PyInduct Documentation

Release 0.5.1rc0

Stefan Ecklebe, Marcus Riesmeier

Sep 23, 2020

CONTENTS

Contents:

CHAPTER

ONE

PYINDUCT

PyInduct is a python toolbox for control and observer design for infinite dimensional systems.

- **Documentation**: https://pyinduct.readthedocs.org.
- **Bug Reports**: https://github.com/pyinduct/pyinduct/issues

PyInduct supports easy simulation of common distributed parameter systems using ready-to-go FEM implementations or custom modal approximations. With the included eigenfunctions for parabolic problems up to 2nd order or case-agnostic Lagrangian polynomials, automated controller and observer approximation routines are provided. The included visualization methods help verifying the controllers performance.

CHAPTER TWO

INSTALLATION

At the command line:

```
$ pip install pyinduct
```

Or, if you have virtualenvwrapper installed:

```
$ mkvirtualenv pyinduct
$ pip install pyinduct
```

CHAPTER

THREE

USAGE

To use PyInduct in a project we recommend:

```
import pyinduct as pi
```

CHAPTER

FOUR

BACKGROUND INFORMATION

4.1 Curing an Interval

All classes contained in this module can easily be used to cure a given interval. For example let's approximate the interval from $z = 0$ to $z = 1$ with 3 piecewise linear functions:

```
>>> from pyinduct import Domain, LagrangeFirstOder
>>> nodes = Domain(bounds0(0, 1), num=3)
>>> list(nodes)
[0.0, 0.5, 1.0]
>>> funcs = LagrangeFirstOrder.cure_interval(nodes)
```

4.2 Simulation

4.2.1 PDE Simulation Basics

Write something interesting here :-)

4.2.2 Multiple PDE Simulation

The aim of the class `CanonicalEquation` is to handle more than one pde. For one pde `CanonicalForm` would be sufficient. The simplest way to get the required N `CanonicalEquation`'s is to define your problem in N `WeakFormulation`'s and make use of `parse_weak_formulations()`. The thus obtained N `CanonicalEquation`'s you can pass to `create_state_space` to derive a state space representation of your multi pde system.

Each `CanonicalEquation` object hold one dominant `CanonicalForm` and at maximum $N-1$ other

`CanonicalForm`'s.

$$
\begin{aligned}
&\text{1st CanonicalForms object}\\
&\left.E_{1,n_1}\boldsymbol{x}_1^{*(n_1)}(t)+\cdots+E_{1,0}\boldsymbol{x}_1^{*(0)}(t)+\boldsymbol{f}_1+G_1\boldsymbol{u}(t)=0\right\}\text{dynamic CanonicalForm}\\
&\left.\begin{aligned}
H_{1:2,n_2-1}\boldsymbol{x}_2^{*(n_2-1)}(t)+\cdots+H_{1:2,0}\boldsymbol{x}_2^{*(0)}(t)&=0\\
&\vdots\\
H_{1:N,n_N-1}\boldsymbol{x}_N^{*(n_N-1)}(t)+\cdots+H_{1:N,0}\boldsymbol{x}_N^{*(0)}(t)&=0
\end{aligned}\right\}\text{N-1 static CanonicalForm's}\\
&\vdots\\
&\vdots\\
&\vdots\\
&N\text{th CanonicalForms object}\\
&\left.E_{N,n_N}\boldsymbol{x}_N^{*(n_N)}(t)+\cdots+E_{N,0}\boldsymbol{x}_N^{*(0)}(t)+\boldsymbol{f}_N+G_N\boldsymbol{u}(t)=0\right\}\text{dynamic CanonicalForm}\\
&\left.\begin{aligned}
H_{N:1,n_1-1}\boldsymbol{x}_1^{*(n_1-1)}(t)+\cdots+H_{N:1,0}\boldsymbol{x}_1^{*(0)}(t)&=0\\
&\vdots\\
H_{N:N-1,n_{N-1}-1}\boldsymbol{x}_{N-1}^{*(n_{N-1}-1)}(t)+\cdots+H_{N:N-1,0}\boldsymbol{x}_N^{*(0)}(t)&=0
\end{aligned}\right\}\text{N-1 static CanonicalForm's}
\end{aligned}
$$

They are interpreted as

$$
\begin{aligned}
0=&\,E_{1,n_1}\boldsymbol{x}_1^{*(n_1)}(t)+\cdots+E_{1,0}\boldsymbol{x}_1^{*(0)}(t)+\boldsymbol{f}_1+G_1\boldsymbol{u}(t)\\
&+H_{1:2,n_2-1}\boldsymbol{x}_2^{*(n_2-1)}(t)+\cdots+H_{1:2,0}\boldsymbol{x}_2^{*(0)}(t)+\cdots\\
&\cdots+H_{1:N,n_N-1}\boldsymbol{x}_N^{*(n_N-1)}(t)+\cdots+H_{1:N,0}\boldsymbol{x}_N^{*(0)}(t)\\
&\vdots\\
&\vdots\\
&\vdots\\
0=&\,E_{N,n_N}\boldsymbol{x}_N^{*(n_N)}(t)+\cdots+E_{N,0}\boldsymbol{x}_N^{*(0)}(t)+\boldsymbol{f}_N+G_N\boldsymbol{u}(t)\\
&+H_{N:1,n_1-1}\boldsymbol{x}_1^{*(n_1-1)}(t)+\cdots+H_{N:1,0}\boldsymbol{x}_1^{*(0)}(t)+\cdots\\
&\cdots+H_{N:N-1,n_{N-1}-1}\boldsymbol{x}_{N-1}^{*(n_{N-1}-1)}(t)+\cdots+H_{N:N-1,0}\boldsymbol{x}_{N-1}^{*(0)}(t).
\end{aligned}
$$

These N equations can simply expressed in a state space model

$$\dot{\boldsymbol{x}}^*(t)=A\boldsymbol{x}*(t)+B\boldsymbol{u}(t)+\boldsymbol{f}$$

with the weights vector

$$\boldsymbol{x}^{*^T}=\Big(\underbrace{0^T}_{\mathbb{R}^{\dim(\boldsymbol{x}_1^*)\times(n_1-1)}},\boldsymbol{x}_1^{*^T},\quad\ldots\quad,\underbrace{0^T}_{\mathbb{R}^{\dim(\boldsymbol{x}_N^*)\times(n_N-1)}},\boldsymbol{x}_N^{*^T}\Big).$$

CHAPTER FIVE

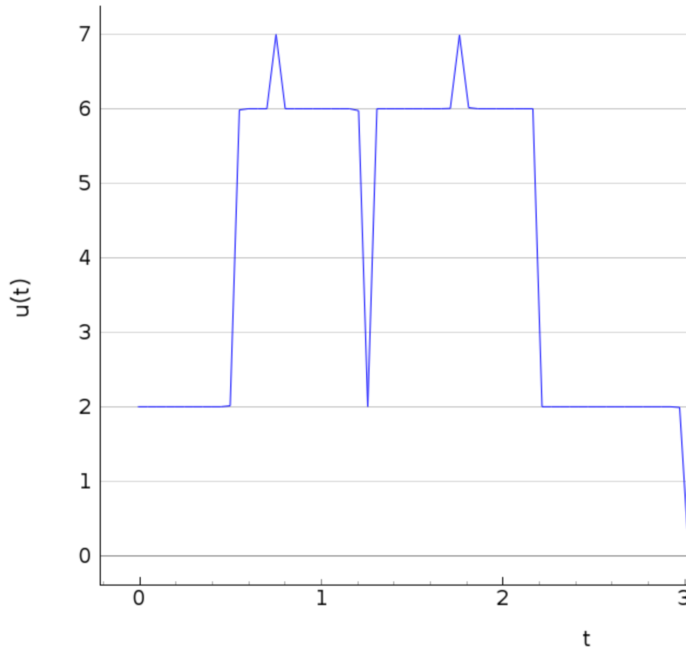
EXAMPLES

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

5.1 Transport System

$$\begin{aligned} \dot{x}(z,t) + v x'(z,t) &= 0 && z \in (0,l], t > 0 \\ x(z,0) &= x_0(z) && z \in [0,l] \\ x(0,t) &= u(t) && t > 0 \end{aligned}$$

- $x_0(z) = 0$



- $u(t)$ (**pyinduct.trajectory.SignalGenerator**):
- $x(z,t)$:
- source code:

```
import numpy as np
import pyinduct as pi
import pyqtgraph as pg


def run(show_plots):
    sys_name = 'transport system'
```

(continues on next page)

(continued from previous page)

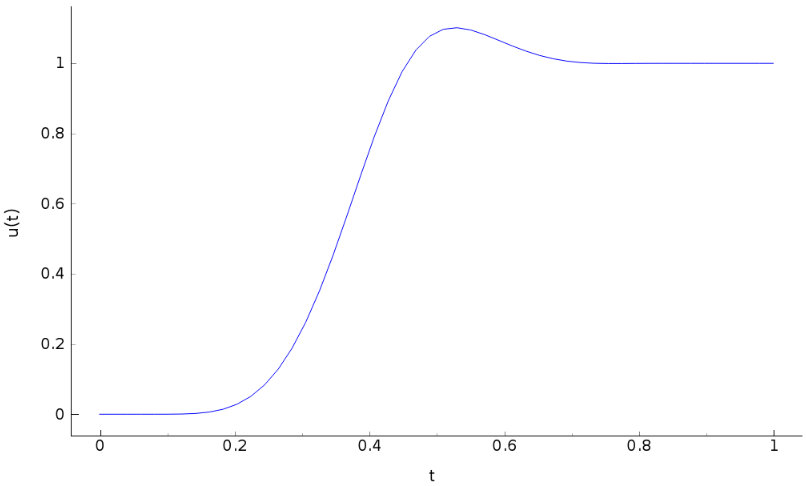
```
    v = 10
    l = 5
    T = 5
    spat_bounds = (0, l)
    spat_domain = pi.Domain(bounds=spat_bounds, num=51)
    temp_domain = pi.Domain(bounds=(0, T), num=100)

    init_x = pi.Function(lambda z: 0, domain=spat_bounds)

    init_funcs = pi.LagrangeFirstOrder.cure_interval(spat_domain)
    func_label = 'init_funcs'
    pi.register_base(func_label, init_funcs)

    u = pi.SimulationInputSum([
        pi.SignalGenerator('square', np.array(temp_domain), frequency=0.1,
                           scale=1, offset=1, phase_shift=1),
        pi.SignalGenerator('square', np.array(temp_domain), frequency=0.2,
                           scale=2, offset=2, phase_shift=2),
        pi.SignalGenerator('square', np.array(temp_domain), frequency=0.3,
                           scale=3, offset=3, phase_shift=3),
        pi.SignalGenerator('square', np.array(temp_domain), frequency=0.4,
                           scale=4, offset=4, phase_shift=4),
        pi.SignalGenerator('square', np.array(temp_domain), frequency=0.5,
                           scale=5, offset=5, phase_shift=5),
    ])

    x = pi.FieldVariable(func_label)
    phi = pi.TestFunction(func_label)
    weak_form = pi.WeakFormulation([
        pi.IntegralTerm(pi.Product(x.derive(temp_order=1), phi),
                        spat_bounds),
        pi.IntegralTerm(pi.Product(x, phi.derive(1)),
                        spat_bounds,
                        scale=-v),
        pi.ScalarTerm(pi.Product(x(l), phi(l)), scale=v),
        pi.ScalarTerm(pi.Product(pi.Input(u), phi(0)), scale=-v),
    ], name=sys_name)

    eval_data = pi.simulate_system(weak_form, init_x, temp_domain, spat_domain)
    pi.tear_down(labels=(func_label,))

    if show_plots:
        # pyqtgraph visualization
        win0 = pg.plot(np.array(eval_data[0].input_data[0]).flatten(),
                       u.get_results(eval_data[0].input_data[0]).flatten(),
                       labels=dict(left='u(t)', bottom='t'), pen='b')
        win0.showGrid(x=False, y=True, alpha=0.5)
        # vis.save_2d_pg_plot(win0, 'transport_system')
        win1 = pi.PgAnimatedPlot(eval_data,
                                 title=eval_data[0].name,
                                 save_pics=False,
                                 labels=dict(left='x(z,t)', bottom='z'))
        pi.show()


if __name__ == "__main__":
    run(True)
```

5.2 R.a.d. eq. with dirichlet b.c. (fem approximation)

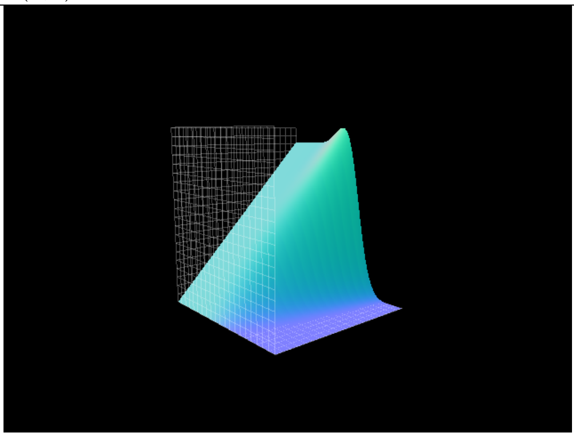
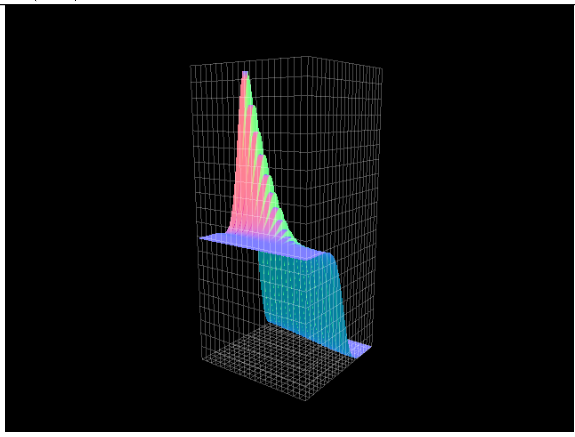
Simulation of the reaction-advection-diffusion equation with dirichlet boundary condition by $z = 0$ and dirichlet actuation by $z = l$.

$$\begin{aligned} \dot{x}(z,t) &= a_2 x''(z,t) + a_1 x'(z,t) + a_0 x(z,t) && z \in (0,l), t > 0 \\ x(z,0) &= x_0(z) && z \in [0,l] \\ x(0,t) &= 0 && t > 0 \\ x(l,t) &= u(t) && t > 0 \end{aligned}$$

- example: heat equation
  - $a_2 = 1, \quad a_1 = 0, \quad a_0 = 0, \quad x_0(z) = 0$
  - $u(t)$ –> `pyinduct.trajectory.RadTrajectory`



  - $x(z,t)$
  - $x'(z,t)$
  - corresponding 3d plots

$x(z,t)$	$x'(z,t)$
	

- with:
  - inital functions $\varphi_1(z), ..., \varphi_{n+1}(z)$

- test functions $\varphi_1(z), ..., \varphi_n(z)$
- where the functions $\varphi_1(z), .., \varphi_n(z)$ met the homogeneous b.c.

$$\varphi_1(l), .., \varphi_n(l) = \varphi_1(0), .., \varphi_n(0) = 0$$

- only φ_{n+1} can draw the actuation
- functions $\varphi_1(z), ..., \varphi_{n+1}(z)$ e.g. from type `pyinduct.shapefunctions.LagrangeFirstOrder` or `pyinduct.shapefunctions.LagrangeSecondOrder`, see `pyinduct.shapefunctions`

- approach:

$$x(z,t) = \sum_{i=1}^{n+1} x_i^*(t)\varphi_i(z)\Big|_{x_{n+1}^*=u} = \underbrace{\sum_{i=1}^{n} x_i^*(t)\varphi_i(z)}_{\hat{x}(z,t)} + \varphi_{n+1}(z)u(t)$$

- weak formulation...

$$\begin{aligned}\langle \dot{x}(z,t), \varphi_j(z)\rangle &= a_2\langle x''(z,t), \varphi_j(z)\rangle \\ &+ a_1\langle x'(z,t), \varphi_j(z)\rangle + a_0\langle x(z,t), \varphi_j(z)\rangle && j = 1, ..., n\end{aligned}$$

- ... and derivation shift to work with lagrange 1st order initial functions

$$\begin{aligned}\langle \dot{x}(z,t), \varphi_j(z)\rangle &= \overbrace{[a_2[x'(z,t)\varphi_j(z)]_0^l}^{=0} - a_2\langle x'(z,t), \varphi_j'(z)\rangle \\ &+ a_1\langle x'(z,t), \varphi_j(z)\rangle + a_0\langle x(z,t), \varphi_j(z)\rangle && j = 1, ..., n \\ \langle \dot{\hat{x}}(z,t), \varphi_j(z)\rangle + \langle \varphi_{N+1}(z), \varphi_j(z)\rangle \dot{u}(t) &= -a_2\langle \hat{x}'(z,t), \varphi_j'(z)\rangle - a_2\langle \varphi_{N+1}'(z), \varphi_j'(z)\rangle u(t) \\ &+ a_1\langle \hat{x}'(z,t), \varphi_j(z)\rangle + a_1\langle \varphi_{N+1}'(z), \varphi_j(z)\rangle u(t) + \\ &+ a_0\langle \hat{x}(z,t), \varphi_j(z)\rangle + a_0\langle \varphi_{N+1}(z), \varphi_j(z)\rangle u(t) && j = 1, ..., n\end{aligned}$$

- leads to state space model for the weights $\boldsymbol{x}^* = (x_1^*, ..., x_n^*)^T$:

$$\dot{\boldsymbol{x}}^*(t) = A\boldsymbol{x}^*(t) + \boldsymbol{b}_0 u(t) + \boldsymbol{b}_1 \dot{u}(t)$$

- input derivative elimination through the transformation:
  - $\bar{\boldsymbol{x}}^* = \tilde{A}\boldsymbol{x}^* - \boldsymbol{b}_1 u$
  - e.g.: $\tilde{A} = I$
  - leads to

$$\begin{aligned}\dot{\bar{\boldsymbol{x}}}^*(t) &= \tilde{A}A\tilde{A}^{-1}\bar{\boldsymbol{x}}^*(t) + \tilde{A}(A\boldsymbol{b}_1 + \boldsymbol{b}_0)u(t) \\ &= \bar{A}\bar{\boldsymbol{x}}^*(t) + \bar{\boldsymbol{b}}u(t)\end{aligned}$$

- source code:

```
import numpy as np
import pyinduct as pi
import pyinduct.parabolic as parabolic


def run(show_plots):
    n_fem = 17
    T = 1
```

(continues on next page)

(continued from previous page)

```
l = 1
y0 = -1
y1 = 4

param = [1, 0, 0, None, None]
# or try these:
# param = [1, -0.5, -8, None, None]   #  :)))
a2, a1, a0, _, _ = param

temp_domain = pi.Domain(bounds=(0, T), num=100)
spat_domain = pi.Domain(bounds=(0, l), num=n_fem * 11)

# initial and test functions
nodes = pi.Domain(spat_domain.bounds, num=n_fem)
fem_base = pi.LagrangeFirstOrder.cure_interval(nodes)
act_fem_base = pi.Base(fem_base[-1])
not_act_fem_base = pi.Base(fem_base[1:-1])
vis_fems_base = pi.Base(fem_base)

pi.register_base("act_base", act_fem_base)
pi.register_base("sim_base", not_act_fem_base)
pi.register_base("vis_base", vis_fems_base)

# trajectory
u = parabolic.RadFeedForward(l, T,
                             param_original=param,
                             bound_cond_type="dirichlet",
                             actuation_type="dirichlet",
                             y_start=y0, y_end=y1)

# weak form
x = pi.FieldVariable("sim_base")
x_dt = x.derive(temp_order=1)
x_dz = x.derive(spat_order=1)
phi = pi.TestFunction("sim_base")
phi_dz = phi.derive(1)
act_phi = pi.ScalarFunction("act_base")
act_phi_dz = act_phi.derive(1)

weak_form = pi.WeakFormulation([
    # ... of the homogeneous part of the system
    pi.IntegralTerm(pi.Product(x_dt, phi),
                    limits=spat_domain.bounds),
    pi.IntegralTerm(pi.Product(x_dz, phi_dz),
                    limits=spat_domain.bounds,
                    scale=a2),
    pi.IntegralTerm(pi.Product(x_dz, phi),
                    limits=spat_domain.bounds,
                    scale=-a1),
    pi.IntegralTerm(pi.Product(x, phi),
                    limits=spat_domain.bounds,
                    scale=-a0),

    # ... of the inhomogeneous part of the system
    pi.IntegralTerm(pi.Product(pi.Product(act_phi, phi),
                               pi.Input(u, order=1)),
                    limits=spat_domain.bounds),
    pi.IntegralTerm(pi.Product(pi.Product(act_phi_dz, phi_dz),
                               pi.Input(u)),
                    limits=spat_domain.bounds,
                    scale=a2),
```

(continues on next page)

(continued from previous page)

```
        pi.IntegralTerm(pi.Product(pi.Product(act_phi_dz, phi),
                                   pi.Input(u)),
                        limits=spat_domain.bounds,
                        scale=-a1),
        pi.IntegralTerm(pi.Product(pi.Product(act_phi, phi),
                                   pi.Input(u)),
                        limits=spat_domain.bounds,
                        scale=-a0)],
        name="main_system")

    # system matrices \dot x = A x + b0 u + b1 \dot u
    cf = pi.parse_weak_formulation(weak_form)
    ss = pi.create_state_space(cf)

    a_mat = ss.A[1]
    b0 = ss.B[0][1]
    b1 = ss.B[1][1]

    # transformation into \dot \bar x = \bar A \bar x + \bar b u
    a_tilde = np.diag(np.ones(a_mat.shape[0]), 0)
    a_tilde_inv = np.linalg.inv(a_tilde)

    a_bar = (a_tilde @ a_mat) @ a_tilde_inv
    b_bar = a_tilde @ (a_mat @ b1) + b0

    # simulation
    def x0(z):
        return 0 + y0 * z

    start_func = pi.Function(x0, domain=spat_domain.bounds)
    full_start_state = np.array([pi.project_on_base(start_func,
                                                    pi.get_base("vis_base")
                                                    )]).flatten()
    initial_state = full_start_state[1:-1]

    start_state_bar = a_tilde @ initial_state - (b1 * u(time=0)).flatten()
    ss = pi.StateSpace(a_bar, b_bar, base_lbl="sim", input_handle=u)
    sim_temp_domain, sim_weights_bar = pi.simulate_state_space(ss,
                                                               start_state_bar,
                                                               temp_domain)

    # back-transformation
    u_vec = np.reshape(u.get_results(sim_temp_domain), (len(temp_domain), 1))
    sim_weights = sim_weights_bar @ a_tilde_inv + u_vec @ b1.T

    # visualisation
    plots = list()
    save_pics = False
    vis_weights = np.hstack((np.zeros_like(u_vec), sim_weights, u_vec))

    eval_d = pi.evaluate_approximation("vis_base",
                                       vis_weights,
                                       sim_temp_domain,
                                       spat_domain,
                                       spat_order=0)
    der_eval_d = pi.evaluate_approximation("vis_base",
                                           vis_weights,
                                           sim_temp_domain,
                                           spat_domain,
                                           spat_order=1)
```

(continues on next page)

(continued from previous page)

```
    if show_plots:
        plots.append(pi.PgAnimatedPlot(eval_d,
                                        labels=dict(left='x(z,t)', bottom='z'),
                                        save_pics=save_pics))
        plots.append(pi.PgAnimatedPlot(der_eval_d,
                                        labels=dict(left='x\'(z,t)', bottom='z'),
                                        save_pics=save_pics))

        win1 = pi.surface_plot(eval_d, title="x(z,t)")
        win2 = pi.surface_plot(der_eval_d, title="x'(z,t)")

        # save pics
        if save_pics:
            path = pi.save_2d_pg_plot(u.get_plot(), 'rad_dirichlet_traj')[1]
            win1.gl_widget.grabFrameBuffer().save(path + 'rad_dirichlet_3d_x.png')
            win2.gl_widget.grabFrameBuffer().save(path + 'rad_dirichlet_3d_dx.png')
        pi.show()

    pi.tear_down(("act_base", "sim_base", "vis_base"))


if __name__ == "__main__":
    run(True)
```

5.3 Multiple pde example / pipe model

This example considers the thermal behavior (simulation) of plug flow of an incompressible fluid through a pipe from [BacEtAl17], which can be described with the normed variables/parameters:

- $x_1(z,t)$ ~ fluid temperature
- $x_2(z,t)$ ~ pipe wall temperature
- $x_3(z,t) = 0$ ~ ambient temperature
- $u(t)$ ~ system input
- $H(t)$ ~ heaviside step function
- v ~ fluid velocity
- c_1 ~ heat transfer coefficient (fluid - wall)
- c_2 ~ heat transfer coefficient (wall - ambient)

by the following equations:

$$\begin{aligned}
\dot{x}_1(z,t) + v x_1'(z,t) &= c_1(x_2(z,t) - x_1(z,t)), & z \in (0,l] \\
\dot{x}_2(z,t) &= c_1(x_1(z,t) - x_2(z,t)) + c_2(x_3(z,t) - x_2(z,t)), & z \in [0,l] \\
x_1(z,0) &= 0 \\
x_2(z,0) &= 0 \\
x_1(0,t) &= u(t) = 2H(t)
\end{aligned}$$

```
def run(show_plots):
    v = 10
    c1, c2 = [1, 1]
    l = 5
    T = 5
    spat_bounds = (0, l)
    spat_domain = pi.Domain(bounds=spat_bounds, num=51)
```

(continues on next page)

(continued from previous page)

```
    temp_domain = pi.Domain(bounds=(0, T), num=100)

    init_funcs1 = pi.LagrangeSecondOrder.cure_interval(spat_domain)
    nodes = pi.Domain(spat_domain.bounds, num=30)
    init_funcs2 = pi.LagrangeFirstOrder.cure_interval(nodes)
    pi.register_base("x1_funcs", init_funcs1)
    pi.register_base("x2_funcs", init_funcs2)

    u = pi.SimulationInputSum([
        pi.SignalGenerator('square', temp_domain, frequency=.03,
                           scale=2, offset=4, phase_shift=1),
    ])

    x1 = pi.FieldVariable("x1_funcs")
    psi1 = pi.TestFunction("x1_funcs")
    x2 = pi.FieldVariable("x2_funcs")
    psi2 = pi.TestFunction("x2_funcs")

    weak_form1 = pi.WeakFormulation(
        [
            pi.IntegralTerm(pi.Product(x1.derive(temp_order=1), psi1),
                            limits=spat_bounds),
            pi.IntegralTerm(pi.Product(x1, psi1.derive(1)),
                            limits=spat_bounds,
                            scale=-v),
            pi.ScalarTerm(pi.Product(x1(l), psi1(l)), scale=v),
            pi.ScalarTerm(pi.Product(pi.Input(u), psi1(0)), scale=-v),
            pi.IntegralTerm(pi.Product(x1, psi1),
                            limits=spat_bounds,
                            scale=c1),
            pi.IntegralTerm(pi.Product(x2, psi1),
                            limits=spat_bounds,
                            scale=-c1),
        ],
        name="fluid temperature"
    )
    weak_form2 = pi.WeakFormulation(
        [
            pi.IntegralTerm(pi.Product(x2.derive(temp_order=1), psi2),
                            limits=spat_bounds),
            pi.IntegralTerm(pi.Product(x1, psi2),
                            limits=spat_bounds,
                            scale=-c2),
            pi.IntegralTerm(pi.Product(x2, psi2),
                            limits=spat_bounds,
                            scale=c2 + c1),
        ],
        name="wall temperature"
    )

    ics = {weak_form1.name: [pi.Function(lambda z: np.sin(z/2),
                                         domain=spat_bounds)],
           weak_form2.name: [pi.Function(lambda z: 0, domain=spat_bounds)]}
    spat_domains = {weak_form1.name: spat_domain, weak_form2.name: spat_domain}
    evald1, evald2 = pi.simulate_systems([weak_form1, weak_form2],
                                         ics,
                                         temp_domain,
                                         spat_domains)
    pi.tear_down(["x1_funcs", "x2_funcs"])

    if show_plots:
```

(continues on next page)

(continued from previous page)

```
        win1 = pi.PgAnimatedPlot([evald1, evald2], labels=dict(bottom='z'))
        win3 = pi.surface_plot(evald1, title=weak_form1.name)
        win4 = pi.surface_plot(evald2, title=weak_form2.name)
        pi.show()


if __name__ == "__main__":
    run(True)
```

5.4 Simulation of the Euler-Bernoulli Beam

In this example, the hyperbolic equation of an euler bernoulli beam, clamped at one side is considered. The domain of the vertical beam excitation $x(z,t)$ is regarded to be $[0,1] \times \mathbb{R}^+$.

The governing equation reads:

$$\begin{aligned}\partial_t^2 x(z,t) &= -\frac{EI}{\mu}\partial_z^4 x(z,t)\\ x(0,t) &= 0\\ \partial_z x(0,t) &= 0\\ \partial_z^2 x(0,t) &= 0\\ \partial_z^3 x(0,t) &= u(t)\end{aligned}$$

With the E-module E, the second moment of area I and the specific density μ . In this example, the input $u(t)$ mimics the force impulse occurring if the beam is hit by a hammer.

5.4.1 Spatial disretization

For further analysis let $D_z(x) = -\frac{EI}{\mu}\partial_z^4 x$ denote the spatial operator and

$$R(x) = \begin{pmatrix} x(0,t)\\ \partial_z x(0,t)\\ \partial_z^2 x(1,t)\\ \partial_z^3 x(1,t)\end{pmatrix} = \mathbf{0}$$

denote the boundary operator.

Repeated partial integration of the expression

$$\begin{aligned}\langle D_z x|\varphi\rangle &= \frac{EI}{\mu}\left\langle \partial_z^4 x|y\right\rangle\\ &= \frac{EI}{\mu}\left(\left[\partial_z^3 x\varphi\right]_0^1 - \left[\partial_z^2 x\partial_z\varphi\right]_0^1\left[\partial_z^1 x\partial_z^2\varphi\right]_0^1 - \left[x\partial_z^3\varphi\right]_0^1\right)\\ &\quad + \frac{EI}{\mu}\left\langle x|\partial_z^4 y\right\rangle\end{aligned}$$

and application of the boundary conditions shows that $\langle D_z x|y\rangle = \langle x|D_z y\rangle$ if $Rx = R\varphi$. Therefore, the spatial operator is self-adjoint.

5.4.2 Modal Analysis

Since the operator is self-adjoined, the eigenvectors of the operator generate a orthonormal basis, which can be used for the approximation.

Hence, the problem to solve reads:

$$\frac{EI}{\nu}\partial_z^4\varphi(z,t) = \lambda\varphi(z,t)$$

Which is achieved by choosing

$$\begin{aligned}\varphi(z) = {} & \cos(\gamma z) - \cosh(\gamma z) \\ & - \frac{\left(e^{2\gamma} + 2e^{\gamma}\cos(\gamma) + 1\right)\sin(\gamma z)}{e^{2\gamma} + 2e^{\gamma}\sin(\gamma) - 1} \\ & + \frac{\left(e^{2\gamma} + 2e^{\gamma}\cos(\gamma) + 1\right)\sinh(\gamma z)}{e^{2\gamma} + 2e^{\gamma}\sin(\gamma) - 1}\end{aligned}$$

where $\gamma = \left(-\lambda\frac{\nu}{EI}\right)^{\frac{1}{4}}$. This is done in `calc_eigen()` .

Using this basis, the approximation

$$x(z,t) \approx \sum_{i=1}^{N} c_i(t)\varphi_i(z)$$

is introduced.

Projecting the equation on the basis of eigenvectors $\boldsymbol{\varphi}(z)$ yields

$$\left\langle \partial_t^2 x | \varphi_k \right\rangle = \langle D_z x | \varphi_k \rangle$$

for every $k = 1, \dots, N$. Substituting the approximation leads to

$$\left\langle \partial_t^2 x | \varphi_k \right\rangle = \sum_{i=1}^{N} c_i(t) \langle D_z \varphi_i | \varphi_k \rangle$$

where the application of D_z and the inner product can be swapped since D_z is a bounded operator. Finally, using the solution of the eigen problem yields

$$\left\langle \partial_t^2 x | \varphi_k \right\rangle = \sum_{i=1}^{N} c_i(t)\lambda_i \langle \varphi_i | \varphi_k \rangle$$

which simplifies to

$$\left\langle \partial_t^2 x | \varphi_k \right\rangle = c_k(t)\lambda_k$$

since, due to orthonormality, $\langle \varphi_i | \varphi_k \rangle$ is zero for all $i \neq k$ and 1 for $i = k$.

Performing the same steps for the left-hand side yields:

$$\ddot{c}_k(t) = \lambda_k c_k(t).$$

Thus, the ordinary differential equation system

$$\dot{\boldsymbol{b}}(t) = \begin{pmatrix} \boldsymbol{A} \\ \boldsymbol{\Lambda} \end{pmatrix} \boldsymbol{b}(t)$$

with the new state vector

$$\boldsymbol{b}(t) = \left(c_1(t), \dots, c_N(t), \dot{c}_1(t), \dots, \dot{c}_N(t)\right)^T$$

the integrator chain $\boldsymbol{A}$ and eigenvalue matrix $\boldsymbol{\Lambda} = \operatorname{diag}(\lambda_1, \dots, \lambda_N)$ is derived. Since the resulting system is autonomous, apart from interesting simulations, not much can be done fro a control perspective.

5.4.3 Alternative Variant

Using the weak formulation, which is gained by projecting the original equation on a set of test functions and fully shifting the spatial operator onto the test functions and substituting the boundary conditions

$$\begin{aligned}\langle D_z x|\varphi\rangle &= \frac{EI}{\mu}\left\langle \partial_z^4 x|y\right\rangle \\ &= \frac{EI}{\mu}\big(u(t)\varphi(1) - \partial_z^3 x(0)\varphi(0) + \partial_z^2 x(0)\partial_z\varphi(0) \\ &\quad + \partial_z x(1)\partial_z^2\varphi(1) - x(1)\partial_z^3\varphi(1) \\ &\quad + \left\langle x|\partial_z^4 y\right\rangle\big)\end{aligned}$$

and inserting the modal approximation from above, the system can be simulated for every arbitrary input $u(t)$. Note that this approximation converges over the whole spatial domain, but not punctually, since using the eigenvectors $\partial_z^3\varphi(1) = 0$ but $\partial_z^3 x(1) = u(t)$.

