PyGOM Documentation

Release 0.1.7.dev61+g5c10d9c

Edwin Tye

Jul 02, 2020

Contents

pygom is a package that aims to facilitate the application of ordinary differential equations (ODEs) in the real world, with a focus in epidemiology. This package helps the end user define their ODE system in an intuitive manner and provides convenience functions - making use of various algebraic and numerical libraries in the backend - that can be used in a straight forward fashion.

This is an open source project hosted on Github.

CHAPTER 1

User Documentation

1.1 Getting started

1.1.1 What this package does

The purpose of this package is to allow the end user to easily define a set of ordinary differential equations (ode) and obtain information about the ode by simply invoking the the appropriate methods. Here, we define the set of ode's as

$$\frac{\partial \mathbf{x}}{\partial t} = f(\mathbf{x}, \boldsymbol{\theta})$$

where $\mathbf{x} = (x_1, x_2, \ldots, x_n)$ is the state vector with d state and $\boldsymbol{\theta}$ the parameters of p dimension. Currently, this package allows the end user to find the algebraic expression of the ode, Jacobian, gradient and forward sensitivity of the ode. A numerical output is given when all the state and parameter values are provided. Note that the only important class is `DeterministicOde` all the functionality described previously are exposed.

The current plan is to extend the functionality to include

- Solving the ode analytically when it is linear
- Analysis of the system via eigenvalues during the integration
- Detection of DAE

1.1.2 Obtaining the package

The location of the package is current on GitHub and can be pulled via https from:

```
https://github.com/PublicHealthEngland/pygom.git
```

The package is currently as follows:

```
pygom/
    bin/
    doc/
    pygom/
        loss/
            tests/
        model/
            tests/
        sbml_translate/
        utilR/
    LICENSE.txt
    README.rst
    requirements.txt
    setup.py
```

with files in each of the three main folder not shown. You can install the package via command line:

```
python setup.py install
```

or locally on a user level:

```
python setup.py install --user
```

Please note that there are current redundant file are kept for development purposes for the time being.

1.1.3 Testing the package

Testing can be performed prior or after the installation. Some standard test files can be found in their respective folder and they can be run in the command line:

```
python setup.py test
```

which can be performed prior to installing the package if desired.

1.2 Transition Object

The most important part of setting up the model is to correctly define the set odes, which is based solely on the classes defined in `transition`. All transitions that gets fed into the ode system needs to be defined as a transition object, `Transition`. It takes a total of four input arguments

1. The origin state
2. Equation that describe the process
3. The type of transition
4. The destination state

where the first three are mandatory. To demonstrate, we go back to the SIR model defined previously in the section sir. Recall that the set of odes are

$$\frac{\partial S}{\partial t} = -\beta SI$$
$$\frac{\partial I}{\partial t} = \beta SI - \gamma I$$
$$\frac{\partial R}{\partial t} = \gamma I.$$

We can simply define the set of ode, as seen previously, via

```
In [1]: from pygom import Transition, TransitionType, common_models

In [2]: ode1 = Transition(origin='S', equation='-beta*S*I', transition_
↪type=TransitionType.ODE)

In [3]: ode2 = Transition(origin='I', equation='beta*S*I - gamma*I', transition_
↪type=TransitionType.ODE)

In [4]: ode3 = Transition(origin='R', equation='gamma*I', transition_
↪type=TransitionType.ODE)
```

Note that we need to state explicitly the type of equation we are inputting, which is simply of type **ODE** in this case. We can confirm this has been entered correctly by putting it into `DeterministicOde`

```
In [5]: from pygom import DeterministicOde

In [6]: stateList = ['S', 'I', 'R']

In [7]: paramList = ['beta', 'gamma']

In [8]: model = DeterministicOde(stateList,
   ...:                          paramList,
   ...:                          ode=[ode1, ode2, ode3])
   ...:
```

and check it

```
In [9]: model.get_ode_eqn()
Out[9]:
Matrix([
[         -I*S*beta],
[I*S*beta - I*gamma],
[           I*gamma]])
```

An alternative print function `print_ode()` is also available which may be more suitable in other situation. The default prints the formula in a rendered format and another which prints out the latex format which can be used directly in a latex document. The latter is useful as it saves typing out the formula twice, once in the code and another in documents.

```
In [10]: model.print_ode(False)
dS/dt=    -ISβ

dI/dt=  ISβ - Iγ

dR/dt=      Iγ

In [11]: model.print_ode(True)
\begin{array}{cc}dS/dt= & - I S \beta\\dI/dt= & I S \beta - I \gamma\\dR/dt= & I␣
↪\gamma\end{array}
```

Now we are going to show the different ways of defining the same set of odes.

1.2.1 Defining the equations

Recognizing that the set of odes defining the SIR model is the result of two transitions,

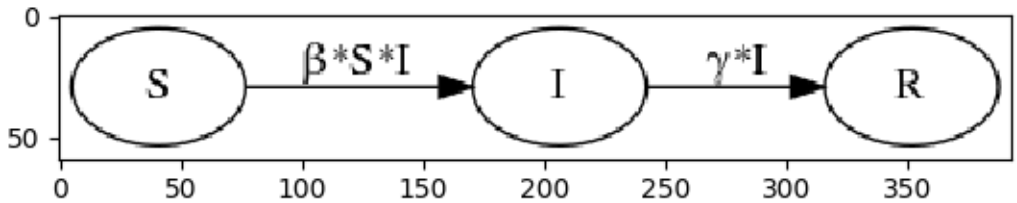
$$S \rightarrow I = \beta SI$$
$$I \rightarrow R = \gamma I$$

where $S \rightarrow I$ denotes a transition from state S to state I. Therefore, we can simply define our model by these two transition, but now these two transition needs to be inputted via the `transition` argument instead of the `ode` argument. Note that we are initializing the model using a different class, because the stochastic implementation has more operation on transitions.

```
In [12]: from pygom import SimulateOde

In [13]: t1 = Transition(origin='S', destination='I', equation='beta*S*I', transition_
↪type=TransitionType.T)

In [14]: t2 = Transition(origin='I', destination='R', equation='gamma*I', transition_
↪type=TransitionType.T)

In [15]: modelTrans = SimulateOde(stateList,
   ....:                          paramList,
   ....:                          transition=[t1, t2])
   ....:

In [16]: modelTrans.get_ode_eqn()
Out[16]:
Matrix([
[         -I*S*beta],
[I*S*beta - I*gamma],
[           I*gamma]])
```

