nlControl Documentation

Release 1.0.2

Jasper Juchem

Oct 21, 2020

CONTENTS:

Nlcontrol is a comprehensive library for simulating nonlinear control loops with Python. The toolbox is developed to be used by people who are not shy to dive into Python code, as well as for users who are just interested in results.

The toolbox is far from complete, so contribute your own systems and controllers, based on the base classes. This allows easy integration in the closed loop class.

Note: This module is originally developed in the Dynamical Systems & Control group of Ghent University.

CHAPTER

ONE

GET STARTED

1.1 Installation

The installation procedure requires Python 3. Some additional packages are required and are installed upon installation of the *nlcontrol*. Currently, only pip is available.

1.1.1 pip

If you use *pip* you can install the package as follows:

```
pip install nlcontrol
```

Warning: the dependency module python-control has an optional dependency *slycot*, which should be installed separately. More info can be found here.

1.1.2 Current Release

- **2020-10-07** nlcontrol-1.0.2.tar.gz

1.1.3 Past Releases

- **2020-10-01** nlcontrol-1.0.1.tar.gz

1.1.4 Development Source

The main repository for *nlcontrol* is located on github at https://github.com/jjuch/nlcontrol.

You can obtain a copy of the active source code by issuing the following command

```
git clone https://github.com/jjuch/nlcontrol.git
```

1.2 Usage

Import the module in your Python code by using the following statement:

```
import nlcontrol
```

To import specific parts of the *nlcontrol* module use the following statement:

```
from nlcontrol import < *what-you-want-to-import* >
```

CHAPTER

TWO

API

Here, you can find all information on the different classes, definitions, etc. of the *nlcontrol* module. There are three main classes: SystemBase, ControllerBase, and ClosedLoop. Next to these base classes, there are more advanced system and controller classes. This list is far from completed. If you created a new controller or system based on the base classes, do not hesitate to contribute it to this toolbox to help humankind.

2.1 The Idea

The advantage of using this SystemBase and ControllerBase classes is that it can easily be implemented in a closed loop configuration with another SystemBase and/or controllerBase object.

This toolbox is strongly based on the SimuPy module. The contribution of this module is to create a more accessible nonlinear control toolbox, which can be used by proficient Python programmers as well as for users who do not want to focus on programming at all.

2.2 The Docs

2.2.1 Systems

Base System

class nlcontrol.systems.system.**SystemBase**(*states*, *inputs*, *sys=None*)

Bases: object

Returns a base structure for a system with outputs, optional inputs, and optional states. The system is defines by it state equations (optional):

$$\frac{dx(t)}{dt} = h(x(t), u(t), t)$$

with x(t) the state vector, u(t) the input vector and t the time in seconds. Next, the output is given by the output equation:

$$y(t) = g(x(t), u(t), t)$$

A SystemBase object contains several basic functions to manipulate and simulate the system.

Parameters

states [string or array-like] if *states* is a string, it is a comma-separated listing of the state names. If *states* is array-like it contains the states as sympy's dynamic symbols.

inputs [string or array-like] if *inputs* is a string, it is a comma-separated listing of the input names. If *inputs* is array-like it contains the inputs as sympy's dynamic symbols.

system [simupy's DynamicalSystem object (simupy.systems.symbolic), optional] the object containing output and state equations, default: None.

Examples

- **Statefull system with one state, one input, and one output:**

```
>>> from simupy.systems.symbolic import MemorylessSystem,␣
↪DynamicalSystem
>>> from sympy.tensor.array import Array
>>> states = 'x'
>>> inputs = 'u'
>>> sys = SystemBase(states, inputs)
>>> x, xdot, u = sys.create_variables()
>>> sys.system = DynamicalSystem(state_equation=Array([-x + u1]),␣
↪state=x, output_equation=x, input_=u1)
```

- **Statefull system with two states, one input, and two outputs:**

```
>>> states = 'x1, x2'
>>> inputs = 'u'
>>> sys = SystemBase(states, inputs)
>>> x1, x2, x1dot, x2dot, u = sys.create_variables()
>>> sys.system = DynamicalSystem(state_equation=Array([-x1 + x2**2␣
↪+ u, -x2 + 0.5 * x1]), state=Array([x1, x2]), output_
↪equation=Array([x1 * x2, x2]), input_=u)
```

- **Stateless system with one input:**

```
>>> states = None
>>> inputs = 'w'
>>> sys = SystemBase(states, inputs)
>>> w = sys.create_variables()
>>> sys.system = MemorylessSystem(input_=Array([w]), output_
↪equation= Array([5 * w]))
```

- **Create a copy a SystemBase object `sys' and linearize around the working point of state [0, 0] and working point**

```
>>> new_sys = SystemBase(sys.states, sys.inputs, sys.system)
>>> new_sys_lin = new_sys.linearize([0, 0], 0)
>>> new_sys_lin.simulation(10)
```

Attributes

block_configuration Returns info on the systems: the dimension of the inputs, the states, and the output.

output_equation `expression` containing `dynamicsymbols`

state_equation `expression` containing `dynamicsymbols`

system `simupy's DynamicalSystem`

Methods

create_variables([input_diffs, states])	Returns a tuple with all variables.
linearize(working_point_states[, ...])	In many cases a nonlinear system is observed around a certain working point.
parallel(sys_append)	A system is generated which is the result of a parallel connection of two systems.
series(sys_append)	A system is generated which is the result of a serial connection of two systems.
simulation(tspan[, number_of_samples, ...])	Simulates the system in various conditions.

property block_configuration
the dimension of the inputs, the states, and the output. This property is mainly intended for debugging.

Type Returns info on the systems

create_variables(*input_diffs: bool = False*, *states=None*) → tuple
Returns a tuple with all variables. First the states are given, next the derivative of the states, and finally the inputs, optionally followed by the diffs of the inputs. All variables are sympy dynamic symbols.

Parameters

input_diffs [boolean] also return the differentiated versions of the inputs, default: false.

states [array-like] An alternative list of states, used by more complex system models, optional. (see e.g. EulerLagrange.create_variables)

Returns

variables [tuple] all variables of the system.

