DynamicistToolKit Documentation

Release 0.5.3

Jason K. Moore

Jun 22, 2023

Contents

Contents:

CHAPTER 1

dtk Package

1.1 bicycle Module

dtk.bicycle.**basu_sig_figs**()
Returns the number of significant figures reported in Table 1 of Basu-Mandal2007.

dtk.bicycle.**basu_table_one_input**()

dtk.bicycle.**basu_table_one_output**()

dtk.bicycle.**basu_to_moore_input**(*basu*, *rr*, *lam*)
Returns the coordinates and speeds of the Moore2012 derivation of the Whipple bicycle model as a function of the states and speeds of the Basu-Mandal2007 coordinates and speeds.

Parameters

basu [dictionary] A dictionary containing the states and speeds of the Basu-Mandal formulation. The states are represented with words corresponding to the greek letter and the speeds are the words with *d* appended, e.g. *psi* and *psid*.

rr [float] Rear wheel radius.

lam [float] Steer axis tilt.

Returns

moore [dictionary] A dictionary with the coordinates, q's, and speeds, u's, for the Moore formulation.

dtk.bicycle.**benchmark_matrices**()
Returns the entries to the M, C1, K0, and K2 matrices for the benchmark parameter set printed in [R03d0179118b9-Meijaard2007].

Returns

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

C1 [ndarray, shape(2,2)] The speed proportional damping matrix.

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

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

Notes

The equations of motion take this form:

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

where q = [roll angle, steer angle]

and f = [roll torque, steer torque]

References

[R03d0179118b9-Meijaard2007]

dtk.bicycle.**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 [Meijaard2007].

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.

dtk.bicycle.**benchmark_parameters**()
Returns the benchmark bicycle parameters from [Rce87380b8da1-Meijaard2007].

References

[Rce87380b8da1-Meijaard2007]

dtk.bicycle.**benchmark_state_space**(*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)] System dynamic 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]

dtk.bicycle.**benchmark_state_space_vs_speed**(*M*, *C1*, *K0*, *K2*, *speeds=None*, *v0=0.0*, *vf=10.0*, *num=50*, *g=9.81*)

Returns the state and input matrices for a set of speeds.

Parameters

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

C1 [array_like, shape(2,2)] The speed proportional damping matrix.

K0 [array_like, shape(2,2)] The gravity proportional stiffness matrix.

K2 [array_like, shape(2,2)] The speed squared proportional stiffness matrix.

speeds [array_like, shape(n,), optional] An array of speeds in meters per second at which to compute the state and input matrices. If none, the *v0*, *vf*, and *num* parameters are used to generate a linearly spaced array.

v0 [float, optional, default: 0.0] The initial speed.

vf [float, optional, default: 10.0] The final speed.

num [int, optional, default: 50] The number of speeds.

g [float, optional, default: 9.81] Acceleration due to gravity in meters per second squared.

Returns

speeds [ndarray, shape(n,)] An array of speeds in meters per second.

As [ndarray, shape(n,4,4)] The state matrices evaluated at each speed in *speeds*.

Bs [ndarray, shape(n,4,2)] The input matrices

Notes

The second order equations of motion take this form:

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

where q = [roll angle, steer angle]

and f = [roll torque, steer torque]

The first order equations of motion take this form:

x' = A * x + B * u

where x = [roll angle, steer angle, roll rate, steer rate]

and u = [roll torque, steer torque]

dtk.bicycle.**benchmark_to_moore**(*benchmarkParameters*, *oldMassCenter=False*)

Returns the parameters for the Whipple model as derived by Jason K. Moore.

Parameters

benchmarkParameters [dictionary] Contains the set of parameters for the Whipple bicycle model as presented in [Meijaard2007].

oldMassCenter [boolean] If true it returns the fork mass center dimensions, l3 and l4, with respect to the rear offset intersection with the steer axis, otherwise the dimensions are with respect to the front wheel.

Returns

mooreParameters [dictionary] The parameter set for the Moore derivation of the whipple bicycle model as presented in Moore2012.

dtk.bicycle.**front_contact**(*q1*, *q2*, *q3*, *q4*, *q7*, *d1*, *d2*, *d3*, *rr*, *rf*, *guess=None*)

Returns the location in the ground plane of the front wheel contact point.

