pytrajectory Documentation

Release 0.3.2

Andreas Kunze

April 16, 2014

CONTENTS

CHAPTER

ONE

PYTRAJECTORY USER'S GUIDE

1.1 About PyTrajectory

PyTrajectory is being developed at Dresden University of Technology at the Institute for Control Theory. Based upon a study work of Oliver Schnabel under the supervision of Carsten Knoll in February 2013 it has been further developed by Andreas Kunze to increase its numeric performance.

1.2 Getting Started

This section provides an overview on what PyTrajectory is and how to use it. For a more detailed view please have a look at the *PyTrajectory Modules Reference*.

1.2.1 What is PyTrajectory?

PyTrajectory is a Python library for the determination of the feed forward control to achieve a transition between desired states of a nonlinear control system.

Planning and designing of trajectories represents an important task in the control of technological processes. Here the problem is attributed on a multi-dimensional boundary value problem with free parameters. In general this problem can not be solved analytically. It is therefore resorted to the method of collocation in order to obtain a numerical approximation.

PyTrajectory allows a flexible implementation of various tasks.

1.2.2 Installation

PyTrajectory was developed and tested on Python 2.7

Dependencies

- numpy
- sympy
- scipy
- optional
  - matplotlib
  - ipython [debugging]

Windows

... to do

Linux

... to do

Mac OS

... to do

1.2.3 Usage

... to do

1.2.4 A First Example

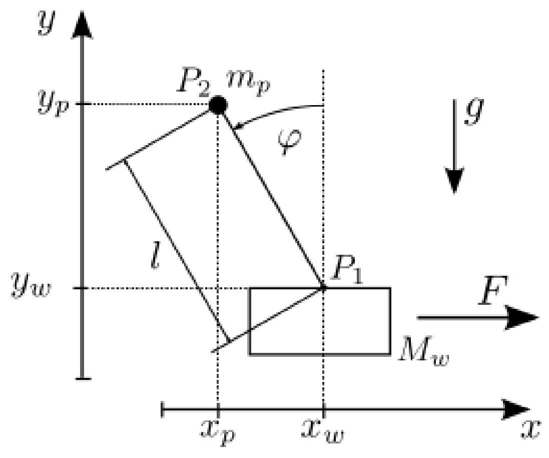
... to do

1.3 Examples

The following example systems from mechanics demonstrate the application of PyTrajectory. The deriving of the model equations is omittted here.

1.3.1 Translation of the inverted pendulum

An example often used in literature is the inverted pendulum. Here a force F acts on a cart with mass M_w. In addition the cart is connected by a massless rod with a pendulum mass m_p. The mass of the pendulum is concentrated in P_2 and that of the cart in P_1. The state vector of the system can be specified using the carts position $x_w(t)$ and the pendulum deflection $\varphi(t)$ and their derivatives.



With the *Lagrangian Formalism* the model has the following state representation where $u_1 = F$ and $x = [x_1, x_2, x_3, x_4] = [x_w, \dot{x}_w, \varphi, \dot{\varphi}]$

$$\begin{aligned}
\dot{x}_1 &= x_2 \\
\dot{x}_2 &= \frac{m_p \sin(x_3)(-l x_4^2 + g \cos x_3)}{M_w l + m_p \sin^2(x_3)} + \frac{\cos(x_3)}{M_w l + m_p l \sin^2(x_3)} u_1 \\
\dot{x}_3 &= x_4 \\
\dot{x}_4 &= \frac{\sin(x_3)(-m_p l x_4^2 \cos(x_3) + g(M_w + m_p))}{M_w l + m_p \sin^2(x_3)} + \frac{\cos(x_3)}{M_w l + m_p l \sin^2(x_3)} u_1
\end{aligned}$$