Examples

- Return the variables of `sys', which has two states and two inputs and add a system to the SytemBase object:

```
>>> from sympy.tensor.array import Array
>>> from simupy.systems.symbolic import DynamicalSystem
>>> x1, x2, x1dot, x2dot, u1, u2, u1dot, u2dot = sys.create_
↪variables(input_diffs=True)
>>> state_eq = Array([-5 * x1 + x2 + u1**2, x1/2 - x2**3 + u2])
>>> output_eq = Array([x1 + x2])
>>> sys.system = DynamicalSystem(input_=Array([u1, u2],␣
↪state=Array([x1, x2], state_equation=state_eq, output_
↪equation=output_eq)
```

linearize(*working_point_states*, *working_point_inputs=None*)
In many cases a nonlinear system is observed around a certain working point. In the state space close to this working point it is save to say that a linearized version of the nonlinear system is a sufficient approximation. The linearized model allows the user to use linear control techniques to examine the nonlinear system close to this working point. A first order Taylor expansion is used to obtain the linearized system. A working point for the states is necessary, but the working point for the input is optional.

Parameters

working_point_states [list or int] the state equations are linearized around the working point of the states.

working_point_inputs [list or int] the state equations are linearized around the working point of the states and inputs.

Returns

sys_lin: SystemBase object with the same states and inputs as the original system. The state and output equation is linearized.

sys_control: control.StateSpace object

Examples

- **Print the state equation of the linearized system of `sys' around the state's working point x[1] = 1 and x[2] =**

```
>>> sys_lin, sys_control = sys.linearize([1, 5], 2)
>>> print('Linearized state equation: ', sys_lin.state_equation)
```

property output_equation

`expression` containing `dynamicsymbols`

The output equation contains sympy's dynamicsymbols.

parallel (*sys_append*)

A system is generated which is the result of a parallel connection of two systems. The inputs of this object are connected to the system that is placed in parallel and a new system is achieved with the output the sum of the outputs of both systems in parallel. Notice that the dimensions of the inputs and the outputs of both systems should be equal.

Parameters

sys_append [SystemBase object] the system that is added in parallel.

Returns

A SystemBase object with the parallel system's equations.

Examples

- **Place 'sys2' in parallel with 'sys1' and show the inputs, states, state equations and output equations:**

```
>>> parallel_sys = sys1.parallel(sys2)
>>> print('inputs: ', parallel_sys.system.input_)
>>> print('States: ', parallel_sys.system.state)
>>> print('State eqs: ', parallel_sys.system.state_equation)
>>> print('Output eqs: ', parallel_sys.system.output_equation)
```

series (*sys_append*)

A system is generated which is the result of a serial connection of two systems. The outputs of this object are connected to the inputs of the appended system and a new system is achieved which has the inputs of the current system and the outputs of the appended system. Notice that the dimensions of the output of the current system should be equal to the dimension of the input of the appended system.

Parameters

sys_append [SystemBase object] the system that is placed in a serial configuration. 'sys_append' follows the current system.

Returns

A SystemBase object with the serial system's equations.

Examples

- **Place 'sys1' behind 'sys2' in a serial configuration and show the inputs, states, state equations and output eq**

```
>>> series_sys = sys1.series(sys2)
>>> print('inputs: ', series_sys.system.input_)
>>> print('States: ', series_sys.system.state)
>>> print('State eqs: ', series_sys.system.state_equation)
>>> print('Output eqs: ', series_sys.system.output_equation)
```

simulation(*tspan*, *number_of_samples=100*, *initial_conditions=None*, *input_signals=None*, *plot=False*, *custom_integrator_options=None*)

Simulates the system in various conditions. It is possible to impose initial conditions on the states of the system. A specific input signal can be applied to the system to check its behavior. The results of the simulation are numerically available. Also, a plot of the states, inputs, and outputs is available. To simulate the system scipy's ode is used if the system has states. Both the option of variable time-step and fixed time step are available. If there are no states, a time signal is applied to the system. # TODO: output_signal -> a disturbance on the output signal.

Parameters

tspan [float or list-like] the parameter defines the time vector for the simulation in seconds. An integer indicates the end time. A list-like object with two elements indicates the start and end time respectively. And more than two elements indicates at which time instances the system needs to be simulated.

number_of_samples [int, optional] number of samples in the case that the system is stateless and tspan only indicates the end and/or start time (span is length two or smaller), default: 100

initial_conditions [int, float, list-like object, optional] the initial conditions of the states of a statefull system. If none is given, all are zero, default: None

input_signals [SystemBase object] the input signal that is directly connected to the system's inputs. Preferably, the signals in nlcontrol.signals are used. If no input signal is specified and the system has inputs, all inputs are defaulted to zero, default: None

plot [boolean, optional] the plot boolean decides whether to show a plot of the inputs, states, and outputs, default: False

custom_integrator_options [dict, optional (default: None)] Specify specific integrator options top pass to integrator_class.set_integrator (scipy ode)`. The options are 'name', 'rtol', 'atol', 'nsteps', and 'max_step', which specify the integrator name, relative tolerance, absolute tolerance, number of steps, and maximal step size respectively. If no custom integrator options are specified the `DEFAULT_INTEGRATOR_OPTIONS` are used:

```
{
    'name': 'dopri5',
    'rtol': 1e-6,
    'atol': 1e-12,
    'nsteps': 500,
    'max_step': 0.0
}
```

Returns

A tuple:

-> statefull system :

t [ndarray] time vector.

x [ndarray] state vectors.

y [ndarray] input and ouput vectors.

res [SimulationResult object] A class object which contains information on events, next to the above vectors.

-> stateless system :

t [ndarray] time vector.

y [ndarray] output vectors.

u [ndarray] input vectors. Is an empty list if the system has no inputs.

Examples

- **A simulation of 20 seconds of the statefull system 'sys' for a set of initial conditions [x0_0, x1_0, x2_0] and p**

```
>>> init_cond = [0.3, 5.7, 2]
>>> t, x, y, u, res = sys.simulation(20, initial_
↪conditions=init_cond)
```

- **A simulation from second 2 to 18 of the statefull system 'sys' for an input signal, which is a step from 0.4 to**

```
>>> from nlcontrol.signals import step
>>> step_signal = step(step_times=[5, 7], begin_values=[0.4,␣
↪0.9], end_values=[1.3, 11])
>>> integrator_options = {'nsteps': 1000}
>>> t, x, y, u, res = sys.simulation([2, 18], input_
↪signals=step_signal, custom_integrator_options=integrator_
↪options)
```

- **Plot the stateless signal step from previous example for a custom time axis (a time axis going from 3 seconds**

```
>>> import numpy as np
>>> time_axis = np.linspace(3, 20, 1000)
>>> t, y, _ = step_signal.simulation(time_axis, plot=True)
Or
>>> t, y, _ = step_signal.simulation([3, 20], number_of_
↪samples=1000, plot=True)
```

- **Simulate the stateless system 'sys_stateless' with input signal step_signal from the previous examples for 40**

```
>>> t, y, u = sys_stateless.simulation(40, number_of_
↪samples=1500, input_signals=step_signal, plot=True)
```

property state_equation

`expression` containing `dynamicsymbols`

The state equation contains sympy's dynamicsymbols.

property system

`simupy's DynamicalSystem`

The system attribute of the SystemBase class. The system is defined using simupy's DynamicalSystem.