We can see that the resulting ode is exactly the same, as expected. The transition matrix that defines this process can easily be visualized using graphviz. Because only certain renderer permit the use of sub and superscript, operators such as $**$ are left as they are in the equation.

```
In [17]: import matplotlib.pyplot as plt

In [18]: f = plt.figure()

In [19]: modelTrans.get_transition_matrix()
Out[19]:
Matrix([
[0, I*S*beta,       0],
[0,        0, I*gamma],
[0,        0,       0]])

In [20]: dot = modelTrans.get_transition_graph()
```



If we put in via the wrong argument like below (not run), then an error will appear.

```
In [21]: # modelTrans = DeterministicOde(stateList, paramList, ode=[t1, t2])
```

because `TranstionType` was defined explicitly as a transition instead of an ode. The same can be observed when the wrong `TransitionType` is used for any of the input argument.

This though, only encourages us to define the transitions carefully. We can also pretend that the set of odes are in fact just a set of birth process

```
In [22]: birth1 = Transition(origin='S', equation='-beta*S*I', transition_
↪type=TransitionType.B)

In [23]: birth2 = Transition(origin='I', equation='beta*S*I - gamma*I', transition_
↪type=TransitionType.B)

In [24]: birth3 = Transition(origin='R', equation='gamma*I', transition_
↪type=TransitionType.B)

In [25]: modelBirth = DeterministicOde(stateList,
   ....:                               paramList,
   ....:                               birth_death=[birth1, birth2, birth3])
   ....:

In [26]: modelBirth.get_ode_eqn()
Out[26]:
```

(continues on next page)

(continued from previous page)

```
Matrix([
[         -I*S*beta],
[I*S*beta - I*gamma],
[           I*gamma]])
```

which will yield the same set result. Alternatively, we can use the negative of the equation but set it to be a death process. For example, we multiply the equations for state S and R with a negative sign and set the transition type to be a death process instead.

```
In [27]: death1 = Transition(origin='S', equation='beta*S*I', transition_
↪type=TransitionType.D)

In [28]: birth2 = Transition(origin='I', equation='beta*S*I - gamma*I', transition_
↪type=TransitionType.B)

In [29]: death3 = Transition(origin='R', equation='-gamma*I', transition_
↪type=TransitionType.D)

In [30]: modelBD = DeterministicOde(stateList,
   ....:                            paramList,
   ....:                            birth_death=[death1, birth2, death3])
   ....:

In [31]: modelBD.get_ode_eqn()
Out[31]:
Matrix([
[         -I*S*beta],
[I*S*beta - I*gamma],
[           I*gamma]])
```

We can see that all the above ways yield the same set of ode at the end.

1.2.2 Model Addition

Because we allow the separation of transitions between states and birth/death processes, the birth/death processes can be added later on.

```
In [32]: modelBD2 = modelTrans

In [33]: modelBD2.param_list = paramList + ['mu', 'B']

In [34]: birthDeathList = [Transition(origin='S', equation='B', transition_
↪type=TransitionType.B),
   ....:                   Transition(origin='S', equation='mu*S', transition_
↪type=TransitionType.D),
   ....:                   Transition(origin='I', equation='mu*I', transition_
↪type=TransitionType.D)]
   ....:

In [35]: modelBD2.birth_death_list = birthDeathList

In [36]: modelBD2.get_ode_eqn()
Out[36]:
Matrix([
[      B - I*S*beta - S*mu],
```

(continues on next page)

(continued from previous page)

```
[I*S*beta - I*gamma - I*mu],
[                   I*gamma]])
```

So modeling can be done in stages. Start with a standard closed system and extend it with additional flows that interact with the environment.

1.2.3 Transition type

There are currently four different type of transitions allowed, which is defined in an enum class also located in `transition`. The four types are B, D, ODE and T, where they represent different type of process with explanation in their corresponding value.

```
In [37]: from pygom import transition

In [38]: for i in transition.TransitionType:
   ....:     print(str(i) + " = " + i.value)
   ....:
TransitionType.B = Birth process
TransitionType.D = Death process
TransitionType.T = Between states
TransitionType.ODE = ODE _equation
```

Each birth process are added to the origin state while each death process are deducted from the state, i.e. added to the state after multiplying with a negative sign. An ode type is also added to the state and we forbid the number of input ode to be greater than the number of state inputted.