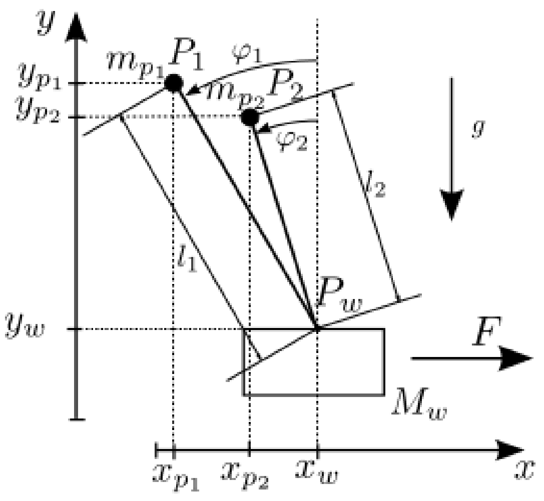
A possibly wanted trajectory is the translation of the cart along the x-axis (i.e. by $0.5m$). In the beginning and end of the process the cart and pendulum should remain at rest and the pendulum should be aligned vertically upwards ($\varphi = 0$). As a further condition u_1 should start and end steadily in the rest position ($u_1(0) = u_1(T) = 0$). The operating time here is $T = 1[s]$.

```python
# translation of the inverse pendulum

# import trajectory class and necessary dependencies
from pytrajectory import Trajectory
from sympy import sin, cos
import numpy as np

# define the function that returns the vectorfield
def f(x,u):
    x1, x2, x3, x4 = x       # system state variables
    u1, = u                  # input variable

    l = 0.5     # length of the pendulum rod
    g = 9.81    # gravitational acceleration
    M = 1.0     # mass of the cart
    m = 0.1     # mass of the pendulum

    s = sin(x3)
    c = cos(x3)

    ff = np.array([                     x2,
                    m*s*(-l*x4**2+g*c)/(M+m*s**2)+1/(M+m*s**2)*u1,
                                        x4,
            s*(-m*l*x4**2*c+g*(M+m))/(M*l+m*l*s**2)+c/(M*l+l*m*s**2)*u1
                ])
    return ff

# boundary values at the start (a = 0.0 [s])
xa = [  0.0,
        0.0,
        0.0,
        0.0]

# boundary values at the end (b = 1.0 [s])
xb = [  0.5,
        0.0,
        0.0,
        0.0]

# create trajectory object
T = Trajectory(f, a=0.0, b=1.0, xa=xa, xb=xb)

# run iteration
```

```
T.startIteration()

# show results
T.plot()
```

1.3.2 Oscillation of the inverted double pendulum

In this example we add another pendulum to the cart in the system.



The system has the state vector $x = [x_1, \dot{x}_1, \varphi_1, \dot{\varphi}_1, \varphi_2, \dot{\varphi}_2]$. A partial linearization with $y = x_1$ yields the following system state representation where $\tilde{u} = \ddot{y}$.

$$\begin{aligned}
\dot{x}_1 &= x_2 \\
\dot{x}_2 &= \tilde{u} \\
\dot{x}_3 &= x_4 \\
\dot{x}_4 &= \frac{1}{l_1}(g\sin(x_3) + \tilde{u}\cos(x_3)) \\
\dot{x}_5 &= x_6 \\
\dot{x}_6 &= \frac{1}{l_2}(g\sin(x_5) + \tilde{u}\cos(x_5))
\end{aligned}$$

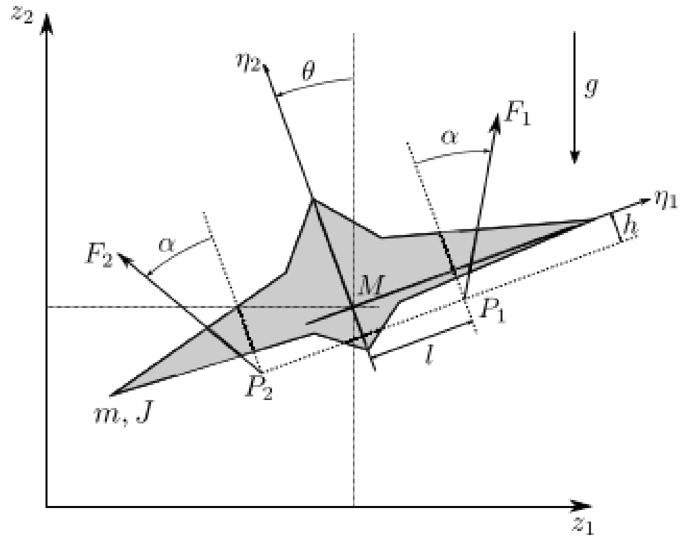
Here a trajectory should be planned that transfers the system between the following two positions of rest. At the beginning both pendulums should be directed downwards ($\varphi_1 = \varphi_2 = \pi$). After a operating time of $T = 2[s]$ the cart should be at the same position again and the pendulums should be at rest with $\varphi_1 = \varphi_2 = 0$.

