model the students will use the canonical linear form of the equations of motion to derive various vibrational parameters.

After the completion of this assignment students will be able to:

- derive the equations of motion of a compound pendulum with Lagrange's method
- derive the equations of motion of a torsional pendulum with Lagrange's method
- linearize the compound pendulum equation
- put equations in canonical form
- review solutions to ODEs

1.2.2 11. Modeling a non-linear spring

TODO : Think this out more.

After the completion of this assignment students will be able to:

- will be able to derive the nonlinear euqations of motion of a system with simple kinmeatics with lagrange's method

1.2.3 12. Modeling the car on the bumpy road

Here will will present the base excitation single degree of freedom system and the students will derive the equations of motion. They will then explore the displacement and force transmisiblity frequency response functions.

After the completion of this assignment students will be able to:

- derive the linear equations of motion ofa system with simple kinematics using lagrange's method
- create system object with custom euqations of motion an simulate the system

1.2.4 13. Modeling the book on a cup

The book balancing on the cup will be revisited. The students will derive the equations of motion which require more complex kinematic analysis and explore the analytical equations of motion. The stability thresholds will be determined as well as the period from the linear model.

After the completion of this assignment students will be able to:

- derive the euqations of motion of a system with non-trivial kinematics with lagrange's method
- apply a linearization procedure to non-linear equations of motion
- determine the stability of a linear system analytically and verify through simulation

1.2.5 14. Balancing your car tire at the autoshop

The mass imbalance problem will be presented through the analytical model of an unbalance car tire. The frequency response will be derived and examined.

After the completion of this assignment students will be able to:

- derive the equations of motion fo a mass imbalance system

1.2.6 15. Engine cam non-sinusoidal periodic forcing

Using an engine cam piecewise periodic function the students will learn how a Fourier series can be used to find the solution to the differential equations symbolicaly.

After the completion of this assignment students will be able to:

- generate a Fourier series of a periodic function
- find the analytic solution of the the mass-spring-damper system

1.2.7 16. Modeling a bulding during an earthquake

We will revisit the multi-story building model and derive the equations of motion for the system. The students will use eigenanalysis of the simple system to discover the modes of motion and simulate the behavior.

After the completion of this assignment students will be able to:

- perform modal analysis of the system to determine its modal frequencies and mode shapes
- represent model using a matric equation of motion (canoncial form)
- formulate the equations of motion for a MDoF system
- use eignvalue analyssis to determine the modeshapes of a mDoF system
- plot the motion of a MDoF system (with no damping) using the analytical solution
- form a MDoF model corresponding to a chain of floors in a buliding

1.2.8 17. Bicycle Model

The students will be given the analytical canocial form of the bicycle equations that do not have simple damping. They will have to convert to state space form and do a full eigenanalysis of the general form. The modes will be examined and the nature of the bicycle motion discovered.

After the completion of this assignment students will be able to:

- convert the canonical linear form into state space form
- interpret eigenvalues and eienvectors of a general 2 DoF linear system

1.3 Designing Vibrating Systems

1.3.1 18. Design a Clock that Keeps Time

The students will be presented with a compound pendulum model of a clock's bob that does not keep time well due to friction and air drag. They will be tasked with designing a system that adds in the right amount of additional energy so that the pendulum has the desired constant period.

After the completion of this assignment students will be able to:

- develop an analytic model of a energy injection system
- simulate the motion of clock and determine its time varying period
- choose the energy injection system parameters that will cause the clock to work as intended

1.3.2 19. Isolator Selection

The students will be presented with a model of X and asked to select and/or design a commercially available vibration isolator that ensures the system meets specific vibrational design criteria.

After the completion of this assignment students will be able to:

- discuss and justify trade-offs and design decisions
- model the system with additional damping provided by isolation
- select/design a vibration isolator to meet given vibration specifications

- analyze a system's motion to determine its vibrational characteristics

1.3.3 20. Designing a Tuned Mass Damper to Earthquake Proof a Building

Students will be presented with a single (or multi?) floor building model. They will need to modify the model to includes a laterally actuated mass on the roof. They will be asked to design an actuation scheme that prevents the building from having too large of displacements or resonance while excited by a earthquake-like vibration at its base.