Specific Systems

class nlcontrol.systems.eula.**EulerLagrange**(*states*, *inputs*, *sys=None*)
Bases: *nlcontrol.systems.system.SystemBase*

A class that defines SystemBase object using an Euler-Lagrange formulation:

$$M(x).x'' + C(x, x').x' + K(x) = F(u)$$

Here, x represents a minimal state:

$$[x_1, x_2, ...]$$

the apostrophe represents a time derivative, and u is the input vector:

$$[u_1, u_2, ...]$$

A SystemBase object uses a state equation function of the form:

$$x' = f(x, u)$$

However, as system contains second time derivatives of the state, an extended state x* is necessary containing the minimized states and its first time derivatives:

$$x^* = [x_1, x_1', x_2, x_2', ...]$$

which makes it possible to adhere to the SystemBase formulation:

$$x^{*'} = f(x^*, u)$$

Parameters

states [string or array-like] if *states* is a string, it is a comma-separated listing of the state names. If *states* is array-like it contains the states as sympy's dynamic symbols.

inputs [string or array-like] if *inputs* is a string, it is a comma-separated listing of the input names. If *inputs* is array-like it contains the inputs as sympy's dynamic symbols.

sys [simupy's DynamicalSystem object (simupy.systems.symbolic), optional] the object containing output and state equations, default: None.

Examples

- **Create a EulerLagrange object with two states and two inputs:**

```
>>> states = 'x1, x2'
>>> inputs = 'u1, u2'
>>> sys = EulerLagrange(states, inputs)
>>> x1, x2, dx1, dx2, u1, u2, du1, du2 = sys.create_
↪variables(input_diffs=True)
>>> M = [[1, x1*x2],
    [x1*x2, 1]]
>>> C = [[2*dx1, 1 + x1],
    [x2 - 2, 3*dx2]]
>>> K = [x1, 2*x2]
>>> F = [u1, 0]
>>> sys.define_system(M, C, K, F)
```

- **Get the Euler-Lagrange matrices and the state equations:**

```
>>> M = sys.inertia_matrix
>>> C = sys.damping_matrix
>>> K = sys.stiffness_matrix
>>> F = sys.force_vector
>>> xdot = sys.state_equation
```

- **Linearize an Euler-Lagrange system around the state's working point [0, 0, 0, 0] and the input's working poin**

```
>>> sys_lin, _ = sys.linearize([0, 0, 0, 0], [0, 0])
>>> from nlcontrol.signals import step
>>> step_sgnl = step(2)
>>> init_cond = [1, 2, 0.5, 4]
>>> sys_lin.simulation(5, initial_conditions=init_cond, input_
↪signals=step_sgnl, plot=True)
```

Attributes

block_configuration Returns info on the systems: the dimension of the inputs, the states, and the output.

damping_matrix sympy Matrix

force_vector sympy Matrix

inertia_matrix sympy Matrix

output_equation expression containing dynamicsymbols

state_equation expression containing dynamicsymbols

stiffness_matrix sympy Matrix

system simupy's DynamicalSystem

Methods

check_symmetry(matrix)	Check if matrix is symmetric.
create_state_equations()	As the system contains a second derivative of the states, an extended state should be used, which contains the first derivative of the states as well.
create_variables([input_diffs])	Returns a tuple with all variables.
define_system(M, C, K, F)	Define the Euler-Lagrange system using the differential equation representation:
linearize(working_point_states[, ...])	In many cases a nonlinear system is observed around a certain working point.
parallel(sys_append)	A system is generated which is the result of a parallel connection of two systems.
series(sys_append)	A system is generated which is the result of a serial connection of two systems.
simulation(tspan[, number_of_samples, ...])	Simulates the system in various conditions.

check_symmetry(*matrix*) → bool

Check if matrix is symmetric. Returns a bool.

Returns

value [bool] the matrix being symmetric or not.

create_state_equations()

As the system contains a second derivative of the states, an extended state should be used, which contains the first derivative of the states as well. Therefore, the state equation has to be adapted to this new state vector.

Returns

result [sympy array object] the state equation for each element in self.states

create_variables(*input_diffs: bool = False*)

Returns a tuple with all variables. First the states are given, next the derivative of the states, and finally the inputs, optionally followed by the diffs of the inputs. All variables are sympy dynamic symbols.

Parameters

input_diffs [boolean] also return the differentiated versions of the inputs, default: false.

Returns

variables [tuple] all variables of the system.

Examples

- **Return the variables of 'sys', which has two states and two inputs and add a system to the EulerLagrange**

```
>>> x1, x2, x1dot, x2dot, u1, u2, u1dot, u2dot = sys.create_
↪variables(input_diffs=True)
>>> M = [[1, x1*x2],
    [x1*x2, 1]]
>>> C = [[2*x1dot, 1 + x1],
    [x2 - 2, 3*x2dot]]
>>> K = [x1, 2*x2]
>>> F = [u1, 0]
>>> sys.define_system(M, C, K, F)
```

property damping_matrix

sympy Matrix

The matrix represents the damping and coriolis forces. More on sympy's Matrix.

define_system(*M*, *C*, *K*, *F*)

Define the Euler-Lagrange system using the differential equation representation:

$$M(x).x'' + C(x, x').x' + K(x) = F(u)$$

Here, x is the minimal state vector created in the constructor. The state-space model is generated in the form $x^{*'} = f(x^*, u)$, with $x^* = [x_1, dx_1, x_2, dx_2, ...]$, the extended state vector. The output is the minimal state vector.

Note: Use create_variables() for an easy notation of state[i] and dstate[i].

---

property force_vector
sympy Matrix

The matrix represents the external force or torque vector. This is a non-square matrix. More on sympy's Matrix.

property inertia_matrix
sympy Matrix

The matrix represents the inertia forces and it is checked that it is positive definite and symmetric. More on sympy's Matrix.

property stiffness_matrix
sympy Matrix

The matrix represents the elastic and centrifugal forces. More on sympy's Matrix.

