numericalmodel Documentation

Release 0.1.1

Yann Büchau

May 05, 2017

Contents:

CHAPTER 1

What is `numericalmodel`?

The Python package *numericalmodel* is an attempt to **make prototyping simple numerical models** in Python an **easy and painless** task.

Why should I prototype a numerical model in Python?

During development phase of a model, it is very handy to be able to flexibly change crucial model parts. One might want to quickly:

- add more **variables/parameters/forcings**
- add another **model equation**
- have specific **forcings be time-dependent**
- test another **numerical scheme**
- use **different numerical schemes for each equation**
- **combine numerical schemes** to solve different equation parts
- etc.

Quickly achieving this in a compiled, inflexible language like **Fortran** or **C** may not be that easy. Also, debugging or unit testing such a language is way more inconvenient than it is in Python. With Python, one can take advantage of the extreme flexibility that object orientation and object introspection provides.

While it is obvious, that such a prototyped model written in an interpreted language like Python will never come up to the speed and efficiency of a compiled language, it may seem very appealing in terms of flexibility. Once it's clear how the model should look like and fundamental changes are unlikely to occur anymore, it can be translated into another, faster language.

Can I implement any numerical model with `numericalmodel`?

No, at least for now `:-)`. Currently, there are a couple of restrictions:

- **only zero-dimensional models** are supported for now. Support for more dimensions is on the TODO-list.
- *`numericalmodel`* is focused on **physical models**

But that doesn't mean that you can't create new subclasses from *`numericalmodel`* to fit your needs.

Is setting up a numerical model with `numericalmodel` really that easy?

Have a look at the *Basics*, *Setting up a model* and the *Examples* and see for yourself.

CHAPTER 2

Installation

numericalmodel is best installed via `pip`:

```
pip3 install --user numericalmodel
```

CHAPTER 3

Basics

Basic structure

The basic model structure is assumed the following:

- **The model itself (*NumericalModel*) knows:**
  - some metadata
  - the variables (*SetOfStateVariables*)
  - the parameters (*SetOfParameters*)
  - the forcing (*SetOfForcingValues*)
  - the numerical schemes (*SetOfNumericalSchemes*)
  - *[output facilities]*
- **The numerical schemes (*SetOfNumericalSchemes*) know:**
  - the individual numerical schemes (*NumericalScheme*) each for a specific equation (*Equation*)
  - how to integrate their equations (*NumericalScheme.integrate*)
- **The equations (*SetOfEquations*) know:**
  - the equation variable (*StateVariable*)
  - the equation input (*SetOfInterfaceValues*) - which may contain other variables
  - how to calculate the equation (e.g. the linear and nonlinear part of a derivative)

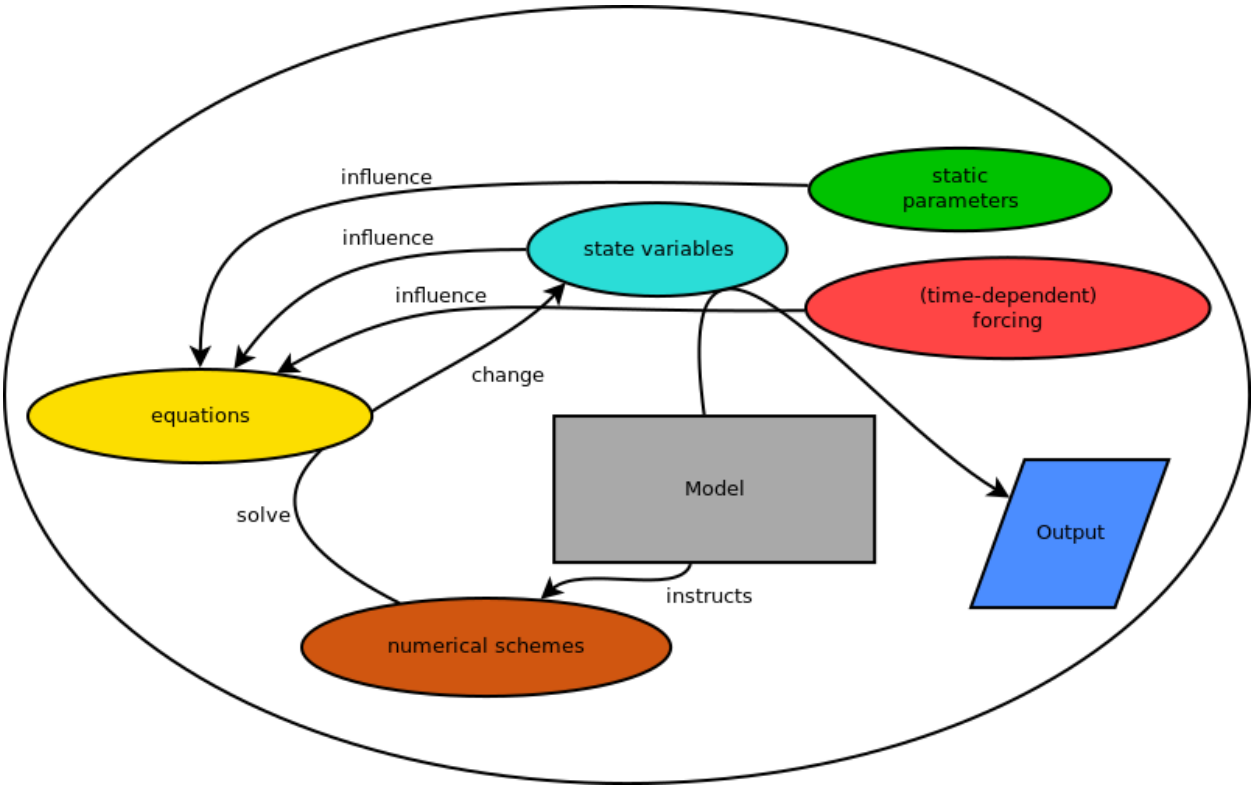

Fig. 3.1: The basic model structure

CHAPTER 4

Setting up a model

We are going to implement a simple linear decay equation:

$$\frac{dT}{dt} = -a \cdot T + F$$

Where T is the temperature in Kelvin, $a > 0$ is the linear constant and F the forcing parameter.

To set up the model, some straight-forward steps are necessary. First, import the *numericalmodel* module:

```
import numericalmodel
```

Creating a model

First initialize an object of *NumericalModel*:

```
model = numericalmodel.numericalmodel.NumericalModel()
```

We may tell the model to start from time 0:

```
model.initial_time = 0
```

Defining variables, parameters and forcing

The *StateVariable*, *Parameter*, and *ForcingValue* classes all derive from *InterfaceValue*, which is a convenient class for time/value management. It also provides interpolation (*InterfaceValue.__call__*). An *InterfaceValue* has a (sensibly unique) *id* for it to be referencable in a *SetOfInterfaceValues*.

