pytrajectory Documentation

Release 1.3.1

Andreas Kunze

Jul 20, 2017

Contents

This documentation is built automatically from the source code (commit: 2016-01-15 14:12:08)

CHAPTER 1

PyTrajectory User's Guide

About PyTrajectory

PyTrajectory is a Python library for trajectory generation for nonlinear control systems. It relies on solving a boundary value problem (bvp) via a collocation method. It is based on the scientific work of *Graichen et al.*, but does not depend on proprietary code like Matlabs *bvp4c*.

PyTrajectory is developed at Dresden University of Technology at the Institute for Control Theory, see also other control related software.

Based on the project thesis 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.

Contacts

If you face any problems using PyTrajectory, feel free to contact us.

- andreas.kunze <at> mailbox.tu-dresden.de
- carsten.knoll <at> tu-dresden.de

Licence

```
     notice, this list of conditions and the following disclaimer in the
     documentation and/or other materials provided with the distribution.
  c. Neither the name of PyTrajectory nor the names of its contributors
     may be used to endorse or promote products derived from this software
     without specific prior written permission.


THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
ARE DISCLAIMED. IN NO EVENT SHALL THE REGENTS OR CONTRIBUTORS BE LIABLE FOR
ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR
SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER
CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH
DAMAGE.
```

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*.

Contents

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 and enables an easy implementation. It suffices to supply a function $f(x, u)$ that represents the vectorfield of a control system and to specify the desired boundary

values.

Installation

PyTrajectory has been developed and tested on Python 2.7

If you have troubles installing PyTrajectory, please don't hesitate to *contact* us.

Dependencies

Before you install PyTrajectory make sure you have the following dependencies installed on your system.

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

PyPI

The easiest way of installing PyTrajectory would be

```
$ pip install pytrajectory
```

provided that you have the Python module *pip* installed on your system.

Source

To install PyTrajectory from the source files please download the latest release from here. After the download is complete open the archive and change directory into the extracted folder. Then all you have to do is run the following command

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

Please note that there are different versions of PyTrajectory available (development version in github repository [various branches], release versions at PyPI). Because the documentation is build automatically upon the source code, there are also different versions of the docs available. Please make sure that you always use matching versions of code and documentation.

Windows

To install PyTrajectory on Windows machines please make sure you have already installed Python version 2.7 on your system. If not, please download the latest version and install it by double-clicking the installer file.

To be able to run the Python interpreter from any directory we have to append the *PATH* environment variable. This can be done by right-clicking the machine icon (usually on your Desktop, called *My Computer*), choosing *Properties*, selecting *Advance* and hitting *Environment Variables*. Then select the *PATH* (or *Path*) variable, click *Edit* an append the following at the end of the line

```
;C:\Python27\;C:\Python27\Scripts\
```

If you can't find a variable called *PATH* you can create it by clicking *New*, naming it *PATH* and insert the line above without the first *';'* as the value.

Before going on, open a command line with the shortcut consisting of the *Windows-key* and the *R*-key. Run *cmd* and after the command line interface started type the following:

```
C:\> pip --version
```

If it prints the version number of *pip* you can skip the next two steps. Else, the next thing to do is to install a Python software package called *Setuptools* that extends packaging and installation facilities. To do so, download the Python script ez_setup.py and run it by typing

```
C:>\path\to\file\python ez_setup.py
```

To simplify the installation of new packages we install a software called *pip*. This is simply done by downloading the file get_pip.py and running

```
C:>\path\to\file\python get_pip.py
```

from the command line again.

After that, (and after you have installed the *dependencies* with a similar command like the next one) you can run

```
C:>\pip install pytrajectory
```

and pip should manage to install PyTrajectory.

Again, if you have troubles installing PyTrajectory, please *contact* us.

Note: The information provided in this section follows the guide available here.

MAC OSX

To install PyTrajectory on machines running OSX you first have to make sure there is Python version 2.7 installed on your system (should be with OSX >= 10.8). To check this, open a terminal and type

```
$ python --version
```

If this is not the case we have to install it (obviously). To do so we will use a package manager called *Homebrew* that allows an installation procedure similar to Linux environments. But before we do this pease check if you have XCode installed.

Homebrew can be installed by opening a terminal and typing

```
$ ruby -e "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/master/
↪install)"
```

Once Homebrew is installed we insert its directory at the top of the *PATH* environment variable by adding the following line at the bottom of your *~.profile* file (you have to relogin for this to take effect)

```
export PATH=/usr/local/bin:/usr/local/sbin:$PATH
```

Now, installing Python version 2.7 is as easy as typing

```
$ brew install python2
```

into a terminal. Homebrew also will install packages called *Setuptools* and *pip* that manage the installation of additional Python packages.

Now, before installing PyTrajectory please make sure to install its *dependencies* via

```
$ pip install sympy
```

and similar commands for the others. After that you can install Pytrajectory by typing

```
$ pip install pytrajectory
```

or install it from the *source files*.

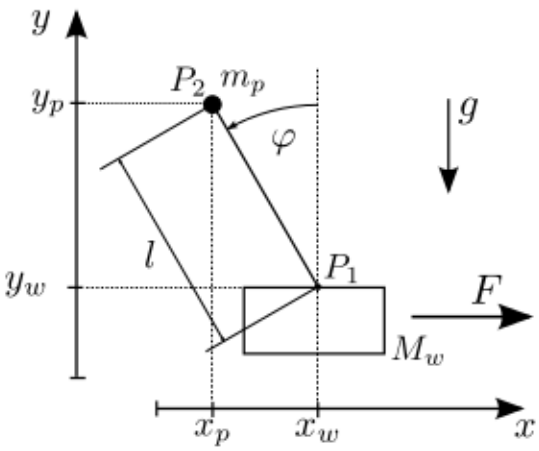
Again, if you have troubles installing PyTrajectory, please *contact* us.

Note: The information provided in this section follows the guide available here.

Usage

In order to illustrate the usage of PyTrajectory we consider the following simple example.

A pendulum mass m_p is connected by a massless rod of length l to a cart M_w on which a force F acts to accelerate it.



A possible task would be the transfer between two angular positions of the pendulum. In this case, the pendulum should hang at first down ($\varphi = \pi$) and is to be turned upwards ($\varphi = 0$). At the end of the process, the car should be at the same position and both the pendulum and the cart should be at rest. The (partial linearised) system is represented by the following differential equations, where $[x_1, x_2, x_3, x_4] = [x_w, \dot{x_w}, \varphi, \dot{\varphi}]$ and $u = \ddot{x}_w$ is our control variable:

$$\begin{aligned} \dot{x_1} &= x_2 \\ \dot{x_2} &= u \\ \dot{x_3} &= x_4 \\ \dot{x_4} &= \frac{1}{l}(g\ sin(x_3) + u\ cos(x_3)) \end{aligned}$$

To solve this problem we first have to define a function that returns the vectorfield of the system above. Therefor it is important that you use SymPy functions if necessary, which is the case here with sin and cos.

So in Python this would be

```
>>> from sympy import sin, cos
>>>
>>> def f(x,u):
...     x1, x2, x3, x4 = x  # system variables
...     u1, = u             # input variable
...
...     l = 0.5     # length of the pendulum
...     g = 9.81    # gravitational acceleration
...
...     # this is the vectorfield
...     ff = [          x2,
...                     u1,
...                     x4,
...         (1/l)*(g*sin(x3)+u1*cos(x3))]
...
...     return ff
...
>>>
```

Wanted is now the course for $u(t)$, which transforms the system with the following start and end states within $T = 2[s]$.

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

so we have to specify the boundary values at the beginning

```
>>> from numpy import pi
>>>
>>> a = 0.0
>>> xa = [0.0, 0.0, pi, 0.0]
```

and end

```
>>> b = 2.0
>>> xb = [0.0, 0.0, 0.0, 0.0]
```

The boundary values for the input variable are

```
>>> ua = [0.0]
>>> ub = [0.0]
```

because we want $u(0) = u(T) = 0$.

Now we import all we need from PyTrajectory

```
>>> from pytrajectory import ControlSystem
```

and pass our parameters.

```
>>> S = ControlSystem(f, a, b, xa, xb, ua, ub)
```

All we have to do now to solve our problem is

```
>>> x, u = S.solve()
```

After the iteration has finished *x(t)* and *u(t)* are returned as callable functions for the system and input variables, where t has to be in (a,b).

In this example we get a solution that satisfies the default tolerance for the boundary values of 10^{-2} after the 7th iteration step with 320 spline parts. But PyTrajectory enables you to improve its performance by altering some of its method parameters.

For example if we increase the factor for raising the spline parts (default: 2)

```
>>> S.set_param('kx', 5)
```

and don't take advantage of the system structure (integrator chains)

```
>>> S.set_param('use_chains', False)
```

we get a solution after 3 steps with 125 spline parts.

There are more method parameters you can change to speed things up, i.e. the type of collocation points to use or the number of spline parts for the input variables. To do so, just type:

```
>>> S.set_param('<param>', <value>)
```

Please have a look at the *PyTrajectory Modules Reference* for more information.

Visualisation

Beyond the simple `plot` method (see: *PyTrajectory Modules Reference*) PyTrajectory offers basic capabilities to animate the given system. This is done via the `Animation` class from the `utilities` module. To explain this feature we take a look at the example above.

When instanciated, the `Animation` requires the calculated simulation results *T.sim* and a callable function that draws an image of the system according to given simulation data.