$$x(0) = \begin{bmatrix} 0 \\ 0 \\ \pi \\ 0 \\ \pi \\ 0 \end{bmatrix} \rightarrow x(T) = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

```
# oscillation of the inverted double pendulum with partial linearization

# import trajectory class and necessary dependencies
from pytrajectory.trajectory import Trajectory
from sympy import cos, sin
import numpy as np
from numpy import pi

# define the function that returns the vectorfield
def f(x,u):
        x1, x2, x3, x4, x5, x6 = x  # system variables
        u, = u                      # input variable

    # length of the pendulums
        l1 = 0.7
        l2 = 0.5

        g = 9.81    # gravitational acceleration

        ff = np.array([         x2,
                            u,
                            x4,
                (1/l1)*(g*sin(x3)+u*cos(x3)),
                            x6,
                (1/l2)*(g*sin(x5)+u*cos(x5))
                    ])

        return ff

# system state boundary values for a = 0.0 [s] and b = 2.0 [s]
xa = [0.0, 0.0,  pi, 0.0,  pi, 0.0]
xb = [0.0, 0.0, 0.0, 0.0, 0.0, 0.0]

# boundary values for the input
g= [0.0, 0.0]

# create trajectory object
T = Trajectory(f, a=0.0, b=2.0, xa=xa, xb=xb, g=g)

# alter some method parameters to increase performance
T.setParam('su', 10)
T.setParam('eps', 8e-2)

# run iteration
T.startIteration()

# show results
T.plot()
```

1.3.3 Aircraft

In this section we consider the model of a unmanned vertical take-off aircraft. The aircraft has two permanently mounted thrusters on the wings which can apply the thrust forces F_1 and F_2 independently of each other. The two engines are inclined by an angle α with respect to the aircraft-fixed axis η_2 and engage in the points $P_1 = (l, h)$ and $P_2 = (-l, -h)$. The coordinates of the center of mass M of the aircraft in the inertial system are denoted by z_1 and z_2. At the same time, the point is the origin of the plane coordinate system. The aircraft axes are rotated by the angle θ with respect to the z_2-axis.



Through the establishment of the momentum balances for the model one obtains the equations

$$\begin{aligned} m\ddot{z}_1 &= -\sin(\theta)(F_1+F_2)\cos(\alpha)+\cos(\theta)(F_1-F_2)\sin(\alpha) \\ m\ddot{z}_2 &= \cos(\theta)(F_1+F_2)\sin(\alpha)+\sin(\theta)(F_1-F_2)\cos(\alpha)-mg \\ J\ddot{\theta} &= (F_1-F_2)(l\cos(\alpha)+h\sin(\alpha)) \end{aligned}$$

With the state vector $x = [z_1, \dot{z}_1, z_2, \dot{z}_2, \theta, \dot{\theta}]^T$ and $u = [u_1, u_2]^T = [F_1, F_2]^T$ the state space representation of the system is as follows.