For convenience, let's import everything from the `interfaces` submodule:

```
from numericalmodel.interfaces import *
```

Let's define our state variable. For the simple case of the linear decay equation, our only state variable is the temperature T:

```
temperature = StateVariable( id = "T", name = "temperature", unit = "K" )
```

Providing a name and a unit documents your model on the fly.

Tip: All classes in *numericalmodel* are subclasses of *ReprObject*. This makes them have a proper *__repr__* method to provide an as-exact-as-possible representation. So at any time you might do a `print(repr(temperature))` or just `temperature<ENTER>` in an interactive python session to see a representation of this object:

```
numericalmodel.interfaces.StateVariable(
        unit = 'K',
        bounds = [-inf, inf],
        name = 'temperature',
        times = array([], dtype=float64),
        time_function = numericalmodel.utils.utcnow,
        id = 'T',
        values = array([], dtype=float64),
        interpolation = 'zero'
        )
```

The others - a and F - are created similarly:

```
parameter = Parameter( id = "a", name = "linear parameter", unit = "1/s" )
forcing = ForcingValue( id = "F", name = "forcing parameter", unit = "K/s" )
```

Now we add them to the model:

```
model.variables  = SetOfStateVariables( [ temperature ] )
model.parameters = SetOfParameters(     [ parameter ]   )
model.forcing    = SetOfForcingValues(  [ forcing ]     )
```

Note: When an *InterfaceValue*'s *value* is set, a corresponding time is determined to record it. The default is to use the return value of the *InterfaceValue.time_function*, which in turn defaults to the current utc timestamp. When the model was told to use `temperature`, `parameter` and `forcing`, it automatically set the *InterfaceValue.time_function* to its internal *model_time*. That's why it makes sense to define initial values **after** adding the *InterfaceValue* s to the model.

Now that we have defined our model and added the variables, parameters and forcing, we may set initial values:

```
temperature.value = 20 + 273.15
parameter.value   = 0.1
forcing.value     = 28
```

Tip: We could also have made use of *SetOfInterfaceValues*' handy indexing features and have said:

```
model.variables["T"].value  = 20 + 273.15
model.parameters["a"].value = 0.1
model.forcing["F"].value    = 28
```

Tip: A lot of objects in *numericalmodel* also have a sensible `__str__` method, which enables them to print a summary of themselves. For example, if we do a `print(model)`:

```
###
### "numerical model"
### - a numerical model -
###  version 0.0.1
###

by:
anonymous

a numerical model
------------------------------------------------------------------
This is a numerical model.



##################
### Model data ###
##################

initial time: 0

#################
### Variables ###
#################

 "temperature"
--- T [K] ---
currently: 293.15 [K]
bounds: [-inf, inf]
interpolation: zero
1 total recorded values

##################
### Parameters ###
##################

 "linear parameter"
--- a [1/s] ---
currently: 0.1 [1/s]
bounds: [-inf, inf]
interpolation: linear
1 total recorded values

###############
### Forcing ###
###############

 "forcing parameter"
--- F [K/s] ---
currently: 28.0 [K/s]
bounds: [-inf, inf]
interpolation: linear
1 total recorded values
```

```
###############
### Schemes ###
###############

none
```

Defining equations

We proceed by defining our equation. In our case, we do this by subclassing *PrognosticEquation*, since the linear decay equation is a prognostic equation:

```
class LinearDecayEquation(numericalmodel.equations.PrognosticEquation):
    """
    Class for the linear decay equation
    """
    def linear_factor(self, time = None ):
        # take the "a" parameter from the input, interpolate it to the given
        # "time" and return the negative value
        return - self.input["a"](time)

    def independent_addend(self, time = None ):
        # take the "F" forcing parameter from the input, interpolate it to
        # the given "time" and return it
        return self.input["F"](time)

    def nonlinear_addend(self, *args, **kwargs):
        return 0 # nonlinear addend is always zero (LINEAR decay equation)
```

Now we initialize an object of this class:

```
decay_equation = LinearDecayEquation(
    variable = temperature,
    input = SetOfInterfaceValues( [parameter, forcing] ),
    )
```

We can now calculate the derivative of the equation with the *derivative* method:

```
>>> decay_equation.derivative()
-28.314999999999998
```

Choosing numerical schemes

Alright, we have all input we need and an equation. Now everything that's missing is a numerical scheme to solve the equation. *numericalmodel* ships with the most common numerical schemes. They reside in the submodule *numericalmodel.numericalschemes*. For convenience, we import everything from there:

```
from numericalmodel.numericalschemes import *
```

For a linear decay equation whose parameters are independent of time, the *EulerImplicit* scheme is a good choice:

```
implicit_scheme = numericalmodel.numericalschemes.EulerImplicit(
    equation = decay_equation
    )
```

We may now add the scheme to the model:

```
model.numericalschemes = SetOfNumericalSchemes( [ implicit_scheme ] )
```

That's it! The model is ready to run!

Running the model

Running the model is as easy as telling it a final time:

```
model.integrate( final_time = model.model_time + 60 )
```

Model results

The model results are written directly into the *StateVariable*'s cache. You may either access the values directly via the *values* property (a `numpy.ndarray`) or interpolated via the *InterfaceValue.__call__* method.

One may plot the results with `matplotlib.pyplot`:

```
import matplotlib.pyplot as plt

plt.plot( temperature.times, temperature.values,
    linewidth = 2,
    label = temperature.name,
    )
plt.xlabel( "time [seconds]" )
plt.ylabel( "{} [{}]".format( temperature.name, temperature.unit ) )
plt.legend()
plt.show()
```

The full code can be found in the *Examples* section.