Parameters

q1 [float] The location of the rear wheel contact point with respect to the inertial origin along the 1 axis (forward).

q2 [float] The location of the rear wheel contact point with respect to the inertial origin along the 2 axis (right).

q3 [float] The yaw angle.

q4 [float] The roll angle.

q7 [float] The steer angle.

d1 [float] The distance from the rear wheel center to the steer axis.

d2 [float] The distance between the front and rear wheel centers along the steer axis.

d3 [float] The distance from the front wheel center to the steer axis.

rr [float] The radius of the rear wheel.

rf [float] The radius of the front wheel.

guess [float, optional] A guess for the pitch angle. This may be only needed for extremely large steer and roll angles.

Returns

q9 [float] The location of the front wheel contact point with respect to the inertial origin along the 1 axis.

q10 [float] The location of the front wheel contact point with respect to the inertial origin along the 2 axis.

dtk.bicycle.**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] Front wheel radius.

rR [float] Rear wheel radius.

a [float] The rear wheel offset from the steer axis.

b [float] The front wheel offset from the steer axis.

c [float] The distance along the steer axis between the front wheel and rear wheel.

Returns

lam [float] The steer axis tilt as described in [R209a2d5d8884-Meijaard2007].

References

[R209a2d5d8884-Meijaard2007]

dtk.bicycle.**meijaard_figure_four**(*time*, *rollRate*, *steerRate*, *speed*)
Returns a figure that matches Figure #4 in [R3b3f57acabc6-Meijaard2007].

References

[R3b3f57acabc6-Meijaard2007]

dtk.bicycle.**moore_to_basu**(*moore*, *rr*, *lam*)
Returns the coordinates, speeds, and accelerations in BasuMandal2007's convention.

Parameters

moore [dictionary] A dictionary containg values for the q's, u's and u dots.

rr [float] Rear wheel radius.

lam [float] Steer axis tilt.

Returns

basu [dictionary] A dictionary containing the coordinates, speeds and accelerations.

dtk.bicycle.**pitch_from_roll_and_steer**(*q4*, *q7*, *rF*, *rR*, *d1*, *d2*, *d3*, *guess=None*)
Returns the pitch angle of the bicycle frame for a given roll, steer and geometry.

Parameters

q4 [float] Roll angle.

q5 [float] Steer angle.

rF [float] Front wheel radius.

rR [float] Rear wheel radius.

d1 [float] The rear wheel offset from the steer axis.

d2 [float] The distance along the steer axis between the intersection of the front and rear offset lines.

d3 [float] The front wheel offset from the steer axis.

guess [float, optional] A good guess for the pitch angle. If not specified, the program will make a good guess for most roll and steer combinations.

Returns

q5 [float] Pitch angle.

Notes

All of the geometry parameters should be expressed in the same units.

dtk.bicycle.**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.

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 unsing the derivative of the curves could make it more robust.

dtk.bicycle.**trail**(*rF*, *lam*, *fo*)

Returns 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

1.2 inertia Module

dtk.inertia.**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.

dtk.inertia.**cylinder_inertia**(*l*, *m*, *ro*, *ri*)

Calculate the moment of inertia for a hollow cylinder (or solid cylinder) where the x axis is aligned with the cylinder's axis.

Parameters

l [float] The length of the cylinder.

m [float] The mass of the cylinder.

ro [float] The outer radius of the cylinder.

ri [float] The inner radius of the cylinder. Set this to zero for a solid cylinder.

Returns

Ix [float] Moment of inertia about cylinder axis.

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

`dtk.inertia.`**`euler_123`**(*angles*)

Returns the direction cosine matrix as a function of the Euler 123 angles.

Parameters

angles [numpy.array or list or tuple, shape(3,)] Three angles (in units of radians) that specify the orientation of a new reference frame with respect to a fixed reference frame. The first angle, phi, is a rotation about the fixed frame's x-axis. The second angle, theta, is a rotation about the new y-axis (which is realized after the phi rotation). The third angle, psi, is a rotation about the new z-axis (which is realized after the theta rotation). Thus, all three angles are "relative" rotations with respect to the new frame. Note: if the rotations are viewed as occuring in the opposite direction (z, then y, then x), all three rotations are with respect to the initial fixed frame rather than "relative".

Returns

R [numpy.matrix, shape(3,3)] Three dimensional rotation matrix about three different orthogonal axes.