- source code:

```
"""
This example simulates an euler-bernoulli beam, please refer to the
documentation for an exhaustive explanation.
"""

import numpy as np
import sympy as sp
import pyinduct as pi
from matplotlib import pyplot as plt


class ImpulseExcitation(pi.SimulationInput):
    """
    Simulate that the free end of the beam is hit by a hammer
    """

    def _calc_output(self, **kwargs):
        t = kwargs["time"]
        a = 1/20
        value = 100 / (a * np.sqrt(np.pi)) * np.exp(-((t-1)/a)**2)
        return dict(output=value)


def calc_eigen(order, l_value, EI, mu, der_order=4, debug=False):
    r"""
    Solve the eigenvalue problem and return the eigenvectors

    Args:
        order: Approximation order.
        l_value: Length of the spatial domain.
        EI: Product of e-module and second moment of inertia.
        mu: Specific density.
        der_order: Required derivative order of the generated functions.

    Returns:
        pi.Base: Modal base.
    """
    C, D, E, F = sp.symbols("C D E F")
    gamma, l = sp.symbols("gamma l")
    z = sp.symbols("z")

    eig_func = (C*sp.cos(gamma*z)
                + D*sp.sin(gamma*z)
                + E*sp.cosh(gamma*z)
```

(continues on next page)

(continued from previous page)

```
                + F*sp.sinh(gamma*z))

    bcs = [eig_func.subs(z, 0),
           eig_func.diff(z, 1).subs(z, 0),
           eig_func.diff(z, 2).subs(z, l),
           eig_func.diff(z, 3).subs(z, l),
           ]
    e_sol = sp.solve(bcs[0], E)[0]
    f_sol = sp.solve(bcs[1], F)[0]
    new_bcs = [bc.subs([(E, e_sol), (F, f_sol)]) for bc in bcs[2:]]
    d_sol = sp.solve(new_bcs[0], D)[0]
    char_eq = new_bcs[1].subs([(D, d_sol), (l, l_value), (C, 1)])
    char_func = sp.lambdify(gamma, char_eq, modules="numpy")

    def char_wrapper(z):
        try:
            return char_func(z)
        except FloatingPointError:
            return 1

    grid = np.linspace(-1, 30, num=1000)
    roots = pi.find_roots(char_wrapper, grid, n_roots=order)
    if debug:
        pi.visualize_roots(roots, grid, char_func)

    # build eigenvectors
    eig_vec = eig_func.subs([(E, e_sol),
                             (F, f_sol),
                             (D, d_sol),
                             (l, l_value),
                             (C, 1)])

    # print(sp.latex(eig_vec))

    # build derivatives
    eig_vec_derivatives = [eig_vec]
    for i in range(der_order):
        eig_vec_derivatives.append(eig_vec_derivatives[-1].diff(z, 1))

    # construct functions
    eig_fractions = []
    for root in roots:
        # localize and lambdify
        callbacks = [sp.lambdify(z, vec.subs(gamma, root), modules="numpy")
                     for vec in eig_vec_derivatives]

        frac = pi.Function(domain=(0, l_value),
                           eval_handle=callbacks[0],
                           derivative_handles=callbacks[1:])
        frac.eigenvalue = - root**4 * EI / mu
        eig_fractions.append(frac)

    eig_base = pi.Base(eig_fractions)
    normed_eig_base = pi.normalize_base(eig_base)

    if debug:
        pi.visualize_functions(eig_base.fractions)
        pi.visualize_functions(normed_eig_base.fractions)

    return normed_eig_base
```

(continues on next page)

(continued from previous page)

```
def run(show_plots):
    sys_name = 'euler bernoulli beam'

    # domains
    spat_bounds = (0, 1)
    spat_domain = pi.Domain(bounds=spat_bounds, num=101)
    temp_domain = pi.Domain(bounds=(0, 10), num=1000)

    if 0:
        # physical properties
        height = .1  # [m]
        width = .1  # [m]
        e_module = 210e9  # [Pa]
        EI = 210e9 * (width * height**3)/12
        mu = 1e6  # [kg/m]
    else:
        # normed properties
        EI = 1e0
        mu = 1e0

    # define approximation bases
    if 0:
        # somehow, fem is still problematic
        approx_base = pi.LagrangeNthOrder.cure_interval(spat_domain,
                                                        order=4)
        approx_lbl = "complete_base"
    else:
        approx_base = calc_eigen(7, 1, EI, mu)
        approx_lbl = "eig_base"

    pi.register_base(approx_lbl, approx_base)

    # system definition

    u = ImpulseExcitation("Hammer")
    x = pi.FieldVariable(approx_lbl)
    phi = pi.TestFunction(approx_lbl)

    weak_form = pi.WeakFormulation([
        pi.ScalarTerm(pi.Product(pi.Input(u), phi(1)), scale=EI),
        pi.ScalarTerm(pi.Product(x.derive(spat_order=3)(0), phi(0)),
                      scale=-EI),
        pi.ScalarTerm(pi.Product(x.derive(spat_order=2)(0), phi.derive(1)(0)),
                      scale=EI),
        pi.ScalarTerm(pi.Product(x.derive(spat_order=1)(1), phi.derive(2)(1)),
                      scale=EI),
        pi.ScalarTerm(pi.Product(x(1), phi.derive(3)(1)),
                      scale=-EI),
        pi.IntegralTerm(pi.Product(x, phi.derive(4)),
                        spat_bounds,
                        scale=EI),
        pi.IntegralTerm(pi.Product(x.derive(temp_order=2), phi),
                        spat_bounds,
                        scale=mu),
    ], name=sys_name)

    # initial conditions
    init_form = pi.ConstantFunction(0, domain=spat_bounds)
    init_form_dt = pi.ConstantFunction(0, domain=spat_bounds)
    initial_conditions = [init_form, init_form_dt]
```

(continues on next page)

(continued from previous page)

```
    # simulation
    with np.errstate(under="ignore"):
        eval_data = pi.simulate_system(weak_form,
                                       initial_conditions,
                                       temp_domain,
                                       spat_domain,
                                       settings=dict(name="vode",
                                                     method="bdf",
                                                     order=5,
                                                     nsteps=1e8,
                                                     max_step=temp_domain.step))
    pi.tear_down([approx_lbl])

    # recover the input trajectory
    u_data = u.get_results(eval_data[0].input_data[0], as_eval_data=True)

    # visualization
    if show_plots:
        plt.plot(u_data.input_data[0], u_data.output_data)
        win1 = pi.PgAnimatedPlot(eval_data,
                                 labels=dict(left='x(z,t)', bottom='z'))
        pi.show()


if __name__ == "__main__":
    run(True)
```

5.5 Simulation with observer based state feedback of the reaction-convection-diffusion equation

Implementation of the approximation scheme presented in [RW2018b]. The system

$$
\begin{aligned}
\dot{x}(z,t) &= a_2 x''(z,t) + a_1 x'(z,t) + a_0 x(z,t) \\
x'(0,t) &= \alpha x(0,t) \\
x'(1,t) &= -\beta x(1,t) + u(t)
\end{aligned}
$$

and the observer

$$
\begin{aligned}
\dot{\hat{x}}(z,t) &= a_2 \hat{x}''(z,t) + a_1 \hat{x}'(z,t) + a_0 \hat{x}(z,t) + l(z)\tilde{y}(t) \\
\hat{x}'(0,t) &= \alpha \hat{x}(0,t) + l_0 \tilde{y}(t) \\
\hat{x}'(1,t) &= -\beta \hat{x}(1,t) + u(t)
\end{aligned}
$$

are approximated with `LagrangeFirstOrder` (FEM) shapefunctions and the backstepping controller and observer are approximated with the eigenfunctions respectively the adjoint eigenfunction of the system operator, see [RW2018b].

Note: For now, only `a0 = 0` and `a0_t_o = 0` are supported, because of some limitations of the automatic observer gain transformation, see `evaluate_transformations()` docstring.

References

class ReversedRobinEigenfunction(*om*, *param*, *l*, *scale=1*, *max_der_order=2*)

Bases: `pyinduct.SecondOrderRobinEigenfunction`

This class provides an eigenfunction $\varphi(z)$ to the eigenvalue problem given by

$$a_2\varphi''(z) + a_1\varphi'(z) + a_0\varphi(z) = \lambda\varphi(z)$$
$$\varphi'(0) = \alpha\varphi(0)$$
$$\varphi'(l) = -\beta\varphi(l).$$

The eigenfrequency $\omega = \sqrt{-\frac{a_1^2}{4a_2^2} + \frac{a_0-\lambda}{a_2}}$ must be provided (for example with the *eigfreq_eigval_hint()* of this class).

Parameters

- **om** (*numbers.Number*) – eigenfrequency ω
- **param** (*array_like*) – $\left(a_2, a_1, a_0, \alpha, \beta\right)^T$
- **l** (*numbers.Number*) – End of the domain $z \in [0, l]$.
- **scale** (*numbers.Number*) – Factor to scale the eigenfunctions (corresponds to $\varphi(0) =$ phi_0).
- **max_der_order** (*int*) – Number of derivative handles that are needed.

static eigfreq_eigval_hint(*param*, *l*, *n_roots*, *show_plot=False*)

Return the first *n_roots* eigenfrequencies ω and eigenvalues λ.

$$\omega_i = \sqrt{-\frac{a_1^2}{4a_2^2} + \frac{a_0 - \lambda_i}{a_2}} \quad i = 1, \dots, \text{n\_roots}$$

to the considered eigenvalue problem.

Parameters

- **param** (*array_like*) – Parameters $\left(a_2, a_1, a_0, \alpha, \beta\right)^T$
- **l** (*numbers.Number*) – Right boundary value of the domain $[0, l] \ni z$.
- **n_roots** (*int*) – Amount of eigenfrequencies to compute.
- **show_plot** (*bool*) – Show a plot window of the characteristic equation.

Returns

$$\left(\left[\omega_1, \dots, \omega_{\text{n\_roots}}\right], \left[\lambda_1, \dots, \lambda_{\text{n\_roots}}\right]\right)$$

Return type tuple –> booth tuple elements are numpy.ndarrays of length *nroots*

function_handle_factory(*self*, *old_handle*, *l*, *der_order=0*)

approximate_observer(*obs_params*, *sys_params*, *sys_domain*, *sys_lbl*, *obs_sys_lbl*, *test_lbl*, *tar_test_lbl*, *system_input*)

run(*show_plots*)

5.6 Simulation with observer based state feedback of the string with mass model

5.6.1 Simulation environment

Simulation of the string with mass example, with flatness based state feedback and flatness based state observer (design + approximation), presented in [RW2018a].

References

class FlatString(*y0*, *y1*, *z0*, *z1*, *t0*, *dt*, *params*)

Bases: `pyinduct.simulation.SimulationInput`

Flatness based feedforward for the "string with mass" model.

The flat output y of this system is given by the mass position at $z = z_0$. This output will be transferred from *y0* to *y1* starting at *t0*, lasting *dt* seconds.

Parameters

- **y0** (*float*) – Initial value for the flat output.
- **y1** (*float*) – Final value for the flat output.
- **z0** (*float*) – Position of the flat output (left side of the string).
- **z1** (*float*) – Position of the actuation (right side of the string).
- **t0** (*float*) – Time to start the transfer.
- **dt** (*float*) – Duration of the transfer.
- **params** (*bunch*) – Structure containing the physical parameters: * m: the mass * tau: the * sigma: the strings tension

class Parameters

class PgDataPlot(*data*)

Bases: `pyinduct.visualization.DataPlot`, `pyqtgraph.QtCore.QObject`

Base class for all pyqtgraph plotting related classes.

class SecondOrderFeedForward(*desired_handle*)

Bases: `pyinduct.examples.string_with_mass.system.pi.SimulationInput`

Base class for all objects that want to act as an input for the time-step simulation.

The calculated values for each time-step are stored in internal memory and can be accessed by `get_results()` (after the simulation is finished).

Note: Due to the underlying solver, this handle may get called with time arguments, that lie outside of the specified integration domain. This should not be a problem for a feedback controller but might cause problems for a feedforward or trajectory implementation.

class SwmBaseCanonicalFraction(*functions*, *scalars*)

Bases: `pyinduct.ComposedFunctionVector`

Implementation of composite function vector $\boldsymbol{x}$.

$$\boldsymbol{x} = \begin{pmatrix} x_1(z) \\ \vdots \\ x_n(z) \\ \xi_1 \\ \vdots \\ \xi_m \end{pmatrix}$$

derive (*self*, *order*)

Basic implementation of derive function.

Empty implementation, overwrite to use this functionality. For an example implementation see `Function`

Parameters **order** (`numbers.Number`) – derivative order

Returns derived object

Return type `BaseFraction`

evaluation_hint (*self*, *values*)

If evaluation can be accelerated by using special properties of a function, this function can be overwritten to performs that computation. It gets passed an array of places where the caller wants to evaluate the function and should return an array of the same length, containing the results.

Note: This implementation just calls the normal evaluation hook.

Parameters **values** – places to be evaluated at

Returns Evaluation results.

Return type numpy.ndarray

get_member (*self*, *idx*)

Getter function to access members. Empty function, overwrite to implement custom functionality. For an example implementation see `Function`

Note: Empty function, overwrite to implement custom functionality.

Parameters **idx** – member index

static scalar_product (*left*, *right*)

scalar_product_hint (*self*)

Scalar product for the canonical form of the string with mass system:

Returns Scalar product function handle wrapped inside a list.

Return type list(callable)

class SwmBaseFraction (*functions*, *scalars*)

Bases: `pyinduct.ComposedFunctionVector`

Implementation of composite function vector $\boldsymbol{x}$.

$$\boldsymbol{x} = \begin{pmatrix} x_1(z) \\ \vdots \\ x_n(z) \\ \xi_1 \\ \vdots \\ \xi_m \end{pmatrix}$$

derive (*self*, *order*)

Basic implementation of derive function.

Empty implementation, overwrite to use this functionality. For an example implementation see *Function*

Parameters **order** (numbers.Number) – derivative order

Returns derived object

Return type *BaseFraction*

evaluation_hint (*self*, *values*)

If evaluation can be accelerated by using special properties of a function, this function can be overwritten to performs that computation. It gets passed an array of places where the caller wants to evaluate the function and should return an array of the same length, containing the results.

Note: This implementation just calls the normal evaluation hook.

Parameters **values** – places to be evaluated at

Returns Evaluation results.

Return type numpy.ndarray

get_member (*self*, *idx*)

Getter function to access members. Empty function, overwrite to implement custom functionality. For an example implementation see *Function*

Note: Empty function, overwrite to implement custom functionality.

Parameters **idx** – member index

l2_scalar_product = True

static scalar_product (*left*, *right*)

scalar_product_hint (*self*)

Scalar product for the string with mass system:

$$\langle x, y\rangle = \int_0^1 (x_1'(z)y_1'(z) + x_2(z)y_2(z)\,dz + x_3y_3 + mx_4y_4$$

Returns Scalar product function handle wrapped inside a list.

Return type list(callable)

class SwmObserverError (*control_law*, *smooth=None*)

Bases: `pyinduct.examples.string_with_mass.system.pi.StateFeedback`

For a smooth fade-in of the observer error.

Parameters

- **control_law** (*WeakFormulation*) – Function handle that calculates the control output if provided with correct weights.
- **smooth** (*array-like*) – Arguments for *SmoothTransition*

class SwmPgAnimatedPlot (*data*, *title=''*, *refresh_time=40*, *replay_gain=1*, *save_pics=False*, *create_video=False*, *labels=None*)

Bases: *pyinduct.visualization.PgDataPlot*

Animation for the string with mass example. Compare with *PgAnimatedPlot*.

Parameters

- **data** ((iterable of) *EvalData*) – results to animate
- **title** (*basestring*) – window title
- **refresh_time** (*int*) – time in msec to refresh the window must be greater than zero
- **replay_gain** (*float*) – values above 1 acc- and below 1 decelerate the playback process, must be greater than zero
- **save_pics** (*bool*) –
- **labels** –

Return:

exported_files (*self*)

alpha = 0

apply_control_mode (*sys_fem_lbl, sys_modal_lbl, obs_fem_lbl, obs_modal_lbl, mode*)

approximate_controller (*sys_lbl, modal_lbl*)

build_canonical_weak_formulation (*obs_lbl, spatial_domain, u, obs_err, name='system'*)

Observer canonical form of the string with mass example

$$\begin{aligned}\dot{x}_1(t) &= \frac{2}{m}u(t)\\ \dot{x}_2(t) &= x_1(t) + \frac{2}{m}u(t)\\ \dot{x}_3(z,t) &= -x_3'(z,t) - \frac{2}{m}(1-h(z))zu(t) - m^{-1}y(t)\end{aligned}$$

Boundary condition

$$x_3(-1,t) = x_2(t) - y(t)$$

Weak formulation

$$\begin{aligned}-\langle \dot{x}(z,t), \psi(z)\rangle &= \frac{2}{m}u(t)\psi_1 + \frac{2}{m}u(t)\psi_2 + x_1\psi_2 - x_3(1,t)\psi_3(1) - m^{-1}\langle y(t), \psi_3(z)\rangle\\ &+ \underbrace{x_3(-1,t)\psi_3(-1)}_{x_2(t)\psi_3(-1)-y(t)\psi_3(-1)} + \langle x_3(z,t), \psi_3'(z)\rangle + \frac{2}{m}\langle (1-h(z))z, \psi_3(z)\rangle u(t)\end{aligned}$$

Output equation

$$x_3(1,t) = y(t)$$

Parameters

- **sys_approx_label** (*string*) – Shapefunction label for system approximation.
- **obs_approx_label** (*string*) – Shapefunction label for observer approximation.
- **input_vector** (*pyinduct.simulation.SimulationInputVector*) – Holds the input variable.
- **params** – Python class with the members:
  - *m* (mass)
  - *k1_ob*, *k2_ob*, *alpha_ob* (observer parameters)

Returns Observer

Return type `pyinduct.simulation.Observer`

build_controller (*sys_lbl*, *ctrl_lbl*)

The control law from [Woi2012] (equation 29)

$$u(t) = -\frac{1-\alpha}{1+\alpha} x_2(1) + \frac{(1-mk_1)\bar{y}'(1) - \alpha(1+mk_1)\bar{y}'(-1)}{1+\alpha} - \frac{mk_0}{1+\alpha}(\bar{y}(1) + \alpha\bar{y}(-1))$$

is simply tipped off in this function, whereas

$$\bar{y}(\theta) = \begin{cases} \xi_1 + m(1-e^{-\theta/m})\xi_2 + \int_0^\theta (1-e^{-(\theta-\tau)/m})(x_1'(\tau) + x_2(\tau))\,dz & \forall \quad \theta \in [-1, 0) \\ \xi_1 + m(e^{\theta/m} - 1)\xi_2 + \int_0^\theta (e^{(\theta-\tau)/m} - 1)(x_1'(-\tau) - x_2(-\tau))\,dz & \forall \quad \theta \in [0, 1] \end{cases}$$

$$\bar{y}'(\theta) = \begin{cases} e^{-\theta/m}\xi_2 + \frac{1}{m}\int_0^\theta e^{-(\theta-\tau)/m}(x_1'(\tau) + x_2(\tau))\,dz & \forall \quad \theta \in [-1, 0) \\ e^{\theta/m}\xi_2 + \frac{1}{m}\int_0^\theta e^{(\theta-\tau)/m}(x_1'(-\tau) - x_2(-\tau))\,dz & \forall \quad \theta \in [0, 1]. \end{cases}$$

Parameters **approx_label** (*string*) – Shapefunction label for approximation.

Returns Control law

Return type *StateFeedback*

build_fem_bases (*base_lbl*, *n1*, *n2*, *cf_base_lbl*, *ncf*, *modal_base_lbl*)

build_modal_bases (*base_lbl*, *n*, *cf_base_lbl*, *ncf*)

build_original_weak_formulation (*sys_lbl*, *spatial_domain*, *u*, *name='system'*)

Projection (see `SwmBaseFraction.scalar_product_hint()`

$$\langle \dot{x}(z,t), \psi(z) \rangle = \langle x_2(z,t), \psi_1(z) \rangle + \langle x_1''(z,t), \psi_2(z) \rangle + \xi_2(t)\psi_3 + x_1'(0)\psi_4$$

Boundary conditions

$$x_1(0,t) = \xi_1(t), \qquad u(t) = x_1'(1,t)$$

Implemented

$$\langle \dot{x}(z,t), \psi(z) \rangle = \langle x_2(z,t), \psi_1(z) \rangle + \langle x_1'(z,t), \psi_2'(z) \rangle + u(t)\psi_2(1) - x_1'(0,t)\psi_2(0) + \xi_2(t)\psi_3 + x_1'(0)\psi_4$$

Parameters

- **sys_lbl** (*str*) – Base label
- **spatial_domain** (*Domain*) – Spatial domain of the system.
- **name** (*str*) – Name of the system.

Returns *WeakFormulation*

check_eigenvalues (*sys_fem_lbl*, *obs_fem_lbl*, *obs_modal_lbl*, *ceq*, *ss*)

ctrl_gain

find_eigenvalues (*n*)

flatness_based_controller (*x2_plus1*, *y_bar_plus1*, *y_bar_minus1*, *dz_y_bar_plus1*, *dz_y_bar_minus1*, *name*)

get_colors (*cnt*, *scheme='tab10'*, *samples=10*)

Create a list of colors.

Parameters

- **cnt** (*int*) – Number of colors in the list.
- **scheme** (*str*) – Mpl color scheme to use.
- **samples** (*cnt*) – Number of samples to take from the scheme before starting from the beginning.

Returns List of *np.Array* holding the rgb values.

get_modal_base_for_ctrl_approximation()

get_primal_eigenvector(*according_paper=False*)

init_observer_gain(*sys_fem_lbl*, *sys_modal_lbl*, *obs_fem_lbl*, *obs_modal_lbl*)

integrate_function(*func*, *interval*)

Numerically integrate a function on a given interval using `complex_quadrature()`.

Parameters

- **func** (`callable`) – Function to integrate.
- **interval** (`list of tuples`) – List of (start, end) values of the intervals to integrate on.

Returns (Result of the Integration, errors that occurred during the integration).

Return type tuple

k0 = 2

k1 = 2

m = 1

obs_gain

ocf_inverse_state_transform(*org_state*)

Transformation of the the state $x(z,t) = (x(z,t), \dot{x}(z,t), x(0,t), \dot{x}(0,t))^T = (x_1(z,t), x_2(z,t), \xi_1(t), \xi_2(t))^T$ into the coordinates of the observer canonical form

$$\begin{aligned}
\bar{x}_1(t) &= w_2'(1) \\
\bar{x}_2(t) &= w_1'(1) + w_2'(1) \\
\bar{x}_3(\theta,t) &= \frac{1}{2}(w_2(1-\theta) + w_1'(1-\theta)), \quad \forall \theta > 0 \\
\bar{x}_3(\theta,t) &= \frac{1}{2}(w_2(1+\theta) - w_1'(1+\theta)) + w_1'(1) - \theta w_2'(1), \quad \forall \theta \le 0 \\
w_i(z) &= 2\int_0^z \left(\xi_i + \frac{1}{m}\int_0^\zeta x_i(\bar{\zeta})\,d\bar{\zeta}\right) d\zeta, \quad i = 1, 2.
\end{aligned}$$

Parameters **org_state** (`SwmBaseFraction`) – State

Returns Transformation

Return type `SwmBaseCanonicalFraction`

param

plot_eigenvalues(*eigenvalues*, *return_figure=False*)

pprint(*expression='\n\n\n'*)

register_evp_base(*base_lbl*, *eigenvectors*, *sp_var*, *domain*)

run(*show_plots*)

scale_equation_term_list(*eqt_list*, *factor*)

Temporary function, as long `EquationTerm` can only be scaled individually.

Parameters

- **eqt_list** (`list`) – List of `EquationTerm`'s
- **factor** (`numbers.Number`) – Scale factor.

Returns Scaled copy of `EquationTerm`'s (eqt_list).

sigma = 1

sort_eigenvalues (*eigenvalues*)

subs_list = [None]

sym

tau = 1

5.6.2 Weak formulations and definition of the bases

class Parameters

class PgDataPlot (*data*)

Bases: `pyinduct.visualization.DataPlot`, `pyqtgraph.QtCore.QObject`

Base class for all pyqtgraph plotting related classes.

class SwmBaseCanonicalFraction (*functions*, *scalars*)

Bases: `pyinduct.ComposedFunctionVector`

Implementation of composite function vector $\boldsymbol{x}$.

$$\boldsymbol{x} = \begin{pmatrix} x_1(z) \\ \vdots \\ x_n(z) \\ \xi_1 \\ \vdots \\ \xi_m \end{pmatrix}$$

derive (*self*, *order*)

Basic implementation of derive function.

Empty implementation, overwrite to use this functionality. For an example implementation see `Function`

Parameters **order** (`numbers.Number`) – derivative order

Returns derived object

Return type `BaseFraction`

evaluation_hint (*self*, *values*)

If evaluation can be accelerated by using special properties of a function, this function can be overwritten to performs that computation. It gets passed an array of places where the caller wants to evaluate the function and should return an array of the same length, containing the results.

Note: This implementation just calls the normal evaluation hook.

Parameters **values** – places to be evaluated at

Returns Evaluation results.

Return type numpy.ndarray

get_member (*self*, *idx*)

Getter function to access members. Empty function, overwrite to implement custom functionality. For an example implementation see `Function`

Note: Empty function, overwrite to implement custom functionality.

Parameters **idx** – member index

static scalar_product (*left*, *right*)

scalar_product_hint (*self*)
Scalar product for the canonical form of the string with mass system:

Returns Scalar product function handle wrapped inside a list.

Return type list(callable)

class SwmBaseFraction (*functions*, *scalars*)
Bases: `pyinduct.ComposedFunctionVector`

Implementation of composite function vector $\boldsymbol{x}$.

$$\boldsymbol{x} = \begin{pmatrix} x_1(z) \\ \vdots \\ x_n(z) \\ \xi_1 \\ \vdots \\ \xi_m \end{pmatrix}$$

derive (*self*, *order*)
Basic implementation of derive function.

Empty implementation, overwrite to use this functionality. For an example implementation see *Function*

Parameters order (`numbers.Number`) – derivative order

Returns derived object

Return type *BaseFraction*

evaluation_hint (*self*, *values*)
If evaluation can be accelerated by using special properties of a function, this function can be overwritten to performs that computation. It gets passed an array of places where the caller wants to evaluate the function and should return an array of the same length, containing the results.

Note: This implementation just calls the normal evaluation hook.

Parameters values – places to be evaluated at

Returns Evaluation results.

Return type numpy.ndarray

get_member (*self*, *idx*)
Getter function to access members. Empty function, overwrite to implement custom functionality. For an example implementation see *Function*

Note: Empty function, overwrite to implement custom functionality.

Parameters idx – member index

l2_scalar_product = True

static scalar_product (*left*, *right*)

scalar_product_hint (*self*)
Scalar product for the string with mass system:

$$\langle x, y \rangle = \int_0^1 (x_1'(z)y_1'(z) + x_2(z)y_2(z)\,dz + x_3y_3 + mx_4y_4$$

Returns Scalar product function handle wrapped inside a list.

Return type list(callable)

class SwmPgAnimatedPlot(*data*, *title=''*, *refresh_time=40*, *replay_gain=1*, *save_pics=False*, *create_video=False*, *labels=None*)

Bases: `pyinduct.visualization.PgDataPlot`

Animation for the string with mass example. Compare with `PgAnimatedPlot`.

Parameters

- **data** ((iterable of) `EvalData`) – results to animate
- **title** (*basestring*) – window title
- **refresh_time** (*int*) – time in msec to refresh the window must be greater than zero
- **replay_gain** (*float*) – values above 1 acc- and below 1 decelerate the playback process, must be greater than zero
- **save_pics** (*bool*) –
- **labels** –

Return:

exported_files(*self*)

alpha = 0

build_canonical_weak_formulation(*obs_lbl*, *spatial_domain*, *u*, *obs_err*, *name='system'*)

Observer canonical form of the string with mass example

$$\begin{aligned}\dot{x}_1(t) &= \frac{2}{m}u(t)\\ \dot{x}_2(t) &= x_1(t) + \frac{2}{m}u(t)\\ \dot{x}_3(z,t) &= -x_3'(z,t) - \frac{2}{m}(1-h(z))zu(t) - m^{-1}y(t)\end{aligned}$$

Boundary condition

$$x_3(-1,t) = x_2(t) - y(t)$$

Weak formulation

$$\begin{aligned}-\langle\dot{x}(z,t),\psi(z)\rangle &= \frac{2}{m}u(t)\psi_1 + \frac{2}{m}u(t)\psi_2 + x_1\psi_2 - x_3(1,t)\psi_3(1) - m^{-1}\langle y(t),\psi_3(z)\rangle\\ &+ \underbrace{x_3(-1,t)\psi_3(-1)}_{x_2(t)\psi_3(-1)-y(t)\psi_3(-1)} + \langle x_3(z,t),\psi_3'(z)\rangle + \frac{2}{m}\langle(1-h(z))z,\psi_3(z)\rangle u(t)\end{aligned}$$

Output equation

$$x_3(1,t) = y(t)$$

Parameters

- **sys_approx_label** (*string*) – Shapefunction label for system approximation.
- **obs_approx_label** (*string*) – Shapefunction label for observer approximation.
- **input_vector** (`pyinduct.simulation.SimulationInputVector`) – Holds the input variable.
- **params** – Python class with the members:
  - *m* (mass)
  - *k1_ob*, *k2_ob*, *alpha_ob* (observer parameters)

Returns Observer

Return type `pyinduct.simulation.Observer`

build_fem_bases (*base_lbl*, *n1*, *n2*, *cf_base_lbl*, *ncf*, *modal_base_lbl*)

build_modal_bases (*base_lbl*, *n*, *cf_base_lbl*, *ncf*)

build_original_weak_formulation (*sys_lbl*, *spatial_domain*, *u*, *name='system'*)

Projection (see `SwmBaseFraction.scalar_product_hint()`

$$\langle \dot{x}(z,t), \psi(z) \rangle = \langle x_2(z,t), \psi_1(z) \rangle + \langle x_1''(z,t), \psi_2(z) \rangle + \xi_2(t)\psi_3 + x_1'(0)\psi_4$$

Boundary conditions

$$x_1(0,t) = \xi_1(t), \qquad u(t) = x_1'(1,t)$$

Implemented

$$\begin{aligned}\langle \dot{x}(z,t), \psi(z) \rangle =& \langle x_2(z,t), \psi_1(z) \rangle + \langle x_1'(z,t), \psi_2'(z) \rangle \\ &+ u(t)\psi_2(1) - x_1'(0,t)\psi_2(0) + \xi_2(t)\psi_3 + x_1'(0)\psi_4\end{aligned}$$

Parameters

- **sys_lbl** (*str*) – Base label
- **spatial_domain** (*Domain*) – Spatial domain of the system.
- **name** (*str*) – Name of the system.

Returns *WeakFormulation*

check_eigenvalues (*sys_fem_lbl*, *obs_fem_lbl*, *obs_modal_lbl*, *ceq*, *ss*)

ctrl_gain

find_eigenvalues (*n*)

get_colors (*cnt*, *scheme='tab10'*, *samples=10*)

Create a list of colors.

Parameters

- **cnt** (*int*) – Number of colors in the list.
- **scheme** (*str*) – Mpl color scheme to use.
- **samples** (*cnt*) – Number of samples to take from the scheme before starting from the beginning.

Returns List of *np.Array* holding the rgb values.

get_modal_base_for_ctrl_approximation ()

get_primal_eigenvector (*according_paper=False*)

integrate_function (*func*, *interval*)

Numerically integrate a function on a given interval using `complex_quadrature()`.

Parameters

- **func** (*callable*) – Function to integrate.
- **interval** (*list of tuples*) – List of (start, end) values of the intervals to integrate on.

Returns (Result of the Integration, errors that occurred during the integration).

Return type tuple

k0 = 2

k1 = 2

m = 1

obs_gain

param

plot_eigenvalues (*eigenvalues*, *return_figure=False*)

pprint (*expression='\n\n\n'*)

register_evp_base (*base_lbl*, *eigenvectors*, *sp_var*, *domain*)

sigma = 1

sort_eigenvalues (*eigenvalues*)

subs_list = [None]

sym

tau = 1

5.6.3 State feedback control

class Parameters

class PgDataPlot (*data*)

Bases: `pyinduct.visualization.DataPlot`, `pyqtgraph.QtCore.QObject`

Base class for all pyqtgraph plotting related classes.

class SecondOrderFeedForward (*desired_handle*)

Bases: `pyinduct.examples.string_with_mass.system.pi.SimulationInput`

Base class for all objects that want to act as an input for the time-step simulation.

The calculated values for each time-step are stored in internal memory and can be accessed by `get_results()` (after the simulation is finished).

Note: Due to the underlying solver, this handle may get called with time arguments, that lie outside of the specified integration domain. This should not be a problem for a feedback controller but might cause problems for a feedforward or trajectory implementation.

---

class SwmBaseCanonicalFraction (*functions*, *scalars*)

Bases: `pyinduct.ComposedFunctionVector`

Implementation of composite function vector $\boldsymbol{x}$.

$$\boldsymbol{x} = \begin{pmatrix} x_1(z) \\ \vdots \\ x_n(z) \\ \xi_1 \\ \vdots \\ \xi_m \end{pmatrix}$$

derive (*self*, *order*)

Basic implementation of derive function.

Empty implementation, overwrite to use this functionality. For an example implementation see `Function`

Parameters order (`numbers.Number`) – derivative order

Returns derived object

Return type `BaseFraction`

`evaluation_hint` (*self*, *values*)

If evaluation can be accelerated by using special properties of a function, this function can be overwritten to performs that computation. It gets passed an array of places where the caller wants to evaluate the function and should return an array of the same length, containing the results.

Note: This implementation just calls the normal evaluation hook.

Parameters **`values`** – places to be evaluated at

Returns Evaluation results.

Return type numpy.ndarray

`get_member` (*self*, *idx*)

Getter function to access members. Empty function, overwrite to implement custom functionality. For an example implementation see *`Function`*

Note: Empty function, overwrite to implement custom functionality.

Parameters **`idx`** – member index

`static scalar_product` (*left*, *right*)

`scalar_product_hint` (*self*)

Scalar product for the canonical form of the string with mass system:

Returns Scalar product function handle wrapped inside a list.

Return type list(callable)

`class SwmBaseFraction` (*functions*, *scalars*)

Bases: `pyinduct.ComposedFunctionVector`

Implementation of composite function vector $\boldsymbol{x}$.

$$\boldsymbol{x} = \begin{pmatrix} x_1(z) \\ \vdots \\ x_n(z) \\ \xi_1 \\ \vdots \\ \xi_m \end{pmatrix}$$

`derive` (*self*, *order*)

Basic implementation of derive function.

Empty implementation, overwrite to use this functionality. For an example implementation see *`Function`*

Parameters **`order`** (`numbers.Number`) – derivative order

Returns derived object

Return type *`BaseFraction`*

`evaluation_hint` (*self*, *values*)

If evaluation can be accelerated by using special properties of a function, this function can be overwritten to performs that computation. It gets passed an array of places where the caller wants to evaluate the function and should return an array of the same length, containing the results.

Note: This implementation just calls the normal evaluation hook.

Parameters **values** – places to be evaluated at

Returns Evaluation results.

Return type numpy.ndarray

get_member (*self*, *idx*)

Getter function to access members. Empty function, overwrite to implement custom functionality. For an example implementation see *Function*

Note: Empty function, overwrite to implement custom functionality.

Parameters **idx** – member index

l2_scalar_product = True

static scalar_product (*left*, *right*)

scalar_product_hint (*self*)

Scalar product for the string with mass system:

$$\langle x, y \rangle = \int_0^1 (x_1'(z) y_1'(z) + x_2(z) y_2(z)\, dz + x_3 y_3 + m x_4 y_4$$

Returns Scalar product function handle wrapped inside a list.

Return type list(callable)

class SwmObserverError (*control_law*, *smooth=None*)

Bases: `pyinduct.examples.string_with_mass.system.pi.StateFeedback`

For a smooth fade-in of the observer error.

Parameters

- **control_law** (*WeakFormulation*) – Function handle that calculates the control output if provided with correct weights.
- **smooth** (*array-like*) – Arguments for *SmoothTransition*

class SwmPgAnimatedPlot (*data*, *title=''*, *refresh_time=40*, *replay_gain=1*, *save_pics=False*, *create_video=False*, *labels=None*)

Bases: *pyinduct.visualization.PgDataPlot*

Animation for the string with mass example. Compare with *PgAnimatedPlot*.

Parameters

- **data** ((iterable of) *EvalData*) – results to animate
- **title** (*basestring*) – window title
- **refresh_time** (*int*) – time in msec to refresh the window must be greater than zero
- **replay_gain** (*float*) – values above 1 acc- and below 1 decelerate the playback process, must be greater than zero
- **save_pics** (*bool*) –
- **labels** –

Return:

exported_files (*self*)

alpha = 0

apply_control_mode (*sys_fem_lbl*, *sys_modal_lbl*, *obs_fem_lbl*, *obs_modal_lbl*, *mode*)

approximate_controller (*sys_lbl*, *modal_lbl*)

build_canonical_weak_formulation (*obs_lbl*, *spatial_domain*, *u*, *obs_err*, *name='system'*)
Observer canonical form of the string with mass example

$$\dot{x}_1(t) = \frac{2}{m}u(t)$$
$$\dot{x}_2(t) = x_1(t) + \frac{2}{m}u(t)$$
$$\dot{x}_3(z,t) = -x_3'(z,t) - \frac{2}{m}(1-h(z))zu(t) - m^{-1}y(t)$$

Boundary condition

$$x_3(-1,t) = x_2(t) - y(t)$$

Weak formulation

$$\begin{aligned} -\langle \dot{x}(z,t), \psi(z)\rangle = &\frac{2}{m}u(t)\psi_1 + \frac{2}{m}u(t)\psi_2 + x_1\psi_2 - x_3(1,t)\psi_3(1) - m^{-1}\langle y(t), \psi_3(z)\rangle \\ &+ \underbrace{x_3(-1,t)\psi_3(-1)}_{x_2(t)\psi_3(-1)-y(t)\psi_3(-1)} + \langle x_3(z,t), \psi_3'(z)\rangle + \frac{2}{m}\langle (1-h(z))z, \psi_3(z)\rangle u(t) \end{aligned}$$

Output equation

$$x_3(1,t) = y(t)$$

Parameters

- **sys_approx_label** (*string*) – Shapefunction label for system approximation.
- **obs_approx_label** (*string*) – Shapefunction label for observer approximation.
- **input_vector** (*pyinduct.simulation.SimulationInputVector*) – Holds the input variable.
- **params** – Python class with the members:
  - *m* (mass)
  - *k1_ob*, *k2_ob*, *alpha_ob* (observer parameters)

Returns Observer

Return type `pyinduct.simulation.Observer`

build_controller (*sys_lbl*, *ctrl_lbl*)
The control law from [Woi2012] (equation 29)

$$\begin{aligned} u(t) = &-\frac{1-\alpha}{1+\alpha}x_2(1) + \frac{(1-mk_1)\bar{y}'(1) - \alpha(1+mk_1)\bar{y}'(-1)}{1+\alpha} \\ &-\frac{mk_0}{1+\alpha}(\bar{y}(1) + \alpha\bar{y}(-1)) \end{aligned}$$

is simply tipped off in this function, whereas

$$\bar{y}(\theta) = \begin{cases} \xi_1 + m(1-e^{-\theta/m})\xi_2 + \int_0^\theta (1-e^{-(\theta-\tau)/m})(x_1'(\tau) + x_2(\tau))\,dz & \forall \quad \theta \in [-1,0) \\ \xi_1 + m(e^{\theta/m}-1)\xi_2 + \int_0^\theta (e^{(\theta-\tau)/m}-1)(x_1'(-\tau) - x_2(-\tau))\,dz & \forall \quad \theta \in [0,1] \end{cases}$$
$$\bar{y}'(\theta) = \begin{cases} e^{-\theta/m}\xi_2 + \frac{1}{m}\int_0^\theta e^{-(\theta-\tau)/m}(x_1'(\tau) + x_2(\tau))\,dz & \forall \quad \theta \in [-1,0) \\ e^{\theta/m}\xi_2 + \frac{1}{m}\int_0^\theta e^{(\theta-\tau)/m}(x_1'(-\tau) - x_2(-\tau))\,dz & \forall \quad \theta \in [0,1]. \end{cases}$$

Parameters **approx_label** (*string*) – Shapefunction label for approximation.

