AntimonyCombinations

Release 0.0.1

Jan 15, 2020

Contents

AntimonyCombinations is a package developed on top of tellurium and antimony for building sbml models in a combinatorial way.

The idea is that you have a core model which you are more confident in regarding its structure and an arbitrary number of additional hypotheses, called hypothesis extensions. *AntimonyCombinations* provides a way of quickly building the comprehensive set of model topologies, given the core hypothesis and hypothesis extensions.

CHAPTER 1

Installation

CHAPTER 2

Importing the package

2.1 Combinations

class antimony_combinations.**Combinations**(*mutually_exclusive_reactions: List[Tuple[AnyStr]] = []*, *directory: Optional[str] = None*)

Builds combinations of SBML model using antimony

Create every combination of core hypothesis and extension hypotheses and creates SBML models using antimony from the tellurium package.

Combinations is designed to be subclassed. The necessary user input is given by overriding core functions and providing hypothesis extensions.

The following methods must be implemented (see below for an example):

- *core__reactions()*
- *core__parameters()*
- *core__variables()*

However the following methods are optional:

- *core__functions()*
- *core__events()*
- core__units()

Each of these methods should return a valid antimony string, since these strings are used to build up a full antimony model.

Extension hypotheses are added by adding methods to your subclass that begin with *extension_hypothesis__*. Any method that begins with *extension_hypothesis__* will be picked up and used to combinatorially build sbml models.

Any *extension_hypothesis__* method should return an instance of the *HypothesisExtension* class, which is merely a container for some needed information.

Note: Notice the double underscore after *extension_hypothesis*

Extension Hypotheses can operate in either *additive* or *replace* mode, depending on how the models should be combined. *additive* is simpler. An extension hypothesis is additive when your reaction doesn't override another, or make another reaction superflous. Examples of such instances might be when adding a mass action reaction to a preexisting set of mass action reactions.

replace mode on the other hand should be used when your reaction should be used *instead* of another reaction.

Examples:

```
# imports
from antimony_combinations import Combinations, ExtensionHypothesis
# Not needed to actually build the model set but we
#  might as well import tellurium and pycotools since we'll probably
#  want to use them for working with the model set.
import telluirum as te

class MyCombModel(Combinations):

    # no __init__ is necessary as we use the __init__ from parent class

    def core__functions(self):
        return ''' '''

    def core__variables(self):
        return '''
        compartment Cell;
        var A in Cell;
        var pA in Cell;
        var B in Cell;
        var pB in Cell;
        var C in Cell;
        var pC in Cell;

        const S in Cell
        '''

    def core__reactions(self):
        return '''
        R1f: A -> pA; k1f*A*S;
        R2f: B -> pB; k2f*B*A;
        R3f: C -> pC; k3f*C*B;
        '''

    def core__parameters(self):
        return '''
        k1f    = 0.1;
        k2f    = 0.1;
        k3f    = 0.1;

        k2b    = 0.1;
        k3b    = 0.1;
        VmaxB  = 0.1;
        kmB    = 0.1;
        VmaxA  = 0.1;
        kmA    = 0.1;
```

(continues on next page)

(continued from previous page)

```
        k4     = 0.1;

        S = 1;
        A = 10;
        pA = 0;
        B = 10;
        pB = 0;
        C = 10;
        pC = 0;
        Cell = 1;
        '''

    def core__units(self):
        return None  # Not needed for now

    def core__events(self):
        return None  # No events needed

    def extension_hypothesis__additive1(self):
        return HypothesisExtension(
            name='AdditiveReaction1',
            reaction='pB -> B',
            rate_law='k2b * pB',
            mode='additive',
            to_replace=None,  # not needed for additive mode
        )

    def extension_hypothesis__additive2(self):
        return HypothesisExtension(
            name='AdditiveReaction2',
            reaction='pC -> C',
            rate_law='k3b * C',
            mode='additive',
            to_replace=None,  # not needed for additive mode
        )

    def extension_hypothesis__replace_reaction(self):
        return HypothesisExtension(
            name='ReplaceReaction',
            reaction='pB -> B',
            rate_law='VmaxB * pB / (kmB + pB)',
            mode='replace',
            to_replace='R2f',  # name of reaction we want to replace
        )

    def extension_hypothesis__feedback1(self):
        return HypothesisExtension(
            name='Feedback1',
            reaction='pA -> A',
            rate_law='VmaxA * pA / (kmA + pA)',
            mode='additive',
            to_replace=None,  # name of reaction we want to replace
        )

    def extension_hypothesis__feedback2(self):
        return HypothesisExtension(
            name='Feedback2',
```