`dtk.inertia.`**`euler_rotation`**(*angles*, *order*)

Returns a rotation matrix for a reference frame, B, in another reference frame, A, where the B frame is rotated relative to the A frame via body fixed rotations (Euler angles).

Parameters

angles [array_like] An array of three angles in radians that are in order of rotation.

order [tuple] A three tuple containing a combination of `1`, `2`, and `3` where `1` is about the x axis of the first reference frame, `2` is about the y axis of the this new frame and `3` is about the z axis. Note that (1, 1, 1) is a valid entry and will give you correct results, but combinations like this are not necessarily useful for describing a general configuration.

Returns

R [numpy.matrix, shape(3,3)] A rotation matrix.

Notes

The rotation matrix is defined such that a R times a vector v equals the vector expressed in the rotated reference frame.

v' = R * v

Where v is the vector expressed in the original reference frame and v' is the same vector expressed in the rotated reference frame.

Examples

```
>>> import numpy as np
>>> from dtk.inertia import euler_rotation
>>> angles = [np.pi, np.pi / 2., -np.pi / 4.]
>>> rotMat = euler_rotation(angles, (3, 1, 3))
>>> rotMat
matrix([[ -7.07106781e-01,   1.29893408e-16,  -7.07106781e-01],
        [ -7.07106781e-01,   4.32978028e-17,   7.07106781e-01],
        [  1.22464680e-16,   1.00000000e+00,   6.12323400e-17]])
>>> v = np.matrix([[1.], [0.], [0.]])
>>> vp = rotMat * v
>>> vp
matrix([[ -7.07106781e-01],
        [ -7.07106781e-01],
        [  1.22464680e-16]])
```

dtk.inertia.**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 orthoganal 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

dtk.inertia.**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 of a body about a point located by the distances in d.

dtk.inertia.**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.

dtk.inertia.**rotate3**(*angles*)
Produces a three-dimensional rotation matrix as rotations around the three cartesian axes.

Parameters

angles [numpy.array or list or tuple, shape(3,)] Three angles (in units of radians) that specify the orientation of a new reference frame with respect to a fixed reference frame. The first angle is a pure rotation about the x-axis, the second about the y-axis, and the third about the z-axis. All rotations are with respect to the initial fixed frame, and they occur in the order x, then y, then z.

Returns

R [numpy.matrix, shape(3,3)] Three dimensional rotation matrix about three different orthogonal axes.

dtk.inertia.**rotate3_inertia**(*RotMat*, *relInertia*)
Rotates an inertia tensor. A derivation of the formula in this function can be found in Crandall 1968, Dynamics of mechanical and electromechanical systems. This function only transforms an inertia tensor for rotations with respect to a fixed point. To translate an inertia tensor, one must use the parallel axis analogue for tensors. An inertia tensor contains both moments of inertia and products of inertia for a mass in a cartesian (xyz) frame.

Parameters

RotMat [numpy.matrix, shape(3,3)] Three-dimensional rotation matrix specifying the coordinate frame that the input inertia tensor is in, with respect to a fixed coordinate system in which one desires to express the inertia tensor.

relInertia [numpy.matrix, shape(3,3)] Three-dimensional cartesian inertia tensor describing the inertia of a mass in a rotated coordinate frame.

Returns

Inertia [numpy.matrix, shape(3,3)] Inertia tensor with respect to a fixed coordinate system ("unrotated").

dtk.inertia.**rotate_inertia_about_y**(*I*, *angle*)
Returns inertia tensor rotated through angle about the Y axis.

Parameters

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

angle [float] Angle in radians about the positive Y axis of which to rotate the inertia tensor.

dtk.inertia.**torsional_pendulum_inertia**(*k*, *T*)
Calculate the moment of inertia for an ideal torsional pendulum.

Parameters

k [float] Torsional stiffness.

T [float] Period of oscillation.

Returns

I [float] Moment of inertia.

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

masses [ndarray, 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.

dtk.inertia.**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.

dtk.inertia.**x_rot**(*angle*)

Returns the rotation matrix for a reference frame rotated through an angle about the x axis.

Parameters

angle [float] The angle in radians.

Returns

Rx [np.matrix, shape(3,3)] The rotation matrix.