1.3 Stochastic representation of ode

There are multiple interpretation of stochasticity of a deterministic ode. We have implemented two of the most common interpretation; when the parameters are realizations of some underlying distribution, and when we have a so called chemical master equation where each transition represent a jump. Again, we use the standard SIR example as previously seen in ref:*sir*.

```
In [1]: from pygom import SimulateOde, Transition, TransitionType

In [2]: import matplotlib.pyplot as plt

In [3]: import numpy as np

In [4]: x0 = [1, 1.27e-6, 0]

In [5]: t = np.linspace(0, 150, 100)

In [6]: stateList = ['S', 'I', 'R']

In [7]: paramList = ['beta', 'gamma']

In [8]: transitionList = [
   ...:                   Transition(origin='S', destination='I', equation='beta*S*I',
↪ transition_type=TransitionType.T),
   ...:                   Transition(origin='I', destination='R', equation='gamma*I',␣
↪transition_type=TransitionType.T)
```

(continues on next page)

(continued from previous page)

```
   ...:                    ]
   ...:

In [9]: odeS = SimulateOde(stateList, paramList, transition=transitionList)

In [10]: odeS.parameters = [0.5, 1.0/3.0]

In [11]: odeS.initial_values = (x0, t[0])

In [12]: solutionReference = odeS.integrate(t[1::], full_output=False)
```

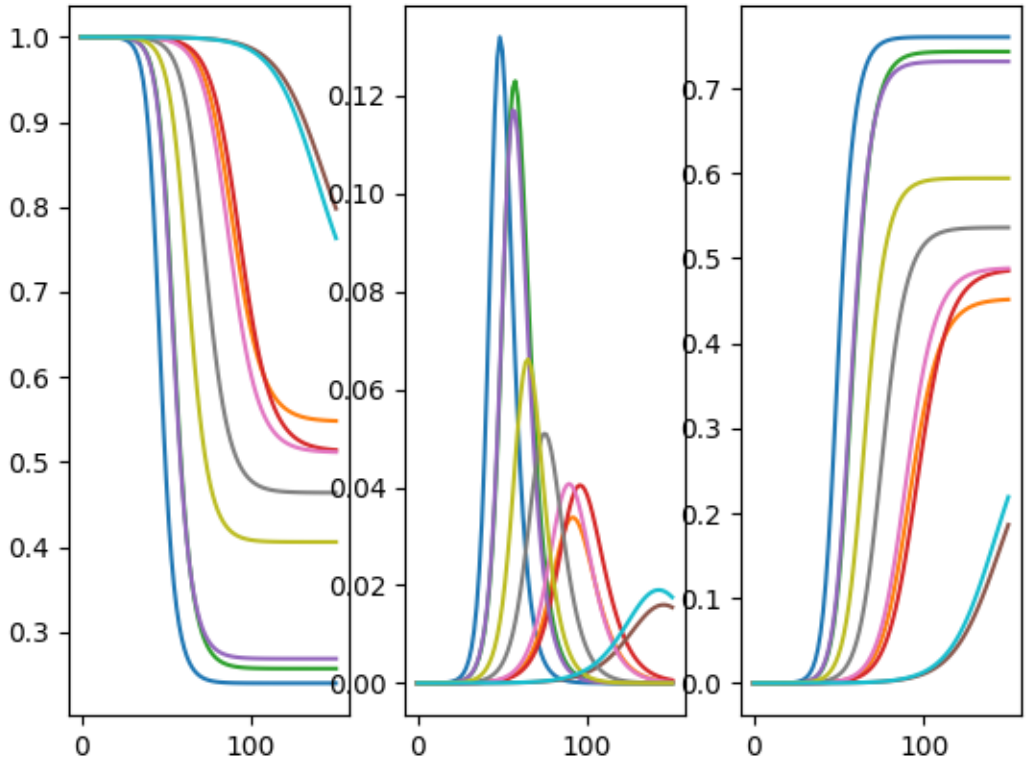
1.3.1 Stochastic Parameter

In our first scenario, we assume that the parameters follow some underlying distribution. Given that both β and γ in our SIR model has to be non-negative, it seemed natural to use a Gamma distribution. We make use of the familiar syntax from R to define our distribution. Unfortunately, we have to define it via a tuple, where the first is the function handle (name) while the second the parameters. Note that the parameters can be defined as either a dictionary or as the same sequence as R, which is the shape then the rate in the Gamma case.

```
In [13]: from pygom.utilR import rgamma

In [14]: d = dict()

In [15]: d['beta'] = (rgamma,{'shape':100.0, 'rate':200.0})

In [16]: d['gamma'] = (rgamma,(100.0, 300.0))

In [17]: odeS.parameters = d

In [18]: Ymean, Yall = odeS.simulate_param(t[1::], 10, full_output=True)
```

Note that a message is printed above where it is trying to connect to an mpi backend, as our module has the capability to compute in parallel using the IPython. We have simulated a total of 10 different solutions using different parameters, the plots can be seen below

```
In [19]: f, axarr = plt.subplots(1,3)

In [20]: for solution in Yall:
   ....:     axarr[0].plot(t, solution[:,0])
   ....:     axarr[1].plot(t, solution[:,1])
   ....:     axarr[2].plot(t, solution[:,2])
   ....:

In [21]: plt.show()

In [22]: plt.close()
```