(continues on next page)

(continued from previous page)

```
            reaction='pA -> A',
            rate_law='k4 * pA',  # mass action variant
            mode='additive',
            to_replace=None,  # name of reaction we want to replace
        )
```

Now that we have built a Combinations subclass we can use it as follows:

```
>>> project_root = os.path.dirname(__file__)
>>> c = MyCombModel(mutually_exclusive_reactions=[
>>>         ('Feedback1', 'Feedback2')
>>>     ], directory=project_root       # optionally specify project root
>>> )
```

MyCombModel behaves like an iterator, though it doesn't store all model topologies on the outset but builds models of the fly as the *topology* attribute is incremented. Topology always starts on model 0, the core model that doesn't have additional hypothesis extensions.

```
>>> print(c)
MyCombModel(topology=0)
```

The complete set of model topologies is enumerated by the *topology* attribute. The *__len__* method is set to the size of this set, accounting for mutually exclusive topologies, which is a mechanism for reducing the topology space.

```
>>> print(len(c))
24
```

You can pick out any of these topologies using the selection operator

```
>>> print(c[4])
MyCombModel(topology=4)
```

To see which topologies correspond to which hypothesis extensions we can use `antimony_combinations.get_topologies()`, which returns a pandas.DataFrame.

```
>>> c.get_topologies()
                                                    Topology
ModelID
0                                                       Null
1                                                  additive1
2                                                  additive2
3                                                  feedback1
4                                                  feedback2
5                                             replace_reaction
6                                        additive1__additive2
7                                        additive1__feedback1
8                                        additive1__feedback2
9                                   additive1__replace_reaction
10                                       additive2__feedback1
11                                       additive2__feedback2
12                                  additive2__replace_reaction
13                                  feedback1__replace_reaction
14                                  feedback2__replace_reaction
15                             additive1__additive2__feedback1
16                             additive1__additive2__feedback2
```

(continues on next page)

(continued from previous page)

```
17                  additive1__additive2__replace_reaction
18                  additive1__feedback1__replace_reaction
19                  additive1__feedback2__replace_reaction
20                  additive2__feedback1__replace_reaction
21                  additive2__feedback2__replace_reaction
22       additive1__additive2__feedback1__replace_reaction
23       additive1__additive2__feedback2__replace_reaction
```

You can extract all topologies into a list using the `antimony_combinations.Combinations.to_list()` method.

```
>>> print(c.to_list()[:4])
[MyCombModel(topology=0),
 MyCombModel(topology=1),
 MyCombModel(topology=2),
 MyCombModel(topology=3)]
```

You can iterate over the set of topologies

```
>>> for i in c[:3]:
>>> ... print(i)
MyCombModel(topology=0)
MyCombModel(topology=1)
MyCombModel(topology=2)
```

Or use the items method, which is similar to *dict.items()*.

```
>>> for i, model in c.items()[:3]:
>>> ... print(i, model)
0 MyCombModel(topology=0)
1 MyCombModel(topology=1)
2 MyCombModel(topology=2)
```

Selecting a single model, we can create an antimony string

```
>>> first_model = c[0]
>>> print(first_model.to_antimony())
model MyCombModelTopology0
    compartment Cell;
    var A in Cell;
    var pA in Cell;
    var B in Cell;
    var pB in Cell;
    var C in Cell;
    var pC in Cell;
    const S in Cell
    R1f: A -> pA; k1f*A*S;
    R2f: B -> pB; k2f*B*A;
    R3f: C -> pC; k3f*C*B;
    k1f = 0.1;
    k2f = 0.1;
    k3f = 0.1;
    S = 1;
    A = 10;
    pA = 0;
    B = 10;
```

(continues on next page)

(continued from previous page)

```
    pB = 0;
    C = 10;
    pC = 0;
    Cell = 1;
end
```

