BicycleParameters Documentation

Release 1.2.0.dev0

BicycleParameters Authors

Apr 27, 2024

CONTENTS

The **bicycleparameters** package is a python program designed to produce and manipulate the basic parameters needed for basic bicycle dynamic models.

CHAPTER

ONE

TABLE OF CONTENTS

1.1 Description

This is code based off of the work done to measure the physical parameters of a bicycle and rider at both the UCD Sports Biomechanics Lab and the TU Delft Bicycle Dynamics Lab. Physical parameters include but are not limited to the geometry, mass, mass location and mass distribution of the bicycle rider system. The code is structured around the Whipple bicycle model and fundamentally works with and produces the parameters presented in Meijaard 2007 [Meijaard2007], due to the fact that these parameters have been widely adopted as a benchmark. But the software is also capable of generating parameter sets for more complex rider biomechanical models. More detail can be found in our papers and the website and in *References*.

1.1.1 Features

Parameter Manipulation

- Loads bicycle parameter sets from a text file into a python object.
- Generates the benchmark parameter set for a real bicycle from experimental data.
- Generates the rider parameter set from human measurements based on the Yeadon model configured to sit on the bicycle.
- Plots a descriptive drawing of the bicycle and/or rider.
- Generates publication quality tables of parameters.

Basic Linear Analysis

- Calculates the A and B matrices for the Whipple bicycle model linearized about the upright configuration.
- Calculates the canonical matrices for the Whipple bicycle model linearized about the upright configuration.
- Calculates the eigenvalues for the Whipple bicycle model linearized about the upright configuration.
- Plots the eigenvalue root loci as a function of speed as eigenvalue vs speed.
- Plots Bode diagrams of the open loop transfer functions.

Refer to *Example Usage* for examples of the features.

1.1.2 Upcoming Features

- Converts benchmark parameters to other parametrizations.
- Calculates the transfer functions of the open loop system.

1.1.3 Example Code

```
>>> import bicycleparameters as bp
>>> import numpy as np
>>> rigid = bp.Bicycle('Rigid')
>>> par = rigid.parameters['Benchmark']
>>> rigid.plot_bicycle_geometry()
>>> speeds = np.linspace(0., 10., num=100)
>>> rigid.plot_eigenvalues_vs_speed(speeds, show=True)
```

1.1.4 References

The methods associated with this software were built upon these previous works, among others.

1.2 Bicycleparameters Installation

1.2.1 Dependencies

These are the versions that I tested the code with, but the code will most likely work with older versions.

Required

- DynamicistToolKit >= 0.5.3
- Matplotlib >= 3.5.1
- NumPy >= 1.21.5
- Python >= 3.8
- SciPy >= 1.8.0
- Uncertainties >= 3.1.5
- yeadon >= 1.3.0

Optional

These are required to run the Dash web application:

- Dash >= 2.0
- dash-bootstrap-components
- Pandas >= 1.3.5

These are required to build the documentation:

- Sphinx >= 4.3.2
- Numpydoc >= 1.2

1.2.2 Installation

We recommend installing BicycleParameters with conda or pip.

For conda:

```
$ conda install -c conda-forge bicycleparameters
```

For pip:

```
$ pip install BicycleParameters
```

The package can also be installed from the source code. The options for getting the source code are:

1. Clone the source code with Git: `git clone git://github.com/moorepants/BicycleParameters.git`
2. Download the source from Github.
3. Download the source from pypi.

Once you have the source code navigate to the directory and run:

```
>>> python setup.py install
```

This will install the software into your system. You can check if it installs with:

```
$ python -c "import bicycleparameters"
```

1.3 Example Usage

1.3.1 The Data

The program requires input data in the form of basic text files and a particular file system directory structure to organize the data. To use the program you will need to navigate to a directory where you have at least one directory named for a bicycle that contains parameters files or raw data measurements. Refer to the document *BicycleParameters Data File Information* for details about what the data files should contain.

Data Directory

You will need to setup a directory (a directory named `data` is used in the following examples) somewhere for the data input and output files. The structure of the directory should look like this:

```
/data
|
-->/bicycles
|  |
|  -->/Bicyclea
|  |  |
|  |  -->/Parameters
|  |  |
|  |  -->/Photos
|  |  |
|  |  -->/RawData
|  |
|  -->/Bicycleb
|     |
|     -->/Parameters
|     |
|     -->/Photos
|     |
|     -->/RawData
-->/riders
   |
   -->/Ridera
      |
      -->/Parameters
      |
      -->/RawData
```

Bicycle/rider name

A bicycle or rider name is a descriptive word (or compound word) for a bicycle or rider in which the first letter is capitalized. Examples of bicycle short names include `Bianchipista`, `Bike`, `Mybike`, `Rigidrider`, `Schwintandem`, `Gyrobike`, `Bicyclea`, etc. Examples of rider names include `Jason`, `Mont`, `Lukepeterson`, etc. The program relies on CamelCase words, so make sure the first letter is capitalized and no others are.

bicycles Directory

The `bicycles` directory contains subdirectories for each bicycle. The directory name for a bicycle should be its bicycle name. Each directory in `bicycles` should contain at least a `RawData` directory or a `Parameters` directory. `Photos` is an optional directory.

RawData directory

You can supply raw measurement data in two forms:

1. A file containing all the manual measurements (including the oscillation periods for each rigid body). Refer to *bicycles/<bicycle name>/RawData/<bicycle name>Measured.txt* for details about the contents of this file.
2. A file containing all the manual measurements (not including the oscillation periods for each rigid body) and a set of data files containing oscillatory signals from which the periods can be estimated. Refer to *Pendulum Data Files* for details about these files.

The manual measurement data file should follow the naming convention `<bicycle name>Measure.txt`. This data is used to generate parameter files that can be saved to the `Parameters` directory.

Parameters directory

If you don't have any raw measurements for the bicycle it is also an option to supply a parameter file in the `Parameters` directory. Simply add a file named `<bicycle name>Benchmark.txt` with the benchmark parameter set into the `Parameters` directory for the particular bicycle. Refer to *bicycles/<bicycle name>/Parameters/<bicycle name>Benchmark.txt* for details about the contents of the file.

Photos directory

The `Photos` folder should contain photos of the bicycle parts hung as the various pendulums in the various orientations. The filename should follow the conventions of the raw signal data files.

riders directory

The riders directory should contain a directory for each rider that you have data for. The individual rider directory contains a `Parameters` for the rider inertial parameters sets and a `RawData` directory for the raw measurements needed for the Yeadon inertia model. Refer to *riders/<rider name>/Parameters/* for details about these input files.

Example Data

Example data is available here:

http://dx.doi.org/10.6084/m9.figshare.1198429

1.3.2 Loading bicycle data

The easiest way to load a bicycle is:

```
>>> import bicycleparameters as bp
>>> bicycle = bp.Bicycle('Stratos')
```

This will create an instance of the Bicycle class in the variable bicycle based off of input data from the `./bicycles/Stratos/` directory. The program first looks to see if there are any parameter sets in `./bicycles/Stratos/Parameters/`. If so, it loads the data, if not it looks for `./bicycles/Stratos/RawData/StratosMeasurments.txt` so that it can generate the parameter set. The raw measurement file may or may not contain the oscillation period data for the bicycle moment of inertia caluclations. If it doesn't then the program will look for the series of `.mat` files need to calculate the periods. If no data is there, then you get an error.

There are other loading options:

```
>>> bicycle = bp.Bicycle('Stratos', pathToData='<some path to the data directory>',␣
↪forceRawCalc=True, forcePeriodCalc=True)
```

The `pathToData` option allows you specify a directory other than the current directory as your data directory. The `forceRawCalc` forces the program to load `./bicycles/Stratos/RawData/StratosMeasurments.txt` and recalculate the parameters regarless if there are any parameter files available in `./bicycles/Stratos/Parameters/`. The `forcePeriodCalc` option forces the period calcluation from the `.mat` files regardless if they already exist in the raw measurement file.

1.3.3 Exploring bicycle parameter data

The bicycle has a name:

```
>>> bicycle.bicycleName
'Stratos'
```

and a directory where its data is stored:

```
>>> bicycle.direcotory
'./bicycles/Stratos'
```

The benchmark bicycle parameters are the fundamental parameter set that is used behind the scenes for calculations. To access them type:

```
>>> bPar = bicycle.parameters['Benchmark']
>>> bPar['xB']
0.32631503794489763+/-0.0032538862692938642
```

The program automatically calculates the uncertainties in the parameters based on the raw measurements or the uncertainties provided in the parameter files. If you'd like to work with the pure values you can remove them:

```
>>> bParPure = bp.io.remove_uncertainties(bPar)
>>> bParPure['xB']
0.32631503794489763
```

That goes the same for all values with uncertainties. Check out the uncertainties package details for more ways to manipulate the quantities.

If the bicycle was calculated from raw data measurements you can access them by:

```
>>> bicycle.parameters['Measurements']
```

All parameter sets are stored in the parameter dictionary of the bicycle instance.

To modify a parameter type:

```
>>> bicycle.parameters['Benchmark']['mB'] = 50.
```

You can regenerate the parameter sets from the raw data stored in the bicycle's directory by calling:

```
>>> bicycle.calculate_from_measured()
```

1.3.4 Basic Analysis

The program has some basic bicycle analysis tools based on the Whipple bicycle model which has been linearized about the upright configuration.

The canonical matrices for the equations of motion can be computed:

```
>>> M, C1, K0, K2 = bicycle.canonical()
>>> M
array([[4.87735569387+/-0.0239343413077, 0.407911475492+/-0.00852495589396],
       [0.407911475492+/-0.00852495589396,
        0.203245633856+/-0.00235820505536]], dtype=object)
>>> C1
array([[0.0, 4.85200252888+/-0.0242948940194],
       [-0.488808930325+/-0.00358710467251,
        0.751423298199+/-0.0118190412791]], dtype=object)
>>> K0
array([[-8.1786550655+/-0.0281976329402,
        -0.709791925937+/-0.013158888468],
       [-0.709791925937+/-0.013158888468,
        -0.206338069868+/-0.00571395841832]], dtype=object)
>>> K2
array([[0.0,
        8.39212115462+/-0.0313979563061],
       [0.0,
        0.778591689057+/-0.0128042478172]], dtype=object))
```

as well as the state and input matrices for state space form at a particular speed (1.34 m/s):

```
>>> A, B = bicycle.state_space(1.34)
>>> A
array([[0.0, 0.0, 1.0, 0.0],
       [0.0, 0.0, 0.0, 1.0],
       [16.324961319+/-0.039204516678, -2.30677907291+/-0.00824125009025,
        -0.323894489886+/-0.00578005141184,
        -1.10401174487+/-0.00684136323415],
       [1.49533216875+/-0.153839831788, 7.71036924174+/-0.170699435465,
        3.8727732103+/-0.0342265538607, -2.73840155487+/-0.0133666010323]], dtype=object)
>>> B
array([[0.0, 0.0],
       [0.0, 0.0],
```

(continues on next page)

(continued from previous page)

```
       [0.246385378456+/-0.00169761878443,
        -0.494492409794+/-0.00770906162244],
       [-0.494492409794+/-0.00770906162244, 5.91259504914+/-0.0401866728435]],␣
↪dtype=object)
```

You can calculate the eigenvalues and eigenvectors at any speed by calling:

```
>>> w, v = bicycle.eig(4.28) # the speed should be in meters/second
>>> w # eigenvalues
array([[-6.83490195+0.j        ,  0.46085314+2.77336727j,
         0.46085314-2.77336727j, -1.58257375+0.j        ]])
>>> v # eigenvectors
array([[[ 0.04283049+0.j        ,  0.50596715+0.33334818j,
          0.50596715-0.33334818j,  0.55478588+0.j        ],
        [ 0.98853840+0.j        ,  0.72150298+0.j        ,
          0.72150298+0.j        ,  0.63786241+0.j        ],
        [-0.00626644+0.j        ,  0.14646768-0.15809917j,
          0.14646768+0.15809917j, -0.35055926+0.j        ],
        [-0.14463096+0.j        ,  0.04206844-0.25316359j,
          0.04206844+0.25316359j, -0.40305383+0.j        ]]])
```

The `eig` function also accepts a one dimensional array of speeds and returns eigenvalues for all speeds. Note that uncertainty propagation into the eigenvalue calculations is not supported yet.

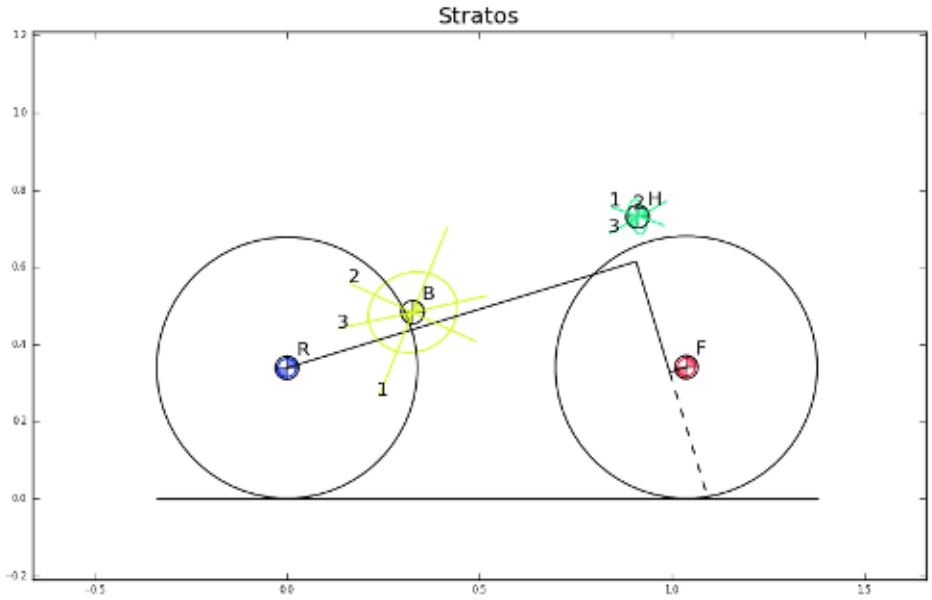
The moment of inertia of the steer assembly (handlebar, fork and/or front wheel) can be computed either about the center of mass or a point on the steer axis, both with reference to a frame aligned with the steer axis:

```
>>> bicycle.steer_assembly_moment_of_inertia(aboutSteerAxis=True)
array([[0.539931205836+/-0.00362870864185, 0.0,
        0.00921422347873+/-0.00191753741975],
       [0.0, 0.578940852064+/-0.00311525776442, 0.0],
       [0.00921422347873+/-0.00191753741975, 0.0,
        0.143206097868+/-0.00100279291208]], dtype=object)
```

Plots

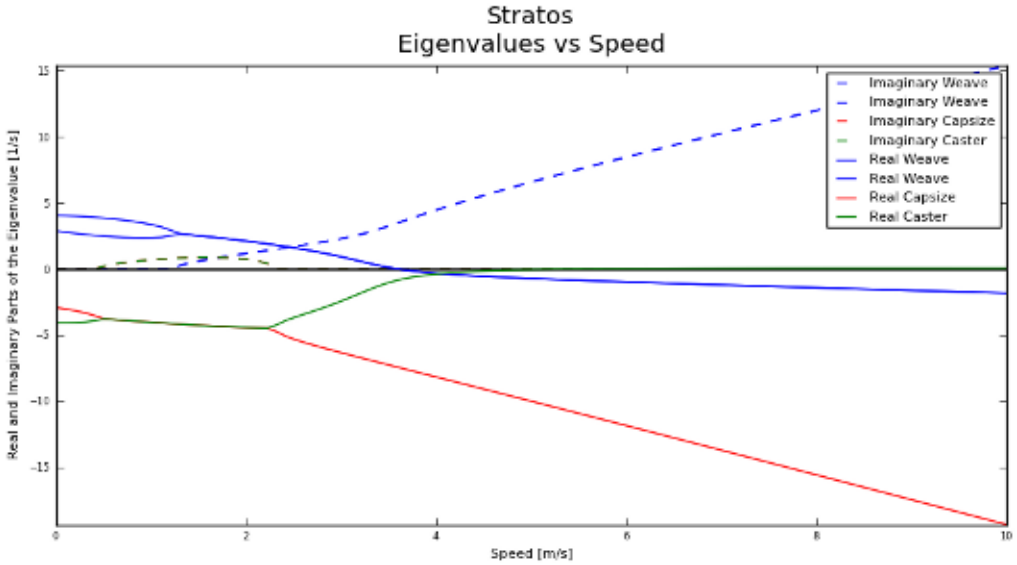
You can plot the geometry of the bicycle and include the mass centers of the various bodies, the inertia ellipsoids and the torsional pendulum axes from the raw measurement data:

```
>>> bicycle.plot_bicycle_geometry()
```



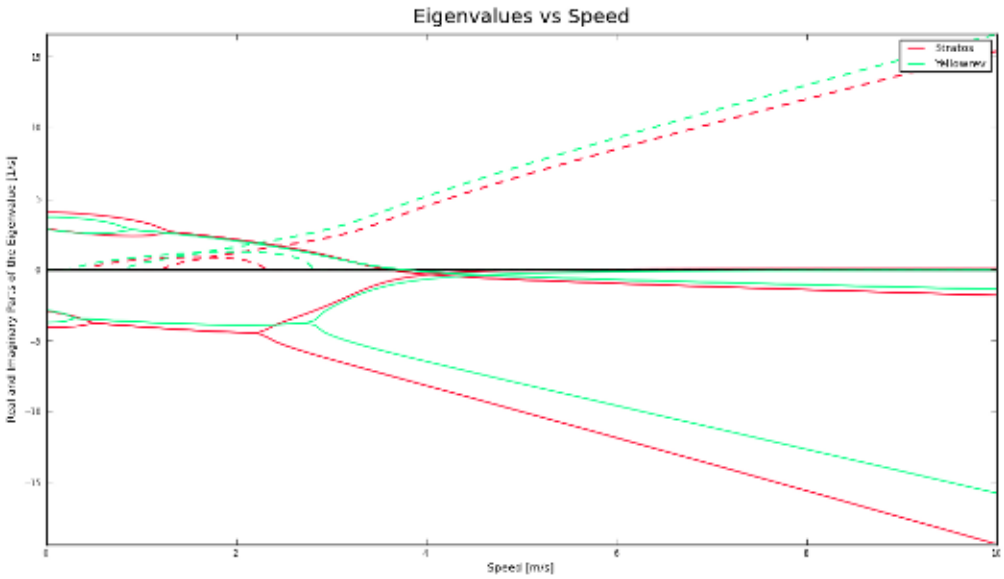
For visualization of the linear analysis you can plot the root loci of the real and imaginary parts of the eigenvalues as a function of speed:

```
>>> import numpy as np
>>> speeds = np.linspace(0., 10., num=100)
>>> bicycle.plot_eigenvalues_vs_speed(speeds, show=True)
```



You can also compare the eigenvalues of two or more bicycles:

```
>>> yellowrev = bp.Bicycle('Yellowrev')
>>> bp.plot_eigenvalues([bicycle, yellowrev], speeds, show=True)
```



Tables

You can generate reStructuredText tables of the bicycle parameters with the `Table` class:

```
>>> from bicycleparameters import tables
>>> tab = tables.Table('Measured', False, bicycle, yellowrev)
>>> rst = tab.create_rst_table()
>>> print rst
+----------+-----------------+-----------------+
|          | Stratos         | Yellowrev       |
+==========+=========+=======+=========+=======+
| Variable | v       | sigma | v       | sigma |
+----------+---------+-------+---------+-------+
| IBxx     | 0.373   | 0.002 | 0.2254  | 0.0009|
+----------+---------+-------+---------+-------+
| IBxz     | -0.0383 | 0.0004| 0.0179  | 0.0001|
+----------+---------+-------+---------+-------+
| IByy     | 0.717   | 0.003 | 0.388   | 0.005 |
+----------+---------+-------+---------+-------+
| IBzz     | 0.455   | 0.002 | 0.2147  | 0.0009|
+----------+---------+-------+---------+-------+
| IFxx     | 0.0916  | 0.0004| 0.0852  | 0.0003|
+----------+---------+-------+---------+-------+
```

(continues on next page)

(continued from previous page)

```
| IFyy     | 0.157   | 0.001  | 0.147   | 0.002  |
+----------+---------+--------+---------+--------+
| IHxx     | 0.1768  | 0.0008 | 0.1475  | 0.0006 |
+----------+---------+--------+---------+--------+
| IHxz     | -0.0273 | 0.0006 | -0.0172 | 0.0005 |
+----------+---------+--------+---------+--------+
| IHyy     | 0.144   | 0.002  | 0.120   | 0.002  |
+----------+---------+--------+---------+--------+
| IHzz     | 0.0446  | 0.0003 | 0.0294  | 0.0004 |
+----------+---------+--------+---------+--------+
| IRxx     | 0.0939  | 0.0004 | 0.0877  | 0.0004 |
+----------+---------+--------+---------+--------+
| IRyy     | 0.154   | 0.001  | 0.149   | 0.001  |
+----------+---------+--------+---------+--------+
| c        | 0.056   | 0.002  | 0.180   | 0.002  |
+----------+---------+--------+---------+--------+
| g        | 9.81    | 0.01   | 9.81    | 0.01   |
+----------+---------+--------+---------+--------+
| lam      | 0.295   | 0.003  | 0.339   | 0.003  |
+----------+---------+--------+---------+--------+
| mB       | 7.22    | 0.02   | 3.31    | 0.02   |
+----------+---------+--------+---------+--------+
| mF       | 3.33    | 0.02   | 1.90    | 0.02   |
+----------+---------+--------+---------+--------+
| mH       | 3.04    | 0.02   | 2.45    | 0.02   |
+----------+---------+--------+---------+--------+
| mR       | 3.96    | 0.02   | 2.57    | 0.02   |
+----------+---------+--------+---------+--------+
| rF       | 0.3400  | 0.0001 | 0.3419  | 0.0001 |
+----------+---------+--------+---------+--------+
| rR       | 0.3385  | 0.0001 | 0.3414  | 0.0001 |
+----------+---------+--------+---------+--------+
| w        | 1.037   | 0.002  | 0.985   | 0.002  |
+----------+---------+--------+---------+--------+
| xB       | 0.326   | 0.003  | 0.412   | 0.004  |
+----------+---------+--------+---------+--------+
| xH       | 0.911   | 0.004  | 0.919   | 0.005  |
+----------+---------+--------+---------+--------+
| zB       | -0.483  | 0.003  | -0.618  | 0.004  |
+----------+---------+--------+---------+--------+
| zH       | -0.730  | 0.002  | -0.816  | 0.002  |
+----------+---------+--------+---------+--------+
```

Which renders in Sphinx like:

	Stratos		Yellowrev	
Variable	v	sigma	v	sigma
IBxx	0.373	0.002	0.2254	0.0009
IBxz	-0.0383	0.0004	0.0179	0.0001
IByy	0.717	0.003	0.388	0.005
IBzz	0.455	0.002	0.2147	0.0009
IFxx	0.0916	0.0004	0.0852	0.0003
IFyy	0.157	0.001	0.147	0.002
IHxx	0.1768	0.0008	0.1475	0.0006
IHxz	-0.0273	0.0006	-0.0172	0.0005
IHyy	0.144	0.002	0.120	0.002
IHzz	0.0446	0.0003	0.0294	0.0004
IRxx	0.0939	0.0004	0.0877	0.0004
IRyy	0.154	0.001	0.149	0.001
c	0.056	0.002	0.180	0.002
g	9.81	0.01	9.81	0.01
lam	0.295	0.003	0.339	0.003
mB	7.22	0.02	3.31	0.02
mF	3.33	0.02	1.90	0.02
mH	3.04	0.02	2.45	0.02
mR	3.96	0.02	2.57	0.02
rF	0.3400	0.0001	0.3419	0.0001
rR	0.3385	0.0001	0.3414	0.0001
w	1.037	0.002	0.985	0.002
xB	0.326	0.003	0.412	0.004
xH	0.911	0.004	0.919	0.005
zB	-0.483	0.003	-0.618	0.004
zH	-0.730	0.002	-0.816	0.002

1.3.5 Rigid Rider

The program also allows one to add the inertial affects of a rigid rider to the Whipple bicycle system.

Rider Data

You can provide rider data in one of two ways, much in the same way as the bicycle. If you have the inertial parameters of a rider, e.g. Jason, simply add a file into the `./riders/Jason/Parameters/` directory. Or if you have raw measurements of the rider add the two files to `./riders/Jason/RawData/`. The yeadon documentation explains how to collect the data for a rider.