After the completion of this assignment students will be able to:

- add a generic vibration absorber to a building model
- use a building model to simulate the motion of a building without damping
- choose design criteria for the building and justify decisions (with ISO standards)
- design an absorber that meets their design criteria
- use the frequency response function to demonstrate the effect of the vibration absorber

1.3.4 21. Designing a stable bicycle

The students will be presented with a 2 DoF linear model of a bicycle in canonical form with analytical expressions for the M, C, and K matrix entries that are functions of the 25 bicycle parameters. The students will be asked to discover bicycle designs that meet certain criteria through eigenanalysis and simulation.

After the completion of this assignment students will be able to:

- determine parameters which cause the 2 DoF system to be stable/unstable
- simulate and visualize the motion of a bicycle with difference parameters
- determine and describe the influence of the physical parameters, initial conditions, and steering input on the dynamics of the vehicle
- design a bicycle that meets specific design criteria

1.3.5 22. Designing Shock Absorbtion for a Car

The students will be presented with 2D planar data generated from a "ground truth" 3 DoF half car model. Their job will be to design a quarter car model that behaves similarly to the ground truth model. Once they have a working simple model, then they will design an improved shock absorber for the quarter car model using analytic and computational methods. The instructors will then provide the students with the ground truth model, i.e. the "real" car, and the students will need to show that the ride quality is improved and that design criteria is met.

After the completion of this assignment students will be able to:

- develop a simple analytic model that predicts motion provided from planar 2D "experimental" data
- select springs and dampers to meet given design criteria by demonstrating performance with the simple analytic model
- demonstrate that the designed shock absorber works well for the "real" car
- discuss why the design does or does not meet the design criteria
- reflect on their modeling and design decisions after having tested it against the ground truth model

API

2.1 resonance/system.py

class `resonance.system.`**`System`**

> This is the abstract base class for any single or multi degree of freedom system. It can be sub-classed to make a custom system or the necessary methods can be added dynamically.

> **`add_measurement`**(*name*, *func*)
>
>> Creates a new measurement entry in the measurements attribute that uses the provided function to compute the measurement.
>>
>> **Parameters**
>>
>>> - **`name`** (`string`) – This must be a valid Python variable name and it should not clash with any names in the constants or coordinates dictionary.
>>>
>>> - **`func`** (`function`) – This function must only have existing parameter, coordinate, or measurement names in the function signature. These can be a subset of the available choices and any order is permitted. The function must be able to operate on arrays, i.e. use NumPy vectorized functions inside. It should return a single variable, scalar or array, that gives the values of the measurement.
>>
>> **Examples**
>>
>> ```
>> >>> import numpy as np
>> >>> def f(par2, meas4, par1, coord5):
>> ...         return par2 + meas4 + par1 + np.abs(coord5)
>> >>> f(1.0, 2.0, 3.0, -4.0):
>> 10.0
>> >>> f(1.0, 2.0, 3.0, np.array([1.0, 2.0, -4.0]))
>> array([ 7.,   8.,  10.])
>> >>> sys.add_measurement('meas5', f)
>> >>> sys.measurements['meas5']
>> 10.0
>> ```

---

animate_configuration(*fps=30, **kwargs*)

> Returns a matplotlib animation function based on the configuration plot and the configuration plot update function.
>
> > **Parameters**
> >
> > - **fps** (`integer`) – The frames per second that should be displayed in the animation. The latest trajectory will be resampled via linear interpolation to create the correct number of frames. Note that the frame rate will depend on the CPU speed of the computer. You'll likely have to adjust this by trial and error to get something that matches well for your computer if you want the animation to run in real time.
> >
> > - ****kwargs** – Any extra keyword arguments will be passed to `matplotlib.animation.FuncAnimation()`. The `interval` keyword argument will be ignored.

config_plot_func

> The configuration plot function arguments should be any of the system's constants, coordinates, measurements, or 'time'. No other arguments are valid. The function has to return the matplotlib figure as the first item but can be followed by any number of mutable matplotlib objects that you may want to change during an animation. Refer to the matplotlib documentation for tips on creating figures.

