Numerous Documentation

FossilFree

Aug 03, 2020

Tutorial

Numerous is a simulation library for simulation

CHAPTER 1

Equation

Equations governing the behavior of the objects in the model and their interactions are defined in classes extending the *numerous.multiphysics.Equation*. Once created an equation object can easily be added to any item or subsystem. To define an equation two things are needed:

- Describe the variables needed for the equation
- Define a function that implements the equation and decorate it with the @equation decorator

Here we will describe how to create an equation object using numerous simulation engine.

1.1 Creating variable description

We start by stating a description that will be used to create a set of *numerous.engine.Variable* needed for out equation. In the constructor of the equation we can use add methods for this.

```
class Bouncing(Equation):
    def __init__(self, g=9.82, f_loss=5, x=1, y=0):
        super().__init__(tag='bouncing_eq')
        self.add_constant('g', g)
        self.add_constant('f_loss', f_loss)
        self.add_state('x', x)
        self.add_state('y', y)
        self.add_parameter('direction', 1)
```

There are 3 types of variables that can be added in such way: 1. Constant. 2. Parameter. 3. State. Adding state will although create a derivative that will be used during the solver steps. Each method requires a name of the variable and its initial value. Variable name should be unique inside the equation.

Note that some of this types are incompatible with direct assignments of values after creation. If assignments are incompatible error will be raised - like assign to STATE or CONSTANT.

1.2 Writing equations

Inside the *numerous.multiphysics.Equation* one or more functions can be defined and decorated with @equation. This decorator tells the numerous engine that it needs to execute this function and let it operate on the equations scope in each solver step.

```
@equation
def eval(self, scope):
    scope.x_dot = scope.y
    scope.y_dot = -scope.g
```

numerous.engine.Scope is a collection of all variables registered in the equation as well as global variables. it is possible to access variables scope.<Local Variable Name> for globals scope.globals.<Global Variable Name>.

Adding equations to a namespace is discussed in *Creating an Item*.

CHAPTER 2

Creating an Item

The basic building block of models in numerous. Any basic object (like a pump or pipe for instance) is created as a class extending the *numerous.engine.system.Item*. Item defines an interface for the numerous engine to determine how different objects are interacting and from that setup the function that solves the system of objects.

2.1 Creating an empty Item

To create an empty item we inherit *numerous.engine.system.Item* and pass a tag parameter to it's constructor.

```
class ThermalMass(Item):
    def __init__(self, tag):
        super().__init__(tag)
```

After defining a ThermalMass we can instantiate an object and work with it in numerous engine.

```
tm = ThermalMass(tag='ThermalMass1')
```

2.2 Adding a namespace to an empty Item

In order avoid unintended variable manipulation from different equations with same variable names `numerous.engine.system.Namespace` are created. Equations added to the same namespace will operate on the same variables – equations added to different namespaces will operate on different sets of variables and interactions will have to be explicitly defined through mappings. Namespaces can be added inside Item constructor.

```
class ThermalMass(Item):
    def __init__(self, tag='ThermalMass'):
        super().__init__(tag)
        mechanics = self.create_namespace('mechanics')
```

Now we can add `numerous.engine.system.Variable` and `numerous.engine.system.Equation` to the namespaces. On adding an Equation all variables required for the equation will be created.

CHAPTER 3

Creating a System

To enable building complex systems in a modular way, subsystems are used to define combinations of items and subsystems. Items and subsystems can be registered to a subsystem to denote it as the parent for the registered items. This allows for a hierarchical representation of the model system.