Adding a Rider

To add a rider key in:

```
>>> bicycle.add_rider('Jason')
```

The program first looks for a parameter for for Jason sitting on the Stratos and if it can't find one, it looks for the raw data for Jason and computes the inertial parameters. You can force calculation from raw data with:

```
>>> bicycle.add_rider('Jason', reCalc=True)
```

Exploring the rider

The bicycle has a few new attributes now that it has a rider:

```
>>> bicycle.hasRider
True
>>> bicycle.riderName
'Jason'
>>> bicycle.riderPar # inertial parmeters of the rider
{'Benchmark': {'IBxx': 7.8188533619237681,
               'IBxz': -0.035425693766513083,
               'IByy': 8.2729669089020437,
               'IBzz': 2.354736583109867,
               'mB': 79.152920866435153,
               'xB': 0.46614935153554904,
               'yB': 2.1457815736317352e-07,
               'zB': -1.0385521459829261}}
>>> bicycle.human # this is a yeadon.human object representing the Jason
<yeadon.human.human instance at 0x2b19dd0>
```

The bicycle parameters now reflect that a rigid rider has been added to the bicycle frame:

```
>>> bicycle.parameters['Benchmark']['mB']
86.37292086643515+/-0.02
```

At this point, the uncertainties don't necessarily offer much information for any of the parameters that are functions of the rider, because we do not have a good idea of the uncertainty in the human inertia calculations in the Yeadon method.

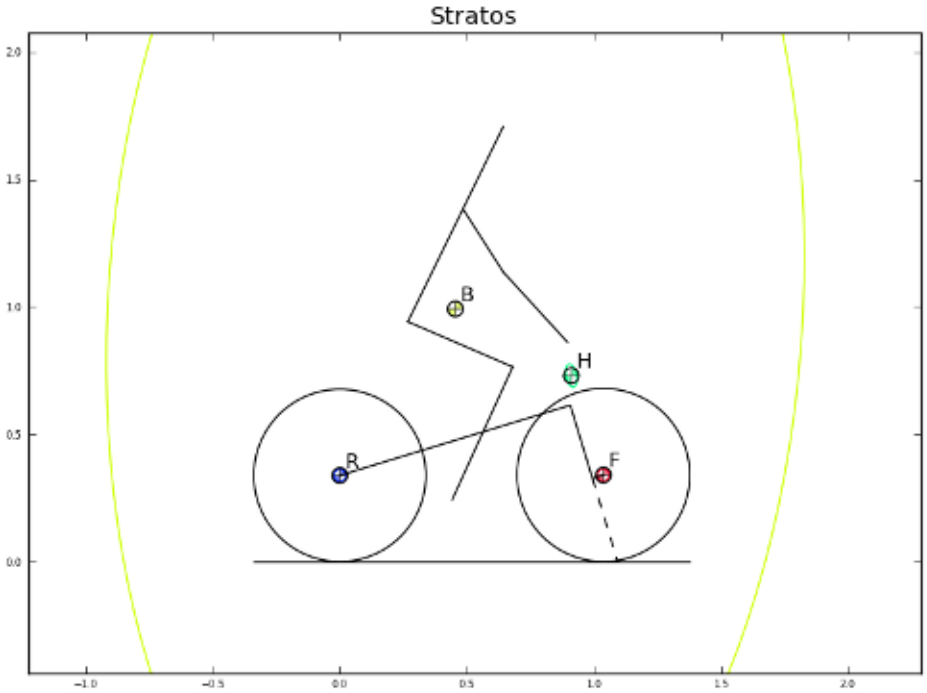
Analysis

The same linear analysis can be performed now that a rider has been added, albeit the reported values and graphs will reflect the fact that the bicycle frame has the added inertial effects of the rider.

Plots

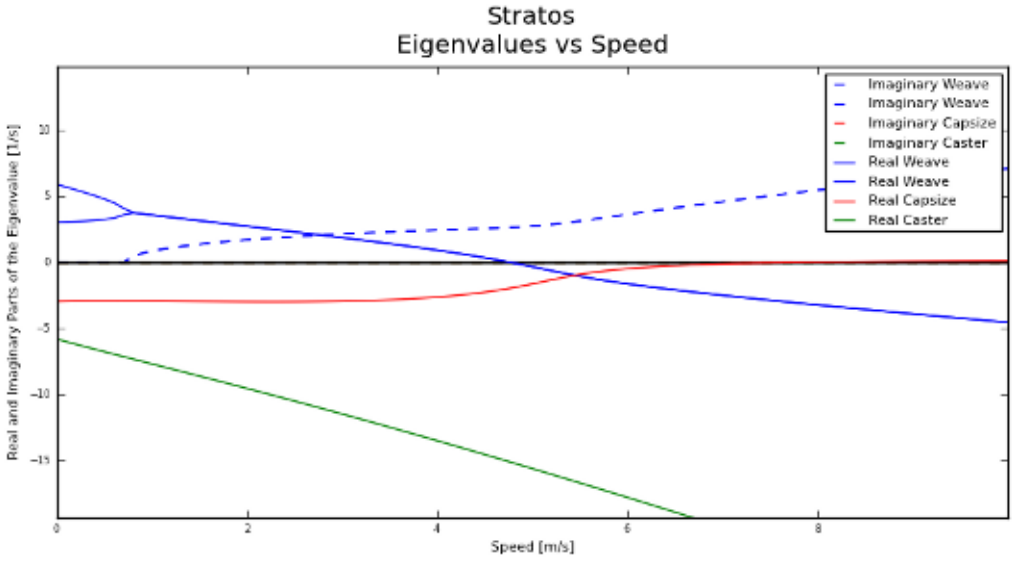
The bicycle geometry plot now reflects that there is a rider on the bicycle and displays a simplified depiction:

```
>>> bicycle.plot_bicycle_geometry()
```



The eigenvalue plot also relfects the changes:

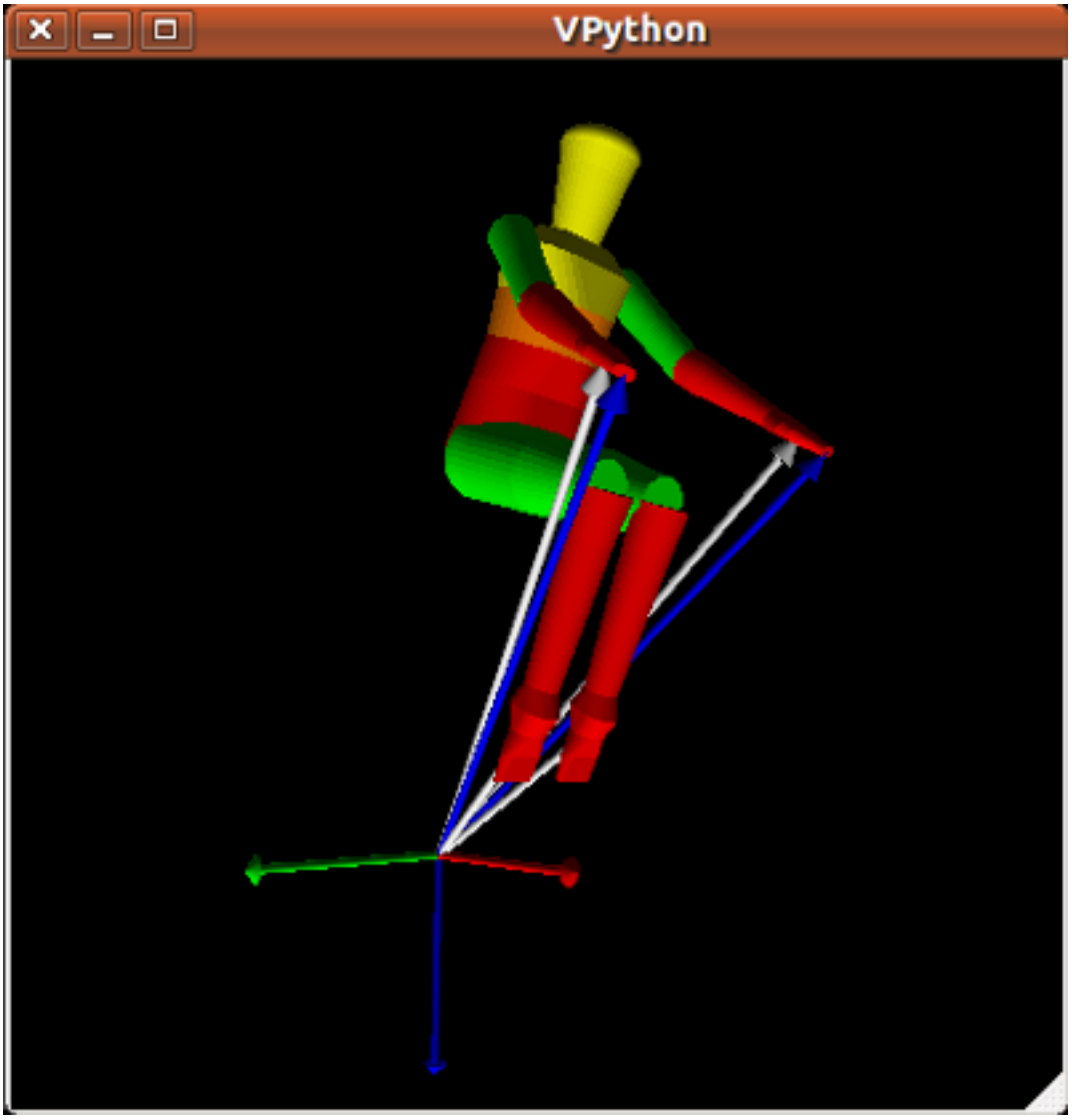
```
>>> bicycle.plot_eigenvalues_vs_speed(speeds, show=True)
```



Rider Visualization

If you have the optional dependency, visual python, for yeadon installed then you can output a three dimensional picture of the Yeadon model configured to be seated on the bicycle. This is a bit buggy due to the nature of visual python, but is useful none-the-less.:

```
>>> bicycle.add_rider('Jason', draw=True)
```



1.3.6 Using Models and Parameter Sets

Parameter Sets

Parameter sets represent a set of constants in a multibody dynamics model. These constants have a name and an associated floating point value. This mapping from name to value is stored in a dictionary and then passed to a `ParameterSet`. Below are the parameters for the Meijaard et al. 2007 paper with some realistic initial values.

```
par = {
    'IBxx': 11.3557360401,
    'IBxz': -1.96756380745,
    'IByy': 12.2177848012,
    'IBzz': 3.12354397008,
    'IFxx': 0.0904106601579,
    'IFyy': 0.149389340425,
    'IHxx': 0.253379594731,
    'IHxz': -0.0720452391817,
    'IHyy': 0.246138810935,
    'IHzz': 0.0955770796289,
    'IRxx': 0.0883819364527,
    'IRyy': 0.152467620286,
    'c': 0.0685808540382,
    'g': 9.81,
    'lam': 0.399680398707,
    'mB': 81.86,
    'mF': 2.02,
    'mH': 3.22,
    'mR': 3.11,
    'rF': 0.34352982332,
    'rR': 0.340958858855,
    'v': 1.0,
    'w': 1.121,
    'xB': 0.289099434117,
    'xH': 0.866949640247,
    'zB': -1.04029228321,
    'zH': -0.748236400835,
}
```

The associated parameter set can be created with this dictionary:

```
from bicycleparameters.parameter_sets import Meijaard2007ParameterSet

par_set = Meijaard2007ParameterSet(par, True)
```

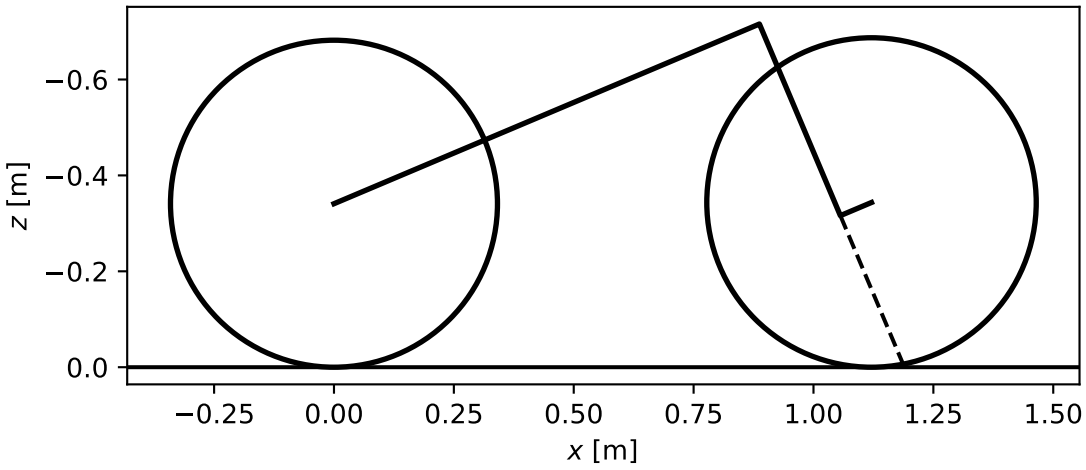
Once the parameter set is available there are various methods that help you calculate and visualize the properties of this parameter set. This set describes the geometry, mass, and inertia of a bicycle. You can plot the geometry like so:

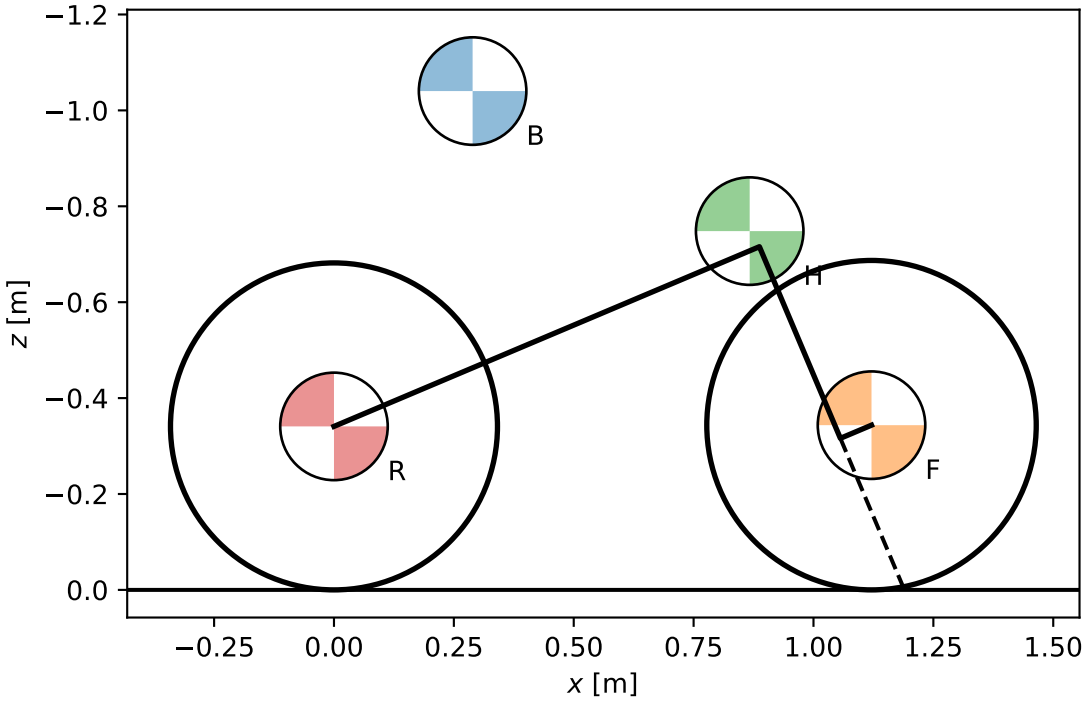
```
par_set.plot_geometry()
```

You can then add symbols representing the mass centers of the four bodies like so:

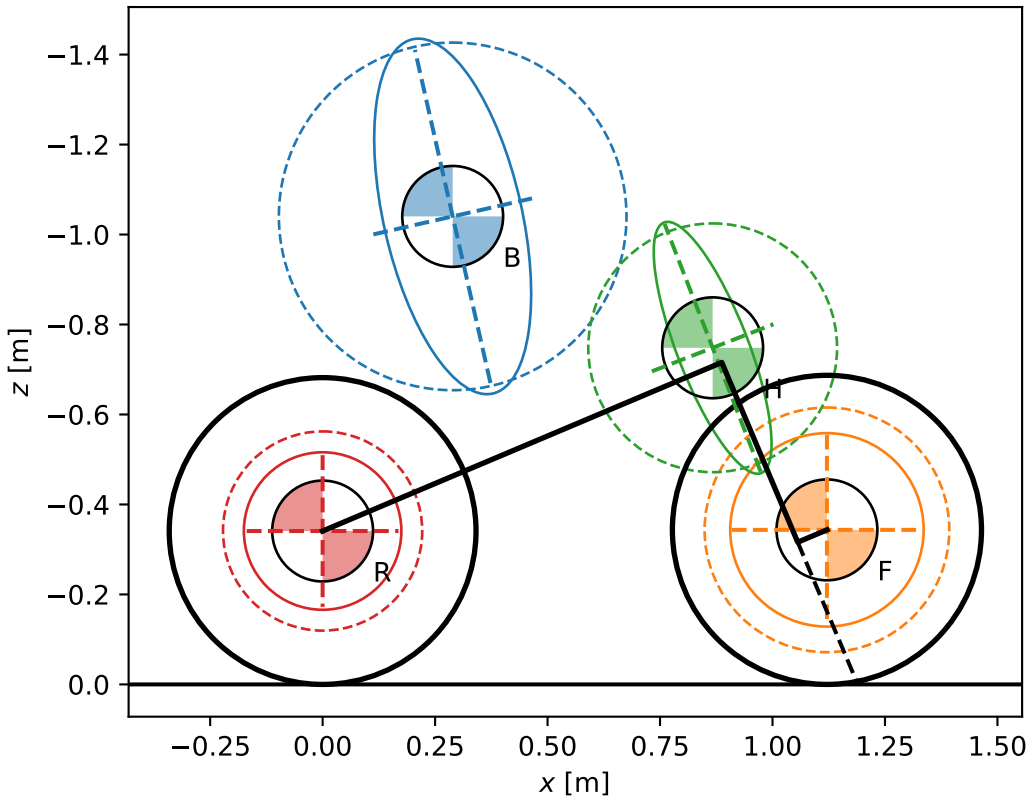
```
ax = par_set.plot_geometry()
par_set.plot_mass_centers(ax=ax)
```

The geometry, mass, and inertial information can all be plotted:


−0.6
−0.4
−0.2
0.0
z [m]
−0.25
0.00
0.25
0.50
0.75
1.00
1.25
1.50
x [m]


−1.2
−1.0
−0.8
−0.6
−0.4
−0.2
0.0
z [m]
B
H
R
F
−0.25
0.00
0.25
0.50
0.75
1.00
1.25
1.50
x [m]

```
par_set.plot_all()
```



Models

Parameter sets can be associated with a model and the model can be used to compute and visualize properties of the model's dynamics.

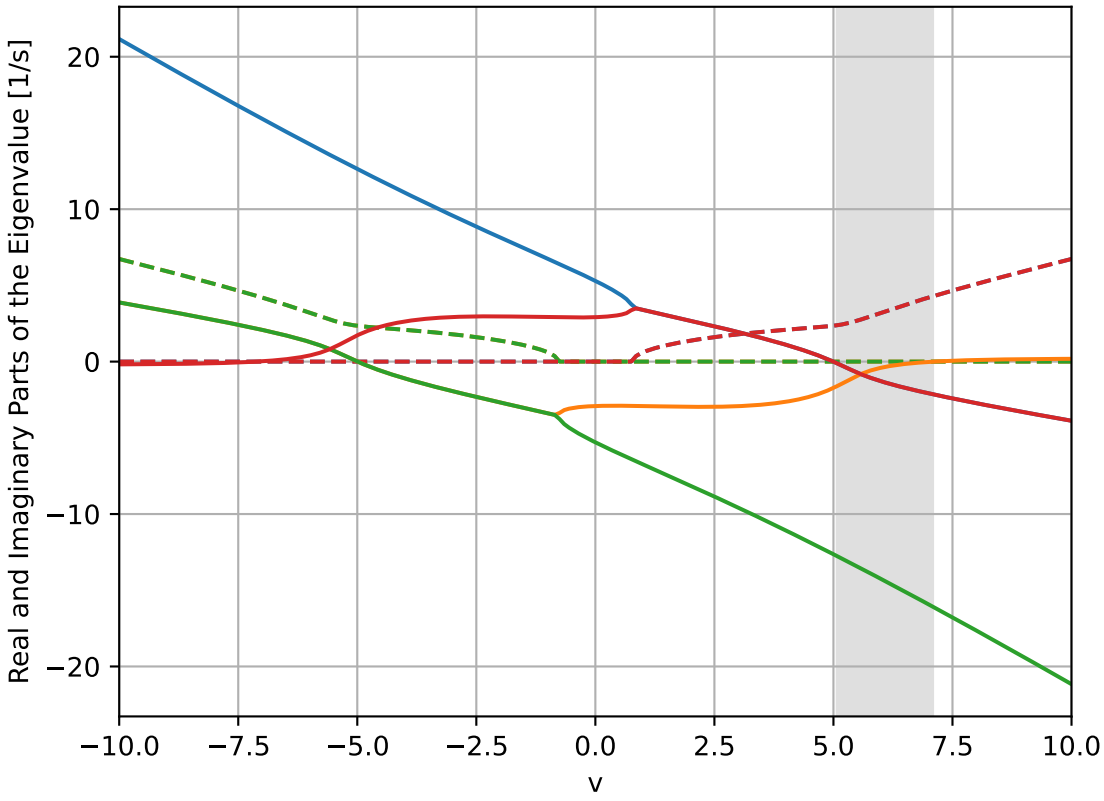
```
from bicycleparameters.models import Meijaard2007Model

model = Meijaard2007Model(par_set)
```

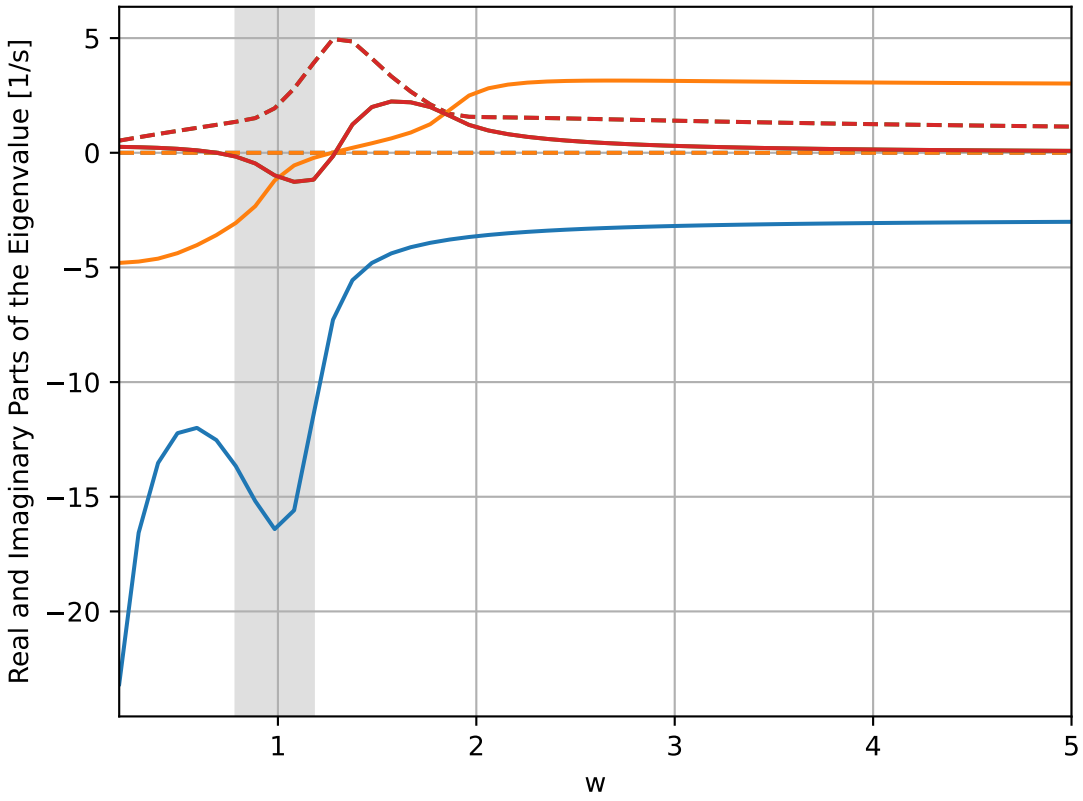
The root locus with respect to any parameter, for example speed `v`, can be plotted:

```
speeds = np.linspace(-10.0, 10.0, num=200)

model.plot_eigenvalue_parts(v=speeds)
```

You can choose any parameter in the dictionary to generate the root locus and also override other parameters.