First we import what we need by:

```
>>> import matplotlib as mpl
>>> from pytrajectory.visualisation import Animation
```

Then we define our function that takes simulation data *x* of a specific time and an instance *image* of *Animation.Image* which is just a container for the image. In the considered example *xt* is of the form

$$xt = [x_1, x_2, x_3, x_4] = [x_w, \dot{x}_w, \varphi, \dot{\varphi}]$$

and *image* is just a container for the drawn image.

```
def draw(xt, image):
    # to draw the image we just need the translation `x` of the
    # cart and the deflection angle `phi` of the pendulum.
    x = xt[0]
    phi = xt[2]

    # next we set some parameters
    car_width = 0.05
    car_heigth = 0.02

    rod_length = 0.5
```

```
    pendulum_size = 0.015

    # then we determine the current state of the system
    # according to the given simulation data
    x_car = x
    y_car = 0

    x_pendulum = -rod_length * sin(phi) + x_car
    y_pendulum = rod_length * cos(phi)

    # now we can build the image

    # the pendulum will be represented by a black circle with
    # center: (x_pendulum, y_pendulum) and radius `pendulum_size
    pendulum = mpl.patches.Circle(xy=(x_pendulum, y_pendulum), radius=pendulum_size,␣
↪color='black')

    # the cart will be represented by a grey rectangle with
    # lower left: (x_car - 0.5 * car_width, y_car - car_heigth)
    # width: car_width
    # height: car_height
    car = mpl.patches.Rectangle((x_car-0.5*car_width, y_car-car_heigth), car_width,␣
↪car_heigth,
                                fill=True, facecolor='grey', linewidth=2.0)

    # the joint will also be a black circle with
    # center: (x_car, 0)
    # radius: 0.005
    joint = mpl.patches.Circle((x_car,0), 0.005, color='black')

    # and the pendulum rod will just by a line connecting the cart and the pendulum
    rod = mpl.lines.Line2D([x_car,x_pendulum], [y_car,y_pendulum],
                           color='black', zorder=1, linewidth=2.0)

    # finally we add the patches and line to the image
    image.patches.append(pendulum)
    image.patches.append(car)
    image.patches.append(joint)
    image.lines.append(rod)

    # and return the image
    return image
```

If we want to save the latest simulation result, maybe because the iteration took much time and we don't want to run it again every time, we can do this.

```
    S.save(fname='ex0_InvertedPendulumSwingUp.pcl')
```

Next, we create an instance of the `Animation` class and pass our `draw` function, the simulation data and some lists that specify what trajectory curves to plot along with the picture.

If we would like to either plot the system state at the end time or want to animate the system we need to create an *Animation* object. To set the limits correctly we calculate the minimum and maximum value of the cart's movement along the x-axis.

```
    A = Animation(drawfnc=draw, simdata=S.sim_data,
                  plotsys=[(0,'x'), (2,'phi')], plotinputs=[(0,'u')])
```

```
    # as for now we have to explicitly set the limits of the figure
    # (may involves some trial and error)
    xmin = np.min(S.sim_data[1][:,0]); xmax = np.max(S.sim_data[1][:,0])
    A.set_limits(xlim=(xmin - 0.5, xmax + 0.5), ylim=(-0.6,0.6))
```

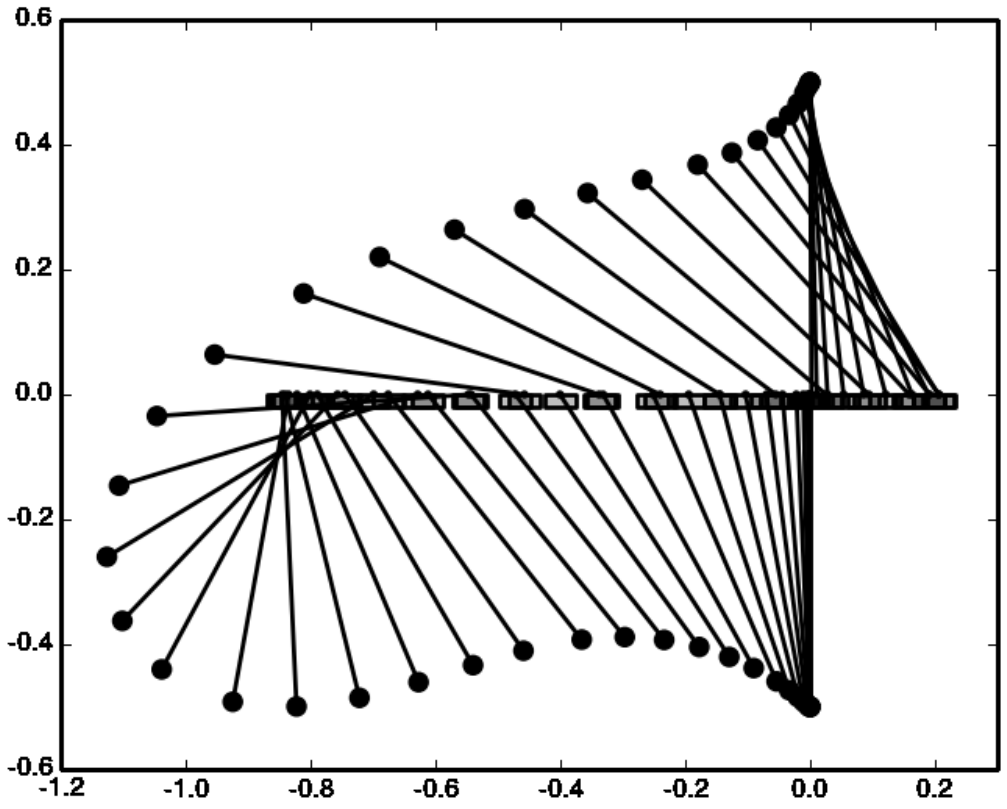
Finally, we can plot the system and/or start the animation.

```
if 'plot' in sys.argv:
    A.show(t=S.b)

if 'animate' in sys.argv:
    # if everything is set, we can start the animation
    # (might take some while)
    A.animate()
```

The animation can be saved either as animated .gif file or as a .mp4 video file.

```
    A.save('ex0_InvertedPendulum.gif')
```

If saved as an animated .gif file you can view single frames using for example *gifview* (GNU/Linux) or the standard Preview app (OSX).



Background

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

Contents

Trajectory planning with BVP's

The task in the field of trajectory planning PyTrajectory is intended to perform, is the transition of a control system between desired states. A possible way to solve such a problem is to treat it as a two-point boundary value problem with free parameters. This approach is based on the work of K. Graichen and M. Zeitz (e.g. see *[Graichen05]*) and was picked up by O. Schnabel (*[Schnabel13]*) in the project thesis from which PyTrajectory emerged. An impressive application of this method is the swingup of the triple pendulum, see *[TU-Wien-video]* and *[Glueck13]*.

Collocation Method

Given a system of autonomous differential equations

$$\begin{aligned} \dot{x}_1(t) &= f_1(x_1(t), ..., x_n(t)) \\ &\vdots \quad \vdots \\ \dot{x}_n(t) &= f_n(x_1(t), ..., x_n(t)) \end{aligned}$$

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 $t_j,\ j = 0, ..., N$ from the interval $[a, b]$ where $t_0 = a,\ t_N = b$ and search for functions $S_i : [a, b] \to \mathbb{R}$ which for all $j = 0, .., N$ satisfy the following conditions:

$$S_i(t_0) = \alpha_i, \qquad S_i(t_N) = \beta_i \tag{1.1}$$

$$\frac{d}{dt} S_i(t_j) = f_i(S_1(t_j), ..., S_n(t_j)) \quad i = 1, ..., n \tag{1.2}$$

Through these demands the exact solution of the differential equation system will be approximated. The demands on the boundary values (1) can be sure already by suitable construction of the candidate functions. This results in the following system of equations.

$$
\begin{aligned}
G_1^0(c) &:= \frac{d}{dt}S_1(t_0) - f(S_1(t_0)) = 0 \\
&\vdots \\
G_n^0(c) &:= \frac{d}{dt}S_n(t_0) - f(S_n(t_0)) = 0 \\
&\vdots \\
G_1^1(c) &:= \frac{d}{dt}S_1(t_1) - f(S_1(t_1)) = 0 \\
&\vdots \\
G_n^N(c) &:= \frac{d}{dt}S_n(t_N) - f(S_n(t_N)) = 0
\end{aligned}
$$

Solving the boundary value problem is thus reduced to the finding of a zero point of $G = (G_1^0, ..., G_n^N)^T$, where c is the vector of all independent parameters that result from the candidate functions.

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_k between the polynomial sections are equidistantly and are referred to as nodes.

$$
\begin{aligned}
t_0 = \tau_0 < \tau_1 < ... < \tau_\eta = t_N \qquad h = \frac{t_N - t_0}{\eta} \\
\tau_{k+1} = \tau_k + h \quad k = 0, ..., \eta - 1
\end{aligned}
$$

The η polynomial sections can be created as follows.

$$
\begin{aligned}
P_k(t) = c_{k,0}(t - kh)^3 + c_{k,1}(t - kh)^2 + c_{k,2}(t - kh) + c_{k,3} \\
c_{k,l} \in \mathbb{R}, \qquad k = 0, ..., \eta - 1,\ l = 0, ..., 3
\end{aligned}
$$

Then, each spline function is defined by

$$
S_i(t) = \begin{cases}
P_1(t) & t_0 \leq t < h \\
\vdots & \vdots \\
P_k(t) & (k-1)h \leq t < kh \\
\vdots & \vdots \\
P_\eta(t) & (\eta - 1)h \leq t \leq \eta h
\end{cases}
$$

The spline functions should be twice continuously differentiable in the nodes τ. Therefore, three smoothness conditions can be set up in all $\tau_k, k = 1, ..., \eta - 1$.

$$
\begin{aligned}
P_k(kh) &= P_{k+1}(kh) \\
\frac{d}{dt}P_k(kh) &= \frac{d}{dt}P_{k+1}(kh) \\
\frac{d^2}{dt^2}P_k(kh) &= \frac{d^2}{dt^2}P_{k+1}(kh)
\end{aligned}
$$

In the later equation system these demands result in the block diagonal part of the matrix. Furthermore, conditions can be set up at the edges arising from the boundary conditions of the differential equation system.

$$\frac{d^j}{dt^j}P_1(\tau_0) = \tilde{\alpha}_j \qquad \frac{d^j}{dt^j}P_\eta(\tau_\eta) = \tilde{\beta}_j \qquad j = 0, ..., \nu$$

The degree ν of the boundary conditions depends on the structure of the differential equation system. With these conditions and those above one obtains the following equation system ($\nu = 2$).