Notes

v' = Rx * v where v is the vector expressed the reference in the original reference frame and v' is the vector expressed in the new rotated reference frame.

dtk.inertia.**y_rot**(*angle*)

Returns the rotation matrix for a reference frame rotated through an angle about the y axis.

Parameters

angle [float] The angle in radians.

Returns

Rx [np.matrix, shape(3,3)] The rotation matrix.

Notes

v' = Rx * v where v is the vector expressed the reference in the original reference frame and v' is the vector expressed in the new rotated reference frame.

dtk.inertia.**z_rot**(*angle*)

Returns the rotation matrix for a reference frame rotated through an angle about the z axis.

Parameters

angle [float] The angle in radians.

Returns

Rx [np.matrix, shape(3,3)] The rotation matrix.

Notes

v' = Rx * v where v is the vector expressed the reference in the original reference frame and v' is the vector expressed in the new rotated reference frame.

1.3 `process` Module

dtk.process.**butterworth**(*data*, *cutoff*, *samplerate*, *order=2*, *axis=-1*, *btype='lowpass'*, ***kwargs*)
Returns the data filtered by a forward/backward Butterworth filter.

Parameters

data [ndarray, shape(n,) or shape(n,m)] The data to filter. Only handles 1D and 2D arrays.

cutoff [float] The filter cutoff frequency in hertz.

samplerate [float] The sample rate of the data in hertz.

order [int] The order of the Butterworth filter.

axis [int] The axis to filter along.

btype [{'lowpass'|'highpass'|'bandpass'|'bandstop'}] The type of filter. Default is 'lowpass'.

kwargs [keyword value pairs] Any extra arguments to get passed to scipy.signal.filtfilt.

Returns

filtered_data [ndarray] The low pass filtered version of data.

Notes

The provided cutoff frequency is corrected by a multiplicative factor to ensure the double pass filter cutoff frequency matches that of a single pass filter, see [Winter2009].

References

[Winter2009]

dtk.process.**coefficient_of_determination**(*measured*, *predicted*)
Computes the coefficient of determination with respect to a measured and predicted array.

Parameters

measured [array_like, shape(n,)] The observed or measured values.

predicted [array_like, shape(n,)] The values predicted by a model.

Returns

r_squared [float] The coefficient of determination.

Notes

The coefficient of determination [also referred to as R^2 and VAF (variance accounted for)] is computed either of these two ways:

```
      sum( [predicted - mean(measured)] ** 2 )
R^2 = ----------------------------------------
      sum( [measured - mean(measured)] ** 2 )
```

or:

```
          sum( [measured - predicted] ** 2 )
R^2 = 1 - ----------------------------------
          sum( [measured - mean(measured)] ** 2 )
```

dtk.process.**curve_area_stats**(*x*, *y*)

Return the box plot stats of a curve based on area.

Parameters

x [ndarray, shape (n,)] The x values

y [ndarray, shape (n,m)] The y values n are the time steps m are the various curves

Returns

A dictionary containing:

median [ndarray, shape (m,)] The x value corresponding to 0.5*area under the curve

lq [ndarray, shape (m,)] lower quartile

uq [ndarray, shape (m,)] upper quartile

98p [ndarray, shape (m,)] 98th percentile

2p [ndarray, shape (m,)] 2nd percentile

dtk.process.**derivative**(*x*, *y*, *method='forward'*, *padding=None*)

Returns the derivative of y with respect to x.

Parameters

x [ndarray, shape(n,)] The monotonically increasing independent variable.

y [ndarray, shape(n,) or shape(n, m)] The dependent variable(s).

method [string, optional]

'forward' Use the forward difference method.

'backward' Use the backward difference method.

'central' Use the central difference method.

'combination' This is equivalent to `method='central', padding='second order'` and is in place for backwards compatibility. Selecting this method will ignore and user supplied padding settings.

padding [None, float, 'adjacent' or 'second order', optional] The default, None, will result in the derivative vector being n-a in length where a=1 for forward and backward and a=2 for central. If you provide a float this value will be used to pad the result so that len(dydx) == n. If 'adjacent' is used, the nearest neighbor will be used for padding. If 'second order' is chosen second order foward and backward difference are used to pad the end points.

Returns

dydx [ndarray, shape(n,) or shape(n-1,)] for combination else shape(n-1,)

dtk.process.**find_timeshift**(*signal1*, *signal2*, *sample_rate*, *guess=None*, *plot=False*)
Returns the timeshift, tau, of the second signal relative to the first signal.