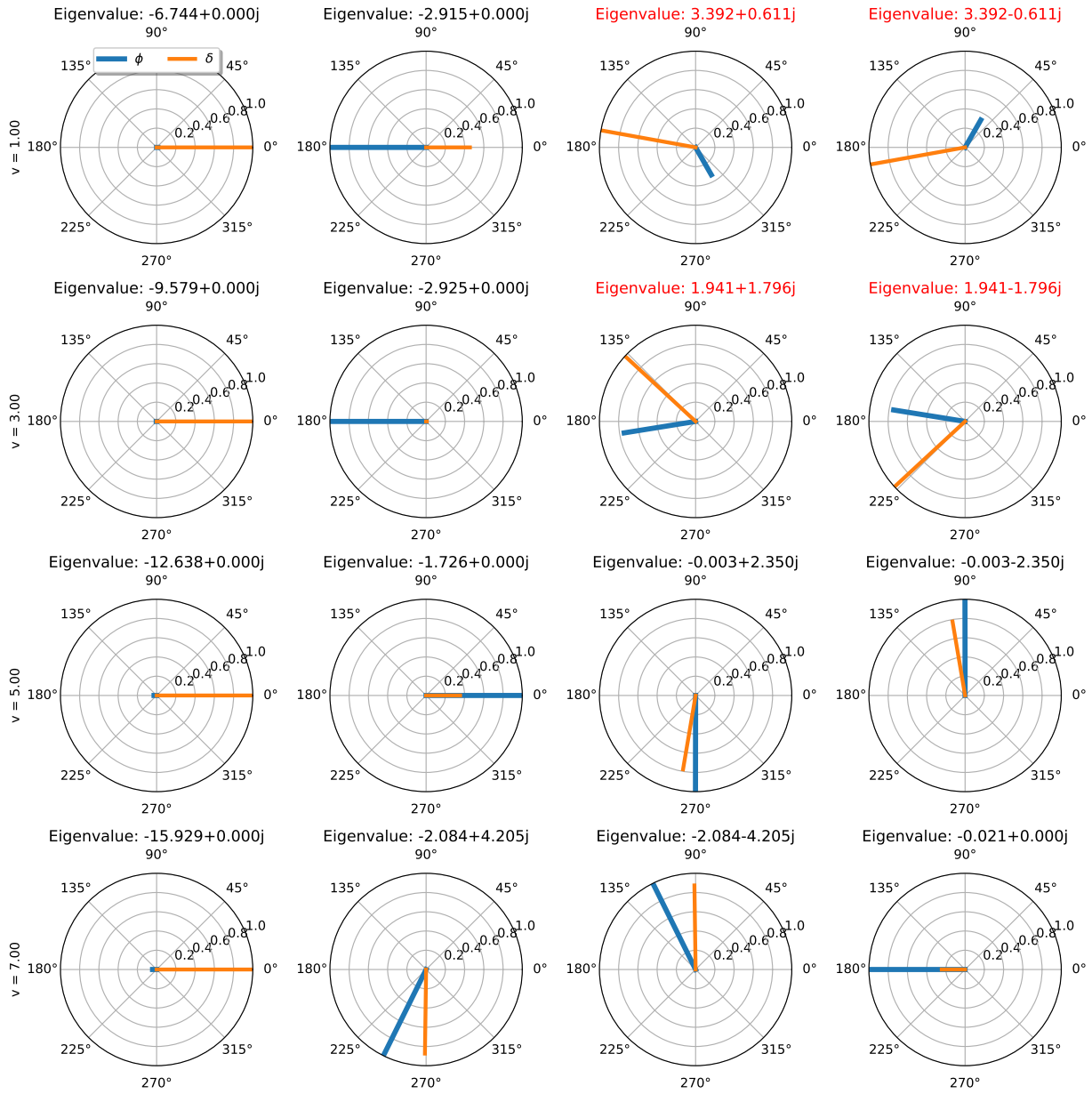
```
wheelbases = np.linspace(0.2, 5.0, num=50)

model.plot_eigenvalue_parts(v=6.0, w=wheelbases)
```


Real and Imaginary Parts of the Eigenvalue [1/s]
20
10
0
−10
−20
−10.0
−7.5
−5.0
−2.5
0.0
2.5
5.0
7.5
10.0
v


Real and Imaginary Parts of the Eigenvalue [1/s]
5
0
−5
−10
−15
−20
1
2
3
4
5
w

The eigenvector components can be created for each mode and for a series of parameter values:

```
model.plot_eigenvectors(v=[1.0, 3.0, 5.0, 7.0])
```



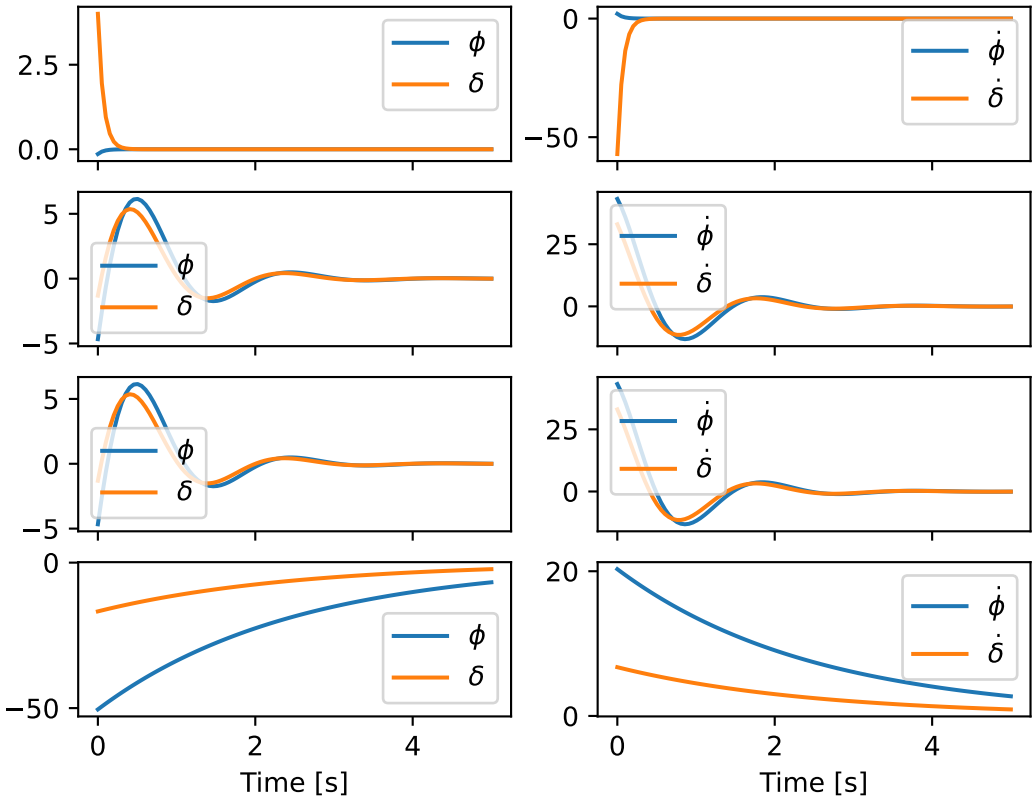
The eigenmodes can be simulated for specific parameter values:

```
times = np.linspace(0.0, 5.0, num=100)

model.plot_mode_simulations(times, v=6.0)
```

A general simulation from initial conditions can also be run:

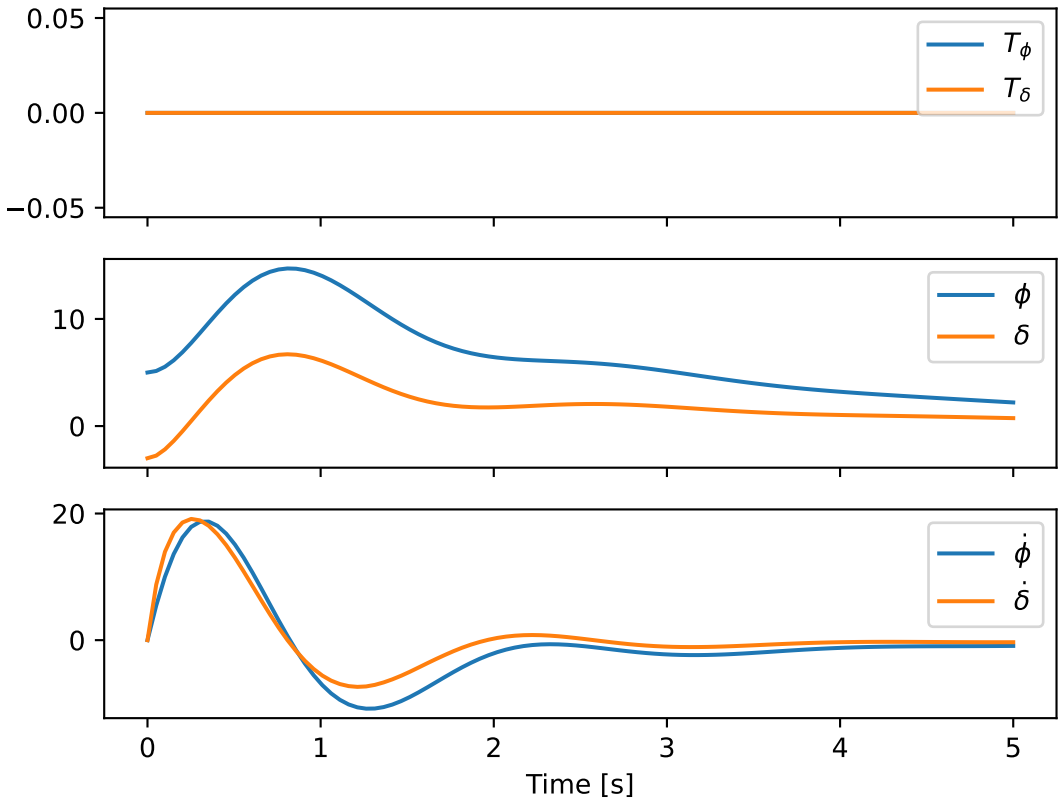
```
x0 = np.deg2rad([5.0, -3.0, 0.0, 0.0])
```

(continues on next page)


2.5
0.0
0
−50
5
0
−5
25
0
5
0
−5
25
0
0
−50
20
0
0
2
4
Time [s]
0
2
4
Time [s]
ϕ
δ
$\dot{\phi}$
$\dot{\delta}$

(continued from previous page)

```
model.plot_simulation(times, x0, v=6.0)
```



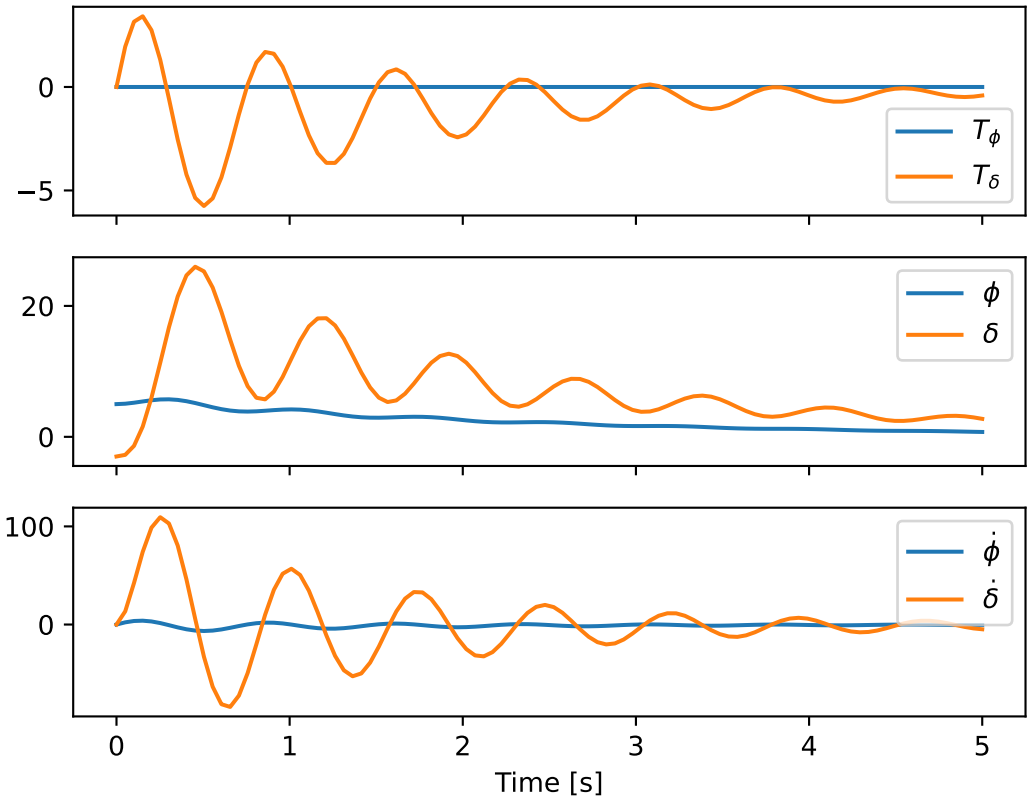
Inputs can be applied in the simulation, for example a simple positive feedback derivative controller on roll:

```
x0 = np.deg2rad([5.0, -3.0, 0.0, 0.0])

model.plot_simulation(times, x0,
    input_func=lambda t, x: np.array([0.0, 50.0*x[2]]),
    v=2.0)
```

1.4 BicycleParameters Data File Information

1.4.1 bicycles/<bicycle name>/Parameters/<bicycle name>Benchmark.txt

`<bicycle name>Benchmark.txt` contains the complete parameter set needed to analyze the Whipple bicycle model linearized about the upright configuration. Each line should have one of the 24 benchmark parameters in the following format:


0
−5
T_ϕ
T_δ
20
0
ϕ
δ
100
0
$\dot{\phi}$
$\dot{\delta}$
0
1
2
3
4
5
Time [s]

```
c = 0.080+/-0.001
```

The first characters are a unique variable name, followed by an equal sign, the value of the parameter, a plus or minus symbol (+/-), and the standard deviation of the value. There can be spaces between the parts. Use `0.0` for the standard deviation if this is unknown or if you are not concerned with the uncertainties. Use the same units as the benchmark bicycle paper. These are the possible variables:

- `g` : acceleration due to gravity [m/s**2]
- `c` : trail [m]
- `w` : wheelbase [m]
- `lam` : steer axis tilt [rad]
- `rR` : rear wheel radius [m]
- `rF` : front wheel radius [m]
- `mB` : frame/rider mass [kg]
- `mF` : front wheel mass [kg]
- `mH` : handlebar/fork assembly mass [kg]
- `mR` : rear wheel mass [kg]
- `xB` : x distance to the frame/rider center of mass [m]
- `zB` : z distance to the frame/rider center of mass [m]
- `xH` : x distance to the frame/rider center of mass [m]
- `zH` : z distance to the frame/rider center of mass [m]
- `IBxx` : x moment of inertia of the frame/rider [kg*m**2]
- `IBzz` : z moment of inertia of the frame/rider [kg*m**2]
- `IBxz` : xz product of inertia of the frame/rider [kg*m**2]
- `IFxx` : x moment of inertia of the front wheel [kg*m**2]
- `IFyy` : y moment of inertia of the front wheel [kg*m**2]
- `IHxx` : x moment of inertia of the handlebar/fork [kg*m**2]
- `IHzz` : z moment of inertia of the handlebar/fork [kg*m**2]
- `IHxz` : xz product of inertia of the handlebar/fork [kg*m**2]
- `IRxx` : x moment of inertia of the rear wheel [kg*m**2]
- `IRyy` : y moment of inertia of the rear wheel [kg*m**2]

Optional Parameters

These parameters are assumed to equal zero if not given.

- `yB` : y distance to the frame/rider center of mass [m]
- `yH` : y distance to the handlebar/fork center of mass [m]
- `IBxy` : xy product of inertia of the frame/rider [kg*m**2]
- `IByy` : y moment of inertia of the frame/rider [kg*m**2]
- `IByz` : yz product of inertia of the frame/rider [kg*m**2]
- `IHxy` : xy product of inertia of the handlebar/fork [kg*m**2]
- `IHyy` : y moment of inertia of the handlebar/fork [kg*m**2]
- `IHyz` : yz product of inertia of the handlebar/fork [kg*m**2]

1.4.2 bicycles/<bicycle name>/RawData/<bicycle name>Measured.txt

This documentation does not contain the complete details of acquiring the raw data. Please refer to [Moore2012] for more information.

`<bicycle name>Measured.txt` contains the raw measurement data for a bicycle. The file should have one variable on each line in the following format:

```
mR = 1.38+/-0.02, 1.37+/-0.02
```

This is the same as the previous parameter variable definition accept that multiple measurements can be included as comma separated values. The values will be averaged together on import. The following gives the measured values:

- `aB1` : perpendicular distance from the pendulum axis to the rear axle center, first orienation [m]
- `aB2` : perpendicular distance from the pendulum axis to the rear axle center, second orienation [m]
- `aB3` : perpendicular distance from the pendulum axis to the rear axle center, third orienation [m]
- `aH1` : perpendicular distance from the pendulum axis to the front axle center, first orienation [m]
- `aH2` : perpendicular distance from the pendulum axis to the front axle center, second orienation [m]
- `aH3` : perpendicular distance from the pendulum axis to the front axle center, third orienation [m]
- `alphaB1` : angle of the head tube with respect to horizontal, first orientation [deg]
- `alphaB2` : angle of the head tube with respect to horizontal, second orientation [deg]
- `alphaB3` : angle of the head tube with respect to horizontal, third orientation [deg]
- `alphaH1` : angle of the steer tube with respect to horizontal, first orientation [deg]
- `alphaH2` : angle of the steer tube with respect to horizontal, second orientation [deg]
- `alphaH3` : angle of the steer tube with respect to horizontal, third orientation [deg]
- `dF` : distance the front wheel travels [m]
- `dP` : diameter of the calibration rod [m]
- `dR` : distance the rear wheel travels [m]
- `f` : fork offset [m]
- `g` : acceleration due to gravity [m/s**2]

- `gamma` : head tube angle [deg]
- `lF` : front wheel compound pendulum length [m]
- `lP` : calibration rod length [m]
- `lR` : rear wheel compound pendulum length [m]
- `mB` : frame mass [kg]
- `mF` : front wheel mass [kg]
- `mH` : fork/handlebar mass [kg]
- `mP` : calibration rod mass [kg]
- `mR` : rear wheel mass [kg]
- `nF` : number of rotations of the front wheel
- `nR` : number of rotations of the rear wheel
- `TcB1` : frame compound pendulum oscillation period [s]
- `TcF1` : front wheel compound pendulum oscillation period [s]
- `TcH1` : fork/handlebar compound pendulum oscillation period [s]
- `TcR1` : rear wheel compound pendulum oscillation period [s]
- `TtB1` : frame torsional pendulum oscillation period, first orientation [s]
- `TtB2` : frame torsional pendulum oscillation period, second orientation [s]
- `TtB3` : frame torsional pendulum oscillation period, third orientation [s]
- `TtF1` : front wheel torsional pendulum oscillation period, first orientation [s]
- `TtH1` : handlebar/fork torsional pendulum oscillation period, first orientation [s]
- `TtH2` : handlebar/fork torsional pendulum oscillation period, second orientation [s]
- `TtH3` : handlebar/fork torsional pendulum oscillation period, third orientation [s]
- `TtP1` : calibration torsional pendulum oscillation period [s]
- `TtR1` : rear wheel torsional pendulum oscillation period [s]
- `w` : wheelbase [m]

Geometry Option

The default option is to provide the wheelbase `w`, fork offset `f`, head tube angle `gamma` and the wheel radii `rR` `rF`, but there is a secondary option for the geometric variables using the perpendicular distances from the steer axis to the wheel centers and the distance between their respective intersection points. To use these, simply replace w, gamma, and f with these dimensions:

- `h1` : distance from the base of the height gage to the top of the rear wheel axis [m]
- `h2` : distance from the table surface to the base of the height gage [m]
- `h3` : distance from the table surface to the top of the head tube [m]
- `h4` : height of the top of the front wheel axle [m]
- `h5` : height of the top of the steer tube [m]
- `d1` : outer diameter of the head tube [m]

- `d2` : diameter of the dummy rear axle [m]
- `d3` : diameter of of the dummy front axle [m]
- `d4` : outer diameter of the steer tube [m]
- `d` : inside distance between the rear and the front axles with the fork reversed [m]

The details of how to take these measurements can be found in our raw data sheet and in [Moore2012].

Rider Configuration Details

A rider can be situated on the bicycle if other raw bicycle measurements are provided.

- `lsp` : the length of the seat post (i.e. the length from the intersection of the top tube with the seat tube to the top of the seat along the axis of the seat tube. [m]
- `lst` : the length of the seat tube (i.e. the distance from the center of the bottom bracket to the intersection of the seat tube and the top tube) [m]
- `hbb` : the height of the bottom bracket off the ground [m]
- `lamst` : the acute angle between horizontal and the seat tube [rad]
- `lcs` : the distance from the center of the bottom bracket to the center of the rear wheel [m]
- `LhbR` : the distance from the center of the rear wheel to either the left or right the handlebar grip (roughly where the center of the hand would fall) [m]
- `LhbF` : the distance from the center of the front wheel to either the left or right the handlebar grip (roughly where the center of the hand would fall) [m]

Other

You may see these values, `xcl` and `zcl`, in the `Rigid` and `Rigidcl` bicycles input files. They locate the lateral force point in the benchmark coordinates. Also, `ds1` and `ds3` locate the accelerometer with respect to a point on the steer axis which is aligned with the handlebar center of mass.

Fork/Handlebar Separation

The measurement of the fork and the handlebar as two rigid bodies is also supported. See the example bicycle called `Rigid` for more details. The fork subscript is `S` and the handlebar subscript is `G`.

Notes

- The periods `T` are not required if you provide oscillation signal data files.
- You have to specify at least three orientations but more can increase the accuracy of the parameter estimations. Currently you can specify up to six orientation for each rigid body.

1.4.3 Pendulum Data Files

If you have raw signal data that the periods can be estimated from, then these should be included in the `RawData` directory. There should be at least one file for every period typically found in `<bicycle name>Measured.txt` file. The signals collected should exhibit very typical decayed oscillations. Currently the only supported file is a Matlab mat file with these variables:

- `data` : signal vector of a decaying oscillation
- `sampleRate` : sample rate of data in hertz

The files should be named in this manner `<short name><part><pendulum><orientation><trial>.mat` where:

- `<bicycle name>` is the short name of the bicycle
- `<part>` is either `Fork`, `Handlebar`, `Frame`, `Rwheel`, or `Fwheel`
- `<orientation>` is either `First`, `Second`, `Third`, `Fourth`, `Fifth`, or `Sixth`
- `<trial>` is an integer greater than or equal to 1

Notes

- `Fork` is the handlebar/fork assembly if they are measured as one rigid body (subscript is `H`). Otherwise `Fork` (`S`) is the fork and `Handlebar` (`G`) is the handlebar when they are measured separately.

1.4.4 riders/<rider name>/Parameters/

<rider name><bicycle name>Benchmark.txt

This file contains the inertial parameters for a rigid rider configured to sit on a particular bicycle expressed with reference to the benchmark reference frame and the rider's center of mass. You can provide these values or let the program generate them.

- `mB` : rider mass [kg]
- `xB` : x distance to the rider center of mass [m]
- `yB` : y distance to the rider center of mass [m]
- `zB` : z distance to the rider center of mass [m]
- `IBxx` : x moment of inertia of the rider [kg*m**2]
- `IByy` : y moment of inertia of the rider [kg*m**2]
- `IBzz` : z moment of inertia of the rider [kg*m**2]
- `IBxy` : xy product of inertia of the rider [kg*m**2]
- `IBxz` : xz product of inertia of the rider [kg*m**2]
- `IByz` : yz product of inertia of the rider [kg*m**2]

`yB`, `IBxy`, and `IByz` are optional due to the assumed symmetry of the rider.