$$\underbrace{\begin{bmatrix}
h^3 & h^2 & h & 1 & 0 & 0 & 0 & -1 & & & & & & & & \\
3h^2 & 2h & 1 & 0 & 0 & 0 & -1 & 0 & & & & & & & & \\
6h & 2 & 0 & 0 & 0 & 0 & -2 & 0 & & & & & & & & \\
 & & & & h^3 & h^2 & h & 1 & 0 & 0 & 0 & -1 & & & & \\
 & & & & 3h^2 & 2h & 1 & 0 & 0 & 0 & -1 & 0 & & & & \\
 & & & & 6h & 2 & 0 & 0 & 0 & -2 & 0 & 0 & & & & \\
 & & & & & & & & & & & & \ddots & & & \\
 & & & & & & & & h^3 & h^2 & h & 1 & 0 & 0 & 0 & -1 \\
 & & & & & & & & 3h^2 & 2h & 1 & 0 & 0 & 0 & -1 & 0 \\
 & & & & & & & & 6h & 2 & 0 & 0 & 0 & -2 & 0 & 0 \\
0 & 0 & 0 & -1 & & & & & & & & & & & & \\
0 & 0 & -1 & 0 & & & & & & & & & & & & \\
0 & -2 & 0 & 0 & & & & & & & & & & & & \\
 & & & & & & & & & & & & h^3 & h^2 & h & 1 \\
 & & & & & & & & & & & & 3h^2 & 2h & 1 & 0 \\
 & & & & & & & & & & & & 6h & 2 & 0 & 0
\end{bmatrix}}_{=:\boldsymbol{M}} \cdot \underbrace{\begin{bmatrix} c_{1,0} \\ c_{1,1} \\ c_{1,2} \\ c_{1,3} \\ c_{2,0} \\ c_{2,1} \\ c_{2,2} \\ c_{2,3} \\ \vdots \\ c_{\eta,0} \\ c_{\eta,1} \\ c_{\eta,2} \\ c_{\eta,3} \end{bmatrix}}_{=:\boldsymbol{c}} = \underbrace{\begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ \vdots \\ 0 \\ 0 \\ 0 \\ \tilde{\alpha}_0 \\ \tilde{\alpha}_1 \\ \tilde{\alpha}_2 \\ \tilde{\beta}_0 \\ \tilde{\beta}_1 \\ \tilde{\beta}_2 \end{bmatrix}}_{=:\boldsymbol{r}}$$

The matrix $\boldsymbol{M}$ of dimension $N_1 \times N_2,\ N_1 < N_2$, where $N_2 = 4\eta$ and $N_1 = 3(\eta-1)+2(\nu+1)$, can be decomposed into two subsystems $\boldsymbol{A} \in \mathbb{R}^{N_1 \times (N_2 - N_1)}$ and $\boldsymbol{B} \in \mathbb{R}^{N_1 \times N_1}$. The vectors $\boldsymbol{a}$ and $\boldsymbol{b}$ belong to the two matrices with the respective coefficients of $\boldsymbol{c}$.

$$\begin{aligned} \boldsymbol{M}\boldsymbol{c} &= \boldsymbol{r} \\ \boldsymbol{A}\boldsymbol{a} + \boldsymbol{B}\boldsymbol{b} &= \boldsymbol{r} \\ \boldsymbol{b} &= \boldsymbol{B}^{-1}(\boldsymbol{r} - \boldsymbol{A}\boldsymbol{a}) \end{aligned}$$

With this allocation, the system of equations can be solved for $\boldsymbol{b}$ and the parameters in $\boldsymbol{a}$ remain as the free parameters of the spline function.

Note: Optionally, there is available an alternative approach for defining the candidate functions, see non-standard-approach.

Use of the system structure

In physical models often occur differential equations of the form

$$\dot{x}_i = x_{i+1}$$

For these equations, it is not necessary to determine a solution through collocation. Without severe impairment of the solution method, it is sufficient to define a candidate function for x_i and to win that of x_{i+1} by differentiation.

$$S_{i+1}(t) = \frac{d}{dt}S_i(t)$$

Then in addition to the boundary conditions of $S_i(t)$ applies

$$\frac{d}{dt}S_i(t_0 = a) = \alpha_{i+1} \qquad \frac{d}{dt}S_i(t_N = b) = \beta_{i+1}$$

Similar simplifications can be made if relations of the form $\dot{x}_i = u_j$ arise.

Levenberg-Marquardt Method

The Levenberg-Marquardt method can be used to solve nonlinear least squares problems. It is an extension of the Gauss-Newton method and solves the following minimization problem.

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

The real number μ is a parameter that is used for the attenuation of the step size $(x_{k+1} - x_k)$ and is free to choose. Thus, the generation of excessive correction is prevented, as is often the case with the Gauss-Newton method and leads to a possible non-achievement of the local minimum. With a vanishing attenuation, $\mu = 0$ the Gauss-Newton method represents a special case of the Levenberg-Marquardt method. The iteration can be specified in the following form.

$$x_{k+1} = x_k - (F'(x_k)^T F'(x_k) + \mu^2 I)^{-1} F'(x_k) F(x_k)$$

The convergence can now be influenced by means of the parameter μ. Disadvantage is that in order to ensure the convergence, μ must be chosen large enough, at the same time, this also leads however to a very small correction. Thus, the Levenberg-Marquardt method has a lower order of convergence than the Gauss-Newton method but approaches the desired solution at each step.

Control of the parameter μ

The feature after which the parameter is chosen, is the change of the actual residual

$$\begin{aligned} R(x_k, s_k) &:= \|F(x_k)\|_2^2 - \|F(x_k + s_k)\|_2^2 \\ s_k &:= x_{k+1} - x_k \end{aligned}$$

and the change of the residual of the linearized approximation.

$$\tilde{R}(x_k, s_k) := \|F(x_k)\|_2^2 - \|F(x_k) + F'(x_k)s_k\|_2^2$$

As a control criterion, the following quotient is introduced.

$$\rho = \frac{R(x_k, s_k)}{\tilde{R}(x_k, s_k)}$$

It follows that $R(x_k, s_k) \geq 0$ and for a meaningful correction $\tilde{R}(x_k, s_k) \geq 0$ must also hold. Thus, ρ is also positive and $\rho \to 1$ for $\mu \to \infty$. Therefor ρ should lie between 0 and 1. To control μ two new limits b_0 and b_1 are introduced with $0 < b_0 < b_1 < 1$ and for $b_0 = 0.2, b_1 = 0.8$ we use the following criteria.

- $\rho \leq b_0$: μ is doubled and s_k is recalculated
- $b_0 < \rho < b_1$: in the next step μ is maintained and s_k is used
- $\rho \geq b_1$: s_k is accepted and μ is halved during the next iteration

Handling constraints

In practical situations it is often desired or necessary that the system state variables comply with certain limits. To achieve this PyTrajectory uses an approach similar to the one presented by K. Graichen and M. Zeitz in *[Graichen06]*.

The basic idea is to transform the dynamical system into a new one that satisfies the constraints. This is done by projecting the constrained state variables on new unconstrained coordinates using socalled *saturation functions*.

Suppose the state x should be bounded by x_0, x_1 such that $x_0 \leq x(t) \leq x_1$ for all $t \in [a, b]$. To do so the following saturation function is introduced

$$x = \psi(y, y^{\pm})$$

that depends on the new unbounded variable y and satisfies the *saturation limits* y^-, y^+, i.e. $y^- \leq \psi(y(t), y^{\pm}) \leq y^+$ for all t. It is assumed that the limits are asymptotically and $\psi(\cdot, y^{\pm})$ is strictly increasing , that is $\frac{\partial \psi}{\partial y} > 0$. For the constraints $x \in [x_0, x_1]$ to hold it is obvious that $y^- = x_0$ and $y^+ = x_1$. Thus the constrained variable x is projected on the new unconstrained varialbe y.

By differentiating the equation above one can replace $\dot{x}$ in the vectorfield with a new term for $\dot{y}$.

$$\dot{x} = \frac{\partial}{\partial y}\psi(y, y^{\pm})\dot{y} \qquad \Leftrightarrow \qquad \dot{y} = \frac{\dot{x}}{\frac{\partial}{\partial y}\psi(y, y^{\pm})}$$

Next, one has to calculate new boundary values $y_a = y(a)$ and $y_b = y(b)$ for the variable y from those, $x_a = x(a)$ and $x_b = x(b)$, of x. This is simply done by

$$y_a = \psi^{-1}(x_a, y^{\pm}) \qquad y_b = \psi^{-1}(x_b, y^{\pm})$$

Now, the transformed dynamical system can be solved where all state variables are unconstrained. At the end a solution for the original state variable x is obtained via a composition of the calculated solution $y(t)$ and the saturation function $\psi(\cdot, y^{\pm})$.

There are some aspects to take into consideration when dealing with constraints:

- The boundary values of a constrained variable have to be strictly within the saturation limits
- It is not possible to make use of an integrator chain that contains a constrained variable

Choice of the saturation functions

As mentioned before the saturation functions should be continuously differentiable and strictly increasing. A possible approach for such functions is the following.

$$\psi(y, y^{\pm}) = y^+ - \frac{y^+ - y^-}{1 + exp(my)}$$

The parameter m affects the slope of the function at $y = 0$ and is chosen such that $\frac{\partial}{\partial y}\psi(0, y^{\pm}) = 1$, i.e.

$$m = \frac{4}{y^+ - y^-}$$

An example

For examples on how to handle constraints with PyTrajectory please have a look at the *Examples* section, e.g. the *Constrained double integrator* or the *Constrained swing up of the inverted pundulum*.

References

Examples

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

The source code of the examples can be downloaded here. In order to run them simply type:

```
$ python ex<ExampleNumber>_<ExampleName>.py
```

The results of the examples latest simulation are saved in a pickle dump file by default. To prevent this add the *no-pickle* command line argument to the command above.

If you want to plot the results and/or animate the example system add the *plot* and/or the *animate* argument to the command.