Examples

```
>>> sys = SingleDoFLinearSystem()
>>> sys.constants['radius'] = 5.0
>>> sys.constants['center_y'] = 10.0
>>> sys.coordinates['center_x'] = 0.0
>>> def plot(radius, center_x, center_y, time):
...     fig, ax = plt.subplots(1, 1)
...     circle = Circle((center_x, center_y), radius=radius)
...     ax.add_patch(circle)
...     ax.set_title(time)
...     return fig, circle, ax
...
>>> sys.config_plot_function = plot
>>> sys.plot_configuration()
```

config_plot_update_func

> The configuration plot update function arguments should be any of the system's constants, coordinates, measurements, or 'time' in any order with the returned values from the `config_plot_func` as the last arguments in the exact order as in the configuration plot return statement. No other arguments are valid. Nothing need be returned from the function. See the matplotlib animation documentation for tips on creating these update functions.

Examples

```
>>> sys = SingleDoFLinearSystem()
>>> sys.constants['radius'] = 5.0
>>> sys.constants['center_y'] = 10.0
>>> sys.coordinates['center_x'] = 0.0
>>> def plot(radius, center_x, center_y, time):
...     fig, ax = plt.subplots(1, 1)
```

```
...         circle = Circle((center_x, center_y), radius=radius)
...         ax.add_patch(circle)
...         ax.set_title(time)
...         return fig, circle, ax
...
>>> sys.config_plot_function = plot
>>> def update(center_y, center_x, time, circle, ax):
...         # NOTE : that circle and ax have to be the last arguments and be
...         # in the same order as returned from plot()
...         circle.set_xy((center_x, center_y))
...         ax.set_title(time)
...         fig.canvas.draw()
...
>>> sys.config_plot_update_func = update
>>> sys.animate_configuration()
```

constants

A dictionary containing the all of the system's constants, i.e. parameters that do not vary with time.

Examples

```
>>> sys = System()
>>> sys.constants
{}
>>> sys.constants['mass'] = 5.0
>>> sys.constants
{'mass': 5.0}
>>> del sys.constants['mass']
>>> sys.constants
{}
>>> sys.constants['mass'] = 5.0
>>> sys.constants['length'] = 10.0
>>> sys.constants
{'mass': 5.0, 'length': 10.0}
```

coordinates

A dictionary containing the system's generalized coordinates, i.e. coordinate parameters that vary with time. These values will be used as initial conditions in simulations.

Examples

```
>>> sys = System()
>>> sys.coordinates['angle'] = 0.0
>>> sys.coordinates
{'angle': 0.0}
```

free_response(*final_time*, *initial_time=0.0*, *sample_rate=100*, ***kwargs*)

Returns a data frame with monotonic time values as the index and columns for each coordinate and measurement at the time value for that row. Note that this data frame is stored on the system as the variable `result` until this method is called again, which will overwrite it.

> **Parameters**

> - **final_time** (*float*) – A value of time in seconds corresponding to the end of the simulation.

- **initial_time** (*float, optional*) – A value of time in seconds corresponding to the start of the simulation.

- **sample_rate** (*integer, optional*) – The sample rate of the simulation in Hertz (samples per second). The time values will be reported at the initial time and final time, i.e. inclusive, along with times space equally based on the sample rate.

> **Returns df** – A data frame indexed by time with all of the coordinates and measurements as columns.

> **Return type** pandas.DataFrame

measurements

A dictionary containing the all of the system's measurements, i.e. parameters that are functions of the constants, coordinates, speeds, and other measurements.

plot_configuration ()

Returns a matplotlib figure generated by the function assigned to the `config_plot_func` attribute. You may need to call `matplotlib.pyplot.show()` to display the figure.

> **Returns**

> - **fig** (*matplotlib.figure.Figure*) – The first returned object is always a figure.

> - ***args** (*matplotlib objects*) – Any matplotlib objects can be returned after the figure.

speeds

A dictionary containing the system's generalized speeds, i.e. speed parameters that vary with time. These values will be used as initial conditions in simulations.

