PyBioNetGen Documentation

Release 0.2.9

Ali Sinan Saglam

Jun 07, 2021

CONTENTS:

PyBioNetGen is a lightweight command line interface (CLI) for BioNetGen. PyBioNetGen comes with a command line entry point as well as a library with useful functions. Please see *Quickstart* to learn how to install and use PyBioNetGen.

CHAPTER

ONE

QUICKSTART

1.1 Installation

You will need both python (3.7 and above) and perl installed. Once both are available you can use the following `pip` command to install PyBioNetGen

```
pip install bionetgen
```

which comes with the latest version of BioNetGen. Please note that, at the moment, PyBioNetGen does not support Atomizer but eventually will.

1.2 Basic usage

After installation complete you can test to see if PyBioNetGen is properly installed with

```
bionetgen -h
```

if this command prints out help, the command line tool is installed.

You can use PyBioNetGen to simply run a BNGL model

```
bionetgen run -i mymodel.bngl -o output_folder
```

which will create `output_folder` and run `mymodel.bngl` inside that folder. For more information on how to use PyBioNetGen please see *Command line Interface* and *Library*.

CHAPTER

TWO

COMMAND LINE INTERFACE

The command line tool comes with several subcommands. For each command you can see the help text with the command

```
bionetgen subcommand -h
```

2.1 Run

You can use this subcommand to run a model with

```
bionetgen run -i mymodel.bngl -o output_folder
```

which will run `mymodel.bngl` under the folder `output_folder`.

2.2 Plot

This subcommand allows you to make a simple plot from a gdat/cdat file

```
bionetgen plot -i mymodel.gdat -o gdat_plot.png
```

You can see all the available options by running `bionetgen plot -h`

```
optional arguments:
  -h, --help            show this help message and exit
  -i INPUT, --input INPUT
                        Path to .gdat/.cdat file to use plot
  -o OUTPUT, --output OUTPUT
                        Optional path for the plot (default:
                        "$model_name.png")
  --legend              To plot the legend or not (default: False)
  --xmin XMIN           x-axis minimum (default: determined from data)
  --xmax XMAX           x-axis maximum (default: determined from data)
  --ymin YMIN           y-axis minimum (default: determined from data)
  --ymax YMAX           y-axis maximum (default: determined from data)
  --xlabel XLABEL       x-axis label (default: time)
  --ylabel YLABEL       y-axis label (default: concentration)
  --title TITLE         title of plot (default: determined from input file)
```

2.3 Notebook

This subcommand is in it's early stages of development. The subcommand is used to generate a simple Jupyter notebook. You can also give your model as an argument and the resulting notebook will be ready to load in your model using PyBioNetGen library.

```
bionetgen notebook -i mymodel.bngl -o mynotebook.ipynb
```

CHAPTER

THREE

LIBRARY

PyBioNetGen also comes with a library that allows you to programatically run and do simple modifications of BNGL models.

3.1 run

This method allows you to do a simple run of a BNGL model and returns the results as numpy record arrays.

```
import bionetgen
result = bionetgen.run("mymodel.bngl", output="myfolder")
result["mymodel"] # this will contain the gdat results of the run
```

3.2 bngmodel

This method allows you to load in a model into a python object.

```
import bionetgen
model = bionetgen.bngmodel("mymodel.bngl") # generates BNG-XML and reads it
print(model)
```

Please see *bngmodel* for more all the features this object supports.

3.2.1 bngmodel.setup_simulator

This method allows you to get a libroadrunner simulator of the loaded model.

```
import bionetgen
model = bionetgen.bngmodel("mymodel.bngl") # generates BNG-XML and reads it
librr_simulator = model.setup_simulator().simulator
librr_simulator.simulate(0,1,10) # librr_simulator is the simulator object
```

This is an easy way to generate data for analyses of your model using Python.

CHAPTER

FOUR

BNGMODEL

4.1 Basic usage

This is designed to be a pythonic object representing the BNGL model given. It currently has some limited options to modify the model. You can load the model object using

```
import bionetgen
model = bionetgen.bngmodel("mymodel.bngl") # generates BNG-XML and reads it
```

The underlying code will attempt to generate a BNG-XML of the model which it then reads to generate this object.

One core principle of this object is that the object and every object associated with it can be converted to a string to get the BNGL string of the object itself. For example

```
print(model) # this prints the entire model
print(model.observables) # prints the observables block
print(model.parameters) # prints the parameters block
model.parameters.k = 10 # sets parameter k to 10
print(model.parameters) # block updated to reflect change
```

The BNGL string is dynamically generated and not just returning the block string from the original model. This allows for making simple changes to your model, e.g.

```
for i in range(10):
    model.parameters.k = i
    with open("param_k_{}.bngl".format(i), "w") as f:
        f.write(str(model))
```

This will write 10 models with different k parameters.