Parameters

signal1 [array_like, shape(n,)] The base signal.

signal2 [array_like, shape(n,)] A signal shifted relative to the first signal. The second signal should be leading the first signal.

sample_rate [integer or float] Sample rate of the signals. This should be the same for each signal.

guess [float, optional, default=None] If you've got a good guess for the time shift then supply it here.

plot [boolean, optional, defaul=False] If true, a plot of the error landscape will be shown.

Returns

tau [float] The timeshift between the two signals.

dtk.process.**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.

Notes

SST = SSR + SSE

dtk.process.**freq_spectrum**(*data*, *sampleRate*)
Return the frequency spectrum of a data set.

Parameters

data [ndarray, shape (m,) or shape(n,m)] The array of time signals where n is the number of variables and m is the number of time steps.

sampleRate [int] The signal sampling rate in hertz.

Returns

frequency [ndarray, shape (p,)] The frequencies where p is a power of 2 close to m.

amplitude [ndarray, shape (p,n)] The amplitude at each frequency.

dtk.process.**least_squares_variance**(*A*, *sum_of_residuals*)
Returns the variance in the ordinary least squares fit and the covariance matrix of the estimated parameters.

Parameters

A [ndarray, shape(n,d)] The left hand side matrix in Ax=B.

sum_of_residuals [float] The sum of the residuals (residual sum of squares).

Returns

variance [float] The variance of the fit.

covariance [ndarray, shape(d,d)] The covariance of x in Ax = b.

dtk.process.**normalize**(*sig*, *hasNans=False*)
Normalizes the vector with respect to the maximum value.

Parameters

sig [ndarray, shape(n,)]

hasNans [boolean, optional] If your data has nans use this flag if you want to ignore them.

Returns

normSig [ndarray, shape(n,)] The signal normalized with respect to the maximum value.

dtk.process.**spline_over_nan**(*x*, *y*)
Returns a vector of which a cubic spline is used to fill in gaps in the data from nan values.

Parameters

x [ndarray, shape(n,)] This x values should not contain nans.

y [ndarray, shape(n,)] The y values may contain nans.

Returns

ySpline [ndarray, shape(n,)] The splined y values. If *y* doesn't contain any nans then *ySpline* is *y*.

Notes

The splined data is identical to the input data, except that the nan's are replaced by new data from the spline fit.

dtk.process.**subtract_mean**(*sig*, *hasNans=False*)
Subtracts the mean from a signal with nanmean.

Parameters

sig [ndarray, shape(n,)]

hasNans [boolean, optional] If your data has nans use this flag if you want to ignore them.

Returns

ndarray, shape(n,) sig minus the mean of sig

dtk.process.**sync_error**(*tau*, *signal1*, *signal2*, *time*, *plot=False*)
Returns the error between two signal time histories given a time shift, tau.

Parameters

tau [float] The time shift.

signal1 [ndarray, shape(n,)] The signal that will be interpolated. This signal is typically "cleaner" that signal2 and/or has a higher sample rate.

signal2 [ndarray, shape(n,)] The signal that will be shifted to syncronize with signal 1.

time [ndarray, shape(n,)] The time vector for the two signals

plot [boolean, optional, default=False] If true a plot will be shown of the resulting signals.

Returns

error [float] Error between the two signals for the given tau.

dtk.process.**time_vector**(*num_samples*, *sample_rate*, *start_time=0.0*)

Returns a time vector starting at zero.

Parameters

num_samples [int] Total number of samples.

sample_rate [float] Sample rate of the signal in hertz.

start_time [float, optional, default=0.0] The start time of the time series.

Returns

time [ndarray, shape(numSamples,)] Time vector starting at zero.

dtk.process.**truncate_data**(*tau*, *signal1*, *signal2*, *sample_rate*)

Returns the truncated vectors with respect to the time shift tau. It assume you've found the time shift between two signals with find_time_shift or something similar.

Parameters

tau [float] The time shift.

signal1 [array_like, shape(n,)] A time series.

signal2 [array_like, shape(n,)] A time series.

sample_rate [integer] The sample rate of the two signals.

Returns

truncated1 [ndarray, shape(m,)] The truncated time series.

truncated2 [ndarray, shape(m,)] The truncated time series.