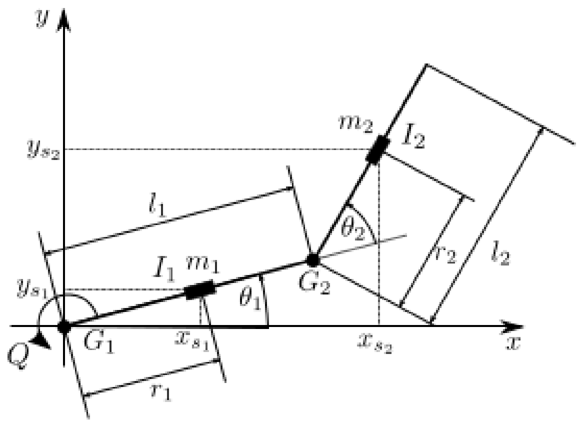
$$\begin{aligned} \dot{x}_1 &= x_2 \\ \dot{x}_2 &= \frac{1}{m}(-\sin(x_5)(u_1+u_2)\cos(\alpha)+\cos(x_5)(u_1-u_2)\sin(\alpha)) \\ \dot{x}_3 &= x_4 \\ \dot{x}_2 &= \frac{1}{m}(\cos(x_5)(u_1+u_2)\cos(\alpha)+\sin(x_5)(u_1-u_2)\sin(\alpha))-g \\ \dot{x}_5 &= x_6 \\ \dot{x}_6 &= \frac{1}{J}(l\cos(\alpha)+h\sin(\alpha)) \end{aligned}$$

For the aircraft, a trajectory should be planned that translates the horizontally aligned flying object from a rest position (hovering) along the z_1 and z_2 axis back into a hovering position. The hovering is to be realized on the boundary conditions of the input. Therefor the derivatives of the state variables should satisfy the following conditions. $\dot{z}_1 = \ddot{z}_1 = \dot{z}_2 = \ddot{z}_2 = \dot{\theta} = \ddot{\theta} = 0$ For the horizontal position applies $\theta = 0$. These demands yield the boundary conditions for the inputs. $F_1(0) = F_1(T) = F_2(0) = F_2(T) = \frac{mg}{2\cos(\alpha)}$

```
# vertical take-off aircraft

# import trajectory class and necessary dependencies
from pytrajectory import Trajectory
from sympy import sin, cos
import numpy as np
from numpy import pi

# define the function that returns the vectorfield
def f(x,u):
    x1, x2, x3, x4, x5, x6 = x  # system state variables
    u1, u2 = u                  # input variables
    # coordinates for the points in which the engines engage [m]
    l = 1.0
    h = 0.1

    g = 9.81    # graviational acceleration [m/s^2]
    M = 50.0    # mass of the aircraft [kg]
    J = 25.0    # moment of inertia about M [kg*m^2]

    alpha = 5/360.0*2*pi    # deflection of the engines

    sa = sin(alpha)
    ca = cos(alpha)

    s = sin(x5)
    c = cos(x5)

    ff = np.array([             x2,
                    -s/M*(u1+u2) + c/M*(u1-u2)*sa,
                                x4,
                    -g+c/M*(u1+u2) +s/M*(u1-u2)*sa ,
                                x6,
                    1/J*(u1-u2)*(l*ca+h*sa)])

    return ff

# system state boundary values for a = 0.0 [s] and b = 3.0 [s]
xa = [ 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]
xb = [10.0, 0.0, 5.0, 0.0, 0.0, 0.0]

# boundary values for the inputs
g = [0.5*9.81*50.0/(cos(5/360.0*2*pi)),
     0.5*9.81*50.0/(cos(5/360.0*2*pi))]

# create trajectory object
T = Trajectory(f, a=0.0, b=3.0, xa=xa, xb=xb, g=g)

# don't take advantage of the system structure (integrator chains)
# (this will result in a faster solution here)
T.setParam('use_chains', False)

# also alter some other method parameters to increase performance
T.setParam('kx', 5)

# run iteration
T.startIteration()

# show results
T.plot()
```

1.3.4 Underactuated Manipulator

In this section, the model of an underactuated manipulator is treated. The system consists of two bars with the mass M_1 and M_2 which are connected to each other via the joint G_2. The angle between them is designated by θ_2. The joint G_1 connects the first rod with the inertial system, the angle to the x-axis is labeled θ_1. In the joint G_1 the actuating torque Q is applied. The bars have the moments of inertia I_1 and I_2. The distances between the centers of mass to the joints are r_1 and r_2.