Utilities

nlcontrol.systems.utils.**read_simulation_result_from_csv**(*file_name*, *plot=False*)

Read a csv file created with *write_simulation_result_to_csv()* containing simulation results. Based on the header it is determined if the results contains input or event vector. There is a possibility to create plot of the data.

Parameters

file_name [string] The filename of the csv file, containing the extension.

plot [boolean, optional] Create a plot, default: False

Returns

tuple :

t [numpy array] The time vector.

x [numpy array] The state vectors.

y [numpy array] The output vectors. Contains the inputs, when the data contains the event vector.

u or e [numpy array] The input vectors or event vectors. See boolean 'contains_u' to know which one.

contains_u [boolean] Indicates whether the output contains the input or event vector.

Examples

- **Read and plot a csv file 'results.csv' with an input vector:**

```
>>> t, x, y, u, contains_u = read_simulation_result_from_csv(
↪'results.csv', plot=True)
>>> print(contains_u)
    True
```

- **Read and plot a csv file 'results.csv' with an event vector:**

```
>>> t, x, y, e, contains_u = read_simulation_result_from_csv(
↪'results.csv', plot=True)
>>> print(contains_u)
    False
```

nlcontrol.systems.utils.**write_simulation_result_to_csv**(*simulation_result*, *file_name=None*)

Write the results of a SimulationResult object (see simupy.BlockDiagram.simulate) to a csv file. This object type is also returned by a SystemBase's simulation function.

Parameters

simulation_result [SimulationResult object or list] Results of a simulation packaged as Simupy's SimulationResult object or a list which includes the time, input, state, and output vector in this order.

file_name [string] The filename of the newly created csv file. Defaults to a timestamp.

Examples

- **Simulate a SystemBase object called 'sys' and store the results:**

```
>>> t, x, y, u, res = sys.simulation(1)
>>> write_simulation_result_to_csv(res, file_name='use_
↪simulation_result_object')
>>> write_simulation_result_to_csv([t, u, x, y], file_name='use_
↪separate_vectors')
```

2.2.2 Controllers

Base Controller

class nlcontrol.systems.controllers.controller.**ControllerBase**(*states*, *inputs*, *sys=None*)

Bases: *nlcontrol.systems.system.SystemBase*

Returns a base structure for a controller with outputs, optional inputs, and optional states. The controller is an instance of a SystemBase, which is defined by it state equations (optional):

$$\frac{dx(t)}{dt} = h(x(t), u(t), t)$$

with x(t) the state vector, u(t) the input vector and t the time in seconds. Next, the output is given by the output equation:

$$y(t) = g(x(t), u(t), t)$$

Parameters

states [string or array-like] if *states* is a string, it is a comma-separated listing of the state names. If *states* is array-like it contains the states as sympy's dynamic symbols.

inputs [string or array-like] if *inputs* is a string, it is a comma-separated listing of the input names. If *inputs* is array-like it contains the inputs as sympy's dynamic symbols.

system [simupy's DynamicalSystem object (simupy.systems.symbolic), optional] the object containing output and state equations, default: None.

Examples

- **Statefull controller with one state, one input, and one output:**

```
>>> from simupy.systems.symbolic import MemorylessSystem,␣
↪DynamicalSystem
>>> from sympy.tensor.array import Array
>>> st = 'z'
>>> inp = 'w'
>>> contr = ControllerBase(states=st, inputs=inp)
>>> z, zdot, w = contr.create_variables()
>>> contr.system = DynamicalSystem(state_equation=Array([-z +␣
↪w]), state=z, output_equation=z, input_=w)
```

- **Statefull controller with two states, one input, and two outputs:**

```
>>> st = 'z1, z2'
>>> inp = 'w'
>>> contr = ControllerBase(states=st, inputs=inp)
>>> z1, z2, z1dot, z2dot, w = contr.create_variables()
>>> contr.system = DynamicalSystem(state_equation=Array([-z1 +␣
↪z2**2 + w, -z2 + 0.5 * z1]), state=Array([z1, z2]), output_
↪equation=Array([z1 * z2, z2]), input_=w)
```

- **Stateless controller with one input:**

```
>>> st = None
>>> inp = 'w'
>>> contr = ControllerBase(states=st, inputs=inp)
>>> w = contr.create_variables()
>>> contr.system = MemorylessSystem(input_=Array([w]), output_
↪equation= Array([5 * w]))
```

- **Create a copy a ControllerBase object 'contr' and linearize around the working point of state [0, 0] and worki**

```
>>> new_contr = ControllerBase(states=contr.states,␣
↪inputs=contr.inputs, sys=contr.system)
>>> new_contr_lin = new_contr.linearize([0, 0], 0)
>>> new_contr_lin.simulation(10)
```

Attributes

block_configuration Returns info on the systems: the dimension of the inputs, the states, and the output.

output_equation `expression` containing `dynamicsymbols`

state_equation `expression` containing `dynamicsymbols`

system `simupy's DynamicalSystem`

Methods

`create_variables`([input_diffs, states])	Returns a tuple with all variables.
`linearize`(working_point_states[, ...])	In many cases a nonlinear system is observed around a certain working point.
`parallel`(contr_append)	A controller is generated which is the result of a parallel connection of two controllers.
`series`(contr_append)	A controller is generated which is the result of a serial connection of two controllers.
`simulation`(tspan[, number_of_samples, ...])	Simulates the system in various conditions.

parallel(*contr_append*)

A controller is generated which is the result of a parallel connection of two controllers. The inputs of this object are connected to the system that is placed in parallel and a new system is achieved with the output the sum of the outputs of both systems in parallel. Notice that the dimensions of the inputs and the outputs of both systems should be equal.

Parameters

contr_append [ControllerBase object] the controller that is added in parallel.

Returns

A ControllerBase object with the parallel system's equations.

Examples

- Place 'contr2' in parallel with 'contr1' and show the inputs, states, state equations and output equations:

```
>>> parallel_sys = contr1.parallel(contr2)
>>> print('inputs: ', parallel_sys.system.input_)
>>> print('States: ', parallel_sys.system.state)
>>> print('State eqs: ', parallel_sys.system.state_equation)
>>> print('Output eqs: ', parallel_sys.system.output_equation)
```

series(*contr_append*)

A controller is generated which is the result of a serial connection of two controllers. The outputs of this object are connected to the inputs of the appended system and a new controller is achieved which has the inputs of the current system and the outputs of the appended system. Notice that the dimensions of the output of the current system should be equal to the dimension of the input of the appended system.

Parameters

contr_append [ControllerBase object] the controller that is placed in a serial configuration. 'contr_append' follows the current system.

Returns

A ControllerBase object with the serial system's equations.