CHAPTER 2

References

CHAPTER 3

Introduction

This is a collection of Python modules which contain tools that are helpful for a dynamicist. Right now it is basically a place I place general tools that don't necessarily need a distribution of their own.

CHAPTER 4

Modules

bicycle Generic tools for basic bicycle dynamics analysis.

inertia Various functions for calculating and manipulating inertial quantities.

process Various tools for common signal processing tasks.

CHAPTER 5

Installation

You will need Python 2.7 or 3.3+ and setuptools to install the packages. Its best to install the dependencies first (NumPy, SciPy, matplotlib, Pandas). The SciPy Stack instructions are helpful for this: http://www.scipy.org/stackspec.html.

We recommend installing with conda so that dependency installation is not an issue:

```
$ conda install -c moorepants dynamicisttoolkit
```

You can install using pip. Pip will theoretically[1] get the dependencies for you (or at least check if you have them):

```
$ pip install DynamicistToolKit
```

Or download the source with your preferred method and install manually.

Using Git:

```
$ git clone git@github.com:moorepants/DynamicistToolKit.git
$ cd DynamicistToolKit
```

Or wget:

```
$ wget https://github.com/moorepants/DynamicistToolKit/archive/master.zip
$ unzip master.zip
$ cd DynamicistToolKit-master
```

Then for basic installation:

```
$ python setup.py install
```

Or install for development purposes:

```
$ python setup.py develop
```

[1] You will need all build dependencies and also note that matplotlib doesn't play nice with pip.

CHAPTER 6

Tests

Run the tests with nose:

```
$ nosetests
```

CHAPTER 7

Vagrant

A vagrant file and provisioning script are included to test the code on an Ubuntu 13.10 box. To load the box and run the tests simply type:

```
$ vagrant up
```

See `bootstrap.sh` and `VagrantFile` to see what's going on.

CHAPTER 8

Documentation

The documentation is hosted at ReadTheDocs:

http://dynamicisttoolkit.readthedocs.org

You can build the documentation (currently sparse) if you have Sphinx and numpydoc:

```
$ cd docs
$ make html
$ firefox _build/html/index.html
```

CHAPTER 9

Release Notes

9.1 0.5.3

- Added the license and readme to the source distriubtion.

9.2 0.5.2

- Screwed up pypi upload on 0.5.1, so bumping one more time.

9.3 0.5.1

- Import nanmean from numpy instead of scipy and fix float slices. [PR #34]

9.4 0.5.0

- bicycle.py functions now output numpy arrays instead of matrices.
- Support for Python 3 [PR #30 and #32].

9.5 0.4.0

- Made the numerical derivative function more robust and featureful. [PR #27]
- `butterworth` now uses a corrected cutoff frequency to adjust for the double filtering. [PR #28]

9.6 0.3.5

- Fixed bug in coefficient_of_determination. [PR #23]

9.7 0.3.4

- Fixed bug in normalized cutoff frequency calculation. [PR #21]

9.8 0.3.2

- Fixed bug in butterworth function and added tests.

9.9 0.3.1

- Fixed butterworth to work with SciPy 0.9.0. [PR #18]

9.10 0.3.0

- Removed pandas dependency.
- Improved time vector function.
- Removed gait analysis code (walk.py), now at http://github.com/csu-hmc/Gait-Analysis-Toolkit.
- TravisCI tests now run, added image to readme.
- Added documentation at ReadTheDocs.

9.11 0.2.0

- Addition of walking dynamics module.

9.12 0.1.0

- Original code base that was used for the computations in this dissertation: https://github.com/moorepants/dissertation

CHAPTER 10

Indices and tables

- genindex
- modindex
- search

Bibliography

[Winter2009] David A. Winter (2009) Biomechanics and motor control of human movement. 4th edition. Hoboken: Wiley.

[Basu-Mandal2007] Basu-Mandal, P.; Chatterjee, A. & Papadopoulos, J. M. Hands-free circular motions of a benchmark bicycle. Proceedings of the Royal Society A: Mathematical, Physical and Engineering Sciences, 2007, 463, 1983-2003

[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

[Moore2012] Moore, J. K. Human Control of a Bicycle. PhD Dissertation. University of California, Davis, 2012

Python Module Index

d

Index

T

X

Y

Z