Returns Control law

Return type *StateFeedback*

build_fem_bases (*base_lbl*, *n1*, *n2*, *cf_base_lbl*, *ncf*, *modal_base_lbl*)

build_modal_bases (*base_lbl*, *n*, *cf_base_lbl*, *ncf*)

build_original_weak_formulation (*sys_lbl*, *spatial_domain*, *u*, *name='system'*)
Projection (see `SwmBaseFraction.scalar_product_hint()`

$$\langle \dot{x}(z,t), \psi(z) \rangle = \langle x_2(z,t), \psi_1(z) \rangle + \langle x_1''(z,t), \psi_2(z) \rangle + \xi_2(t)\psi_3 + x_1'(0)\psi_4$$

Boundary conditions

$$x_1(0,t) = \xi_1(t), \qquad u(t) = x_1'(1,t)$$

Implemented

$$\begin{aligned}\langle \dot{x}(z,t), \psi(z) \rangle =& \langle x_2(z,t), \psi_1(z) \rangle + \langle x_1'(z,t), \psi_2'(z) \rangle \\ &+ u(t)\psi_2(1) - x_1'(0,t)\psi_2(0) + \xi_2(t)\psi_3 + x_1'(0)\psi_4\end{aligned}$$

Parameters

- **sys_lbl** (`str`) – Base label
- **spatial_domain** (`Domain`) – Spatial domain of the system.
- **name** (`str`) – Name of the system.

Returns `WeakFormulation`

check_eigenvalues (*sys_fem_lbl*, *obs_fem_lbl*, *obs_modal_lbl*, *ceq*, *ss*)

ctrl_gain

find_eigenvalues (*n*)

flatness_based_controller (*x2_plus1*, *y_bar_plus1*, *y_bar_minus1*, *dz_y_bar_plus1*, *dz_y_bar_minus1*, *name*)

get_colors (*cnt*, *scheme='tab10'*, *samples=10*)
Create a list of colors.

Parameters

- **cnt** (`int`) – Number of colors in the list.
- **scheme** (`str`) – Mpl color scheme to use.
- **samples** (`cnt`) – Number of samples to take from the scheme before starting from the beginning.

Returns List of *np.Array* holding the rgb values.

get_modal_base_for_ctrl_approximation ()

get_primal_eigenvector (*according_paper=False*)

init_observer_gain (*sys_fem_lbl*, *sys_modal_lbl*, *obs_fem_lbl*, *obs_modal_lbl*)

integrate_function (*func*, *interval*)
Numerically integrate a function on a given interval using `complex_quadrature()`.

Parameters

- **func** (`callable`) – Function to integrate.
- **interval** (`list of tuples`) – List of (start, end) values of the intervals to integrate on.

Returns (Result of the Integration, errors that occurred during the integration).

Return type tuple

k0 = 2

k1 = 2

m = 1

obs_gain

ocf_inverse_state_transform (*org_state*)

Transformation of the the state $x(z,t) = (x(z,t), \dot{x}(z,t), x(0,t), \dot{x}(0,t))^T = (x_1(z,t), x_2(z,t), \xi_1(t), \xi_2(t))^T$ into the coordinates of the observer canonical form

$$\begin{aligned}
\bar{x}_1(t) &= w_2'(1) \\
\bar{x}_2(t) &= w_1'(1) + w_2'(1) \\
\bar{x}_3(\theta, t) &= \frac{1}{2}(w_2(1-\theta) + w_1'(1-\theta)), \quad \forall \theta > 0 \\
\bar{x}_3(\theta, t) &= \frac{1}{2}(w_2(1+\theta) - w_1'(1+\theta)) + w_1'(1) - \theta w_2'(1), \quad \forall \theta \le 0 \\
w_i(z) &= 2\int_0^z \left(\xi_i + \frac{1}{m}\int_0^\zeta x_i(\bar{\zeta})\, d\bar{\zeta}\right) d\zeta, \quad i = 1, 2.
\end{aligned}$$

Parameters **org_state** (`SwmBaseFraction`) – State

Returns Transformation

Return type `SwmBaseCanonicalFraction`

param

plot_eigenvalues (*eigenvalues, return_figure=False*)

pprint (*expression='\n\n\n'*)

register_evp_base (*base_lbl, eigenvectors, sp_var, domain*)

scale_equation_term_list (*eqt_list, factor*)

Temporary function, as long `EquationTerm` can only be scaled individually.

Parameters

- **eqt_list** (*list*) – List of `EquationTerm`'s
- **factor** (*numbers.Number*) – Scale factor.

Returns Scaled copy of `EquationTerm`'s (eqt_list).

sigma = 1

sort_eigenvalues (*eigenvalues*)

subs_list = [None]

sym

tau = 1

5.6.4 Definition of the system parameters and some example related useful tools

class Parameters

class PgDataPlot (*data*)

Bases: `pyinduct.visualization.DataPlot`, `pyqtgraph.QtCore.QObject`

Base class for all pyqtgraph plotting related classes.

class SwmPgAnimatedPlot (*data, title='', refresh_time=40, replay_gain=1, save_pics=False, create_video=False, labels=None*)

Bases: `pyinduct.visualization.PgDataPlot`

Animation for the string with mass example. Compare with `PgAnimatedPlot`.

Parameters

- **data** ((iterable of) *EvalData*) – results to animate
- **title** (*basestring*) – window title
- **refresh_time** (*int*) – time in msec to refresh the window must be greater than zero
- **replay_gain** (*float*) – values above 1 acc- and below 1 decelerate the playback process, must be greater than zero
- **save_pics** (*bool*) –
- **labels** –

Return:

exported_files (*self*)

alpha = 0

alpha = 0

check_eigenvalues (*sys_fem_lbl*, *obs_fem_lbl*, *obs_modal_lbl*, *ceq*, *ss*)

ctrl_gain

find_eigenvalues (*n*)

get_colors (*cnt*, *scheme='tab10'*, *samples=10*)
Create a list of colors.

Parameters

- **cnt** (*int*) – Number of colors in the list.
- **scheme** (*str*) – Mpl color scheme to use.
- **samples** (*cnt*) – Number of samples to take from the scheme before starting from the beginning.

Returns List of *np.Array* holding the rgb values.

get_primal_eigenvector (*according_paper=False*)

k0 = 90

k0 = 90

k1 = 100

k1 = 100

m = 1

obs_gain

param

plot_eigenvalues (*eigenvalues*, *return_figure=False*)

pprint (*expression='\n\n\n'*)

sigma = 1

sort_eigenvalues (*eigenvalues*)

subs_list = [None]

sym

tau = 1

CHAPTER

SIX

CONTRIBUTING

Contributions are welcome, and they are greatly appreciated! Every little bit helps, and credit will always be given.

You can contribute in many ways:

6.1 Types of Contributions

6.1.1 Report Bugs

Report bugs at https://github.com/pyinduct/pyinduct/issues.

If you are reporting a bug, please include:

- Your operating system name and version.
- Any details about your local setup that might be helpful in troubleshooting.
- Detailed steps to reproduce the bug.

6.1.2 Fix Bugs

Look through the GitHub issues for bugs. Anything tagged with "bug" is open to whoever wants to implement it.

6.1.3 Implement Features

Look through the GitHub issues for features. Anything tagged with "feature" is open to whoever wants to implement it.

6.1.4 Write Documentation

PyInduct could always use more documentation, whether as part of the official PyInduct docs, in docstrings, or even on the web in blog posts, articles, and such.

6.1.5 Submit Feedback

The best way to send feedback is to file an issue at https://github.com/pyinduct/pyinduct/issues.

If you are proposing a feature:

- Explain in detail how it would work.
- Keep the scope as narrow as possible, to make it easier to implement.
- Remember that this is a volunteer-driven project, and that contributions are welcome :)

6.2 Get Started!

Ready to contribute? Here's how to set up *pyinduct* for local development.

1. Fork the *pyinduct* repo on GitHub.
2. Clone your fork locally:

   ```
   $ git clone git@github.com:your_name_here/pyinduct.git
   ```

3. Install your local copy into a virtualenv. Assuming you have virtualenvwrapper installed, this is how you set up your fork for local development:

   ```
   $ mkvirtualenv pyinduct
   $ cd pyinduct/
   $ python setup.py develop
   ```

4. Create a branch for local development:

   ```
   $ git checkout -b name-of-your-bugfix-or-feature
   ```

   Now you can make your changes locally.

5. When you're done making changes, check that your changes pass flake8 and the tests, including testing other Python versions with tox:

   ```
   $ flake8 pyinduct tests
   $ python setup.py test
   $ tox
   ```

   To get flake8 and tox, just pip install them into your virtualenv.

6. Commit your changes and push your branch to GitHub:

   ```
   $ git add .
   $ git commit -m "Your detailed description of your changes."
   $ git push origin name-of-your-bugfix-or-feature
   ```

7. Submit a pull request through the GitHub website.

6.3 Pull Request Guidelines

Before you submit a pull request, check that it meets these guidelines:

1. The pull request should include tests.
2. If the pull request adds functionality, the docs should be updated. Put your new functionality into a function with a docstring, and add the feature to the list in README.rst.
3. The pull request should work for Python 3.5, and for PyPy. Check on https://travis-ci.org/pyinduct/pyinduct/pull_requests whether all tests have passed.

6.4 Tips

Run a subset of tests with:

```
$ python -m unittest -v pyinduct/tests/test_<module_name>.py
```

or all tests with:

```
$ python -m unittest discover -v pyinduct/tests/
```

respectively:

```
$ python setup.py test
```

from project root.

CHAPTER

SEVEN

PYINDUCT MODULES REFERENCE

Because every feature of PyInduct must have a test case, when you are not sure how to use something, just look into the `tests/` directories, find that feature and read the tests for it, that will tell you everything you need to know.

Most of the things are already documented though in this document, that is automatically generated using PyInduct's docstrings.

Click the "modules" (modindex) link in the top right corner to easily access any PyInduct module, or use this table of contents:

7.1 Core

In the Core module you can find all basic classes and functions which form the backbone of the toolbox.

class ApproximationBasis

Base class for an approximation basis.

An approximation basis is formed by some objects on which given distributed variables may be projected.

abstract function_space_hint (*self*)

Hint that returns properties that characterize the functional space of the fractions. It can be used to determine if function spaces match.

Note: Overwrite to implement custom functionality.

is_compatible_to (*self*, *other*)

Helper functions that checks compatibility between two approximation bases.

In this case compatibility is given if the two bases live in the same function space.

Parameters other (`Approximation Base`) – Approximation basis to compare with.

Returns: True if bases match, False if they do not.

abstract scalar_product_hint (*self*)

Hint that returns steps for scalar product calculation with elements of this base.

Note: Overwrite to implement custom functionality.

class Base (*fractions*, *matching_base_lbls=None*, *intermediate_base_lbls=None*)

Bases: `pyinduct.core.ApproximationBasis`

Base class for approximation bases.

In general, a `Base` is formed by a certain amount of `BaseFractions` and therefore forms finite-dimensional subspace of the distributed problem's domain. Most of the time, the user does not need to interact with this class.

Parameters

- **`fractions`** (iterable of `BaseFraction`) – List, array or dict of `BaseFraction`'s
- **`matching_base_lbls`** (*`list of str`*) – List of labels from exactly matching bases, for which no transformation is necessary. Useful for transformations from bases that 'live' in different function spaces but evolve with the same time dynamic/coefficients (e.g. modal bases).
- **`intermediate_base_lbls`** (*`list of str`*) – If it is certain that this base instance will be asked (as destination base) to return a transformation to a source base, whose implementation is cumbersome, its label can be provided here. This will trigger the generation of the transformation using build-in features. The algorithm, implemented in `get_weights_transformation` is then called again with the intermediate base as destination base and the 'old' source base. With this technique arbitrary long transformation chains are possible, if the provided intermediate bases again define intermediate bases.

`derive` (*self*, *order*)
Basic implementation of derive function. Empty implementation, overwrite to use this functionality.

Parameters **`order`** (`numbers.Number`) – derivative order

Returns derived object

Return type *`Base`*

`function_space_hint` (*self*)
Hint that returns properties that characterize the functional space of the fractions. It can be used to determine if function spaces match.

Note: Overwrite to implement custom functionality.

`get_attribute` (*self*, *attr*)
Retrieve an attribute from the fractions of the base.

Parameters **`attr`** (*`str`*) – Attribute to query the fractions for.

Returns Array of `len(fractions)` holding the attributes. With *None* entries if the attribute is missing.

Return type `np.ndarray`

`raise_to` (*self*, *power*)
Factory method to obtain instances of this base, raised by the given power.

Parameters **`power`** – power to raise the basis onto.

`scalar_product_hint` (*self*)
Hint that returns steps for scalar product calculation with elements of this base.

Note: Overwrite to implement custom functionality.

`scale` (*self*, *factor*)
Factory method to obtain instances of this base, scaled by the given factor.

Parameters **`factor`** – factor or function to scale this base with.

`transformation_hint` (*self*, *info*)
Method that provides a information about how to transform weights from one `BaseFraction` into another.

In Detail this function has to return a callable, which will take the weights of the source- and return the weights of the target system. It may have keyword arguments for other data which is required to perform the transformation. Information about these extra keyword arguments should be provided in form of a dictionary whose keys are keyword arguments of the returned transformation handle.

Note: This implementation covers the most basic case, where the two *BaseFraction*'s are of same type. For any other case it will raise an exception. Overwrite this Method in your implementation to support conversion between bases that differ from yours.

Parameters **info** – *TransformationInfo*

Raises **NotImplementedError** –

Returns Transformation handle

class BaseFraction(*members*)

Abstract base class representing a basis that can be used to describe functions of several variables.

abstract add_neutral_element(*self*)

Return the neutral element of addition for this object.

In other words: *self + ret_val == self*.

derive(*self, order*)

Basic implementation of derive function.

Empty implementation, overwrite to use this functionality. For an example implementation see *Function*

Parameters **order** (numbers.Number) – derivative order

Returns derived object

Return type *BaseFraction*

evaluation_hint(*self, values*)

If evaluation can be accelerated by using special properties of a function, this function can be overwritten to performs that computation. It gets passed an array of places where the caller wants to evaluate the function and should return an array of the same length, containing the results.

Note: This implementation just calls the normal evaluation hook.

Parameters **values** – places to be evaluated at

Returns Evaluation results.

Return type numpy.ndarray

function_space_hint(*self*)

Empty Hint that can return properties which uniquely define the function space of the *BaseFraction*.

Note: Overwrite to implement custom functionality. For an example implementation see *Function*.

---

abstract get_member(*self, idx*)

Getter function to access members. Empty function, overwrite to implement custom functionality. For an example implementation see *Function*

Note: Empty function, overwrite to implement custom functionality.

Parameters **idx** – member index

abstract mul_neutral_element (*self*)
Return the neutral element of multiplication for this object.

In other words: *self * ret_val == self*.

raise_to (*self*, *power*)
Raises this fraction to the given *power*.

Parameters **power** (`numbers.Number`) – power to raise the fraction onto

Returns raised fraction

scalar_product_hint (*self*)
Empty Hint that can return steps for scalar product calculation.

Note: Overwrite to implement custom functionality. For an example implementation see *Function*

---

abstract scale (*self*, *factor*)
Factory method to obtain instances of this base fraction, scaled by the given factor. Empty function, overwrite to implement custom functionality. For an example implementation see *Function*.

Parameters **factor** – Factor to scale the vector.

class ComposedFunctionVector (*functions*, *scalars*)
Bases: *pyinduct.core.BaseFraction*

Implementation of composite function vector $\boldsymbol{x}$.

$$\boldsymbol{x} = \begin{pmatrix} x_1(z) \\ \vdots \\ x_n(z) \\ \xi_1 \\ \vdots \\ \xi_m \end{pmatrix}$$

add_neutral_element (*self*)
Create neutral element of addition that is compatible to this object.

Returns: Comp. Function Vector with constant functions returning 0 and scalars of value 0.

function_space_hint (*self*)
Return the hint that this function is an element of the an scalar product space which is uniquely defined by

- the scalar product `ComposedFunctionVector.scalar_product()`
- `len(self.members["funcs"])` functions
- and `len(self.members["scalars"])` scalars.

get_member (*self*, *idx*)
Getter function to access members. Empty function, overwrite to implement custom functionality. For an example implementation see *Function*

Note: Empty function, overwrite to implement custom functionality.

Parameters **idx** – member index

mul_neutral_element (*self*)
Create neutral element of multiplication that is compatible to this object.

Returns: Comp. Function Vector with constant functions returning 1 and scalars of value 1.

scalar_product_hint (*self*)
Empty Hint that can return steps for scalar product calculation.

Note: Overwrite to implement custom functionality. For an example implementation see `Function`

scale (*self*, *factor*)
Factory method to obtain instances of this base fraction, scaled by the given factor. Empty function, overwrite to implement custom functionality. For an example implementation see `Function`.

Parameters **factor** – Factor to scale the vector.

class ConstantComposedFunctionVector (*func_constants*, *scalar_constants*, ***func_kwargs*)
Bases: `pyinduct.core.ComposedFunctionVector`

Constant composite function vector $\boldsymbol{x}$.

$$\boldsymbol{x} = \begin{pmatrix} z \mapsto x_1(z) = c_1 \\ \vdots \\ z \mapsto x_n(z) = c_n \\ d_1 \\ \vdots \\ c_n \end{pmatrix}$$

Parameters

- **func_constants** (*array-like*) – Constants for the functions.
- **scalar_constants** (*array-like*) – The scalar constants.
- ****func_kwargs** – Keyword args that are passed to the ConstantFunction.

class ConstantFunction (*constant*, ***kwargs*)
Bases: `pyinduct.core.Function`

A `Function` that returns a constant value.

This function can be differentiated without limits.

Parameters **constant** (*number*) – value to return

Keyword Arguments ****kwargs** – All other kwargs get passed to `Function`.

derive (*self*, *order=1*)
Spatially derive this `Function`.

This is done by neglecting *order* derivative handles and to select handle order -1 as the new evaluation_handle.

Parameters **order** (*int*) – the amount of derivations to perform

Raises

- **TypeError** – If *order* is not of type int.
- **ValueError** – If the requested derivative order is higher than the provided one.

Returns `Function` the derived function.

class Domain (*bounds=None*, *num=None*, *step=None*, *points=None*)
Bases: `object`

Helper class that manages ranges for data evaluation, containing parameters.

Parameters

- **bounds** (*tuple*) – Interval bounds.
- **num** (*int*) – Number of points in interval.

- **step** (*numbers.Number*) – Distance between points (if homogeneous).
- **points** (*array_like*) – Points themselves.

Note: If num and step are given, num will take precedence.

bounds (*self*)

ndim (*self*)

points (*self*)

step (*self*)

class EvalData (*input_data*, *output_data*, *input_labels=None*, *input_units=None*, *enable_extrapolation=False*, *fill_axes=False*, *fill_value=None*, *name=None*)

This class helps managing any kind of result data.

The data gained by evaluation of a function is stored together with the corresponding points of its evaluation. This way all data needed for plotting or other postprocessing is stored in one place. Next to the points of the evaluation the names and units of the included axes can be stored. After initialization an interpolator is set up, so that one can interpolate in the result data by using the overloaded `__call__()` method.

Parameters

- **input_data** – (List of) array(s) holding the axes of a regular grid on which the evaluation took place.
- **output_data** – The result of the evaluation.

Keyword Arguments

- **input_labels** – (List of) labels for the input axes.
- **input_units** – (List of) units for the input axes.
- **name** – Name of the generated data set.
- **fill_axes** – If the dimension of *output_data* is higher than the length of the given *input_data* list, dummy entries will be appended until the required dimension is reached.
- **enable_extrapolation** (*bool*) – If True, internal interpolators will allow extrapolation. Otherwise, the last giben value will be repeated for 1D cases and the result will be padded with zeros for cases > 1D.
- **fill_value** – If invalid data is encountered, it will be replaced with this value before interpolation is performed.

Examples

When instantiating 1d EvalData objects, the list can be omitted

```
>>> axis = Domain((0, 10), 5)
>>> data = np.random.rand(5,)
>>> e_1d = EvalData(axis, data)
```

For other cases, input_data has to be a list

```
>>> axis1 = Domain((0, 0.5), 5)
>>> axis2 = Domain((0, 1), 11)
>>> data = np.random.rand(5, 11)
>>> e_2d = EvalData([axis1, axis2], data)
```

Adding two Instances (if the boundaries fit, the data will be interpolated on the more coarse grid.) Same goes for subtraction and multiplication.

```
>>> e_1 = EvalData(Domain((0, 10), 5), np.random.rand(5,))
>>> e_2 = EvalData(Domain((0, 10), 10), 100*np.random.rand(5,))
>>> e_3 = e_1 + e_2
>>> e_3.output_data.shape
(5,)
```

Interpolate in the output data by calling the object

```
>>> e_4 = EvalData(np.array(range(5)), 2*np.array(range(5))))
>>> e_4.output_data
array([0, 2, 4, 6, 8])
>>> e_5 = e_4([2, 5])
>>> e_5.output_data
array([4, 8])
>>> e_5.output_data.size
2
```

one may also give a slice

```
>>> e_6 = e_4(slice(1, 5, 2))
>>> e_6.output_data
array([2., 6.])
>>> e_5.output_data.size
2
```

For multi-dimensional interpolation a list has to be provided

```
>>> e_7 = e_2d([[.1, .5], [.3, .4, .7]])
>>> e_7.output_data.shape
(2, 3)
```

abs (*self*)

Get the absolute value of the elements form *self.output_data* .

Returns `EvalData` with self.input_data and output_data as result of absolute value calculation.

add (*self*, *other*, *from_left=True*)

Perform the element-wise addition of the output_data arrays from *self* and *other*

This method is used to support addition by implementing __add__ (fromLeft=True) and __radd__(fromLeft=False)). If *other*** is a `EvalData`, the *input_data* lists of *self* and *other* are adjusted using `adjust_input_vectors()` The summation operation is performed on the interpolated output_data. If *other* is a `numbers.Number` it is added according to numpy's broadcasting rules.

Parameters

- **other** (`numbers.Number` or `EvalData`) – Number or EvalData object to add to self.
- **from_left** (*bool*) – Perform the addition from left if True or from right if False.

Returns `EvalData` with adapted input_data and output_data as result of the addition.

adjust_input_vectors (*self*, *other*)

Check the the inputs vectors of *self* and *other* for compatibility (equivalence) and harmonize them if they are compatible.

The compatibility check is performed for every input_vector in particular and examines whether they share the same boundaries. and equalize to the minimal discretized axis. If the amount of discretization steps between the two instances differs, the more precise discretization is interpolated down onto the less precise one.

Parameters **other** (`EvalData`) – Other EvalData class.

Returns

- (list) - New common input vectors.
- (numpy.ndarray) - Interpolated self output_data array.
- (numpy.ndarray) - Interpolated other output_data array.

Return type tuple

interpolate (*self*, *interp_axis*)
Main interpolation method for output_data.

If one of the output dimensions is to be interpolated at one single point, the dimension of the output will decrease by one.

Parameters

- **interp_axis** (`list(list)`) – axis positions in the form
- **1D** (–) – axis with axis=[1,2,3]
- **2D** (–) – [axis1, axis2] with axis1=[1,2,3] and axis2=[0,1,2,3,4]

Returns `EvalData` with *interp_axis* as new input_data and interpolated output_data.

matmul (*self*, *other*, *from_left=True*)
Perform the matrix multiplication of the output_data arrays from *self* and *other* .

This method is used to support matrix multiplication (@) by implementing __matmul__ (from_left=True) and __rmatmul__(from_left=False)). If *other*** is a `EvalData`, the *input_data* lists of *self* and *other* are adjusted using `adjust_input_vectors()`. The matrix multiplication operation is performed on the interpolated output_data. If *other* is a `numbers.Number` it is handled according to numpy's broadcasting rules.

Parameters

- **other** (`EvalData`) – Object to multiply with.
- **from_left** (`boolean`) – Matrix multiplication from left if True or from right if False.

Returns `EvalData` with adapted input_data and output_data as result of matrix multiplication.

mul (*self*, *other*, *from_left=True*)
Perform the element-wise multiplication of the output_data arrays from *self* and *other* .

This method is used to support multiplication by implementing __mul__ (from_left=True) and __rmul__(from_left=False)). If *other*** is a `EvalData`, the *input_data* lists of *self* and *other* are adjusted using `adjust_input_vectors()`. The multiplication operation is performed on the interpolated output_data. If *other* is a `numbers.Number` it is handled according to numpy's broadcasting rules.

Parameters

- **other** (`numbers.Number` or `EvalData`) – Factor to multiply with.
- **boolean** (`from_left`) – Multiplication from left if True or from right if False.

Returns `EvalData` with adapted input_data and output_data as result of multiplication.

sqrt (*self*)
Radicate the elements form *self.output_data* element-wise.

Returns `EvalData` with self.input_data and output_data as result of root calculation.

sub (*self*, *other*, *from_left=True*)
Perform the element-wise subtraction of the output_data arrays from *self* and *other* .

This method is used to support subtraction by implementing __sub__ (from_left=True) and __rsub__(from_left=False)). If *other*** is a `EvalData`, the *input_data* lists of *self* and *other* are adjusted using `adjust_input_vectors()`. The subtraction operation is performed on the interpolated output_data. If *other* is a `numbers.Number` it is handled according to numpy's broadcasting rules.

Parameters

- **other** (`numbers.Number` or `EvalData`) – Number or EvalData object to subtract.
- **from_left** (*boolean*) – Perform subtraction from left if True or from right if False.

Returns `EvalData` with adapted input_data and output_data as result of subtraction.

class Function(*eval_handle*, *domain=- np.inf, np.inf*, *nonzero=- np.inf, np.inf*, *derivative_handles=None*)

Bases: `pyinduct.core.BaseFraction`

Most common instance of a `BaseFraction`. This class handles all tasks concerning derivation and evaluation of functions. It is used broad across the toolbox and therefore incorporates some very specific attributes. For example, to ensure the accurateness of numerical handling functions may only evaluated in areas where they provide nonzero return values. Also their domain has to be taken into account. Therefore the attributes *domain* and *nonzero* are provided.

To save implementation time, ready to go version like `LagrangeFirstOrder` are provided in the `pyinduct.simulation` module.

For the implementation of new shape functions subclass this implementation or directly provide a callable *eval_handle* and callable *derivative_handles* if spatial derivatives are required for the application.

Parameters

- **eval_handle** (*callable*) – Callable object that can be evaluated.
- **domain** (*(list of) tuples*) – Domain on which the eval_handle is defined.
- **nonzero** (*tuple*) – Region in which the eval_handle will return
- **output. Must be a subset of domain** (*nonzero*) –
- **derivative_handles** (*list*) – List of callable(s) that contain
- **of eval_handle** (*derivatives*) –

add_neutral_element(*self*)

Return the neutral element of addition for this object.

In other words: *self + ret_val == self*.

derivative_handles(*self*)

derive(*self*, *order=1*)

Spatially derive this `Function`.

This is done by neglecting *order* derivative handles and to select handle order -1 as the new evaluation_handle.

Parameters order (*int*) – the amount of derivations to perform

Raises

- **TypeError** – If *order* is not of type int.
- **ValueError** – If the requested derivative order is higher than the provided one.

Returns `Function` the derived function.

static from_data(*x*, *y*, ***kwargs*)

Create a `Function` based on discrete data by interpolating.

The interpolation is done by using `interp1d` from scipy, the *kwargs* will be passed.

Parameters

- **x** (*array-like*) – Places where the function has been evaluated .
- **y** (*array-like*) – Function values at *x*.
- ****kwargs** – all kwargs get passed to `Function` .

Returns An interpolating function.

Return type `Function`

function_handle (*self*)

function_space_hint (*self*)
Return the hint that this function is an element of the an scalar product space which is uniquely defined by the scalar product `scalar_product_hint()`.

Note: If you are working on different function spaces, you have to overwrite this hint in order to provide more properties which characterize your specific function space. For example the domain of the functions.

get_member (*self, idx*)
Implementation of the abstract parent method.

Since the `Function` has only one member (itself) the parameter *idx* is ignored and *self* is returned.

Parameters **idx** – ignored.

Returns self

mul_neutral_element (*self*)
Return the neutral element of multiplication for this object.

In other words: *self * ret_val == self.*

raise_to (*self, power*)
Raises the function to the given *power*.

Warning: Derivatives are lost after this action is performed.

Parameters **power** (`numbers.Number`) – power to raise the function to

Returns raised function

scalar_product_hint (*self*)
Return the hint that the `_dot_product_l2()` has to calculated to gain the scalar product.

scale (*self, factor*)
Factory method to scale a `Function`.

Parameters **factor** – `numbers.Number` or a callable.

class Parameters (***kwargs*)
Handy class to collect system parameters. This class can be instantiated with a dict, whose keys will the become attributes of the object. (Bunch approach)

Parameters **kwargs** – parameters

class StackedBase (*base_info*)
Bases: `pyinduct.core.ApproximationBasis`

Implementation of a basis vector that is obtained by stacking different bases onto each other. This typically occurs when the bases of coupled systems are joined to create a unified system.

Parameters **`base_info`** (*`OrderedDict`*) – Dictionary with *base_label* as keys and dictionaries holding information about the bases as values. In detail, these Information must contain:

- sys_name (str): Name of the system the base is associated with.
- **order (int): Highest temporal derivative order with which the** base shall be represented in the stacked base.
- base (`ApproximationBase`): The actual basis.

`function_space_hint`(*self*)

Hint that returns properties that characterize the functional space of the fractions. It can be used to determine if function spaces match.

Note: Overwrite to implement custom functionality.

`is_compatible_to`(*self*, *other*)

Helper functions that checks compatibility between two approximation bases.

In this case compatibility is given if the two bases live in the same function space.

Parameters **`other`** (`Approximation Base`) – Approximation basis to compare with.

Returns: True if bases match, False if they do not.

`scalar_product_hint`(*self*)

Hint that returns steps for scalar product calculation with elements of this base.

Note: Overwrite to implement custom functionality.

`abstract scale`(*self*, *factor*)

`transformation_hint`(*self*, *info*)

If *info.src_lbl* is a member, just return it, using to correct derivative transformation, otherwise return *None*

Parameters **`info`** (*`TransformationInfo`*) – Information about the requested transformation.

Returns transformation handle

`class TransformationInfo`

Structure that holds information about transformations between different bases.

This class serves as an easy to use structure to aggregate information, describing transformations between different *`BaseFraction`* s. It can be tested for equality to check the equity of transformations and is hashable which makes it usable as dictionary key to cache different transformations.

`src_lbl`

label of source basis

Type str

`dst_lbl`

label destination basis

Type str

`src_base`

source basis in form of an array of the source Fractions

Type `numpy.ndarray`

`dst_base`

destination basis in form of an array of the destination Fractions

Type `numpy.ndarray`

src_order
available temporal derivative order of source weights

dst_order
needed temporal derivative order for destination weights

as_tuple(*self*)

mirror(*self*)
Factory method, that creates a new TransformationInfo object by mirroring *src* and *dst* terms. This helps handling requests to different bases.

back_project_from_base(*weights*, *base*)
Build evaluation handle for a distributed variable that was approximated as a set of *weights* om a certain *base*.

Parameters

- **weights** (*numpy.ndarray*) – Weight vector.
- **base** (`ApproximationBase`) – Base to be used for the projection.

Returns evaluation handle

calculate_base_transformation_matrix(*src_base*, *dst_base*, *scalar_product=None*)
Calculates the transformation matrix V, so that the a set of weights, describing a function in the *src_base* will express the same function in the *dst_base*, while minimizing the reprojection error. An quadratic error is used as the error-norm for this case.

> **Warning:** This method assumes that all members of the given bases have the same type and that their *BaseFraction* s, define compatible scalar products.

Raises **TypeError** – If given bases do not provide an `scalar_product_hint()` method.

Parameters

- **src_base** (`ApproximationBase`) – Current projection base.
- **dst_base** (`ApproximationBase`) – New projection base.
- **scalar_product** (*list of callable*) – Callbacks for product calculation. Defaults to *scalar_product_hint* from *src_base*.

Returns Transformation matrix V.

Return type `numpy.ndarray`

calculate_expanded_base_transformation_matrix(*src_base*, *dst_base*, *src_order*, *dst_order*, *use_eye=False*)
Constructs a transformation matrix $\bar{V}$ from basis given by *src_base* to basis given by *dst_base* that also transforms all temporal derivatives of the given weights.

See: *calculate_base_transformation_matrix()* for further details.

Parameters

- **dst_base** (`ApproximationBase`) – New projection base.
- **src_base** (`ApproximationBase`) – Current projection base.
- **src_order** – Temporal derivative order available in *src_base*.
- **dst_order** – Temporal derivative order needed in *dst_base*.
- **use_eye** (*bool*) – Use identity as base transformation matrix. (For easy selection of derivatives in the same base)

Raises **ValueError** – If destination needs a higher derivative order than source can provide.

Returns Transformation matrix

Return type `numpy.ndarray`

calculate_scalar_matrix(*values_a*, *values_b*)

Convenience version of py:function:*calculate_scalar_product_matrix* with `numpy.multiply()` hard-coded as *scalar_product_handle*.

Parameters

- **values_a** (*numbers.Number or numpy.ndarray*) – (array of) value(s) for rows
- **values_b** (*numbers.Number or numpy.ndarray*) – (array of) value(s) for columns

Returns Matrix containing the pairwise products of the elements from *values_a* and *values_b*.

Return type numpy.ndarray

calculate_scalar_product_matrix(*base_a*, *base_b*, *scalar_product=None*, *optimize=True*)

Calculates a matrix A , whose elements are the scalar products of each element from *base_a* and *base_b*, so that $a_{ij} = \langle \mathrm{a}_i\,,\, \mathrm{b}_j \rangle$.

Parameters

- **base_a** (`ApproximationBase`) – Basis a
- **base_b** (`ApproximationBase`) – Basis b
- **scalar_product** – (List of) function objects that are passed the members of the given bases as pairs. Defaults to the scalar product given by *base_a*
- **optimize** (*bool*) – Switch to turn on the symmetry based speed up. For development purposes only.

Returns matrix A

Return type numpy.ndarray

change_projection_base(*src_weights*, *src_base*, *dst_base*)

Converts given weights that form an approximation using *src_base* to the best possible fit using *dst_base*.

Parameters

- **src_weights** (*numpy.ndarray*) – Vector of numbers.
- **src_base** (`ApproximationBase`) – The source Basis.
- **dst_base** (`ApproximationBase`) – The destination Basis.

Returns target weights

Return type `numpy.ndarray`

complex_quadrature(*func*, *a*, *b*, ***kwargs*)

Wraps the scipy.qaudpack routines to handle complex valued functions.