Examples

```
>>> sys = System()
>>> sys.speeds['angle_vel'] = 0.0
>>> sys.speeds
{'angle_vel': 0.0}
```

states

An ordered dictionary containing the system's state variables and values. The coordinates are always ordered before the speeds and the individual order of the values depends on the order they were added to coordinates and speeds.

Examples

```
>>> sys = System()
>>> sys.coordinates['angle'] = 0.2
>>> sys.speeds['angle_vel'] = 0.1
>>> sys.states
{'angle': 0.2, 'angle_vel': 0.1}
>>> list(sys.states.keys())
['angle', 'angle_vel']
>>> list(sys.states.values())
[0.2, 0.1]
```

2.2 resonance/linear_systems.py

class resonance.linear_systems.**BaseExcitationSystem**

    Bases: *resonance.linear_systems.SingleDoFLinearSystem*

This system represents a mass connected to a moving massless base via a spring and damper in parallel. The motion of the mass is subject to viscous damping. The system is described by:

constants

    mass, m [kg] The suspended mass.

    damping, c [kg / s] The viscous linear damping coefficient which represents any energy dissipation from things like air resistance, friction, etc.

    stiffness, k [N / m] The linear elastic stiffness of the spring.

coordinates

    position, x [m] The absolute position of the mass.

speeds

    velocity, x_dot [m / s] The absolute velocity of the mass.

periodic_base_displacing_response(*twice_avg*, *cos_coeffs*, *sin_coeffs*, *frequency*, *final_time*, *initial_time=0.0*, *sample_rate=100*, *force_col_name='forcing_function'*, *displace_col_name='displacing_function'*)

Returns the trajectory of the system's coordinates, speeds, accelerations, and measurements if a periodic function defined by a Fourier series is applied as displacement of the base in the same direction as the system's coordinate. The displacing function is defined as:

```
              N
y(t)  = a0 / 2 +  (an * cos(n*ω*t) + bn * sin(n*ω*t))
             n=1
```

Where a0, a1...an, and b1...bn are the Fourier coefficients. If N=∞ then the Fourier series can describe any periodic function with a period $(2*\pi)/\omega$.

    Parameters

- **twice_avg** (*float*) – Twice the average value over one cycle, a0.

- **cos_coeffs** (*float or sequence of floats*) – The N cosine Fourier coefficients: a1, ..., aN.

- **sin_coeffs** (*float or sequence of floats*) – The N sine Fourier coefficients: b1, ..., bN.

- **frequency** (*float*) – The frequency, ω, in radians per second corresponding to one full cycle of the function.

- **final_time** (*float*) – A value of time in seconds corresponding to the end of the simulation.

- **initial_time** (*float, optional*) – A value of time in seconds corresponding to the start of the simulation.

- **sample_rate** (*integer, optional*) – The sample rate of the simulation in Hertz (samples per second). The time values will be reported at the initial time and final time, i.e. inclusive, along with times space equally based on the sample rate.

- **force_col_name** (*string, optional*) – A valid Python identifier that will be used as the column name for the forcing function trajectory in the returned data frame.

- **displace_col_name** (*string, optional*) – A valid Python identifier that will be used as the column name for the forcing function trajectory in the returned data frame.

Returns A data frame indexed by time with all of the coordinates, speeds, measurements, and forcing/displacing functions as columns.

Return type pandas.DataFrame

sinusoidal_base_displacing_response(*amplitude, frequency, final_time, initial_time=0.0, sample_rate=100, force_col_name='forcing_function', displace_col_name='displacing_function'*)

Returns the trajectory of the system's coordinates, speeds, accelerations, and measurements if a sinusoidal displacement function described by:

y(t) = Y * sin(ω*t)

is specified for the movement of the base in the direction of the system's coordinate.

Parameters

- **amplitude** (*float*) – The amplitude of the displacement function, Y, in meters.

- **frequency** (*float*) – The frequency, ω, in radians per second of the sinusoidal displacement.