Examples

- **Place 'contr1' behind 'contr2' in a serial configuration and show the inputs, states, state equations and o**

```
>>> series_sys = contr1.series(contr2)
>>> print('inputs: ', series_sys.system.input_)
>>> print('States: ', series_sys.system.state)
>>> print('State eqs: ', series_sys.system.state_equation)
>>> print('Output eqs: ', series_sys.system.output_equation)
```

Typical Controllers

class nlcontrol.systems.controllers.basic.**PID**(*inputs=w*) *PID(ksi0, chi0, psi0, inputs=inputs)*

Bases: *nlcontrol.systems.controllers.controller.ControllerBase*

A nonlinear PID controller can be created using the PID class. This class is based on the ControllerBase object. The nonlinear PID is is based on the input vector w(t), containing sympy's dynamicsymbols. The formulation is the following:

$$u(t) = \xi_0(w(t)) + \chi_0\left(\int (w(t), t)\right) + \psi_0(w'(t))$$

with $.'(t)$ indicating the time derivative of the signal. The class object allows the construction of P, PI, PD and PID controllers, by setting chi0 or psi0 to None. The system is based on a MemorylessSystem object from simupy.

Parameters

args [optional]

ksi0 [array-like] A list of P-action expressions, containing the input signal.

chi0 [array-like] A list of I-action expressions, containing the integral of the input signal.

psi0 [array-like] A list of D-action expressions, containing the derivative of the input signal.

kwargs :

inputs [array-like or string] if *inputs* is a string, it is a comma-separated listing of the input names. If *inputs* is array-like it contains the inputs as sympy's dynamic symbols.

Examples

- **Create a classic PD controller with two inputs:**

```
>>> C = PID(inputs='w1, w2')
>>> w1, w2, w1dot, w2dot = C.create_variables()
>>> kp = 1, kd = 5
>>> ksi0 = [kp * w1, kp * w2]
>>> psi0 = [kd * w1dot, kd * w2dot]
>>> C.define_PID(ksi0, None, psi0)
```

- **Same as exercise as above, but with a different constructor:**

```
>>> from sympy.physics.mechanics import dynamicsymbols
>>> from sympy import Symbol, diff
>>> w = dynamicsymbols('w1, w2')
>>> w1, w2 = tuple(inputs)
>>> kp = 1, kd = 5
>>> ksi0 = [kp * w1, kp * w2]
>>> psi0 = [kd * diff(w1, Symbol('t')), kd * diff(w2, Symbol('t
↪'))]
>>> C = PID(ksi0, None, psi0, inputs=w)
```

- **Formulate a standard I-action chi0:**

```
>>> from sympy.physics.mechanics import dynamicsymbols
>>> from sympy import Symbol, integrate
>>> w = dynamicsymbols('w1, w2')
>>> w1, w2 = tuple(inputs)
>>> ki = 0.5
>>> chi0 = [ki * integrate(w1, Symbol('t')), ki * integrate(w2,␣
↪Symbol('t'))]
```

Attributes

D_action

I_action

P_action

block_configuration Returns info on the systems: the dimension of the inputs, the states, and the output.

output_equation `expression` containing `dynamicsymbols`

state_equation `expression` containing `dynamicsymbols`

system `simupy's DynamicalSystem`

Methods

`create_variables`([input_diffs, states])	Returns a tuple with all variables.
`define_PID`(P, I, D)	Set all three PID actions with one function, instead of using the setter functions for each individual action.
`linearize`(working_point_states[, ...])	In many cases a nonlinear system is observed around a certain working point.
`parallel`(contr_append)	A controller is generated which is the result of a parallel connection of two controllers.
`series`(contr_append)	A controller is generated which is the result of a serial connection of two controllers.
`simulation`(tspan[, number_of_samples, ...])	Simulates the system in various conditions.

property D_action

!! processed by numpydoc !!

property I_action

!! processed by numpydoc !!

property P_action

!! processed by numpydoc !!

define_PID(*P*, *I*, *D*)

Set all three PID actions with one function, instead of using the setter functions for each individual action. Automatic checking of the dimensions is done as well. The PID's system arguments is set to a simupy's MemorylessSystem object, containing the proper PID expressions. Both P, PI, PD and PID can be formed by setting the appropriate actions to None.

Parameters

P [list or expression] A list of expressions or an expression defining ksi0.

I [list or expression or None] A list of expressions or an expression defining chi0. If I is None, the controller does not contain an I-action.

D [list or expression or None] A list of expressions or an expression defining psi0. If D is None, the controller does not contain a D-action.

Statefull Controllers

class nlcontrol.systems.controllers.eulaC.**DynamicController**(*states=None*, *inputs=None*, *sys=None*)

Bases: *nlcontrol.systems.controllers.controller.ControllerBase*

The DynamicController object is based on the ControllerBase class. A dynamic controller is defined by the following differential equations:

$$\frac{dz(t)}{dt} = A.z(t) - B.f(\sigma(t)) + \eta\left(w(t), \frac{dw(t)}{dt}\right)$$

$$\sigma(t) = C'.z$$

$$u_0 = \phi\left(z(t), \frac{dz(t)}{dt}\right)$$

with z(t) the state vector, w(t) the input vector and t the time in seconds. the symbol ' refers to the transpose.

Conditions:

- A is Hurwitz,
- (A, B) should be controllable,
- (A, C) is observable,
- rank(B) = rang (C) = s <= n, with s the dimension of sigma, and n the number of states.

More info on the controller can be found in [1, 2].

Parameters

states [string or array-like] if *states* is a string, it is a comma-separated listing of the state names. If *states* is array-like it contains the states as sympy's dynamic symbols.

inputs [string or array-like] if *inputs* is a string, it is a comma-separated listing of the input names. If *inputs* is array-like it contains the inputs as sympy's dynamic symbols. Do not provide the derivatives as these will be added automatically.

system [simupy's DynamicalSystem object (simupy.systems.symbolic), optional] the object containing output and state equations, default: None.

References

[1] L. Luyckx, The nonlinear control of underactuated mechanical systems. PhD thesis, UGent, Ghent, Belgium, 5 2006.

[2] M. Loccufier, "Stabilization and set-point regulation of underactuated mechanical systems", Journal of Physics: Conference Series, 2016, vol. 744, no. 1, p.012065.