The modeling was taken from the thesis of Carsten Knoll (April, 2009) where in addition the inertia parameter η was introduced.

$$\eta = \frac{m_2 l_1 r_2}{I_2 + m_2 r_2^2}$$

For the example shown here, strong inertia coupling was assumed with $\eta = 0.9$. By partial linearization to the output $y = \theta_1$ one obtains the state representation with the states $x = [\theta_1, \dot{\theta}_1, \theta_2, \dot{\theta}_2]^T$ and the new input $\tilde{u} = \ddot{\theta}_1$.

$$\begin{aligned}
\dot{x}_1 &= x_2 \\
\dot{x}_2 &= \tilde{u} \\
\dot{x}_3 &= x_4 \\
\dot{x}_4 &= -\eta x_2^2 \sin(x_3) - (1 + \eta\cos(x_3))\tilde{u}
\end{aligned}$$

For the system, a trajectory is to be determined for the transfer between two equilibrium positions within an operating time of $T = 1.8[s]$.

$$x(0) = \begin{bmatrix} 0 \\ 0 \\ 0.4\pi \\ 0 \end{bmatrix} \rightarrow x(T) = \begin{bmatrix} 0.2\pi \\ 0 \\ 0.2\pi \\ 0 \end{bmatrix}$$

The trajectory of the inputs should be without cracks in the transition to the equilibrium positions ($\tilde{u}(0) = \tilde{u}(T) = 0$).

```
# underactuated manipulator

# import trajectory class and necessary dependencies
from pytrajectory.trajectory import Trajectory
import numpy as np
from sympy import cos, sin
from numpy import pi

# define the function that returns the vectorfield
def f(x,u):
    x1, x2, x3, x4  = x     # state variables
    u1, = u                 # input variable

    e = 0.9     # inertia coupling

    s = sin(x3)
    c = cos(x3)

    ff = np.array([         x2,
                            u1,
                            x4,
                    -e*x2**2*s-(1+e*c)*u1
                    ])

    return ff

# system state boundary values for a = 0.0 [s] and b = 1.8 [s]
xa = [  0.0,
        0.0,
        0.4*pi,
        0.0]

xb = [  0.2*pi,
        0.0,
        0.2*pi,
        0.0]

# boundary values for the inputs
g = [0.0, 0.0]

# create trajectory object
T = Trajectory(f, a=0.0, b=1.8, xa=xa, xb=xb, g=g)

# also alter some method parameters to increase performance
T.setParam('su', 20)
T.setParam('kx', 3)

# run iteration
T.startIteration()

# show results
T.plot()
```

1.4 Background

This section is intended to give some insights into the mathematical background that is the basis of PyTrajectory.

1.4.1 Collocation method

Given a system of autonomous differential equations

$$\begin{aligned}\dot{x}_1(t) &= f_n(x_1(t),...,x_n(t)) \\ \vdots \quad & \quad \vdots \\ \dot{x}_n(t) &= f_n(x_1(t),...,x_n(t))\end{aligned}$$

with $t \in [a,b]$ and *Dirichlet* boundary conditions

$$x_i(a) = \alpha_i, \quad x_i(b) = \beta_i \qquad i = 1,...,n$$

the collocation method to solve the problem basically works as follows.

We choose $N+1$ collocation points $t_i,\ i = 0,...,N$ from the interval $[a,b]$ where $t_0 = a,\ t_N = b$ and search for functions $P_i : [a, b] \rightarrow R$ which satisfy the following conditions:

$$P_i(t_0) = \alpha_i, \qquad P_i(t_N) = \beta_i$$
$$\frac{d}{dt}P_i(t) = f_i(P_1(t), ..., P_n(t)) \quad i = 1, ..., n$$

Through these demands the exact solution of the ode system will be approximated.