3.1 Starting with Connector

Here we will look at how to create a simple system that contains 2 basic items and the connector. We will start by inheriting a `numerous.engine.system.ConnectorTwoWay` that is special case of *numerous.engine.system.Connector* with two sides with default names for them side1 and side2.

```
class ThermalConductor(ConnectorTwoWay):
    def __init__(self, tag):
        super(ThermalConductor, self).__init__(tag)
```

Alternatively we can specify our own names for the sides of the

```
super().__init__(tag,side1_name='inlet', side2_name='outlet')
```

After we are creating namespace to this item and adding two equations to it. *update_bindings* flag show that we expect to have variables of HeatConductance in side1 and side2. *numerous.engine.OverloadAction* enum is describing an action that should be used in case of multiple reassign to variable during same step. *OverloadAction.SUM* will sum values instead of overwriting.

```
hc1 = HeatConductance(h=1001)
hc2 = HeatConductance(h=1001)

thermal = self.create_namespace('thermal')

thermal.add_equations([hc1, hc2], on_assign_overload=OverloadAction.SUM, update_
↪bindings=True)
```

Now we have namespace created not only inside the item but inside the defined bindings (in this case inside side1 and side2). We can continue with mapping variables inside the namespace to variables in bindings. Inside an item, mappings are used to map the value of one variable onto another. This is used to tell the engine that the value of one variable inside one equation is actually the value of another variable inside another equation. Mappings are defining interactions between variables not in the same namespace explicitly.

```
thermal.T1 = self.side1.thermal.T
self.side2.ns.thermal = thermal.T1
```

Now in equation in namespace thermal any access to value of variable T1 will be readdressed to item that is binded to side1.

3.2 Creating a System

Now we can create a class that inherits *numerous.engine.system.Subsystem*. and inside of it we define 2 basic items and 1 Connector item.

```
class System_Level_1(Subsystem):
    def __init__(self, tag):
        super().__init__(tag)
        TM1 = ThermalMass('TM1')
        TM2 = ThermalMass('TM2')
        C12 = ThermalConductor('TC1')

        #now we have to registered created items for the current subsystem
        self.register_items([TM1, TM2, C12])
```

Different objects in the system needs a way to interact with each other. This can be achieved by passing some object instances as an argument to another item and mapping there variables explicitly. However, in numerous bindings can be used to specify prototypes of other items and their required namespaces and variables.

Bindings furthermore enables two items to bi-directionally interact with each other, since both items can be instantiated and then bounded to each other.

Having an instance of ConnectorTwoWay and two items we can create binding by defining corespondance between sides and items:

```
C12.bind(side1=TM1,side2=TM2)
```

If we are planning to use some parts of the subsystem for other binding we can define ports inside an subsystem:

```
self.add_port('inlet', TM1)
self.add_port('outlet', TM2)
```

CHAPTER 4

Working with the model

For the engine to setup the equations, variables and mappings between these the top-level subsystem is passed to a model object. *numerous.engine.model.Model* traverses the system to collect all information needed to pass to the solver for computation – the model also back-propagates the numerical results from the solver into the system, so they can be accessed as variable values there. Model is created from the *numerous.engine.system.Subsystem*:

```
m1 = Model(model_system)
sim = Simulation(m1, t_start=0, t_stop=10, num=100, max_step=0.01)
```

4.1 Adding callbacks to model

It is possible to add two types of callbacks that will be executed during simulation. - *callback* functions. These are functions that are called each time a solver step has been completed.

```
m1 = Model(model_system)
m1.add_callback('hitground', hitground_callback_ms1)
sim = Simulation(m1, t_start=0, t_stop=10, num=100, max_step=0.01)
```

- ***event* functions. An event function uses a root-finding algorithm to detect when a certain condition is triggered.** A callback can be attached to run after any specific event.

```
m1 = Model(model_system)
m1.add_event("hitground_event", hitground_event_fun)
m1.add_event_callback("hitground_event", hitground_event_callback_fun)
sim = Simulation(m1, t_start=0, t_stop=10, num=100, max_step=0.01)
```

4.2 Creating aliases for variables

Having many nested subsytems can make it difficult to follow the changes of important variable. to highlight one of the variables we can add a special alias to it. Later we can only save such variables to history data frame

4.3 Saving and restoring state of the model

It is possible to periodically save the states of the system to the file. Such that the long running solution would not be lost.

```
hdf = HistoryDataFrame()
m1 = Model(S3('S3'), historian=hdf)

c1 = _SimulationCallback("test")
m1.save_variables_schedule(0.1, filename)

s1 = Simulation(m1, t_start=0, t_stop=2, num=100)

hdf2 = HistoryDataFrame.load(filename)
m2 = Model(S3('S3'), historian=hdf2)
m2.restore_state()
```

CHAPTER 5

Variable

These are the variables the equation function will operate on.