Examples

- **Statefull controller with two states, one input, and two outputs:**

```
>>> inp = 'w'
>>> st = 'z1, z2'
>>> contr = DynamicController(states=st, inputs=inp)
>>> z1, z2, z1dot, z2dot, w, wdot = contr.create_variables()
>>> a0, a1, k1 = 12.87, 6.63, 0.45
>>> b0 = (48.65 - a1) * k1
>>> b1 = (11.79 - 1) * k1
>>> A = [[0, 1], [-a0, -a1]]
>>> B = [[0], [1]]
>>> C = [[b0], [b1]]
>>> f = lambda x: x**2
>>> eta = [[w + wdot], [(w + wdot)**2]]
>>> phi = [[z1], [z2dot]]
>>> contr.define_controller(A, B, C, f, eta, phi)
>>> print(contr)
```

Attributes

block_configuration Returns info on the systems: the dimension of the inputs, the states, and the output.

output_equation `expression` containing `dynamicsymbols`

state_equation `expression` containing `dynamicsymbols`

system `simupy's DynamicalSystem`

Methods

controllability_linear(A, B)	Controllability check of two matrices using the Kalman rank condition for time-invariant linear systems [1].
create_variables([input_diffs, states])	Returns a tuple with all variables.
define_controller(A, B, C, f, eta, phi)	Define the Dynamic controller given by the differential equations:
hurwitz(matrix)	Check whether a time-invariant matrix is Hurwitz.
linearize(working_point_states[, ...])	In many cases a nonlinear system is observed around a certain working point.
observability_linear(A, C)	Observability check of two matrices using the Kalman rank condition for time-invariant linear systems [1].
parallel(contr_append)	A controller is generated which is the result of a parallel connection of two controllers.
series(contr_append)	A controller is generated which is the result of a serial connection of two controllers.
simulation(tspan[, number_of_samples, ...])	Simulates the system in various conditions.

controllability_linear (*A*, *B*)

Controllability check of two matrices using the Kalman rank condition for time-invariant linear systems [1].

Reference:

[1]. R.E. Kalman, "On the general theory of control systems", IFAC Proc., vol. 1(1), pp. 491-502, 1960. doi.10.1016/S1474-6670(17)70094-8.

Parameters

A [array-like] Size: n x n

B [array-like] Size: s x n

define_controller (*A*, *B*, *C*, *f*, *eta*, *phi*)

Define the Dynamic controller given by the differential equations:

$$\frac{dz(t)}{dt} = A.z(t) - B.f(\sigma(t)) + \eta\left(w(t), \frac{dw(t)}{dt}\right)$$

$$\sigma(t) = C'.z$$

$$u_0 = \phi\left(z(t), \frac{dz(t)}{dt}\right)$$

with z(t) the state vector, w(t) the input vector and t the time in seconds. the symbol ' refers to the transpose. Conditions:

- A is Hurwitz,
- (A, B) should be controllable,
- (A, C) is observable,
- rank(B) = rang (C) = s <= n, with s the dimension of sigma, and n the number of states.

HINT: use create_variables() for an easy notation of the equations.

Parameters

A [array-like] Hurwitz matrix. Size: n x n

B [array-like] In combination with matrix A, the controllability is checked. The linear definition can be used. Size: s x n

C [array-like] In combination with matrix A, the observability is checked. The linear definition can be used. Size: n x 1

f [callable (lambda-function)] A (non)linear lambda function with argument sigma, which equals C'.z.

eta [array-like] The (non)linear relation between the inputs plus its derivatives to the change in state. Size: n x 1

phi [array-like] The (non)linear output equation. The equations should only contain states and its derivatives. Size: n x 1

hurwitz(*matrix*)

Check whether a time-invariant matrix is Hurwitz. The real part of the eigenvalues should be smaller than zero.

Parameters

matrix: array-like A square matrix.

observability_linear(*A*, *C*)

Observability check of two matrices using the Kalman rank condition for time-invariant linear systems [1].

Reference:

[1] R.E. Kalman, "On the general theory of control systems", IFAC Proc., vol. 1(1), pp. 491-502, 1960. doi.10.1016/S1474-6670(17)70094-8.

Parameters

A [array-like] Size: n x n

C [array-like] Size: n x 1

class nlcontrol.systems.controllers.eulaC.**EulerLagrangeController**(*D0*, *C0*, *K0*, *C1*, *f*, *NA*, *NB*, *inputs*, *nonlinearity_type='stiffness'*)

Bases: *nlcontrol.systems.controllers.eulaC.DynamicController*

The EulerLagrangeController object is based on the DynamicController class. The control equation is:

$$D0.p'' + C0.p' + K0.p + C1.f(C1^T.p) + N0.w' = 0$$

The apostrophe represents a time derivative, $.^T$ is the transpose of the matrix.

The output equation is:

$$NA^T.D0^{-1}.K0^{-1}.D0.K0.p - NB^T.D0^{-1}.K0^{-1}.D0.K0.p'$$

More info on the controller can be found in [1, 2]. Here, the parameters are chosen to be

- $\bar{\gamma} = 0$
- $\bar{\alpha} = I$

with I the identity matrix.

Parameters

D0 [matrix-like] inertia matrix with numerical values. The matrix should be positive definite and symmetric.

C0 [matrix-like] linear damping matrix with numerical values. The matrix should be positive definite and symmetric.

K0 [matrix-like] linear stiffness matrix with numerical values. The matrix should be positive definite and symmetric.

C1 [matrix-like] nonlinear function's constant matrix with numerical values.

f [matrix-like] nonlinear functions containing lambda functions.

NA [matrix-like] the numerical multipliers of the position feedback variables.

NB [matrix-like] the numerical multipliers of the velocity feedback variables.

nonlinearity_type [string] the nonlinear part acts on the state or the derivative of the state of the dynamic controller. The only options are `stiffness' and `damping'.

References

[1] L. Luyckx, The nonlinear control of underactuated mechanical systems. PhD thesis, UGent, Ghent, Belgium, 5 2006.

[2] M. Loccufier, "Stabilization and set-point regulation of underactuated mechanical systems", Journal of Physics: Conference Series, 2016, vol. 744, no. 1, p.012065.

Examples

- **An Euler-Lagrange controller with two states, the input is the minimal state of a BasicSystem `sys' and the no**

```
>>> from sympy import atan
>>> D0 = [[1, 0], [0, 1.5]]
>>> C0 = [[25, 0], [0, 35]]
>>> K0 = [[1, 0], [0, 1]]
>>> C1 = [[0.5, 0], [0, 0.5]]
>>> s_star = 0.01
>>> f0 = 10
>>> f1 = 1
>>> f2 = (f0 - f1)*s_star
>>> func = lambda x: f1 * x + f2 * atan((f0 - f1)/f2 * x)
>>> f = [[func], [func]]
>>> NA = [[0, 0], [0, 0]]
```

(continues on next page)

(continued from previous page)

```
>>> NB = [[3, 0], [2.5, 0]]
>>> contr = EulerLagrangeController(D0, C0, K0, C1, f, NA, NB,␣
↪sys.minimal_states, nonlinearity_type='stiffness')
```

Attributes

D0 [inertia_matrix] Inertia forces.