Parameters

- **func** (*callable*) – function
- **a** (`numbers.Number`) – lower limit
- **b** (`numbers.Number`) – upper limit
- ****kwargs** – Arbitrary keyword arguments for desired scipy.qaudpack routine.

Returns (real part, imaginary part)

Return type tuple

complex_wrapper (*func*)
Wraps complex valued functions into two-dimensional functions. This enables the root-finding routine to handle it as a vectorial function.

Parameters **func** (`callable`) – Callable that returns a complex result.

Returns function handle, taking x = (re(x), im(x) and returning [re(func(x), im(func(x)].

Return type two-dimensional, callable

domain_intersection (*first*, *second*)
Calculate intersection(s) of two domains.

Parameters

- **first** (`set`) – (Set of) tuples defining the first domain.
- **second** (`set`) – (Set of) tuples defining the second domain.

Returns Intersection given by (start, end) tuples.

Return type set

domain_simplification (*domain*)
Simplify a domain, given by possibly overlapping subdomains.

Parameters **domain** (`set`) – Set of tuples, defining the (start, end) points of the subdomains.

Returns Simplified domain.

Return type list

dot_product (*first*, *second*)
Calculate the inner product of two vectors.

Parameters

- **first** (`numpy.ndarray`) – first vector
- **second** (`numpy.ndarray`) – second vector

Returns inner product

dot_product_l2 (*first*, *second*)
Calculate the inner product on L2.

Given two functions $\varphi(z)$ and $\psi(z)$ this functions calculates

$$\langle\varphi(z)|\psi(z)\rangle = \int_{\Gamma_0}^{\Gamma_1} \bar{\varphi}(\zeta)\psi(\zeta)\,\mathrm{d}\zeta\ .$$

Parameters

- **first** (`Function`) – first function
- **second** (`Function`) – second function

Returns inner product

find_roots (*function*, *grid*, *n_roots=None*, *rtol=1e-05*, *atol=1e-08*, *cmplx=False*, *sort_mode='norm'*)
Searches *n_roots* roots of the *function* $f(\boldsymbol{x})$ on the given *grid* and checks them for uniqueness with aid of *rtol*.

In Detail `scipy.optimize.root()` is used to find initial candidates for roots of $f(\boldsymbol{x})$. If a root satisfies the criteria given by atol and rtol it is added. If it is already in the list, a comprehension between the already present entries' error and the current error is performed. If the newly calculated root comes with a smaller error it supersedes the present entry.

Raises **ValueError** – If the demanded amount of roots can't be found.

Parameters

- **function** (*callable*) – Function handle for math:*f(boldsymbol{x})* whose roots shall be found.
- **grid** (*list*) – Grid to use as starting point for root detection. The i th element of this list provides sample points for the i th parameter of $\boldsymbol{x}$.
- **n_roots** (*int*) – Number of roots to find. If none is given, return all roots that could be found in the given area.
- **rtol** – Tolerance to be exceeded for the difference of two roots to be unique: $f(r1) - f(r2) >$ rtol .
- **atol** – Absolute tolerance to zero: $f(x^0) <$ atol .
- **cmplx** (*bool*) – Set to True if the given *function* is complex valued.
- **sort_mode** (*str*) – Specify tho order in which the extracted roots shall be sorted. Default "norm" sorts entries by their l_2 norm, while "component" will sort them in increasing order by every component.

Returns numpy.ndarray of roots; sorted in the order they are returned by $f(\boldsymbol{x})$.

generic_scalar_product (*b1*, *b2=None*, *scalar_product=None*)

Calculates the pairwise scalar product between the elements of the `ApproximationBase` *b1* and *b2*.

Parameters

- **b1** (ApproximationBase) – first basis
- **b2** (ApproximationBase) – second basis, if omitted defaults to *b1*
- **scalar_product** (*list of callable*) – Callbacks for product calculation. Defaults to *scalar_product_hint* from *b1*.

Note: If *b2* is omitted, the result can be used to normalize *b1* in terms of its scalar product.

get_base (*label*)

Retrieve registered set of initial functions by their label.

Parameters **label** (*str*) – String, label of functions to retrieve.

Returns initial_functions

get_transformation_info (*source_label*, *destination_label*, *source_order=0*, *destination_order=0*)

Provide the weights transformation from one/source base to another/destination base.

Parameters

- **source_label** (*str*) – Label from the source base.
- **destination_label** (*str*) – Label from the destination base.
- **source_order** – Order from the available time derivative of the source weights.
- **destination_order** – Order from the desired time derivative of the destination weights.

Returns Transformation info object.

Return type *TransformationInfo*

get_weight_transformation (*info*)

Create a handle that will transform weights from *info.src_base* into weights for *info-dst_base* while paying respect to the given derivative orders.

This is accomplished by recursively iterating through source and destination bases and evaluating their `transformation_hints`.

Parameters **info** (*TransformationInfo*) – information about the requested transformation.

Returns transformation function handle

Return type callable

integrate_function (*func*, *interval*)

Numerically integrate a function on a given interval using `complex_quadrature()`.

Parameters

- **func** (`callable`) – Function to integrate.
- **interval** (`list of tuples`) – List of (start, end) values of the intervals to integrate on.

Returns (Result of the Integration, errors that occurred during the integration).

Return type tuple

normalize_base (*b1*, *b2=None*)

Takes two `ApproximationBase`'s $\boldsymbol{b}_1$, $\boldsymbol{b}_1$ and normalizes them so that $\langle \boldsymbol{b}_{1i} \, , \, \boldsymbol{b}_{2i} \rangle = 1$. If only one base is given, $\boldsymbol{b}_2$ defaults to $\boldsymbol{b}_1$.

Parameters

- **b1** (`ApproximationBase`) – $\boldsymbol{b}_1$
- **b2** (`ApproximationBase`) – $\boldsymbol{b}_2$

Raises **ValueError** – If $\boldsymbol{b}_1$ and $\boldsymbol{b}_2$ are orthogonal.

Returns if *b2* is None, otherwise: Tuple of 2 `ApproximationBase`'s.

Return type `ApproximationBase`

project_on_base (*state*, *base*)

Projects a *state* on a basis given by *base*.

Parameters

- **state** (`array_like`) – List of functions to approximate.
- **base** (`ApproximationBase`) – Basis to project onto.

Returns Weight vector in the given *base*

Return type numpy.ndarray

project_on_bases (*states*, *canonical_equations*)

Convenience wrapper for `project_on_base()`. Calculate the state, assuming it will be constituted by the dominant base of the respective system. The keys from the dictionaries *canonical_equations* and *states* must be the same.

Parameters

- **states** – Dictionary with a list of functions as values.
- **canonical_equations** – List of `CanonicalEquation` instances.

Returns Finite dimensional state as 1d-array corresponding to the concatenated dominant bases from *canonical_equations*.

Return type numpy.array

project_weights (*projection_matrix*, *src_weights*)

Project *src_weights* on new basis using the provided *projection_matrix*.

Parameters

- **projection_matrix** (`numpy.ndarray`) – projection between the source and the target basis; dimension (m, n)

- **src_weights** (numpy.ndarray) – weights in the source basis; dimension (1, m)

Returns weights in the target basis; dimension (1, n)

Return type numpy.ndarray

real (*data*)

Check if the imaginary part of `data` vanishes and return its real part if it does.

Parameters data (*numbers.Number or array_like*) – Possibly complex data to check.

Raises ValueError – If provided data can't be converted within the given tolerance limit.

Returns Real part of `data`.

Return type numbers.Number or array_like

sanitize_input (*input_object*, *allowed_type*)

Sanitizes input data by testing if *input_object* is an array of type *allowed_type*.

Parameters

- **input_object** – Object which is to be checked.
- **allowed_type** – desired type

Returns input_object

vectorize_scalar_product (*first*, *second*, *scalar_product*)

Call the given `scalar_product` in a loop for the arguments in `left` and `right`.

Given two vectors of functions

$$\boldsymbol{\varphi}(z) = (\varphi_0(z), \dots, \varphi_N(z))^T$$

and

$$\boldsymbol{\psi}(z) = (\psi_0(z), \dots, \psi_N(z))^T \text{`},$$

this function computes $\langle \boldsymbol{\varphi}(z) | \boldsymbol{\psi}(z) \rangle_{L2}$ where

$$\langle \varphi_i(z) | \psi_j(z) \rangle_{L2} = \int_{\Gamma_0}^{\Gamma_1} \bar{\varphi}_i(\zeta) \psi_j(\zeta) \, \mathrm{d}\zeta \, .$$

Herein, $\bar{\varphi}_i(\zeta)$ denotes the complex conjugate and Γ_0 as well as Γ_1 are derived by computing the intersection of the nonzero areas of the involved functions.

Parameters

- **first** (callable or numpy.ndarray) – (1d array of n) callable(s)
- **second** (callable or numpy.ndarray) – (1d array of n) callable(s)

Raises ValueError, if the provided arrays are not equally long. –

Returns Array of inner products

Return type numpy.ndarray

7.2 Shapefunctions

The shapefunctions module contains generic shapefunctions that can be used to approximate distributed systems without giving any information about the systems themselves. This is achieved by projecting them on generic, piecewise smooth functions.

class ShapeFunction(**args*, ***kwargs*)

Base class for approximation functions with compact support.

When a continuous variable of e.g. space and time $x(z,t)$ is decomposed in a series $\tilde{x} = \sum_{i=1}^{\infty} \varphi_i(z) c_i(t)$ the $\varphi_i(z)$ denote the shape functions.

classmethod cure_interval(*cls*, *interval*, ***kwargs*)

Create a network or set of functions from this class and return an approximation base (`Base`) on the given interval.

The `kwargs` may hold the order of approximation or the amount of functions to use. Use them in your child class as needed.

If you don't need to now from which class this method is called, overwrite the `@classmethod` decorator in the child class with the `@staticmethod` decorator.

Short reference: Inside a `@staticmethod` you know nothing about the class from which it is called and you can just play with the given parameters. Inside a `@classmethod` you can additionally operate on the class, since the first parameter is always the class itself.

Parameters

- **interval** (`Domain`) – Interval to cure.
- ****kwargs** – Various arguments, depending on the implementation.

Returns Approximation base, generated by the created shape functions.

Return type `Base`

7.2.1 Shapefunction Types

class LagrangeFirstOrder(*start*, *top*, *end*, ***kwargs*)

Bases: `pyinduct.shapefunctions.ShapeFunction`

Lagrangian shape functions of order 1.

Parameters

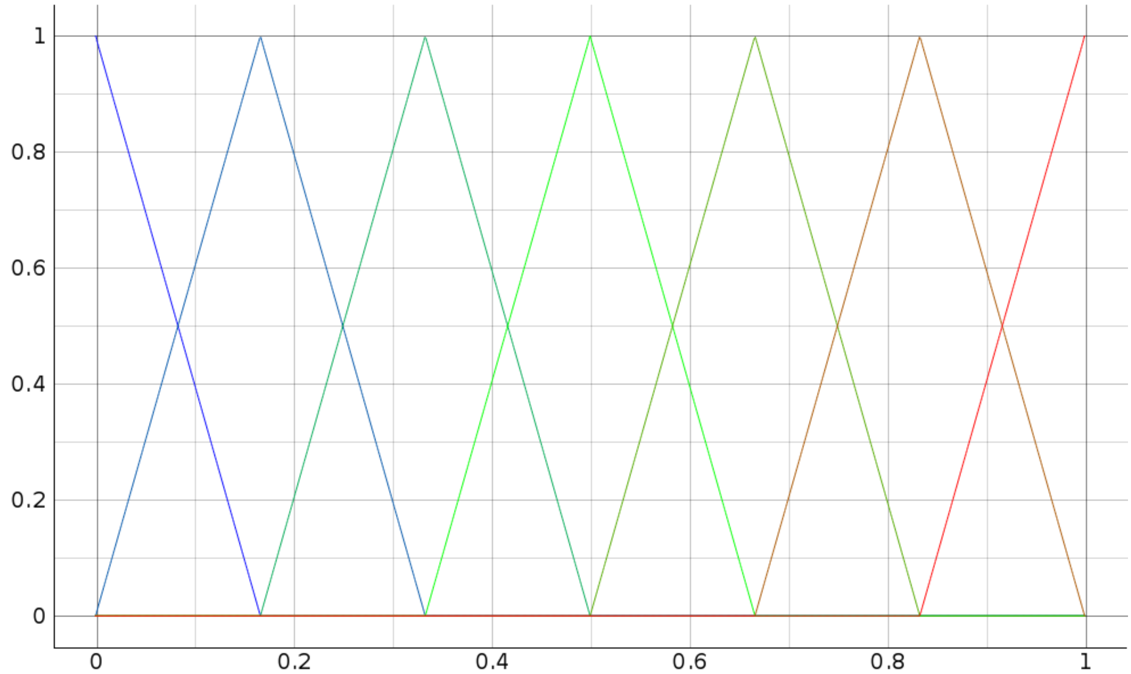
- **start** – Start node
- **top** – Top node, where $f(x) = 1$
- **end** – End node

Keyword Arguments

- **half** –
- **right_border** –
- **left_border** –

Example plot of the functions `funcs` generated with

```
>>> nodes, funcs = cure_interval(LagrangeFirstOrder, (0, 1), node_count=7)
```



static cure_interval(*domain*, ***kwargs*)

Cure the given interval with LagrangeFirstOrder shape functions.

Parameters **domain** (*Domain*) – Domain to be cured, the points specify the nodes which will be used.

Returns Base, generated by a set of *LagrangeFirstOrder* shapefunctions.

Return type pi.Base

class LagrangeSecondOrder(*start*, *mid*, *end*, ***kwargs*)

Bases: *pyinduct.shapefunctions.ShapeFunction*

Lagrangian shape functions of order 2.

Parameters

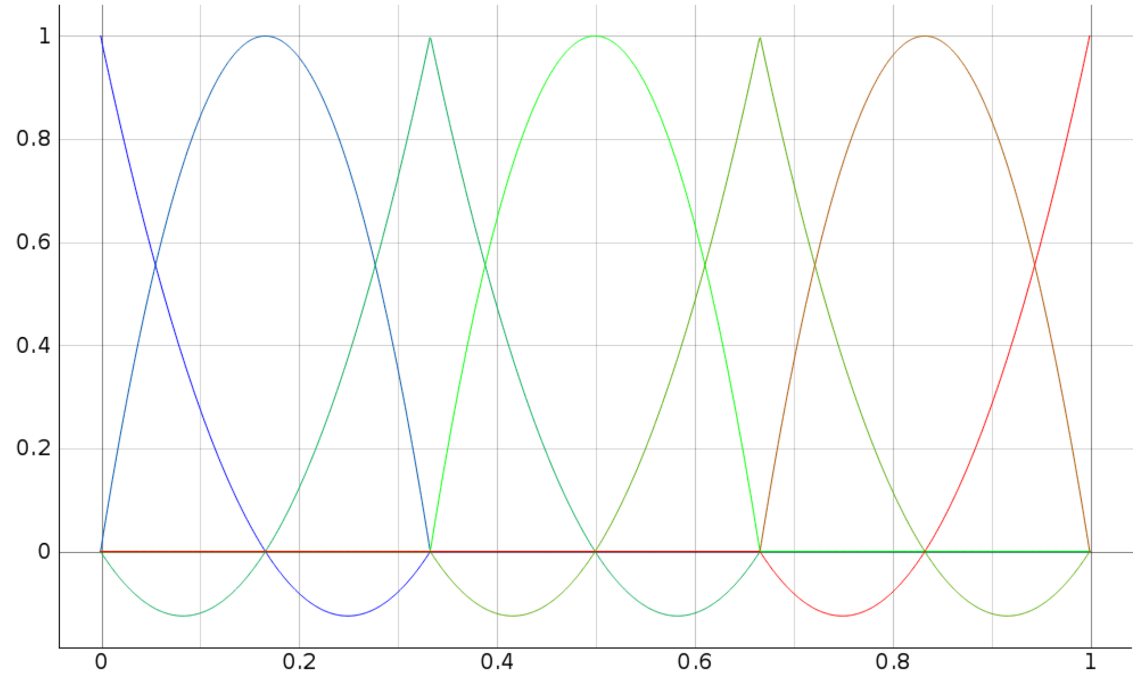
- **start** – start node
- **mid** – middle node, where $f(x) = 1$
- **end** – end node

Keyword Arguments

- **curvature** (*str*) – "concave" or "convex"
- **half** (*str*) – Generate only "left" or "right" half.
- **domain** (*tuple*) – Domain on which the function is defined.

Example plot of the functions `funcs` generated with

```
>>> nodes, funcs = cure_interval(LagrangeSecondOrder, (0, 1), node_count=7)
```



static cure_interval (*domain*, ***kwargs*)

Hint function that will cure the given interval with `LagrangeSecondOrder`.

Parameters **domain** (`Domain`) – domain to be cured

Returns (domain, funcs), where funcs is set of `LagrangeSecondOrder` shapefunctions.

Return type tuple

class LagrangeNthOrder (*order*, *nodes*, *left=False*, *right=False*, *mid_num=None*, *boundary=None*, *domain=- np.inf, np.inf*)

Bases: `pyinduct.shapefunctions.ShapeFunction`

Lagrangian shape functions of order n.

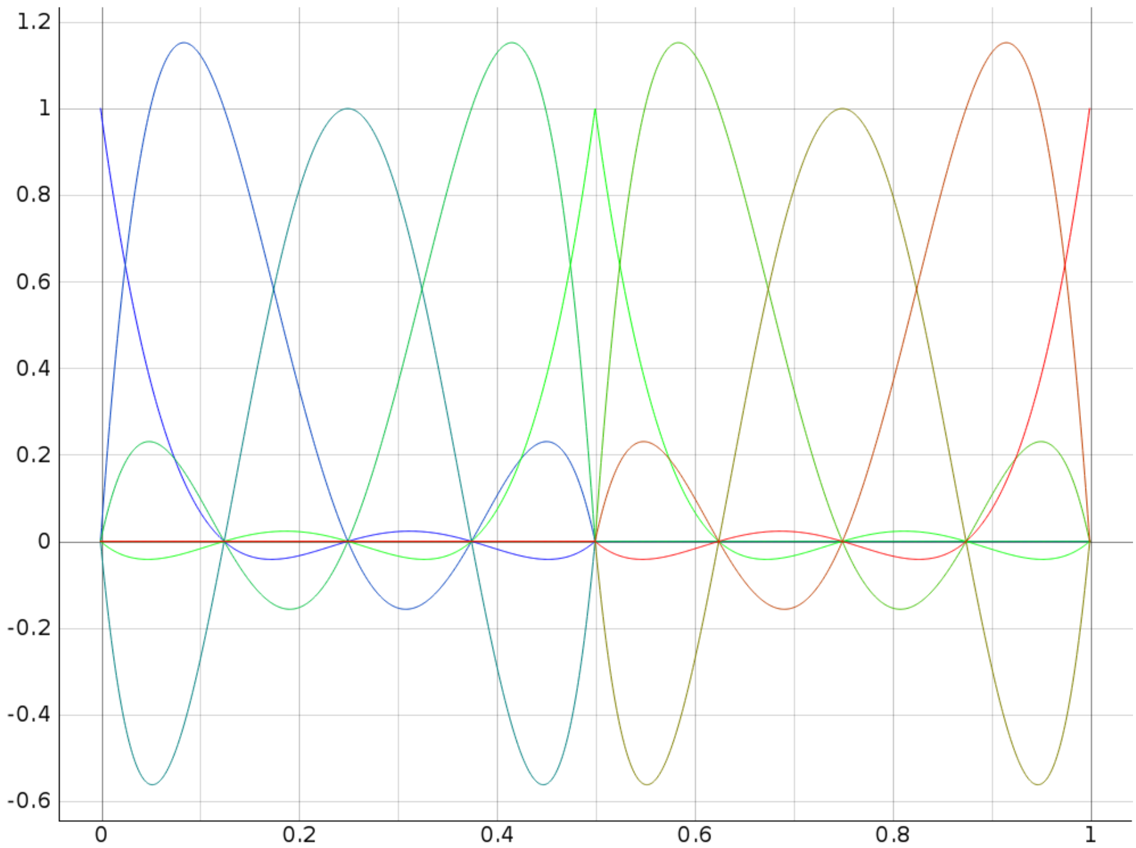
Note: The polynomials between the boundary-polynomials and the peak-polynomials, respectively between peak-polynomials and peak-polynomials, are called mid-polynomials.

Parameters

- **order** (*int*) – Order of the lagrangian polynomials.
- **nodes** (*numpy.array*) – Nodes on which the piecewise defined functions have to be one/zero. Length of nodes must be either $order * 2 + 1$ (for peak-polynomials, see notes) or 'order +1' (for boundary- and mid-polynomials).
- **left** (*bool*) – State the first node (*nodes[0]*) to be the left boundary of the considered domain.
- **right** (*bool*) – State the last node (*nodes[-1]*) to be the right boundary of the considered domain.
- **mid_num** (*int*) – Local number of mid-polynomials (see notes) to use (only used for *order* >= 2). mid_num $\in \{1, ..., \text{order} - 1\}$
- **boundary** (*str*) – provide "left" or "right" to instantiate the according boundary-polynomial.
- **domain** (*tuple*) – Domain of the function.

Example plot of the functions `funcs` generated with

```
>>> nodes, funcs = pi.cure_interval(sh.LagrangeNthOrder, (0, 1), node_count=9,
↪order=4)
```



static cure_interval (*domain*, ***kwargs*)

Hint function that will cure the given interval with `LagrangeNthOrder`. Length of the domain argument L must satisfy the condition

$$L = 1 + (1 + n)order \quad \forall n \in \mathbb{N}.$$

E.g. n - order = 1 -> $L \in \{2, 3, 4, 5, ...\}$ - order = 2 -> $L \in \{3, 5, 7, 9, ...\}$ - order = 3 -> $L \in \{4, 7, 10, 13, ...\}$ - and so on.

Parameters

- **domain** (`Domain`) – Domain to be cured.
- **order** (`int`) – Order of the lagrange polynomials.

Returns Base, generated by the created shapefunctions.

Return type `Base`

7.3 Eigenfunctions

This modules provides eigenfunctions for a certain set of second order spatial operators. Therefore functions for the computation of the corresponding eigenvalues are included. The functions which compute the eigenvalues are deliberately separated from the predefined eigenfunctions in order to handle transformations and reduce effort within the controller implementation.

class AddMulFunction (*function*)

Bases: `object`

(Temporary) Function class which can multiplied with scalars and added with functions. Only needed to compute the matrix (of scalars) vector (of functions) product in `FiniteTransformFunction`. Will be no longer needed when `Function` is overloaded with `__add__` and `__mul__` operator.

Parameters **`function`** (*`callable`*) –

`class Base`(*fractions*, *matching_base_lbls=None*, *intermediate_base_lbls=None*)

Bases: `pyinduct.core.ApproximationBasis`

Base class for approximation bases.

In general, a `Base` is formed by a certain amount of `BaseFractions` and therefore forms finite-dimensional subspace of the distributed problem's domain. Most of the time, the user does not need to interact with this class.

Parameters

- **`fractions`** (iterable of `BaseFraction`) – List, array or dict of `BaseFraction`'s
- **`matching_base_lbls`** (*`list of str`*) – List of labels from exactly matching bases, for which no transformation is necessary. Useful for transformations from bases that 'live' in different function spaces but evolve with the same time dynamic/coefficients (e.g. modal bases).
- **`intermediate_base_lbls`** (*`list of str`*) – If it is certain that this base instance will be asked (as destination base) to return a transformation to a source base, whose implementation is cumbersome, its label can be provided here. This will trigger the generation of the transformation using build-in features. The algorithm, implemented in `get_weights_transformation` is then called again with the intermediate base as destination base and the 'old' source base. With this technique arbitrary long transformation chains are possible, if the provided intermediate bases again define intermediate bases.

`derive`(*self*, *order*)

Basic implementation of derive function. Empty implementation, overwrite to use this functionality.

Parameters **`order`** (`numbers.Number`) – derivative order

Returns derived object

Return type `Base`

`function_space_hint`(*self*)

Hint that returns properties that characterize the functional space of the fractions. It can be used to determine if function spaces match.

Note: Overwrite to implement custom functionality.

`get_attribute`(*self*, *attr*)

Retrieve an attribute from the fractions of the base.

Parameters **`attr`** (*`str`*) – Attribute to query the fractions for.

Returns Array of `len(fractions)` holding the attributes. With *None* entries if the attribute is missing.

Return type `np.ndarray`

`raise_to`(*self*, *power*)

Factory method to obtain instances of this base, raised by the given power.

Parameters **`power`** – power to raise the basis onto.

`scalar_product_hint`(*self*)

Hint that returns steps for scalar product calculation with elements of this base.

Note: Overwrite to implement custom functionality.

scale (*self*, *factor*)
Factory method to obtain instances of this base, scaled by the given factor.

Parameters **factor** – factor or function to scale this base with.

transformation_hint (*self*, *info*)
Method that provides a information about how to transform weights from one *BaseFraction* into another.

In Detail this function has to return a callable, which will take the weights of the source- and return the weights of the target system. It may have keyword arguments for other data which is required to perform the transformation. Information about these extra keyword arguments should be provided in form of a dictionary whose keys are keyword arguments of the returned transformation handle.

Note: This implementation covers the most basic case, where the two *BaseFraction*'s are of same type. For any other case it will raise an exception. Overwrite this Method in your implementation to support conversion between bases that differ from yours.

Parameters **info** – *TransformationInfo*

Raises **NotImplementedError** –

Returns Transformation handle

class Domain (*bounds=None*, *num=None*, *step=None*, *points=None*)
Bases: `object`

Helper class that manages ranges for data evaluation, containing parameters.

Parameters

- **bounds** (*tuple*) – Interval bounds.
- **num** (*int*) – Number of points in interval.
- **step** (*numbers.Number*) – Distance between points (if homogeneous).
- **points** (*array_like*) – Points themselves.

Note: If num and step are given, num will take precedence.

bounds (*self*)

ndim (*self*)

points (*self*)

step (*self*)

class FiniteTransformFunction (*function*, *M*, *l*, *scale_func=None*, *nested_lambda=False*)
Bases: *pyinduct.core.Function*

This class provides a transformed *Function* $\bar{x}(z)$ through the transformation $\bar{\boldsymbol{\xi}} = T * \boldsymbol{\xi}$, with the function vector $\boldsymbol{\xi} \in \mathbb{R}^{2n}$ and with a given matrix $T \in \mathbb{R}^{2n\times 2n}$. The operator $*$ denotes the matrix (of scalars) vector (of functions) product. The interim result $\bar{\boldsymbol{\xi}}$ is a vector $\bar{\boldsymbol{\xi}} = (\bar{\xi}_{1,0}, ..., \bar{\xi}_{1,n-1}, \bar{\xi}_{2,0}, ..., \bar{\xi}_{2,n-1})^T$ of functions

$$\begin{aligned}\bar{\xi}_{1,j} &= \bar{x}(jl_0 + z), \qquad j = 0, ..., n-1, \quad l_0 = l/n, \quad z \in [0, l_0]\\ \bar{\xi}_{2,j} &= \bar{x}(l - jl_0 + z).\end{aligned}$$

Finally, the provided function $\bar{x}(z)$ is given through $\bar{\xi}_{1,0}, ..., \bar{\xi}_{1,n-1}$.

Note: For a more extensive documentation see section 4.2 in:

- Wang, S. und F. Woittennek: Backstepping-Methode für parabolische Systeme mit punktförmigem inneren Eingriff. Automatisierungstechnik, 2015. http://dx.doi.org/10.1515/auto-2015-0023

Parameters

- **function** (*callable*) – Function $x(z)$ that will act as start for the generation of $2n$ Functions $\xi_{i,j}$ in $\boldsymbol{\xi} = (\xi_{1,0}, ..., \xi_{1,n-1}, \xi_{2,0}, ..., \xi_{2,n-1})^T$.
- **M** (*numpy.ndarray*) – Matrix $T \in \mathbb{R}^{2n \times 2n}$ of scalars.
- **l** (*numbers.Number*) – Length of the domain ($z \in [0, l]$).

class Function (*eval_handle*, *domain=- np.inf*, *np.inf*, *nonzero=- np.inf*, *np.inf*, *derivative_handles=None*)

Bases: *pyinduct.core.BaseFraction*

Most common instance of a *BaseFraction*. This class handles all tasks concerning derivation and evaluation of functions. It is used broad across the toolbox and therefore incorporates some very specific attributes. For example, to ensure the accurateness of numerical handling functions may only evaluated in areas where they provide nonzero return values. Also their domain has to be taken into account. Therefore the attributes *domain* and *nonzero* are provided.

To save implementation time, ready to go version like *LagrangeFirstOrder* are provided in the *pyinduct.simulation* module.

For the implementation of new shape functions subclass this implementation or directly provide a callable *eval_handle* and callable *derivative_handles* if spatial derivatives are required for the application.

Parameters

- **eval_handle** (*callable*) – Callable object that can be evaluated.
- **domain** (*(list of) tuples*) – Domain on which the eval_handle is defined.
- **nonzero** (*tuple*) – Region in which the eval_handle will return
- **output. Must be a subset of domain** (*nonzero*) –
- **derivative_handles** (*list*) – List of callable(s) that contain
- **of eval_handle** (*derivatives*) –

add_neutral_element (*self*)

Return the neutral element of addition for this object.

In other words: *self + ret_val == self*.

derivative_handles (*self*)

derive (*self*, *order=1*)

Spatially derive this *Function*.

This is done by neglecting *order* derivative handles and to select handle order -1 as the new evaluation_handle.

Parameters order (*int*) – the amount of derivations to perform

Raises

- **TypeError** – If *order* is not of type int.
- **ValueError** – If the requested derivative order is higher than the provided one.

Returns *Function* the derived function.

static from_data (*x*, *y*, ***kwargs*)
Create a `Function` based on discrete data by interpolating.

The interpolation is done by using `interp1d` from scipy, the *kwargs* will be passed.

Parameters

- **x** (*array-like*) – Places where the function has been evaluated .
- **y** (*array-like*) – Function values at *x*.
- ****kwargs** – all kwargs get passed to `Function` .

Returns An interpolating function.

Return type `Function`

function_handle (*self*)

function_space_hint (*self*)
Return the hint that this function is an element of the an scalar product space which is uniquely defined by the scalar product `scalar_product_hint()`.

Note: If you are working on different function spaces, you have to overwrite this hint in order to provide more properties which characterize your specific function space. For example the domain of the functions.

get_member (*self*, *idx*)
Implementation of the abstract parent method.

Since the `Function` has only one member (itself) the parameter *idx* is ignored and *self* is returned.

Parameters **idx** – ignored.

Returns self

mul_neutral_element (*self*)
Return the neutral element of multiplication for this object.

In other words: *self * ret_val == self*.

raise_to (*self*, *power*)
Raises the function to the given *power*.

Warning: Derivatives are lost after this action is performed.

Parameters **power** (`numbers.Number`) – power to raise the function to

Returns raised function

scalar_product_hint (*self*)
Return the hint that the `_dot_product_l2()` has to calculated to gain the scalar product.

scale (*self*, *factor*)
Factory method to scale a `Function`.

Parameters **factor** – `numbers.Number` or a callable.

class LambdifiedSympyExpression (*sympy_funcs*, *spat_symbol*, *spatial_domain*, *complex_=False*)
Bases: `pyinduct.core.Function`

This class provides a `Function` $\varphi(z)$ based on a lambdified sympy expression. The sympy expressions for the function and it's spatial derivatives must be provided as the list *sympy_funcs*. The expressions must be provided with increasing derivative order, starting with order 0.

Parameters

- **sympy_funcs** (*array_like*) – Sympy expressions for the function and the derivatives: $\varphi(z), \varphi'(z), \ldots$.
- **spat_symbol** – Sympy symbol for the spatial variable z.
- **spatial_domain** (*tuple*) – Domain on which $\varphi(z)$ is defined (e.g.: `spatial_domain=(0, 1)`).
- **complex** (*bool*) – If False the Function raises an Error if it returns complex values. Default: False.

class SecondOrderDirichletEigenfunction(*om, param, l, scale=1, max_der_order=2*)

Bases: `pyinduct.eigenfunctions.SecondOrderEigenfunction`

This class provides an eigenfunction $\varphi(z)$ to eigenvalue problems of the form

$$\begin{aligned} a_2\varphi''(z) + a_1\varphi'(z) + a_0\varphi(z) &= \lambda\varphi(z) \\ \varphi(0) &= 0 \\ \varphi(l) &= 0. \end{aligned}$$

The eigenfrequency

$$\omega = \sqrt{-\frac{a_1^2}{4a_2^2} + \frac{a_0 - \lambda}{a_2}}$$

must be provided (for example with the `eigfreq_eigval_hint()` of this class).

Parameters

- **om** (*numbers.Number*) – eigenfrequency ω
- **param** (*array_like*) – $\Big(a_2, a_1, a_0, None, None\Big)^T$
- **l** (*numbers.Number*) – End of the domain $z \in [0, l]$.
- **scale** (*numbers.Number*) – Factor to scale the eigenfunctions.
- **max_der_order** (*int*) – Number of derivative handles that are needed.

static eigfreq_eigval_hint(*param, l, n_roots*)

Return the first *n_roots* eigenfrequencies ω and eigenvalues λ.

$$\omega_i = \sqrt{-\frac{a_1^2}{4a_2^2} + \frac{a_0 - \lambda_i}{a_2}} \quad i = 1, \ldots, \text{n\_roots}$$

to the considered eigenvalue problem.

Parameters

- **param** (*array_like*) – $\Big(a_2, a_1, a_0, None, None\Big)^T$
- **l** (*numbers.Number*) – Right boundary value of the domain $[0, l] \ni z$.
- **n_roots** (*int*) – Amount of eigenfrequencies to be compute.

Returns

$$\Big(\big[\omega_1, \ldots, \omega_{\text{n\_roots}}\big], \big[\lambda_1, \ldots, \lambda_{\text{n\_roots}}\big]\Big)$$

Return type tuple –> two numpy.ndarrays of length *n_roots*

class SecondOrderEigenVector(*char_pair*, *coefficients*, *domain*, *derivative_order*)

Bases: `pyinduct.shapefunctions.ShapeFunction`

This class provides eigenvectors of the form

$$\varphi(z) = e^{\eta z}\big(\kappa_1 \cos(\nu z) + \sin(\nu z)\big) \, ,$$

of a linear second order spatial operator A denoted by

$$(\mathrm{A}\varphi)(z) = a_2 \partial_z^2 \varphi(z) + a_1 \partial_z \varphi(z) + a_0 \varphi(z)$$

where the a_i are constant and whose boundary conditions are given by

$$\begin{aligned} \alpha_1 \partial_z x(z_1) + \alpha_0 x(z_1) &= 0 \\ \beta_1 \partial_z x(z_2) + \beta_0 x(z_2) &= 0. \end{aligned}$$

To calculate the corresponding eigenvectors, the problem

$$(\mathrm{A}\varphi)(z) = \lambda \varphi(z)$$

is solved for the eigenvalues λ , making use of the characteristic roots p given by

$$p = \underbrace{-\frac{a_1}{a_2}}_{=:\eta} + j \underbrace{\sqrt{\frac{a_0 - \lambda}{a_2} - \left(\frac{a_1}{2a_2}\right)^2}}_{=:\nu}$$

Note: To easily instantiate a set of eigenvectors for a certain system, use the `cure_hint()` of this class or even better the helper-function `cure_interval()` .

Warns

- **Since an eigenvalue corresponds to a pair of conjugate complex**
- **characteristic roots, latter are only calculated for the positive**
- **half-plane since the can be mirrored.**
- **To obtain the orthonormal properties of the generated**
- **eigenvectors, the eigenvalue corresponding to the characteristic**
- **root 0+0j is ignored, since it leads to the zero function.**

Parameters

- **char_pair** (*tuple of complex*) – Characteristic root, corresponding to the eigenvalue λ for which the eigenvector is to be determined. (Can be obtained by `convert_to_characteristic_root()`)
- **coefficients** (*tuple*) – Constants of the exponential ansatz solution.

Returns The eigenvector.

Return type *SecondOrderEigenVector*

static calculate_eigenvalues(*domain*, *params*, *count*, *extended_output=False*, ***kwargs*)

Determine the eigenvalues of the problem given by *parameters* defined on *domain* .

Parameters

- **domain** (*Domain*) – Domain of the spatial problem.
- **params** (*bunch-like*) – Parameters of the system, see `__init__()` for details on their definition. Long story short, it must contain $a_2, a_1, a_0, \alpha_0, \alpha_1, \beta_0$ and β_1 .