- **final_time** (*float*) – A value of time in seconds corresponding to the end of the simulation.

- **initial_time** (*float, optional*) – A value of time in seconds corresponding to the start of the simulation.

- **sample_rate** (*integer, optional*) – The sample rate of the simulation in Hertz (samples per second). The time values will be reported at the initial time and final time, i.e. inclusive, along with times space equally based on the sample rate.

- **force_col_name** (*string, optional*) – A valid Python identifier that will be used as the column name for the forcing function trajectory in the returned data frame.

- **displace_col_name** (*string, optional*) – A valid Python identifier that will be used as the column name for the forcing function trajectory in the returned data frame.

Returns A data frame indexed by time with all of the coordinates and measurements as columns.

Return type pandas.DataFrame

class resonance.linear_systems.**BookOnCupSystem**

Bases: *resonance.linear_systems.SingleDoFLinearSystem*

This system represents dynamics of a typical engineering textbook set atop a cylinder (a coffee cup) such that the book can vibrate without slip on the curvature of the cup. It is described by:

constants

thickness, t [meters] the thickness of the book

length, l [meters] the length of the edge of the book which is tangent to the cup's surface

mass, m [kilograms] the mass of the book

radius, r [meters] the outer radius of the cup

coordinates

book_angle, theta [radians] the angle of the book with respect to the gravity vector

speeds

book_angle_vel, theta [radians] the angular rate of the book with respect to the gravity vector

class resonance.linear_systems.**ClockPendulumSystem**

Bases: *resonance.linear_systems.SingleDoFLinearSystem*

This system represents dynamics of a simple compound pendulum in which a rigid body is attached via a revolute joint to a fixed point. Gravity acts on the pendulum to bring it to an equilibrium state and there is no friction in the joint. It is described by:

constants

pendulum_mass, m [kg] The mass of the compound pendulum.

inertia_about_joint, i [kg m2]** The moment of inertia of the compound pendulum about the revolute joint.

joint_to_mass_center, l [m] The distance from the revolute joint to the mass center of the compound pendulum.

acc_due_to_gravity, g [m/s2]** The acceleration due to gravity.

coordinates

angle, theta [rad] The angle of the pendulum relative to the direction of gravity. When theta is zero the pendulum is hanging down in it's equilibrium state.

speeds

angle_vel, theta_dot [rad / s] The angular velocity of the pendulum about the revolute joint axis.

class resonance.linear_systems.**CompoundPendulumSystem**

Bases: *resonance.linear_systems.SingleDoFLinearSystem*

This system represents dynamics of a simple compound pendulum in which a rigid body is attached via a revolute joint to a fixed point. Gravity acts on the pendulum to bring it to an equilibrium state and there is no friction in the joint. It is described by:

constants

pendulum_mass, m [kg] The mass of the compound pendulum.

inertia_about_joint, i [kg m2]** The moment of inertia of the compound pendulum about the revolute joint.

joint_to_mass_center, l [m] The distance from the revolute joint to the mass center of the compound pendulum.

acc_due_to_gravity, g [m/s2]** The acceleration due to gravity.

coordinates

angle, theta [rad] The angle of the pendulum relative to the direction of gravity. When theta is zero the pendulum is hanging down in it's equilibrium state.

speeds

angle_vel, theta_dot [rad / s] The angular velocity of the pendulum about the revolute joint axis.

class resonance.linear_systems.**MassSpringDamperSystem**

Bases: *resonance.linear_systems.SingleDoFLinearSystem*

This system represents dynamics of a mass connected to a spring and damper (dashpot). The mass moves horizontally without friction and is acted on horizontally by the spring and damper in parallel. The system is described by:

constants

    mass, M [kg] The system mass.

    damping, C [kg / s] The viscous linear damping coefficient which represents any energy dissipation from things like air resistance, slip, etc.

    stiffness, K [N / m] The linear elastic stiffness of the spring.

coordinates
    position, x [m]

speeds
    velocity, x_dot [m / s]

class resonance.linear_systems.**SimplePendulumSystem**
    Bases: *resonance.linear_systems.SingleDoFLinearSystem*