1.4.2 Candidate functions

PyTrajectory uses cubic spline functions as candidates for the approximation of the solution. Splines are piecewise polynomials with a global differentiability. The connection points τ_i between the polynomial sections are equidistantly and are referred to as nodes.

$$t_0 = \tau_0 < \tau_1 < ... < \tau_\eta = t_N \qquad h = \frac{t_0 - t_N}{\eta}$$
$$\tau_{i+1} = \tau_i + h \quad i = 0, ..., \eta - 1$$

The polynomial sections can be created as follows.

$$P_i(t) = c_{i,0}(t - ih)^3 + c_{i,1}(t - ih)^2 + c_{i,2}(t - ih) + c_{i,3}$$
$$c_{i,j} \in R, \qquad i = 1, ..., \eta, \ j = 0, ..., 3$$

In addition to the steadiness the spline functions should be twice steadily differentiable in the nodes τ.

1.4.3 Equation system

... to do

CHAPTER

TWO

PYTRAJECTORY MODULES REFERENCE

2.1 PyTrajectory

PyTrajectory is a Python library for the determination of the feed forward control to achieve a transition between desired states of a nonlinear control system.

2.2 `trajectory` Module

class `pytrajectory.trajectory.`**`Trajectory`**(*ff*, *a=0.0*, *b=1.0*, *xa=None*, *xb=None*, *g=None*, *sx=5*, *su=5*, *kx=2*, *delta=2*, *maxIt=10*, *eps=0.01*, *tol=1e-05*, *algo='leven'*, *use_chains=True*)

Base class of the PyTrajectory project.

Parameters

- **ff** (*callable*) – Vectorfield (rhs) of the control system
- **a** (*float*) – Left border
- **b** (*float*) – Right border
- **xa** (*list*) – Boundary values at the left border
- **xb** (*list*) – Boundary values at the right border
- **g** (*list*) – Boundary values of the input variables
- **sx** (*int*) – Initial number of spline parts for the system variables
- **su** (*int*) – Initial number of spline parts for the input variables
- **kx** (*int*) – Factor for raising the number of spline parts for the system variables
- **delta** (*int*) – Constant for calculation of collocation points
- **maxIt** (*int*) – Maximum number of iterations
- **eps** (*float*) – Tolerance for the solution of the initial value problem
- **tol** (*float*) – Tolerance for the solver of the equation system
- **algo** (*str*) – Solver to use
- **use_chains** (*bool*) – Whether or not to use integrator chains

DG(*c*)

Returns the Jacobian matrix of the collocation system w.r.t. the independent parameters.

G(*c*)
Returns the collocation system evaluated with numeric values for the independent parameters.

analyseSystem()
Analyses the systems structure and sets values for some of the method parameters.

By now, this method determines the number of state and input variables and searches for integrator chains.

buildEQS()
Builds the collocation equation system.

checkAccuracy()
Checks whether desired accuracy for the boundary values was reached.

clear()

dx(*t*)
This function returns the left hand sites state at a given (time-) point `t`.

getGuess()
This method is used to determine a starting value (guess) for the solver of the collocation equation system.

If it is the first iteration step, then a vector with the same length as the vector of independent parameters with arbitrarily values is returned.

Else, for every variable a spline has been created for, the old spline of the iteration before and the new spline are evaluated at specific points and a equation system is solved which ensures that they are equal in these points.

The solution of this equation system is the new start value for the solver.

initSplines()
This method is used to initialise the provisionally splines.

iterate()
This method is used to run one iteration step.

First, new splines are initialised for the variables that are the upper end of an integrator chain. TODO: ...

plot()
Plot the calculated trajectories and error functions.

This method just calls `plot()` function from `utilities`

save()
Save system data, callable solution functions and simulation results.

setCoeff()
Set found numerical values for the independent parameters od each spline.

This method is used to get the actual splines by using the numerical solutions to set up the coefficients of the polynomial spline parts of every created spline.

setParam(*param=''*, *val=None*)
Method to assign value `val`to method parameter :attr:`param`.

Parameters

- **param** (*str*) – Parameter of which to alter the value
- **val** – New value for the passed parameter

simulate()
This method is used to solve the initial value problem.

solve()
This method is used to solve the collocation equation system.

startIteration()
This is the main loop.