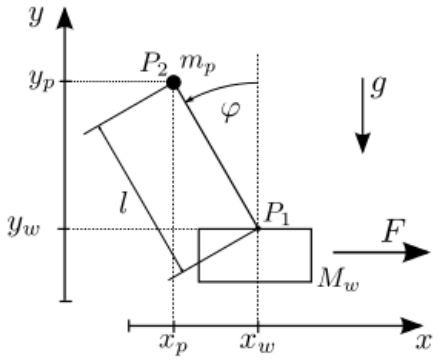
So the command may look something like:

```
$ python ex0_InvertedPendulumSwingUp.py no-pickle plot animate
```

For even more examples, which might not be part of the documentation, have a look at the repository.

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

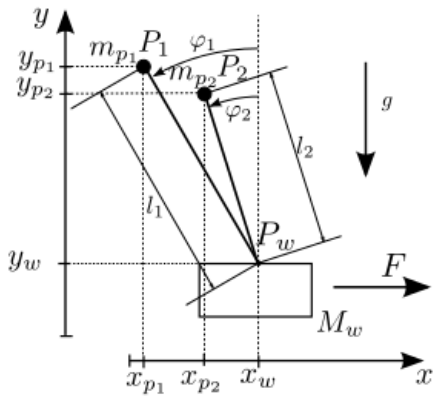
($\varphi = 0$). As a further condition u_1 should start and end steadily in the rest position ($u_1(0) = u_1(T) = 0$). The operating time here is $T = 1[s]$.

Source Code

```
# translation of the inverted pendulum

# import trajectory class and necessary dependencies
from pytrajectory import ControlSystem
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 = 2.0 [s])
xb = [  1.0,
        0.0,
        0.0,
        0.0]

# create trajectory object
S = ControlSystem(f, a=0.0, b=2.0, xa=xa, xb=xb)

# change method parameter to increase performance
S.set_param('use_chains', False)

# run iteration
S.solve()
```

Swing up of the inverted dual 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 $\varphi_1 = \varphi_2 = 0$.

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

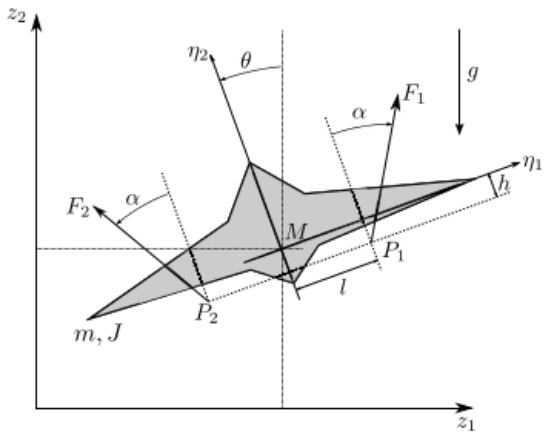
Source Code

```
# swing up of the inverted dual pendulum with partial linearization

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

# 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,  np.pi, 0.0,  np.pi, 0.0]
xb = [0.0, 0.0, 0.0, 0.0, 0.0, 0.0]

# boundary values for the input
ua = [0.0]
ub = [0.0]

# create trajectory object
S = ControlSystem(f, a=0.0, b=2.0, xa=xa, xb=xb, ua=ua, ub=ub)

# alter some method parameters to increase performance
S.set_param('su', 10)
S.set_param('eps', 8e-2)

# run iteration
S.solve()
```

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. $F_1(0) = F_1(T) = F_2(0) = F_2(T) = \frac{mg}{2\cos(\alpha)}$

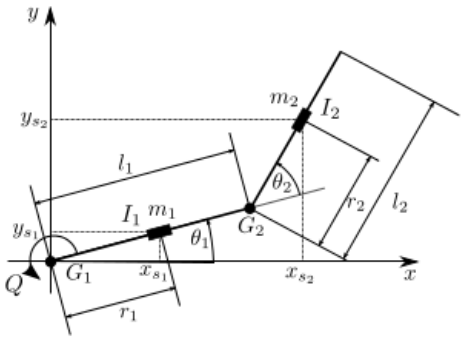
Source Code

```
# vertical take-off aircraft

# import trajectory class and necessary dependencies
from pytrajectory import ControlSystem
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
ua = [0.5*9.81*50.0/(cos(5/360.0*2*pi)), 0.5*9.81*50.0/(cos(5/360.0*2*pi))]
ub = [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
S = ControlSystem(f, a=0.0, b=3.0, xa=xa, xb=xb, ua=ua, ub=ub)

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

# also alter some other method parameters to increase performance
S.set_param('kx', 5)

# run iteration
S.solve()
```

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 ($\tilde{u}(0) = \tilde{u}(T) = 0$).

Source Code

```
# underactuated manipulator

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

# 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,
```

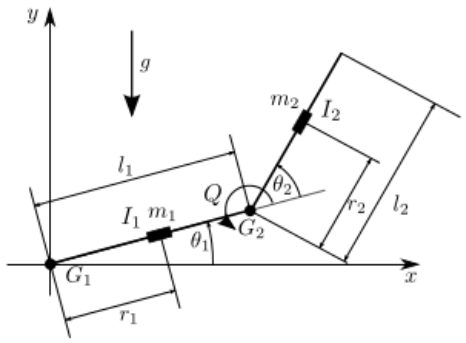
```
                    -e*x2**2*s-(1+e*c)*u1
                    ])

    return ff

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

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

# boundary values for the inputs
ua = [0.0]
ub = [0.0]

# create trajectory object
S = ControlSystem(f, a=0.0, b=1.8, xa=xa, xb=xb, ua=ua, ub=ub)

# also alter some method parameters to increase performance
S.set_param('su', 20)
S.set_param('kx', 3)

# run iteration
S.solve()
```

Acrobot

One further interesting example is that of the acrobot. The model can be regarded as a simplified gymnast hanging on a horizontal bar with both hands. The movements of the entire system is to be controlled only by movement of the hip. The body of the gymnast is represented by two rods which are jointed in the joint G_2. The first rod is movably connected at joint G_1 with the inertial system, which corresponds to the encompassing of the stretching rod with the hands.

For the model, two equal-length rods with a length $l_1 = l_2 = l$ are assumed with a homogeneous distribution of mass $m_1 = m_2 = m$ over the entire rod length. This does not correspond to the proportions of a man, also no restrictions were placed on the mobility of the hip joint.

The following figure shows the schematic representation of the model.



Using the previously assumed model parameters and the write abbreviations

$$\begin{aligned} I &= \frac{1}{3}ml^2 \\ d_{11} &= \frac{ml^2}{4} + m(l^2 + \frac{l^2}{4} + l^2\cos(\theta_2)) + 2I \\ h_1 &= -\frac{ml^2}{2}\sin(\theta_2)(\dot{\theta}_2(\dot{\theta}_2 + 2\dot{\theta}_1)) \\ d_{12} &= m(\frac{l^2}{4} + \frac{l^2}{2}\cos(\theta_1)) + I \\ \varphi_1 &= \frac{3}{2}mlg\cos(\theta_1) + \frac{1}{2}mlg\cos(\theta_1 + \theta_2) \end{aligned}$$

as well as the state vector $x = [\theta_2, \dot{\theta}_2, \theta_1, \dot{\theta}_1]$ one obtains the following state representation with the virtual input $u = \ddot{\theta}_2$

$$\begin{aligned} \dot{x}_1 &= x_2 \\ \dot{x}_2 &= u \\ \dot{x}_3 &= x_4 \\ \dot{x}_4 &= -d_{11}^{-1}(h_1 + \varphi_1 + d_{12}u) \end{aligned}$$

Now, the trajectory of the manipulated variable for an oscillation of the gymnast should be determined. The starting point of the exercise are the two downward hanging rods. These are to be transferred into another rest position in which the two bars show vertically upward within an operating time of $T = 2[s]$. At the beginning and end of the process, the input variable is to merge continuously into the rest position $u(0) = u(T) = 0$.

The initial and final states thus are

$$x(0) = \begin{bmatrix} 0 \\ 0 \\ \frac{3}{2}\pi \\ 0 \end{bmatrix} \rightarrow x(T) = \begin{bmatrix} 0 \\ 0 \\ \frac{1}{2}\pi \\ 0 \end{bmatrix}$$

Source Code

```
# acrobot

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

# define the function that returns the vectorfield
```

```
def f(x,u):
    x1, x2, x3, x4 = x
    u1, = u

    m = 1.0             # masses of the rods [m1 = m2 = m]
    l = 0.5             # lengths of the rods [l1 = l2 = l]

    I = 1/3.0*m*l**2    # moments of inertia [I1 = I2 = I]
    g = 9.81            # gravitational acceleration

    lc = l/2.0

    d11 = m*lc**2+m*(l**2+lc**2+2*l*lc*cos(x1))+2*I
    h1 = -m*l*lc*sin(x1)*(x2*(x2+2*x4))
    d12 = m*(lc**2+l*lc*cos(x1))+I
    phi1 = (m*lc+m*l)*g*cos(x3)+m*lc*g*cos(x1+x3)

    ff = np.array([     x2,
                        u1,
                        x4,
                -1/d11*(h1+phi1+d12*u1)
                ])

    return ff


# system state boundary values for a = 0.0 [s] and b = 2.0 [s]
xa = [  0.0,
        0.0,
        3/2.0*np.pi,
        0.0]

xb = [  0.0,
        0.0,
        1/2.0*np.pi,
        0.0]

# boundary values for the inputs
ua = [0.0]
ub = [0.0]

# create System
first_guess = {'seed' : 1529} # choose a seed which leads to quick convergence
S = ControlSystem(f, a=0.0, b=2.0, xa=xa, xb=xb, ua=ua, ub=ub, use_chains=True, first_
↪guess=first_guess)

# alter some method parameters to increase performance
S.set_param('su', 10)

# run iteration
S.solve()
```

Constrained double integrator

This example is intended to present PyTrajectory's capabilities on handling system constraints. To do so, consider the double integrator which models the dynamics of a simple mass in an one-dimensional space, where a force effects the

acceleration. The state space representation is given by the following dynamical system.

$$\dot{x_1} = x_2$$
$$\dot{x_2} = u_1$$

A possibly wanted trajectory is the translation from $x_1(t_0 = 0) = 0$ to $x_1(T) = 1$ within $T = 2[s]$. At the beginning and end the mass should stay at rest, that is $x_2(0) = x_2(2) = 0$.