C0 [damping_matrix] Damping and Coriolis forces.

K0 [stiffness_matrix] Elastic en centrifugal forces.

C1 [nonlinear_coefficient_matrix] Coëfficient of the nonlinear functions.

nl [nonlinear_fcts] Nonlinear functions of the controller.

NA [gain_inputs] Coëfficients of the position inputs.

NB [gain_dinputs] Coëfficients of the velocity inputs.

inputs [sympy array of dynamicsymbols] input variables.

dinputs [sympy array of dynamicsymbols] derivative of the input array

states [sympy array of dynamicsymbols] state variables.

Methods

check_symmetry(matrix)	Check if matrix is symmetric.
controllability_linear(A, B)	Controllability check of two matrices using the Kalman rank condition for time-invariant linear systems [1].
convert_to_dynamic_controller()	The Euler-Lagrange formalism is transformed to the state and output equation notation of the DynamicController class.
create_variables([input_diffs, states])	Returns a tuple with all variables.
define_controller(A, B, C, f, eta, phi)	Define the Dynamic controller given by the differential equations:
hurwitz(matrix)	Check whether a time-invariant matrix is Hurwitz.
linearize(working_point_states[, ...])	In many cases a nonlinear system is observed around a certain working point.
observability_linear(A, C)	Observability check of two matrices using the Kalman rank condition for time-invariant linear systems [1].
parallel(contr_append)	A controller is generated which is the result of a parallel connection of two controllers.
series(contr_append)	A controller is generated which is the result of a serial connection of two controllers.
simulation(tspan[, number_of_samples, ...])	Simulates the system in various conditions.

check_positive_definite	
create_states	

check_positive_definite(*matrix: sympy.matrices.dense.MutableDenseMatrix*)

check_symmetry(*matrix*) → bool
Check if matrix is symmetric. Returns a bool.

convert_to_dynamic_controller()
The Euler-Lagrange formalism is transformed to the state and output equation notation of the DynamicController class.

create_states(*size: int*, *level: int = 0*)

property damping_matrix
!! processed by numpydoc !!

property gain_dinputs
!! processed by numpydoc !!

property gain_inputs
!! processed by numpydoc !!

property inertia_matrix
!! processed by numpydoc !!

property nonlinear_coefficient_matrix
!! processed by numpydoc !!

property nonlinear_fcts
!! processed by numpydoc !!

property stiffness_matrix
!! processed by numpydoc !!

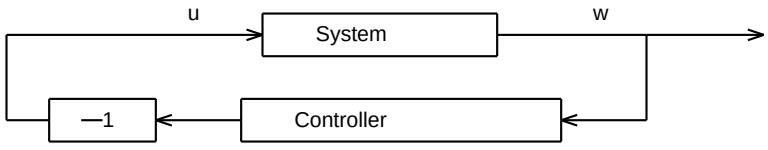
2.2.3 Closed Loop

Basis

class nlcontrol.closedloop.feedback.**ClosedLoop**(*system=None*, *controller=None*)

Bases: `object`

The object contains a closed loop configuration using BlockDiagram objects of the simupy module. The closed loop systems is given by the following block scheme:



Parameters

system [`Systembase` or `list` of `Systembase`] A state-full or state-less system. The number of inputs should be equal to the number of controller outputs.

controller [`ControllerBase` or `list` of `ControllerBase`] A state-full or state-less controller. The number of inputs should be equal to the number of system outputs.

Examples

- **Create a closed-loop object of SystemBase object 'sys', which uses the Euler-Lagrange formulation, and Contr**

```
>>> from nlcontrol import PID, DynamicController, EulerLagrange
>>> $
>>> # Define the system:
>>> states = 'x1, x2'
>>> inputs = 'u1, u2'
>>> sys = EulerLagrange(states, inputs)
>>> x1, x2, dx1, dx2, u1, u2, du1, du2 = sys.create_
↪variables(input_diffs=True)
>>> M = [[1, x1*x2],
    [x1*x2, 1]]
>>> C = [[2*dx1, 1 + x1],
    [x2 - 2, 3*dx2]]
>>> K = [x1, 2*x2]
>>> F = [u1, 0]
>>> sys.define_system(M, C, K, F)
>>> $
>>> # Define the DynamicController controller:
>>> st = 'z1, z2'
>>> dyn_contr = DynamicController(states=st, inputs=sys.minimal_
↪states)
>>> z1, z2, z1dot, z2dot, w, wdot = contr.create_variables()
>>> a0, a1, k1 = 12.87, 6.63, 0.45
>>> b0 = (48.65 - a1) * k1
>>> b1 = (11.79 - 1) * k1
>>> A = [[0, 1], [-a0, -a1]]
>>> B = [[0], [1]]
>>> C = [[b0], [b1]]
>>> f = lambda x: x**2
>>> eta = [[w + wdot], [(w + wdot)**2]]
>>> phi = [[z1], [z2dot]]
>>> contr.define_controller(A, B, C, f, eta, phi)
>>> $
>>> # Define the PID:
>>> kp = 1
>>> kd = 1
>>> ksi0 = [kp * x1, kp * x2]
>>> psi0 = [kd * dx1, kd * dx2]
>>> pid = PID(ksi0, None, psi0, inputs=sys.minimal_states)
>>> $
>>> # Create the controller:
>>> contr = dyn_contr.parallel(pid)
>>> $
>>> # Create a closed-loop object:
>>> CL = ClosedLoop(sys, contr)
```

Attributes

backward_system ControllerBase

forward_system Systembase

Methods

create_block_diagram([forward_systems, ...])	Create a closed loop block diagram with negative feedback.
create_closed_loop_system()	Create a SystemBase object of the closed-loop system.
linearize(working_point_states)	In many cases a nonlinear closed-loop system is observed around a certain working point.
simulation(tspan, initial_conditions[, ...])	Simulates the closed-loop in various conditions.

property backward_system
ControllerBase

The controller in the backward path of the closed loop.

create_block_diagram(*forward_systems: list = None*, *backward_systems: list = None*)
Create a closed loop block diagram with negative feedback. The loop contains a list of SystemBase objects in the forward path and ControllerBase objects in the backward path.