At first the equations that have to be solved by collocation will be determined, according to the integrator chains. Next, one step of the iteration is done by calling `iterate()`. After that the accuracy of the found solution is checked. If it is within the tolerance range the iteration will stop. Else, the number of spline parts is raised and another iteration step starts.

u(*t*)
This function returns the inputs state at a given (time-) point `t`.

x(*t*)
This function returns the system state at a given (time-) point `t`.

2.3 `spline` Module

class `pytrajectory.spline.`**CubicSpline**(*a=0.0*, *b=1.0*, *n=10*, *tag=''*, *bc=None*, *bcd=None*, *bcdd=None*, *steady=True*)
This class provides an object that represents a cubic spline ...

Parameters

- **a** (*float*) – Left border of the spline interval.
- **b** (*float*) – Right border of the spline interval.
- **n** (*int*) – Number of polynomial parts the spline will be devided into.
- **tag** (*str*) – The 'name' of the spline object.
- **bc** (*tuple*) – Boundary values for the spline function itself.
- **bcd** (*tuple*) – Boundary values for the splines 1st derivative
- **bcdd** (*tuple*) – Boundary values for the splines 2nd derivative
- **steady** (*bool*) – Whether or not to call `makesteady()` when instanciated.

dddf(*x*)
This is just a wrapper for `evalf()` to evaluate the splines 3rd derivative.

ddf(*x*)
This is just a wrapper for `evalf()` to evaluate the splines 2nd derivative.

df(*x*)
This is just a wrapper for `evalf()` to evaluate the splines 1st derivative.

evalf(*x*, *d*)
Returns the value of the splines `d`-th derivative at `x`.

Parameters

- **x** (*float*) – The point to evaluate the spline at
- **d** (*int*) – The derivation order

f(*x*)
This is just a wrapper for `evalf()` to evaluate the spline itself.

makesteady()
This method sets up and solves equations that satisfy boundary conditions and ensure steadiness and smoothness conditions of the spline in every joining point.

prov_evalf(*x*, *d*)
This function returns a matrix and vector to evaluate the spline or a derivative at x by multiplying the matrix with numerical values of the independent variables and adding the vector.

Parameters

- **x** (*real*) – The point to evaluate the spline at
- **d** (*int*) – The derivation order

Returns Matrix and vector that represent how the splines coefficients depend on the free parameters.

Return type tuple

set_coeffs(*c_sol*)
This function is used to set up numerical values for the spline coefficients.

Parameters **c_sol** (*numpy.ndarray*) – Array with numerical values for the free spline coefficients

pytrajectory.spline.**fdiff**(*func*)
This function is used to get the derivative of of a callable splinefunction.

2.4 `solver` Module

class pytrajectory.solver.**Solver**(*F*, *DF*, *x0*, *tol=0.01*, *maxx=10*, *algo='leven'*)
This class provides solver for the collocation equation system.

Parameters

- **F** (*callable*) – The callable function that represents the equation system
- **DF** (*callable*) – The function for the jacobian matrix of the eqs
- **x0** (*numpy.ndarray*) – The start value for the sover
- **tol** (*float*) – The (absolute) tolerance of the solver
- **maxx** (*int*) – The maximum number of iterations of the solver
- **algo** (*str*) – The solver to use

gauss()

leven()
This method is an implementation of the Levenberg-Marquardt-Method to approximatively solve a system of non-linear equations by minimizing

$$\|F'(x_k)(x_{k+1} - x_k) + F(x_k)\|_2^2 + \mu^2\|x_{k+1} - x_k\|$$

newton()

solve()
This is just a wrapper to call the chosen algorithm for solving the equation system

2.5 `simulation` Module

class `pytrajectory.simulation.`**`Simulation`** (*ff*, *T*, *start*, *u*, *dt=0.01*)

This class does something ...