- **count** (*int*) – Amount of eigenvalues to generate.
- **extended_output** (*bool*) – If true, not only eigenvalues but also the corresponding characteristic roots and coefficients of the eigenvectors are returned. Defaults to False.

Keyword Arguments **debug** (*bool*) – If provided, this parameter will cause several debug windows to open.

Returns λ , ordered in increasing order or tuple of $(\lambda, p, \boldsymbol{\kappa})$ if *extended_output* is True.

Return type array or tuple of arrays

static convert_to_characteristic_root (*params*, *eigenvalue*)

Converts a given eigenvalue λ into a characteristic root p by using the provided parameters. The relation is given by

$$p = -\frac{a_1}{a_2} + j\sqrt{\frac{a_0 - \lambda}{a_2} - \left(\frac{a_1}{2a_2}\right)^2}$$

Parameters

- **params** (*bunch*) – system parameters, see `cure_hint()` .
- **eigenvalue** (*real*) – eigenvalue λ

Returns characteristic root p

Return type complex number

static convert_to_eigenvalue (*params*, *char_roots*)

Converts a pair of characteristic roots $p_{1,2}$ into an eigenvalue λ by using the provided parameters. The relation is given by

$$\lambda = a_2 p^2 + a_1 p + a_0$$

Parameters

- **params** (*SecondOrderOperator*) – System parameters.
- **char_roots** (*tuple or array of tuples*) – Characteristic roots

static cure_interval (*interval*, *params*, *count*, *derivative_order*, ***kwargs*)

Helper to cure an interval with eigenvectors.

Parameters

- **interval** (*Domain*) – Domain of the spatial problem.
- **params** (*SecondOrderOperator*) – Parameters of the system, see `__init__()` for details on their definition. Long story short, it must contain $a_2, a_1, a_0, \alpha_0, \alpha_1, \beta_0$ and β_1 .
- **count** (*int*) – Amount of eigenvectors to generate.
- **derivative_order** (*int*) – Amount of derivative handles to provide.
- **kwargs** – will be passed to `calculate_eigenvalues()`

Keyword Arguments **debug** (*bool*) – If provided, this parameter will cause several debug windows to open.

Returns An array holding the eigenvalues paired with a basis spanned by the eigenvectors.

Return type tuple of (array, *Base*)

class SecondOrderEigenfunction (**args*, ***kwargs*)

Bases: `pyinduct.shapefunctions.ShapeFunction`

Wrapper for all eigenvalue problems of the form

$$a_2\varphi''(z) + a_1\varphi'(z) + a_0\varphi(z) = \lambda\varphi(z), \qquad a_2, a_1, a_0, \lambda \in \mathbb{C}$$

with eigenfunctions φ and eigenvalues λ. The roots of the characteristic equation (belonging to the ode) are denoted by

$$p = \eta \pm j\omega, \qquad \eta \in \mathbb{R}, \quad \omega \in \mathbb{C}$$

$$\eta = -\frac{a_1}{2a_2}, \quad \omega = \sqrt{-\frac{a_1^2}{4a_2^2} + \frac{a_0 - \lambda}{a_2}}$$

In the following the variable ω is called an eigenfrequency.

classmethod cure_interval(*cls*, *interval*, *param=None*, *n=None*, *eig_val=None*, *eig_freq=None*, *max_order=2*, *scale=None*)

Provide the first *n* eigenvalues and eigenfunctions (wraped inside a pyinduct base). For the exact formulation of the considered eigenvalue problem, have a look at the docstring from the eigenfunction class from which you will call this method.

You must call this *classmethod* with one and only one of the kwargs:

- *n* (*eig_val* and *eig_freq* will be computed with the `eigfreq_eigval_hint()`)
- *eig_val* (*eig_freq* will be calculated with `eigval_tf_eigfreq()`)
- *eig_freq* (*eig_val* will be calculated with `eigval_tf_eigfreq()`),

or (and) pass the kwarg scale (then n is set to len(scale)). If you have the kwargs *eig_val* and *eig_freq* already calculated then these are preferable, in the sense of performance.

Parameters **interval** (`Domain`) – Domain/Interval of the eigenvalue problem.

Keyword Arguments

- **param** – Parameters $(a_2, a_1, a_0, ...)$ see *evp_class.__doc__*.
- **n** – Number of eigenvalues/eigenfunctions to compute.
- **eig_freq** (*array_like*) – Pass your own choice of eigenfrequencies here.
- **eig_val** (*array_like*) – Pass your own choice of eigenvalues here.
- **max_order** – Maximum derivative order which must provided by the eigenfunctions.
- **scale** (*array_like*) – Here you can pass a list of values to scale the eigenfunctions.

Returns

- eigenvalues (numpy.array)
- eigenfunctions (`Base`)

Return type tuple

static eigfreq_eigval_hint(*param*, *l*, *n_roots*)

Parameters

- **param** (*array_like*) – Parameters $(a_2, a_1, a_0, None, None)$.
- **l** – End of the domain $z \in [0, 1]$.
- **n_roots** (*int*) – Number of eigenfrequencies/eigenvalues to be compute.

Returns

Booth tuple elements are numpy.ndarrays of the same length, one for eigenfrequencies and one for eigenvalues.

$$\Big(\big[\omega_1, ..., \omega_{\text{n\_roots}}\big], \big[\lambda_1, ..., \lambda_{\text{n\_roots}}\big]\Big)$$

Return type tuple

static eigval_tf_eigfreq(*param*, *eig_val=None*, *eig_freq=None*)

Provide corresponding of eigenvalues/eigenfrequencies for given eigenfreqeuncies/eigenvalues, depending on which type is given.

$$\omega = \sqrt{-\frac{a_1^2}{4a_2^2} + \frac{a_0 - \lambda}{a_2}}$$

respectively

$$\lambda = -\frac{a_1^2}{4a_2} + a_0 - a_2\omega.$$

Parameters

- **param** (*array_like*) – Parameters $(a_2, a_1, a_0, None, None)$.
- **eig_val** (*array_like*) – Eigenvalues λ.
- **eig_freq** (*array_like*) – Eigenfrequencies ω.

Returns Eigenfrequencies ω or eigenvalues λ.

Return type numpy.array

static get_adjoint_problem(*param*)

Return the parameters of the adjoint eigenvalue problem for the given parameter set. Hereby, dirichlet or robin boundary condition at $z = 0$

$$\varphi(0) = 0 \quad \text{or} \quad \varphi'(0) = \alpha\varphi(0)$$

and dirichlet or robin boundary condition at $z = l$

$$\varphi`(l) = 0 \quad \text{or} \quad \varphi'(l) = -\beta\varphi(l)$$

can be imposed.

Parameters **param** (*array_like*) – To define a homogeneous dirichlet boundary condition set alpha or beta to *None* at the corresponding side. Possibilities:

- $\left(a_2, a_1, a_0, \alpha, \beta\right)^T$,
- $\left(a_2, a_1, a_0, None, \beta\right)^T$,
- $\left(a_2, a_1, a_0, \alpha, None\right)^T$ or
- $\left(a_2, a_1, a_0, None, None\right)^T$.

Returns

Parameters $\left(a_2, \tilde{a}_1, a_0, \tilde{\alpha}, \tilde{\beta}\right)$ for the adjoint problem

$$\begin{aligned} a_2\psi''(z) + \tilde{a}_1\psi'(z) + a_0\psi(z) &= \lambda\psi(z) \\ \psi(0) = 0 \quad &\text{or} \quad \psi'(0) = \tilde{\alpha}\psi(0) \\ \psi`(l) = 0 \quad &\text{or} \quad \psi'(l) = -\tilde{\beta}\psi(l) \end{aligned}$$

with

$$\tilde{a}_1 = -a_1, \quad \tilde{\alpha} = \frac{a_1}{a_2}\alpha, \quad \tilde{\beta} = -\frac{a_1}{a_2}\beta.$$

Return type tuple

class SecondOrderOperator(*a2=0, a1=0, a0=0, alpha1=0, alpha0=0, beta1=0, beta0=0, domain=- np.inf, np.inf*)

Interface class to collect all important parameters that describe a second order ordinary differential equation.

Parameters

- **a2** (*Number or callable*) – coefficient a_2.
- **a1** (*Number or callable*) – coefficient a_1.
- **a0** (*Number or callable*) – coefficient a_0.
- **alpha1** (*Number*) – coefficient α_1.
- **alpha0** (*Number*) – coefficient α_0.
- **beta1** (*Number*) – coefficient β_1.
- **beta0** (*Number*) – coefficient β_0.

static from_dict(*param_dict, domain=None*)

static from_list(*param_list, domain=None*)

get_adjoint_problem(*self*)

Return the parameters of the operator A^* describing the the problem

$$(\mathrm{A}^*\psi)(z) = \bar a_2 \partial_z^2 \psi(z) + \bar a_1 \partial_z \psi(z) + \bar a_0 \psi(z) \ ,$$

where the $\bar a_i$ are constant and whose boundary conditions are given by

$$\begin{aligned} \bar\alpha_1 \partial_z \psi(z_1) + \bar\alpha_0 \psi(z_1) &= 0 \\ \bar\beta_1 \partial_z \psi(z_2) + \bar\beta_0 \psi(z_2) &= 0. \end{aligned}$$

The following mapping is used:

$$\begin{aligned} \bar a_2 = a_2, \quad \bar a_1 = -a_1, \quad \bar a_0 = a_0, \\ \bar\alpha_1 = -1, \quad \bar\alpha_0 = \frac{a_1}{a_2} - \frac{\alpha_0}{\alpha_1}, \\ \bar\beta_1 = -1, \quad \bar\beta_0 = \frac{a_1}{a_2} - \frac{\beta_0}{\beta_1}\ . \end{aligned}$$

Returns Parameter set describing A^* .

Return type *SecondOrderOperator*

class SecondOrderRobinEigenfunction(*om, param, l, scale=1, max_der_order=2*)

Bases: *pyinduct.eigenfunctions.SecondOrderEigenfunction*

This class provides an eigenfunction $\varphi(z)$ to the eigenvalue problem given by

$$\begin{aligned} a_2\varphi''(z) + a_1\varphi'(z) + a_0\varphi(z) &= \lambda\varphi(z) \\ \varphi'(0) &= \alpha\varphi(0) \\ \varphi'(l) &= -\beta\varphi(l). \end{aligned}$$

The eigenfrequency $\omega = \sqrt{-\frac{a_1^2}{4a_2^2} + \frac{a_0-\lambda}{a_2}}$ must be provided (for example with the *eigfreq_eigval_hint()* of this class).

Parameters

- **om** (*numbers.Number*) – eigenfrequency ω
- **param** (*array_like*) – $\left(a_2, a_1, a_0, \alpha, \beta\right)^T$
- **l** (*numbers.Number*) – End of the domain $z \in [0, l]$.
- **scale** (*numbers.Number*) – Factor to scale the eigenfunctions (corresponds to $\varphi(0) =$ phi_0).

- **max_der_order** (*int*) – Number of derivative handles that are needed.

static eigfreq_eigval_hint (*param*, *l*, *n_roots*, *show_plot=False*)

Return the first *n_roots* eigenfrequencies ω and eigenvalues λ.

$$\omega_i = \sqrt{-\frac{a_1^2}{4a_2^2} + \frac{a_0 - \lambda_i}{a_2}} \quad i = 1, \dots, \text{n\_roots}$$

to the considered eigenvalue problem.

Parameters

- **param** (*array_like*) – Parameters $\left(a_2, a_1, a_0, \alpha, \beta\right)^T$
- **l** (*numbers.Number*) – Right boundary value of the domain $[0, l] \ni z$.
- **n_roots** (*int*) – Amount of eigenfrequencies to compute.
- **show_plot** (*bool*) – Show a plot window of the characteristic equation.

Returns

$$\left(\left[\omega_1, \dots, \omega_{\text{n\_roots}}\right], \left[\lambda_1, \dots, \lambda_{\text{n\_roots}}\right]\right)$$

Return type tuple –> booth tuple elements are numpy.ndarrays of length *nroots*

class ShapeFunction (**args*, ***kwargs*)

Bases: `pyinduct.core.Function`

Base class for approximation functions with compact support.

When a continuous variable of e.g. space and time $x(z, t)$ is decomposed in a series $\tilde{x} = \sum_{i=1}^{\infty} \varphi_i(z) c_i(t)$ the $\varphi_i(z)$ denote the shape functions.

classmethod cure_interval (*cls*, *interval*, ***kwargs*)

Create a network or set of functions from this class and return an approximation base (`Base`) on the given interval.

The `kwargs` may hold the order of approximation or the amount of functions to use. Use them in your child class as needed.

If you don't need to now from which class this method is called, overwrite the `@classmethod` decorator in the child class with the `@staticmethod` decorator.

Short reference: Inside a `@staticmethod` you know nothing about the class from which it is called and you can just play with the given parameters. Inside a `@classmethod` you can additionally operate on the class, since the first parameter is always the class itself.

Parameters

- **interval** (`Domain`) – Interval to cure.
- ****kwargs** – Various arguments, depending on the implementation.

Returns Approximation base, generated by the created shape functions.

Return type `Base`

class TransformedSecondOrderEigenfunction (*target_eigenvalue*, *init_state_vector*, *dgl_coefficients*, *domain*)

Bases: `pyinduct.core.Function`

This class provides an eigenfunction $\varphi(z)$ to the eigenvalue problem given by

$$a_2(z)\varphi''(z) + a_1(z)\varphi'(z) + a_0(z)\varphi(z) = \lambda\varphi(z) \quad ,$$

where $\lambda \in \mathbb{C}$ denotes an eigenvalue and $z \in [z_0, \dots, z_n]$ the domain.

Parameters

- **target_eigenvalue** (*numbers.Number*) – λ
- **init_state_vector** (*array_like*) –

$$\Big(\mathrm{Re}\{\varphi(0)\}, \mathrm{Re}\{\varphi'(0)\}, \mathrm{Im}\{\varphi(0)\}, \mathrm{Im}\{\varphi'(0)\}\Big)^T$$

- **dgl_coefficients** (*array_like*) – Function handles $\Big(a2(z), a1(z), a0(z)\Big)^T$.
- **domain** (*Domain*) – Spatial domain of the problem.

find_roots (*function*, *grid*, *n_roots=None*, *rtol=1e-05*, *atol=1e-08*, *cmplx=False*, *sort_mode='norm'*)

Searches *n_roots* roots of the *function* $f(\boldsymbol{x})$ on the given *grid* and checks them for uniqueness with aid of *rtol*.

In Detail `scipy.optimize.root()` is used to find initial candidates for roots of $f(\boldsymbol{x})$. If a root satisfies the criteria given by atol and rtol it is added. If it is already in the list, a comprehension between the already present entries' error and the current error is performed. If the newly calculated root comes with a smaller error it supersedes the present entry.

Raises **ValueError** – If the demanded amount of roots can't be found.

Parameters

- **function** (*callable*) – Function handle for math:*f(boldsymbol{x})* whose roots shall be found.
- **grid** (*list*) – Grid to use as starting point for root detection. The i th element of this list provides sample points for the i th parameter of $\boldsymbol{x}$.
- **n_roots** (*int*) – Number of roots to find. If none is given, return all roots that could be found in the given area.
- **rtol** – Tolerance to be exceeded for the difference of two roots to be unique: $f(r1) - f(r2) > \mathrm{rtol}$.
- **atol** – Absolute tolerance to zero: $f(x^0) < \mathrm{atol}$.
- **cmplx** (*bool*) – Set to True if the given *function* is complex valued.
- **sort_mode** (*str*) – Specify tho order in which the extracted roots shall be sorted. Default "norm" sorts entries by their l_2 norm, while "component" will sort them in increasing order by every component.

Returns numpy.ndarray of roots; sorted in the order they are returned by $f(\boldsymbol{x})$.

generic_scalar_product (*b1*, *b2=None*, *scalar_product=None*)

Calculates the pairwise scalar product between the elements of the `ApproximationBase` *b1* and *b2*.

Parameters

- **b1** (`ApproximationBase`) – first basis
- **b2** (`ApproximationBase`) – second basis, if omitted defaults to *b1*
- **scalar_product** (*list of callable*) – Callbacks for product calculation. Defaults to *scalar_product_hint* from *b1*.

Note: If *b2* is omitted, the result can be used to normalize *b1* in terms of its scalar product.

normalize_base (*b1*, *b2=None*)

Takes two `ApproximationBase`'s $\boldsymbol{b}_1$, $\boldsymbol{b}_1$ and normalizes them so that $\langle \boldsymbol{b}_{1i}\, ,\, \boldsymbol{b}_{2i} \rangle = 1$. If only one base is given, $\boldsymbol{b}_2$ defaults to $\boldsymbol{b}_1$.

Parameters

- **b1** (`ApproximationBase`) – $\boldsymbol{b}_1$

- **b2** (`ApproximationBase`) – $\boldsymbol{b}_2$

Raises **`ValueError`** – If $\boldsymbol{b}_1$ and $\boldsymbol{b}_2$ are orthogonal.

Returns if *b2* is None, otherwise: Tuple of 2 `ApproximationBase`'s.

Return type `ApproximationBase`

`real`(*data*)
Check if the imaginary part of `data` vanishes and return its real part if it does.

Parameters **data** (*numbers.Number or array_like*) – Possibly complex data to check.

Raises **`ValueError`** – If provided data can't be converted within the given tolerance limit.

Returns Real part of `data`.

Return type numbers.Number or array_like

`visualize_roots`(*roots*, *grid*, *func*, *cmplx=False*, *return_window=False*)
Visualize a given set of roots by examining the output of the generating function.

Parameters

- **roots** (*array like*) – Roots to display, if *None* is given, no roots will be displayed, this is useful to get a view of *func* and choosing an appropriate *grid*.
- **grid** (*list*) – List of arrays that form the grid, used for the evaluation of the given *func*.
- **func** (*callable*) – Possibly vectorial function handle that will take input of of the shape ('len(grid)',).
- **cmplx** (*bool*) – If True, the complex valued *func* is handled as a vectorial function returning [Re(func), Im(func)].
- **return_window** (*bool*) – If True the graphics window is not shown directly. In this case, a reference to the plot window is returned.

Returns: A PgPlotWindow if *delay_exec* is True.

7.4 Registry

pyinduct.registry covers the interface for registration of bases (a base is a set of initial functions).

`clear_registry`()
Deregister all bases.

`deregister_base`(*label*)
Removes a set of initial functions from the packages registry.

Parameters **label** (*str*) – String, label of functions that are to be removed.

Raises **`ValueError`** – If label is not found in registry.

`get_base`(*label*)
Retrieve registered set of initial functions by their label.

Parameters **label** (*str*) – String, label of functions to retrieve.

Returns initial_functions

`is_registered`(*label*)
Checks whether a specific label has already been registered.

Args: label (str): Label to check for.

Returns True if registered, False if not.

Return type bool

register_base (*label*, *base*, *overwrite=False*)
Register a basis to make it accessible all over the `pyinduct` framework.

Parameters

- **base** (ApproximationBase) – base to register
- **label** (*str*) – String that will be used as label.
- **overwrite** – Force overwrite if a basis is already registered under this label.

7.5 Placeholder

In *pyinduct.placeholder* you find placeholders for symbolic Term definitions.

class FieldVariable (*function_label*, *order=0*, *0*, *weight_label=None*, *location=None*, *exponent=1*, *raised_spatially=False*)
Bases: *pyinduct.placeholder.Placeholder*

Class that represents terms of the systems field variable $x(z,t)$.

Parameters

- **function_label** (*str*) – Label of shapefunctions to use for approximation, see *register_base()* for more information about how to register an approximation basis.
- **tuple of int** (*order*) – Tuple of temporal_order and spatial_order derivation order.
- **weight_label** (*str*) – Label of weights for which coefficients are to be calculated (defaults to function_label).
- **location** – Where the expression is to be evaluated.
- **exponent** – Exponent of the term.

Examples

Assuming some shapefunctions have been registered under the label `"phi"` the following expressions hold:

- $\frac{\partial^3}{\partial t \partial z^2} x(z,t)$

```
>>> x_dt_dzz = FieldVariable("phi", order=(1, 2))
```

- $\frac{\partial^2}{\partial t^2} x(3,t)$

```
>>> x_dtt_at_3 = FieldVariable("phi", order=(2, 0), location=3)
```

class TestFunction (*function_label*, *order=0*, *location=None*, *approx_label=None*)
Bases: *pyinduct.placeholder.SpatialPlaceholder*

Class that works as a placeholder for test functions in an equation.

Parameters

- **function_label** (*str*) – Label of the function test base.
- **order** (*int*) – Spatial derivative order.
- **location** (*Number*) – Point of evaluation / argument of the function.
- **approx_label** (*str*) – Label of the approximation test base.

class Base (*fractions*, *matching_base_lbls=None*, *intermediate_base_lbls=None*)
Bases: `pyinduct.core.ApproximationBasis`

Base class for approximation bases.

In general, a `Base` is formed by a certain amount of `BaseFractions` and therefore forms finite-dimensional subspace of the distributed problem's domain. Most of the time, the user does not need to interact with this class.

Parameters

- **fractions** (iterable of `BaseFraction`) – List, array or dict of `BaseFraction`'s
- **matching_base_lbls** (*list of str*) – List of labels from exactly matching bases, for which no transformation is necessary. Useful for transformations from bases that 'live' in different function spaces but evolve with the same time dynamic/coefficients (e.g. modal bases).
- **intermediate_base_lbls** (*list of str*) – If it is certain that this base instance will be asked (as destination base) to return a transformation to a source base, whose implementation is cumbersome, its label can be provided here. This will trigger the generation of the transformation using build-in features. The algorithm, implemented in `get_weights_transformation` is then called again with the intermediate base as destination base and the 'old' source base. With this technique arbitrary long transformation chains are possible, if the provided intermediate bases again define intermediate bases.

derive (*self*, *order*)
Basic implementation of derive function. Empty implementation, overwrite to use this functionality.

Parameters **order** (`numbers.Number`) – derivative order

Returns derived object

Return type `Base`

function_space_hint (*self*)
Hint that returns properties that characterize the functional space of the fractions. It can be used to determine if function spaces match.

Note: Overwrite to implement custom functionality.

get_attribute (*self*, *attr*)
Retrieve an attribute from the fractions of the base.

Parameters **attr** (*str*) – Attribute to query the fractions for.

Returns Array of `len(fractions)` holding the attributes. With *None* entries if the attribute is missing.

Return type `np.ndarray`

raise_to (*self*, *power*)
Factory method to obtain instances of this base, raised by the given power.

Parameters **power** – power to raise the basis onto.

scalar_product_hint (*self*)
Hint that returns steps for scalar product calculation with elements of this base.

Note: Overwrite to implement custom functionality.

scale (*self*, *factor*)
Factory method to obtain instances of this base, scaled by the given factor.

Parameters **`factor`** – factor or function to scale this base with.

`transformation_hint`(*self*, *info*)

Method that provides a information about how to transform weights from one *`BaseFraction`* into another.

In Detail this function has to return a callable, which will take the weights of the source- and return the weights of the target system. It may have keyword arguments for other data which is required to perform the transformation. Information about these extra keyword arguments should be provided in form of a dictionary whose keys are keyword arguments of the returned transformation handle.

Note: This implementation covers the most basic case, where the two *`BaseFraction`*'s are of same type. For any other case it will raise an exception. Overwrite this Method in your implementation to support conversion between bases that differ from yours.

Parameters **`info`** – *`TransformationInfo`*

Raises **`NotImplementedError`** –

Returns Transformation handle

class `ConstantFunction`(*constant*, ***kwargs*)

Bases: *`pyinduct.core.Function`*

A *`Function`* that returns a constant value.

This function can be differentiated without limits.

Parameters **`constant`** (*`number`*) – value to return

Keyword Arguments **`**kwargs`** – All other kwargs get passed to *`Function`*.

`derive`(*self*, *order=1*)

Spatially derive this *`Function`*.

This is done by neglecting *order* derivative handles and to select handle order $-$ 1 as the new evaluation_handle.

Parameters **`order`** (*`int`*) – the amount of derivations to perform

Raises

- **`TypeError`** – If *order* is not of type int.
- **`ValueError`** – If the requested derivative order is higher than the provided one.

Returns *`Function`* the derived function.

class `EquationTerm`(*scale*, *arg*)

Bases: `object`

Base class for all accepted terms in a weak formulation.

Parameters

- **`scale`** –
- **`arg`** –

class `Function`(*eval_handle*, *domain=- np.inf, np.inf*, *nonzero=- np.inf, np.inf*, *derivative_handles=None*)

Bases: *`pyinduct.core.BaseFraction`*

Most common instance of a *`BaseFraction`*. This class handles all tasks concerning derivation and evaluation of functions. It is used broad across the toolbox and therefore incorporates some very specific attributes. For example, to ensure the accurateness of numerical handling functions may only evaluated in areas where they provide nonzero return values. Also their domain has to be taken into account. Therefore the attributes *domain* and *nonzero* are provided.

To save implementation time, ready to go version like `LagrangeFirstOrder` are provided in the `pyinduct.simulation` module.

For the implementation of new shape functions subclass this implementation or directly provide a callable *eval_handle* and callable *derivative_handles* if spatial derivatives are required for the application.

Parameters

- **eval_handle** (*callable*) – Callable object that can be evaluated.
- **domain** (*(list of) tuples*) – Domain on which the eval_handle is defined.
- **nonzero** (*tuple*) – Region in which the eval_handle will return
- **output. Must be a subset of domain** (*nonzero*) –
- **derivative_handles** (*list*) – List of callable(s) that contain
- **of eval_handle** (*derivatives*) –

add_neutral_element (*self*)

Return the neutral element of addition for this object.

In other words: *self + ret_val == self*.

derivative_handles (*self*)

derive (*self, order=1*)

Spatially derive this `Function`.

This is done by neglecting *order* derivative handles and to select handle $order - 1$ as the new evaluation_handle.

Parameters **order** (*int*) – the amount of derivations to perform

Raises

- **TypeError** – If *order* is not of type int.
- **ValueError** – If the requested derivative order is higher than the provided one.

Returns `Function` the derived function.

static from_data (*x, y, **kwargs*)

Create a `Function` based on discrete data by interpolating.

The interpolation is done by using `interp1d` from scipy, the *kwargs* will be passed.

Parameters

- **x** (*array-like*) – Places where the function has been evaluated .
- **y** (*array-like*) – Function values at *x*.
- ****kwargs** – all kwargs get passed to `Function` .

Returns An interpolating function.

Return type `Function`

function_handle (*self*)

function_space_hint (*self*)

Return the hint that this function is an element of the an scalar product space which is uniquely defined by the scalar product `scalar_product_hint()`.

Note: If you are working on different function spaces, you have to overwrite this hint in order to provide more properties which characterize your specific function space. For example the domain of the functions.

get_member (*self*, *idx*)
Implementation of the abstract parent method.

Since the `Function` has only one member (itself) the parameter *idx* is ignored and *self* is returned.

Parameters **idx** – ignored.

Returns self

mul_neutral_element (*self*)
Return the neutral element of multiplication for this object.

In other words: *self * ret_val == self*.

raise_to (*self*, *power*)
Raises the function to the given *power*.

> **Warning:** Derivatives are lost after this action is performed.

Parameters **power** (`numbers.Number`) – power to raise the function to

Returns raised function

scalar_product_hint (*self*)
Return the hint that the `_dot_product_l2()` has to calculated to gain the scalar product.

scale (*self*, *factor*)
Factory method to scale a `Function`.

Parameters **factor** – `numbers.Number` or a callable.

class Input (*function_handle*, *index=0*, *order=0*, *exponent=1*)
Bases: `pyinduct.placeholder.Placeholder`

Class that works as a placeholder for an input of the system.

Parameters

- **function_handle** (*callable*) – Handle that will be called by the simulation unit.
- **index** (*int*) – If the system's input is vectorial, specify the element to be used.
- **order** (*int*) – temporal derivative order of this term (See `Placeholder`).
- **exponent** (*numbers.Number*) – See `FieldVariable`.

Note: if *order* is nonzero, the callable is expected to return the temporal derivatives of the input signal by returning an array of `len(order)+1`.

class IntegralTerm (*integrand*, *limits*, *scale=1.0*)
Bases: `pyinduct.placeholder.EquationTerm`

Class that represents an integral term in a weak equation.

Parameters

- **integrand** –
- **limits** (*tuple*) –
- **scale** –

class ObserverGain (*observer_feedback*)
Bases: `pyinduct.placeholder.Placeholder`

Class that works as a placeholder for the observer error gain.

Parameters **observer_feedback** (*ObserverFeedback*) – Handle that will be called by the simulation unit.

class Placeholder (*data*, *order=0, 0*, *location=None*)
Bases: `object`

Base class that works as a placeholder for terms that are later parsed into a canonical form.

Parameters

- **data** (*arbitrary*) – data to store in the placeholder.
- **order** (*tuple*) – (temporal_order, spatial_order) derivative orders that are to be applied before evaluation.
- **location** (*numbers.Number*) – Location to evaluate at before further computation.

Todo: convert order and location into attributes with setter and getter methods. This will close the gap of unchecked values for order and location that can be sneaked in by the copy constructors by circumventing code doubling.

---

derivative (*self*, *temp_order=0*, *spat_order=0*)
Mimics a copy constructor and adds the given derivative orders.

Note: The desired derivative order `order` is added to the original order.

Parameters

- **temp_order** – Temporal derivative order to be added.
- **spat_order** – Spatial derivative order to be added.

Returns New *Placeholder* instance with the desired derivative order.

class Product (*a*, *b=None*)
Bases: `object`

Represents a product.

Parameters

- **a** –
- **b** –

get_arg_by_class (*self*, *cls*)
Extract element from product that is an instance of cls.

Parameters **cls** –

Returns

Return type list

class ScalarFunction (*function_label*, *order=0*, *location=None*)
Bases: *pyinduct.placeholder.SpatialPlaceholder*

Class that works as a placeholder for spatial functions in an equation. An example could be spatial dependent coefficients.

Parameters

- **function_label** (*str*) – label under which the function is registered
- **order** (*int*) – spatial derivative order to use

- **location** – location to evaluate at

Warns

- **There seems to be a problem when this function is used in combination**
- **with the :py:class:`.Product` class. Make sure to provide this class as**
- **first argument to any product you define.**

Todo: see warning.

---

static from_scalar (*scalar*, *label*, ***kwargs*)
create a `ScalarFunction` from scalar values.

Parameters

- **scalar** (`array like`) – Input that is used to generate the placeholder. If a number is given, a constant function will be created, if it is callable it will be wrapped in a `Function` and registered.
- **label** (`string`) – Label to register the created base.
- ****kwargs** – All kwargs that are not mentioned below will be passed to `Function`.

Keyword Arguments

- **order** (`int`) – See constructor.
- **location** (`int`) – See constructor.
- **overwrite** (`bool`) – See `register_base()`

Returns Placeholder object that can be used in a weak formulation.

Return type `ScalarFunction`

class ScalarProductTerm (*arg1*, *arg2*, *scale=1.0*)
Bases: `pyinduct.placeholder.EquationTerm`

Class that represents a scalar product in a weak equation.

Parameters

- **arg1** – Fieldvariable (Shapefunctions) to be projected.
- **arg2** – Testfunctions to project on.
- **scale** (`Number`) – Scaling of expression.

class ScalarTerm (*argument*, *scale=1.0*)
Bases: `pyinduct.placeholder.EquationTerm`

Class that represents a scalar term in a weak equation.

Parameters

- **argument** –
- **scale** –

class Scalars (*values*, *target_term=None*, *target_form=None*, *test_func_lbl=None*)
Bases: `pyinduct.placeholder.Placeholder`

Placeholder for scalar values that scale the equation system, gained by the projection of the pde onto the test basis.

Note: The arguments *target_term* and *target_form* are used inside the parser. For frontend use, just specify the *values*.

Parameters

- **values** – Iterable object containing the scalars for every k-th equation.
- **target_term** – Coefficient matrix to `add_to()`.
- **target_form** – Desired weight set.

class SpatialDerivedFieldVariable(*function_label*, *order*, *weight_label=None*, *location=None*)

Bases: `pyinduct.placeholder.FieldVariable`

Class that represents terms of the systems field variable $x(z, t)$.

Parameters

- **function_label** (*str*) – Label of shapefunctions to use for approximation, see `register_base()` for more information about how to register an approximation basis.
- **tuple of int** (*order*) – Tuple of temporal_order and spatial_order derivation order.
- **weight_label** (*str*) – Label of weights for which coefficients are to be calculated (defaults to function_label).
- **location** – Where the expression is to be evaluated.
- **exponent** – Exponent of the term.

Examples

Assuming some shapefunctions have been registered under the label `"phi"` the following expressions hold:

- $\frac{\partial^3}{\partial t \partial z^2} x(z, t)$

```
>>> x_dt_dzz = FieldVariable("phi", order=(1, 2))
```

- $\frac{\partial^2}{\partial t^2} x(3, t)$

```
>>> x_dtt_at_3 = FieldVariable("phi", order=(2, 0), location=3)
```

class SpatialPlaceholder(*data*, *order=0*, *location=None*)

Bases: `pyinduct.placeholder.Placeholder`

Base class for all spatially-only dependent placeholders. The deeper meaning of this abstraction layer is to offer an easier to use interface.

derive(*self*, *order=1*)

Take the (spatial) derivative of this object. :param order: Derivative order.

Returns The derived expression.

Return type `Placeholder`

class TemporalDerivedFieldVariable(*function_label*, *order*, *weight_label=None*, *location=None*)

Bases: `pyinduct.placeholder.FieldVariable`

Class that represents terms of the systems field variable $x(z, t)$.

Parameters

- **function_label** (*str*) – Label of shapefunctions to use for approximation, see `register_base()` for more information about how to register an approximation basis.

- **tuple of int** (*order*) – Tuple of temporal_order and spatial_order derivation order.
- **weight_label** (*str*) – Label of weights for which coefficients are to be calculated (defaults to function_label).
- **location** – Where the expression is to be evaluated.
- **exponent** – Exponent of the term.

Examples

Assuming some shapefunctions have been registered under the label "phi" the following expressions hold:

- $\frac{\partial^3}{\partial t \partial z^2} x(z, t)$

```
>>> x_dt_dzz = FieldVariable("phi", order=(1, 2))
```

- $\frac{\partial^2}{\partial t^2} x(3, t)$

```
>>> x_dtt_at_3 = FieldVariable("phi", order=(2, 0), location=3)
```

evaluate_placeholder_function(*placeholder*, *input_values*)
Evaluate a given placeholder object, that contains functions.

Parameters

- **placeholder** – Instance of *FieldVariable*, *TestFunction* or *ScalarFunction*.
- **input_values** – Values to evaluate at.

Returns numpy.ndarray of results.

get_base(*label*)
Retrieve registered set of initial functions by their label.

Parameters **label** (*str*) – String, label of functions to retrieve.

Returns initial_functions

get_common_form(*placeholders*)
Extracts the common target form from a list of scalars while making sure that the given targets are equivalent.

Parameters **placeholders** – Placeholders with possibly differing target forms.

Returns Common target form.

Return type str

get_common_target(*scalars*)
Extracts the common target from list of scalars while making sure that targets are equivalent.

Parameters **scalars** (*Scalars*) –

Returns Common target.

Return type dict

is_registered(*label*)
Checks whether a specific label has already been registered.

Args: label (str): Label to check for.

Returns True if registered, False if not.

Return type bool

register_base (*label*, *base*, *overwrite=False*)
Register a basis to make it accessible all over the `pyinduct` framework.

Parameters

- **base** (`ApproximationBase`) – base to register
- **label** (*str*) – String that will be used as label.
- **overwrite** – Force overwrite if a basis is already registered under this label.

sanitize_input (*input_object*, *allowed_type*)
Sanitizes input data by testing if *input_object* is an array of type *allowed_type*.

Parameters

- **input_object** – Object which is to be checked.
- **allowed_type** – desired type

Returns input_object

7.6 Simulation

Simulation infrastructure with helpers and data structures for preprocessing of the given equations and functions for postprocessing of simulation data.

class CanonicalEquation (*name*, *dominant_lbl=None*)
Bases: `object`

Wrapper object, holding several entities of canonical forms for different weight-sets that form an equation when summed up. After instantiation, this object can be filled with information by passing the corresponding coefficients to `add_to()`. When the parsing process is completed and all coefficients have been collected, calling `finalize()` is required to compute all necessary information for further processing. When finalized, this object provides access to the dominant form of this equation.

Parameters

- **name** (*str*) – Unique identifier of this equation.
- **dominant_lbl** (*str*) – Label of the variable that dominates this equation.

add_to (*self*, *weight_label*, *term*, *val*, *column=None*)
Add the provided *val* to the canonical form for *weight_label*, see `CanonicalForm.add_to()` for further information.