Combined/<rider name><bicycle name>Benchmark.txt

This file contains the geometric and inertial benchmark parameters for a rider seated on a bicycle. The rider is assumed to be rigidly attached to the bicycle frame. These parameters are the same as the ones stored in `bicycles/Parameters/<bicycle name>Benchmark.txt`. These file are only output files.

1.4.5 riders/<rider name>/RawData/

These files must follow the YAML format used in the `yeadon package`_.

<rider name><bicycle name>YeadonCFG.txt

This is an input file to set the configuration of the joint angles for the yeadon package. All values should be set to zero except the `sommersault` value. The `sommersault` value is pi minus the hunch angle of the rider on the bicycle. The hunch angle is the angle between the horizontal and the rider's torso mid line. It is essentially the angle at which the rider is leaned forward.

<rider name><bicycle name>YeadonMeas.txt

This is the yeadon measurement input file for the yeadon package. It contains all of the geometric measurements of the rider. See the yeadon documentation for more details.

1.5 Bicycle Dynamics Analysis App

The Bicycle Dynamics Analysis App provides a GUI for using the BicycleParameters Python program on the web. Using Dash, we transform Python code into HTML/CSS/JS which is nicely rendered within a web browser. The application is available at https://bicycle-dynamics.onrender.com/.

1.5.1 Directory Structure

Within the `BicycleParameters/bicycleparameters/` directory, the primary module for the application is `app.py` and it is written entirely in Python. The `/assets/` folder contains files which are to be displayed by the app, and the `/data/` folder contains raw bicycle measurement data. Some custom CSS is contained within `/assets/styles.css`, but most of the CSS can be written directly in `app.py` using `dash-bootstrap-components`. You can read more about the purpose of the `/assets/` folder on the Dash documentation.

1.5.2 Development

To develop the application, you need to have an environment which contains all the necessary dependencies for running `app.py`. If you are using `conda`, you can use the `cycle-app-environment.yml` located in the top-level `BicycleParameters/conda/` directory to build a conda environment which contains all the packages you need to develop and run this code. Alternatively, you can simply view the contents of `cycle-app-environment.yml` and install those packages accordingly or use the `requirements.txt` file available in the top level directory.

To run `app.py` and view the contents locally, navigate to the `BicycleParameters/` directory and run:

```
python -m bicycleparameters.app
```

This command calls python and passes the file `bicycleparameters.app` (this is like `/bicycleparameters/app.py`) using the `-m` flag as if we were simply running a python script. If you instead navigate to `/bicycleparameters/` and run `python app.py`, the plot images will not appear in your browser. Stick to the first command above.

If `app.py` has been executed properly, you should see an output similar to the following;

```
(bp) user@host:~/BicycleParameters/bicycleparameters$ python -m bicycleparameters.app
Dash is running on http://127.0.0.1:8050/

 Warning: This is a development server. Do not use app.run_server
 in production, use a production WSGI server like gunicorn instead.

 * Serving Flask app "app" (lazy loading)
 * Environment: production
   WARNING: This is a development server. Do not use it in a production deployment.
   Use a production WSGI server instead.
 * Debug mode: on
```

Now if you navigate to http://127.0.0.1:8050/, you should see your local version of the app displayed in your browser. Congratulations! As you play with the application online you should see feedback within your terminal window. Debug information will also display here. In addition, I recommend using the inspect element tool available with most browsers to debug things live within your browser.

1.5.3 Additional Resources

Here are some resources that I found very useful when first developing this application:

- The offical Dash documentation. Just about every single link on this page will have useful information for you.
- Dash Bootstrap Components documentation. This is used to write CSS Bootstraps using the Python language.
- The Mozilla Web Development guide. I highly recommened this guide for learning about HTML, CSS, and general web development.
- The example usage page for Bicycle Parameters. Useful for understanding how the backend code works.
- w3schools.com. Has great HTML/CSS reference pages as well as tutorials. Also has some for Python.
- This Software Carpentery site. Has nice general programming tutorials as well as an in-depth git tutorial.

Feel free to extend this list as you develop and learn. Overall, I ended up needing to learn and use web development skills far more than I needed to really understand Python itself. Program in whichever way brings you the most joy. I wish you the best, future devs!

1.6 bicycleparameters Package

1.6.1 `main` Module

class Bicycle(*bicycleName*, *pathToData='.'*, *forceRawCalc=False*, *forcePeriodCalc=False*)

Bases: `object`

An object for a bicycle. A bicycle has parameters and can have a rider attached to it. That's about it for now.

add_rider(*riderName*, *reCalc=False*, *draw=False*)

Adds the inertial effects of a rigid rider to the bicycle.

Parameters

riderName
[string] A rider name that corresponds to a folder in *<pathToData>/riders/*.

reCalc
[boolean, optional] If true, the rider parameters will be recalculated.

draw
[boolean, optional] If true, visual python will be used to draw a three dimensional image of the rider.

`calculate_from_measured`(*forcePeriodCalc=False*)

Calculates the parameters from measured data.

`canonical`(*nominal=False*)

Returns the canonical velocity and gravity independent matrices for the Whipple bicycle model linearized about the nominal configuration.

Parameters

nominal
[boolean, optional] The default is false and uarrays are returned with the calculated uncertainties. If true ndarrays are returned without uncertainties.

Returns

M
[uarray, shape(2,2)] Mass matrix.

C1
[uarray, shape(2,2)] Velocity independent damping matrix.

K0
[uarray, shape(2,2)] Gravity independent part of the stiffness matrix.

K2
[uarray, shape(2,2)] Velocity squared independent part of the stiffness matrix.

Notes

The canonical matrices complete the following equation:

M * q'' + v * C1 * q' + [g * K0 + v**2 * K2] * q = f

where:

q = [phi, delta] f = [Tphi, Tdelta]

phi
Bicycle roll angle.

delta
Steer angle.

Tphi
Roll torque.

Tdelta
Steer torque.

v
Bicylce speed.

If you have a flywheel defined, body D, it will completely be ignored in these results. These results are strictly for the Whipple bicycle model.

`compare_bode_speeds`(*speeds*, *u*, *y*, *fig=None*)

Returns a figure with the Bode plots of multiple bicycles.

Parameters

speeds
[list] A list of speeds at which to evaluate the system.

u
[integer] An integer between 0 and 1 corresponding to the inputs roll torque and steer torque.

y
[integer] An integer between 0 and 3 corresponding to the inputs roll angle, steer angle, roll rate, steer rate.

Returns

fig
[matplotlib.Figure instance] The Bode plot.

Notes

The phases are matched around zero degrees at with respect to the first frequency.

`eig`(*speeds*)

Returns the eigenvalues and eigenvectors of the Whipple bicycle model linearized about the nominal configuration.

Parameters

speeds
[ndarray, shape (n,) or float] The speed at which to calculate the eigenvalues.

Returns

evals
[ndarray, shape (n, 4)] eigenvalues

evecs
[ndarray, shape (n, 4, 4)] eigenvectors

Notes

If you have a flywheel defined, body D, it will completely be ignored in these results. These results are strictly for the Whipple bicycle model.

`plot_bicycle_geometry`(*show=True*, *pendulum=True*, *centerOfMass=True*, *inertiaEllipse=True*)

Returns a figure showing the basic bicycle geometry, the centers of mass and the moments of inertia.

Parameters

show
[boolean, optional] If true `matplotlib.pyplot.show()` will be called before exiting the function.

pendulum
[boolean, optional] If true the axes of the torsional pendulum will be displayed (only useful if raw measurement data is availabe).

centerOfMass
[boolean, optional] If true the mass center of each rigid body will be displayed.

inertiaEllipse
[boolean optional] If true inertia ellipses for each rigid body will be displayed.

Returns

fig
[matplotlib.pyplot.Figure]

Notes

If the flywheel is defined, it's center of mass corresponds to the front wheel and is not depicted in the plot.

`plot_bode`(*speed*, *u*, *y*, ***kwargs*)

Returns a Bode plot.

Parameters

speed
[float] The speed at which to evaluate the system.

u
[integer] An integer between 0 and 1 corresponding to the inputs roll torque and steer torque.

y
[integer] An integer between 0 and 3 corresponding to the inputs roll angle steer angle, roll rate, steer rate.

kwargs
[keyword pairs] Any options that can be passed to dtk.bode.

Returns

mag
[ndarray, shape(1000,)] The magnitude in dB of the frequency reponse.

phase
[ndarray, shape(1000,)] The phase in degress of the frequency response.

fig
[matplotlib figure] The Bode plot.

`plot_eigenvalues_vs_speed`(*speeds*, *fig=None*, *generic=False*, *color='black'*, *show=False*, *largest=False*, *linestyle='-'*, *grid=False*, *show_legend=True*)

Returns a plot of the eigenvalues versus speed for the current benchmark parameters.

Parameters

speeds
[ndarray, shape(n,)] An array of speeds to calculate the eigenvalues at.

fig
[matplotlib figure, optional] A figure to plot to.

generic
[boolean] If true the lines will all be the same color and the modes will not be labeled.

color
[matplotlib color] If generic is true this will be the color of the plot lines.

largest
[boolean] If true, only the largest eigenvalue is plotted.

grid
[boolean, optional] If true, displays a grid on the plot.

show_legend: boolean, optional
If true, displays a legend describing the different parts of the solution shown.

Returns

fig
[matpolib.pyplot.Figure] The figure.

Notes

If you have a flywheel defined, body D, it will completely be ignored in these results. These results are strictly for the Whipple bicycle model.

`save_parameters`(*filetype='text'*)

Saves all the parameter sets to file.

Parameters

filetype
[string, optional]

- 'text' : a text file with parameters as `c = 0.10+/-0.01`
- 'matlab' : matlab .mat file
- 'pickle' : python pickled dictionary

`show_pendulum_photos`()

Opens up the pendulum photos in eye of gnome for inspection.

This only works in Linux and if eog is installed. Maybe check pythons xdg-mime model for having this work cross platform.

`state_space`(*speed*, *nominal=False*)

Returns the A and B matrices for the Whipple model linearized about the upright constant velocity configuration.

Parameters

speed
[float] The speed of the bicycle.

nominal
[boolean, optional] The default is false and uarrays are returned with the calculated uncertainties. If true ndarrays are returned without uncertainties.

Returns

A
[ndarray, shape(4,4)] The state matrix.

B
[ndarray, shape(4,2)] The input matrix.

Notes

`A` and `B` describe the Whipple model in state space form:

x' = A * x + B * u

where

The states are [roll angle,
steer angle, roll rate, steer rate]

The inputs are [roll torque,
steer torque]

If you have a flywheel defined, body D, it will completely be ignored in these results. These results are strictly for the Whipple bicycle model.

`steer_assembly_moment_of_inertia`(*handlebar=True*, *fork=True*, *wheel=True*, *aboutSteerAxis=False*, *nominal=False*)

Returns the inertia tensor of the steer assembly with respect to a reference frame aligned with the steer axis.

Parameters

handlebar
[boolean, optional] If true the handlebar will be included in the calculation.

fork
[boolean, optional] If true the fork will be included in the calculation.

wheel
[boolean, optional] If true then the wheel will be included in the calculation.

aboutSteerAxis
[boolean, optional] If true the inertia tensor will be with respect to a point made from the projection of the center of mass onto the steer axis.

nominal
[boolean, optional] If true the nominal values will be returned instead of a uarray.

Returns

iAss
[float] Inertia tensor of the specified steer assembly parts with respect to a reference frame aligned with the steer axis.

Notes

The 3 component is aligned with the steer axis (pointing downward), the 1 component is perpendicular to the steer axis (pointing forward) and the 2 component is perpendicular to the steer axis (pointing to the right).

This function does not currently take into account the flywheel, D, if it is defined, beware.

`calculate_benchmark_from_measured`(*mp*)

Returns the benchmark (Meijaard 2007) parameter set based on the measured data.

Parameters

mp
[dictionary] Complete set of measured data.

Returns

par
[dictionary] Benchmark bicycle parameter set.

`get_parts_in_parameters`(*par*)

Returns a list of parts in a parameter dictionary.

Parameters

par
[dictionary] Benchmark bicycle parameters.

Returns

parts
[list] Unique list of parts that contain one or more of 'H', 'B', 'F', 'R', 'S', 'G', 'D'.

`is_fork_split`(*mp*)

Returns true if the fork was split into two parts and false if not.

Parameters

mp
[dictionary] The measured data.

Returns

forkIsSplit
[boolean]

1.6.2 `com` Module

`cartesian`(*arrays*, *out=None*)

Generate a cartesian product of input arrays.

Parameters

arrays
[list of array-like] 1-D arrays to form the cartesian product of.

out
[ndarray] Array to place the cartesian product in.

Returns

out
[ndarray] 2-D array of shape (M, len(arrays)) containing cartesian products formed of input arrays.

Examples

```
>>> cartesian(([1, 2, 3], [4, 5], [6, 7]))
array([[1, 4, 6],
       [1, 4, 7],
       [1, 5, 6],
       [1, 5, 7],
       [2, 4, 6],
       [2, 4, 7],
       [2, 5, 6],
       [2, 5, 7],
       [3, 4, 6],
       [3, 4, 7],
       [3, 5, 6],
       [3, 5, 7]])
```

`center_of_mass`(*slopes*, *intercepts*)

Returns the center of mass relative to the slopes and intercepts coordinate system.

Parameters

slopes
[ndarray, shape(n,)] The slope of every line used to calculate the center of mass.

intercepts
[ndarray, shape(n,)] The intercept of every line used to calculate the center of mass.

Returns

x
[float] The abscissa of the center of mass.

y
[float] The ordinate of the center of mass.

`com_line`(*alpha*, *a*, *par*, *part*, *l1*, *l2*)

Returns the slope and intercept for the line that passes through the part's center of mass with reference to the benchmark bicycle coordinate system.

Parameters

alpha
[float] The angle the head tube makes with the horizontal. When looking at the bicycle from the right side this is the angle between a vector point out upwards along the steer axis and the earth horizontal with the positve direction pointing from the left to the right. If the bike is in its normal configuration this would be 90 degrees plus the steer axis tilt (lambda).

a
[float] The distance from the pendulum axis to a reference point on the part, typically the wheel centers. This is positive if the point falls to the left of the axis and negative otherwise.

par
[dictionary] Benchmark parameters. Must include lam, rR, rF, w

part
[string] The subscript denoting which part this refers to.

l1, l2
[floats] The location of the handlebar reference point relative to the front wheel center when the fork is split. This is measured perpendicular to and along the steer axis, respectively.

Returns

m
[float] The slope of the line in the benchmark coordinate system.

b
[float] The z intercept in the benchmark coordinate system.

`part_com_lines`(*mp*, *par*, *forkIsSplit*)

Returns the slopes and intercepts for all of the center of mass lines for each part.

Parameters

mp
[dictionary] Dictionary with the measured parameters.

Returns

slopes
[dictionary] Contains a list of slopes for each part.

intercepts
[dictionary] Contains a list of intercepts for each part.

The slopes and intercepts lists are in order with respect to each other and the keyword is either 'B', 'H', 'G' or 'S' for Frame, Handlebar/Fork, Handlerbar, and Fork respectively.

`total_com`(*coordinates*, *masses*)

Returns the center of mass of a group of objects if the indivdual centers of mass and mass is provided.

coordinates
[array_like, shape(3,n)] The rows are the x, y and z coordinates, respectively and the columns are for each object.

masses
[array_like, shape(3,)] An array of the masses of multiple objects, the order should correspond to the columns of coordinates.

Returns

mT
[float] Total mass of the objects.

cT
[ndarray, shape(3,)] The x, y, and z coordinates of the total center of mass.

1.6.3 bicycle Module

ab_matrix(*M*, *C1*, *K0*, *K2*, *v*, *g*)

Calculate the A and B matrices for the Whipple bicycle model linearized about the upright configuration.

Parameters

M
[ndarray, shape(2,2)] The mass matrix.

C1
[ndarray, shape(2,2)] The damping like matrix that is proportional to the speed, v.

K0
[ndarray, shape(2,2)] The stiffness matrix proportional to gravity, g.

K2
[ndarray, shape(2,2)] The stiffness matrix proportional to the speed squared, v**2.

v
[float] Forward speed.

g
[float] Acceleration due to gravity.

Returns

A
[ndarray, shape(4,4)] State matrix.

B
[ndarray, shape(4,2)] Input matrix.

The states are [roll angle,
steer angle, roll rate, steer rate]

The inputs are [roll torque,
steer torque]

benchmark_par_to_canonical(*p*)

Returns the canonical matrices of the Whipple bicycle model linearized about the upright constant velocity configuration. It uses the parameter definitions from Meijaard et al. 2007.

Parameters

p
[dictionary] A dictionary of the benchmark bicycle parameters. Make sure your units are correct, best to ue the benchmark paper's units!

Returns

M
[ndarray, shape(2,2)] The mass matrix.

C1
[ndarray, shape(2,2)] The damping like matrix that is proportional to the speed, v.

K0
[ndarray, shape(2,2)] The stiffness matrix proportional to gravity, g.

K2
[ndarray, shape(2,2)] The stiffness matrix proportional to the speed squared, v**2.

Notes

This function handles parameters with uncertanties.

lambda_from_abc(*rF*, *rR*, *a*, *b*, *c*)

Returns the steer axis tilt, lamba, for the parameter set based on the offsets from the steer axis.

Parameters

rF
[float or ufloat] Front wheel radius.

rR
[float or ufloat] Rear wheel radius.

a
[float or ufloat] The rear wheel offset. The minimum distance from the steer axis to the center of the rear wheel.

b
[float or ufloat] The front wheel offset. The minimum distance from the steer axis to the center of the front wheel.

c
[float or ufloat] The steer axis distance. The distance along the steer axis between the intersection of the front and rear wheel offset lines.

sort_eigenmodes(*evals*, *evecs*)

Sort eigenvalues and eigenvectors.

Parameters

evals
[ndarray, shape (n, 4)] A sequence of n sets of eigenvalues.

evecs
[ndarray, shape (n, 4, 4)] A sequence of n sets of eigenvectors.

Returns

evalsorg
[ndarray, shape (n, 4)] A sequence of n sets of eigenvalues.

evecsorg
[ndarray, shape (n, 4, 4)] A sequence of n sets of eigenvectors.

sort_modes(*evals*, *evecs*)

Sort eigenvalues and eigenvectors into weave, capsize, caster modes.

Parameters

evals
[ndarray, shape (n, 4)] eigenvalues

evecs
[ndarray, shape (n, 4, 4)] eigenvectors

Returns

weave['evals']
[ndarray, shape (n, 2)] The eigen value pair associated with the weave mode.

weave['evecs']
[ndarray, shape (n, 4, 2)] The associated eigenvectors of the weave mode.

capsize['evals']
[ndarray, shape (n,)] The real eigenvalue associated with the capsize mode.

capsize['evecs']
[ndarray, shape(n, 4, 1)] The associated eigenvectors of the capsize mode.

caster['evals']
[ndarray, shape (n,)] The real eigenvalue associated with the caster mode.

caster['evecs']
[ndarray, shape(n, 4, 1)] The associated eigenvectors of the caster mode.

Notes

This only works on the standard bicycle eigenvalues, not necessarily on any general eigenvalues for the bike model (e.g. there isn't always a distinct weave, capsize and caster). Some type of check using the derivative of the curves could make it more robust.

`trail`(*rF*, *lam*, *fo*)

Calculate the trail and mechanical trail.

Parameters

rF
[float] The front wheel radius

lam
[float] The steer axis tilt (pi/2 - headtube angle). The angle between the headtube and a vertical line.

fo
[float] The fork offset

Returns

c: float
Trail

cm: float
Mechanical Trail

1.6.4 `geometry` Module

`calc_two_link_angles`(*L1*, *L2*, *D*)

Solves a simple case of the two-link revolute joint inverse kinematics problem. Both output angles are positive. The simple case is that the end of the second link lies on the x-axis.

Parameters

L1
[float] Length of the first link.

L2
[float] Length of the second link.

D
[float] Distance from the base of first link to the end of the second link.

Returns

theta1
[float] (radians) Angle between x-axis and first link; always positive.

theta2
[float] (radians) Angle between first link and second link; always positive.

`calculate_abc_geometry`(*h*, *d*)

Returns the perpendicular distance geometry for the bicycle from the raw measurements.

Parameters

h
[tuple] Tuple containing the measured parameters h1-h5. (h1, h2, h3, h4, h5)

d
[tuple] Tuple containing the measured parameters d1-d4 and d. (d1, d2, d3, d4, d)

Returns

a
[ufloat or float] The rear frame offset.

b
[ufloat or float] The fork offset.

c
[ufloat or float] The steer axis distance.

`calculate_benchmark_geometry`(*mp*, *par*)

Returns the wheelbase, steer axis tilt and the trail.

Parameters

mp
[dictionary] Dictionary with the measured parameters.

par
[dictionary] Dictionary with the benchmark parameters.

Returns

par
[dictionary] par with the benchmark geometry added.

`calculate_l1_l2`(*h6*, *h7*, *d5*, *d6*, *l*)

Returns the distance along (l2) and perpendicular (l1) to the steer axis from the front wheel center to the handlebar reference point.

Parameters

h6
[float] Distance from the table to the top of the front axle.

h7
[float] Distance from the table to the top of the handlebar reference circle.

d5
[float] Diameter of the front axle.

d6
[float] Diameter of the handlebar reference circle.

l
[float] Outer distance from the front axle to the handlebar reference circle.

Returns

l1
[float] The distance from the front wheel center to the handlebar reference center perpendicular to the steer axis. The positive sense is if the handlebar reference point is more forward than the front wheel center relative to the steer axis normal.

l2
[float] The distance from the front wheel center to the handlebar reference center parallel to the steer axis. The positive sense is if the handlebar reference point is above the front wheel center with reference to the steer axis.

`distance_to_steer_axis`(*w*, *c*, *lam*, *point*)

Returns the minimal distance from the steer axis to the given point when the bicycle is in the nominal configuration.

Parameters

w
[float or ufloat] Wheelbase.

c
[float or ufloat] Trail.

lam
[float or ufloat] Steer axis tilt in radians.

point
[narray, shape(3,)] A point that lies in the symmetry plane of the bicycle.

Returns

d
[float or ufloat] The minimal distance from the given point to the steer axis.

`fundamental_geometry_plot_data`(*par*)

Returns the coordinates for line end points of the bicycle fundamental geometry.

Parameters

par
[dictionary] Benchmark bicycle parameters.

Returns

x
[ndarray]

z
[ndarray]

`fwheel_to_handlebar_ref`(*lam*, *l1*, *l2*)

Returns the distance along the benchmark coordinates from the front wheel center to the handlebar reference center.

Parameters

lam
[float] Steer axis tilt.

l1, l2
[float] The distance from the front wheel center to the handlebar refernce center perpendicular to and along the steer axis.

Returns

u1, u2
[float]

`point_to_line_distance`(*point*, *pointsOnLine*)

Returns the minimal distance from a point to a line in three dimensional space.

Parameters

point
[ndarray, shape(3,)] The x, y, and z coordinates of a point.

pointsOnLine
[ndarray, shape(3,2)] The x, y, and z coordinates of two points on a line. Rows are coordinates and columns are points.

Returns

distance
[float] The minimal distance from the line to the point.

`project_point_on_line`(*line*, *point*)

Returns point of projection.

Parameters

line
[tuple] Slope and intercept of the line.

point
[tuple] Location of the point.

Returns

newPoint
[tuple] The location of the projected point.

`vec_angle`(*v1*, *v2*)

Returns the interior angle between two vectors using the dot product. Inputs do not need to be unit vectors.

Parameters

v1
[np.array (3,1)] input vector.

v2
[np.array (3,1)] input vector.

Returns

angle
[float] (radians) interior angle between v1 and v2.

`vec_project`(*vec*, *direction*)

Vector projection into a plane, where the plane is defined by a normal vector.

Parameters

vec
[np.array(3,1)] vector to be projected into a plane

direction
[int or np.array, shape(3,)] If int, it is one of the three orthogonal directions, (0,1 or 2) of

the input vector (essentially, that component of vec is set to zero). If np.array, can be in any direction (not necessarily a coordinate direction).

Returns

vec_out
[np.array(3,1)] Projected vector.

1.6.5 inertia Module

`combine_bike_rider`(*bicyclePar*, *riderPar*)

Combines the inertia of the bicycle frame with the inertia of a rider.

Parameters

bicyclePar
[dictionary] The benchmark parameter set of a bicycle.

riderPar
[dictionary] The rider's mass, center of mass, and inertia expressed in the benchmark bicycle reference frame.