There are several types of variables:

-States -State derivatives -Parameters -Constants

States are the dependent variables of the system. The solver will integrate the system using the state derivates and store the result in the states.

Parameters can be used to store intermediate calculations or parameters for the system that can be changed during the simulation.

Constants are initialized with a fixed initial value and will not change over the course of the simulation.

CHAPTER 6

Scope

Each equation instance will have a scope consisting of the variables defined in the equation class with the current values of each variable in the simulation. The equation functions can only operate on the variables in its scope.

CHAPTER 7

numerous.multiphysics

7.1 Equation

class numerous.multiphysics.**Equation**

7.2 EquationDecorators

CHAPTER 8

numerous.engine.system

8.1 Node

class numerous.engine.system.**Node**(*tag=None*, *id=None*)

Top element in an items hierarchy. Contains fields to represent as uniques modelling object.

tag
Not unique tag that will be used in reports or printed output.

Type string

id
UUID assigned to the node. Generated if not provided.

Type string

get_id
returns: **id** – UUID of the element. :rtype: string

8.2 Item

class numerous.engine.system.**Item**(*tag=None*)

Base class for items hierarchy. Contains fields and methods required to process item objects. An Item in a Numerous engine are represent any object needed for simulation model.

registered_namespaces
Dictionary of namespaces registered in the current item. All registered namespaces are although added to to the __dict__ of an object and can be referenced as a class attribute.

Type dictionary of *VariableNamespaceBase*

add_callback(*callback*)

Parameters **callback** (*func*) – function to be run after each solver step.

create_namespace(*tag*)
Creating a namespace.

Parameters **tag** (*string*) – Name of a class:*numerous.engine.VariableNamespace*

Returns **new_namespace** – Empty namespace with given name

Return type class:*numerous.engine.VariableNamespace*

get_default_namespace()
Returns a new namespace with name default for the current item. This namespace is not registered.

Returns **namespace** – a default namespace.

Return type *VariableNamespaceBase*

Examples

```
>>> item = Item('example')
>>> dn = item.get_default_namespace()
>>> print(dn.item is None )
True
>>> item.register_namespace(dn)
>>> print(dn.item is None )
False
```

get_item(*item_path*)
Get an item using item path.

Parameters **item_path** (*'ItemPath'*) – hierarchical path to an item inside the subsystem

Returns **Item** – returns an item found at given path or None

Return type 'Item'

get_variables()
Get variables from registered namespaces.

Returns **variables** – all variables with corresponding registered namespace. In for of tuple (variable,namespace).

Return type list of tuples

register_namespace(*namespace*)
Registering an already existed namespace for item.

Parameters **namespace** (*VariableNamespace*) – namespace to be registered.

Raises ValueError – If namespace is already registered for this item.

8.3 ItemPath

class numerous.engine.system.**ItemPath**(*path*, *delimiter='.'*)

Represent a unique item path to item from the top of gierarchy..

path
path to item in string form

Type string

delimiter
delimiter between item objects

Type string

get_next_item_path()

Returns **item_path** – path to a nested items from the item.

Return type 'ItemPath'

get_top_item()

Returns **Item** – a name of item on top of ItemPath

Return type 'Item'

8.4 Connector

class numerous.engine.system.**Connector**(*tag*, ***kw*)
Base class for representing connectors. Object that inherited connector can be used as a connection between items.

bindings
List of binding that connector have.

Type dictionary of *Binding*

create_binding(*binding_name*)
Creating a new binding inside the connector

Parameters **binding_name** (*string*) – name of the new binding

Raises ValueError – If *binding_name* is already registered in this connector.

static create_new_binding(*binding_name*)
Creates a new *Binding* without registering it inside the connector.