This system represents dynamics of a simple pendulum in which a point mass is fixed on a massless pendulum arm of some length to a revolute joint. Gravity acts on the pendulum to bring it to an equilibrium state and there is no friction in the joint. It is described by:

constants

    pendulum_mass, m [kg] The mass of the compound pendulum.

    pendulum_length, l [m] The distance from the revolute joint to the point mass location.

    acc_due_to_gravity, g [m/s2]** The acceleration due to gravity.

coordinates

    angle, theta [rad] The angle of the pendulum relative to the direction of gravity. When theta is zero the pendulum is hanging down in it's equilibrium state.

speeds

    angle_vel, theta_dot [rad / s] The angular velocity of the pendulum about the revolute joint axis.

class resonance.linear_systems.**SimpleQuarterCarSystem**
    Bases: *resonance.linear_systems.BaseExcitationSystem*

This system represents a mass connected to a moving massless base via a spring and damper in parallel. The motion of the mass is subject to viscous damping. The system is described by:

constants

    mass, m [kg] The suspended mass.

    damping, c [kg / s] The viscous linear damping coefficient which represents any energy dissipation from things like air resistance, friction, etc.

    stiffness, k [N / m] The linear elastic stiffness of the spring.

coordinates

    position, x [m] The absolute position of the mass.

speeds

    velocity, x_dot [m / s] The absolute velocity of the mass.

class `resonance.linear_systems.`**`SingleDoFLinearSystem`**

    Bases: *`resonance.system.System`*

This is the abstract base class for any single degree of freedom linear system. It can be sub-classed to make a custom system or the necessary methods can be added dynamically.

`canonical_coeffs_func`

    A function that returns the three linear coefficients of the left hand side of a canonical second order ordinary differential equation. This equation looks like the following for linear motion:

$$mv' + cv + kx = F(t)$$

and like the following for angular motion:

$$I\omega' + c\omega + k\theta = T(t)$$

where:

- m: mass of the moving particle

- I: moment of inertia of a rigid body

- c: viscous damping coefficient (linear or angular)

- k: spring stiffness (linear or angular)

- x: the positional coordinate of the mass

- v: the positional speed of the mass

- θ: the angular coordinate of the body

- ω: the angular speed of the body

The coefficients (m, c, k, I) must be defined in terms of the system's constants.

Example

```
>>> sys = SingleDoFLinearSystem()
>>> sys.constants['gravity'] = 9.8  # m/s**2
>>> sys.constants['length'] = 1.0   # m
>>> sys.constnats['mass'] = 0.5  # kg
>>> sys.coordinates['theta'] = 0.3  # rad
>>> sys.speeds['omega'] = 0.0  # rad/s
>>> def coeffs(gravity, length, mass):
>>>     # Represents a linear model of a simple pendulum:
...     #   m * l**2 ω' + m * g * l * θ = 0
...     I = mass * length**2
...     c = 0.0
...     k = mass * gravity * length
...     return I, c, k
>>> sys.canonical_coeffs_func = coeffs
```

`frequency_response` (*frequencies*, *amplitude*)

    Returns the amplitude and phase shift for simple sinusoidal forcing of the system. The first holds the plot of the coordinate's amplitude as a function of forcing frequency and the second holds a plot of the coordinate's phase shift with respect to the forcing function.

        Parameters

- **`frequencies`** (*array_like, shape(n,)*) –

- **`amplitude`** (*float*) – The value of the forcing amplitude.

Returns

- **amp_curve** (*ndarray, shape(n,)*) – The amplitude values of the coordinate at different frequencies.

- **phase_curve** (*ndarray, shape(n,)*) – The phase shift values in radians of the coordinate relative to the forcing.

frequency_response_plot (*amplitude*, *log=False*, *axes=None*)

Returns an array of two matplotlib axes. The first holds the plot of the coordinate's amplitude as a function of forcing frequency and the second holds a plot of the coordinate's phase shift with respect to the forcing function.

Parameters

- **amplitude** (*float*) – The value of the forcing amplitude.