or a tellurium model

```
>>> rr = first_model.to_roadrunner()
>>> print(rr)
<roadrunner.RoadRunner() {
'this' : 0x555a52c8cb90
'modelLoaded' : true
'modelName' :
'libSBMLVersion' : LibSBML Version: 5.17.2
'jacobianStepSize' : 1e-05
'conservedMoietyAnalysis' : false
'simulateOptions' :
< roadrunner.SimulateOptions()
{
'this' : 0x555a5309cd00,
'reset' : 0,
'structuredResult' : 0,
'copyResult' : 1,
'steps' : 50,
'start' : 0,
'duration' : 5
}>,
'integrator' :
< roadrunner.Integrator() >
  name: cvode
  settings:
      relative_tolerance: 0.000001
      absolute_tolerance: 0.000000000001
                   stiff: true
       maximum_bdf_order: 5
     maximum_adams_order: 12
       maximum_num_steps: 20000
       maximum_time_step: 0
       minimum_time_step: 0
       initial_time_step: 0
          multiple_steps: false
      variable_step_size: false
}>
```

```
>>> print(rr.simulate(0, 10, 11))
    time,      [A],      [pA],       [B],     [pB],      [C],     [pC]
 [[    0,       10,         0,        10,        0,       10,        0],
  [    1, 9.04837, 0.951626,   3.86113, 6.13887, 5.27257, 4.72743],
  [    2, 8.18731,  1.81269,   1.63214, 8.36786, 4.07751, 5.92249],
  [    3, 7.40818,  2.59182,  0.748842, 9.25116, 3.64313, 6.35687],
  [    4,  6.7032,   3.2968,  0.370018, 9.62998, 3.45361, 6.54639],
  [    5, 6.06531,  3.93469,  0.195519, 9.80448,  3.3609,  6.6391],
  [    6, 5.48812,  4.51188,  0.109779, 9.89022, 3.31158, 6.68842],
  [    7, 4.96585,  5.03415, 0.0651185, 9.93488,  3.2835,  6.7165],
  [    8, 4.49329,  5.50671, 0.0405951,  9.9594, 3.26657, 6.73343],
```

(continues on next page)

(continued from previous page)

```
  [    9,  4.0657,   5.9343, 0.0264712, 9.97353, 3.25584, 6.74416],
  [   10, 3.67879,  6.32121, 0.0179781, 9.98202, 3.24872, 6.75128]]
```

Or an interface to copasi, via pycotools3

```
>>> c.to_copasi()
Model(name=NoName, time_unit=s, volume_unit=l, quantity_unit=mol)
```

Which could be used to configure parameter estimations. Currently, support for parameter estimation configuration has in COPASI not been included but this is planned for the near future.

__init__(*mutually_exclusive_reactions: List[Tuple[AnyStr]] = []*, *directory: Optional[str] = None*) → None

Args:

mutually_exclusive_reactions: An arbitrary length list of tuples of pairs that are names of reactions that should never occur together in the same model. Defaults to an empty list.

directory: Root directory for analysis. The default is the directory containing the script being run or the current working directory of the interpreter.

copasi_file
A full path to copasi file for current topology Returns:

core__events()
Antimony events string. Do not use directly but override in subclass. Optional method.

Examples:

Returns: str

core__functions()
An optional set of functions for use in rate laws. Do not use directly but instead override in subclass.

For example:

Returns: str

core__parameters()
Parameter list. Do not use directly but over ride in subclass. This method is required.

Examples:

Returns: str

core__reactions()
List of core reactions; reactions to be shared among all models. Do not use directly as this method is designed to be subclassed. This method is required.

Examples:

Returns: str

core__variables()
List your variables whilst specifying their compartment. Method not to be used directly but overriden in subclass. This is a required method.

Examples:

Returns: str

get_hypotheses() → List[str]
Get a list of hypotheses and their index Returns:

get_parameters_as_list() → List[str]
Returns:

get_reaction_names() → List[str]

Returns: List of reaction names in current model

get_topologies() → pandas.core.frame.DataFrame
Retrieve the topology indexes and the hypotheses contained within them. This is your map between topology numbers and model hypotheses.

Returns: A pandas.DataFrame

items() → List
Similar to a *dict.items()*.

Returns: a list of tuples of the form [(i, Combinations(topology=i), . . .]

to_antimony() → str
Construct the antimony string for the current topology Returns:

to_copasi() → pycotools3.model.Model
Build a copasi file from the sbml generated from tellurium

Returns: A `tasks.Model`

topology
The ID of the current model, i.e. which topology you are currently pointing at.

Returns: Number

topology
The ID of the current model, i.e. which topology you are currently pointing at.

Returns: Number

topology_dir
A full path to a directory for files pertaining to the current topology. Currently only used for generating copasi files.

Returns:

2.2 HypothesisExtension

class antimony_combinations.**HypothesisExtension**(*name*, *reaction*, *rate_law*, *mode='additive'*, *to_replace=None*)
Data class for storing information about a hypothesis extension. For usage see *Combinations*.

Index

Symbols

C

G

H

I

T