Now, suppose we want the velocity to be bounded by $x_{2,min} = 0.0 \leq x_2 \leq 0.65 = x_{2,max}$. To achieve this PyTrajectory needs a dictionary containing the index of the constrained variable in $x = [x_1, x_2]$ and a tuple with the corresponding constraints. So, normally this would look like

```
>>> con = {1 : [0.0, 0.65]}
```

But, due to how the approach for handling system constraints is implemented, this would throw an exception because the lower bound of the constraints $x_{2,min}$ is equal to $x_2(0)$ and has to be smaller. So instead we use the dictionary

```
>>> con = {1 : [-0.1, 0.65]}
```

Source Code

```
'''
This example of the double integrator demonstrates how to pass constraints to␣
↪PyTrajectory.
'''
# imports
from pytrajectory import ControlSystem
import numpy as np

# define the vectorfield
def f(x,u):
    x1, x2 = x
    u1, = u

    ff = [x2,
          u1]

    return ff

# system state boundary values for a = 0.0 [s] and b = 2.0 [s]
xa = [0.0, 0.0]
xb = [1.0, 0.0]

# constraints dictionary
con = {1 : [-0.1, 0.65]}

# create the trajectory object
S = ControlSystem(f, a=0.0, b=2.0, xa=xa, xb=xb, constraints=con, use_chains=False)

# start
x, u = S.solve()
```

Constrained swing up of the inverted pundulum

Reconsider the example of the inverted pendulum in the *Usage* section.

This example is intended to show how PyTrajectory can handle constraints that affect some state variables. Assume we want to restrict the carts movement along the x-axis to the interval $[-0.8, 0.3]$ that is $\forall t \quad -0.8 \leq x_1(t) \leq 0.3$ (remember: $[x_1, x_2, x_3, x_4] = [x_w, \dot{x_w}, \varphi, \dot{\varphi}]$). Furthermore we want the velocity of the cart to be bounded by $[-2.0, 2.0]$.

To set these constraints PyTrajectory expects a dictionary containing the index of the constrained variables as keys and the box constraints as corresponding values. In our case this dictionary would look like

```
>>> con = {0 : [-0.8, 0.3], 1 : [-2.0, 2.0]}
```

(remember that Python starts indexing at 0).

In order to get a solution we raise the translation time from $T = 2[s]$ to $T = 3[s]$. Next, the only different thing to do is to pass the dictionary when instantiating the trajectory object.

```
>>> T = Trajectory(f, a, b=3.0, xa, xb, uab, constraints=con)
```

Source Code

```
'''
This example of the inverted pendulum demonstrates how to handle possible state␣
↪constraints.
'''

# import all we need for solving the problem
from pytrajectory import ControlSystem
import numpy as np
from sympy import cos, sin

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

    l = 0.5     # length of the pendulum
    g = 9.81    # gravitational acceleration

    # this is the vectorfield
    ff = [          x2,
                    u1,
                    x4,
            (1/l)*(g*sin(x3)+u1*cos(x3))]

    return ff

# then we specify all boundary conditions
a = 0.0
xa = [0.0, 0.0, np.pi, 0.0]

b = 3.0
xb = [0.0, 0.0, 0.0, 0.0]

ua = [0.0]
ub = [0.0]

# next, this is the dictionary containing the constraints
con = { 0 : [-0.8, 0.3],
```

```
        1 : [-2.0, 2.0] }

# now we create our Trajectory object and alter some method parameters via the␣
↪keyword arguments
S = ControlSystem(f, a, b, xa, xb, ua, ub, constraints=con, kx=5, use_chains=False)

# time to run the iteration
S.solve()
```

Constrained swing up of the inverted double pendulum

In this example we consider the inverted double pendulum. ... to be continued!

Source Code

```
# constrained double pendulum

# import all we need for solving the problem
from pytrajectory import ControlSystem
import numpy as np
import sympy as sp
from sympy import cos, sin, Matrix
from numpy import pi


# to define a callable function that returns the vectorfield
# we first solve the motion equations of form Mx = B

def solve_motion_equations(M, B, state_vars=[], input_vars=[], parameters_
↪values=dict()):
    '''
    Solves the motion equations given by the mass matrix and right hand side
    to define a callable function for the vector field of the respective
    control system.

    Parameters
    ----------

    M : sympy.Matrix
        A sympy.Matrix containing sympy expressions and symbols that represents
        the mass matrix of the control system.

    B : sympy.Matrix
        A sympy.Matrix containing sympy expressions and symbols that represents
        the right hand site of the motion equations.

    state_vars : list
        A list with sympy.Symbols's for each state variable.

    input_vars : list
        A list with sympy.Symbols's for each input variable.

    parameter_values : dict
        A dictionary with a key:value pair for each system parameter.
```

```
    Returns
    -------

    callable
    '''

    M_shape = M.shape
    B_shape = B.shape
    assert(M_shape[0] == B_shape[0])

    # at first we create a buffer for the string that we complete and execute
    # to dynamically define a function and return it
    fnc_str_buffer ='''
def f(x, u):
    # System variables
    %s  # x_str
    %s  # u_str

    # Parameters
    %s  # par_str

    # Sympy Common Expressions
    %s # cse_str

    # Vectorfield
    %s  # ff_str

    return ff
'''

    ################################
    # handle system state variables #
    ################################
    # --> leads to x_str which shows how to unpack the state variables
    x_str = ''
    for var in state_vars:
        x_str += '%s, '%str(var)

    # as a last we remove the trailing '; ' to avoid syntax erros
    x_str = x_str + '= x'

    ##########################
    # handle input variables #
    ##########################
    # --> leads to u_str which will show how to unpack the inputs of the control␣
↪system
    u_str = ''
    for var in input_vars:
        u_str += '%s, '%str(var)

    # after we remove the trailing '; ' to avoid syntax errors x_str will look like:
    #   'u1, u2, ... , um = u'
    u_str = u_str + '= u'

    ############################
    # handle system parameters #
    ############################
    # --> leads to par_str
```

```
    par_str = ''
    for k, v in parameters_values.items():
        # 'k' is the name of a system parameter such as mass or gravitational␣
↪acceleration
        # 'v' is its value in SI units
        par_str += '%s = %s; '%(str(k), str(v))

    # as a last we remove the trailing '; ' from par_str to avoid syntax errors
    par_str = par_str[:-2]

    # now solve the motion equations w.r.t. the accelerations
    sol = M.solve(B)

    # use SymPy's Common Subexpression Elimination
    cse_list, cse_res = sp.cse(sol, symbols=sp.numbered_symbols('q'))

    ################################
    # handle common subexpressions #
    ################################
    # --> leads to cse_str
    cse_str = ''
    #cse_list = [(str(l), str(r)) for l, r in cse_list]
    for cse_pair in cse_list:
        cse_str += '%s = %s; '%(str(cse_pair[0]), str(cse_pair[1]))

    # add result of cse
    for i in xrange(M_shape[0]):
        cse_str += 'q%d_dd = %s; '%(i, str(cse_res[0][i]))

    cse_str = cse_str[:-2]

    ######################
    # create vectorfield #
    ######################
    # --> leads to ff_str
    ff_str = 'ff = ['

    for i in xrange(M_shape[0]):
        ff_str += '%s, '%str(state_vars[2*i+1])
        ff_str += 'q%s_dd, '%(i)

    # remove trailing ',' and add closing brackets
    ff_str = ff_str[:-2] + ']'

    ############################
    # Create callable function #
    ############################
    # now we can replace all placeholders in the function string buffer
    fnc_str = fnc_str_buffer%(x_str, u_str, par_str, cse_str, ff_str)
    # and finally execute it which will create a python function 'f'
    exec(fnc_str)

    # now we have defined a callable function that can be used within PyTrajectory
    return f
```

```
# system and input variables
state_vars = sp.symbols('x, dx, phi1, dphi1, phi2, dphi2')
input_vars = sp.symbols('F,')
x, dx, phi1, dphi1, phi2, dphi2 = state_vars
F, = input_vars

# parameters
l1 = 0.25                   # 1/2 * length of the pendulum 1
l2 = 0.25                   # 1/2 * length of the pendulum
m1 = 0.1                    # mass of the pendulum 1
m2 = 0.1                    # mass of the pendulum 2
m = 1.0                     # mass of the car
g = 9.81                    # gravitational acceleration
I1 = 4.0/3.0 * m1 * l1**2   # inertia 1
I2 = 4.0/3.0 * m2 * l2**2   # inertia 2

param_values = {'l1':l1, 'l2':l2, 'm1':m1, 'm2':m2, 'm':m, 'g':g, 'I1':I1, 'I2':I2}

# mass matrix
M = Matrix([[      m+m1+m2,           (m1+2*m2)*l1*cos(phi1),    m2*l2*cos(phi2)],
            [(m1+2*m2)*l1*cos(phi1),    I1+(m1+4*m2)*l1**2,    2*m2*l1*l2*cos(phi2-
↪phi1)],
            [  m2*l2*cos(phi2),     2*m2*l1*l2*cos(phi2-phi1),      I2+m2*l2**2]])

# and right hand site
B = Matrix([[ F + (m1+2*m2)*l1*sin(phi1)*dphi1**2 + m2*l2*sin(phi2)*dphi2**2 ],
            [ (m1+2*m2)*g*l1*sin(phi1) + 2*m2*l1*l2*sin(phi2-phi1)*dphi2**2 ],
            [ m2*g*l2*sin(phi2) + 2*m2*l1*l2*sin(phi1-phi2)*dphi1**2 ]])

f = solve_motion_equations(M, B, state_vars, input_vars)

# then we specify all boundary conditions
a = 0.0
xa = [0.0, 0.0, pi, 0.0, pi, 0.0]

b = 4.0
xb = [0.0, 0.0, 0.0, 0.0, 0.0, 0.0]

ua = [0.0]
ub = [0.0]

# here we specify the constraints for the velocity of the car
con = {0 : [-1.0, 1.0],
       1 : [-2.0, 2.0]}

# now we create our Trajectory object and alter some method parameters via the␣
↪keyword arguments
S = ControlSystem(f, a, b, xa, xb, ua, ub, constraints=con,
                  eps=2e-1, su=20, kx=2, use_chains=False,
                  use_std_approach=False)

# time to run the iteration
x, u = S.solve()
```

Swing up of a 3-bar pendulum

Now we consider a cart with 3 pendulums attached to it.