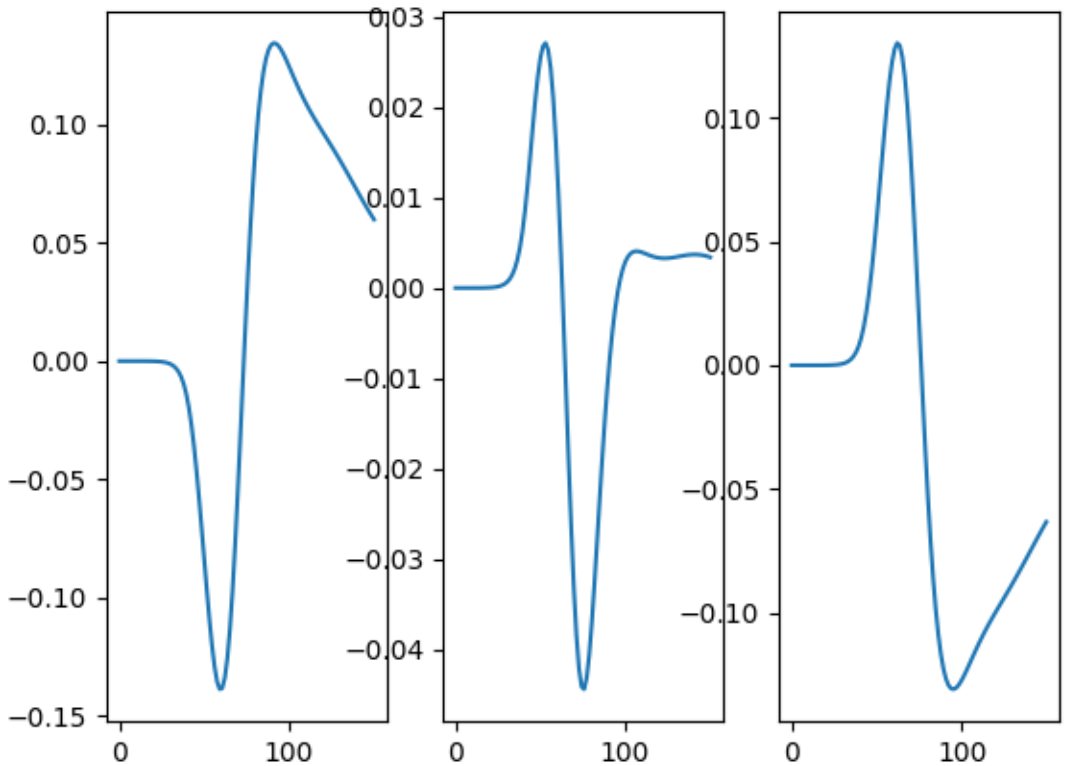
We then see how the expected results, using the sample average of the simulations

$$\tilde{x}(T) = \mathbb{E}\left[\int_{t_0}^{T} f(\theta, x, t)dt\right]$$

differs from the reference solution

$$\hat{x}(T) = \int_{t_0}^{T} f(\mathbb{E}\left[\theta\right], x, t)dt$$

```
In [23]: f, axarr = plt.subplots(1,3)

In [24]: for i in range(3): axarr[i].plot(t, Ymean[:,i] - solutionReference[:,i])

In [25]: plt.show()

In [26]: plt.close()
```



The difference is relatively large especially for the S state. We can decrease this difference as we increase the number of simulation, and more sophisticated sampling method for the generation of random variables can also decrease the difference.

In addition to using the built-in functions to represent stochasticity, we can also use standard frozen distributions from scipy. Note that it must be a frozen distribution as that is the only for the parameters of the distributions to propagate through the model.

```
In [27]: import scipy.stats as st

In [28]: d = dict()

In [29]: d['beta'] = st.gamma(a=100.0, scale=1.0/200.0)

In [30]: d['gamma'] = st.gamma(a=100.0, scale=1.0/300.0)

In [31]: odeS.parameters = d
```

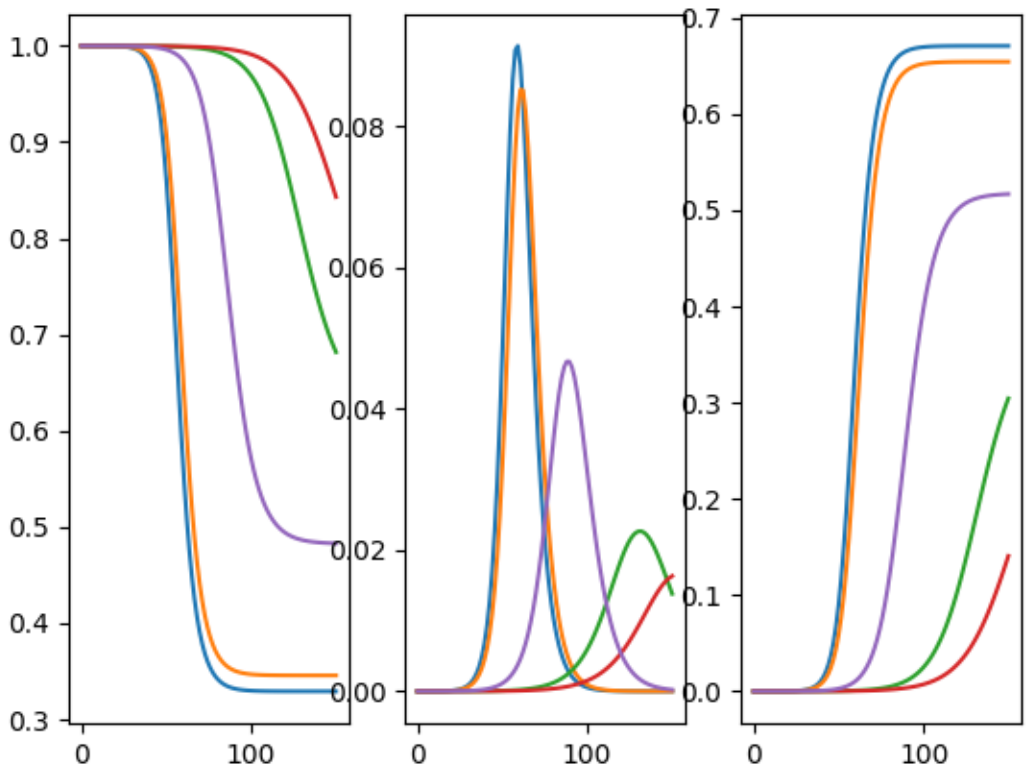
Obviously, there may be scenarios where only some of the parameters are stochastic. Let's say that the γ parameter is fixed at $1/3$, then simply replace the distribution information with a scalar. A quick visual inspection at the resulting plot suggests that the system of ODE potentially has less variation when compared to the case where both parameters are stochastic.

```
In [32]: d['gamma'] = 1.0/3.0
```

(continues on next page)