Parameters **binding_name** (*string*) – Name of a binding to be created.

Returns **binding** – new binding with given name.

Return type *Binding*

get_binded_items()
Get items that are binded to the connector.

Returns **items** – all items that are binded to the connector as one list.

Return type list

update_bindings(*list_of_eq*, *binding_name*)
Updating an existing binding with the equations that are expected to be in the binded items.

Parameters

- **list_of_eq** (list of *numerous.multiphysics.Equation*) – List of a *Equation* that are expected to be in binded items.
- **binding_name** (*string*) – Name of a binding to be updated.

8.5 ConnectorItem

class numerous.engine.system.**ConnectorItem**(*tag*)
Item that can be used as a connector.

bind(***kwargs*)
Method to bind item to the bindings in current item. Biding items creating all mappings that a linked to Binding.

Parameters ****kwargs** (*Item*) – items to bind in form, binding_name = Item

create_namespace(*namespace_name*)
Creating a namespace in item and all bindings.

Parameters **namespace_name** (*string*) – Name of a *VariableNamespace*

Returns **new_namespace** – Empty namespace with given name

Return type *VariableNamespace*

8.6 Subsystem

class numerous.engine.system.**Subsystem**(*tag*)
Hierarchical representation of a group of items. Subsystems contains a set of registered items, some of such items can be subsystems itself. This allow us to create Subsystems of any complexity.

add_port(*port_tag*, *item*)
Creates a port for a subsystem. Ports can be used as an elements for binding.

Parameters

- **port_tag** (*string*) – Name of the port.
- **item** (*numerous.engine.system.Item*) – item that is be used as a port.

get_item(*item_path*)
Get an item using item path.

Parameters **item_path** (*numerous.engine.system.ItemPath*) – hierarchical path to an item inside the subsystem

Returns **Item** – returns an item found at given path or None

Return type *numerous.engine.system.Item*

register_item(*item*)

Parameters **item** (*numerous.engine.system.Item*) – Item to register in the subsystem.

register_items(*items*)

Parameters **items** (list of *numerous.engine.system.Item*) – List of items to register in the subsystem.

CHAPTER 9

numerous.engine.Variable

9.1 OverloadAction

class numerous.engine.**OverloadAction**
An enumeration.

9.2 Variable

class numerous.engine.**Variable**(*detailed_variable_description*, *base_variable=None*)

9.3 Scope

class numerous.engine.**Scope**(*scopeid*)

Allows to collect a copy of variables relevant for the specific equation.

variables
Variables relevant for the specific equation.

Type dict

add_variable(*scope_var*)
Function to add variables to the scope.

Parameters **variable** (Variable) – Original variable associated with namespace.

apply_differential_equation_rules(*is_true*)
Set if equation allow to update scope variables

is_true [bool] if we allow variable updates.

CHAPTER 10

numerous.engine.system.Binding

10.1 Binding

class numerous.engine.system.**Binding**(*binding_name*)

Represents expected binding of an *Item*. When binding is point to an actual item it checks for mapped variables and namespaces.

binding_name

Type string

add_binding(*item*)

Add an 'Item' to binded items.

Parameters **item** (*Item*) – Item to be used as a binding.

is_bindend()

Checks if item is binded to this binding.

Returns **is_bindend**

Return type bool

update_namespace(*ns*)

Updating an existing or creating a new namespace.

Parameters **ns** (*Item*) – Namespace.

CHAPTER 11

numerous.engine.system.VariableNamespaceBase

11.1 VariableNamespaceBase

class numerous.engine.system.**VariableNamespaceBase**(*item*, *tag*, *is_connector=False*, *_id=UUID('b23ff25a-d58b-11ea-9676-0242ac110002')*)

Represents a set of variables.

add_equations(*list_of_equations*, *update_bindings=True*)

Adding a list of equations to namespace. Each equation in the list is parsed and all required variables are created and registered in the namespace.