To get a callable function for the vector field of this dynamical system we need to set up and solve its motion equations for the accelaration.

Therefore, the function `n_bar_pendulum` generates the mass matrix M and right hand site B of the motion equations $M\ddot{x} = B$ for a general n-bar pendulum, which we use for the case $n = 3$.

The formulas this function uses are taken from the project report *'Simulation of the inverted pendulum'* written by C. Wachinger, *M. Pock* and *P. Rentrop* at the *Mathematics Departement, Technical University Munich* in December 2004.

Source Code

```
# 3-bar pendulum

# import all we need for solving the problem
from pytrajectory import ControlSystem

import numpy as np
import sympy as sp

from sympy import cos, sin
from numpy import pi

def n_bar_pendulum(N=1, param_values=dict()):
    '''
    Returns the mass matrix :math:`M` and right hand site :math:`B` of motion␣
↪equations

    .. math::
       M * (d^2/dt^2) x = B

    for the :math:`N`\ -bar pendulum.

    Parameters
    ----------

    N : int
        Number of bars.

    param_values : dict
        Numeric values for the system parameters,
        such as lengths, masses and gravitational acceleration.

    Returns
    -------

    sympy.Matrix
        The mass matrix `M`

    sympy.Matrix
        The right hand site `B`

    list
        List of symbols for state variables
```

```
    list
        List with symbol for input variable
    '''

    # first we have to create some symbols
    F = sp.Symbol('F')                      # the force that acts on the car
    g = sp.Symbol('g')                      # the gravitational acceleration
    m = sp.symarray('m', N+1)               # masses of the car (`m0`) and the bars
    l = sp.symarray('l', N+1)#[1:]          # length of the bars (`l0` is not needed␣
↪nor used)
    phi = sp.symarray('phi', N+1)#[1:]      # deflaction angles of the bars (`phi0`␣
↪is not needed nor used)
    dphi = sp.symarray('dphi', N+1)#[1:]    # 1st derivative of the deflaction angles␣
↪(`dphi0` is not needed nor used)

    if param_values.has_key('F'):
        F = param_values['F']
    elif param_values.has_key(F):
        F = param_values[F]

    if param_values.has_key('g'):
        g = param_values['g']
    elif param_values.has_key(g):
        g = param_values[g]
    else:
        g = 9.81

    for i, mi in enumerate(m):
        if param_values.has_key(mi.name):
            m[i] = param_values[mi.name]
        elif param_values.has_key(mi):
            m[i] = param_values[mi]

    for i, li in enumerate(l):
        if param_values.has_key(li.name):
            l[i] = param_values[li.name]
        elif param_values.has_key(li):
            l[i] = param_values[li]

    C = np.empty((N,N), dtype=object)
    S = np.empty((N,N), dtype=object)
    I = np.empty((N), dtype=object)
    for i in xrange(1,N+1):
        for j in xrange(1,N+1):
            C[i-1,j-1] = cos(phi[i] - phi[j])
            S[i-1,j-1] = sin(phi[i] - phi[j])

    for i in xrange(1,N+1):
        if param_values.has_key('I_%d'%i):
            I[i-1] = param_values['I_%d'%i]
        #elif param_values.has_key(Ii):
        #    I[i] = param_values[Ii]
        else:
            I[i-1] = 4.0/3.0 * m[i] * l[i]**2

    #-------------#
    # Mass matrix #
    #-------------#
```

```
    M = np.empty((N+1, N+1), dtype=object)

    # 1st row
    M[0,0] = m.sum()
    for j in xrange(1,N):
        M[0,j] = (m[j] + 2*m[j+1:].sum()) * l[j] * cos(phi[j])
    M[0,N] = m[N] * l[N] * cos(phi[N])

    # rest of upper triangular part, except last column
    for i in xrange(1,N):
        M[i,i] = I[i-1] + (m[i] + 4.0*m[i+1:].sum()) * l[i]**2
        for j in xrange(i+1,N):
            M[i,j] = 2.0*(m[j] + 2.0*m[j+1:].sum())*l[i]*l[j]*C[j-1,i-1]

    # the last column
    for i in xrange(1,N):
        M[i,N] = 2.0*(m[N]*l[i]*l[N]*C[N-1,i-1])
    M[N,N] = I[N-1] + m[N]*l[N]**2

    # the rest (lower triangular part)
    for i in xrange(N+1):
        for j in xrange(i,N+1):
            M[j,i] = 1 * M[i,j]

    #-----------------#
    # Right hand site #
    #-----------------#
    B = np.empty((N+1), dtype=object)

    # first row
    B[0] = F
    for j in xrange(1,N):
        B[0] += (m[j] + 2.0*m[j+1:].sum())*l[j]*sin(phi[j]) * dphi[j]**2
    B[0] += (m[N]*l[N]*sin(phi[N])) * dphi[N]**2

    # rest except for last row
    for i in xrange(1,N):
        B[i] = (m[i] + 2.0*m[i+1:].sum())*g*l[i]*sin(phi[i])
        for j in xrange(1,N):
            B[i] += (2.0*(m[j] + 2.0*m[j+1:].sum())*l[j]*l[i]*S[j-1,i-1]) * dphi[j]**2
        B[i] += (2.0*m[N]*l[N]*l[N]*S[N-1,i-1]) * dphi[N]**2

    # last row
    B[N] = m[N]*g*l[N]*sin(phi[N])
    for j in xrange(1,N+1):
        B[N] += (2.0*m[N]*l[j]*l[N]*S[j-1,N-1]) * dphi[j]**2

    # build lists of state and input variables
    x, dx = sp.symbols('x, dx')
    state_vars = [x, dx]
    for i in xrange(1,N+1):
        state_vars.append(phi[i])
        state_vars.append(dphi[i])
    input_vars = [F]

    # return stuff
    return sp.Matrix(M), sp.Matrix(B), state_vars, input_vars
```

```
def solve_motion_equations(M, B, state_vars=[], input_vars=[], parameters_
↪values=dict()):
    '''
    Solves the motion equations given by the mass matrix and right hand side
    to define a callable function for the vector field of the respective
    control system.

    Parameters
    ----------

    M : sympy.Matrix
        A sympy.Matrix containing sympy expressions and symbols that represent
        the mass matrix of the control system.

    B : sympy.Matrix
        A sympy.Matrix containing sympy expressions and symbols that represent
        the right hand site of the motion equations.

    state_vars : list
        A list with sympy.Symbols's for each state variable.

    input_vars : list
        A list with sympy.Symbols's for each input variable.

    parameter_values : dict
        A dictionary with a key:value pair for each system parameter.

    Returns
    -------

    callable
        A callable function for the vectorfield.
    '''

    M_shape = M.shape
    B_shape = B.shape
    assert(M_shape[0] == B_shape[0])

    # at first we create a buffer for the string that we complete and execute
    # to dynamically define a function and return it
    fnc_str_buffer ='''
def f(x, u):
    # System variables
    %s  # x_str
    %s  # u_str

    # Parameters
    %s  # par_str

    # Sympy Common Expressions
    %s # cse_str

    # Vectorfield
    %s  # ff_str

    return ff
'''
```

```
    ########################################
    # handle system state and input variables #
    ########################################
    # --> leads to x_str and u_str which show how to unpack the variables
    x_str = ''
    u_str = ''

    for var in state_vars:
        x_str += '%s, '%str(var)

    for var in input_vars:
        u_str += '%s, '%str(var)

    x_str = x_str + '= x'
    u_str = u_str + '= u'

    ############################
    # handle system parameters #
    ############################
    # --> leads to par_str
    par_str = ''
    for k, v in parameters_values.items():
        # 'k' is the name of a system parameter such as mass or gravitational␣
↪acceleration
        # 'v' is its value in SI units
        par_str += '%s = %s; '%(str(k), str(v))

    # as a last we remove the trailing '; ' from par_str to avoid syntax errors
    par_str = par_str[:-2]

    # now solve the motion equations w.r.t. the accelerations
    # (might take some while...)
    #print "    -> solving motion equations w.r.t. accelerations"

    # apply sympy.cse() on M and B to speed up solving the eqs
    M_cse_list, M_cse_res = sp.cse(M, symbols=sp.numbered_symbols('M_cse'))
    B_cse_list, B_cse_res = sp.cse(B, symbols=sp.numbered_symbols('B_cse'))

    # solve abbreviated equation system
    #sol = M.solve(B)
    Mse = M_cse_res[0]
    Bse = B_cse_res[0]
    cse_sol = Mse.solve(Bse)

    # substitute back the common subexpressions to the solution
    for expr in reversed(B_cse_list):
        cse_sol = cse_sol.subs(*expr)

    for expr in reversed(M_cse_list):
        cse_sol = cse_sol.subs(*expr)

    # use SymPy's Common Subexpression Elimination
    #cse_list, cse_res = sp.cse(sol, symbols=sp.numbered_symbols('q'))
    cse_list, cse_res = sp.cse(cse_sol, symbols=sp.numbered_symbols('q'))

    ##############################
    # handle common subexpressions #
    ##############################
```

```
    # --> leads to cse_str
    cse_str = ''
    #cse_list = [(str(l), str(r)) for l, r in cse_list]
    for cse_pair in cse_list:
        cse_str += '%s = %s; '%(str(cse_pair[0]), str(cse_pair[1]))

    # add result of cse
    for i in xrange(M_shape[0]):
        cse_str += 'q%d_dd = %s; '%(i, str(cse_res[0][i]))

    cse_str = cse_str[:-2]

    ######################
    # create vectorfield #
    ######################
    # --> leads to ff_str
    ff_str = 'ff = ['

    for i in xrange(M_shape[0]):
        ff_str += '%s, '%str(state_vars[2*i+1])
        ff_str += 'q%s_dd, '%(i)

    # remove trailing ',' and add closing brackets
    ff_str = ff_str[:-2] + ']'

    ############################
    # Create callable function #
    ############################
    # now we can replace all placeholders in the function string buffer
    fnc_str = fnc_str_buffer%(x_str, u_str, par_str, cse_str, ff_str)
    # and finally execute it which will create a python function 'f'
    exec(fnc_str)

    # now we have defined a callable function that can be used within PyTrajectory
    return f

# we consider the case of a 3-bar pendulum
N = 3

# set model parameters
l1 = 0.25                   # 1/2 * length of the pendulum 1
l2 = 0.25                   # 1/2 * length of the pendulum 2
l3 = 0.25                   # 1/2 * length of the pendulum 3
m1 = 0.1                    # mass of the pendulum 1
m2 = 0.1                    # mass of the pendulum 2
m3 = 0.1                    # mass of the pendulum 3
m = 1.0                     # mass of the car
g = 9.81                    # gravitational acceleration
I1 = 4.0/3.0 * m1 * l1**2   # inertia 1
I2 = 4.0/3.0 * m2 * l2**2   # inertia 2
I3 = 4.0/3.0 * m2 * l2**2   # inertia 3

param_values = {'l_1':l1, 'l_2':l2, 'l_3':l3,
                'm_1':m1, 'm_2':m2, 'm_3':m3,
                'm_0':m, 'g':g,
                'I_1':I1, 'I_2':I2, 'I_3':I3}

# get matrices of motion equations
```

```
M, B, state_vars, input_vars = n_bar_pendulum(N=3, param_values=param_values)

# get callable function for vectorfield that can be used with PyTrajectory
f = solve_motion_equations(M, B, state_vars, input_vars)

# then we specify all boundary conditions
a = 0.0
xa = [0.0, 0.0, pi, 0.0, pi, 0.0, pi, 0.0]

b = 3.5
xb = [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]

ua = [0.0]
ub = [0.0]

# now we create our Trajectory object and alter some method parameters via the␣
↪keyword arguments
S = ControlSystem(f, a, b, xa, xb, ua, ub, constraints=None,
                  eps=4e-1, su=30, kx=2, use_chains=False,
                  use_std_approach=False)

# time to run the iteration
x, u = S.solve()
```