(continued from previous page)

```
In [33]: odeS.parameters = d

In [34]: YmeanSingle, YallSingle = odeS.simulate_param(t[1::], 5, full_output=True)

In [35]: f, axarr = plt.subplots(1,3)

In [36]: for solution in YallSingle:
    ....:     axarr[0].plot(t,solution[:,0])
    ....:     axarr[1].plot(t,solution[:,1])
    ....:     axarr[2].plot(t,solution[:,2])
    ....:

In [37]: plt.show()

In [38]: plt.close()
```



1.3.2 Continuous Markov Representation

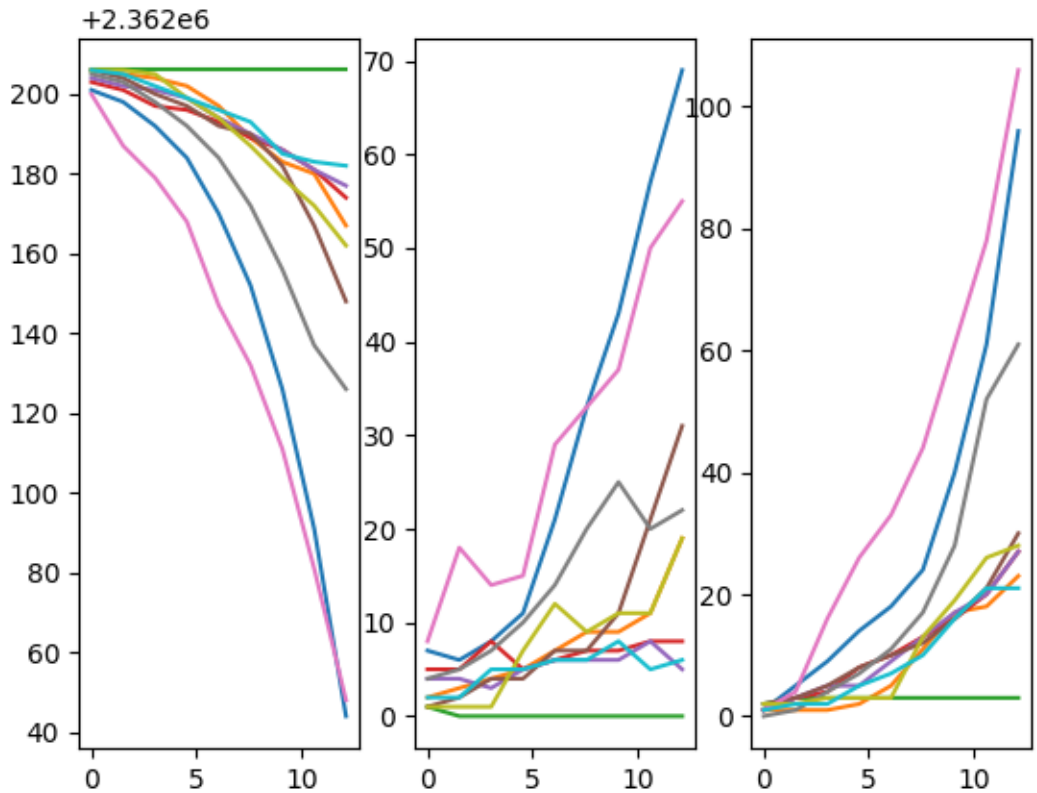
Another common method of introducing stochasticity into a set of ode is by assuming each movement in the system is a result of a jump process. More concretely, the probabilty of a move for transition j is governed by an exponential distribution such that

$$\Pr(\text{process } j \text{ jump within time } \tau) = \lambda_j e^{-\lambda_j \tau},$$

where λ_j is the rate of transition for process j and τ the time elapsed after current time t.

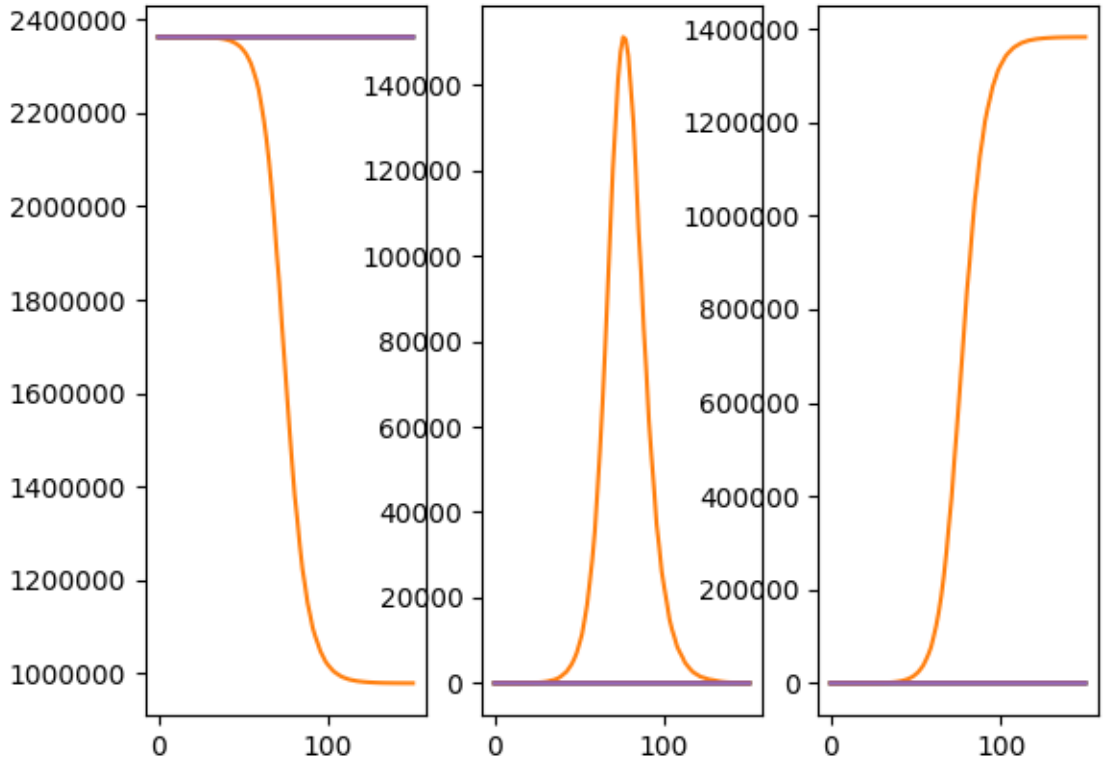
A couple of the commmon implementation for the jump process have been implemented where two of them are used during a normal simulation; the first reaction method [Gillespie1977] and the τ-Leap method [Cao2006]. The two changes interactively depending on the size of the states.

```
In [39]: x0 = [2362206.0, 3.0, 0.0]

In [40]: stateList = ['S', 'I', 'R']

In [41]: paramList = ['beta', 'gamma', 'N']

In [42]: transitionList = [
   ....:                   Transition(origin='S', destination='I', equation='beta*S*I/
↪N', transition_type=TransitionType.T),
   ....:                   Transition(origin='I', destination='R', equation='gamma*I',
↪ transition_type=TransitionType.T)
   ....:                   ]
   ....:

In [43]: odeS = SimulateOde(stateList, paramList, transition=transitionList)

In [44]: odeS.parameters = [0.5, 1.0/3.0, x0[0]]

In [45]: odeS.initial_values = (x0, t[0])

In [46]: solutionReference = odeS.integrate(t[1::])

In [47]: simX, simT = odeS.simulate_jump(t[1:10], 10, full_output=True)

In [48]: f, axarr = plt.subplots(1, 3)

In [49]: for solution in simX:
   ....:     axarr[0].plot(t[:9], solution[:,0])
   ....:     axarr[1].plot(t[:9], solution[:,1])
   ....:     axarr[2].plot(t[:9], solution[:,2])
   ....:

In [50]: plt.show()

In [51]: plt.close()
```