Parameters

- **list_of_equations** (*list of 'Equation'*) – list of equations to be added
- **update_bindings** (*bool*) –

  if True creates and register a binding variables in all bindings associated with item that namespace is created in.

create_variable(*name*)

Creates a variable in the namespaces with given name.

Parameters **name** (*string*) – Name of a 'Variable'

create_variable_from_desc(*variable_description*)

Creates and register a variable from given description.

Parameters

- **variable_description** (*'VariableDescription'*) – variable_description
- **on_assign_overload** (*'OverloadAction'*) – action on assign overload

get_variable(*var_tag*)

Get a variable with given description.

Parameters

- **var_description** (*'VariableDescription'*) – variable_description

  Returns

- **-------** –
- **variable** (*'Variable'*) – returns a variable matching description or None

register_variable(*variable*)

Registering existing Variable in the namespace.

Parameters **variable** (*'Variable'*) – Variable to be registered.

CHAPTER 12

numerous.engine.model

12.1 Model

class numerous.engine.model.**Model**(*system=None*, *historian=None*, *assemble=True*, *validate=False*)

The model object traverses the system to collect all information needed to pass to the solver for computation – the model also back-propagates the numerical results from the solver into the system, so they can be accessed as variable values there.

add_callback(*callback_class: numerous.utils.numba_callback.NumbaCallbackBase*) → None

add_event(*name*, *event_function*, *callbacks=None*)

Creating and adding Event callback.

Parameters

- **name** (*string*) – name of the event
- **event_function** (*callable*) –

callbacks [list of callable] callback associated with event

add_event_callback(*event_name*, *event_callback*)

Adding the callback to existing event

Parameters

- **event_name** (*string*) – name of the registered event
- **event_callback** (*callable*) – callback associated with event

assemble()

Assembles the model.

create_alias(*variable_name*, *alias*)

Parameters

- **variable_name** –
- **alias** –

get_states()

Returns **states** – list of all states.

Return type list of states

restore_state(*timestep=-1*)

Parameters

- **timestep** (*time*) – timestep that should be restored in the model. Default last known state is restored.
- **last saved state from the historian.** (*Restores*) –

save_variables_schedule(*period*, *filename*)

Save data to file on given period.

Parameters

- **period** (*timedelta*) – timedelta of saving history to file
- **filename** (*string*) – Name of a file

search_items(*item_tag*)

Search an item in items registered in the model by a tag

Returns **items** – set of items with given tag

Return type list of *numerous.engine.system.Item*

states_as_vector

Returns current states values.

Returns **state_values**

Return type array of state values

synchornize_variables()

Updates all the values of all Variable instances stored in *self.variables* with the values stored in *self.scope_vars_3d*.

validate()

Checks that all bindings are fulfilled.

CHAPTER 13

numerous.engine.simulation

13.1 Simulation

class numerous.engine.simulation.**Simulation**(*model*, *solver_type=<SolverType.SOLVER_IVP: 0>*, *t_start=0*, *t_stop=20000*, *num=1000*, *num_inner=1*, *max_event_steps=100*, *start_datetime=datetime.datetime(2020, 8, 3, 13, 17, 36, 809041)*, ***kwargs*)

Class that wraps simulation solver. currently only solve_ivp.

time
Not unique tag that will be used in reports or printed output.

Type ndarray

solver
Instance of differantial eqautions solver that will be used for the model.

Type BaseSolver

delta_t
timestep.

Type float

callbacks
Not unique tag that will be used in reports or printed output.

Type list

model
Not unique tag that will be used in reports or printed output.

CHAPTER 14

numerous.utils

14.1 HistoryDataFrame

14.2 OutputFilter

class numerous.utils.**OutputFilter**(*only_aliases=False*, *level=None*, *list_to_save=None*)

CHAPTER 15

How to Contribute

CHAPTER 16

Indices and tables

- genindex
- modindex

Python Module Index

n

Index

I

M

N

O

P

R

S

T

U

V