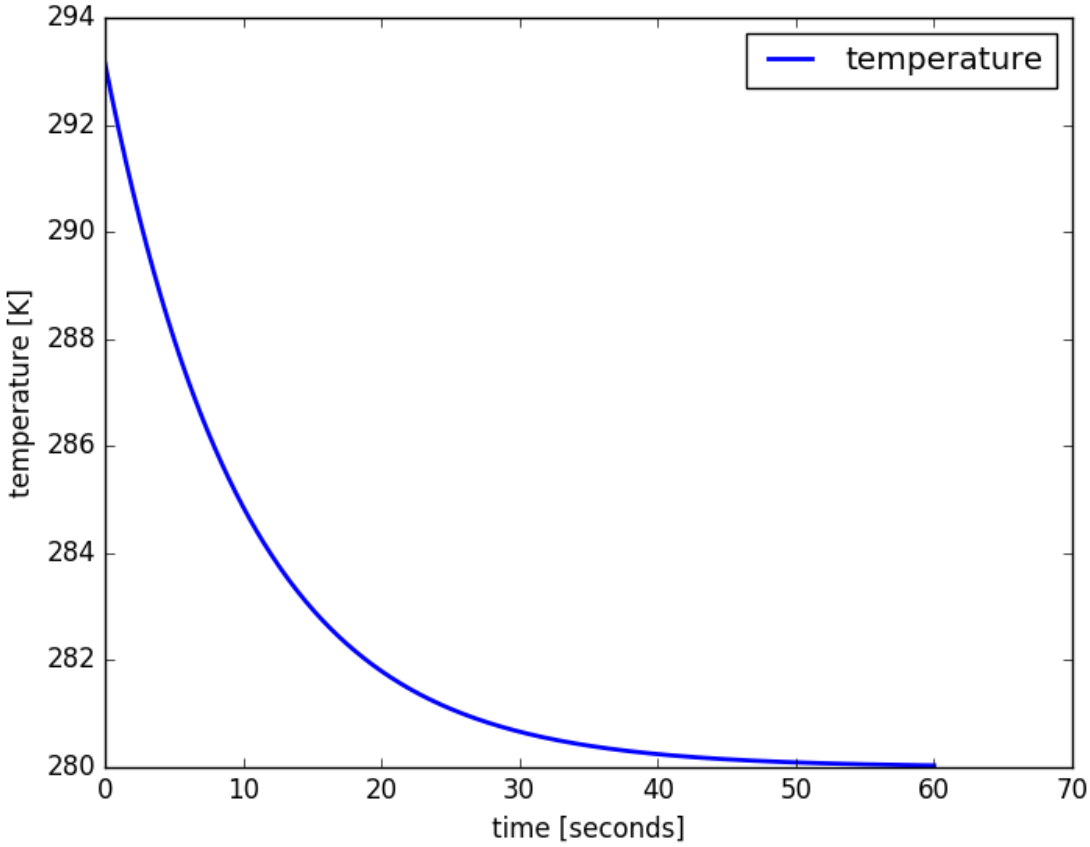

Fig. 4.1: The linear decay model results

CHAPTER 5

Examples

Linear decay equation

This is the full code from the *Setting up a model* section.

```
# import the module
import numericalmodel
from numericalmodel.interfaces import *
from numericalmodel.numericalschemes import *

# create a model
model = numericalmodel.numericalmodel.NumericalModel()
model.initial_time = 0

# define values
temperature = StateVariable( id = "T", name = "temperature", unit = "K"  )
parameter = Parameter( id = "a", name = "linear parameter", unit = "1/s"  )
forcing = ForcingValue( id = "F", name = "forcing parameter", unit = "K/s" )

# add the values to the model
model.variables  = SetOfStateVariables( [ temperature  ]  )
model.parameters = SetOfParameters(     [ parameter  ]    )
model.forcing    = SetOfForcingValues(  [ forcing  ]      )

# set initial values
model.variables["T"].value  = 20 + 273.15
model.parameters["a"].value = 0.1
model.forcing["F"].value    = 28

# define the equation
class LinearDecayEquation(numericalmodel.equations.PrognosticEquation):
    """
    Class for the linear decay equation
    """
    def linear_factor(self, time = None ):
```

```
        # take the "a" parameter from the input, interpolate it to the given
        # "time" and return the negative value
        return - self.input["a"](time)

    def independent_addend(self, time = None ):
        # take the "F" forcing parameter from the input, interpolate it to
        # the given "time" and return it
        return self.input["F"](time)

    def nonlinear_addend(self, *args, **kwargs):
        return 0 # nonlinear addend is always zero (LINEAR decay equation)

# create an equation object
decay_equation = LinearDecayEquation(
    variable = temperature,
    input = SetOfInterfaceValues( [parameter, forcing] ),
    )

# create a numerical scheme
implicit_scheme = numericalmodel.numericalschemes.EulerImplicit(
    equation = decay_equation
    )

# add the numerical scheme to the model
model.numericalschemes = SetOfNumericalSchemes( [ implicit_scheme ] )

# integrate the model
model.integrate( final_time = model.model_time + 60 )

# plot the results
import matplotlib.pyplot as plt

plt.plot( temperature.times, temperature.values,
    linewidth = 2,
    label = temperature.name,
    )
plt.xlabel( "time [seconds]" )
plt.ylabel( "{} [{}]".format( temperature.name, temperature.unit ) )
plt.legend()
plt.show()
```

Heat transfer equation

This is an implementation of the heat transfer equation to simulate heat transfer between two reservoirs:

$$c_1 m_1 \frac{dT_1}{dt} = -a \cdot (T_2 - T_1)$$

$$c_2 m_2 \frac{dT_2}{dt} = -a \cdot (T_1 - T_2)$$

```
# system module
import logging

# own modules
import numericalmodel
```