Parameters

forward_systems [list, optional (at least one system should be present in the loop)] A list of SystemBase objects. All input and output dimensions should match.

backward_systems: list, optional (at least one system should be present in the loop) A list of ControllerBase objects. All input and output dimensions should match.

Returns

BD [a simupy's BlockDiagram object] contains the configuration of the closed-loop.

indices [dict] information on the ranges of the states and outputs in the output vectors of a simulation dataset.

create_closed_loop_system()
Create a SystemBase object of the closed-loop system.

Returns

system [SystemBase] A Systembase object of the closed-loop system.

property forward_system
Systembase

The system in the forward path of the closed loop.

linearize(*working_point_states*)
In many cases a nonlinear closed-loop system is observed around a certain working point. In the state space close to this working point it is save to say that a linearized version of the nonlinear system is a sufficient approximation. The linearized model allows the user to use linear control techniques to examine the nonlinear system close to this working point. A first order Taylor expansion is used to obtain the linearized system. A working point for the states needs to be provided.

Parameters

working_point_states [list or int] the state equations are linearized around the working point of the states.

Returns

sys_lin: SystemBase object with the same states and inputs as the original system. The state and output equation is linearized.

sys_control: control.StateSpace object

Examples

- **Print the state equation of the linearized closed-loop object of `CL' around the state's working point x[1]**

```
>>> CL_lin, CL_control = CL.linearize([1, 5])
>>> print('Linearized state equation: ', CL_lin.state_
↪equation)
```

simulation(*tspan*, *initial_conditions*, *plot=False*, *custom_integrator_options=None*)

Simulates the closed-loop in various conditions. It is possible to impose initial conditions on the states of the system. The results of the simulation are numerically available. Also, a plot of the states and outputs is available. To simulate the system scipy's ode is used. # TODO: output_signal -> a disturbance on the output signal.

Parameters

tspan [float or list-like] the parameter defines the time vector for the simulation in seconds. An integer indicates the end time. A list-like object with two elements indicates the start and end time respectively. And more than two elements indicates at which time instances the system needs to be simulated.

initial_conditions [int, float, list-like object] the initial conditions of the states of a statefull system. If none is given, all are zero, default: None

plot [boolean, optional] the plot boolean decides whether to show a plot of the inputs, states, and outputs, default: False

custom_integrator_options [dict, optional (default: None)] Specify specific integrator options to pass to *integrator_class.set_integrator (scipy ode)*. The options are 'name', 'rtol', 'atol', 'nsteps', and 'max_step', which specify the integrator name, relative tolerance, absolute tolerance, number of steps, and maximal step size respectively. If no custom integrator options are specified the `DEFAULT_INTEGRATOR_OPTIONS` are used:

```
{
    "name": "dopri5",
    "rtol": 1e-6,
    "atol": 1e-12,
    "nsteps": 500,
    "max_step": 0.0
}
```

Returns

t [array] time vector

data [tuple] four data vectors, the states and the outputs of the systems in the forward path and the states and outputs of the systems in the backward path.

Examples

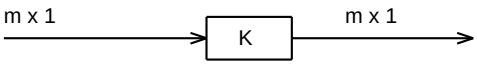
- **A simulation of 5 seconds of the statefull SystemBase object 'sys' in the forward path and the statefull Co**

```
>>> CL = ClosedLoop(sys, contr)
>>> t, data = CL.simulation(5, [x0_0, x1_0], custom_
↪integrator_options={'nsteps': 1000}, plot=True)
>>> (x_p, y_p, x_c, y_c) = data
```

Building blocks

nlcontrol.closedloop.blocks.**gain_block**(*value*, *dim*)

Multiply the output of system with dimension 'dim' with a contant value 'K'.



Parameters

value [int or float] Multiply the input signal with a value.

dim [int]

Returns

simupy's MemorylessSystem

Examples

A negative gain block with dimension 3:

```
>>> negative_feedback = gain_block(-1, 3)
```

CHAPTER

THREE

WANT TO CONTRIBUTE?

As the module is open-source, contributions are highly appreciated. If you are not feeling confident enough to dive into the code, just add an issue on the github page

3.1 Contribute - Git workflow

This is just a practical guide that can help you making contributions to the nlcontrol toolbox. It is very basic, so don't expect too much.

3.1.1 Commit message

Refer to a component name, give a short description, and add a reference to the issue, if relevant (with 'fix #<number>' it means it is fixed)

```
COMPONENT_NAME: fix *some_text* (fix #1234)

More details here...
```

3.1.2 Initiate your work repository

Fork the jjuch/nlcontrol from github UI, and then

```
git clone https://github.com/jjuch/nlcontrol.git
cd nlcontrol
git remote add my_user_name https://github.com/my_user_name/nlcontrol.git
```

3.1.3 Update your local master against upstream master

In command line do the following

```
git checkout master
git fetch origin
# Be careful: this will remove all local changes you might have done now
git reset --hard origin/master
```

3.1.4 Working with a feature branch

In command line do the following

```
git checkout master
(potentially update your local master against upstream, as described above)
git checkout -b my_new_feature_branch

# do something. For instance:
git add my_new_file
git add my_modified_message
git rm old_file
git commit -a

# you may need to resynchronize against master if you need some bugfix
# or new capability that has been added since you created your branch
git fetch origin
git rebase origin/master

# At end of your work, make sure history is reasonable by folding non
# significant commits into a consistent set
git rebase -i master (use 'fixup' for example to merge several commits together,
and 'reword' to modify commit messages)

# or alternatively, in case there is a big number of commits and marking
# all them as 'fixup' is tedious
git fetch origin
git rebase origin/master
git reset --soft origin/master
git commit -a -m "Put here the synthetic commit message"

# push your branch
git push my_user_name my_new_feature_branch
From GitHub UI, issue a pull request
```

If the pull request discussion checks 'requires changes', commit locally and push. To get a clean history, you may need to `git rebase -i master`, in which case you will have to force-push your branch with `git push -f my_user_name my_new_feature_branch`.

3.1.5 Things you should NOT do

(For anyone with push rights to https://github.com/jjuch/nlcontrol,) Never modify a commit or the history of anything that has been committed to https://github.com/jjuch/nlcontrol

Disclaimer: Thank you GDAL repo for the inspiration.

CHAPTER

FOUR

LICENSE

CHAPTER

FIVE

CONTACT

Email: Jasper_Juchem@hotmail.com

CHAPTER

SIX

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

n

INDEX

W