Above, we see ten different simulation, again using the SIR model but without standardization of the initial conditions. We restrict our time frame to be only the first 10 time points so that the individual changes can be seen more clearly above. If we use the same time frame as the one used previously for the deterministic system (as shown below), the trajectories are smoothed out and we no longer observe the *jumps*. Looking at the raw trajectories of the ODE below, it is obvious that the mean from a jump process is very different to the deterministic solution. The reason behind this is that the jump process above was able to fully remove all the initial infected individuals before any new ones.

```
In [52]: simX,simT = odeS.simulate_jump(t, 5, full_output=True)

In [53]: simMean = np.mean(simX, axis=0)

In [54]: f, axarr = plt.subplots(1,3)

In [55]: for solution in simX:
   ....:     axarr[0].plot(t, solution[:,0])
   ....:     axarr[1].plot(t, solution[:,1])
   ....:     axarr[2].plot(t, solution[:,2])
   ....:

In [56]: plt.show()

In [57]: plt.close()
```



1.3.3 Repeatable Simulation

One of the possible use of compartmental models is to generate forecasts. Although most of the time the requirement would be to have (at least point-wise) convergence in the limit, reproducibility is also important. For both types of interpretation explained above, we have given the package the capability to repeat the simulations by setting a seed. When the assumption is that the parameters follows some sort of distribution, we simply set the seed which governs the global state of the random number generator.

```
In [58]: x0 = [2362206.0, 3.0, 0.0]

In [59]: odeS = SimulateOde(stateList, paramList, transition=transitionList)

In [60]: d = {'beta': st.gamma(a=100.0, scale=1.0/200.0), 'gamma': st.gamma(a=100.0,␣
↪scale=1.0/300.0), 'N': x0[0]}

In [61]: odeS.parameters = d

In [62]: odeS.initial_values = (x0, t[0])

In [63]: Ymean, Yall = odeS.simulate_param(t[1::], 10, full_output=True)

In [64]: np.random.seed(1)

In [65]: Ymean1, Yall1 = odeS.simulate_param(t[1::], 10, full_output=True)
```

(continues on next page)

(continued from previous page)

```
In [66]: np.random.seed(1)

In [67]: Ymean2, Yall2 = odeS.simulate_param(t[1::], 10, full_output=True)

In [68]: sim_diff = [np.linalg.norm(Yall[i] - yi) for i, yi in enumerate(Yall1)]

In [69]: sim_diff12 = [np.linalg.norm(Yall2[i] - yi) for i, yi in enumerate(Yall1)]

In [70]: print("Different in the simulations and the mean: (%s, %s) " % (np.sum(sim_
↪diff), np.sum(np.abs(Ymean1 - Ymean))))
Different in the simulations and the mean: (76408206.74509755, 15052214.00569588)

In [71]: print("Different in the simulations and the mean using same seed: (%s, %s) "
↪% (np.sum(sim_diff12), np.sum(np.abs(Ymean2 - Ymean1))))
Different in the simulations and the mean using same seed: (0.0, 0.0)
```

In the alternative interpretation, setting the global seed is insufficient. Unlike simulation based on the parameters, where we can pre-generate all the parameter values and send them off to individual processes in the parallel backend, this is prohibitive here. In a nutshell, the seed does not propagate when using a parallel backend because each *integration* requires an unknown number of random samples. Therefore, we have an additional flag **parallel** in the function signature. By ensuring that the computation runs in serial, we can make use of the global seed and generate identical runs.

```
In [72]: x0 = [2362206.0, 3.0, 0.0]

In [73]: odeS = SimulateOde(stateList, paramList, transition=transitionList)

In [74]: odeS.parameters = [0.5, 1.0/3.0, x0[0]]

In [75]: odeS.initial_values = (x0, t[0])

In [76]: simX, simT = odeS.simulate_jump(t[1:10], 10, parallel=False, full_
↪output=True)

In [77]: np.random.seed(1)

In [78]: simX1, simT1 = odeS.simulate_jump(t[1:10], 10, parallel=False, full_
↪output=True)

In [79]: np.random.seed(1)

In [80]: simX2, simT2 = odeS.simulate_jump(t[1:10], 10, parallel=False, full_
↪output=True)

In [81]: sim_diff = [np.linalg.norm(simX[i] - x1) for i, x1 in enumerate(simX1)]

In [82]: sim_diff12 = [np.linalg.norm(simX2[i] - x1) for i, x1 in enumerate(simX1)]

In [83]: print("Difference in simulation: %s" % np.sum(np.abs(sim_diff)))
Difference in simulation: 2534.4614923250074

In [84]: print("Difference in simulation using same seed: %s" % np.sum(np.abs(sim_
↪diff12)))
Difference in simulation using same seed: 0.0
```