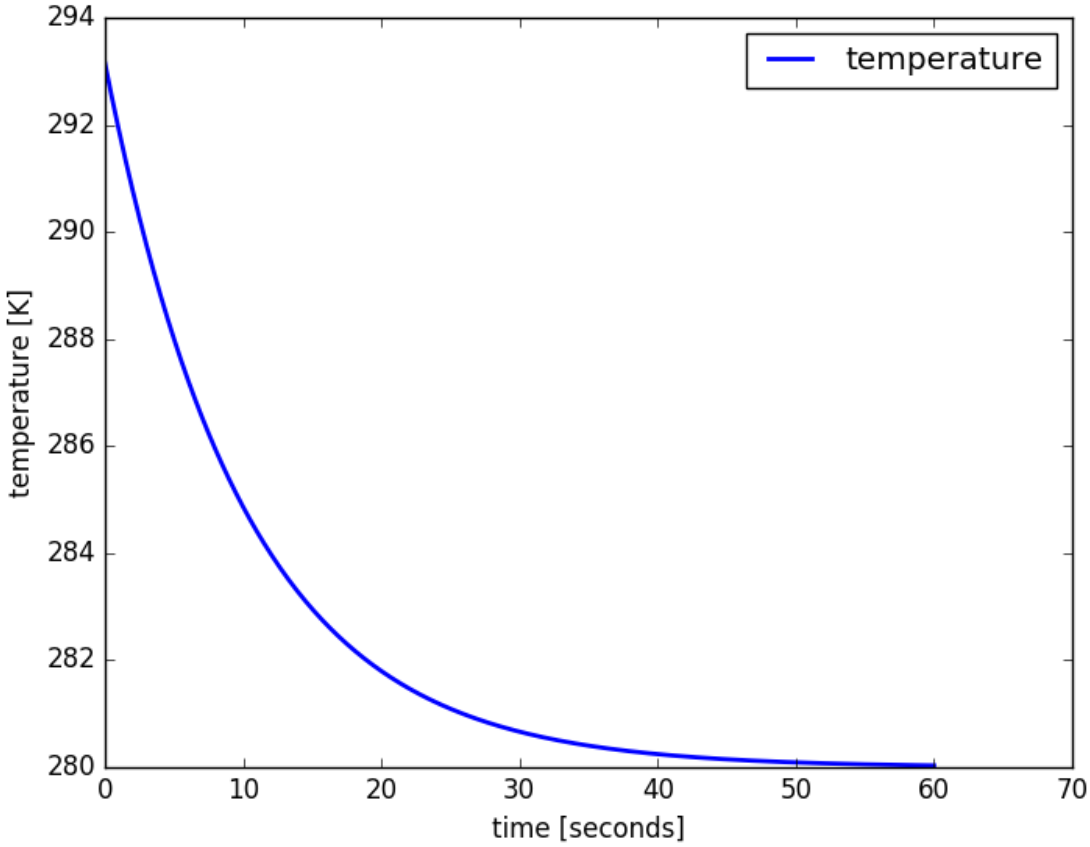

Fig. 5.1: The linear decay model results

```
from numericalmodel.numericalmodel import NumericalModel
from numericalmodel.interfaces import *
from numericalmodel.equations import *
from numericalmodel.numericalschemes import *

# external modules
import numpy as np
import matplotlib.pyplot as plt

logging.basicConfig(level = logging.INFO)

model = NumericalModel()
model.name = "simple heat transfer model"

### Variables ###
temperature_1 = StateVariable(id = "T1", name = "air temperature",   unit = "K")
temperature_2 = StateVariable(id = "T2", name = "water temperature", unit = "K")

### Parameters ###
transfer_parameter = Parameter(
    id = "a", name = "heat transfer parameter", unit = "W/K")
spec_heat_capacity_1 = Parameter(
    id = "c1", name = "specific heat capacity of dry air", unit = "J/(kg*K)")
spec_heat_capacity_2 = Parameter(
    id = "c2", name = "specific heat capacity of water", unit = "J/(kg*K)")
mass_1 = Parameter(id = "m1", name = "mass of air", unit = "kg")
mass_2 = Parameter(id = "m2", name = "mass of water", unit = "kg")

# add variables and parameters to model
model.variables = \
    SetOfStateVariables( [temperature_1, temperature_2] )
model.parameters = \
    SetOfParameters( [ transfer_parameter, spec_heat_capacity_1,
    spec_heat_capacity_2, mass_1, mass_2, ] )

### set initial values ###
model.initial_time = 0
temperature_1.value = 30 + 273.15
temperature_2.value = 10 + 273.15
mass_1.value = 20
mass_2.value = 10
spec_heat_capacity_1.value = 1005
spec_heat_capacity_2.value = 4190
transfer_parameter.value = 1.5

### define the heat transfer equation ###
class HeatTransferEquation( PrognosticEquation ):
    """
    Heat transfer equation:
        c1 * m1 * dT1/dt = a * ( T2 - T1 )
        c2 * m2 * dT2/dt = a * ( T1 - T2 )
    """
    def linear_factor( self, time = None ):
        v = lambda var: self.input[var](time)
        res = {
            "T1" : - v("a") / ( v("c1") * v("m1") ) ,
            "T2" : - v("a") / ( v("c2") * v("m2") ) ,
            }
```

```
        return res.get(self.variable.id,0)

    def nonlinear_addend( self, *args, **kwargs ):
        return 0

    def independent_addend( self, time = None ):
        v = lambda var: self.input[var](time)
        res = {
            "T1": v("a") * v("T2") / ( v("c1") * v("m1") ),
            "T2": v("a") * v("T1") / ( v("c2") * v("m2") ),
            }
        return res.get(self.variable.id,0)

# define equation input
equation_input = SetOfInterfaceValues( [
    temperature_1, temperature_2, transfer_parameter, spec_heat_capacity_1,
    spec_heat_capacity_2, mass_1, mass_2,
    ])

# set up equations
transfer_equation_1 = \
    HeatTransferEquation( variable = temperature_1, input = equation_input )
transfer_equation_2 = \
    HeatTransferEquation( variable = temperature_2, input = equation_input )

### numerical schemes ###
model.numericalschemes = SetOfNumericalSchemes( [
    EulerExplicit( equation = transfer_equation_1 ),
    EulerExplicit( equation = transfer_equation_2 ),
    ] )

# integrate the model
model.integrate( final_time = model.model_time + 3600 * 24 )

### calculate the analytical stationary solution ###
v = lambda var: equation_input[var].value
stationary_temperature = \
    ( v("c1") * v("m1") * v("T1") + v("c2") * v("m2") * v("T2") ) \
    / ( v("c1") * v("m1") + v("c2") * v("m2") )

logging.info("stationary solution: {}".format(stationary_temperature))
logging.info("    air temperature: {}".format(temperature_1.value))
logging.info("  water temperature: {}".format(temperature_2.value))

### Plot ###
fig, ax = plt.subplots()
ax.set_title(model.name)
ax.plot( ( temperature_1.times.min()/3600, temperature_1.times.max()/3600 ),
         ( stationary_temperature, stationary_temperature ),
         linewidth = 2,
         label = "analytical stationary solution",
         )
ax.plot( temperature_1.times/3600, temperature_1.values,
    linewidth = 2,
    label = temperature_1.name,
    )
ax.plot( temperature_2.times/3600, temperature_2.values,
    linewidth = 2,
```

```
    label = temperature_2.name,
    )
ax.set_xlabel( "time [hours]" )
ax.set_ylabel( "temperature [{}]".format( temperature_1.unit ) )
ax.legend()
plt.show()
```