CHAPTER 2

PyTrajectory Modules Reference

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.

Contents

system Module

class pytrajectory.system.**ControlSystem**(*ff*, *a=0.0*, *b=1.0*, *xa=[]*, *xb=[]*, *ua=[]*, *ub=[]*, *constraints=None*, ***kwargs*)

Bases: object

Base class of the PyTrajectory project.

Parameters

- **ff** (*callable*) – Vector field (rhs) of the control system.
- **a** (*float*) – Left border of the considered time interval.

- **b** (*float*) – Right border of the considered time interval.
- **xa** (*list*) – Boundary values at the left border.
- **xb** (*list*) – Boundary values at the right border.
- **ua** (*list*) – Boundary values of the input variables at left border.
- **ub** (*list*) – Boundary values of the input variables at right border.
- **constraints** (*dict*) – Box-constraints of the state variables.
- **kwargs** –

key	default value	meaning
sx	10	Initial number of spline parts for the system variables
su	10	Initial number of spline parts for the input variables
kx	2	Factor for raising the number of spline parts
maxIt	10	Maximum number of iteration steps
eps	1e-2	Tolerance for the solution of the initial value problem
ierr	1e-1	Tolerance for the error on the whole interval
tol	1e-5	Tolerance for the solver of the equation system
dt_sim	1e-2	Sample time for integration (initial value problem)
use_chains	True	Whether or not to use integrator chains
sol_steps	100	Maximum number of iteration steps for the eqs solver
first_guess	None	to initiate free parameters (might be useful: {'seed': value})

set_param(*param=''*, *value=None*)

Alters the value of the method parameters.

Parameters

- **param** (*str*) – The method parameter
- **value** – The new value

unconstrain(*constraints*)

This method is used to enable compliance with desired box constraints given by the user. It transforms the vectorfield by projecting the constrained state variables on new unconstrained ones.

Parameters **constraints** (*dict*) – The box constraints for the state variables

constrain()

This method is used to determine the solution of the original constrained state variables by creating a composition of the saturation functions and the calculated solution for the introduced unconstrained variables.

solve()

This is the main loop.

While the desired accuracy has not been reached, the collocation system will be set up and solved with a iteratively raised number of spline parts.

Returns

- *callable* – Callable function for the system state.
- *callable* – Callable function for the input variables.

simulate()

This method is used to solve the resulting initial value problem after the computation of a solution for the input trajectories.

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

It calculates the difference between the solution of the simulation and the given boundary values at the right border and compares its maximum against the tolerance.

If set by the user it also calculates some kind of consistency error that shows how "well" the spline functions comply with the system dynamic given by the vector field.

plot()
Plot the calculated trajectories and show interval error functions.

This method calculates the error functions and then calls the `visualisation.plotsim` function.

save(*fname=None*)
Save data using the python module `pickle`.

a

b

class pytrajectory.system.**DynamicalSystem**(*f_sym*, *a=0.0*, *b=1.0*, *xa=[]*, *xb=[]*, *ua=[]*, *ub=[]*)
Bases: object

Provides access to information about the dynamical system that is the object of the control process.

Parameters

- **f_sym** (*callable*) – The (symbolic) vector field of the dynamical system
- **b** (*a,*) – The initial end final time of the control process
- **xb** (*xa,*) – The initial and final conditions for the state variables
- **ub** (*ua,*) – The initial and final conditions for the input variables

trajectories Module

class pytrajectory.trajectories.**Trajectory**(*sys*, ***kwargs*)
Bases: object

This class handles the creation and managing of the spline functions that are intended to approximate the desired trajectories.

Parameters **sys** (system.DynamicalSystem) – Instance of a dynamical system providing information like vector field function and boundary values

n_parts_x
Number of polynomial spline parts for system variables.

n_parts_u
Number of polynomial spline parts for input variables.

x(*t*)
Returns the current system state.

Parameters **t** (*float*) – The time point in (a,b) to evaluate the system at.

u(*t*)
Returns the state of the input variables.

Parameters **t** (*float*) – The time point in (a,b) to evaluate the input variables at.

dx(*t*)
Returns the state of the 1st derivatives of the system variables.

Parameters **t** (*float*) – The time point in (a,b) to evaluate the 1st derivatives at.

init_splines()
This method is used to create the necessary spline function objects.

Parameters **boundary_values** (*dict*) – Dictionary of boundary values for the state and input splines functions.

set_coeffs(*sol*)
Set found numerical values for the independent parameters of 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.

Parameters **sol** (*numpy.ndarray*) – The solution vector for the free parameters, i.e. the independent coefficients.

save()

collocation Module

class pytrajectory.collocation.**CollocationSystem**(*sys*, ***kwargs*)
Bases: object

This class represents the collocation system that is used to determine a solution for the free parameters of the control system, i.e. the independent coefficients of the trajectory splines.

Parameters **sys** (system.DynamicalSystem) – Instance of a dynamical system

build()
This method is used to set up the equations for the collocation equation system and defines functions for the numerical evaluation of the system and its jacobian.

get_guess()
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 the free parameters with arbitrary 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 system is the new start value for the solver.

save()

solve(*G*, *DG*, *new_solver=True*)
This method is used to solve the collocation equation system.

Parameters

- **G** (*callable*) – Function that "evaluates" the equation system.
- **DG** (*callable*) – Function for the jacobian.
- **new_solver** (*bool*) – flag to determine whether a new solver instance should be initialized (default True)

class pytrajectory.collocation.**Container**(***kwargs*)
Bases: object

Simple data structure to store additional internal information for debugging and checking the algorithms. Some of the attributes might indeed be neccessary

pytrajectory.collocation.**collocation_nodes**(*a*, *b*, *npts*, *coll_type*)
Create collocation points/nodes for the equation system.

Parameters

- **a** (*float*) – The left border of the considered interval.
- **b** (*float*) – The right border of the considered interval.
- **npts** (*int*) – The number of nodes.
- **coll_type** (*str*) – Specifies how to generate the nodes.

Returns The collocation nodes.

Return type numpy.ndarray

splines Module

class pytrajectory.splines.**Spline**(*a=0.0*, *b=1.0*, *n=5*, *bv={}*, *tag=''*, *use_std_approach=False*, ***kwargs*)
Bases: object

This class provides a representation of a cubic spline function.

It simultaneously enables access to the spline function itself as well as to its derivatives up to the 3rd order. Furthermore it has its own method to ensure the steadiness and smoothness conditions of its polynomial parts in the joining points.

For more information see: *Candidate Functions*

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 up into.
- **tag** (*str*) – The 'name' of the spline object.
- **bv** (*dict*) – Boundary values the spline function and/or its derivatives should satisfy.
- **use_std_approach** (*bool*) – Whether to use the standard spline interpolation approach or the ones used in the project thesis

f(*t*)
This is just a wrapper to evaluate the spline itself.

df(*t*)
This is just a wrapper to evaluate the spline's 1st derivative.

ddf(*t*)
This is just a wrapper to evaluate the spline's 2nd derivative.

dddf(*t*)
This is just a wrapper to evaluate the spline's 3rd derivative.

boundary_values

make_steady()
Please see *pytrajectory.splines.make_steady*

differentiate(*d=1*, *new_tag=''*)
Returns the *d*-th derivative of this spline function object.

Parameters **d** (*int*) – The derivation order.

get_dependence_vectors(*points*, *d=0*)
This method yields a provisionally evaluation of the spline while there are no numerical values for its free parameters.

It returns a two vectors which reflect the dependence of the spline's or its *d*-th derivative's coefficients on its free parameters (independent coefficients).