1.4 Convert ODE into transitions

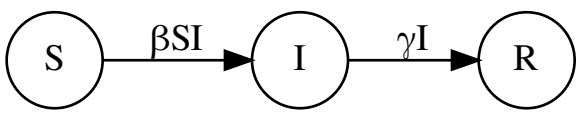
As seen previously in *Transition Object*, we can define the model via the transitions or explicitly as ODEs. There are times when we all just want to test out some model in a paper and the only available information are the ODEs themselves. Even though we know that the ODEs come from some underlying transitions, breaking them down can be a time consuming process. We provide the functionalities to do this automatically.

1.4.1 Simple Problem

For a simple problem, we consider the SIR model defined by

$$\frac{dS}{dt} = -\beta SI$$
$$\frac{dI}{dt} = \beta SI - \gamma I$$
$$\frac{dR}{dt} = \gamma I.$$

which consists of two transitions



Let's define this using the code block below

```
In [1]: from pygom import SimulateOde, Transition, TransitionType

In [2]: ode1 = Transition(origin='S', equation='-beta*S*I', transition_
↪type=TransitionType.ODE)

In [3]: ode2 = Transition(origin='I', equation='beta*S*I - gamma*I', transition_
↪type=TransitionType.ODE)

In [4]: ode3 = Transition(origin='R', equation='gamma*I', transition_
↪type=TransitionType.ODE)

In [5]: stateList = ['S', 'I', 'R']

In [6]: paramList = ['beta', 'gamma']

In [7]: ode = SimulateOde(stateList,
   ...:                   paramList,
   ...:                   ode=[ode1, ode2, ode3])
   ...:

In [8]: ode.get_transition_matrix()
Out[8]:
Matrix([
```

(continues on next page)

(continued from previous page)

```
[0, 0, 0],
[0, 0, 0],
[0, 0, 0]])
```

and the last line shows that the transition matrix is empty. This is the expected result because `SimulateOdeModel` was not initialized using transitions. We populate the transition matrix below and demonstrate the difference.

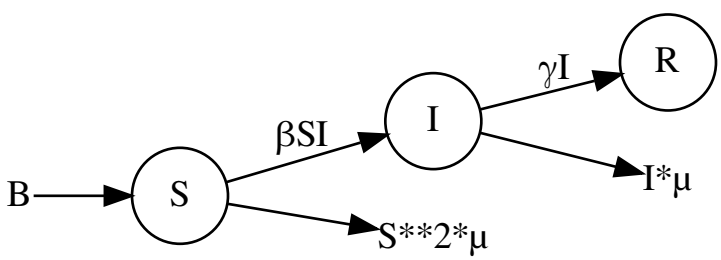
```
In [9]: ode = ode.get_unrolled_obj()

In [10]: ode.get_transition_matrix()
Out[10]:
Matrix([
[0, I*S*beta,       0],
[0,        0, I*gamma],
[0,        0,       0]])
```

1.4.2 ODE With Birth and Death Process

We follow on from the SIR model of *Simple Problem* but with additional birth and death processes.

$$\frac{dS}{dt} = -\beta SI + B - \mu S$$
$$\frac{dI}{dt} = \beta SI - \gamma I - \mu I$$
$$\frac{dR}{dt} = \gamma I.$$

which consists of two transitions and three birth and death process



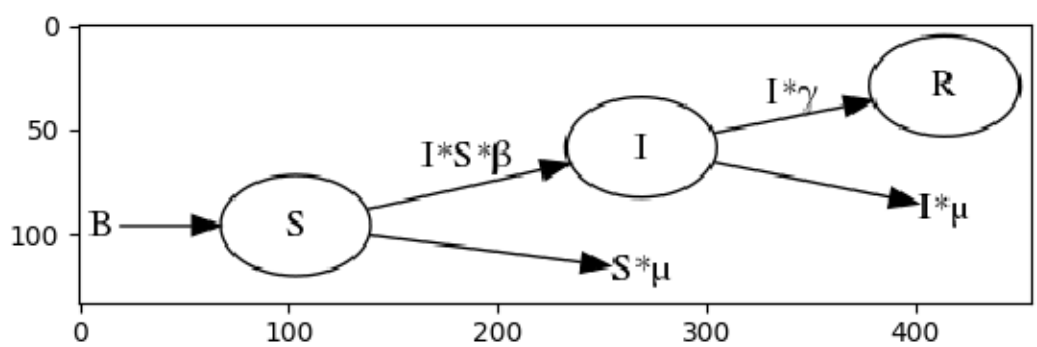
Let's define this in terms of ODEs, and unroll it back to the individual processes.

```
In [1]: from pygom import Transition, TransitionType, SimulateOde, common_models

In [2]: import matplotlib.pyplot as plt

In [3]: stateList = ['S', 'I', 'R']

In [4]: paramList = ['beta', 'gamma', 'B', 'mu']
```

(continues on next page)