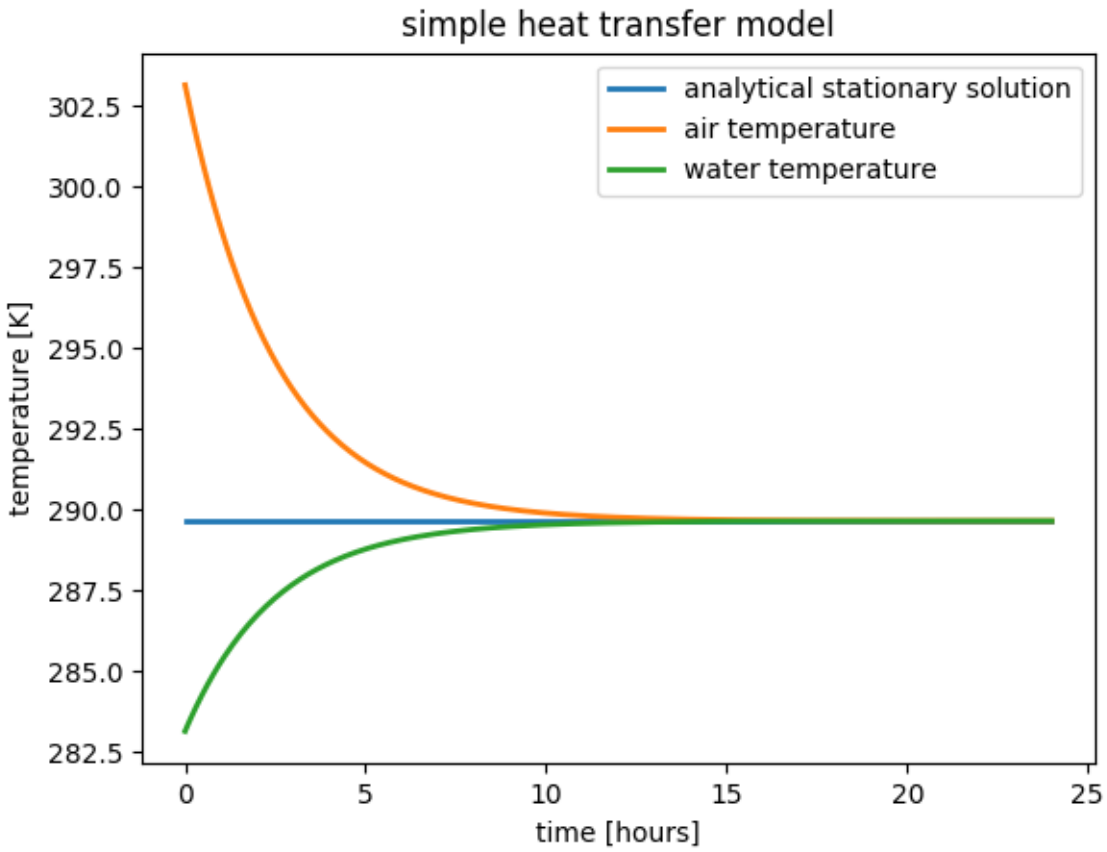

Fig. 5.2: heat transfer model results

CHAPTER 6

numericalmodel

numericalmodel package

numericalmodel Python module

Subpackages

numericalmodel.gui package

Graphical user interface for a NumericalModel. This module is only useful if the system package `python3-gi` is installed to provide the `gi` module.

To install, use your system's package manager and install `python3-gi`.

On Debian/Ubuntu:

```
sudo apt-get install python3-gi
```

Note: If you don't have system privileges, there is also the (experimental) `pgi` module on PyPi that you can install via:

```
pip3 install --user pgi
```

Theoretically, the *NumericalModelGui* might work with this package as well.

class numericalmodel.gui.**NumericalModelGui**(*numericalmodel*)

Bases: *numericalmodel.utils.LoggerObject*

class for a GTK gui to run a *NumericalModel* interactively

Parameters **numericalmodel** (NumericalModel) – the NumericalModel to run

__getitem__(*key*)
When indexed, return the corresponding Glade gui element

Parameters **key** (*str*) – the Glade gui element name

quit(**args*)
Stop the gui

run()
Run the gui

setup_gui()
Set up the GTK gui elements

setup_signals(*signals*, *handler*)
This is a workaround to signal.signal(signal, handler) which does not work with a `GLib.MainLoop` for some reason. Thanks to: http://stackoverflow.com/a/26457317/5433146

Parameters

- **signals** (*list*) – the signals (see `signal` module) to connect to
- **handler** (*callable*) – function to be executed on these signals

builder
The gui's `GtkBuilder`. This is a read-only property.

Getter Return the `GtkBuilder`, load the *gladefile* if necessary.

Type `GtkBuilder`

gladefile
The gui's Glade file. This is a read-only property.

Type `str`

Submodules

numericalmodel.equations module

class numericalmodel.equations.**DerivativeEquation**(*variable=None*, *description=None*, *long_description=None*, *input=None*)
Bases: *numericalmodel.equations.Equation*

Class to represent a derivative equation

derivative(*time=None*, *variablevalue=None*)
Calculate the derivative (right-hand-side) of the equation

Parameters

- **time** (*single numeric, optional*) – the time to calculate the derivative. Defaults to the variable's current (last) time.
- **variablevalue** (*numpy.ndarray, optional*) – the variable vaulue to use. Defaults to the value of self.variable at the given time.

Returns the derivatives corresponding to the given time

Return type numpy.ndarray

independent_addend(*time=None*)
Calculate the derivative's addend part that is independent of the variable.

Parameters **time** (*single numeric, optional*) – the time to calculate the derivative. Defaults to the variable's current (last) time.

Returns the equation's variable-independent addend at the corresponding time

Return type numpy.ndarray

linear_factor(*time=None*)
Calculate the derivative's linear factor in front of the variable

Parameters **time** (*single numeric, optional*) – the time to calculate the derivative. Defaults to the variable's current (last) time.

Returns the equation's linear factor at the corresponding time

Return type numpy.ndarray

nonlinear_addend(*time=None*, *variablevalue=None*)
Calculate the derivative's addend part that is nonlinearly dependent of the variable.

Parameters

- **time** (*single numeric, optional*) – the time to calculate the derivative. Defaults to the variable's current (last) time.
- **variablevalue** (*numpy.ndarray, optional*) – the variable vaulue to use. Defaults to the value of self.variable at the given time.

Returns the equation's nonlinear addend at the corresponding time

Return type numpy.ndarray

class numericalmodel.equations.**DiagnosticEquation**(*variable=None*, *description=None*, *long_description=None*, *input=None*)
Bases: *numericalmodel.equations.Equation*

Class to represent diagnostic equations

class numericalmodel.equations.**Equation**(*variable=None*, *description=None*, *long_description=None*, *input=None*)
Bases: *numericalmodel.utils.LoggerObject*, *numericalmodel.utils.ReprObject*

Base class for equations

Parameters

- **description** (*str, optional*) – short equation description
- **long_description** (*str, optional*) – long equation description
- **variable** (StateVariable, *optional*) – the variable obtained by solving the equation
- **input** (SetOfInterfaceValues, *optional*) – set of values needed by the equation

__str__()
Stringification

Returns a summary

Return type str

depends_on(*id*)
Check if this equation depends on a given *InterfaceValue*

Parameters **id** (*str or InterfaceValue*) – an *InterfaceValue* or an id

Returns `True` if 'id' is in *input*, `False` otherwise

Return type bool

description
The description of the equation

Type `str`

input
The input needed by the equation. Only real dependencies should be included. If the equation depends on the *variable*, it should also be included in *input*.