Parameters

- **weight_label** (*str*) – Basis to add onto.
- **term** – Coefficient to add onto, see `add_to()`.
- **val** – Values to add.
- **column** (*int*) – passed to `add_to()`.

dominant_form (*self*)
direct access to the dominant `CanonicalForm`.

Note: `finalize()` must be called first.

Returns the dominant canonical form

Return type `CanonicalForm`

finalize(*self*)

Finalize the Object. After the complete formulation has been parsed and all terms have been sorted into this Object via `add_to()` this function has to be called to inform this object about it. Furthermore, the f and G parts of the static_form will be copied to the dominant form for easier state-space transformation.

Note: This function must be called to use the `dominant_form` attribute.

finalize_dynamic_forms(*self*)

Finalize all dynamic forms. See method `CanonicalForm.finalize()`.

get_dynamic_terms(*self*)

Returns Dictionary of terms for each weight set.

Return type dict

get_static_terms(*self*)

Returns Terms that do not depend on a certain weight set.

input_function(*self*)

The input handles for the equation.

set_input_function(*self*, *func*)

static_form(*self*)

`WeakForm` that does not depend on any weights. :return:

class CanonicalForm(*name=None*)

Bases: `object`

The canonical form of an nth order ordinary differential equation system.

add_to(*self*, *term*, *value*, *column=None*)

Adds the value `value` to term `term`. `term` is a dict that describes which coefficient matrix of the canonical form the value shall be added to.

Parameters

- **term** (*dict*) – Targeted term in the canonical form h. It has to contain:
  - name: Type of the coefficient matrix: 'E', 'f', or 'G'.
  - order: Temporal derivative order of the assigned weights.
  - exponent: Exponent of the assigned weights.
- **value** (`numpy.ndarray`) – Value to add.
- **column** (*int*) – Add the value only to one column of term (useful if only one dimension of term is known).

convert_to_state_space(*self*)

Convert the canonical ode system of order n a into an ode system of order 1.

Note: This will only work if the highest derivative order of the given form can be isolated. This is the case if the highest order is only present in one power and the equation system can therefore be solved for it.

Returns

Return type `StateSpace` object

finalize (*self*)
Finalizes the object. This method must be called after all terms have been added by `add_to()` and before `convert_to_state_space()` can be called. This functions makes sure that the formulation can be converted into state space form (highest time derivative only comes in one power) and collects information like highest derivative order, it's power and the sizes of current and state-space state vector (*dim_x* resp. *dim_xb*). Furthermore, the coefficient matrix of the highest derivative order *e_n_pb* and it's inverse are made accessible.

get_terms (*self*)
Return all coefficient matrices of the canonical formulation.

Returns Structure: Type > Order > Exponent.

Return type Cascade of dictionaries

input_function (*self*)

set_input_function (*self, func*)

class Domain (*bounds=None, num=None, step=None, points=None*)
Bases: `object`

Helper class that manages ranges for data evaluation, containing parameters.

Parameters

- **bounds** (*tuple*) – Interval bounds.
- **num** (*int*) – Number of points in interval.
- **step** (*numbers.Number*) – Distance between points (if homogeneous).
- **points** (*array_like*) – Points themselves.

Note: If num and step are given, num will take precedence.

---

bounds (*self*)

ndim (*self*)

points (*self*)

step (*self*)

class EmptyInput (*dim*)
Bases: `pyinduct.simulation.SimulationInput`

Base class for all objects that want to act as an input for the time-step simulation.

The calculated values for each time-step are stored in internal memory and can be accessed by `get_results()` (after the simulation is finished).

Note: Due to the underlying solver, this handle may get called with time arguments, that lie outside of the specified integration domain. This should not be a problem for a feedback controller but might cause problems for a feedforward or trajectory implementation.

---

class EquationTerm (*scale, arg*)
Bases: `object`

Base class for all accepted terms in a weak formulation.

Parameters

- **scale** –
- **arg** –

class EvalData(*input_data*, *output_data*, *input_labels=None*, *input_units=None*, *enable_extrapolation=False*, *fill_axes=False*, *fill_value=None*, *name=None*)

This class helps managing any kind of result data.

The data gained by evaluation of a function is stored together with the corresponding points of its evaluation. This way all data needed for plotting or other postprocessing is stored in one place. Next to the points of the evaluation the names and units of the included axes can be stored. After initialization an interpolator is set up, so that one can interpolate in the result data by using the overloaded `__call__()` method.

Parameters

- **input_data** – (List of) array(s) holding the axes of a regular grid on which the evaluation took place.
- **output_data** – The result of the evaluation.

Keyword Arguments

- **input_labels** – (List of) labels for the input axes.
- **input_units** – (List of) units for the input axes.
- **name** – Name of the generated data set.
- **fill_axes** – If the dimension of *output_data* is higher than the length of the given *input_data* list, dummy entries will be appended until the required dimension is reached.
- **enable_extrapolation** (*bool*) – If True, internal interpolators will allow extrapolation. Otherwise, the last giben value will be repeated for 1D cases and the result will be padded with zeros for cases > 1D.
- **fill_value** – If invalid data is encountered, it will be replaced with this value before interpolation is performed.

Examples

When instantiating 1d EvalData objects, the list can be omitted

```
>>> axis = Domain((0, 10), 5)
>>> data = np.random.rand(5,)
>>> e_1d = EvalData(axis, data)
```

For other cases, input_data has to be a list

```
>>> axis1 = Domain((0, 0.5), 5)
>>> axis2 = Domain((0, 1), 11)
>>> data = np.random.rand(5, 11)
>>> e_2d = EvalData([axis1, axis2], data)
```

Adding two Instances (if the boundaries fit, the data will be interpolated on the more coarse grid.) Same goes for subtraction and multiplication.

```
>>> e_1 = EvalData(Domain((0, 10), 5), np.random.rand(5,))
>>> e_2 = EvalData(Domain((0, 10), 10), 100*np.random.rand(5,))
>>> e_3 = e_1 + e_2
>>> e_3.output_data.shape
(5,)
```

Interpolate in the output data by calling the object

```
>>> e_4 = EvalData(np.array(range(5)), 2*np.array(range(5))))
>>> e_4.output_data
array([0, 2, 4, 6, 8])
>>> e_5 = e_4([2, 5])
```

(continues on next page)

(continued from previous page)

```
>>> e_5.output_data
array([4, 8])
>>> e_5.output_data.size
2
```

one may also give a slice

```
>>> e_6 = e_4(slice(1, 5, 2))
>>> e_6.output_data
array([2., 6.])
>>> e_5.output_data.size
2
```

For multi-dimensional interpolation a list has to be provided

```
>>> e_7 = e_2d([[.1, .5], [.3, .4, .7)])
>>> e_7.output_data.shape
(2, 3)
```

abs (*self*)
Get the absolute value of the elements form *self.output_data* .

Returns `EvalData` with self.input_data and output_data as result of absolute value calculation.

add (*self*, *other*, *from_left=True*)
Perform the element-wise addition of the output_data arrays from *self* and *other*

This method is used to support addition by implementing __add__ (fromLeft=True) and __radd__(fromLeft=False)). If *other*** is a `EvalData`, the *input_data* lists of *self* and *other* are adjusted using `adjust_input_vectors()` The summation operation is performed on the interpolated output_data. If *other* is a `numbers.Number` it is added according to numpy's broadcasting rules.

Parameters

- **other** (`numbers.Number` or `EvalData`) – Number or EvalData object to add to self.
- **from_left** (*bool*) – Perform the addition from left if True or from right if False.

Returns `EvalData` with adapted input_data and output_data as result of the addition.

adjust_input_vectors (*self*, *other*)
Check the the inputs vectors of *self* and *other* for compatibility (equivalence) and harmonize them if they are compatible.

The compatibility check is performed for every input_vector in particular and examines whether they share the same boundaries. and equalize to the minimal discretized axis. If the amount of discretization steps between the two instances differs, the more precise discretization is interpolated down onto the less precise one.

Parameters **other** (`EvalData`) – Other EvalData class.

Returns

- (list) - New common input vectors.
- (numpy.ndarray) - Interpolated self output_data array.
- (numpy.ndarray) - Interpolated other output_data array.

Return type tuple

interpolate (*self*, *interp_axis*)
Main interpolation method for output_data.

If one of the output dimensions is to be interpolated at one single point, the dimension of the output will decrease by one.

Parameters

- **interp_axis** (*list(list)*) – axis positions in the form
- **1D** (*-*) – axis with axis=[1,2,3]
- **2D** (*-*) – [axis1, axis2] with axis1=[1,2,3] and axis2=[0,1,2,3,4]

Returns *EvalData* with *interp_axis* as new input_data and interpolated output_data.

matmul (*self, other, from_left=True*)
Perform the matrix multiplication of the output_data arrays from *self* and *other* .

This method is used to support matrix multiplication (@) by implementing __matmul__ (from_left=True) and __rmatmul__(from_left=False)). If *other*** is a *EvalData*, the *input_data* lists of *self* and *other* are adjusted using *adjust_input_vectors()*. The matrix multiplication operation is performed on the interpolated output_data. If *other* is a `numbers.Number` it is handled according to numpy's broadcasting rules.

Parameters

- **other** (*EvalData*) – Object to multiply with.
- **from_left** (*boolean*) – Matrix multiplication from left if True or from right if False.

Returns *EvalData* with adapted input_data and output_data as result of matrix multiplication.

mul (*self, other, from_left=True*)
Perform the element-wise multiplication of the output_data arrays from *self* and *other* .

This method is used to support multiplication by implementing __mul__ (from_left=True) and __rmul__(from_left=False)). If *other*** is a *EvalData*, the *input_data* lists of *self* and *other* are adjusted using *adjust_input_vectors()*. The multiplication operation is performed on the interpolated output_data. If *other* is a `numbers.Number` it is handled according to numpy's broadcasting rules.

Parameters

- **other** (`numbers.Number` or *EvalData*) – Factor to multiply with.
- **boolean** (*from_left*) – Multiplication from left if True or from right if False.

Returns *EvalData* with adapted input_data and output_data as result of multiplication.

sqrt (*self*)
Radicate the elements form *self.output_data* element-wise.

Returns *EvalData* with self.input_data and output_data as result of root calculation.

sub (*self, other, from_left=True*)
Perform the element-wise subtraction of the output_data arrays from *self* and *other* .

This method is used to support subtraction by implementing __sub__ (from_left=True) and __rsub__(from_left=False)). If *other*** is a *EvalData*, the *input_data* lists of *self* and *other* are adjusted using *adjust_input_vectors()*. The subtraction operation is performed on the interpolated output_data. If *other* is a `numbers.Number` it is handled according to numpy's broadcasting rules.

Parameters

- **other** (`numbers.Number` or *EvalData*) – Number or EvalData object to subtract.
- **from_left** (*boolean*) – Perform subtraction from left if True or from right if False.

Returns `EvalData` with adapted input_data and output_data as result of subtraction.

class FieldVariable(*function_label, order=0, 0, weight_label=None, location=None, exponent=1, raised_spatially=False*)
Bases: `pyinduct.placeholder.Placeholder`

Class that represents terms of the systems field variable $x(z,t)$.

Parameters

- **function_label** (`str`) – Label of shapefunctions to use for approximation, see `register_base()` for more information about how to register an approximation basis.
- **tuple of int** (`order`) – Tuple of temporal_order and spatial_order derivation order.
- **weight_label** (`str`) – Label of weights for which coefficients are to be calculated (defaults to function_label).
- **location** – Where the expression is to be evaluated.
- **exponent** – Exponent of the term.

Examples

Assuming some shapefunctions have been registered under the label `"phi"` the following expressions hold:

- $\frac{\partial^3}{\partial t \partial z^2} x(z,t)$

```
>>> x_dt_dzz = FieldVariable("phi", order=(1, 2))
```

- $\frac{\partial^2}{\partial t^2} x(3,t)$

```
>>> x_dtt_at_3 = FieldVariable("phi", order=(2, 0), location=3)
```

derive(*self, *, temp_order=0, spat_order=0*)
Derive the expression to the specified order.

Parameters

- **temp_order** – Temporal derivative order.
- **spat_order** – Spatial derivative order.

Returns The derived expression.

Return type `Placeholder`

Note: This method uses keyword only arguments, which means that a call will fail if the arguments are passed by order.

---

class Function(*eval_handle, domain=- np.inf, np.inf, nonzero=- np.inf, np.inf, derivative_handles=None*)
Bases: `pyinduct.core.BaseFraction`

Most common instance of a `BaseFraction`. This class handles all tasks concerning derivation and evaluation of functions. It is used broad across the toolbox and therefore incorporates some very specific attributes. For example, to ensure the accurateness of numerical handling functions may only evaluated in areas where they provide nonzero return values. Also their domain has to be taken into account. Therefore the attributes *domain* and *nonzero* are provided.

To save implementation time, ready to go version like `LagrangeFirstOrder` are provided in the `pyinduct.simulation` module.

For the implementation of new shape functions subclass this implementation or directly provide a callable *eval_handle* and callable *derivative_handles* if spatial derivatives are required for the application.

Parameters

- **eval_handle** (*callable*) – Callable object that can be evaluated.
- **domain** (*(list of) tuples*) – Domain on which the eval_handle is defined.
- **nonzero** (*tuple*) – Region in which the eval_handle will return
- **output. Must be a subset of domain** (*nonzero*) –
- **derivative_handles** (*list*) – List of callable(s) that contain
- **of eval_handle** (*derivatives*) –

add_neutral_element (*self*)
Return the neutral element of addition for this object.

In other words: *self* + *ret_val* == *self*.

derivative_handles (*self*)

derive (*self*, *order=1*)
Spatially derive this `Function`.

This is done by neglecting *order* derivative handles and to select handle order $-$ 1 as the new evaluation_handle.

Parameters **order** (*int*) – the amount of derivations to perform

Raises

- **TypeError** – If *order* is not of type int.
- **ValueError** – If the requested derivative order is higher than the provided one.

Returns `Function` the derived function.

static from_data (*x*, *y*, ***kwargs*)
Create a `Function` based on discrete data by interpolating.

The interpolation is done by using `interp1d` from scipy, the *kwargs* will be passed.

Parameters

- **x** (*array-like*) – Places where the function has been evaluated .
- **y** (*array-like*) – Function values at *x*.
- ****kwargs** – all kwargs get passed to `Function` .

Returns An interpolating function.

Return type `Function`

function_handle (*self*)

function_space_hint (*self*)
Return the hint that this function is an element of the an scalar product space which is uniquely defined by the scalar product `scalar_product_hint()`.

Note: If you are working on different function spaces, you have to overwrite this hint in order to provide more properties which characterize your specific function space. For example the domain of the functions.

---

get_member (*self*, *idx*)
Implementation of the abstract parent method.

Since the `Function` has only one member (itself) the parameter *idx* is ignored and *self* is returned.

Parameters **idx** – ignored.

Returns self

mul_neutral_element (*self*)

Return the neutral element of multiplication for this object.

In other words: *self * ret_val == self*.

raise_to (*self*, *power*)

Raises the function to the given *power*.

> **Warning:** Derivatives are lost after this action is performed.

Parameters **power** (`numbers.Number`) – power to raise the function to

Returns raised function

scalar_product_hint (*self*)

Return the hint that the `_dot_product_l2()` has to calculated to gain the scalar product.

scale (*self*, *factor*)

Factory method to scale a *Function*.

Parameters **factor** – `numbers.Number` or a callable.

class Input (*function_handle*, *index=0*, *order=0*, *exponent=1*)

Bases: *pyinduct.placeholder.Placeholder*

Class that works as a placeholder for an input of the system.

Parameters

- **function_handle** (*callable*) – Handle that will be called by the simulation unit.
- **index** (*int*) – If the system's input is vectorial, specify the element to be used.
- **order** (*int*) – temporal derivative order of this term (See *Placeholder*).
- **exponent** (*numbers.Number*) – See *FieldVariable*.

> **Note:** if *order* is nonzero, the callable is expected to return the temporal derivatives of the input signal by returning an array of `len(order)+1`.

class IntegralTerm (*integrand*, *limits*, *scale=1.0*)

Bases: *pyinduct.placeholder.EquationTerm*

Class that represents an integral term in a weak equation.

Parameters

- **integrand** –
- **limits** (*tuple*) –
- **scale** –

class ObserverGain (*observer_feedback*)

Bases: *pyinduct.placeholder.Placeholder*

Class that works as a placeholder for the observer error gain.

Parameters **observer_feedback** (*ObserverFeedback*) – Handle that will be called by the simulation unit.

class Parameters(***kwargs*)

Handy class to collect system parameters. This class can be instantiated with a dict, whose keys will the become attributes of the object. (Bunch approach)

Parameters `kwargs` – parameters

class ScalarProductTerm(*arg1*, *arg2*, *scale=1.0*)

Bases: `pyinduct.placeholder.EquationTerm`

Class that represents a scalar product in a weak equation.

Parameters

- `arg1` – Fieldvariable (Shapefunctions) to be projected.
- `arg2` – Testfunctions to project on.
- `scale` (*Number*) – Scaling of expression.

class ScalarTerm(*argument*, *scale=1.0*)

Bases: `pyinduct.placeholder.EquationTerm`

Class that represents a scalar term in a weak equation.

Parameters

- `argument` –
- `scale` –

class Scalars(*values*, *target_term=None*, *target_form=None*, *test_func_lbl=None*)

Bases: `pyinduct.placeholder.Placeholder`

Placeholder for scalar values that scale the equation system, gained by the projection of the pde onto the test basis.

Note: The arguments *target_term* and *target_form* are used inside the parser. For frontend use, just specify the *values*.

Parameters

- `values` – Iterable object containing the scalars for every k-th equation.
- `target_term` – Coefficient matrix to `add_to()`.
- `target_form` – Desired weight set.

class SimulationInput(*name=''*)

Bases: `object`

Base class for all objects that want to act as an input for the time-step simulation.

The calculated values for each time-step are stored in internal memory and can be accessed by `get_results()` (after the simulation is finished).

Note: Due to the underlying solver, this handle may get called with time arguments, that lie outside of the specified integration domain. This should not be a problem for a feedback controller but might cause problems for a feedforward or trajectory implementation.

---

clear_cache(*self*)

Clear the internal value storage.

When the same *SimulationInput* is used to perform various simulations, there is no possibility to distinguish between the different runs when `get_results()` gets called. Therefore this method can be used to clear the cache.

get_results (*self*, *time_steps*, *result_key='output'*, *interpolation='nearest'*, *as_eval_data=False*)
Return results from internal storage for given time steps.

Raises `Error` – If calling this method before a simulation was run.

Parameters

- **time_steps** – Time points where values are demanded.
- **result_key** – Type of values to be returned.
- **interpolation** – Interpolation method to use if demanded time-steps are not covered by the storage, see `scipy.interpolate.interp1d()` for all possibilities.
- **as_eval_data** (*bool*) – Return results as *EvalData* object for straightforward display.

Returns Corresponding function values to the given time steps.

class SimulationInputSum (*inputs*)
Bases: *pyinduct.simulation.SimulationInput*

Helper that represents a signal mixer.

class SimulationInputVector (*input_vector*)
Bases: *pyinduct.simulation.SimulationInput*

A simulation input which combines *SimulationInput* objects into a column vector.

Parameters input_vector (*array_like*) – Simulation inputs to stack.

append (*self*, *input_vector*)
Add an input to the vector.

class StackedBase (*base_info*)
Bases: *pyinduct.core.ApproximationBasis*

Implementation of a basis vector that is obtained by stacking different bases onto each other. This typically occurs when the bases of coupled systems are joined to create a unified system.

Parameters base_info (*OrderedDict*) – Dictionary with *base_label* as keys and dictionaries holding information about the bases as values. In detail, these Information must contain:

- sys_name (str): Name of the system the base is associated with.
- **order (int): Highest temporal derivative order with which the** base shall be represented in the stacked base.
- base (`ApproximationBase`): The actual basis.

function_space_hint (*self*)
Hint that returns properties that characterize the functional space of the fractions. It can be used to determine if function spaces match.

Note: Overwrite to implement custom functionality.

is_compatible_to (*self*, *other*)
Helper functions that checks compatibility between two approximation bases.

In this case compatibility is given if the two bases live in the same function space.

Parameters other (`Approximation Base`) – Approximation basis to compare with.

Returns: True if bases match, False if they do not.

scalar_product_hint (*self*)
Hint that returns steps for scalar product calculation with elements of this base.

Note: Overwrite to implement custom functionality.

abstract scale (*self, factor*)

transformation_hint (*self, info*)

If *info.src_lbl* is a member, just return it, using to correct derivative transformation, otherwise return *None*

Parameters **info** (*TransformationInfo*) – Information about the requested transformation.

Returns transformation handle

class StateSpace (*a_matrices, b_matrices, base_lbl=None, input_handle=None, c_matrix=None, d_matrix=None, obs_fb_handle=None*)

Bases: `object`

Wrapper class that represents the state space form of a dynamic system where

$$\dot{\boldsymbol{x}}(t) = \sum_{k=0}^{L} \boldsymbol{A}_k \boldsymbol{x}^{p_k}(t) + \sum_{j=0}^{V} \sum_{k=0}^{L} \boldsymbol{B}_{j,k} \frac{\mathrm{d}^j u^{p_k}}{\mathrm{d}t^j}(t) + \boldsymbol{L}\tilde{\boldsymbol{y}}(t)$$
$$\boldsymbol{y}(t) = \boldsymbol{C}\boldsymbol{x}(t) + \boldsymbol{D}u(t)$$

which has been approximated by projection on a base given by weight_label.

Parameters

- **a_matrices** (*dict*) – State transition matrices $\boldsymbol{A}_{p_k}$ for the corresponding powers of $\boldsymbol{x}$.
- **b_matrices** (*dict*) – Cascaded dictionary for the input matrices $\boldsymbol{B}_{j,k}$ in the sequence: temporal derivative order, exponent.
- **input_handle** (*SimulationInput*) – System input $u(t)$.
- **c_matrix** – $\boldsymbol{C}$
- **d_matrix** – $\boldsymbol{D}$

rhs (*self, _t, _q*)

Callback for the integration of the dynamic system, described by this object.

Parameters

- **_t** (*float*) – timestamp
- **_q** (*array*) – weight vector

Returns $\dot{\boldsymbol{x}}(t)$

Return type (array)

class TestFunction (*function_label, order=0, location=None, approx_label=None*)

Bases: *pyinduct.placeholder.SpatialPlaceholder*

Class that works as a placeholder for test functions in an equation.

Parameters

- **function_label** (*str*) – Label of the function test base.
- **order** (*int*) – Spatial derivative order.
- **location** (*Number*) – Point of evaluation / argument of the function.
- **approx_label** (*str*) – Label of the approximation test base.

class WeakFormulation (*terms*, *name*, *dominant_lbl=None*)
Bases: `object`

This class represents the weak formulation of a spatial problem. It can be initialized with several terms (see children of `EquationTerm`). The equation is interpreted as

$$term_0 + term_1 + ... + term_N = 0.$$

Parameters

- **terms** (*list*) – List of object(s) of type EquationTerm.
- **name** (*string*) – Name of this weak form.
- **dominant_lbl** (*string*) – Name of the variable that dominates this weak form.

calculate_scalar_product_matrix (*base_a*, *base_b*, *scalar_product=None*, *optimize=True*)
Calculates a matrix A , whose elements are the scalar products of each element from *base_a* and *base_b*, so that $a_{ij} = \langle \mathrm{a}_i , \mathrm{b}_j \rangle$.

Parameters

- **base_a** (`ApproximationBase`) – Basis a
- **base_b** (`ApproximationBase`) – Basis b
- **scalar_product** – (List of) function objects that are passed the members of the given bases as pairs. Defaults to the scalar product given by *base_a*
- **optimize** (*bool*) – Switch to turn on the symmetry based speed up. For development purposes only.

Returns matrix A

Return type numpy.ndarray

create_state_space (*canonical_equations*)
Create a state-space system constituted by several `CanonicalEquations` (created by `parse_weak_formulation()`)

Parameters **canonical_equations** – List of `CanonicalEquation`'s.

Raises **ValueError** – If compatibility criteria cannot be fulfilled

Returns State-space representation of the approximated system

Return type `StateSpace`

domain_intersection (*first*, *second*)
Calculate intersection(s) of two domains.

Parameters

- **first** (*set*) – (Set of) tuples defining the first domain.
- **second** (*set*) – (Set of) tuples defining the second domain.

Returns Intersection given by (start, end) tuples.

Return type set

evaluate_approximation (*base_label*, *weights*, *temp_domain*, *spat_domain*, *spat_order=0*, *name=''*)
Evaluate an approximation given by weights and functions at the points given in spatial and temporal steps.

Parameters

- **weights** – 2d np.ndarray where axis 1 is the weight index and axis 0 the temporal index.
- **base_label** (*str*) – Functions to use for back-projection.
- **temp_domain** (`Domain`) – For steps to evaluate at.

- **spat_domain** (*Domain*) – For points to evaluate at (or in).
- **spat_order** – Spatial derivative order to use.
- **name** – Name to use.

Returns *EvalData*

get_base (*label*)
Retrieve registered set of initial functions by their label.

Parameters **label** (*str*) – String, label of functions to retrieve.

Returns initial_functions

get_common_form (*placeholders*)
Extracts the common target form from a list of scalars while making sure that the given targets are equivalent.

Parameters **placeholders** – Placeholders with possibly differing target forms.

Returns Common target form.

Return type str

get_common_target (*scalars*)
Extracts the common target from list of scalars while making sure that targets are equivalent.

Parameters **scalars** (*Scalars*) –

Returns Common target.

Return type dict

get_sim_result (*weight_lbl*, *q*, *temp_domain*, *spat_domain*, *temp_order*, *spat_order*, *name=''*)
Create handles and evaluate at given points.

Parameters

- **weight_lbl** (*str*) – Label of Basis for reconstruction.
- **temp_order** – Order or temporal derivatives to evaluate additionally.
- **spat_order** – Order or spatial derivatives to evaluate additionally.
- **q** – weights
- **spat_domain** (*Domain*) – Domain object providing values for spatial evaluation.
- **temp_domain** (*Domain*) – Time steps on which rows of q are given.
- **name** (*str*) – Name of the WeakForm, used to generate the data set.

get_sim_results (*temp_domain*, *spat_domains*, *weights*, *state_space*, *names=None*, *derivative_orders=None*)
Convenience wrapper for *get_sim_result()*.

Parameters

- **temp_domain** (*Domain*) – Time domain
- **spat_domains** (*dict*) – Spatial domain from all subsystems which belongs to *state_space* as values and name of the systems as keys.
- **weights** (*numpy.array*) – Weights gained through simulation. For example with *simulate_state_space()*.
- **state_space** (*StateSpace*) – Simulated state space instance.
- **names** – List of names of the desired systems. If not given all available subssystems will be processed.
- **derivative_orders** (*dict*) – Desired derivative orders.

Returns List of `EvalData` objects.

get_transformation_info(*source_label*, *destination_label*, *source_order=0*, *destination_order=0*)

Provide the weights transformation from one/source base to another/destination base.

Parameters

- **source_label** (`str`) – Label from the source base.
- **destination_label** (`str`) – Label from the destination base.
- **source_order** – Order from the available time derivative of the source weights.
- **destination_order** – Order from the desired time derivative of the destination weights.

Returns Transformation info object.

Return type `TransformationInfo`

get_weight_transformation(*info*)

Create a handle that will transform weights from *info.src_base* into weights for *info-dst_base* while paying respect to the given derivative orders.

This is accomplished by recursively iterating through source and destination bases and evaluating their `transformation_hints`.

Parameters **info** (`TransformationInfo`) – information about the requested transformation.

Returns transformation function handle

Return type callable

integrate_function(*func*, *interval*)

Numerically integrate a function on a given interval using `complex_quadrature()`.

Parameters

- **func** (`callable`) – Function to integrate.
- **interval** (`list of tuples`) – List of (start, end) values of the intervals to integrate on.

Returns (Result of the Integration, errors that occurred during the integration).

Return type tuple

parse_weak_formulation(*weak_form*, *finalize=False*, *is_observer=False*)

Parses a `WeakFormulation` that has been derived by projecting a partial differential equation an a set of test-functions. Within this process, the separating approximation $x^n(z,t) = \sum_{i=1}^{n} c_i^n(t)\varphi_i^n(z)$ is plugged into the equation and the separated spatial terms are evaluated, leading to a ordinary equation system for the weights $c_i^n(t)$.

Parameters

- **weak_form** – Weak formulation of the pde.
- **finalize** (`bool`) – Default: False. If you have already defined the dominant labels of the weak formulations you can set this to True. See `CanonicalEquation.finalize()`

Returns The spatially approximated equation in a canonical form.

Return type `CanonicalEquation`

parse_weak_formulations(*weak_forms*)

Convenience wrapper for `parse_weak_formulation()`.

Parameters **weak_forms** – List of `WeakFormulation`'s.

Returns List of `CanonicalEquation`'s.

project_on_bases (*states*, *canonical_equations*)

Convenience wrapper for `project_on_base()`. Calculate the state, assuming it will be constituted by the dominant base of the respective system. The keys from the dictionaries *canonical_equations* and *states* must be the same.

Parameters

- **states** – Dictionary with a list of functions as values.
- **canonical_equations** – List of `CanonicalEquation` instances.

Returns Finite dimensional state as 1d-array corresponding to the concatenated dominant bases from *canonical_equations*.

Return type numpy.array

register_base (*label*, *base*, *overwrite=False*)

Register a basis to make it accessible all over the `pyinduct` framework.

Parameters

- **base** (`ApproximationBase`) – base to register
- **label** (`str`) – String that will be used as label.
- **overwrite** – Force overwrite if a basis is already registered under this label.

sanitize_input (*input_object*, *allowed_type*)

Sanitizes input data by testing if *input_object* is an array of type *allowed_type*.

Parameters

- **input_object** – Object which is to be checked.
- **allowed_type** – desired type

Returns input_object

set_dominant_labels (*canonical_equations*, *finalize=True*)

Set the dominant label (*dominant_lbl*) member of all given canonical equations and check if the problem formulation is valid (see background section: http://pyinduct.readthedocs.io/en/latest/).

If the dominant label of one or more `CanonicalEquation` is already defined, the function raise a UserWarning if the (pre)defined dominant label(s) are not valid.

Parameters

- **canonical_equations** – List of `CanonicalEquation` instances.
- **finalize** (`bool`) – Finalize the equations? Default: True.

simulate_state_space (*state_space*, *initial_state*, *temp_domain*, *settings=None*)

Wrapper to simulate a system given in state space form:

$$\dot{q} = A_p q^p + A_{p-1} q^{p-1} + \dots + A_0 q + Bu.$$

Parameters

- **state_space** (`StateSpace`) – State space formulation of the system.
- **initial_state** – Initial state vector of the system.
- **temp_domain** (`Domain`) – Temporal domain object.
- **settings** (`dict`) – Parameters to pass to the `set_integrator()` method of the `scipy.ode` class, with the integrator name included under the key `name`.

Returns Time `Domain` object and weights matrix.

Return type tuple

simulate_system(*weak_form*, *initial_states*, *temporal_domain*, *spatial_domain*, *derivative_orders=0, 0*, *settings=None*)
Convenience wrapper for `simulate_systems()`.

Parameters

- **weak_form** (`WeakFormulation`) – Weak formulation of the system to simulate.
- **initial_states** (`numpy.ndarray`) – Array of core.Functions for $x(t = 0, z), \dot{x}(t = 0, z), \ldots, x^{(n)}(t = 0, z)$.
- **temporal_domain** (`Domain`) – Domain object holding information for time evaluation.
- **spatial_domain** (`Domain`) – Domain object holding information for spatial evaluation.
- **derivative_orders** (`tuple`) – tuples of derivative orders (time, spat) that shall be evaluated additionally as values
- **settings** – Integrator settings, see `simulate_state_space()`.

simulate_systems(*weak_forms*, *initial_states*, *temporal_domain*, *spatial_domains*, *derivative_orders=None*, *settings=None*, *out=list()*)
Convenience wrapper that encapsulates the whole simulation process.

Parameters

- **weak_forms** ((list of) `WeakFormulation`) – (list of) Weak formulation(s) of the system(s) to simulate.
- **initial_states** (`dict, numpy.ndarray`) – Array of core.Functions for $x(t = 0, z), \dot{x}(t = 0, z), \ldots, x^{(n)}(t = 0, z)$.
- **temporal_domain** (`Domain`) – Domain object holding information for time evaluation.
- **spatial_domains** (`dict`) – Dict with `Domain` objects holding information for spatial evaluation.
- **derivative_orders** (`dict`) – Dict, containing tuples of derivative orders (time, spat) that shall be evaluated additionally as values
- **settings** – Integrator settings, see `simulate_state_space()`.
- **out** (`list`) – List from user namespace, where the following intermediate results will be appended:
  - canonical equations (list of types: `CanocialEquation`)
  - state space object (type: `StateSpace`)
  - initial weights (type: `numpy.array`)
  - simulation results/weights (type: `numpy.array`)

Note: The *name* attributes of the given weak forms must be unique!

Returns List of `EvalData` objects, holding the results for the FieldVariable and demanded derivatives.

Return type list

vectorize_scalar_product(*first*, *second*, *scalar_product*)
Call the given `scalar_product` in a loop for the arguments in `left` and `right`.

Given two vectors of functions

$$\boldsymbol{\varphi}(z) = (\varphi_0(z), \dots, \varphi_N(z))^T$$

and

$$\boldsymbol{\psi}(z) = (\psi_0(z), \dots, \psi_N(z))^T `,$$

this function computes $\langle \boldsymbol{\varphi}(z) | \boldsymbol{\psi}(z) \rangle_{L2}$ where

$$\langle \varphi_i(z) | \psi_j(z) \rangle_{L2} = \int\limits_{\Gamma_0}^{\Gamma_1} \bar\varphi_i(\zeta) \psi_j(\zeta) \,\mathrm{d}\zeta \,.$$

Herein, $\bar\varphi_i(\zeta)$ denotes the complex conjugate and Γ_0 as well as Γ_1 are derived by computing the intersection of the nonzero areas of the involved functions.

Parameters

- **first** (callable or `numpy.ndarray`) – (1d array of n) callable(s)
- **second** (callable or `numpy.ndarray`) – (1d array of n) callable(s)

Raises **ValueError, if the provided arrays are not equally long.** –

Returns Array of inner products

Return type numpy.ndarray

7.7 Feedback

This module contains all classes and functions related to the approximation of distributed feedback as well as their implementation for simulation purposes.

class Feedback(*feedback_law*, ***parse_kwargs*)

Bases: *pyinduct.simulation.SimulationInput*

Base class for all objects that want to act as an input for the time-step simulation.

The calculated values for each time-step are stored in internal memory and can be accessed by *get_results()* (after the simulation is finished).

Note: Due to the underlying solver, this handle may get called with time arguments, that lie outside of the specified integration domain. This should not be a problem for a feedback controller but might cause problems for a feedforward or trajectory implementation.

class ObserverFeedback(*observer_law*, *output_error*)

Bases: *pyinduct.feedback.Feedback*

Wrapper class for all observer gains that have to interact with the simulation environment.

Note: For observer gains (`observer_gain`) which are constructed from different test function bases, dont forget to specify these bases when initialization the *TestFunction* by using the keyword argument `approx_lbl`.

Parameters

- **observer_law** (*WeakFormulation*) – Variational formulation of the Observer gain. (Projected on a set of test functions.)
- **output_error** (*StateFeedback*) – Output error

class SimulationInput (*name=''*)
Bases: `object`

Base class for all objects that want to act as an input for the time-step simulation.

The calculated values for each time-step are stored in internal memory and can be accessed by `get_results()` (after the simulation is finished).

Note: Due to the underlying solver, this handle may get called with time arguments, that lie outside of the specified integration domain. This should not be a problem for a feedback controller but might cause problems for a feedforward or trajectory implementation.

---

clear_cache (*self*)
Clear the internal value storage.

When the same *SimulationInput* is used to perform various simulations, there is no possibility to distinguish between the different runs when `get_results()` gets called. Therefore this method can be used to clear the cache.

get_results (*self*, *time_steps*, *result_key='output'*, *interpolation='nearest'*, *as_eval_data=False*)
Return results from internal storage for given time steps.

Raises Error – If calling this method before a simulation was run.

Parameters

- **time_steps** – Time points where values are demanded.
- **result_key** – Type of values to be returned.
- **interpolation** – Interpolation method to use if demanded time-steps are not covered by the storage, see `scipy.interpolate.interp1d()` for all possibilities.
- **as_eval_data** (*bool*) – Return results as `EvalData` object for straightforward display.