Parameters

- **points** (*float*) – The points to evaluate the provisionally spline at.
- **d** (*int*) – The derivation order.

set_coefficients(*free_coeffs=None*, *coeffs=None*)
This function is used to set up numerical values either for all the spline's coefficients or its independent ones.

Parameters

- **free_coeffs** (*numpy.ndarray*) – Array with numerical values for the free coefficients of the spline.
- **coeffs** (*numpy.ndarray*) – Array with coefficients of the polynomial spline parts.

interpolate(*fnc=None*, *m0=None*, *mn=None*)
Determines the spline's coefficients such that it interpolates a given function.

save()

plot(*show=True*, *ret_array=False*)
Plots the spline function or returns an array with its values at some points of the spline interval.

Parameters

- **show** (*bool*) – Whethter to plot the spline's curve or not.
- **ret_array** (*bool*) – Wheter to return an array with values of the spline at points of the interval.

pytrajectory.splines.**get_spline_nodes**(*a=0.0*, *b=1.0*, *n=10*, *nodes_type='equidistant'*)
Generates n spline nodes in the interval $[a, b]$ of given type.

Parameters

- **a** (*float*) – Lower border of the considered interval.
- **b** (*float*) – Upper border of the considered interval.
- **n** (*int*) – Number of nodes to generate.
- **nodes_type** (*str*) – How to generate the nodes.

pytrajectory.splines.**differentiate**(*spline_fnc*)
Returns the derivative of a callable spline function.

Parameters **spline_fnc** (*callable*) – The spline function to derivate.

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

Please see the documentation for more details: *Candidate Functions*

Parameters **S** (Spline) – The spline function object for which to solve smoothness and boundary conditions.

pytrajectory.splines.**get_smoothness_matrix**(*S*, *N1*, *N2*)
Returns the coefficient matrix and right hand site for the equation system that ensures the spline's smoothness in its joining points and its compliance with the boundary conditions.

Parameters

- **S** (Spline) – The spline function object to get the matrix for.
- **N1** (*int*) – First dimension of the matrix.
- **N2** (*int*) – Second dimension of the matrix.

Returns

- *array_like* – The coefficient matrix for the equation system.
- *array_like* – The right hand site of the equation system.

solver Module

class pytrajectory.solver.**Solver**(*F*, *DF*, *x0*, *tol=1e-05*, *reltol=2e-05*, *maxIt=50*, *method='leven'*, *mu=0.0001*)
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
- **maxIt** (*int*) – The maximum number of iterations of the solver
- **method** (*str*) – The solver to use

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

leven()
This method is an implementation of the Levenberg-Marquardt-Method to solve nonlinear least squares problems.

For more information see: *Levenberg-Marquardt Method*

simulation Module

class pytrajectory.simulation.**Simulator**(*ff*, *T*, *start*, *u*, *dt=0.01*)
Bases: object

This class simulates the initial value problem that results from solving the boundary value problem of the control system.

Parameters

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

rhs(*t*, *x*)
Retruns the right hand side (vector field) of the ode system.

calcStep()
Calculates one step of the simulation.

simulate()
Starts the simulation

Returns

Return type List of numpy arrays with time steps and simulation data of system and input variables.

auxiliary Module

exception pytrajectory.auxiliary.**NanError**
Bases: exceptions.ValueError

class pytrajectory.auxiliary.**IntegChain**(*lst*)
Bases: object

This class provides a representation of an integrator chain.

For the elements $(x_i)_{i=1,\dots,n}$ of the chain the relation $\dot{x}_i = x_{i+1}$ applies.

Parameters **lst** (*list*) – Ordered list of the integrator chain's elements.

elements
tuple – Ordered list of all elements that are part of the integrator chain

upper
str – Upper end of the integrator chain

lower
str – Lower end of the integrator chain

elements
Return an ordered list of the integrator chain's elements.

upper
Returns the upper end of the integrator chain, i.e. the element of which all others are derivatives of.

lower
Returns the lower end of the integrator chain, i.e. the element which has no derivative in the integrator chain.

pytrajectory.auxiliary.**find_integrator_chains**(*dyn_sys*)
Searches for integrator chains in given vector field matrix *fi*, i.e. equations of the form $\dot{x}_i = x_j$.

Parameters **dyn_sys** (pytrajectory.system.DynamicalSystem) – Instance of a dynamical system

Returns

- *list* – Found integrator chains.
- *list* – Indices of the equations that have to be solved using collocation.

pytrajectory.auxiliary.**sym2num_vectorfield**(*f_sym*, *x_sym*, *u_sym*, *vectorized=False*, *cse=False*)

This function takes a callable vector field of a control system that is to be evaluated with symbols for the state and input variables and returns a corresponding function that can be evaluated with numeric values for these variables.

Parameters

- **f_sym** (*callable or array_like*) – The callable ("symbolic") vector field of the control system.
- **x_sym** (*iterable*) – The symbols for the state variables of the control system.
- **u_sym** (*iterable*) – The symbols for the input variables of the control system.
- **vectorized** (*bool*) – Whether or not to return a vectorized function.
- **cse** (*bool*) – Whether or not to make use of common subexpressions in vector field

Returns The callable ("numeric") vector field of the control system.

Return type callable

pytrajectory.auxiliary.**check_expression**(*expr*)

Checks whether a given expression is a sympy epression or a list of sympy expressions.

Throws an exception if not.

pytrajectory.auxiliary.**make_cse_eval_function**(*input_args*, *replacement_pairs*, *ret_filter=None*, *namespace=None*)

Returns a function that evaluates the replacement pairs created by the sympy cse.

Parameters

- **input_args** (*iterable*) – List of additional symbols that are necessary to evaluate the replacement pairs
- **replacement_pairs** (*iterable*) – List of (Symbol, expression) pairs created from sympy cse
- **ret_filter** (*iterable*) – List of sympy symbols of those replacements that should be returned from the created function (if None, all are returned)
- **namespace** (*dict*) – A namespace in which to define the function

pytrajectory.auxiliary.**cse_lambdify**(*args*, *expr*, ***kwargs*)

Wrapper for sympy.lambdify which makes use of common subexpressions.

pytrajectory.auxiliary.**saturation_functions**(*y_fnc*, *dy_fnc*, *y0*, *y1*)

Creates callable saturation function and its first derivative to project the solution found for an unconstrained state variable back on the original constrained one.

For more information, please have a look at *Handling constraints*.

Parameters

- **y_fnc** (*callable*) – The calculated solution function for an unconstrained variable.

- **dy_fnc** (*callable*) – The first derivative of the unconstrained solution function.
- **y0** (*float*) – Lower saturation limit.
- **y1** (*float*) – Upper saturation limit.

Returns

- *callable* – A callable of a saturation function applied to a calculated solution for an unconstrained state variable.
- *callable* – A callable for the first derivative of a saturation function applied to a calculated solution for an unconstrained state variable.

pytrajectory.auxiliary.**consistency_error**(*I*, *x_fnc*, *u_fnc*, *dx_fnc*, *ff_fnc*, *npts=500*, *return_error_array=False*)

Calculates an error that shows how "well" the spline functions comply with the system dynamic given by the vector field.

Parameters

- **I** (*tuple*) – The considered time interval.
- **x_fnc** (*callable*) – A function for the state variables.
- **u_fnc** (*callable*) – A function for the input variables.
- **dx_fnc** (*callable*) – A function for the first derivatives of the state variables.
- **ff_fnc** (*callable*) – A function for the vectorfield of the control system.
- **npts** (*int*) – Number of point to determine the error at.
- **return_error_array** (*bool*) – Whether or not to return the calculated errors (mainly for plotting).

Returns

- *float* – The maximum error between the systems dynamic and its approximation.
- *numpy.ndarray* – An array with all errors calculated on the interval.

visualisation Module

pytrajectory.visualisation.**plot_simulation**(*sim_data*, *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_data** (*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

class pytrajectory.visualisation.**Animation**(*drawfnc*, *simdata*, *plotsys=[]*, *plotinputs=[]*, *rcParams=None*)

Provides animation capabilities.

Given a callable function that draws an image of the system state and smiulation data this class provides a method to created an animated representation of the system.

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

This is just a container for the drawn system.

reset()

Animation.**get_axes**()

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

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

Animation.**show**(*t=0.0*, *xlim=None*, *ylim=None*, *axes_callback=None*, *save_fname=None*, *show=True*)

Plots one frame of the system animation.

Parameters **t** (*float*) – The time for which to plot the system

Animation.**animate**()

Starts the animation of the system.

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

Saves the animation as a video file or animated gif.

CHAPTER 3

Indices and tables

- genindex
- modindex
- search

Bibliography

[Graichen05] Graichen, K. and Hagenmeyer, V. and Zeitz, M. “A new approach to inversion-based feedforward control design for nonlinear systems” Automatica, Volume 41, Issue 12, Pages 2033-2041, 2005

[Graichen06] Graichen, K. and Zeitz, M. “Inversionsbasierter Vorsteuerungsentwurf mit Ein- und Ausgangsbeschränkungen (Inversion-Based Feedforward Control Design under Input and Output Constraints)” at - *Automatisierungstechnik*, 54.4, Pages 187-199, 2006

[Graichen07] Graichen, K. and Treuer, M and Zeitz, M. “Swing-up of the double pendulum on a cart by feed forward and feedback control with experimental validation” Automatica, Volume 43, Issue 1, Pages 63-71, 2007

[Schnabel13] Schnabel, O. “Untersuchungen zur Trajektorienplanung durch Lösung eines Randwertproblems” Technische Universität Dresden, Institut für Regelungs- und Steuerungstheorie, 2013

[TU-Wien-video] Glück, T. et. al. “Triple Pendulum on a Cart”, (laboratory video), https://www.youtube.com/watch?v=cyN-CRNrb3E

[Glueck13] Glück, T. and Eder, A. and Kugi, A. “Swing-up control of a triple pendulum on a cart with experimental validation”, Automatica (2013), doi:10.1016/j.automatica.2012.12.006

Python Module Index

p

Index

P

R

S

T

U

X