Type *SetOfInterfaceValues*

long_description
The longer description of this equation

Type `str`

variable
The variable the equation is able to solve for

Type *StateVariable*

class numericalmodel.equations.**PrognosticEquation**(*variable=None*, *description=None*, *long_description=None*, *input=None*)
Bases: *numericalmodel.equations.DerivativeEquation*

Class to represent prognostic equations

class numericalmodel.equations.**SetOfEquations**(*elements=[]*)
Bases: *numericalmodel.utils.SetOfObjects*

Base class for sets of Equations

Parameters **elements** (`list` of *Equation*, optional) – the list of *Equation* instances

_object_to_key(*obj*)
key transformation function.

Parameters **obj** (*object*) – the element

Returns the unique key for this object. The *Equation.variable*'s *id* is used.

Return type str

numericalmodel.genericmodel module

class numericalmodel.genericmodel.**GenericModel**(*name=None*, *version=None*, *description=None*, *long_description=None*, *authors=None*)
Bases: *numericalmodel.utils.LoggerObject*, *numericalmodel.utils.ReprObject*

Base class for models

Parameters

- **name** (*str, optional*) – the model name
- **version** (*str, optional*) – the model version
- **description** (*str*) – a short model description
- **long_description** (*str*) – an extended model description

- **authors** (str, list or dict, optional) – the model author(s). One of

  str: name of single author

  list of str: list of author names

  dict: dict of {'task': ['name1','name1']} pairs

__str__()
Stringification

Returns a summary

Return type str

authors
The model author(s)

str: name of single author

list of str: list of author names

dict: dict of {'task': ['name1','name1']} pairs

description
The model description

Type str

long_description
Longer model description

Type str

name
The model name

Type str

version
The model version

Type str

numericalmodel.interfaces module

class numericalmodel.interfaces.**ForcingValue**(*name=None*, *id=None*, *unit=None*, *time_function=None*, *interpolation=None*, *values=None*, *times=None*, *bounds=None*, *remembrance=None*)
Bases: *numericalmodel.interfaces.InterfaceValue*

Class for forcing values

class numericalmodel.interfaces.**InterfaceValue**(*name=None*, *id=None*, *unit=None*, *time_function=None*, *interpolation=None*, *values=None*, *times=None*, *bounds=None*, *remembrance=None*)
Bases: *numericalmodel.utils.LoggerObject*, *numericalmodel.utils.ReprObject*

Base class for model interface values

Parameters

- **name** (*str, optional*) – value name

- **id** (*str, optional*) – unique id
- **values** (1d numpy.ndarray, optional) – all values this InterfaceValue had in chronological order
- **times** (1d numpy.ndarray, optional) – the corresponding times to values
- **unit** (*str,optional*) – physical unit of value
- **bounds** (*list, optional*) – lower and upper value bounds
- **interpolation** (*str, optional*) – interpolation kind. See scipy.interpolate.interp1d for documentation. Defaults to "zero".
- **time_function** (*callable, optional*) – function that returns the model time as utc unix timestamp
- **remembrance** (*float, optional*) – maximum *time* difference to keep past *values*

__call__(*times=None*)
When called, return the value, optionally at a specific time

Parameters **times** (*numeric, optional*) – The times to obtain data from

__str__()
Stringification

Returns a summary

Return type str

forget_old_values()
Drop *values* and *times* older than *remembrance*.

Returns True is data was dropped, False otherwise

Return type bool

bounds
The *values*' bounds. Defaults to an infinite interval.

Getter Return the current bounds

Setter If the bounds change, check if all *values* lie within the new bounds.

Type list, [lower, upper]

id
The unique id

Type str

interpolation
The interpolation kind to use in the *__call__* method. See scipy.interpolate.interp1d for documentation.

Getter Return the interplation kind.

Setter Set the interpolation kind. Reset the internal interpolator if the interpolation kind changed.

Type str

interpolator
The interpolator for interpolation of *values* over *times*. Creating this interpolator is costly and thus

only performed on demand, i.e. when *__call__* is called **and** no interpolator was created previously or the previously created interolator was unset before (e.g. by setting a new *value* or changing *interpolation*)

Type scipy.interpolate.interp1d

name
The name.

Type str

next_time
The next time to use when *value* is set.

Getter Return the next time to use. Defaults to the value of *time_function* if no *next_time* was set.

Setter Set the next time to use. Set to None to unset and use the default time in the getter again.

Type float

remembrance
How long should this *InterfaceValue* store it's *values*? This is the greatest difference the current *time* may have to the smallest *time*. Values earlier than the *remembrance* time are discarded. Set to None for no limit.

Type float or None

time
The current time

Getter Return the current time, i.e. the last time recorded in *times*.

Type float

time_function

times
All times the *value* has ever been set in chronological order

Type numpy.ndarray

unit
The SI-unit.

Type str, SI-unit

value
The current value.

Getter the return value of *__call__*, i.e. the current value.

Setter When this property is set, the given value is recorded to the time given by *next_time*. If this time exists already in *times*, the corresponding value in *values* is overwritten. Otherwise, the new time and value are appended to *times* and *values*. The value is also checked to lie within the *bounds*.

Type numeric

values
All values this InterfaceValue has ever had in chronological order

Getter Return the current values

Setter Check if all new values lie within the *bounds*

Type numpy.ndarray

class numericalmodel.interfaces.**Parameter**(*name=None*, *id=None*, *unit=None*, *time_function=None*, *interpolation=None*, *values=None*, *times=None*, *bounds=None*, *remembrance=None*)

Bases: *numericalmodel.interfaces.InterfaceValue*

Class for parameters

class numericalmodel.interfaces.**SetOfForcingValues**(*elements=[]*)

Bases: *numericalmodel.interfaces.SetOfInterfaceValues*

Class for a set of forcing values

class numericalmodel.interfaces.**SetOfInterfaceValues**(*elements=[]*)

Bases: *numericalmodel.utils.SetOfObjects*

Base class for sets of interface values

Parameters **values** (list of *InterfaceValue*, optional) – the list of values

__call__(*id*)

Get the value of an *InterfaceValue* in this set

Parameters **id** (*str*) – the id of an *InterfaceValue* in this set

Returns the *value* of the corresponding *InterfaceValue*

Return type float

_object_to_key(*obj*)

key transformation function.

Parameters **obj** (*object*) – the element

Returns the unique key for this object. The *InterfaceValue.id* is used.

Return type key (str)

time_function

The time function of all the *InterfaceValue* s in the set.