Returns Corresponding function values to the given time steps.

class StateFeedback (*control_law*)
Bases: `pyinduct.feedback.Feedback`

Base class for all feedback controllers that have to interact with the simulation environment.

Parameters control_law (`WeakFormulation`) – Variational formulation of the control law.

calculate_scalar_product_matrix (*base_a*, *base_b*, *scalar_product=None*, *optimize=True*)
Calculates a matrix A , whose elements are the scalar products of each element from *base_a* and *base_b*, so that $a_{ij} = \langle \mathrm{a}_i , \mathrm{b}_j \rangle$.

Parameters

- **base_a** (`ApproximationBase`) – Basis a
- **base_b** (`ApproximationBase`) – Basis b
- **scalar_product** – (List of) function objects that are passed the members of the given bases as pairs. Defaults to the scalar product given by *base_a*
- **optimize** (*bool*) – Switch to turn on the symmetry based speed up. For development purposes only.

Returns matrix A

Return type numpy.ndarray

evaluate_transformations (*ce*, *weight_label*, *vect_shape*, *is_observer=False*)
Transform the different feedback/observer gains in `ce` to the basis `weight_label` and accumulate them to one gain vector.

If the feedback gain $u(t) = k^T c(t)$ was approximated with respect to the weights from the state $x(z,t) = \sum_{i=1}^{n} c_i(t)\varphi_i(z)$ the weight transformations the procedure is straight forward. However, in most of the time, during the simulation only the weights of some base $\bar{x}(z,t) = \sum_{i=1}^{m} \bar{c}_i(t)\bar{\varphi}_i(z)$ are available. Therefore, a weight transformation

$$c(t) = N^{-1}M\bar{c}(t), \qquad N_{(i,j)} = \langle \varphi_i(z), \varphi_j(z)\rangle, \qquad M_{(i,j)} = \langle \varphi_i(z), \bar{\varphi}_j(z)\rangle$$

to this basis will be computed.

The transformation of a approximated observer gain is a little bit more involved. Since, if one wants to know the transformation from the gain vector $l_i = \langle l(z), \psi_i(z)\rangle, i = 1, ..., n$ to the approximation with respect to another test base $\bar{l}_j = \langle l(z), \bar{\psi}_j(z)\rangle, j = 1, ..., m$ one has an additional degree of freedom with the ansatz $l(z) = \sum_{i=1}^{k} c_i\varphi_i(z)$.

In the most cases there is a natural choice for $\varphi_i(z),\ i = 1, ..., k$ and k, such that the the transformation to the desired projections $\bar{l}$ can be acquired with little computational effort. However, for now these more elegant techniques are not covered in this method.

Here only one method is implemented:

$$\langle l(z), \psi_j(z)\rangle = \langle \sum_{i=1}^{n} c_i\varphi_i(z), \psi_j(z)\rangle \quad \Rightarrow c = N^{-1}l, \quad N_{(i,j)} = \langle \varphi_i(z), \psi_j(z)\rangle$$
$$\langle l(z), \bar{\psi}_j(z)\rangle = \langle \sum_{i=1}^{m} \bar{c}_i\bar{\psi}_i(z), \bar{\psi}_j(z) \quad \Rightarrow \bar{l} = M\bar{c}, \quad M_{(i,j)} = \langle \bar{\psi}_i(z), \bar{\psi}_j(z)\rangle$$

Finally the transformation between the weights c and $\bar{c}$ will be computed with `get_weight_transformation`.

For more advanced approximation and transformation features, take a look at upcoming tools in the symbolic simulation branch of pyinduct (comment from 2019/06/27).

> **Warning:** Since neither `CanonicalEquation` nor `StateSpace` know the target test base $\bar{\psi}_j, j = 1, ...m$, which was used in the `WeakFormulation`, at the moment, the observer gain transformation works only if the state approximation base and the test base coincides. Which holds for example, for standard fem approximations methods and modal approximations of self adjoint operators.

Parameters

- **ce** (*CanonicalEquation*) – Feedback/observer gain.
- **weight_label** (*string*) – Label of functions the weights correspond to.
- **vect_shape** (*tuple*) – Shape of the feedback vector.
- **is_observer** (*bool*) – The argument *ce* is interpreted as feedback/observer if *observer* is False/True. Default: False

Returns Accumulated feedback/observer gain.

Return type `numpy.array`

get_base (*label*)

Retrieve registered set of initial functions by their label.

Parameters **label** (*str*) – String, label of functions to retrieve.

Returns initial_functions

get_transformation_info (*source_label, destination_label, source_order=0, destination_order=0*)

Provide the weights transformation from one/source base to another/destination base.

Parameters

- **`source_label`** (*`str`*) – Label from the source base.
- **`destination_label`** (*`str`*) – Label from the destination base.
- **`source_order`** – Order from the available time derivative of the source weights.
- **`destination_order`** – Order from the desired time derivative of the destination weights.

Returns Transformation info object.

Return type *`TransformationInfo`*

`get_weight_transformation`(*info*)

Create a handle that will transform weights from *info.src_base* into weights for *info-dst_base* while paying respect to the given derivative orders.

This is accomplished by recursively iterating through source and destination bases and evaluating their `transformation_hints`.

Parameters **`info`** (*`TransformationInfo`*) – information about the requested transformation.

Returns transformation function handle

Return type callable

`parse_weak_formulation`(*weak_form*, *finalize=False*, *is_observer=False*)

Parses a *`WeakFormulation`* that has been derived by projecting a partial differential equation an a set of test-functions. Within this process, the separating approximation $x^n(z,t) = \sum_{i=1}^{n} c_i^n(t)\varphi_i^n(z)$ is plugged into the equation and the separated spatial terms are evaluated, leading to a ordinary equation system for the weights $c_i^n(t)$.

Parameters

- **`weak_form`** – Weak formulation of the pde.
- **`finalize`** (*`bool`*) – Default: False. If you have already defined the dominant labels of the weak formulations you can set this to True. See *`CanonicalEquation.finalize()`*

Returns The spatially approximated equation in a canonical form.

Return type *`CanonicalEquation`*

7.8 Trajectory

In the module *`pyinduct.trajectory`* are some trajectory generators defined. Besides you can find here a trivial (constant) input signal generator as well as input signal generator for equilibrium to equilibrium transitions for hyperbolic and parabolic systems.

`class ConstantTrajectory`(*const=0*, *name=''*)

Bases: *`pyinduct.simulation.SimulationInput`*

Trivial trajectory generator for a constant value as simulation input signal.

Parameters **`const`** (*`numbers.Number`*) – Desired constant value of the output.

`class Domain`(*bounds=None*, *num=None*, *step=None*, *points=None*)

Bases: `object`

Helper class that manages ranges for data evaluation, containing parameters.

Parameters

- **`bounds`** (*`tuple`*) – Interval bounds.

- **num** (*int*) – Number of points in interval.
- **step** (*numbers.Number*) – Distance between points (if homogeneous).
- **points** (*array_like*) – Points themselves.

Note: If num and step are given, num will take precedence.

bounds (*self*)

ndim (*self*)

points (*self*)

step (*self*)

class InterpolationTrajectory (*t, u, **kwargs*)
Bases: *pyinduct.simulation.SimulationInput*

Provides a system input through one-dimensional linear interpolation in the given vector u .

Parameters

- **t** (*array_like*) – Vector t with time steps.
- **u** (*array_like*) – Vector u with function values, evaluated at t.
- ****kwargs** – see below

Keyword Arguments

- **show_plot** (*bool*) – to open a plot window, showing u(t).
- **scale** (*float*) – factor to scale the output.

get_plot (*self*)
Create a plot of the interpolated trajectory.

Todo: the function name does not really tell that a QtEvent loop will be executed in here

Returns the PlotWindow widget.

Return type (pg.PlotWindow)

scale (*self, scale*)

class SignalGenerator (*waveform, t, scale=1, offset=0, **kwargs*)
Bases: *pyinduct.trajectory.InterpolationTrajectory*

Signal generator that combines `scipy.signal.waveforms` and `InterpTrajectory`.

Parameters

- **waveform** (*str*) – A waveform which is provided from `scipy.signal.waveforms`.
- **t** (*array_like*) – Array with time steps or *Domain* instance.
- **scale** (*numbers.Number*) – Scale factor: output = waveform_output * scale + offset.
- **offset** (*numbers.Number*) – Offset value: output = waveform_output * scale + offset.
- **kwargs** – The corresponding keyword arguments to the desired `scipy.signal` waveform. In addition to the kwargs of the desired waveform function from scipy.signal (which will simply forwarded) the keyword arguments `frequency` (for waveforms: 'sawtooth' and 'square') and `phase_shift` (for all waveforms) provided.

class SimulationInput(*name=''*)
Bases: `object`

Base class for all objects that want to act as an input for the time-step simulation.

The calculated values for each time-step are stored in internal memory and can be accessed by `get_results()` (after the simulation is finished).

Note: Due to the underlying solver, this handle may get called with time arguments, that lie outside of the specified integration domain. This should not be a problem for a feedback controller but might cause problems for a feedforward or trajectory implementation.

clear_cache(*self*)
Clear the internal value storage.

When the same *SimulationInput* is used to perform various simulations, there is no possibility to distinguish between the different runs when `get_results()` gets called. Therefore this method can be used to clear the cache.

get_results(*self*, *time_steps*, *result_key='output'*, *interpolation='nearest'*, *as_eval_data=False*)
Return results from internal storage for given time steps.

Raises `Error` – If calling this method before a simulation was run.

Parameters

- **time_steps** – Time points where values are demanded.
- **result_key** – Type of values to be returned.
- **interpolation** – Interpolation method to use if demanded time-steps are not covered by the storage, see `scipy.interpolate.interp1d()` for all possibilities.
- **as_eval_data** (*bool*) – Return results as `EvalData` object for straightforward display.

Returns Corresponding function values to the given time steps.

class SmoothTransition(*states*, *interval*, *method*, *differential_order=0*)
A smooth transition between two given steady-states *states* on an *interval* using either:

- polynomial method
- trigonometric method

To create smooth transitions.

Parameters

- **states** (*tuple*) – States at beginning and end of interval.
- **interval** (*tuple*) – Time interval.
- **method** (*str*) – Method to use (`poly` or `tanh`).
- **differential_order** (*int*) – Grade of differential flatness γ.

coefficient_recursion(*c0*, *c1*, *param*)
Return the recursion

$$c_k(t) = \frac{\dot{c}_{k-2}(t) - a_1 c_{k-1}(t) - a_0 c_{k-2}(t)}{a_2}$$

with initial values

$$c_0 = \texttt{numpy.array}([c_0^{(0)}, ..., c_0^{(N)}])$$
$$c_1 = \texttt{numpy.array}([c_1^{(0)}, ..., c_1^{(N)}])$$

with as much computable subsequent coefficients as possible

$$
\begin{aligned}
c_2 &= \texttt{numpy.array}([c_2^{(0)}, ..., c_2^{(N-1)}]) \\
c_3 &= \texttt{numpy.array}([c_3^{(0)}, ..., c_3^{(N-1)}]) \\
&\vdots \\
c_{2N-1} &= \texttt{numpy.array}([c_{2N-1}^{(0)}]) \\
c_{2N} &= \texttt{numpy.array}([c_{2N}^{(0)}]).
\end{aligned}
$$

Only constant parameters $a_2, a_1, a_0 \in \mathbb{R}$ supported.

Parameters

- **c0** (*array_like*) – c_0
- **c1** (*array_like*) – c_1
- **param** (*array_like*) – (a_2, a_1, a_0, None, None)

Returns $C = \{0 : c_0, 1 : c_1, ..., 2N-1 : c_{2N-1}, 2N : c_{2N}\}$

Return type dict

gevrey_tanh (*T*, *n*, *sigma=1.1*, *K=2*, *length_t=None*)

Provide Gevrey function

$$
\eta(t) = \begin{cases} 0 & \forall \quad t < 0 \\ \frac{1}{2} + \frac{1}{2}\tanh\left(K\frac{2(2t-1)}{(4(t^2-t))^\sigma}\right) & \forall \quad 0 \le t \le T \\ 1 & \forall \quad t > T \end{cases}
$$

with the Gevrey-order $\rho = 1 + \frac{1}{\sigma}$ and the derivatives up to order n.

Note: For details of the recursive calculation of the derivatives see:

Rudolph, J., J. Winkler und F. Woittennek: Flatness Based Control of Distributed Parameter Systems: Examples and Computer Exercises from Various Technological Domains (Berichte aus der Steuerungs- und Regelungstechnik). Shaker Verlag GmbH, Germany, 2003.

Parameters

- **T** (*numbers.Number*) – End of the time domain=[0, T].
- **n** (*int*) – The derivatives will calculated up to order n.
- **sigma** (*numbers.Number*) – Constant σ to adjust the Gevrey order $\rho = 1 + \frac{1}{\sigma}$ of $\varphi(t)$.
- **K** (*numbers.Number*) – Constant to adjust the slope of $\varphi(t)$.
- **length_t** (*int*) – Ammount of sample points to use. Default: `int(50 * T)`

Returns

- numpy.array([[$\varphi(t)$], … , [$\varphi^{(n)}(t)$]])
- t: numpy.array([0,…,T])

Return type tuple

power_series (*z*, *t*, *C*, *spatial_der_order=0*, *temporal_der_order=0*)

Compute the function values

$$
x^{(j,i)}(z,t) = \sum_{k=0}^{N} c_{k+j}^{(i)}(t)\frac{z^k}{k!}.
$$

Parameters

- **z** (*array_like*) – Spatial steps to compute.
- **t** (*array like*) – Temporal steps to compute.
- **C** (*dict*) – Coeffient dictionary which keys correspond to the coefficient index. The values are 2D numpy.array's. For example C[1] should provide a 2d-array with the coefficient $c_1(t)$ and at least i temporal derivatives

$$\text{np.array}([c_1^{(0)}(t), ..., c_1^{(i)}(t)]).$$

- **spatial_der_order** (*int*) – Spatial derivative order j.
- **temporal_der_order** (*int*) – Temporal derivative order i.

Returns Array of shape (len(t), len(z)).

Return type numpy.array

temporal_derived_power_series (*z*, *C*, *up_to_order*, *series_termination_index*, *spatial_der_order=0*)

Compute the temporal derivatives

$$q^{(j,i)}(z = z^*, t) = \sum_{k=0}^{N} \underbrace{c_{k+j}^{(i)}}_{\text{C[k+j][i,:]}} \frac{z^{*k}}{k!}, \qquad i = 0, ..., n.$$

Parameters

- **z** (*numbers.Number*) – Evaluation point z^*.
- **C** (*dict*) – Coefficient dictionary whose keys correspond to the coefficient index. The values are 2D numpy.arrays. For example C[1] should provide a 2d-array with the coefficient $c_1(t)$ and at least n temporal derivatives

$$\text{np.array}([c_1^{(0)}(t), ..., c_1^{(i)}(t)]).$$

- **up_to_order** (*int*) – Maximum temporal derivative order n to compute.
- **series_termination_index** (*int*) – Series termination index N .
- **spatial_der_order** (*int*) – Spatial derivative order j .

Returns array holding the elements $q^{(j,0)}, \ldots, q^{(j,n)}$

Return type numpy.ndarray

7.9 Visualization

Here are some frequently used plot types with the packages `pyqtgraph` and/or `matplotlib` implemented. The respective *pyinduct.visualization* plotting function get an *EvalData* object whose definition also placed in this module. A *EvalData*-object in turn can easily generated from simulation data. The function *pyinduct.simulation.simulate_system()* for example already provide the simulation result as EvalData object.

class DataPlot (*data*)

Base class for all plotting related classes.

class Domain (*bounds=None*, *num=None*, *step=None*, *points=None*)

Bases: `object`

Helper class that manages ranges for data evaluation, containing parameters.

Parameters

- **bounds** (*tuple*) – Interval bounds.
- **num** (*int*) – Number of points in interval.
- **step** (*numbers.Number*) – Distance between points (if homogeneous).
- **points** (*array_like*) – Points themselves.

Note: If num and step are given, num will take precedence.

bounds (*self*)

ndim (*self*)

points (*self*)

step (*self*)

class EvalData (*input_data*, *output_data*, *input_labels=None*, *input_units=None*, *enable_extrapolation=False*, *fill_axes=False*, *fill_value=None*, *name=None*)

This class helps managing any kind of result data.

The data gained by evaluation of a function is stored together with the corresponding points of its evaluation. This way all data needed for plotting or other postprocessing is stored in one place. Next to the points of the evaluation the names and units of the included axes can be stored. After initialization an interpolator is set up, so that one can interpolate in the result data by using the overloaded `__call__()` method.

Parameters

- **input_data** – (List of) array(s) holding the axes of a regular grid on which the evaluation took place.
- **output_data** – The result of the evaluation.

Keyword Arguments

- **input_labels** – (List of) labels for the input axes.
- **input_units** – (List of) units for the input axes.
- **name** – Name of the generated data set.
- **fill_axes** – If the dimension of *output_data* is higher than the length of the given *input_data* list, dummy entries will be appended until the required dimension is reached.
- **enable_extrapolation** (*bool*) – If True, internal interpolators will allow extrapolation. Otherwise, the last giben value will be repeated for 1D cases and the result will be padded with zeros for cases > 1D.
- **fill_value** – If invalid data is encountered, it will be replaced with this value before interpolation is performed.

Examples

When instantiating 1d EvalData objects, the list can be omitted

```
>>> axis = Domain((0, 10), 5)
>>> data = np.random.rand(5,)
>>> e_1d = EvalData(axis, data)
```

For other cases, input_data has to be a list

```
>>> axis1 = Domain((0, 0.5), 5)
>>> axis2 = Domain((0, 1), 11)
>>> data = np.random.rand(5, 11)
>>> e_2d = EvalData([axis1, axis2], data)
```

Adding two Instances (if the boundaries fit, the data will be interpolated on the more coarse grid.) Same goes for subtraction and multiplication.

```
>>> e_1 = EvalData(Domain((0, 10), 5), np.random.rand(5,))
>>> e_2 = EvalData(Domain((0, 10), 10), 100*np.random.rand(5,))
>>> e_3 = e_1 + e_2
>>> e_3.output_data.shape
(5,)
```

Interpolate in the output data by calling the object

```
>>> e_4 = EvalData(np.array(range(5)), 2*np.array(range(5))))
>>> e_4.output_data
array([0, 2, 4, 6, 8])
>>> e_5 = e_4([2, 5])
>>> e_5.output_data
array([4, 8])
>>> e_5.output_data.size
2
```

one may also give a slice

```
>>> e_6 = e_4(slice(1, 5, 2))
>>> e_6.output_data
array([2., 6.])
>>> e_5.output_data.size
2
```

For multi-dimensional interpolation a list has to be provided

```
>>> e_7 = e_2d([[.1, .5], [.3, .4, .7)])
>>> e_7.output_data.shape
(2, 3)
```

abs (*self*)

Get the absolute value of the elements form *self.output_data* .

Returns `EvalData` with self.input_data and output_data as result of absolute value calculation.

add (*self*, *other*, *from_left=True*)

Perform the element-wise addition of the output_data arrays from *self* and *other*

This method is used to support addition by implementing __add__ (fromLeft=True) and __radd__(fromLeft=False)). If *other*** is a `EvalData`, the *input_data* lists of *self* and *other* are adjusted using `adjust_input_vectors()` The summation operation is performed on the interpolated output_data. If *other* is a `numbers.Number` it is added according to numpy's broadcasting rules.

Parameters

- **other** (`numbers.Number` or `EvalData`) – Number or EvalData object to add to self.
- **from_left** (*bool*) – Perform the addition from left if True or from right if False.

Returns `EvalData` with adapted input_data and output_data as result of the addition.

adjust_input_vectors (*self*, *other*)

Check the the inputs vectors of *self* and *other* for compatibility (equivalence) and harmonize them if they are compatible.

The compatibility check is performed for every input_vector in particular and examines whether they share the same boundaries. and equalize to the minimal discretized axis. If the amount of discretization steps between the two instances differs, the more precise discretization is interpolated down onto the less precise one.

Parameters **other** (*EvalData*) – Other EvalData class.

Returns

- (list) - New common input vectors.
- (numpy.ndarray) - Interpolated self output_data array.
- (numpy.ndarray) - Interpolated other output_data array.

Return type tuple

interpolate (*self*, *interp_axis*)

Main interpolation method for output_data.

If one of the output dimensions is to be interpolated at one single point, the dimension of the output will decrease by one.

Parameters

- **interp_axis** (*list(list)*) – axis positions in the form
- **1D** (*-*) – axis with axis=[1,2,3]
- **2D** (*-*) – [axis1, axis2] with axis1=[1,2,3] and axis2=[0,1,2,3,4]

Returns *EvalData* with *interp_axis* as new input_data and interpolated output_data.

matmul (*self*, *other*, *from_left=True*)

Perform the matrix multiplication of the output_data arrays from *self* and *other* .

This method is used to support matrix multiplication (@) by implementing __matmul__ (from_left=True) and __rmatmul__(from_left=False)). If *other*** is a *EvalData*, the *input_data* lists of *self* and *other* are adjusted using *adjust_input_vectors()*. The matrix multiplication operation is performed on the interpolated output_data. If *other* is a `numbers.Number` it is handled according to numpy's broadcasting rules.

Parameters

- **other** (*EvalData*) – Object to multiply with.
- **from_left** (*boolean*) – Matrix multiplication from left if True or from right if False.

Returns *EvalData* with adapted input_data and output_data as result of matrix multiplication.

mul (*self*, *other*, *from_left=True*)

Perform the element-wise multiplication of the output_data arrays from *self* and *other* .

This method is used to support multiplication by implementing __mul__ (from_left=True) and __rmul__(from_left=False)). If *other*** is a *EvalData*, the *input_data* lists of *self* and *other* are adjusted using *adjust_input_vectors()*. The multiplication operation is performed on the interpolated output_data. If *other* is a `numbers.Number` it is handled according to numpy's broadcasting rules.

Parameters

- **other** (`numbers.Number` or *EvalData*) – Factor to multiply with.
- **boolean** (*from_left*) – Multiplication from left if True or from right if False.

Returns *EvalData* with adapted input_data and output_data as result of multiplication.

sqrt (*self*)

Radicate the elements form *self.output_data* element-wise.

Returns *EvalData* with self.input_data and output_data as result of root calculation.

sub (*self*, *other*, *from_left=True*)
Perform the element-wise subtraction of the output_data arrays from *self* and *other* .

This method is used to support subtraction by implementing __sub__ (from_left=True) and __rsub__(from_left=False)). If *other*** is a `EvalData`, the *input_data* lists of *self* and *other* are adjusted using `adjust_input_vectors()`. The subtraction operation is performed on the interpolated output_data. If *other* is a `numbers.Number` it is handled according to numpy's broadcasting rules.

Parameters

- **other** (`numbers.Number` or `EvalData`) – Number or EvalData object to subtract.
- **from_left** (*boolean*) – Perform subtraction from left if True or from right if False.

Returns `EvalData` with adapted input_data and output_data as result of subtraction.

FORCE_MPL_ON_WINDOWS = True

class Function (*eval_handle*, *domain=- np.inf*, *np.inf*, *nonzero=- np.inf*, *np.inf*, *derivative_handles=None*)
Bases: `pyinduct.core.BaseFraction`

Most common instance of a `BaseFraction`. This class handles all tasks concerning derivation and evaluation of functions. It is used broad across the toolbox and therefore incorporates some very specific attributes. For example, to ensure the accurateness of numerical handling functions may only evaluated in areas where they provide nonzero return values. Also their domain has to be taken into account. Therefore the attributes *domain* and *nonzero* are provided.

To save implementation time, ready to go version like `LagrangeFirstOrder` are provided in the `pyinduct.simulation` module.

For the implementation of new shape functions subclass this implementation or directly provide a callable *eval_handle* and callable *derivative_handles* if spatial derivatives are required for the application.

Parameters

- **eval_handle** (*callable*) – Callable object that can be evaluated.
- **domain** (*(list of) tuples*) – Domain on which the eval_handle is defined.
- **nonzero** (*tuple*) – Region in which the eval_handle will return
- **output. Must be a subset of domain** (*nonzero*) –
- **derivative_handles** (*list*) – List of callable(s) that contain
- **of eval_handle** (*derivatives*) –

add_neutral_element (*self*)
Return the neutral element of addition for this object.

In other words: *self* + *ret_val* == *self*.

derivative_handles (*self*)

derive (*self*, *order=1*)
Spatially derive this `Function`.

This is done by neglecting *order* derivative handles and to select handle order -1 as the new evaluation_handle.

Parameters order (*int*) – the amount of derivations to perform

Raises

- **TypeError** – If *order* is not of type int.
- **ValueError** – If the requested derivative order is higher than the provided one.

Returns `Function` the derived function.

static from_data (*x*, *y*, ***kwargs*)
Create a `Function` based on discrete data by interpolating.

The interpolation is done by using `interp1d` from scipy, the *kwargs* will be passed.

Parameters

- **x** (`array-like`) – Places where the function has been evaluated .
- **y** (`array-like`) – Function values at *x*.
- ****kwargs** – all kwargs get passed to `Function` .

Returns An interpolating function.

Return type `Function`

function_handle (*self*)

function_space_hint (*self*)
Return the hint that this function is an element of the an scalar product space which is uniquely defined by the scalar product `scalar_product_hint()`.

Note: If you are working on different function spaces, you have to overwrite this hint in order to provide more properties which characterize your specific function space. For example the domain of the functions.

get_member (*self*, *idx*)
Implementation of the abstract parent method.

Since the `Function` has only one member (itself) the parameter *idx* is ignored and *self* is returned.

Parameters **idx** – ignored.

Returns self

mul_neutral_element (*self*)
Return the neutral element of multiplication for this object.

In other words: *self * ret_val == self*.

raise_to (*self*, *power*)
Raises the function to the given *power*.

Warning: Derivatives are lost after this action is performed.

Parameters **power** (`numbers.Number`) – power to raise the function to

Returns raised function

scalar_product_hint (*self*)
Return the hint that the `_dot_product_l2()` has to calculated to gain the scalar product.

scale (*self*, *factor*)
Factory method to scale a `Function`.

Parameters **factor** – `numbers.Number` or a callable.

class MplSlicePlot (*eval_data_list*, *time_point=None*, *spatial_point=None*, *ylabel=''*, *legend_label=None*, *legend_location=1*, *figure_size=10, 6*)
Bases: `pyinduct.visualization.PgDataPlot`

Get list (eval_data_list) of ut.EvalData objects and plot the temporal/spatial slice, by spatial_point/time_point, from each ut.EvalData object, in one plot. For now: only ut.EvalData objects with len(input_data) == 2 supported

class MplSurfacePlot (*data*, *keep_aspect=False*, *fig_size=12, 8*, *zlabel='$\\quad x(z,t)$'*, *title=''*)
Bases: `pyinduct.visualization.DataPlot`

Plot as 3d surface.

class PgAnimatedPlot (*data*, *title=''*, *refresh_time=40*, *replay_gain=1*, *save_pics=False*, *create_video=False*, *labels=None*)
Bases: `pyinduct.visualization.PgDataPlot`

Wrapper that shows an updating one dimensional plot of n-curves discretized in t time steps and z spatial steps. It is assumed that time propagates along axis 0 and and location along axis 1 of values. Values are therefore expected to be a array of shape (n, t, z).

Parameters

- **data** ((iterable of) `EvalData`) – results to animate
- **title** (`basestring`) – Window title.
- **refresh_time** (`int`) – Time in msec to refresh the window must be greater than zero
- **replay_gain** (`float`) – Values above 1 acc- and below 1 decelerate the playback process, must be greater than zero
- **save_pics** (`bool`) – Export snapshots for animation purposes.
- **labels** (`dict`) – Axis labels for the plot that are passed to `pyqtgraph.PlotItem` .

exported_files (*self*)

class PgDataPlot (*data*)
Bases: `pyinduct.visualization.DataPlot`, `pyqtgraph.QtCore.QObject`

Base class for all pyqtgraph plotting related classes.

class PgLinePlot3d (*data*, *n=50*, *scale=1*)
Bases: `pyinduct.visualization.PgDataPlot`

Ulots a series of n-lines of the systems state. Scaling in z-direction can be changed with the scale setting.

class PgSlicePlot (*data*, *title=None*)
Bases: `pyinduct.visualization.PgDataPlot`

Plot selected slice of given DataSets.

class PgSurfacePlot (*data*, *scales=None*, *animation_axis=None*, *title=''*)
Bases: `pyinduct.visualization.PgDataPlot`

Plot 3 dimensional data as a surface using OpenGl.

Parameters

- **data** (`EvalData`) – Data to display, if the the input-vector has length of 2, a 3d surface is plotted, if has length 3, this surface is animated. Hereby, the time axis is assumed to be the first entry of the input vector.
- **scales** (`tuple`) – Factors to scale the displayed data, each entry corresponds to an axis in the input vector with one additional scale for the *output_data*. It therefore must be of the size: *len(input_data) + 1* . If no scale is given, all axis are scaled uniformly.
- **animation_axis** (`int`) – Index of the axis to use for animation. Not implemented, yet and therefore defaults to 0 by now.
- **title** (`str`) – Window title to display.

Note: For animation this object spawns a *QTimer* which needs an running event loop. Therefore remember to store a reference to this object.

color_map = viridis

colors = ['g', 'c', 'm', 'b', 'y', 'k', 'w', 'r']

complex_wrapper (*func*)
Wraps complex valued functions into two-dimensional functions. This enables the root-finding routine to handle it as a vectorial function.

Parameters **func** (*callable*) – Callable that returns a complex result.

Returns function handle, taking x = (re(x), im(x) and returning [re(func(x), im(func(x)].

Return type two-dimensional, callable

create_animation (*input_file_mask=''*, *input_file_names=None*, *target_format='.mp4'*)
Create an animation from the given files.

If no file names are given, a file selection dialog will appear.

Parameters

- **input_file_mask** (*basestring*) – file name mask with c-style format string
- **input_file_names** (*iterable*) – names of the files

Returns animation file

create_colormap (*cnt*)
Create a colormap containing cnt values.

Parameters **cnt** (*int*) – Number of colors in the map.

Returns List of *QColor* instances.

create_dir (*dir_name*)
Create a directory with name `dir_name` relative to the current path if it doesn't already exist and return its full path.

Parameters **dir_name** (*str*) – Directory name.

Returns Full absolute path of the created directory.

Return type str

deregister_base (*label*)
Removes a set of initial functions from the packages registry.

Parameters **label** (*str*) – String, label of functions that are to be removed.

Raises **ValueError** – If label is not found in registry.

get_colors (*cnt*, *scheme='tab10'*, *samples=10*)
Create a list of colors.

Parameters

- **cnt** (*int*) – Number of colors in the list.
- **scheme** (*str*) – Mpl color scheme to use.
- **samples** (*cnt*) – Number of samples to take from the scheme before starting from the beginning.

Returns List of *np.Array* holding the rgb values.

mpl_3d_remove_margins ()
Remove thin margins in matplotlib 3d plots. The Solution is from Stackoverflow.

mpl_activate_latex()

Activate full (label, ticks, ...) latex printing in matplotlib plots.

save_2d_pg_plot(*plot*, *filename*)

Save a given pyqtgraph plot in the folder <current path>.pictures_plot under the given filename `filename`.

Parameters

- **plot** (`pyqtgraph.plotItem`) – Pyqtgraph plot.
- **filename** (*str*) – Png picture filename.

Returns Path with filename and path only.

Return type tuple of 2 str's

show(*show_pg=True*, *show_mpl=True*)

Shortcut to show all pyqtgraph and matplotlib plots / animations.

Parameters

- **show_pg** (*bool*) – Show matplotlib plots? Default: True
- **show_mpl** (*bool*) – Show pyqtgraph plots? Default: True

surface_plot(*data*, ***kwargs*)

Compatibility wrapper for PgSurfacePLot and MplSurfacePlot

Since OpenGL suffers under some problems in current windows versions, the matplotlib implementation is used there.

tear_down(*labels*, *plots=None*)

Deregister labels and delete plots.

Parameters

- **labels** (*array-like*) – All labels to deregister.
- **plots** (*array-like*) – All plots to delete.

visualize_functions(*functions*, *points=100*, *return_window=False*)

Visualizes a set of *Function* s on their domain.

Parameters

- **functions** (*iterable*) – collection of *Function* s to display.
- **points** (*int*) – Points to use for sampling the domain.
- **return_window** (*bool*) – If True the graphics window is not shown directly. In this case, a reference to the plot window is returned.

Returns: A PgPlotWindow if *delay_exec* is True.

visualize_roots(*roots*, *grid*, *func*, *cmplx=False*, *return_window=False*)

Visualize a given set of roots by examining the output of the generating function.

Parameters

- **roots** (*array like*) – Roots to display, if *None* is given, no roots will be displayed, this is useful to get a view of *func* and choosing an appropriate *grid*.
- **grid** (*list*) – List of arrays that form the grid, used for the evaluation of the given *func*.
- **func** (*callable*) – Possibly vectorial function handle that will take input of of the shape ('len(grid)',).
- **cmplx** (*bool*) – If True, the complex valued *func* is handled as a vectorial function returning [Re(func), Im(func)].
- **return_window** (*bool*) – If True the graphics window is not shown directly. In this case, a reference to the plot window is returned.

Returns: A PgPlotWindow if *delay_exec* is True.

7.10 Utils

A few helper functions for users and developers.

create_animation(*input_file_mask=''*, *input_file_names=None*, *target_format='.mp4'*)
Create an animation from the given files.

If no file names are given, a file selection dialog will appear.

Parameters

- **input_file_mask** (*basestring*) – file name mask with c-style format string
- **input_file_names** (*iterable*) – names of the files

Returns animation file

create_dir(*dir_name*)
Create a directory with name `dir_name` relative to the current path if it doesn't already exist and return its full path.

Parameters **dir_name** (*str*) – Directory name.

Returns Full absolute path of the created directory.

Return type str

7.11 Parabolic Module

7.11.1 General

class ConstantFunction(*constant*, ***kwargs*)
Bases: `pyinduct.core.Function`

A `Function` that returns a constant value.

This function can be differentiated without limits.

Parameters **constant** (*number*) – value to return

Keyword Arguments ****kwargs** – All other kwargs get passed to `Function`.

derive(*self*, *order=1*)
Spatially derive this `Function`.

This is done by neglecting *order* derivative handles and to select handle order -1 as the new evaluation_handle.

Parameters **order** (*int*) – the amount of derivations to perform

Raises

- **TypeError** – If *order* is not of type int.
- **ValueError** – If the requested derivative order is higher than the provided one.

Returns `Function` the derived function.

class FieldVariable(*function_label*, *order=0, 0*, *weight_label=None*, *location=None*, *exponent=1*, *raised_spatially=False*)
Bases: `pyinduct.placeholder.Placeholder`

Class that represents terms of the systems field variable $x(z, t)$.

Parameters

- **function_label** (*str*) – Label of shapefunctions to use for approximation, see *register_base()* for more information about how to register an approximation basis.
- **tuple of int** (*order*) – Tuple of temporal_order and spatial_order derivation order.
- **weight_label** (*str*) – Label of weights for which coefficients are to be calculated (defaults to function_label).
- **location** – Where the expression is to be evaluated.
- **exponent** – Exponent of the term.

Examples

Assuming some shapefunctions have been registered under the label `"phi"` the following expressions hold:

- $\frac{\partial^3}{\partial t \partial z^2} x(z, t)$

```
>>> x_dt_dzz = FieldVariable("phi", order=(1, 2))
```

- $\frac{\partial^2}{\partial t^2} x(3, t)$

```
>>> x_dtt_at_3 = FieldVariable("phi", order=(2, 0), location=3)
```

derive (*self*, ***, *temp_order=0*, *spat_order=0*)
Derive the expression to the specified order.

Parameters

- **temp_order** – Temporal derivative order.
- **spat_order** – Spatial derivative order.

Returns The derived expression.

Return type *Placeholder*

Note: This method uses keyword only arguments, which means that a call will fail if the arguments are passed by order.

class Function (*eval_handle*, *domain=- np.inf*, *np.inf*, *nonzero=- np.inf*, *np.inf*, *derivative_handles=None*)
Bases: *pyinduct.core.BaseFraction*

Most common instance of a *BaseFraction*. This class handles all tasks concerning derivation and evaluation of functions. It is used broad across the toolbox and therefore incorporates some very specific attributes. For example, to ensure the accurateness of numerical handling functions may only evaluated in areas where they provide nonzero return values. Also their domain has to be taken into account. Therefore the attributes *domain* and *nonzero* are provided.