Returns

bicyclePar
[dictionary] The benchmark bicycle parameters with a rigid rider added to the bicycle frame.

`compound_pendulum_inertia`(*m*, *g*, *l*, *T*)

Returns the moment of inertia for an object hung as a compound pendulum.

Parameters

m
[float] Mass of the pendulum.

g
[float] Acceration due to gravity.

l
[float] Length of the pendulum.

T
[float] The period of oscillation.

Returns

I
[float] Moment of interia of the pendulum.

`inertia_components`(*jay*, *beta*)

Returns the 2D orthogonal inertia tensor.

When at least three moments of inertia and their axes orientations are known relative to a common inertial frame of a planar object, the orthoganl moments of inertia relative the frame are computed.

Parameters

jay
[ndarray, shape(n,)] An array of at least three moments of inertia. (n >= 3)

beta
[ndarray, shape(n,)] An array of orientation angles corresponding to the moments of inertia in jay.

Returns

eye
[ndarray, shape(3,)] Ixx, Ixz, Izz

`parallel_axis`(*Ic*, *m*, *d*)

Returns the moment of inertia of a body about a different point.

Parameters

Ic
[ndarray, shape(3,3)] The moment of inertia about the center of mass of the body with respect to an orthogonal coordinate system.

m
[float] The mass of the body.

d
[ndarray, shape(3,)] The distances along the three ordinates that located the new point relative to the center of mass of the body.

Returns

I
[ndarray, shape(3,3)] The moment of inertia about of the body about a point located by d.

`part_inertia_tensor`(*par*, *part*)

Returns an inertia tensor for a particular part for the benchmark parameter set.

Parameters

par
[dictionary] Complete Benchmark parameter set.

part
[string] Either 'B', 'H', 'F', 'R', 'G', 'S', 'D'

Returns

I
[ndarray, shape(3,3)] Inertia tensor for the part.

Notes

Parts G, S, and D are additional parts not included in the published paper on the benchmark bicycle, they are only relavant if used. See the documentation.

`principal_axes`(*I*)

Returns the principal moments of inertia and the orientation.

Parameters

I
[ndarray, shape(3,3)] An inertia tensor.

Returns

Ip
[ndarray, shape(3,)] The principal moments of inertia. This is sorted smallest to largest.

C
[ndarray, shape(3,3)] The rotation matrix.

`rotate_inertia_tensor`(*I*, *angle*)

Returns inertia tensor rotated through angle. Only for 2D

`tor_inertia`(*k*, *T*)

Calculate the moment of inertia for an ideal torsional pendulm

Parameters

k: torsional stiffness
T: period

Returns

I: moment of inertia

`torsional_pendulum_stiffness`(*I*, *T*)

Calculate the stiffness of a torsional pendulum with a known moment of inertia.

Parameters

I
[moment of inertia]

T
[period]

Returns

k
[stiffness]

`tube_inertia`(*l*, *m*, *ro*, *ri*)

Calculate the moment of inertia for a tube (or rod) where the x axis is aligned with the tube's axis.

Parameters

l
[float] The length of the tube.

m
[float] The mass of the tube.

ro
[float] The outer radius of the tube.

ri
[float] The inner radius of the tube. Set this to zero if it is a rod instead of a tube.

Returns

Ix
[float] Moment of inertia about tube axis.

Iy, Iz
[float] Moment of inertia about normal axis.

1.6.6 io Module

filename_to_dict(*filename*)

Returns a dictionay of values based on the pendulum data file name.

load_parameter_text_file(*pathToFile*)

Returns a dictionary of float and/or ufloat parameters from a parameter file.

Parameters

pathToFile
[string] The path to the text file with the parameters listed in the specified format.

Returns

parameters
[dictionary] A dictionary of the values stored in the text files.

For example::
c = 0.08 +/- 0.01 d=0.314+/-0.002 t = 0.1+/-0.01, 0.12+/-0.02 whb = 0.5

The first item on the line must be the variable name and the second is an equals sign. The values to the right of the equal sign much may or may not contain an uncertainty designated by +/-. Multiple comma seperated values will be averaged.

load_pendulum_mat_file(*pathToFile*)

Returns a dictionay containing the data from the pendulum data mat file.

remove_uncertainties(*dictionary*)

Returns a dictionary with the uncertainties removed.

space_out_camel_case(*s*, *output='string'*)

Adds spaces to a camel case string. Failure to space out string returns the original string.

Examples

```
>>> space_out_camel_case('DMLSServicesOtherBSTextLLC')
'DMLS Services Other BS Text LLC'
>>> space_out_camel_case('DMLSServicesOtherBSTextLLC', output='list')
['DMLS', 'Services', 'Other', 'BS', 'Text', 'LLC']
```

write_parameter_text_file(*pathToTxtFile*, *parDict*)

Writes parameter set to file.

Parameters

pathToTxtFile
[string] The path to the file to write the parameters.

pardict
[dictionary] A dictionary of parameters for the bicycle.

Returns

saved
[boolean] True if the file was saved and false if not.

`write_periods_to_file`(*pathToRawFile*, *mp*)

Writes the provided periods to file.

Parameters

pathToRawFile
[string] The path to the <bicycle name>Measured.txt file

mp
[dictionary] The measured parameters dictionary. Should contain complete period data.

1.6.7 `models` Module

`class Meijaard2007Model`(*parameter_set*)

Bases: `_Model`

Whipple-Carvallo model presented in [Meijaard2007]. It is both linear and the minimal model in terms of states and coordinates that fully describe the vehicle's dynamics: self-stability and non-minimum phase behavior.

Parameters

parameter_set
[ParameterSet] The `paramter_set.to_parameterization('meijaard2007')` must return a dictionary that maps floats to the parameter keys containing:

- `IBxx` : x moment of inertia of the frame/rider [kg*m**2]
- `IBxz` : xz product of inertia of the frame/rider [kg*m**2]
- `IBzz` : z moment of inertia of the frame/rider [kg*m**2]
- `IFxx` : x moment of inertia of the front wheel [kg*m**2]
- `IFyy` : y moment of inertia of the front wheel [kg*m**2]
- `IHxx` : x moment of inertia of the handlebar/fork [kg*m**2]
- `IHxz` : xz product of inertia of the handlebar/fork [kg*m**2]
- `IHzz` : z moment of inertia of the handlebar/fork [kg*m**2]
- `IRxx` : x moment of inertia of the rear wheel [kg*m**2]
- `IRyy` : y moment of inertia of the rear wheel [kg*m**2]
- `c` : trail [m]
- `g` : acceleration due to gravity [m/s**2]
- `lam` : steer axis tilt [rad]
- `mB` : frame/rider mass [kg]
- `mF` : front wheel mass [kg]
- `mH` : handlebar/fork assembly mass [kg]
- `mR` : rear wheel mass [kg]
- `rF` : front wheel radius [m]
- `rR` : rear wheel radius [m]
- `v` : speed [m/s]
- `w` : wheelbase [m]

- `xB` : x distance to the frame/rider center of mass [m]
- `xH` : x distance to the frame/rider center of mass [m]
- `zB` : z distance to the frame/rider center of mass [m]
- `zH` : z distance to the frame/rider center of mass [m]

References

[Meijaard2007]

Attributes

input_vars
[list of strings] Ordered list of ASCII strings that name the model's input variables.

state_vars
[list of strings] Ordered list of ASCII strings that name the model's state variables.

input_vars_latex
[list of raw strings] Ordered list of LaTeX strings that name the model's input variables.

state_vars_latex
[list of raw strings] Ordered list of LaTeX strings that name the model's state variables.

`calc_eigen`(*left=False*, ***parameter_overrides*)

Returns the right (or left) eigenvalues and eigenvectors of the linear model.

Parameters

left
[boolean, optional] If true, the left eigenvectors will be returned, i.e. `A.T*v=lam*v`.

****parameter_overrides**
[dictionary] Parameter keys that map to floats or array_like of floats shape(n,). All keys that map to array_like must be of the same length.

Returns

evals
[ndarray, shape(4,) or shape (n,4)] Eigenvalues.

evecs
[ndarray, shape(4,4) or shape (n,4,4)] Eigenvectors, each columns are eigenvectors and are associated with same index of the eigenvalues.

Examples

```
>>> from bicycleparameters.parameter_dicts import meijaard2007_browser_jason
>>> from bicycleparameters.parameter_sets import Meijaard2007ParameterSet
>>> from bicycleparameters.models import Meijaard2007Model
>>> p = Meijaard2007ParameterSet(meijaard2007_browser_jason, True)
>>> m = Meijaard2007Model(p)
>>> evals, evecs = m.calc_eigen()
>>> evals
array([-6.74423162+0.j        , -2.9146438 +0.j        ,
        3.39244999+0.61085077j,  3.39244999-0.61085077j])
>>> evecs
```

(continues on next page)

(continued from previous page)

```
array([[ 0.00197344+0.j        , -0.2953538 +0.j        ,
         0.04320146-0.0753826j ,  0.04320146+0.0753826j ],
       [ 0.14665803+0.j        ,  0.13447333+0.j        ,
        -0.26053575+0.04691255j, -0.26053575-0.04691255j],
       [-0.01330934+0.j        ,  0.86085111+0.j        ,
         0.1926063 -0.22934205j,  0.1926063 +0.22934205j],
       [-0.98909574+0.j        , -0.39194186+0.j        ,
        -0.91251108+0.j        , -0.91251108-0.j        ]])
```

form_reduced_canonical_matrices(***parameter_overrides*)

Returns the canonical speed and gravity independent matrices for the Whipple-Carvallo bicycle model linearized about the nominal upright configuration.

Parameters

****parameter_overrides**
[dictionary] Parameter keys that map to floats or array_like of floats shape(n,). All keys that map to array_like must be of the same length.

Returns

M
[ndarray, shape(2,2) or shape(n,2,2)] Mass matrix.

C1
[ndarray, shape(2,2) or shape(n,2,2)] Velocity independent damping matrix.

K0
[ndarray, shape(2,2) or shape(n,2,2)] Gravity independent part of the stiffness matrix.

K2
[ndarray, shape(2,2) or shape(n,2,2)] Velocity squared independent part of the stiffness matrix.

Notes

The canonical matrices complete the following equation:

```
M*q'' + v*C1*q' + [g*K0 + v**2*K2]*q = f
```

where:

- `q = [phi, delta]`
- `f = [Tphi, Tdelta]`

phi
Bicycle roll angle.

delta
Steer angle.

Tphi
Roll torque.

Tdelta
Steer torque.

v
Bicylce longitudinal speed.

g
Acceleration due to gravity.

Examples

```
>>> from bicycleparameters.parameter_dicts import meijaard2007_browser_jason
>>> from bicycleparameters.parameter_sets import Meijaard2007ParameterSet
>>> from bicycleparameters.models import Meijaard2007Model
>>> p = Meijaard2007ParameterSet(meijaard2007_browser_jason, True)
>>> m = Meijaard2007Model(p)
>>> M, C1, K0, K2 = m.form_reduced_canonical_matrices()
>>> M
array([[102.78013216,   1.53582801],
       [  1.53582801,   0.24890226]])
>>> C1
array([[ 0.       , 26.3947333],
       [-0.4503006,  1.037066 ]])
>>> K0
array([[-89.32195981,  -1.74159477],
       [ -1.74159477,  -0.67769624]])
>>> K2
array([[ 0.        , 74.12543   ],
       [ 0.        ,  1.57021553]])
>>> M, _, _, _ = m.form_reduced_canonical_matrices(mB=150.0)
>>> M
array([[176.52178763,   2.69074048],
       [  2.69074048,   0.26699004]])
```

form_state_space_matrices(***parameter_overrides*)

Returns the A and B matrices for the Whipple-Carvallo model linearized about the upright constant velocity configuration.

Parameters

****parameter_overrides**
[dictionary] Parameter keys that map to floats or array_like of floats shape(n,). All keys that map to array_like must be of the same length.

Returns

A
[ndarray, shape(4,4) or shape(n,4,4)] The state matrix.

B
[ndarray, shape(4,2) or shape(n,4,2)] The input matrix.

Notes

A and B describe the Whipple model in state space form:

x' = A * x + B * u

where the states are:

```
x = |roll angle | = |phi     |
    |steer angle|   |delta   |
    |roll rate  |   |phidot  |
    |steer rate |   |deltadot|
```

and the inputs are:

```
u = |roll torque | = |Tphi  |
    |steer torque|   |Tdelta|
```

Examples

```
>>> from bicycleparameters.parameter_dicts import meijaard2007_browser_jason
>>> from bicycleparameters.parameter_sets import Meijaard2007ParameterSet
>>> from bicycleparameters.models import Meijaard2007Model
>>> p = Meijaard2007ParameterSet(meijaard2007_browser_jason, True)
>>> m = Meijaard2007Model(p)
>>> A, B = m.form_state_space_matrices()
>>> A
array([[ 0.        ,  0.        ,  1.        ,  0.        ],
       [ 0.        ,  0.        ,  0.        ,  1.        ],
       [ 8.26150335, -0.9471634 , -0.02977958, -0.21430735],
       [17.66475151, 26.24590352,  1.99289841, -2.84419587]])
>>> B
array([[ 0.        ,  0.        ],
       [ 0.        ,  0.        ],
       [ 0.01071772, -0.06613267],
       [-0.06613267,  4.42570676]])
```

input_vars = ['Tphi', 'Tdelta']

input_vars_latex = ['T_\\phi', 'T_\\delta']

plot_eigenvalue_parts(*ax=None*, *colors=None*, *show_stable_regions=True*, ***parameter_overrides*)

Returns a matplotlib axis of the real and imaginary parts of the eigenvalues plotted against the provided parameter.

Parameters

ax
[Axes] Matplotlib axes.

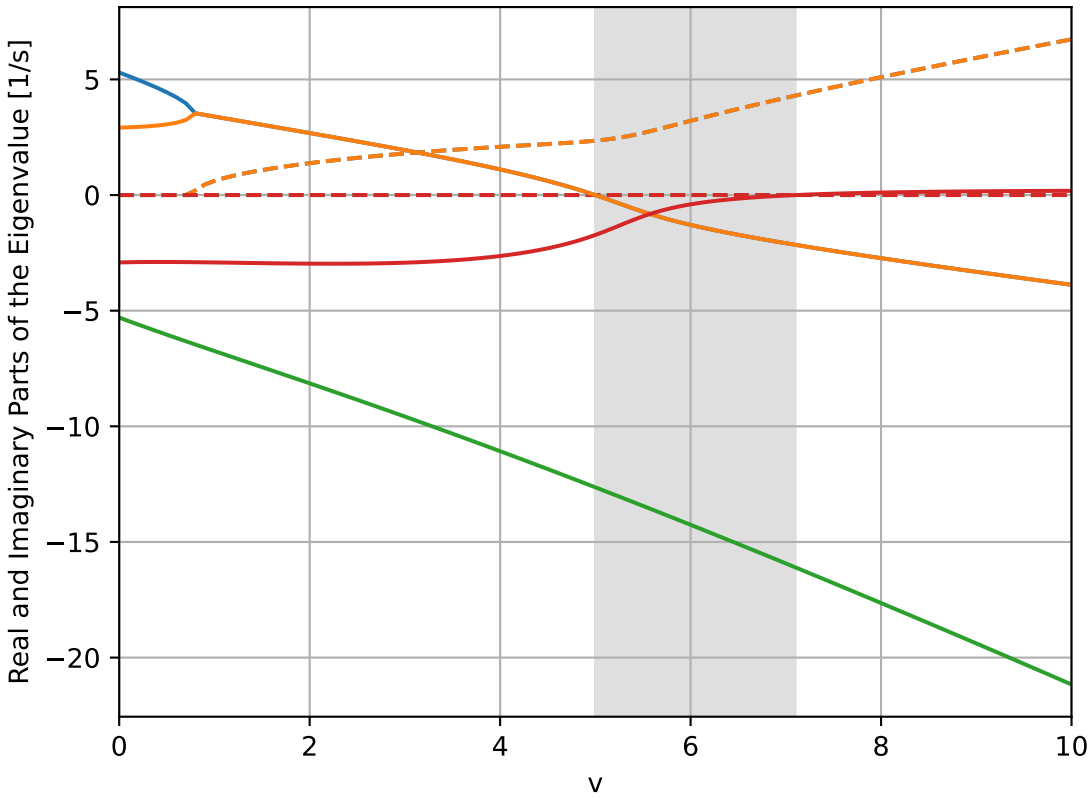
colors
[sequence, len(4)] Matplotlib colors for the 4 modes.

show_stable_regions
[boolean, optional] If true, a grey shaded background will indicate stable regions.

****parameter_overrides**
[dictionary] Parameter keys that map to floats or array_like of floats shape(n,). All keys that map to array_like must be of the same length.

Examples

```
import numpy as np
from bicycleparameters.parameter_dicts import meijaard2007_browser_jason
from bicycleparameters.parameter_sets import Meijaard2007ParameterSet
from bicycleparameters.models import Meijaard2007Model
p = Meijaard2007ParameterSet(meijaard2007_browser_jason, True)
m = Meijaard2007Model(p)
m.plot_eigenvalue_parts(v=np.linspace(0.0, 10.0, num=101))
```



plot_eigenvectors(***parameter_overrides*)

Plots the components of the eigenvectors in the real and imaginary plane.

Parameters

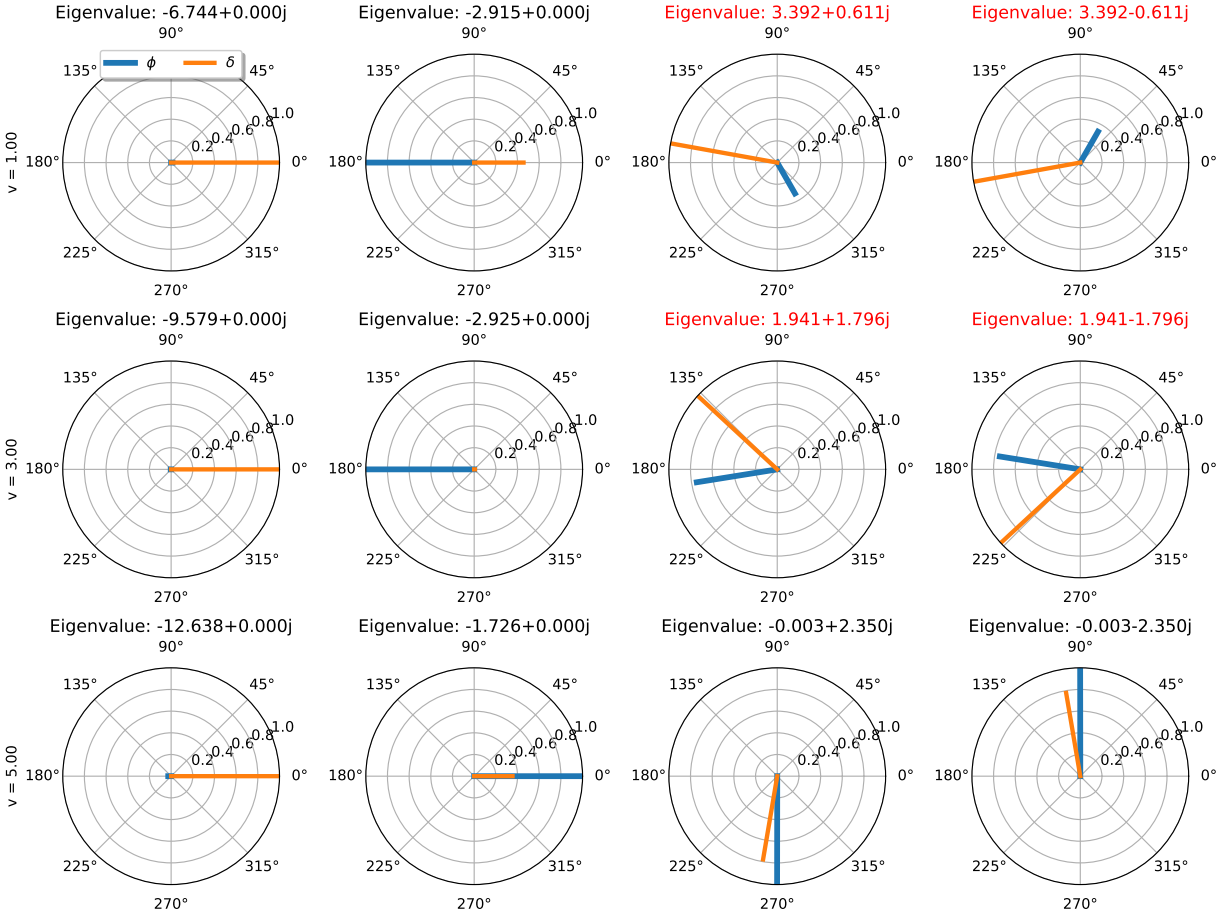
****parameter_overrides**
[dictionary] Parameter keys that map to floats or array_like of floats shape(n,). All keys that map to array_like must be of the same length.

Returns

axes
[ndarray, shape(n, 4)] Polar plot axes for each eigenvector (columns). The rows correspond to a varied parameter.

Examples

```
import numpy as np
from bicycleparameters.parameter_dicts import meijaard2007_browser_jason
from bicycleparameters.parameter_sets import Meijaard2007ParameterSet
from bicycleparameters.models import Meijaard2007Model
p = Meijaard2007ParameterSet(meijaard2007_browser_jason, True)
m = Meijaard2007Model(p)
m.plot_eigenvectors(v=[1.0, 3.0, 5.0])
```



plot_mode_simulations(*times*, ***parameter_overrides*)

Returns matplotlib subplot axes with a simulation of each mode.

Parameters

times
[array_like, shape(n,)] Monotonic increasing time values to simulate over.

****parameter_overrides**
[dictionary] Parameter keys that map to floats or array_like of floats shape(n,). All keys

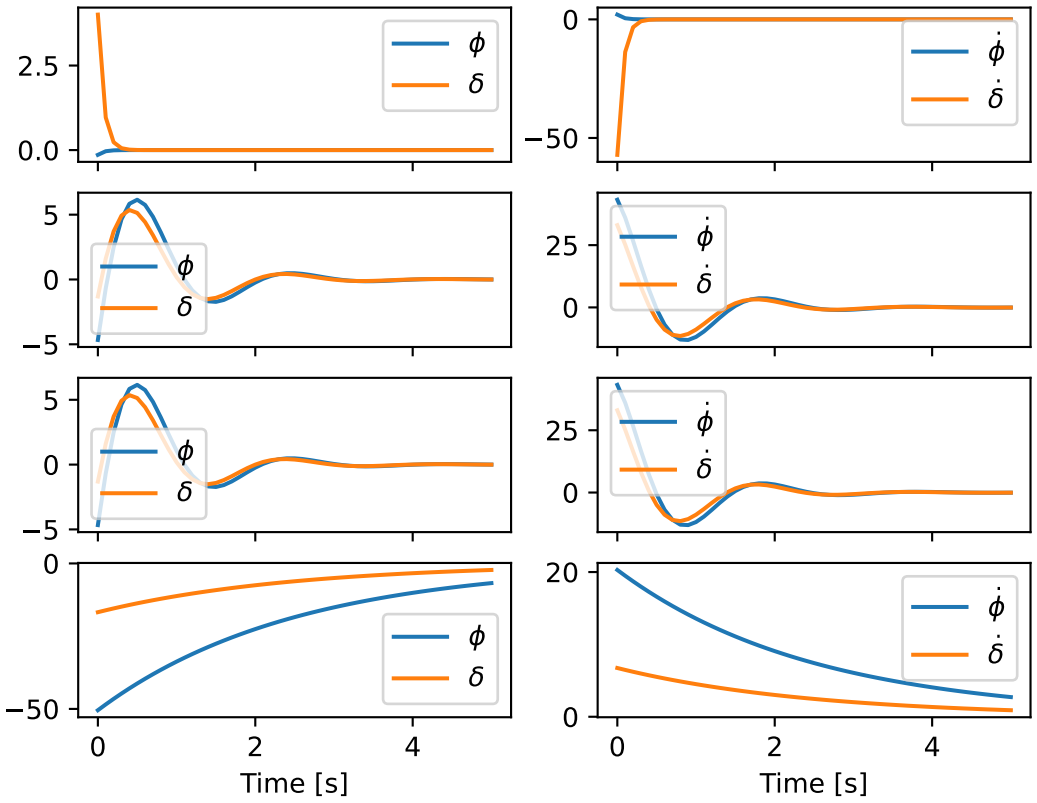
that map to array_like must be of the same length.

Returns

axes
[ndarray, shape(4,2)] Subplot axes with the modes on the rows and the angles in the first column and the angular rates in the second column.