- **log** (*boolean, optional*) – If True, the amplitude will be plotted on a semi-log Y plot.

period ()

Returns the (damped) period of oscillation of the coordinate in seconds.

periodic_forcing_response (*twice_avg*, *cos_coeffs*, *sin_coeffs*, *frequency*, *final_time*, *initial_time=0.0*, *sample_rate=100*, *col_name='forcing_function'*)

Returns the trajectory of the system's coordinates, speeds, accelerations, and measurements if a periodic forcing function defined by a Fourier series is applied as a force or torque in the same direction as the system's coordinate. The forcing function is defined as:

```
                         N
F(t) or T(t) = a0 / 2 +  (an * cos(n*ω*t) + bn * sin(n*ω*t))
                        n=1
```

Where a0, a1...an, and b1...bn are the Fourier coefficients. If N=∞ then the Fourier series can describe any periodic function with a period $(2*\pi)/\omega$.

Parameters

- **twice_avg** (*float*) – Twice the average value over one cycle, a0.

- **cos_coeffs** (*float or sequence of floats*) – The N cosine Fourier coefficients: a1, ..., aN.

- **sin_coeffs** (*float or sequence of floats*) – The N sine Fourier coefficients: b1, ..., bN.

- **frequency** (*float*) – The frequency, ω, in radians per second corresponding to one full cycle of the function.

- **final_time** (*float*) – A value of time in seconds corresponding to the end of the simulation.

- **initial_time** (*float, optional*) – A value of time in seconds corresponding to the start of the simulation.

- **sample_rate** (*integer, optional*) – The sample rate of the simulation in Hertz (samples per second). The time values will be reported at the initial time and final time, i.e. inclusive, along with times space equally based on the sample rate.

- **col_name** (*string, optional*) – A valid Python identifier that will be used as the column name for the forcing function trajectory in the returned data frame.

Returns A data frame indexed by time with all of the coordinates and measurements as columns.

Return type pandas.DataFrame

sinusoidal_forcing_response(*amplitude*, *frequency*, *final_time*, *initial_time=0.0*, *sample_rate=100*, *col_name='forcing_function'*)

Returns the trajectory of the system's coordinates, speeds, accelerations, and measurements if a sinusoidal forcing (or torquing) function defined by:

$F(t) = Fo * \cos(\omega * t)$

or

$T(t) = To * \cos(\omega * t)$

is applied to the moving body in the direction of the system's coordinate.

> **Parameters**
>
> - **amplitude** (*float*) – The amplitude of the forcing/torquing function, Fo or To, in Newtons or Newton-Meters.
>
> - **frequency** (*float*) – The frequency, ω, in radians per second of the sinusoidal forcing.
>
> - **final_time** (*float*) – A value of time in seconds corresponding to the end of the simulation.
>
> - **initial_time** (*float, optional*) – A value of time in seconds corresponding to the start of the simulation.
>
> - **sample_rate** (*integer, optional*) – The sample rate of the simulation in Hertz (samples per second). The time values will be reported at the initial time and final time, i.e. inclusive, along with times space equally based on the sample rate.
>
> - **col_name** (*string, optional*) – A valid Python identifier that will be used as the column name for the forcing function trajectory in the returned data frame.
>
> **Returns** A data frame indexed by time with all of the coordinates and measurements as columns.
>
> **Return type** pandas.DataFrame

class resonance.linear_systems.**TorsionalPendulumSystem**

Bases: *resonance.linear_systems.SingleDoFLinearSystem*

This system represents dynamics of a simple torsional pendulum in which the torsionally elastic member's axis is aligned with gravity and the axis of the torsion member passes through the mass center of an object attached to it's lower end. The top of the torsion rod is rigidly attached to the "ceiling". It is described by:

constants

> **rotational_inertia, I [kg m**2]** The moment of inertia of the object attached to the pendulum.
>
> **torsional_damping, C [N s / m]** The viscous linear damping coefficient which represents any energy dissipation from things like air resistance, slip, etc.
>
> **torsional_stiffness, K [N / m]** The linear elastic stiffness coefficient of the torsion member, typically a round slender rod.