Getter Return a list of time functions from the elements

Setter Set the time function of each element

Type (list of) callables

class numericalmodel.interfaces.**SetOfParameters**(*elements=[]*)

Bases: *numericalmodel.interfaces.SetOfInterfaceValues*

Class for a set of parameters

class numericalmodel.interfaces.**SetOfStateVariables**(*elements=[]*)

Bases: *numericalmodel.interfaces.SetOfInterfaceValues*

Class for a set of state variables

class numericalmodel.interfaces.**StateVariable**(*name=None*, *id=None*, *unit=None*, *time_function=None*, *interpolation=None*, *values=None*, *times=None*, *bounds=None*, *remembrance=None*)

Bases: *numericalmodel.interfaces.InterfaceValue*

Class for state variables

numericalmodel.numericalmodel module

class numericalmodel.numericalmodel.**NumericalModel**(*name=None*, *version=None*, *description=None*, *long_description=None*, *authors=None*, *initial_time=None*, *parameters=None*, *forcing=None*, *variables=None*, *numericalschemes=None*)

Bases: *numericalmodel.genericmodel.GenericModel*

Class for numerical models

Parameters

- **name** (*str, optional*) – the model name
- **version** (*str, optional*) – the model version
- **description** (*str*) – a short model description
- **long_description** (*str*) – an extended model description
- **authors** (str, list or dict, optional) – the model author(s). One of

  str: name of single author

  list of str: list of author names

  dict: dict of {'task': ['name1','name1']} pairs
- **initial_time** (*float*) – initial model time (UTC unix timestamp)
- **parameters** (SetOfParameters, *optional*) – model parameters
- **forcing** (SetOfForcingValues, *optional*) – model forcing
- **variables** (SetOfStateVariables, *optional*) – model state variables
- **numericalschemes** (SetOfNumericalSchemes, *optional*) – model schemes with equation

__str__()

Stringification

Returns a summary

Return type str

get_model_time()

The current model time

Returns current model time

Return type float

integrate(*final_time*)

Integrate the model until final_time

Parameters **final_time** (*float*) – time to integrate until

run_interactively()

Open a GTK window to interactively run the model:

•change the model variables, parameters and forcing on the fly

•directly see the model output

•control simulation speed

•pause and continue or do stepwise simulation

Note: This feature is still in development and currently not really functional.

forcing
The model forcing

Type *SetOfForcingValues*

initial_time
The initial model time

Type float

model_time
The current model time

Type float

numericalschemes
The model numerical schemes

Type str

parameters
The model parameters

Type *SetOfParameters*

variables
The model variables

Type *SetOfStateVariables*

numericalmodel.numericalschemes module

class numericalmodel.numericalschemes.**EulerExplicit**(*description=None*, *long_description=None*, *equation=None*, *fallback_max_timestep=None*, *ignore_linear=None*, *ignore_independent=None*, *ignore_nonlinear=None*)

Bases: *numericalmodel.numericalschemes.NumericalScheme*

Euler-explicit numerical scheme

step(*time=None*, *timestep=None*, *tendency=True*)

class numericalmodel.numericalschemes.**EulerImplicit**(*description=None*, *long_description=None*, *equation=None*, *fallback_max_timestep=None*, *ignore_linear=None*, *ignore_independent=None*, *ignore_nonlinear=None*)

Bases: *numericalmodel.numericalschemes.NumericalScheme*

Euler-implicit numerical scheme

step(*time=None*, *timestep=None*, *tendency=True*)
Integrate one "timestep" from "time" forward with the Euler-implicit scheme and return the resulting variable value.

Parameters

- **time** (*single numeric*) – The time to calculate the step FROM
- **timestep** (*single numeric*) – The timestep to calculate the step
- **tendency** (*bool, optional*) – return the tendency or the actual value of the variable after the timestep?

Returns The resulting variable value or tendency

Return type numpy.ndarray

Raises AssertionError – when the equation's nonlinear part is not zero and *ignore_nonlinear* is not set to True

class numericalmodel.numericalschemes.**LeapFrog**(*description=None*, *long_description=None*, *equation=None*, *fallback_max_timestep=None*, *ignore_linear=None*, *ignore_independent=None*, *ignore_nonlinear=None*)
Bases: *numericalmodel.numericalschemes.NumericalScheme*

Leap-Frog numerical scheme

step(*time=None*, *timestep=None*, *tendency=True*)

class numericalmodel.numericalschemes.**NumericalScheme**(*description=None*, *long_description=None*, *equation=None*, *fallback_max_timestep=None*, *ignore_linear=None*, *ignore_independent=None*, *ignore_nonlinear=None*)
Bases: *numericalmodel.utils.ReprObject*, *numericalmodel.utils.LoggerObject*

Base class for numerical schemes

Parameters

- **description** (*str*) – short equation description
- **long_description** (*str*) – long equation description
- **equation** (DerivativeEquation) – the equation
- **fallback_max_timestep** (*single numeric*) – the fallback maximum timestep if no timestep can be estimated from the equation
- **ignore_linear** (*bool*) – ignore the linear part of the equation?
- **ignore_independent** (*bool*) – ignore the variable-independent part of the equation?
- **ignore_nonlinear** (*bool*) – ignore the nonlinear part of the equation?

__str__()
Stringification

Returns a summary

Return type str

_needed_timesteps_for_integration_step(*timestep=None*)
Given a timestep to integrate from now on, what other timesteps of the dependencies are needed?

Parameters **timestep** (*single numeric*) – the timestep to calculate

Returns the timesteps

Return type numpy.ndarray

Note: timestep 0 means the current time

independent_addend(*time=None*)
Calculate the equation's addend part that is independent of the variable.

Parameters **time** (*single numeric, optional*) – the time to calculate the derivative. Defaults to the variable's current (last) time.

Returns the independent addend or 0 if *ignore_independent* is `True`.

Return type numeric

integrate(*time=None*, *until=None*)
Integrate until a certain time, respecting the *max_timestep*.

Parameters

- **time** (*single numeric, optional*) – The time to begin. Defaults to current variable *time*.
- **until** (*single numeric, optional*) – The time to integrate until. Defaults to one *max_timestep* further.

integrate_step(*time=None*, *timestep=None*)
Integrate "timestep" forward and set results in-place

Parameters

- **time** (*single numeric, optional*) – The time to calculate the step FROM. Defaults to the current variable time.
- **timestep** (*single numeric, optional*) – The timestep to calculate the step. Defaults to *max_timestep*.

linear_factor(*time=None*)
Calculate the equation's linear factor in front of the variable.

Parameters **time** (*single numeric, optional*) – the time to calculate the derivative. Defaults to the variable's current (last) time.

Returns the linear factor or 0 if *ignore_linear* is `True`.

Return type numeric

max_timestep_estimate(*time=None*, *variablevalue=None*)
Based on this numerical scheme and the equation parts, estimate a maximum timestep. Subclasses may override this.

Parameters

- **time** (*single numeric, optional*) – the time to calculate the derivative. Defaults to the variable's current (last) time.
- **variablevalue** (*numpy.ndarray, optional*) – the variable vaulue to use. Defaults to the value of self.variable at the given time.