Examples

```
import numpy as np
from bicycleparameters.parameter_dicts import meijaard2007_browser_jason
from bicycleparameters.parameter_sets import Meijaard2007ParameterSet
from bicycleparameters.models import Meijaard2007Model
p = Meijaard2007ParameterSet(meijaard2007_browser_jason, True)
m = Meijaard2007Model(p)
times = np.linspace(0.0, 5.0, num=51)
m.plot_mode_simulations(times, v=6.0)
```



plot_simulation(*times*, *initial_conditions*, *input_func=None*, ***parameter_overrides*)

Returns the state and input trajectories at each time value.

Parameters

times
[array_like, shape(n,)] Monotonic increasing time values to simulate over.

initial_conditions
[array_like, shape(4,)] Initial values of the states.

input_func
[function] Takes form u = f(t, x) where u is array_like, shape(2,).

****parameter_overrides**
[dictionary] Parameter keys that map to floats or array_like of floats shape(n,). All keys that map to array_like must be of the same length.

Returns

axes
[ndarray, shape(3,)] Three subplots that plot the input trajectories, state angle trajectories, and state angular rates.

Examples

```
import numpy as np
from bicycleparameters.parameter_dicts import meijaard2007_browser_jason
from bicycleparameters.parameter_sets import Meijaard2007ParameterSet
from bicycleparameters.models import Meijaard2007Model
p = Meijaard2007ParameterSet(meijaard2007_browser_jason, True)
m = Meijaard2007Model(p)
times = np.linspace(0.0, 5.0, num=51)
x0 = np.deg2rad([10.0, 5.0, 0.0, 0.0])
m.plot_simulation(times, x0, v=6.0)
```

simulate(*times*, *initial_conditions*, *input_func=None*, ***parameter_overrides*)

Returns the state and input trajectories at each time value.

Parameters

times
[array_like, shape(n,)] Monotonic increasing time values to simulate over.

initial_conditions
[array_like, shape(4,)] Initial values of the states.

input_func
[function] Takes form u = f(t, x) where u is array_like, shape(2,).

****parameter_overrides**
[dictionary] Parameter keys that map to floats or array_like of floats shape(n,). All keys that map to array_like must be of the same length.

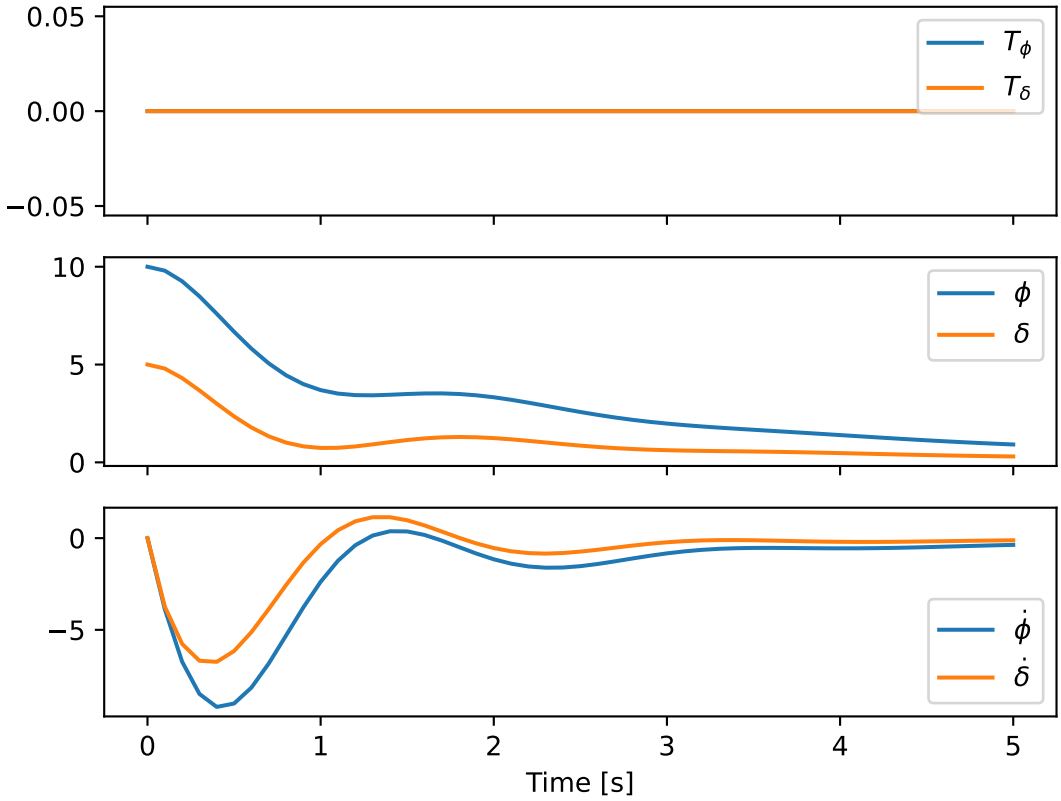
Returns

states
[ndarray, shape(n, 4)] State trajectories over n time values.

inputs
[ndatrray, shape(n, 2)] Input trajectories over n time values.

simulate_modes(*times*, ***parameter_overrides*)

Returns simulation results showing the behavior of each eigenmode.


0.05
0.00
−0.05
T_ϕ
T_δ
10
5
0
ϕ
δ
0
−5
$\dot{\phi}$
$\dot{\delta}$
0
1
2
3
4
5
Time [s]

Parameters

times
[array_like, shape(n,)] Monotonic increasing time values to simulate over.

****parameter_overrides**
[dictionary] Parameter keys that map to floats or array_like of floats shape(n,). All keys that map to array_like must be of the same length.

Returns

results
[ndarray, shape(4, n, 4)] State trajectories for each mode with the shape corresponding to (mode, time, state).

```
state_vars = ['phi', 'delta', 'phidot', 'deltadot']

state_vars_latex = ['\\phi', '\\delta', '\\dot{\\phi}', '\\dot{\\delta}']
```

1.6.8 parameter_sets Module

class Meijaard2007ParameterSet(*parameters*, *includes_rider*)

Bases: *ParameterSet*

Represents the parameters of the benchmark bicycle presented in [Meijaard2007].

The four bodies are:

- B: rear frame + rigid rider
- F: front wheel
- H: front frame (fork & handlebars)
- R: rear wheel

Parameters

parameters
[dictionary] A dictionary mapping variable names to values that contains the following keys:

- `IBxx` : x moment of inertia of the frame/rider [kg*m**2]
- `IBxz` : xz product of inertia of the frame/rider [kg*m**2]
- `IBzz` : z moment of inertia of the frame/rider [kg*m**2]
- `IFxx` : x moment of inertia of the front wheel [kg*m**2]
- `IFyy` : y moment of inertia of the front wheel [kg*m**2]
- `IHxx` : x moment of inertia of the handlebar/fork [kg*m**2]
- `IHxz` : xz product of inertia of the handlebar/fork [kg*m**2]
- `IHzz` : z moment of inertia of the handlebar/fork [kg*m**2]
- `IRxx` : x moment of inertia of the rear wheel [kg*m**2]
- `IRyy` : y moment of inertia of the rear wheel [kg*m**2]
- `c` : trail [m]
- `g` : acceleration due to gravity [m/s**2]

- `lam` : steer axis tilt [rad]
- `mB` : frame/rider mass [kg]
- `mF` : front wheel mass [kg]
- `mH` : handlebar/fork assembly mass [kg]
- `mR` : rear wheel mass [kg]
- `rF` : front wheel radius [m]
- `rR` : rear wheel radius [m]
- `w` : wheelbase [m]
- `xB` : x distance to the frame/rider center of mass [m]
- `xH` : x distance to the frame/rider center of mass [m]
- `zB` : z distance to the frame/rider center of mass [m]
- `zH` : z distance to the frame/rider center of mass [m]

includes_rider
[boolean] True if body B is the combined rear frame and rider in terms of mass and inertia values.

References

[Meijaard2007]

Attributes

par_strings
[dictionary] Maps ASCII strings to their LaTeX string.

body_labels
[list of strings] Single capital letters that correspond to the four rigid bodies in the model.

```
body_labels = ['B', 'F', 'H', 'R']
```

`form_inertia_tensor`(*body*)

Returns the inertia tensor with respect to the global coordinate system and the body's mass center.

Parameters

body
[string] One of the `body_labels`.

Returns

inertia_tensor
[ndarray, shape(3, 3)] Inertia tensor of the body with respect to the body's mass center and the model's coordinate system.

Examples

```
>>> from bicycleparameters.parameter_dicts import meijaard2007_browser_jason
>>> from bicycleparameters.parameter_sets import Meijaard2007ParameterSet
>>> p = Meijaard2007ParameterSet(meijaard2007_browser_jason, True)
>>> p.form_inertia_tensor('H')
array([[ 0.25337959,  0.        , -0.07204524],
       [ 0.        ,  0.24613881,  0.        ],
       [-0.07204524,  0.        ,  0.09557708]])
```

form_mass_center_vector(*body*)

Returns an array representing the 3D vector to the mass center of the body from the origin at the rear wheel contact point.

Parameters

body
[string] One of 'B', 'F', 'H', 'R'.

Returns

ndarray, shape(3,)
A vector containing the X, Y, and X coordinates of the mass center of the body.

Examples

```
>>> from bicycleparameters.parameter_dicts import meijaard2007_browser_jason
>>> from bicycleparameters.parameter_sets import Meijaard2007ParameterSet
>>> p = Meijaard2007ParameterSet(meijaard2007_browser_jason, True)
>>> p.form_mass_center_vector('B')
array([ 0.28909943,  0.        , -1.04029228])
```

mass_center_of(**bodies*)

Returns the vector locating the center of mass of the collection of bodies.

Parameters

bodies
[iterable of strings] One or more of the `body_labels`.

Returns

com
[ndarray, shape(3,)] Vector locating the center of mass of the bodies givien in `bodies`.

Examples

```
>>> from bicycleparameters.parameter_dicts import meijaard2007_browser_jason
>>> from bicycleparameters.parameter_sets import Meijaard2007ParameterSet
>>> p = Meijaard2007ParameterSet(meijaard2007_browser_jason, True)
>>> p.mass_center_of('B', 'H')
array([ 0.31096918,  0.        , -1.02923892])
```

```
par_strings = {'IBxx': 'I_{Bxx}', 'IBxz': 'I_{Bxz}', 'IByy': 'I_{Byy}', 'IBzz':
'I_{Bzz}', 'IFxx': 'I_{Fxx}', 'IFyy': 'I_{Fyy}', 'IHxx': 'I_{Hxx}', 'IHxz':
'I_{Hxz}', 'IHyy': 'I_{Hyy}', 'IHzz': 'I_{Hzz}', 'IRxx': 'I_{Rxx}', 'IRyy':
'I_{Ryy}', 'c': 'c', 'g': 'g', 'lam': '\\lambda', 'mB': 'm_B', 'mF': 'm_F', 'mH':
'm_H', 'mR': 'm_R', 'rF': 'r_F', 'rR': 'r_R', 'v': 'v', 'w': 'w', 'xB': 'x_B',
'xH': 'x_H', 'zB': 'z_B', 'zH': 'z_H'}
```

plot_all(*ax=None*)

Returns matplotlib axes with the geometry and inertial representations of all bodies of the bicycle parameter set.

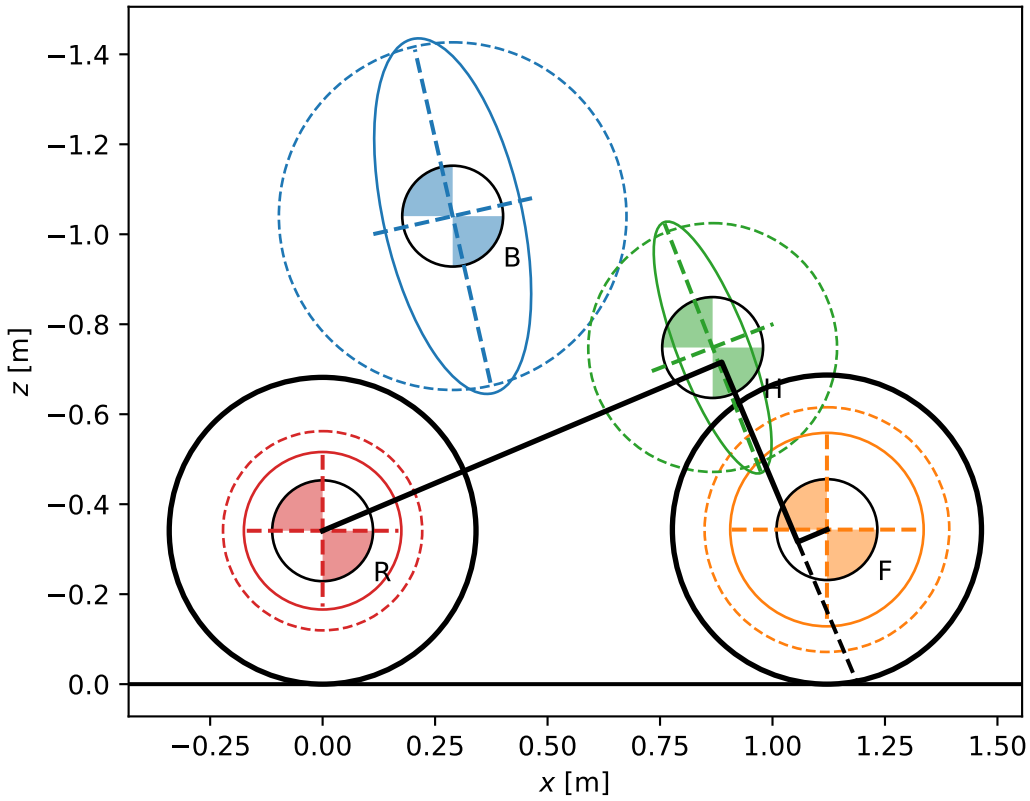
Parameters

ax

[AxesSubplot, optional] An axes to draw on, otherwise one is created.

Examples

```
from bicycleparameters.parameter_dicts import meijaard2007_browser_jason
from bicycleparameters.parameter_sets import Meijaard2007ParameterSet
p = Meijaard2007ParameterSet(meijaard2007_browser_jason, True)
p.plot_all()
```



plot_body_mass_center(*body*, *ax=None*)

Returns a matplotlib axes with a mass center symbol for the specified body to the plot.

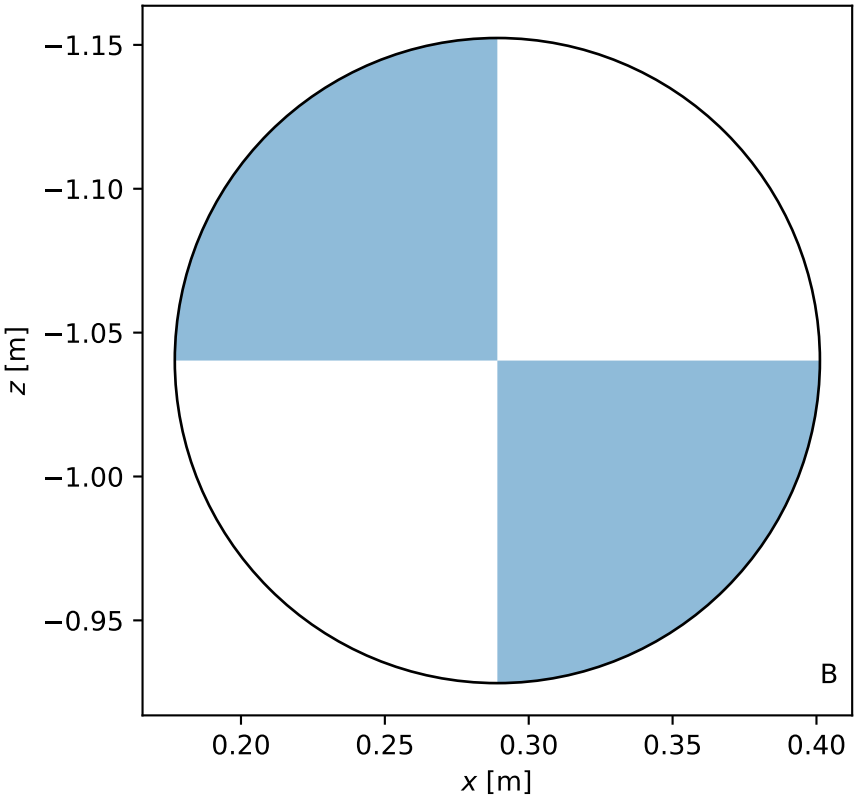
Parameters

body
[string] The body string: `F`, `H`, `B`, or `R`.

ax
[SubplotAxes, optional] Axes to plot on.

Examples

```
from bicycleparameters.parameter_dicts import meijaard2007_browser_jason
from bicycleparameters.parameter_sets import Meijaard2007ParameterSet
p = Meijaard2007ParameterSet(meijaard2007_browser_jason, True)
p.plot_body_mass_center('B')
```



plot_body_principal_inertia_ellipsoid(*body*, *ax=None*)

Returns a matplotlib axes with an ellipse that respresnts the XZ plane view of a constant density ellipsoid which has the same principal moments and axes of inertia as the body.

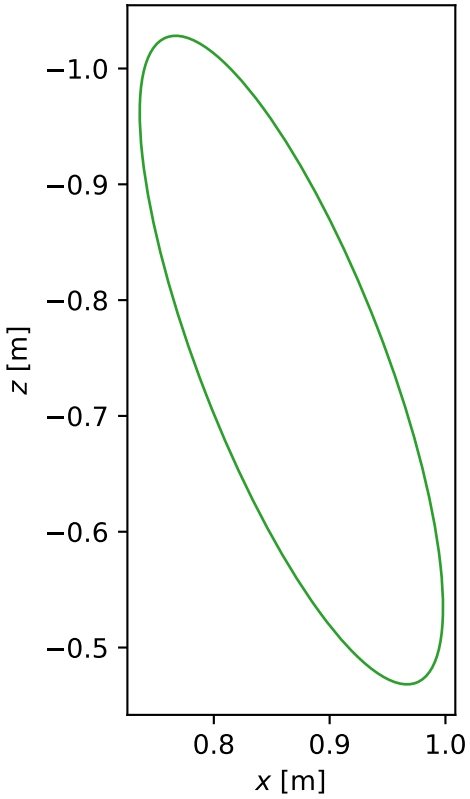
Parameters

body
[string] One of the `body_labels`.

ax
[SubplotAxes, optional] Axes to plot on.

Examples

```
from bicycleparameters.parameter_dicts import meijaard2007_browser_jason
from bicycleparameters.parameter_sets import Meijaard2007ParameterSet
p = Meijaard2007ParameterSet(meijaard2007_browser_jason, True)
p.plot_body_principal_inertia_ellipsoid('H')
```



plot_body_principal_radii_of_gyration(*body*, *ax=None*)

Returns a matplotlib axes with lines and a circle that indicate the principal radii of gyration of the specified body.

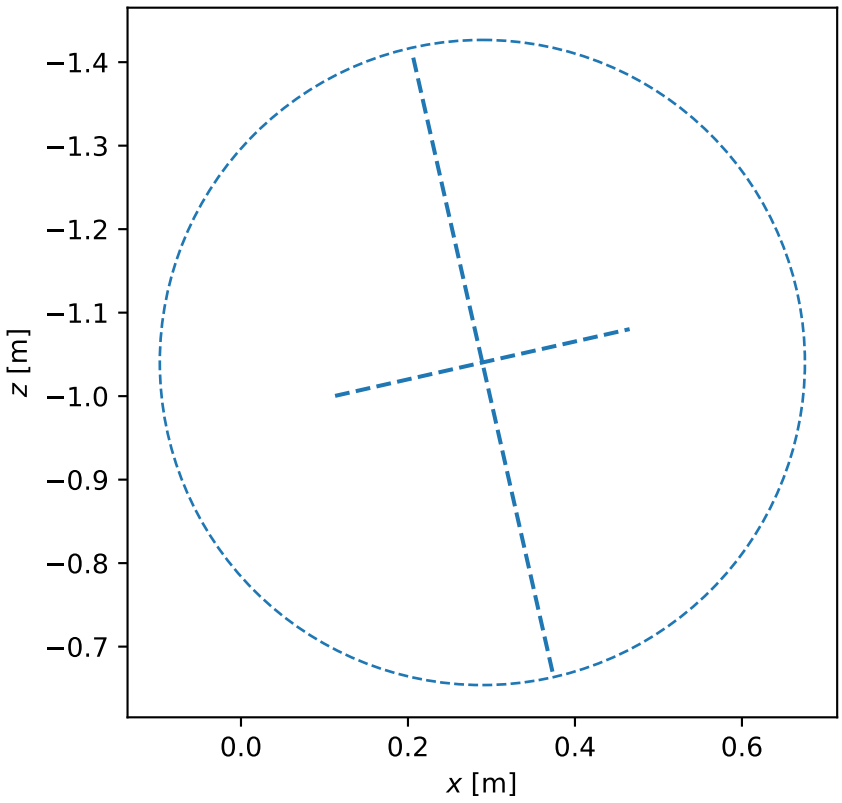
Parameters

body
[string] One of the `body_labels`.

ax
[SubplotAxes, optional] Axes to plot on.

Examples

```
from bicycleparameters.parameter_dicts import meijaard2007_browser_jason
from bicycleparameters.parameter_sets import Meijaard2007ParameterSet
p = Meijaard2007ParameterSet(meijaard2007_browser_jason, True)
p.plot_body_principal_radii_of_gyration('B')
```



plot_geometry(*show_steer_axis=True*, *ax=None*)

Returns a matplotlib axes with a simple drawing of the bicycle's geometry.

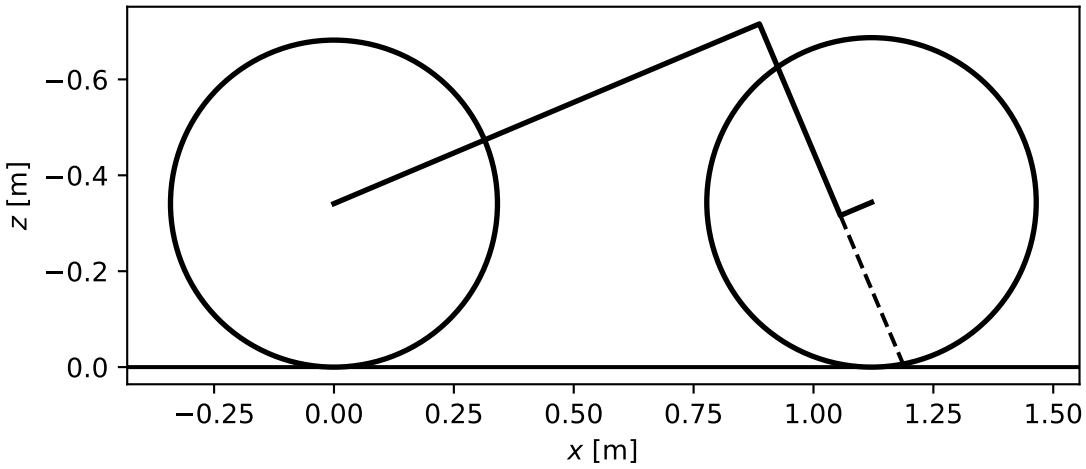
Parameters

show_steer_axis
[boolean] If true, a dotted line will be plotted along the steer axis from the front wheel center to the ground.

ax
[AxesSubplot, optional] An axes to draw on, otherwise one is created.

Examples

```
from bicycleparameters.parameter_dicts import meijaard2007_browser_jason
from bicycleparameters.parameter_sets import Meijaard2007ParameterSet
p = Meijaard2007ParameterSet(meijaard2007_browser_jason, True)
p.plot_geometry()
```



plot_mass_centers(*bodies=None*, *ax=None*)

Returns a matplotlib axes with mass center indicators for each body.