Parameters

- **ff** (*callable*) – Vectorfield of the control system
- **T** (*float*) – Simulation time
- **u** (*callable*) – Function of the input variables
- **dt** (*float*) – Time step

`calcStep` ()

`rhs` (*t*, *x*)

`simulate` ()

2.6 `utilities` Module

class `pytrajectory.utilities.`**`Animation`** (*drawfnc*, *simdata*, *plotsys*=[ ], *plotinputs*=[ ])

Provides animation capabilities.

Parameters

- **drawfnc** (*callable*) – Function that returns an image of the current system state according to `simdata`
- **simdata** (*numpy.ndarray*) – Array that contains simulation data (time, system states, input states)
- **plotsys** (*list*) – List of tuples with indices and labels of system variables that will be plotted along the picture
- **plotinputs** (*list*) – List of tuples with indices and labels of input variables that will be plotted along the picture

class **`Image`**

`reset` ()

`Animation.`**`animate`** ()

`Animation.`**`get_axes`** ()

`Animation.`**`save`** (*fname*, *fps=None*, *dpi=200*)

`Animation.`**`set_label`** (*ax='ax_img'*, *label=''*)

`Animation.`**`set_limits`** (*ax='ax_img'*, *xlim=(0, 1)*, *ylim=(0, 1)*)

class `pytrajectory.utilities.`**`IntegChain`** (*lst*)

This class provides a representation of a integrator chain consisting of sympy symbols.

For the elements $(x_i)_{i=1,...,n}$ the following relation applies:

$$\dot{x}_i = x_{i+1} \quad i = 1, ..., n-1$$

Parameters **lst** (*list*) – Ordered list of elements for the integrator chain

pred(*elem*)

This method returns the predecessor of the given element of the integrator chains, i.e. it returns $\int[elem]$

Parameters **elem** (*sympy.Symbol*) – An element of the integrator chain

succ(*elem*)

This method returns the successor of the given element of the integrator chains, i.e. it returns $\frac{d}{dt}[elem]$

Parameters **elem** (*sympy.Symbol*) – An element of the integrator chain

pytrajectory.utilities.**blockdiag**(*M*, *bshape=None*, *sparse=False*)

Takes blocks of shape `bshape` from matrix `M` and creates block diagonal matrix out of them.

Parameters

- **M** (*numpy.ndarray*) – Matrix to take blocks from
- **bshape** (*tuple*) – Shape of one block
- **sparse** (*bool*) – Whether or not to return created matrix as sparse matrix

Examples

```
>>> A = np.ones((4, 2))
>>> print A
[[ 1.  1.]
 [ 1.  1.]
 [ 1.  1.]
 [ 1.  1.]]
>>> B = blockdiag(A, (2, 2))
>>> print B
[[ 1.  1.  0.  0.]
 [ 1.  1.  0.  0.]
 [ 0.  0.  1.  1.]
 [ 0.  0.  1.  1.]]
```

pytrajectory.utilities.**plot**(*sim*, *H*, *fname=None*)

This method provides graphics for each system variable, manipulated variable and error function and plots the solution of the simulation.

Parameters

- **sim** (*tuple*) – Contains collocation points, and simulation results of system and input variables
- **H** (*dict*) – Dictionary of the callable error functions
- **fname** (*str*) – If not None, plot will be saved as <fname>.png

2.7 `log` Module

pytrajectory.log.**IPS**(*loc=None*)

class pytrajectory.log.**Logger**(*fname*, *mode*, *suppress*)

write(*text*)

class pytrajectory.log.**Timer**(*label='~'*, *verbose=True*) `pytrajectory.log.`**`err`**(*text*, *lvl=0*)

`pytrajectory.log.`**`info`**(*text*, *lvl=0*)

`pytrajectory.log.`**`logtime`**(*text*, *lvl=0*)

`pytrajectory.log.`**`msg`**(*label*, *text*, *lvl=0*)

`pytrajectory.log.`**`set_file`**(*fname='/usr/bin/sphinx-build2_140416-194105.log'*, *suppress=False*)

`pytrajectory.log.`**`warn`**(*text*, *lvl=0*)

CHAPTER THREE

INDICES AND TABLES

- *genindex*
- *modindex*
- *search*

PYTHON MODULE INDEX

INDEX