coordinates

> torsional_angle, theta [rad]

speeds

> torsional_angle_vel, theta_dot [rad / s]

2.3 resonance/nonlinear_systems.py

class resonance.nonlinear_systems.**ClockPendulumSystem**
     Bases: *resonance.nonlinear_systems.SingleDoFNonLinearSystem*

This system represents dynamics of a compound pendulum representing a clock pendulum. It is made up of a thin long cylindrical rod with a thin disc bob on the end. Gravity acts on the pendulum to bring it to an equilibrium state and there is option Coulomb friction in the joint. It is described by:

constants

> **bob_mass, m_b [kg]** The mass of the bob (a thin disc) on the end of the pendulum.
>
> **bob_radius, r [m]** The radius of the bob (a thin disc) on the end of the pendulum.
>
> **rod_mass, m_r [kg]** The mass of the then cylindrical rod.
>
> **rod_length, l [m]** The length of the rod which connects the pivot joint to the center of the bob.
>
> **coeff_of_friction, mu [unitless]** The Coulomb coefficient of friction between the materials of the pivot joint.
>
> **joint_friction_radius, R [m]** The radius of the contact disc at the pivot joint. The joint is assumed to be two flat discs pressed together.
>
> **joint_clamp_force, F_N [N]** The clamping force pressing the two flat discs together at the pivot joint.
>
> **acc_due_to_gravity, g [m/s**2]** The acceleration due to gravity.

coordinates

> **angle, theta [rad]** The angle of the pendulum relative to the direction of gravity. When theta is zero the pendulum is hanging down in it's equilibrium state.

speeds

> **angle_vel, theta_dot [rad / s]** The angular velocity of the pendulum about the revolute joint axis.

class resonance.nonlinear_systems.**MultiDoFNonLinearSystem**
     Bases: *resonance.system.System*

This is the abstract base class for any single degree of freedom nonlinear system. It can be sub-classed to make a custom system or the necessary methods can be added dynamically.

diff_eq_func

> A function that returns the time derivatives of the coordinate and speed, i.e. computes the right hand side of the two order differential equations. This equation looks like the following for linear motion:
>
> dy – = f(t, q1, ..., qn, u1, ..., un, p1, p2, ..., pO) dt
>
> where:
>
> - t: a time value
>
> - q: the coordinates
>
> - u: the speeds
>
> - p: any number of constants or measurements, O is the number of constants.

Notes

Your function should be able to operate on 1d arrays as inputs, i.e. use numpy math functions in your function, e.g. `numpy.sin` instead of `math.sin`.

The function has to return the derivatives of the states in the order of the `state` attribute.

Example

```
>>> sys = SingleDoFNonLinearSystem()
>>> sys.constants['gravity'] = 9.8  # m/s**2
>>> sys.constants['length'] = 1.0  # m
>>> sys.constants['mass'] = 0.5  # kg
>>> sys.coordinates['theta'] = 0.3  # rad
>>> sys.speeds['omega'] = 0.0  # rad/s
>>> def rhs(theta, omega, gravity, length, mass):
>>>     # Represents a linear model of a simple pendulum:
...     #  m * l**2 ω' + m * g * l * sin(θ) = 0
...     thetad = omega
...     omegad = (-m*g*l*np.sin(theta)) / m / l**2
...     return thetad, omegad  # in order of sys.states
>>> sys.diff_eq_func = rhs
```

class resonance.nonlinear_systems.**SingleDoFNonLinearSystem**

> Bases: *resonance.nonlinear_systems.MultiDoFNonLinearSystem*

2.4 resonance/functions.py

resonance.functions.**estimate_period**(*time*, *signal*)

> Computes the period of oscillation based on the given periodic signal.
>
> **Parameters**
>
> - **time** (*array_like, shape(n,)*) – An array of monotonically increasing time values.
>
> - **signal** (*array_like, shape(n,)*) – An array of values for the periodic signal at each time in `t`.
>
> **Returns** period – An estimate of the period of oscillation.
>
> **Return type** float

Indices and tables

- genindex
- modindex
- search

Python Module Index

r

Index