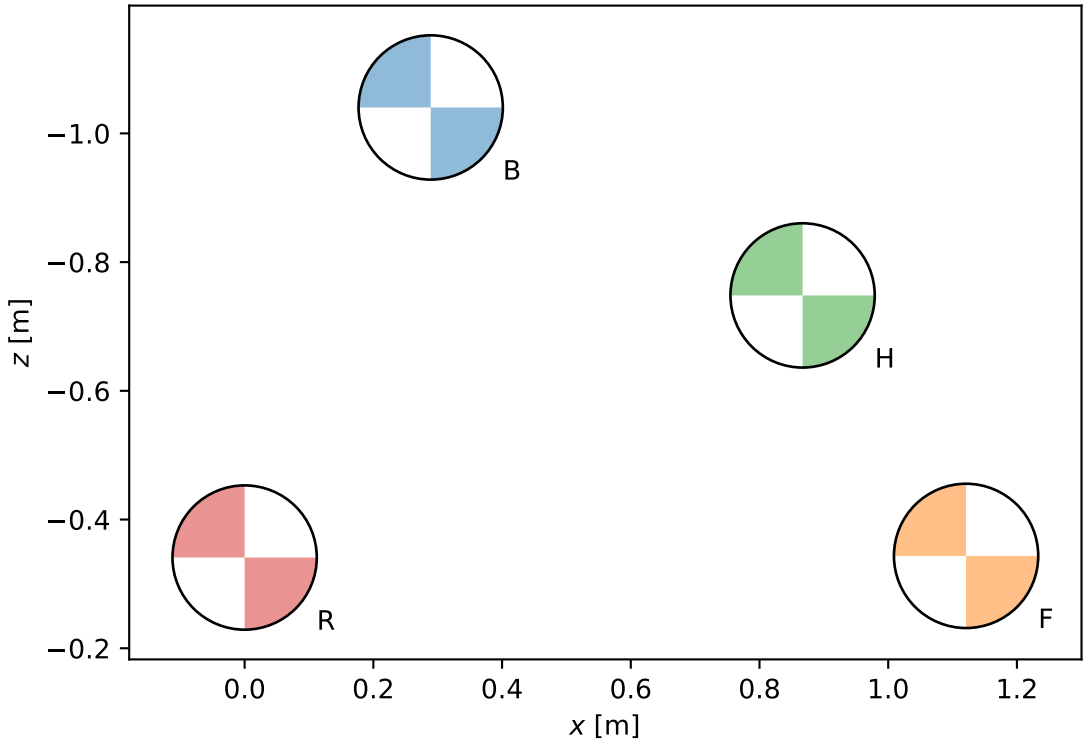
Parameters

bodies: list of strings, optional
A subset of the strings present in the class attribute `body_labels`.

ax: matplotlib Axes, optional
An axes to plot on.

Examples

```
from bicycleparameters.parameter_dicts import meijaard2007_browser_jason
from bicycleparameters.parameter_sets import Meijaard2007ParameterSet
p = Meijaard2007ParameterSet(meijaard2007_browser_jason, True)
p.plot_mass_centers()
```



plot_principal_inertia_ellipsoids(*bodies=None*, *ax=None*)

Returns a Matplotlib axes with 2D representations of 3D solid uniform ellipsoids that have the same inertia as the body.

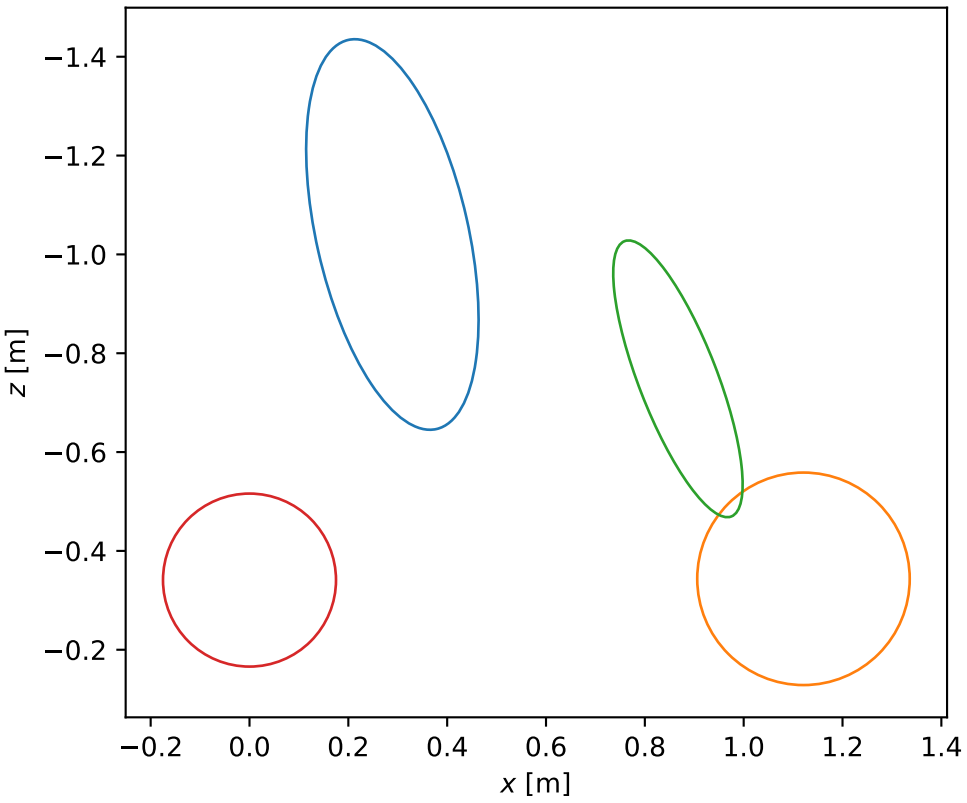
Parameters

bodies: list of strings, optional
A subset of the strings present in the class attribute `body_labels`.

ax
[AxesSubplot, optional] An axes to draw on, otherwise one is created.

Examples

```
from bicycleparameters.parameter_dicts import meijaard2007_browser_jason
from bicycleparameters.parameter_sets import Meijaard2007ParameterSet
p = Meijaard2007ParameterSet(meijaard2007_browser_jason, True)
p.plot_principal_inertia_ellipsoids()
```



plot_principal_radii_of_gyration(*bodies=None*, *ax=None*)

Returns a matplotlib axis with principal radii of all bodies shown.

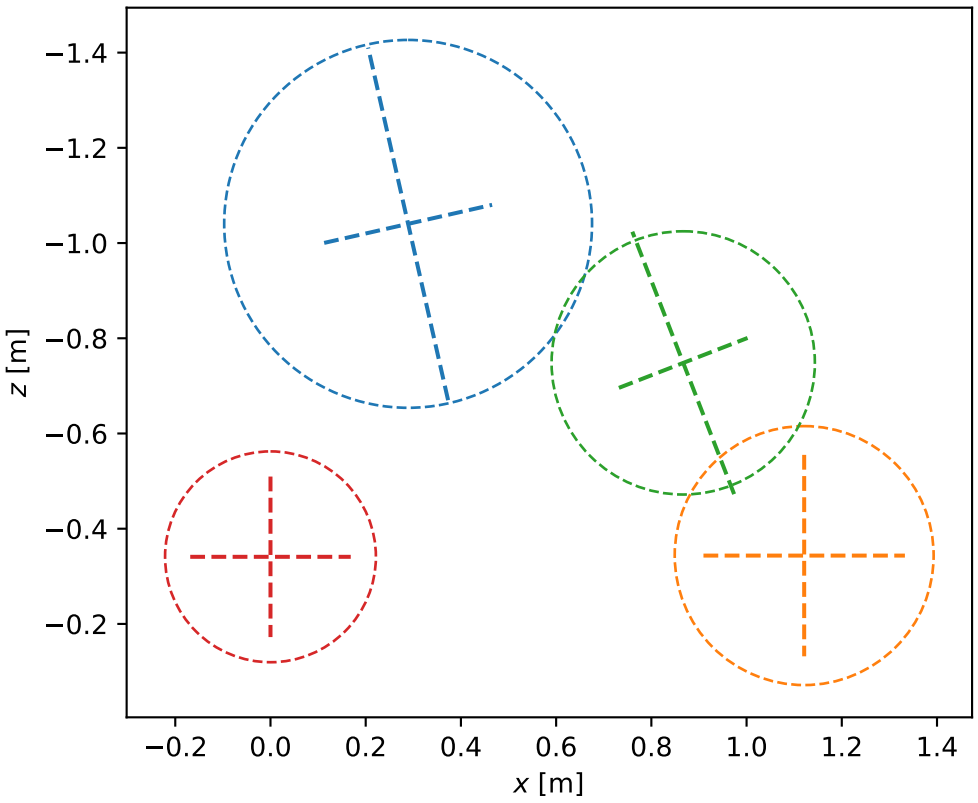
Parameters

bodies: list of strings, optional
A subset of the strings present in the class attribute `body_labels`.

ax: matplotlib Axes, optional
An axes to plot on.

Examples

```
from bicycleparameters.parameter_dicts import meijaard2007_browser_jason
from bicycleparameters.parameter_sets import Meijaard2007ParameterSet
p = Meijaard2007ParameterSet(meijaard2007_browser_jason, True)
p.plot_principal_radii_of_gyration()
```



class Moore2019ParameterSet(*parameters*)

Bases: *ParameterSet*

Represents the parameters of the bicycle parameterization presented in [1].

The four bodies are:

- D: rear frame
- F: front wheel
- H: front frame (fork & handlebars)
- P: rigid rider
- R: rear wheel

Parameters

parameters
[dictionary] A dictionary mapping variable names to values.

References

[1]

Attributes

par_strings
[dictionary] Maps ASCII strings to their LaTeX string.

body_labels
[list of strings] Single capital letters that correspond to the five rigid bodies in the model.

```
body_labels = ['D', 'F', 'H', 'P', 'R']
```

`form_mass_center_vector`(*body*)

Returns an array representing the vector to the mass center of the body.

Parameters

body
[string] One of 'P', 'D', 'F', 'H', 'R'.

Returns

ndarray, shape(3,)
A vector containing the X, Y, and X coordinates of the mass center of the body.

`mass_center_of`(**bodies*)

Returns the vector locating the center of mass of the collection of bodies.

Parameters

bodies
[iterable of strings] Subset from `body_labels`.

Returns

com
[ndarray, shape(3,)] Vector locating the center of mass of the bodies given in `bodies`.

```
non_min_par_strings = {'alphaF': '\\alpha_F', 'alphaR': '\\alpha_R', 'kFbb':
'k_{Fbb}', 'kRbb':  'k_{Rbb}', 'xF': 'x_F', 'xR': 'x_R', 'yD': 'y_D', 'yF': 'y_F',
'yH': 'y-H', 'yP': 'y_P', 'yR': 'y_R', 'zF': 'z_F', 'zR': 'z_R'}
```

```
par_strings = {'alphaD': '\\alpha_D', 'alphaH': '\\alpha_H', 'alphaP': '\\alpha_P',
'c':  'c', 'g':  'g', 'kDaa':  'k_{Daa}', 'kDbb':  'k_{Dbb}', 'kDyy':  'k_{Dyy}',
'kFaa':  'k_{Faa}', 'kFyy':  'k_{Fyy}', 'kHaa':  'k_{Haa}', 'kHbb':  'k_{Hbb}',
'kHyy':  'k_{Hyy}', 'kPaa':  'k_{Paa}', 'kPbb':  'k_{Pbb}', 'kPyy':  'k_{Pyy}',
'kRaa':  'k_{Raa}', 'kRyy':  'k_{Ryy}', 'lP': 'l_P', 'lam':  '\\lambda', 'mD':
'm_D', 'mF': 'm_F', 'mH': 'm_H', 'mP': 'm_B', 'mR': 'm_R', 'rF': 'r_F', 'rR': 'r_R',
'v':  'v', 'w':  'w', 'wP': 'w_P', 'xD': 'x_D', 'xH': 'x_H', 'xP': 'x_P', 'zD':
'z_D', 'zH': 'z-H', 'zP': 'z_P'}
```

`plot_all`(*ax=None*)

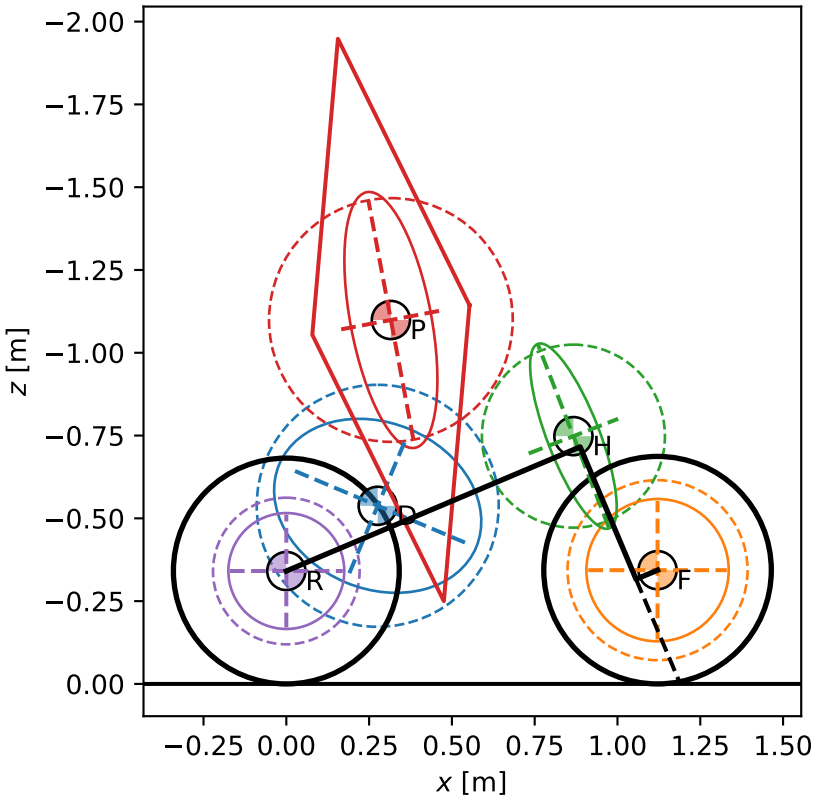
Returns matplotlib axes with the geometry and inertial representations of all bodies of the bicycle parameter set.

Parameters

ax: matplotlib Axes, optional
An axes to plot on.

Examples

```
from bicycleparameters.parameter_dicts import moore2019_browser_jason
from bicycleparameters.parameter_sets import Moore2019ParameterSet
p = Moore2019ParameterSet(moore2019_browser_jason)
p.plot_all()
```



plot_body_mass_center(*body*, *ax=None*)

Returns a matplotlib axes with a mass center symbol for the specified body to the plot.

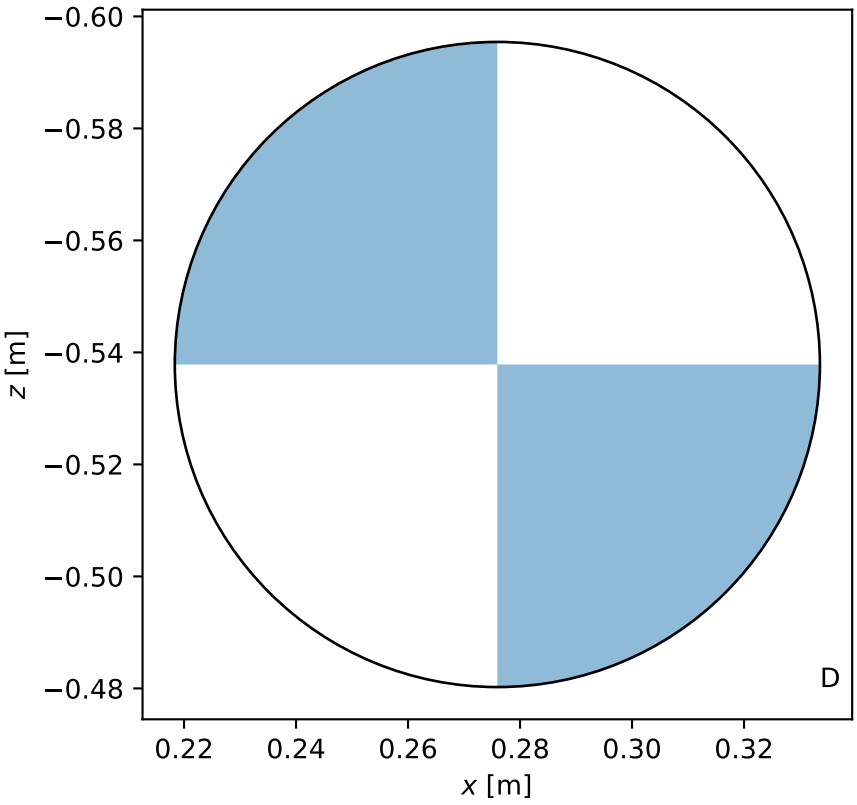
Parameters

body
[string] The body string: D, F, H, P, or R

ax
[SubplotAxes, optional] Axes to plot on.

Examples

```
from bicycleparameters.parameter_dicts import moore2019_browser_jason
from bicycleparameters.parameter_sets import Moore2019ParameterSet
p = Moore2019ParameterSet(moore2019_browser_jason)
p.plot_body_mass_center('D')
```



plot_body_principal_inertia_ellipsoid(*body*, *ax=None*)

Returns a matplotlib axes with an ellipse that respresnts the XZ plane view of a constant density ellipsoid which has the same principal moments and axes of inertia as the body.

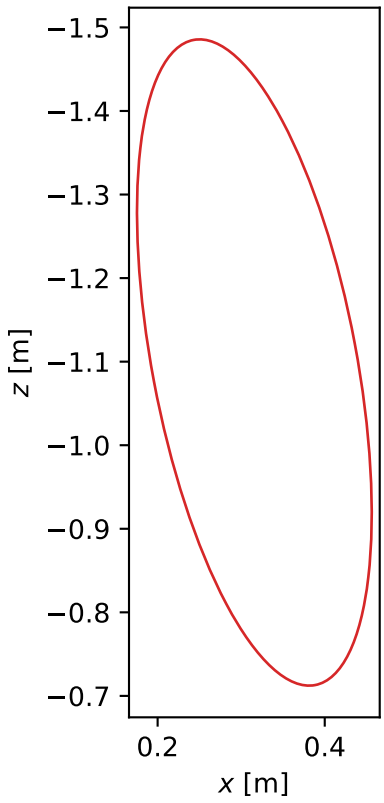
Parameters

body
[string] The body string: D, F, H, P, or R

ax
[SubplotAxes, optional] Axes to plot on.

Examples

```
from bicycleparameters.parameter_dicts import moore2019_browser_jason
from bicycleparameters.parameter_sets import Moore2019ParameterSet
p = Moore2019ParameterSet(moore2019_browser_jason)
p.plot_body_principal_inertia_ellipsoid('P')
```



plot_body_principal_radii_of_gyration(*body*, *ax=None*)

Returns a matplotlib axes with lines and a circle that indicate the principal radii of gyration of the specified body.

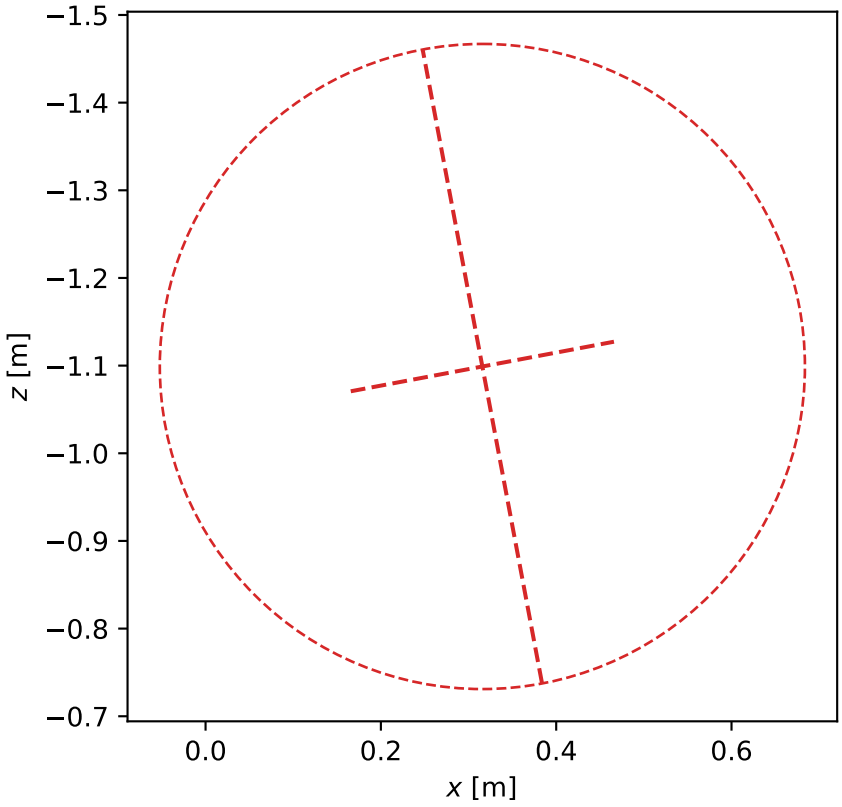
Parameters

body
[string] The body string: D, F, H, P, or R

ax
[SubplotAxes, optional] Axes to plot on.

Examples

```
from bicycleparameters.parameter_dicts import moore2019_browser_jason
from bicycleparameters.parameter_sets import Moore2019ParameterSet
p = Moore2019ParameterSet(moore2019_browser_jason)
p.plot_body_principal_radii_of_gyration('P')
```



plot_geometry(*show_steer_axis=True*, *ax=None*)

Returns a matplotlib axes with the simplest drawing of the bicycle's geometry.

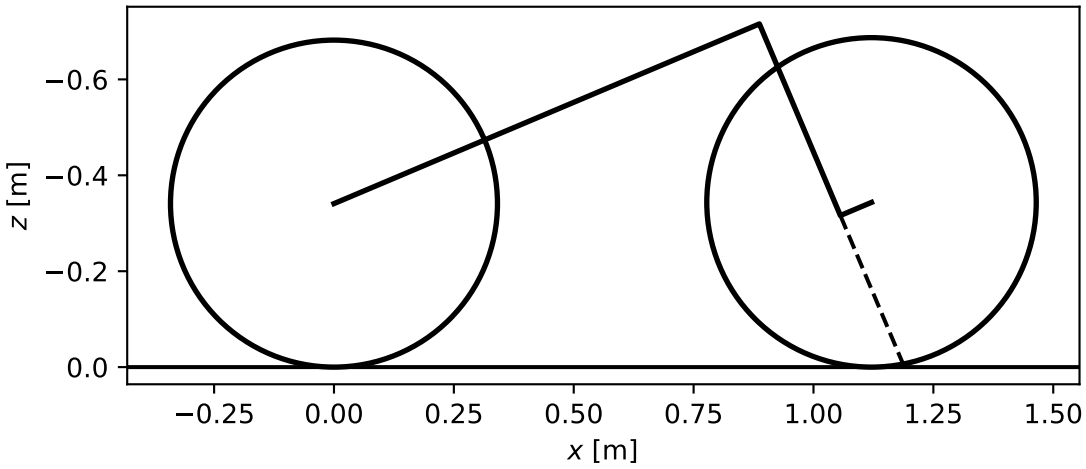
Parameters

show_steer_axis
[boolean] If true, a dotted line will be plotted along the steer axis from the front wheel center to the ground.

ax
[AxesSubplot, optional] An axes to draw on, otherwise one is created.

Examples

```
from bicycleparameters.parameter_dicts import moore2019_browser_jason
from bicycleparameters.parameter_sets import Moore2019ParameterSet
p = Moore2019ParameterSet(moore2019_browser_jason)
p.plot_geometry()
```



plot_mass_centers(*bodies=None*, *ax=None*)

Returns a matplotlib axes with a mass center symbols for the specified bodies to the plot.