4.2 Blocks

All blocks that are active can be seen with print(model.active_blocks). Currently supported blocks are

- Parameters
- Compartments
- Molecule types
- Species (or seed species)
- Observables

- Functions
- Reaction rules

PyBioNetGen bngmodel also recognizes actions within the model but discards them upon loading (this will eventually be optional). All bionetgen features will eventually be supported by this library, including every valid BNGL block.

Blocks also act pythonic and act like other python objects

```
for param in model.parameters:
    print("parameter name: {}".format(param))
    print("parameter value: {}".format(model.parameters[param]))

for obs in model.observables:
    obs_val = model.observables[obs]
    print("observable name: {}".format(obs))
    print("observable type: {}".format(obs_val[0]))
    print("observable pattern: {}".format(obs_val[1]))

for spec in model.species:
    spec_count = model.species[spec]
    print("species name: {}".format(spec))
    print("species count: {}".format(spec_count))
    print("molecules in species: {}".format(spec.molecules))
```

The following sections will detail how each block behaves

4.2.1 Parameters

Parameters are a list of names and values associated with those names. Parameters block also stores the parameter expressions in case they are written as functions in the original model.

```
# let's say we have a parameter k
model.parameters["k"] = 10 # this is the parameter value
model.parameters.k = 10 # this is also the parameter value
model.parameters.expressions["k"] # this is the parameter expression
```

4.2.2 Compartments

Compartments are comprised of a compartment name, dimensionality, volume and an optional parent compartment name

```
# say we have a compartment string "PM 2 10.0 EC"
# which is a 2 dimensional compartment with 2 dimensions and 10 volume
# and is contained under another compartment EC
comp_name = model.compartments[i] # where i is the index of PM compartment, will return
↪"PM"
comp_list = model.compartments[comp_name] # will return [2, 10.0, "EC"]
print(comp_list[0]) # will print 2
print(comp_list[1]) # will print 10.0
print(comp_list[2]) # will print EC
```

4.2.3 Molecule types

Molecule types contains different components and all possible states of those components

```
# let's say we have a molecule type "X()" as the first one
X_obj = model.molecule_types[0] # this is the object for "X()" molecule type
print(X_obj) # will print the molecule type string
X_obj.add_component("p", states=["0","1"]) # adds a component with states
print(X_obj) # prints "X(p~0~1)" now
```

4.2.4 Species

Species are made up of molecules and can contain an overall compartment and label.

```
# let's say we have a species with pattern "X()"
species_obj = model.species[0] # this is the species object
print(species_obj) # prints the pattern
count = model.species[species_obj] # this is the starting count of the species
count = model.species["X()"] # this is the starting count of the species
molecules = species_obj.molecules # this is the list of molecules in the pattern
compartment = species_obj.compartment # this is the overall compartment of the species
label = species_obj.label # this is the overall label of the species
```

4.2.5 Observables

Observables are made up of a list of species patterns

```
# let's say we have a observable with string "Molecules X_phos X(p~1)"
obs_obj = model.observables[0] # this is the observable object
print(obs_obj) # prints the observable patterns, in this case X(p~1)
obs_list = model.observables["X_phos"] # this returns a list of two items
type, obs_obj = obs_list # first one is the type, "Molecules" and second one is the␣
↪object again
patterns = obs_obj.patterns # this is the list of patterns in the observable string
```

4.2.6 Functions

Functions are just a tuple of function name and expression

```
# say we have a function f() = 10*kon
func_name = model.functions[0] # this will return function name f()
func_expr = model.functions[func_name] # this will return function expression "10*kon"
```

4.2.7 Reaction rules

Reaction rules consist of two lists of species, one for reactants and one for products as well as a list of rate constants. There is a single rate constant if the rule is unidirectional and two rate constants if the rule is bidirectional.

```
# Let's say we have a rule: R1: A() + B() <-> C() kon,koff
rule_name = model.rules[0] # this will return "R1" string
rule_obj = model.rules[rule_name] # this is the full rule object
print(rule_obj) # prints bngl string
print(rule_obj.reactants) # prints the list [A(), B()], where each item is a species object
print(rule_obj.products) # prints the list [C()], where each item is a species object
print(rule_obj.rate_constants) # prints the list ["kon","koff"]
print(rule_obj.bidirectional) # prints true since
```

CHAPTER

FIVE

INDICES AND TABLES

- genindex
- modindex
- search