To save implementation time, ready to go version like *LagrangeFirstOrder* are provided in the *pyinduct.simulation* module.

For the implementation of new shape functions subclass this implementation or directly provide a callable *eval_handle* and callable *derivative_handles* if spatial derivatives are required for the application.

Parameters

- **eval_handle** (*callable*) – Callable object that can be evaluated.
- **domain** (*(list of) tuples*) – Domain on which the eval_handle is defined.
- **nonzero** (*tuple*) – Region in which the eval_handle will return

- **output. Must be a subset of domain** (*nonzero*) –
- **derivative_handles** (*list*) – List of callable(s) that contain
- **of eval_handle** (*derivatives*) –

add_neutral_element (*self*)

Return the neutral element of addition for this object.

In other words: *self* + *ret_val* == *self*.

derivative_handles (*self*)

derive (*self*, *order=1*)

Spatially derive this `Function`.

This is done by neglecting *order* derivative handles and to select handle order $-$ 1 as the new evaluation_handle.

Parameters **order** (*int*) – the amount of derivations to perform

Raises

- **TypeError** – If *order* is not of type int.
- **ValueError** – If the requested derivative order is higher than the provided one.

Returns `Function` the derived function.

static from_data (*x*, *y*, ***kwargs*)

Create a `Function` based on discrete data by interpolating.

The interpolation is done by using `interp1d` from scipy, the *kwargs* will be passed.

Parameters

- **x** (*array-like*) – Places where the function has been evaluated .
- **y** (*array-like*) – Function values at *x*.
- ****kwargs** – all kwargs get passed to `Function` .

Returns An interpolating function.

Return type `Function`

function_handle (*self*)

function_space_hint (*self*)

Return the hint that this function is an element of the an scalar product space which is uniquely defined by the scalar product `scalar_product_hint()`.

Note: If you are working on different function spaces, you have to overwrite this hint in order to provide more properties which characterize your specific function space. For example the domain of the functions.

---

get_member (*self*, *idx*)

Implementation of the abstract parent method.

Since the `Function` has only one member (itself) the parameter *idx* is ignored and *self* is returned.

Parameters **idx** – ignored.

Returns self

mul_neutral_element (*self*)

Return the neutral element of multiplication for this object.

In other words: *self* * *ret_val* == *self*.

raise_to (*self*, *power*)
Raises the function to the given *power*.

> **Warning:** Derivatives are lost after this action is performed.

Parameters **power** (`numbers.Number`) – power to raise the function to

Returns raised function

scalar_product_hint (*self*)
Return the hint that the `_dot_product_l2()` has to calculated to gain the scalar product.

scale (*self*, *factor*)
Factory method to scale a `Function`.

Parameters **factor** – `numbers.Number` or a callable.

class Input (*function_handle*, *index=0*, *order=0*, *exponent=1*)
Bases: `pyinduct.placeholder.Placeholder`

Class that works as a placeholder for an input of the system.

Parameters

- **function_handle** (*callable*) – Handle that will be called by the simulation unit.
- **index** (*int*) – If the system's input is vectorial, specify the element to be used.
- **order** (*int*) – temporal derivative order of this term (See `Placeholder`).
- **exponent** (*numbers.Number*) – See `FieldVariable`.

Note: if *order* is nonzero, the callable is expected to return the temporal derivatives of the input signal by returning an array of `len(order)+1`.

class IntegralTerm (*integrand*, *limits*, *scale=1.0*)
Bases: `pyinduct.placeholder.EquationTerm`

Class that represents an integral term in a weak equation.

Parameters

- **integrand** –
- **limits** (*tuple*) –
- **scale** –

class Product (*a*, *b=None*)
Bases: `object`

Represents a product.

Parameters

- **a** –
- **b** –

get_arg_by_class (*self*, *cls*)
Extract element from product that is an instance of cls.

Parameters **cls** –

Returns

Return type list

class ScalarFunction(*function_label*, *order=0*, *location=None*)
Bases: `pyinduct.placeholder.SpatialPlaceholder`

Class that works as a placeholder for spatial functions in an equation. An example could be spatial dependent coefficients.

Parameters

- **function_label** (*str*) – label under which the function is registered
- **order** (*int*) – spatial derivative order to use
- **location** – location to evaluate at

Warns

- **There seems to be a problem when this function is used in combination**
- **with the :py:class:`.Product` class. Make sure to provide this class as**
- **first argument to any product you define.**

Todo: see warning.

---

static from_scalar(*scalar*, *label*, ***kwargs*)
create a `ScalarFunction` from scalar values.

Parameters

- **scalar** (*array like*) – Input that is used to generate the placeholder. If a number is given, a constant function will be created, if it is callable it will be wrapped in a `Function` and registered.
- **label** (*string*) – Label to register the created base.
- ****kwargs** – All kwargs that are not mentioned below will be passed to `Function`.

Keyword Arguments

- **order** (*int*) – See constructor.
- **location** (*int*) – See constructor.
- **overwrite** (*bool*) – See `register_base()`

Returns Placeholder object that can be used in a weak formulation.

Return type `ScalarFunction`

class ScalarTerm(*argument*, *scale=1.0*)
Bases: `pyinduct.placeholder.EquationTerm`

Class that represents a scalar term in a weak equation.

Parameters

- **argument** –
- **scale** –

class SecondOrderOperator(*a2=0*, *a1=0*, *a0=0*, *alpha1=0*, *alpha0=0*, *beta1=0*, *beta0=0*, *domain=- np.inf, np.inf*)
Interface class to collect all important parameters that describe a second order ordinary differential equation.

Parameters

- **a2** (*Number or callable*) – coefficient a_2.
- **a1** (*Number or callable*) – coefficient a_1.
- **a0** (*Number or callable*) – coefficient a_0.

- **alpha1** (*Number*) – coefficient α_1.
- **alpha0** (*Number*) – coefficient α_0.
- **beta1** (*Number*) – coefficient β_1.
- **beta0** (*Number*) – coefficient β_0.

static from_dict (*param_dict*, *domain=None*)

static from_list (*param_list*, *domain=None*)

get_adjoint_problem (*self*)

Return the parameters of the operator A^* describing the the problem

$$(\mathrm{A}^*\psi)(z) = \bar a_2 \partial_z^2 \psi(z) + \bar a_1 \partial_z \psi(z) + \bar a_0 \psi(z) \ ,$$

where the $\bar a_i$ are constant and whose boundary conditions are given by

$$\bar\alpha_1 \partial_z \psi(z_1) + \bar\alpha_0 \psi(z_1) = 0$$
$$\bar\beta_1 \partial_z \psi(z_2) + \bar\beta_0 \psi(z_2) = 0.$$

The following mapping is used:

$$\bar a_2 = a_2, \quad \bar a_1 = -a_1, \quad \bar a_0 = a_0,$$
$$\bar\alpha_1 = -1, \quad \bar\alpha_0 = \frac{a_1}{a_2} - \frac{\alpha_0}{\alpha_1},$$
$$\bar\beta_1 = -1, \quad \bar\beta_0 = \frac{a_1}{a_2} - \frac{\beta_0}{\beta_1} \ .$$

Returns Parameter set describing A^* .

Return type *SecondOrderOperator*

class TestFunction (*function_label*, *order=0*, *location=None*, *approx_label=None*)

Bases: `pyinduct.placeholder.SpatialPlaceholder`

Class that works as a placeholder for test functions in an equation.

Parameters

- **function_label** (*str*) – Label of the function test base.
- **order** (*int*) – Spatial derivative order.
- **location** (*Number*) – Point of evaluation / argument of the function.
- **approx_label** (*str*) – Label of the approximation test base.

class WeakFormulation (*terms*, *name*, *dominant_lbl=None*)

Bases: `object`

This class represents the weak formulation of a spatial problem. It can be initialized with several terms (see children of *EquationTerm*). The equation is interpreted as

$$term_0 + term_1 + ... + term_N = 0.$$

Parameters

- **terms** (*list*) – List of object(s) of type EquationTerm.
- **name** (*string*) – Name of this weak form.
- **dominant_lbl** (*string*) – Name of the variable that dominates this weak form.

compute_rad_robin_eigenfrequencies (*param*, *l*, *n_roots=10*, *show_plot=False*)

Return the first `n_roots` eigenfrequencies ω (and eigenvalues λ)

$$\omega = \sqrt{-\frac{a_1^2}{4a_2^2} + \frac{a_0 - \lambda}{a_2}}$$

to the eigenvalue problem

$$a_2\varphi''(z) + a_1\varphi'(z) + a_0\varphi(z) = \lambda\varphi(z)$$
$$\varphi'(0) = \alpha\varphi(0)$$
$$\varphi'(l) = -\beta\varphi(l).$$

Parameters

- **param** (*array_like*) – $\Big(a_2, a_1, a_0, \alpha, \beta\Big)^T$
- **l** (*numbers.Number*) – Right boundary value of the domain $[0, l] \ni z$.
- **n_roots** (*int*) – Amount of eigenfrequencies to be compute.
- **show_plot** (*bool*) – A plot window of the characteristic equation appears if it is `True`.

Returns

$$\Big(\big[\omega_1, ..., \omega_{\text{n\_roots}}\big], \big[\lambda_1, ..., \lambda_{\text{n\_roots}}\big]\Big)$$

Return type tuple –> two numpy.ndarrays of length `nroots`

eliminate_advection_term(*param*, *domain_end*)

This method performs a transformation

$$\tilde{x}(z,t) = x(z,t)e^{\int_0^z \frac{a_1(\bar{z})}{2a_2}\,d\bar{z}},$$

on the system, which eliminates the advection term $a_1 x(z,t)$ from a reaction-advection-diffusion equation of the type:

$$\dot{x}(z,t) = a_2 x''(z,t) + a_1(z)x'(z,t) + a_0(z)x(z,t).$$

The boundary can be given by robin

$$x'(0,t) = \alpha x(0,t), \quad x'(l,t) = -\beta x(l,t),$$

dirichlet

$$x(0,t) = 0, \quad x(l,t) = 0$$

or mixed boundary conditions.

Parameters

- **param** (*array_like*) – $\Big(a_2, a_1, a_0, \alpha, \beta\Big)^T$
- **domain_end** (*float*) – upper bound of the spatial domain

Raises

- **TypeError** – If $a_1(z)$ is callable but no derivative handle is
- **defined for it.** –

Returns

Parameters

$$\big(a_2, \tilde{a}_1 = 0, \tilde{a}_0(z), \tilde{\alpha}, \tilde{\beta}\big) for$$

the transformed system

$$\dot{\tilde{x}}(z,t) = a_2\tilde{x}''(z,t) + \tilde{a}_0(z)\tilde{x}(z,t)$$

and the corresponding boundary conditions (α and/or β set to None by dirichlet boundary condition).

Return type *SecondOrderOperator* or tuple

find_roots (*function*, *grid*, *n_roots=None*, *rtol=1e-05*, *atol=1e-08*, *cmplx=False*, *sort_mode='norm'*)
Searches *n_roots* roots of the *function* $f(\boldsymbol{x})$ on the given *grid* and checks them for uniqueness with aid of *rtol*.

In Detail `scipy.optimize.root()` is used to find initial candidates for roots of $f(\boldsymbol{x})$. If a root satisfies the criteria given by atol and rtol it is added. If it is already in the list, a comprehension between the already present entries' error and the current error is performed. If the newly calculated root comes with a smaller error it supersedes the present entry.

Raises `ValueError` – If the demanded amount of roots can't be found.

Parameters

- **function** (*callable*) – Function handle for math:*f(boldsymbol{x})* whose roots shall be found.
- **grid** (*list*) – Grid to use as starting point for root detection. The i th element of this list provides sample points for the i th parameter of $\boldsymbol{x}$.
- **n_roots** (*int*) – Number of roots to find. If none is given, return all roots that could be found in the given area.
- **rtol** – Tolerance to be exceeded for the difference of two roots to be unique: $f(r1) - f(r2) > \text{rtol}$.
- **atol** – Absolute tolerance to zero: $f(x^0) < \text{atol}$.
- **cmplx** (*bool*) – Set to True if the given *function* is complex valued.
- **sort_mode** (*str*) – Specify tho order in which the extracted roots shall be sorted. Default "norm" sorts entries by their l_2 norm, while "component" will sort them in increasing order by every component.

Returns numpy.ndarray of roots; sorted in the order they are returned by $f(\boldsymbol{x})$.

get_in_domain_transformation_matrix (*k1*, *k2*, *mode='n_plus_1'*)
Returns the transformation matrix M.

M is one part of a transformation

$$x = My + Ty$$

where x is the field variable of an interior point controlled parabolic system and y is the field variable of an boundary controlled parabolic system. T is a (Fredholm-) integral transformation (which can be approximated with M).

Parameters

- **k1** –
- **k2** –
- **mode** – Available modes
  - *n_plus_1*: M.shape = $(n+1, n+1), w = (w(0), ..., w(n))^T, w \in x, y$
  - *2n*: M.shape = (2n,2n), $w = (w(0), ..., w(n), ..., w(1))^T, w \in x, y$

Returns Transformation matrix M.

Return type numpy.array

get_parabolic_dirichlet_weak_form (*init_func_label*, *test_func_label*, *input_handle*, *param*, *spatial_domain*)
Return the weak formulation of a parabolic 2nd order system, using an inhomogeneous dirichlet boundary at both sides.

Parameters

- **init_func_label** (*str*) – Label of shape base to use.
- **test_func_label** (*str*) – Label of test base to use.
- **input_handle** (*SimulationInput*) – Input.
- **param** (*tuple*) – Parameters of the spatial operator.
- **spatial_domain** (*#*) – Spatial domain of the problem.
- **spatial_domain** – Spatial domain of the
- **problem.** (*#*) –

Returns Weak form of the system.

Return type *WeakFormulation*

get_parabolic_robin_weak_form(*shape_base_label*, *test_base_label*, *input_handle*, *param*, *spatial_domain*, *actuation_type_point=None*)

Provide the weak formulation for the diffusion system with advection term, reaction term, robin boundary condition and robin actuation.

$$\begin{aligned} \dot{x}(z,t) &= a_2 x''(z,t) + a_1(z)x'(z,t) + a_0(z)x(z,t), && z \in (0,l) \\ x'(0,t) &= \alpha x(0,t) \\ x'(l,t) &= -\beta x(l,t) + u(t) \end{aligned}$$

Parameters

- **shape_base_label** (*str*) – State space base label
- **test_base_label** (*str*) – Test base label
- **input_handle** (*SimulationInput*) – System input
- **param** (*array-like*) – List of parameters:
  - a_2 (numbers.Number) ~ diffusion coefficient
  - $a_1(z)$ (callable) ~ advection coefficient
  - $a_0(z)$ (callable) ~ reaction coefficient
  - α, β (numbers.Number) ~ constants for robin boundary conditions
- **spatial_domain** (*tuple*) – Limits of the spatial domain $(0, l) \ni z$
- **actuation_type_point** (*numbers.number*) – Here you can shift the point of actuation from $z = l$ to a other point in the spatial domain.

Returns

- *WeakFormulation*
- strings for the created base lables for the advection and reaction coefficient

Return type tuple

7.11.2 Control

class IntegralTerm(*integrand*, *limits*, *scale=1.0*)

Bases: *pyinduct.placeholder.EquationTerm*

Class that represents an integral term in a weak equation.

Parameters

- **integrand** –
- **limits** (*tuple*) –

- **scale** –

class ScalarTerm(*argument*, *scale=1.0*)
Bases: *pyinduct.placeholder.EquationTerm*

Class that represents a scalar term in a weak equation.

Parameters

- **argument** –
- **scale** –

class SimulationInput(*name=''*)
Bases: object

Base class for all objects that want to act as an input for the time-step simulation.

The calculated values for each time-step are stored in internal memory and can be accessed by *get_results()* (after the simulation is finished).

Note: Due to the underlying solver, this handle may get called with time arguments, that lie outside of the specified integration domain. This should not be a problem for a feedback controller but might cause problems for a feedforward or trajectory implementation.

---

clear_cache(*self*)
Clear the internal value storage.

When the same *SimulationInput* is used to perform various simulations, there is no possibility to distinguish between the different runs when *get_results()* gets called. Therefore this method can be used to clear the cache.

get_results(*self*, *time_steps*, *result_key='output'*, *interpolation='nearest'*, *as_eval_data=False*)
Return results from internal storage for given time steps.

Raises Error – If calling this method before a simulation was run.

Parameters

- **time_steps** – Time points where values are demanded.
- **result_key** – Type of values to be returned.
- **interpolation** – Interpolation method to use if demanded time-steps are not covered by the storage, see scipy.interpolate.interp1d() for all possibilities.
- **as_eval_data** (*bool*) – Return results as *EvalData* object for straightforward display.

Returns Corresponding function values to the given time steps.

class SimulationInputSum(*inputs*)
Bases: *pyinduct.simulation.SimulationInput*

Helper that represents a signal mixer.

class StateFeedback(*control_law*)
Bases: *pyinduct.feedback.Feedback*

Base class for all feedback controllers that have to interact with the simulation environment.

Parameters control_law (*WeakFormulation*) – Variational formulation of the control law.

class WeakFormulation(*terms*, *name*, *dominant_lbl=None*)
Bases: object

This class represents the weak formulation of a spatial problem. It can be initialized with several terms (see children of `EquationTerm`). The equation is interpreted as

$$term_0 + term_1 + ... + term_N = 0.$$

Parameters

- **terms** (*list*) – List of object(s) of type EquationTerm.
- **name** (*string*) – Name of this weak form.
- **dominant_lbl** (*string*) – Name of the variable that dominates this weak form.

get_parabolic_robin_backstepping_controller (*state, approx_state, d_approx_state, approx_target_state, d_approx_target_state, integral_kernel_ll, original_beta, target_beta, scale=None*)

Build a modal approximated backstepping controller $u(t) = (Kx)(t)$, for the (open loop-) diffusion system with reaction term, robin boundary condition and robin actuation

$$\begin{aligned}
\dot{x}(z,t) &= a_2 x''(z,t) + a_0 x(z,t), \qquad && z \in (0,l) \\
x'(0,t) &= \alpha x(0,t) \\
x'(l,t) &= -\beta x(l,t) + u(t)
\end{aligned}$$

such that the closed loop system has the desired dynamic of the target system

$$\begin{aligned}
\dot{\bar{x}}(z,t) &= a_2 \bar{x}''(z,t) + \bar{a}_0 \bar{x}(z,t), \qquad && z \in (0,l) \\
\bar{x}'(0,t) &= \bar{\alpha} \bar{x}(0,t) \\
\bar{x}'(l,t) &= -\bar{\beta} x(l,t)
\end{aligned}$$

where $\bar{a}_0$, $\bar{\alpha}$, $\bar{\beta}$ are controller parameters.

The control design is performed using the backstepping method, whose integral transform

$$\bar{x}(z) = x(z) + \int_0^z k(z,\bar{z}) x(\bar{z})\, d\bar{z}$$

maps from the original system to the target system.

Note: For more details see the example script `pyinduct.examples.rad_eq_const_coeff` that implements the example from [WoiEtAl17] .

Parameters

- **state** (list of `ScalarTerm`'s) – Measurement / value from simulation of $x(l)$.
- **approx_state** (list of `ScalarTerm`'s) – Modal approximated $x(l)$.
- **d_approx_state** (list of `ScalarTerm`'s) – Modal approximated $x'(l)$.
- **approx_target_state** (list of `ScalarTerm`'s) – Modal approximated $\bar{x}(l)$.
- **d_approx_target_state** (list of `ScalarTerm`'s) – Modal approximated $\bar{x}'(l)$.
- **integral_kernel_ll** (`numbers.Number`) – Integral kernel evaluated at $\bar{z} = z = l$:

$$k(l,l) = \bar{\alpha} - \alpha + \frac{a_0 - \bar{a}_0}{a_2} l\,.$$

- **original_beta** (`numbers.Number`) – Coefficient β of the original system.
- **target_beta** (`numbers.Number`) – Coefficient $\bar{\beta}$ of the target system.

- **scale** (numbers.Number) – A constant $c \in \mathbb{R}$ to scale the control law: $u(t) = c\,(Kx)(t)$.

Returns $(Kx)(t)$

Return type *StateFeedback*

scale_equation_term_list (*eqt_list*, *factor*)

Temporary function, as long *EquationTerm* can only be scaled individually.

Parameters

- **eqt_list** (*list*) – List of *EquationTerm*'s
- **factor** (*numbers.Number*) – Scale factor.

Returns Scaled copy of *EquationTerm*'s (eqt_list).

split_domain (*n*, *a_desired*, *l*, *mode='coprime'*)

Consider a domain $[0, l]$ which is divided into the two sub domains $[0, a]$ and $[a, l]$ with the discretization $l_0 = l/n$ and a partition $a + b = l$.

Calculate two numbers k_1 and k_2 with $k_1 + k_2 = n$ such that n is odd and $a = k_1 l_0$ is close to a_desired.

Parameters

- **n** (*int*) – Number of sub-intervals to create (must be odd).
- **a_desired** (*float*) – Desired partition size a .
- **l** (*float*) – Length l of the interval.
- **mode** (*str*) – Operation mode to use:
  - 'coprime': k_1 and k_2 are coprime (default) .
  - 'force_k2_as_prime_number': k_2 is a prime number (k_1 and k_2 are coprime)
  - 'one_even_one_odd': One is even and one is odd.

7.11.3 Feedforward

class InterpolationTrajectory (*t*, *u*, ***kwargs*)

Bases: *pyinduct.simulation.SimulationInput*

Provides a system input through one-dimensional linear interpolation in the given vector u .

Parameters

- **t** (*array_like*) – Vector t with time steps.
- **u** (*array_like*) – Vector u with function values, evaluated at t.
- ****kwargs** – see below

Keyword Arguments

- **show_plot** (*bool*) – to open a plot window, showing u(t).
- **scale** (*float*) – factor to scale the output.

get_plot (*self*)

Create a plot of the interpolated trajectory.

Todo: the function name does not really tell that a QtEvent loop will be executed in here

Returns the PlotWindow widget.

Return type (pg.PlotWindow)

scale (*self*, *scale*)

class RadFeedForward (*l*, *T*, *param_original*, *bound_cond_type*, *actuation_type*, *n=80*, *sigma=None*, *k=None*, *length_t=None*, *y_start=0*, *y_end=1*, ***kwargs*)

Bases: `pyinduct.trajectory.InterpolationTrajectory`

Class that implements a flatness based control approach for the reaction-advection-diffusion equation

$$\dot{x}(z,t) = a_2 x''(z,t) + a_1 x'(z,t) + a_0 x(z,t)$$

with the boundary condition

- `bound_cond_type == "dirichlet"`: $x(0,t) = 0$
  - A transition from $x'(0,0) = y0$ to $x'(0,T) = y1$ is considered.
  - With $x'(0,t) = y(t)$ where $y(t)$ is the flat output.
- `bound_cond_type == "robin"`: $x'(0,t) = \alpha\, x(0,t)$
  - A transition from $x(0,0) = y0$ to $x(0,T) = y1$ is considered.
  - With $x(0,t) = y(t)$ where $y(t)$ is the flat output.

and the actuation

- `actuation_type == "dirichlet"`: $x(l,t) = u(t)$
- `actuation_type == "robin"`: $x'(l,t) = -\beta x(l,t) + u(t)$.

The flat output trajectory $y(t)$ will be calculated with `gevrey_tanh()`.

Parameters

- **l** (*float*) – Domain length.
- **t_end** (*float*) – Transition time.
- **param_original** (*tuple*) – Tuple holding the coefficients of the pde and boundary conditions.
- **bound_cond_type** (*string*) – Boundary condition type. Can be *dirichlet* or *robin*, see above.
- **actuation_type** (*string*) – Actuation condition type. Can be *dirichlet* or *robin*, see above.
- **n** (*int*) – Derivative order to provide (defaults to 80).
- **sigma** (*number.Number*) – *sigma* value for `gevrey_tanh()`.
- **k** (*number.Number*) – *K* value for `gevrey_tanh()`.
- **length_t** (*int*) – *length_t* value for `gevrey_tanh()`.
- **y0** (*float*) – Initial value for the flat output.
- **y1** (*float*) – Desired value for the flat output after transition time.
- ****kwargs** – see below. All arguments that are not specified below are passed to `InterpolationTrajectory` .

class SecondOrderOperator (*a2=0*, *a1=0*, *a0=0*, *alpha1=0*, *alpha0=0*, *beta1=0*, *beta0=0*, *domain=- np.inf, np.inf*)

Interface class to collect all important parameters that describe a second order ordinary differential equation.

Parameters

- **a2** (*Number or callable*) – coefficient a_2.
- **a1** (*Number or callable*) – coefficient a_1.
- **a0** (*Number or callable*) – coefficient a_0.

- **alpha1** (*Number*) – coefficient α_1.
- **alpha0** (*Number*) – coefficient α_0.
- **beta1** (*Number*) – coefficient β_1.
- **beta0** (*Number*) – coefficient β_0.

static from_dict (*param_dict*, *domain=None*)

static from_list (*param_list*, *domain=None*)

get_adjoint_problem (*self*)

Return the parameters of the operator A^* describing the the problem

$$(\mathrm{A}^*\psi)(z) = \bar a_2 \partial_z^2 \psi(z) + \bar a_1 \partial_z \psi(z) + \bar a_0 \psi(z) \ ,$$

where the $\bar a_i$ are constant and whose boundary conditions are given by

$$\begin{aligned}\bar\alpha_1 \partial_z \psi(z_1) + \bar\alpha_0 \psi(z_1) &= 0 \\ \bar\beta_1 \partial_z \psi(z_2) + \bar\beta_0 \psi(z_2) &= 0.\end{aligned}$$

The following mapping is used:

$$\begin{aligned}\bar a_2 = a_2, \quad \bar a_1 = -a_1, \quad \bar a_0 = a_0, \\ \bar\alpha_1 = -1, \quad \bar\alpha_0 = \frac{a_1}{a_2} - \frac{\alpha_0}{\alpha_1}, \\ \bar\beta_1 = -1, \quad \bar\beta_0 = \frac{a_1}{a_2} - \frac{\beta_0}{\beta_1} \ .\end{aligned}$$

Returns Parameter set describing A^* .

Return type *SecondOrderOperator*

eliminate_advection_term (*param*, *domain_end*)

This method performs a transformation

$$\tilde x(z,t) = x(z,t) e^{\int_0^z \frac{a_1(\bar z)}{2 a_2}\,d\bar z} ,$$

on the system, which eliminates the advection term $a_1 x(z,t)$ from a reaction-advection-diffusion equation of the type:

$$\dot x(z,t) = a_2 x''(z,t) + a_1(z) x'(z,t) + a_0(z) x(z,t).$$

The boundary can be given by robin

$$x'(0,t) = \alpha x(0,t), \quad x'(l,t) = -\beta x(l,t),$$

dirichlet

$$x(0,t) = 0, \quad x(l,t) = 0$$

or mixed boundary conditions.

Parameters

- **param** (*array_like*) – $\Big(a_2, a_1, a_0, \alpha, \beta\Big)^T$
- **domain_end** (*float*) – upper bound of the spatial domain

Raises

- **TypeError** – If $a_1(z)$ is callable but no derivative handle is
- **defined for it.** –

Returns

Parameters

$$\left(a_2, \tilde{a}_1 = 0, \tilde{a}_0(z), \tilde{\alpha}, \tilde{\beta}\right) for$$

the transformed system

$$\dot{\tilde{x}}(z,t) = a_2\tilde{x}''(z,t) + \tilde{a}_0(z)\tilde{x}(z,t)$$

and the corresponding boundary conditions (α and/or β set to None by dirichlet boundary condition).

Return type *SecondOrderOperator* or tuple

gevrey_tanh (*T*, *n*, *sigma=1.1*, *K=2*, *length_t=None*)

Provide Gevrey function

$$\eta(t) = \begin{cases} 0 & \forall \quad t < 0 \\ \frac{1}{2} + \frac{1}{2}\tanh\left(K\frac{2(2t-1)}{(4(t^2-t))^\sigma}\right) & \forall \quad 0 \le t \le T \\ 1 & \forall \quad t > T \end{cases}$$

with the Gevrey-order $\rho = 1 + \frac{1}{\sigma}$ and the derivatives up to order n.

Note: For details of the recursive calculation of the derivatives see:

Rudolph, J., J. Winkler und F. Woittennek: Flatness Based Control of Distributed Parameter Systems: Examples and Computer Exercises from Various Technological Domains (Berichte aus der Steuerungs- und Regelungstechnik). Shaker Verlag GmbH, Germany, 2003.

Parameters

- **T** (*numbers.Number*) – End of the time domain=[0, T].
- **n** (*int*) – The derivatives will calculated up to order n.
- **sigma** (*numbers.Number*) – Constant σ to adjust the Gevrey order $\rho = 1 + \frac{1}{\sigma}$ of $\varphi(t)$.
- **K** (*numbers.Number*) – Constant to adjust the slope of $\varphi(t)$.
- **length_t** (*int*) – Ammount of sample points to use. Default: `int(50 * T)`

Returns

- numpy.array([[$\varphi(t)$], ... , [$\varphi^{(n)}(t)$]])
- t: numpy.array([0,...,T])

Return type tuple

power_series_flat_out (*z*, *t*, *n*, *param*, *y*, *bound_cond_type*)

Provides the solution $x(z,t)$ (and the spatial derivative $x'(z,t)$) of the pde

$$\dot{x}(z,t) = a_2x''(z,t) + \underbrace{a_1x'(z,t)}_{=0} + a_0x(z,t), \qquad a_1 = 0, \qquad z \in (0,l), \qquad t \in (0,T)$$

as power series approximation:

- for the boundary condition (`bound_cond_type == "dirichlet"`) $x(0,t) = 0$ and the flat output $y(t) = x'(0,t)$ with

$$x(z,t) = \sum_{n=0}^{\infty} \frac{z^{2n+1}}{a_2^n(2n+1)!} \sum_{k=0}^{n} \binom{n}{k} (-a_0)^{n-k} y^{(k)}(t)$$

- for the boundary condition (`bound_cond_type == "robin"`) $x'(0,t) = \alpha\, x(0,t)$ and the flat output $y(t) = x(0,t)$ with

$$x(z,t) = \sum_{n=0}^{\infty} \left(1 + \alpha \frac{z}{2n+1}\right) \frac{z^{2n}}{a_2^n (2n)!} \sum_{k=0}^{n} \binom{n}{k} (-a_0)^{n-k} y^{(k)}(t).$$

Parameters

- **z** (*array_like*) – $[0, ..., l]$
- **t** (*array_like*) – $[0, ..., T]$
- **n** (*int*) – Series termination index.
- **param** (*array_like*) – Parameters

$$[a_2, a_1, a_0, \alpha, \beta]$$

  – $\alpha =$ None for `bound_cond_type == dirichlet`

  – *beta* is not used from this function but has to be provided (for now)

- **y** (*array_like*) – Flat output $y(t)$ and derivatives:

$$[[y(0), ..., y(T)], ..., [y^{(n/2)}(0), ..., y^{(n/2)}(T)]].$$

- **bound_cond_type** (*str*) – `dirichlet` or `robin`

Returns Solution $x(z,t)$ of the pde and the spatial derivative $x'(z,t)$.

Return type tuple

7.11.4 Trajectory

7.12 Contributions to docs

All contributions are welcome. If you'd like to improve something, look into the sources if they contain the information you need (if not, please fix them), otherwise the documentation generation needs to be improved (look in the docs/ directory).

CHAPTER

EIGHT

CREDITS

8.1 Development Lead

- Stefan Ecklebe <stefan.ecklebe@tu-dresden.de>
- Marcus Riesmeier <marcus.riesmeier@umit.at>

8.2 Contributors

- Jens Wurm <jens.wurm@umit.at>
- Florian Alber <albflo@hotmail.de>

CHAPTER

NINE

HISTORY

CHAPTER

TEN

0.1.0 (2015-01-15)

- First Code

CHAPTER

ELEVEN

0.2.0 (2015-07-10)

- Version 0.2

CHAPTER

TWELVE

0.3.0 (2016-01-01)

- Version 0.3

CHAPTER

THIRTEEN

0.4.0 (2016-03-21)

- Version 0.4
- Change from Python2 to Python3

CHAPTER

FOURTEEN

0.5.0 (2019-09-14)

Features:

- Unification of *cure_interval* which can now be called directly as static
- Added functionality to parse pure *TestFunction* products
- Added visualization of functions with noncontinuous domain
- Support for Observer Approximation via *ObserverFeedback*
- Added complete support for *ComposedFunctionVector*
- Concept of *matching* and *intermediate* bases for easier approximation handling
- Added call to clear the base registry
- Added *StackedBase* for easier handling of compound approximation bases
- Added *ConstantFunction* class
- New Example: Simulation of Euler-Bernoulli-Beam
- New Example: Coupled PDEs within a pipe-flow simulation
- New Example: Output feedback for the String-with-Mass System
- Extended Example: Output Feedback for the Reaction-Advection-Diffusion System

Changes:

- Removed former derivative order limit of two
- Deprecated use of *exponent* in *FieldVariable*
- Made *derive* of FieldVariable keyword-only to avoid error
- Extended the test suite by a great amount of cases
- Speed improvements for dot products (a846d2d)
- Refactored the control module into the feedback module to use common calls for controller and observer design
- Improved handling and computation of transformation hints
- Made scalar product vectorization explicit and accessible
- Changed license from gpl v3 to bsd 3-clause

Bugfixes:

- Bugfix for *fill_axis* parameter of *EvalData*
- Bugfix in *find_roots* if no roots where found or asked for
- Bugfix for several errors in *visualize_roots*
- Bugfix in *_simplify_product* of *Product* where the location of scalar_function was ignored
- Bugfix for *IntegralTerm* where limits were not checked

- Bugfix for boundary values of derivatives in *Lag2ndOrder*
- Fixed Issue concerning complex state space matrices method on the class to be used for curing.
- A few fixes on the way to better plotting (739a70b)
- Fixed various deprecation warnings for scipy, numpy and sphinx
- Fixed bug in *Domain* constructor for degenerated case (1 point domain)
- Bugfix for derivatives of *Input*
- Bugfixes for *SimulationInput*
- Fixed typos in various docstrings

CHAPTER FIFTEEN

0.5.1 (2020-09-23)

Bugfixes:

- Problem with nan values in EvalData
- Activation of numpy strict mode in normal operation
- Comparison warnings in various places
- Issues with evaluation of ComposedFunctionVector
- Errors in evaluate_approximation with CompoundFunctionVectors
- Deprecation warnings in visualization code
- Broken default color scheme now uses matplotlib defaults
- Corner cases for evaluate approximation
- Made EvalData robust against NaN values in int output data array
- Index error in animation handler of SurfacePlot
- Added support for nan values in SurfacePlot
- Removed strict type check to supply different systems for simulation
- Added correct handling an NaN to spline interpolator of EvalData
- Several issues in PgSurfacePlot
- Introduced fill value for EvalData objects
- Deactivated SplineInterpolator due to bad performance
- Cleanup in SWM example tests
- Suppressed plots in examples for global test run
- Complete weak formulation test case for swm example
- Updated test command since call via setup.py got deprecated

CI related changes:

- Solved issues with screen buffer
- restructured test suite
- test now run on the installed package instead of the source tree
- updated rtd config to enable building the documentation again

CHAPTER

SIXTEEN

INDICES AND TABLES

- genindex
- modindex
- search

BIBLIOGRAPHY

[BacEtAl17] On thermal modelling of incrompressible pipe flows (Zur thermischen Modellierung inkompressibler Rohrströmungen), Simon Bachler, Johannes Huber and Frank Woittennek, at-Automatisierungstechnik, DE GRUYTER, 2017

[RW2018b] Marcus Riesmeier and Frank Woittennek; On approximation of backstepping observers for parabolic systems with robin boundary conditions; In: Proceedings of the 57th IEEE, International Conference on Decision and Control (CDC), Miami, Florida, USA, December 17-19, 2018.

[RW2018a] Marcus Riesmeier and Frank Woittennek; Modale Approximation eines verteiltparametrischen Beobachters für das Modell der Saite mit Last. GMA Fachausschuss 1.40 „Systemtheorie und Regelungstechnik“, Salzburg, Austria, September 17-20, 2018.

[WoiEtAl17] Frank Woittennek, Marcus Riesmeier and Stefan Ecklebe; On approximation and implementation of transformation based feedback laws for distributed parameter systems; IFAC World Congress, 2017, Toulouse

PYTHON MODULE INDEX

p

INDEX

A

B

C

D

E

F

G

I

L

M

N

O

P

T

V

W