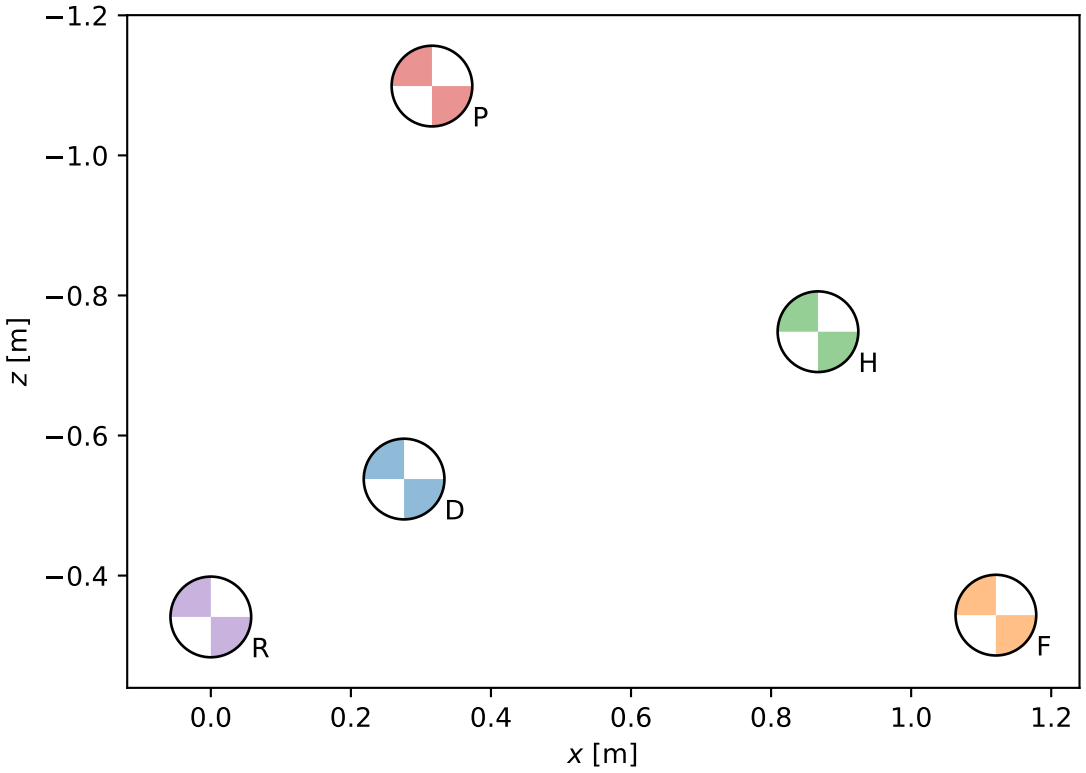
Parameters

bodies: list of strings, optional
A subset of the strings present in the class attribute `body_labels`.

ax: matplotlib Axes, optional
An axes to plot on.

Examples

```
from bicycleparameters.parameter_dicts import moore2019_browser_jason
from bicycleparameters.parameter_sets import Moore2019ParameterSet
p = Moore2019ParameterSet(moore2019_browser_jason)
p.plot_mass_centers()
```



plot_person_diamond(*show_cross=False*, *ax=None*)

Plots a diamond that represents the approximate person's physical extents.

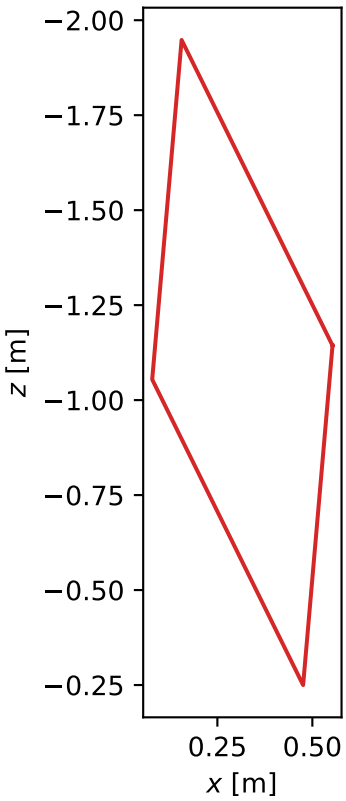
Parameters

show_cross
[boolean, optional] Plots a cross in the diamond that spans opposite vertices.

ax
[AxesSubplot, optional] An axes to draw on, otherwise one is created.

Examples

```
from bicycleparameters.parameter_dicts import moore2019_browser_jason
from bicycleparameters.parameter_sets import Moore2019ParameterSet
p = Moore2019ParameterSet(moore2019_browser_jason)
p.plot_person_diamond()
```



plot_principal_inertia_ellipsoids(*bodies=None*, *ax=None*)

Returns a Matplotlib axes with 2D representations of 3D solid uniform ellipsoids that have the same inertia as the body.

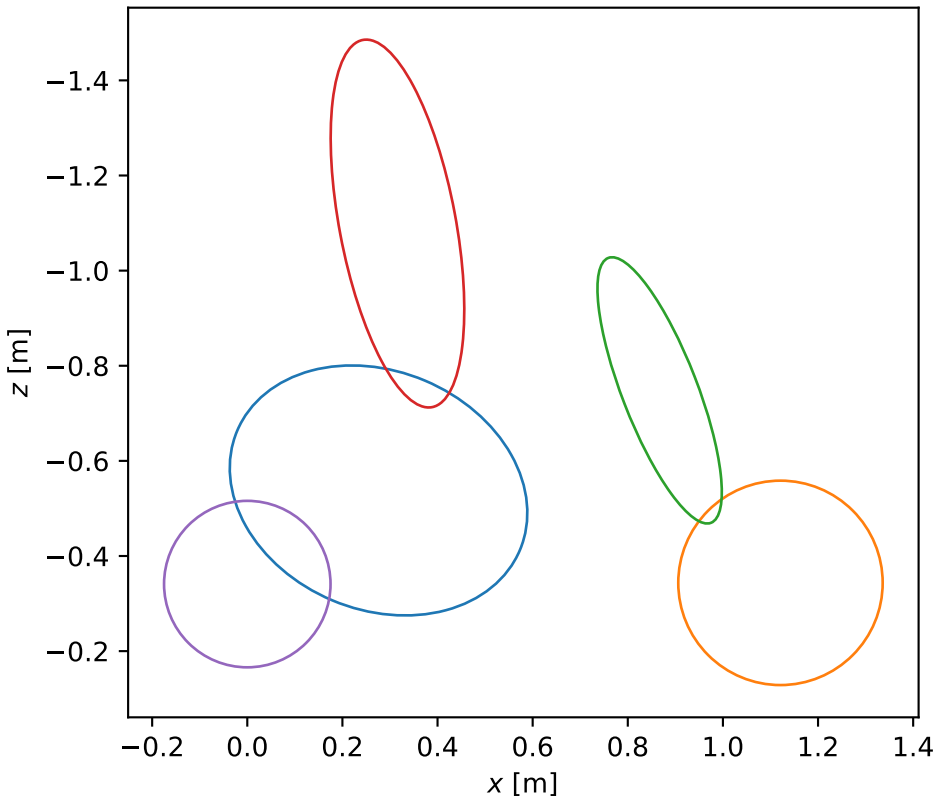
Parameters

bodies: list of strings, optional
A subset of the strings present in the class attribute `body_labels`.

ax: matplotlib Axes, optional
An axes to plot on.

Examples

```
from bicycleparameters.parameter_dicts import moore2019_browser_jason
from bicycleparameters.parameter_sets import Moore2019ParameterSet
p = Moore2019ParameterSet(moore2019_browser_jason)
p.plot_principal_inertia_ellipsoids()
```



plot_principal_radii_of_gyration(*bodies=None*, *ax=None*)

Returns a matplotlib axes with lines and a circle that indicate the principal radii of gyration for all five bodies.

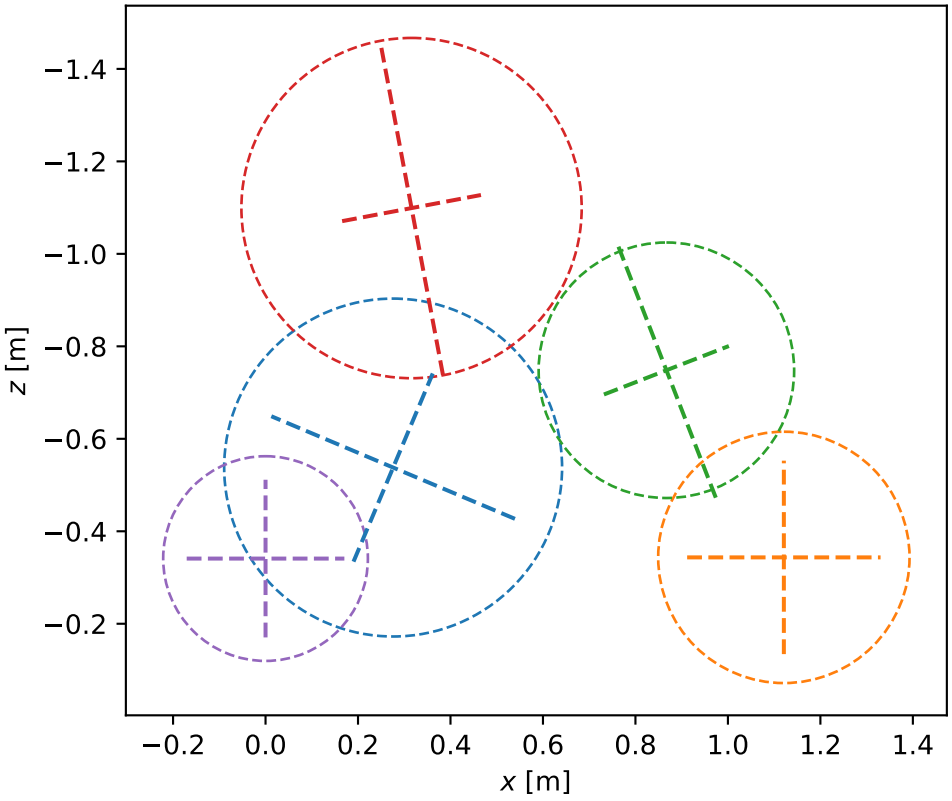
Parameters

bodies
[list of strings, optional] Either ['D', 'F', 'H', 'P', 'R'] or a subset thereof.

ax
[AxesSubplot, optional] An axes to draw on, otherwise one is created.

Examples

```
from bicycleparameters.parameter_dicts import moore2019_browser_jason
from bicycleparameters.parameter_sets import Moore2019ParameterSet
p = Moore2019ParameterSet(moore2019_browser_jason)
p.plot_principal_radii_of_gyration()
```



to_parameterization(*name*)

Returns a specific parameter set based on the provided parameterization name.

Parameters

name

[string] The name of the parameterization. These should correspond to a subclass of a `ParameterSet` and the name will be the string that precedes "ParameterSet". For example, the parameterization name of `Meijaard2007ParameterSet` is `Meijaard2007`.

Returns

ParmeterSet

If a different parameterization is requested and this class can convert itself, it will return a new parameter set of the correct parameterization.

Examples

```
>>> from bicycleparameters.parameter_dicts import moore2019_browser_jason
>>> from bicycleparameters.parameter_sets import Moore2019ParameterSet
>>> moore_set = Moore2019ParameterSet(moore2019_browser_jason)
>>> moore_set.mass_center_of('P', 'D')
array([ 0.31156277,  0.        , -1.03972442])
>>> meijaard_set = moore_set.to_parameterization('Meijaard2007')
>>> meijaard_set.mass_center_of('B')
array([ 0.31156277,  0.        , -1.03972442])
```

class ParameterSet(*par_dict*)

Bases: ABC

A parameter set is a collection of constants with associated floating point values that are present in a set of differential algebraic equations that represent a multibody bicycle model. These pairs are typically defined in a specific academic article, dissertation, book chapter, or section and subclasses should be named in a way that ties them to that written work. Parameter sets must be named with this pattern `NameOfMySetParameterSet` where `NameOfMySet` is a unique name other than something that includes `ParameterSet`.

A parameter set, or a subset of the parameters, can be used with multiple different models. A parameter set is associated with a particular parameterization of one or more models. Parameter sets can be converted to equivalent parameter sets, but only by assuming a particular model configuration. The obvious configuration for a bicycle model is the upright, zero steer state. But if, for example, a rider configuration is included, then some nominal configuration would need to be defined for conversion consistency.

Each parameter set should have a unique set of ASCII strings that represent the constants in a model.

Attributes

par_strings
[dictionary] Maps ASCII strings to their LaTeX string.

par_strings = {'aR': 'a_R', 'beta': '\\beta'}

to_ini(*fname*)

Writes parameters to file in the INI format. Metadata is not included.

Parameters

fname
[string] Path to file.

to_parameterization(*name*)

Returns a specific parameter set based on the provided parameterization name.

Parameters

name
[string] The name of the parameterization. These should correspond to a subclass of a `ParameterSet` and the name will be the string that precedes "ParameterSet". For example, the parameterization name of `Meijaard2007ParameterSet` is `Meijaard2007`.

Returns

ParmeterSet
If a different parameterization is requested and this class can convert itself, it will return a new parameter set of the correct parameterization.

`to_yaml`(*fname*)

Writes parameters to file in the YAML format.

Parameters

fname
[string] Path to file.

1.6.9 `period` Module

`average_rectified_sections`(*data*)

Returns a slice of an oscillating data vector based on the max and min of the mean of the sections created by retifiying the data.

Parameters

data
[ndarray, shape(n,)]

Returns

data
[ndarray, shape(m,)] A slice where m is typically less than n.

Notes

This is a function to try to handle the fact that some of the data from the torsional pendulum had a beating like phenomena and we only want to select a section of the data that doesn't seem to exhibit the phenomena.

`calc_periods_for_files`(*directory*, *filenames*, *forkIsSplit*)

Calculates the period for all filenames in directory.

Parameters

directory
[string] This is the path to the RawData directory.

filenames
[list] List of all the mat file names in the RawData directory.

forkIsSplit
[boolean] True if the fork is broken into a handlebar and fork and false if the fork and handlebar was measured together.

Returns

periods
[dictionary] Contains all the periods for the mat files in the RawData directory.

`check_for_period`(*mp*, *forkIsSplit*)

Returns whether the fork is split into two pieces and whether the period calculations need to happen again.

Parameters

mp
[dictionary] Dictionary the measured parameters.

forkIsSplit
[boolean] True if the fork is broken into a handlebar and fork and false if the fork and handlebar was measured together.

Returns

forcePeriodCalc
[boolean] True if there wasn't enough period data in mp, false if there was.

forkIsSplit
[boolean] True if the fork is broken into a handlebar and fork and false if the fork and handlebar was measured together.

`fit_goodness`(*ym*, *yp*)

Calculate the goodness of fit.

Parameters

ym
[ndarray, shape(n,)] The vector of measured values.

yp
[ndarry, shape(n,)] The vector of predicted values.

Returns

rsq
[float] The r squared value of the fit.

SSE
[float] The error sum of squares.

SST
[float] The total sum of squares.

SSR
[float] The regression sum of squares.

`get_period`(*data*, *sampleRate*, *pathToPlotFile*)

Returns the period and uncertainty for data resembling a decaying oscillation.

Parameters

data
[ndarray, shape(n,)] A time series that resembles a decaying oscillation.

sampleRate
[int] The frequency that data was sampled at.

pathToPlotFile
[string] A path to the file to print the plots.

Returns

T
[ufloat] The period of oscillation and its uncertainty.

`get_period_from_truncated`(*data*, *sampleRate*, *pathToPlotFile*)

`get_period_key`(*matData*, *forkIsSplit*)

Returns a dictionary key for the period entries.

Parameters

matData
[dictionary] The data imported from a pendulum mat file.

forkIsSplit
[boolean] True if the fork is broken into a handlebar and fork and false if the fork and handlebar was measured together.

Returns

key
[string] A key of the form 'T[pendulum][part][orientation]'. For example, if it is the frame that was hung as a torsional pendulum at the second orientation angle then the key would be 'TtB2'.

`get_sample_rate`(*matData*)

Returns the sample rate for the data.

`jac_fitfunc`(*p*, *t*)

Calculate the Jacobian of a decaying oscillation function.

Uses the analytical formulations of the partial derivatives.

Parameters

p
[the five parameters of the equation]

t
[time vector]

Returns

jac
[The jacobian, the partial of the vector function with respect to the]

parameters vector. A 5 x N matrix where N is the number of time steps.

`make_guess`(*data*, *sampleRate*)

Returns a decent starting point for fitting the decaying oscillation function.

`plot_osfit`(*t*, *ym*, *yf*, *p*, *rsq*, *T*, *m=None*, *fig=None*)

Plot fitted data over the measured

Parameters

t
[ndarray (n,)] Measurement time in seconds

ym
[ndarray (n,)] The measured voltage

yf
[ndarray (n,)]

p
[ndarray (5,)] The fit parameters for the decaying osicallation fucntion

rsq
[float] The r squared value of y (the fit)

T
[float] The period

m
[float] The maximum value to plot

Returns

fig
[the figure]

`select_good_data`(*data*, *percent*)

Returns a slice of the data from the index at maximum value to the index at a percent of the maximum value.

Parameters

data
[ndarray, shape(1,)] This should be a decaying function.

percent
[float] The percent of the maximum to clip.

This basically snips of the beginning and end of the data so that the super damped tails are gone and also any weirdness at the beginning.

1.6.10 `plot` Module

`compare_bode_bicycles`(*bikes*, *speed*, *u*, *y*, *fig=None*)

Returns a figure with the Bode plots of multiple bicycles.

Parameters

bikes
[list] A list of bicycleparameters.Bicycle instances.

speed
[float] The speed at which to evaluate the system.

u
[integer] An integer between 0 and 1 corresponding to the inputs roll torque and steer torque.

y
[integer] An integer between 0 and 3 corresponding to the inputs roll rate, steer rate, roll angle and steer angle.

Returns

fig
[matplotlib.Figure instance] The Bode plot.

Notes

The phases are matched around zero degrees at with respect to the first frequency.

`plot_eigenvalues`(*bikes*, *speeds*, *colors=None*, *linestyles=None*, *largest=False*, *show=False*)

Returns a figure with the eigenvalues vs speed for multiple bicycles.

Parameters

bikes
[list] A list of Bicycle objects.

speeds
[ndarray, shape(n,)] An array of speeds.

colors
[list] A list of matplotlib colors for each bicycle.

linestyles
[list] A list of matplotlib linestyles for each bicycle.

largest
[boolean] If true, only plots the largest eigenvalue.

Returns

fig
[matplotlib figure]

1.6.11 `rider` Module

`configure_rider`(*pathToRider*, *bicycle*, *bicyclePar*, *measuredPar*, *draw*)

Returns the rider parameters, bicycle paramaters with a rider and a human object that is configured to sit on the bicycle.

Parameters

pathToRider
[string] Path to the rider's data folder.

bicycle
[string] The short name of the bicycle.

bicyclePar
[dictionary] Contains the benchmark bicycle parameters for a bicycle.

measuredPar
[dictionary] Contains the measured values of the bicycle.

draw
[boolean, optional] If true, visual python will be used to draw a three dimensional image of the rider.

Returns

riderpar
[dictionary] The inertial parameters of the rider with reference to the benchmark coordinate system.

human
[yeadon.human] The human object that represents the rider seated on the bicycle.

bicycleRiderPar
[dictionary] The benchmark parameters of the bicycle with the rider added to the rear frame.

`rider_on_bike`(*benchmarkPar*, *measuredPar*, *yeadonMeas*, *yeadonCFG*, *drawrider*)

Returns a yeadon human configured to sit on a bicycle.

Parameters

benchmarkPar
[dictionary] A dictionary containing the benchmark bicycle parameters.

measuredPar
[dictionary] A dictionary containing the raw geometric measurements of the bicycle.

yeadonMeas
[str] Path to a text file that holds the 95 yeadon measurements. See yeadon documentation.

yeadonCFG
[str] Path to a text file that holds configuration variables. See yeadon documentation. As of now, only 'somersalt' angle can be set as an input. The remaining variables are either zero or calculated in this method.

drawrider
[bool] Switch to draw the rider, with vectors pointing to the desired position of the hands and feet of the rider (at the handles and bottom bracket). Requires python-visual.

Returns

human
[yeadon.Human] Human object is returned with an updated configuration. The dictionary, taken from H.CFG, has the following key's values updated:

'PJ1extension' 'J1J2flexion' 'CA1extension' 'CA1adduction' 'CA1rotation' 'A1A2extension' 'somersault' 'PK1extension' 'K1K2flexion' 'CB1extension' 'CB1abduction' 'CB1rotation' 'B1B2extension'

Notes

Requires that the bike object has a raw data text input file that contains the measurements necessary to situate a rider on the bike (i.e. `<pathToData>/bicycles/<short name>/RawData/<short name>Measurements.txt`).

`yeadon_vec_to_bicycle_vec`(*vector*, *measured_bicycle_par*, *benchmark_bicycle_par*)

Parameters

vector
[ndarray, shape(3, 1)] A vector from the Yeadon origin to a point expressed in the Yeadon reference frame.

measured_bicycle_par
[dictionary] The raw bicycle measurements.

benchmark_bicycle_par
[dictionary] The Meijaard 2007 et. al parameters for this bicycle.

Returns

vector_wrt_bike
[ndarray, shape(3, 1)] The vector from the bicycle origin to the same point above expressed in the bicycle reference frame.

1.6.12 `tables` Module

`class Table`(*source*, *latex*, *bicycles*)

Bases: `object`

A class for generating tables of the measurment and parameter data associated with a bicycle.

`create_rst_table`(*fileName=None*)

Returns a reStructuredText version of the table.

Parameters

fileName
[string] If a path to a file is given, the table will be written to that file.

Returns

rstTable
[string] reStructuredText version of the table.

`generate_table_data()`

Generates a list of data for a table.

`generate_variable_list()`

`to_latex(var)`

Returns a latex representation for a given variable string name.

Parameters

var
[string] One of the variable names used in the bicycleparameters package.

Returns

latex
[string] A string formatting for pretty LaTeX math print.

`uround(value)`

Returns a string representation of a value with an uncertainity which has been rounded to significant digits based on the uncertainty value.

Parameters

value
[ufloat] A single ufloat.

Returns

s
[string] A rounded string representation of *value*.

2.4563752289999+/-0.0003797273827
becomes
2.4564+/-0.0004
This probably doesn't work for weird cases like large uncertainties.

- genindex
- modindex
- search

BIBLIOGRAPHY

[Roland1971] Roland J R ., R. D., and Massing , D. E. A digital computer simulation of bicycle dynamics. Calspan Report YA-3063-K-1, Cornell Aeronautical Laboratory, Inc., Buffalo, NY, 14221, Jun 1971. Prepared for Schwinn Bicycle Company, Chicago, IL 60639.

[Meijaard2007] Meijaard, J. P.; Papadopoulos, J. M.; Ruina, A. & Schwab, A. L. Linearized dynamics equations for the balance and steer of a bicycle: A benchmark and review Proceedings of the Royal Society A: Mathematical, Physical and Engineering Sciences, 2007, 463, 1955-1982

[Kooijman2006] Kooijman, J. D. G. (2006). Experimental validation of a model for the motion of an uncontrolled bicycle. MSc thesis, Delft University of Technology.

[Kooijman2008] Kooijman, J. D. G., Schwab, A. L., and Meijaard, J. P. (2008). Experimental validation of a model of an uncontrolled bicycle. Multibody System Dynamics, 19:115–132.

[Moore2009] Moore, J. K., Kooijman, J. D. G., Hubbard, M., and Schwab, A. L. (2009). A Method for Estimating Physical Properties of a Combined Bicycle and Rider. In Proceedings of the ASME 2009 International Design Engineering Technical Conferences & Computers and Information in Engineering Conference, IDETC/CIE 2009, San Diego, CA, USA. ASME.

[Moore2010] Moore, J. K., Hubbard, M., Peterson, D. L., Schwab, A. L., and Kooijman, J. D. G. (2010). An accurate method of measuring and comparing a bicycle's physical parameters. In Bicycle and Motorcycle Dynamics: Symposium on the Dynamics and Control of Single Track Vehicles, Delft, Netherlands.

[Moore2012] Moore, J. K. (2012). Human Control of a Bicycle. University of California, Davis PhD Thesis. http://moorepants.github.io/dissertation

[Dembia2014] Dembia C, Moore JK and Hubbard M. An object oriented implementation of the Yeadon human inertia model [v1; ref status: awaiting peer review, http://f1000r.es/4cr] F1000Research 2014, 3:223 (doi: 10.12688/f1000research.5292.1)

[Meijaard2007] Meijaard J.P, Papadopoulos Jim M, Ruina Andy and Schwab A.L, 2007, Linearized dynamics equations for the balance and steer of a bicycle: a benchmark and review, Proc. R. Soc. A., 463:1955–1982 http://doi.org/10.1098/rspa.2007.1857

[Meijaard2007] Meijaard J.P, Papadopoulos Jim M, Ruina Andy and Schwab A.L, 2007, Linearized dynamics equations for the balance and steer of a bicycle: a benchmark and review, Proc. R. Soc. A., 463:1955–1982 http://doi.org/10.1098/rspa.2007.1857

[1] Moore, Jason K.; Hubbard, Mont (2019): Expanded Optimization for Discovering Optimal Lateral Handling Bicycles. Proceedings of Bicycle and Motorcycle Dynamics 2019: A Symposium on the Dynamics and Control of Single Track Vehicles https://doi.org/10.6084/m9.figshare.9942938.v1

PYTHON MODULE INDEX

b

INDEX

G

I

J

L

M

N

P

R

S

T

U

V

W

Y