Returns an estimate of the current maximum timestep. Definitely check the result for integrity.

Return type single numeric or bogus

Raises `Exception` – any exception if something goes wrong

needed_timesteps(*timestep*)
Given a timestep to integrate from now on, what other timesteps of the dependencies are needed?

Parameters **timestep** (*single numeric*) – the timestep to calculate

Returns the timesteps

Return type numpy.ndarray

Note: timestep 0 means the current time

---

nonlinear_addend(*time=None*, *variablevalue=None*)
Calculate the derivative's addend part that is nonlinearly dependent of the variable.

Parameters

- **time** (*single numeric, optional*) – the time to calculate the derivative. Defaults to the variable's current (last) time.
- **variablevalue** (*numpy.ndarray, optional*) – the variable vaulue to use. Defaults to the value of self.variable at the given time.

Returns the nonlinear addend or 0 if ignore_nonlinear is True.

Return type res (numeric)

step(*time*, *timestep*, *tendency=True*)
Integrate one "timestep" from "time" forward and return value

Parameters

- **time** (*single numeric*) – The time to calculate the step FROM
- **timestep** (*single numeric*) – The timestep to calculate the step
- **tendency** (*bool, optional*) – return the tendency or the actual value of the variable after the timestep?

Returns The resulting variable value or tendency

Return type numpy.ndarray

description
The numerical scheme description

Type `str`

equation
The equation the numerical scheme's should solve

Type *DerivativeEquation*

fallback_max_timestep
The numerical scheme's fallback maximum timestep

Type `float`

ignore_independent
Should this numerical scheme ignore the equation's variable-independent addend?

Type bool

ignore_linear
Should this numerical scheme ignore the equation's linear factor?

Type bool

ignore_nonlinear
Should this numerical scheme ignore the equation's nonlinear addend?

Type bool

long_description
The longer numerical scheme description

Type str

max_timestep
Return a maximum timestep for the current state. First tries the *max_timestep_estimate*, then the *fallback_max_timestep*.

Parameters

- **time** (*single numeric, optional*) – the time to calculate the derivative. Defaults to the variable's current (last) time.
- **variablevalue** (*numpy.ndarray, optional*) – the variable vaulue to use. Defaults to the value of self.variable at the given time.

Returns an estimate of the current maximum timestep

Return type float

class numericalmodel.numericalschemes.**RungeKutta4**(*description=None*, *long_description=None*, *equation=None*, *fallback_max_timestep=None*, *ignore_linear=None*, *ignore_independent=None*, *ignore_nonlinear=None*)

Bases: *numericalmodel.numericalschemes.NumericalScheme*

Runte-Kutta-4 numerical scheme

step(*time=None*, *timestep=None*, *tendency=True*)

class numericalmodel.numericalschemes.**SetOfNumericalSchemes**(*elements=[]*, *fallback_plan=None*)

Bases: *numericalmodel.utils.SetOfObjects*

Base class for sets of NumericalSchemes

Parameters

- **elements** (list of *NumericalScheme*, optional) – the the numerical schemes
- **fallback_plan** (*list, optional*) – the fallback plan if automatic planning fails. Depending on the combination of numerical scheme and equations, a certain order or solving the equations is crucial. For some cases, the order can be determined automatically, but if that fails, one has to provide this information by hand. Has to be a list of [varname, [timestep1,timestep2,...]] pairs.

  varname: the name of the equation variable. Obviously there has to be at least one entry in the list for each equation.

timestepN: the normed timesteps (betw. 0 and 1) to calculate. Normed means, that if it is requested to integrate the set of numerical equations by an overall timestep, what percentages of this timestep have to be available of this variable. E.g. an overall timestep of 10 is requested. Another equation needs this variable at the timesteps 2 and 8. Then the timesteps would be [0.2,0.8]. Obviously, the equations that looks farest into the future (e.g. Runge-Kutta or Euler-Implicit) has to be last in this fallback_plan list.

_object_to_key(*obj*)
key transformation function.

Parameters **obj** (*object*) – the element

Returns the unique key for this object. The equation's variable's id is used.

Return type key (str)

integrate(*start_time*, *final_time*)
Integrate the model until final_time

Parameters

- **start_time** (*float*) – the starting time
- **final_time** (*float*) – time to integrate until

fallback_plan
The fallback plan if automatic plan determination does not work

Type list

plan
The unified plan for this set of numerical schemes. First try to determine the plan automatically, if that fails, use the *fallback_plan*.

Type list

numericalmodel.utils module

class numericalmodel.utils.**LoggerObject**(*logger=<logging.Logger object>*)
Bases: object

Simple base class that provides a 'logger' property

Parameters **logger** (*logging.Logger*) – the logger to use

logger
the logging.Logger used for logging. Defaults to logging.getLogger(__name__).

class numericalmodel.utils.**ReprObject**
Bases: object

Simple base class that defines a *__repr__* method based on an object's __init__ arguments and properties that are named equally. Subclasses of *ReprObject* should thus make sure to have properties that are named equally as their __init__ arguments.

__repr__()
Python representation of this object

Returns a Python representation of this object based on its __init__ arguments and corresponding properties.

Return type str

classmethod **_full_variable_path**(*var*)
Get the full string of a variable

Parameters **var** (*any*) – The variable to get the full string from

Returns The full usable variable string including the module

Return type str

class numericalmodel.utils.**SetOfObjects**(*elements=[]*, *element_type=<class 'object'>*)
Bases: *numericalmodel.utils.ReprObject*, *numericalmodel.utils.LoggerObject*, collections.abc.MutableMapping

Base class for sets of objects

__str__()
Stringification

Returns a summary

Return type str

_object_to_key(*obj*)
key transformation function. Subclasses should override this.

Parameters **obj** (*object*) – object

Returns the unique key for this object. Defaults to `repr(obj)`

Return type str

add_element(*newelement*)
Add an element to the set

Parameters **newelement** (object of type *element_type*) – the new element

element_type
The base type the elements in the set should have

elements
return the list of values

Getter get the list of values

Setter set the list of values. Make sure, every element in the list is an instance of (a subclass of) *element_type*.

Type list

numericalmodel.utils.**is_numeric**(*x*)
Check if a given value is numeric, i.e. whether numeric operations can be done with it.

Parameters **x** (*any*) – the input value

Returns `True` if the value is numeric, `False` otherwise

Return type bool

numericalmodel.utils.**utcnow**()
Get the current utc unix timestamp, i.e. the utc seconds since 01.01.1970.

Returns the current utc unix timestamp in seconds

Return type float

CHAPTER 7

Indices and tables

- genindex
- modindex
- search

Python Module Index

n

Index

Symbols

A

B

D

E

P

Q

R

S

T

U

V