(continued from previous page)

```
In [5]: odeList = [
   ...:            Transition(origin='S',
   ...:                       equation='-beta*S*I + B - mu*S',
   ...:                       transition_type=TransitionType.ODE),
   ...:            Transition(origin='I',
   ...:                       equation='beta*S*I - gamma*I - mu*I',
   ...:                       transition_type=TransitionType.ODE),
   ...:            Transition(origin='R',
   ...:                       equation='gamma*I',
   ...:                       transition_type=TransitionType.ODE)
   ...:            ]
   ...:

In [6]: ode = SimulateOde(stateList, paramList, ode=odeList)

In [7]: ode2 = ode.get_unrolled_obj()

In [8]: f = plt.figure()

In [9]: ode2.get_transition_graph()
Out[9]: <graphviz.dot.Digraph at 0x7f15799813c8>

In [10]: plt.close()
```

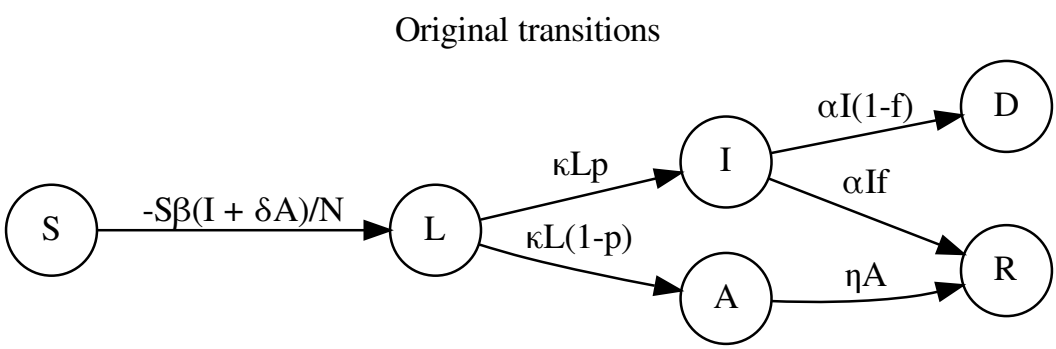


1.4.3 Hard Problem

Now we turn to a harder problem that does not have a one to one mapping between all the transitions and the terms in the ODEs. We use the model in `Influenza_SLIARN()`, defined by

$$
\begin{aligned}
\frac{dS}{dt} &= -S\beta(I + \delta A) \\
\frac{dL}{dt} &= S\beta(I + \delta A) - \kappa L \\
\frac{dI}{dt} &= p\kappa L - \alpha I \\
\frac{dA}{dt} &= (1 - p)\kappa L - \eta A \\
\frac{dR}{dt} &= f\alpha I + \eta A \\
\frac{dN}{dt} &= -(1 - f)\alpha I.
\end{aligned}
$$

The outflow of state **L**, κL, is composed of two transitions, one to **I** and the other to **A** but the ode of **L** only reflects the total flow going out of the state. Same can be said for state **I**, where the flow αI goes to both **R** and **N**. Graphically, it is a rather simple process as shown below.



We slightly change the model by introducing a new state **D** to convert it into a closed system. The combination of state **D** and **N** is a constant, the total population. So we can remove **N** and this new system consist of six transitions. We define them explicitly as ODEs and unroll them into transitions.

```
In [1]: from pygom import SimulateOde, Transition, TransitionType

In [2]: stateList = ['S', 'L', 'I', 'A', 'R', 'D']

In [3]: paramList = ['beta', 'p', 'kappa', 'alpha', 'f', 'delta', 'epsilon', 'N']

In [4]: odeList = [
   ...:             Transition(origin='S', equation='- beta*S/N*(I + delta*A)',␣
↪transition_type=TransitionType.ODE),
   ...:             Transition(origin='L', equation='beta*S/N*(I + delta*A) - kappa*L',
↪ transition_type=TransitionType.ODE),
   ...:             Transition(origin='I', equation='p*kappa*L - alpha*I', transition_
↪type=TransitionType.ODE),
```

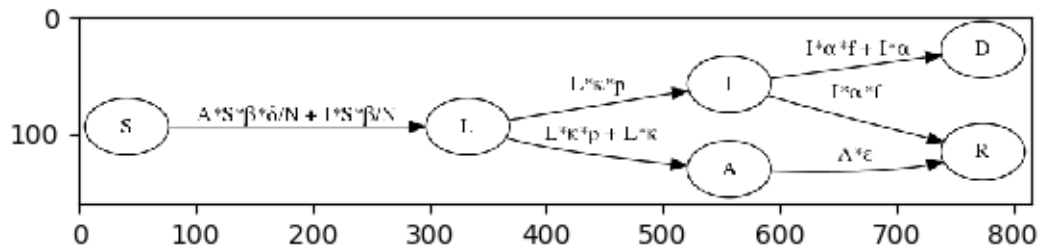
(continues on next page)

(continued from previous page)

```
   ...:             Transition(origin='A', equation='(1 - p)*kappa * L - epsilon*A',␣
↪transition_type=TransitionType.ODE),
   ...:             Transition(origin='R', equation='f*alpha*I + epsilon*A',␣
↪transition_type=TransitionType.ODE),
   ...:             Transition(origin='D', equation='(1 - f)*alpha*I', transition_
↪type=TransitionType.ODE) ]
   ...:

In [5]: ode = SimulateOde(stateList, paramList, ode=odeList)

In [6]: ode.get_transition_matrix()
Out[6]:
Matrix([
[0, 0, 0, 0, 0, 0],
[0, 0, 0, 0, 0, 0],
[0, 0, 0, 0, 0, 0],
[0, 0, 0, 0, 0, 0],
[0, 0, 0, 0, 0, 0],
[0, 0, 0, 0, 0, 0]])

In [7]: ode2 = ode.get_unrolled_obj()

In [8]: ode2.get_transition_matrix()
Out[8]:
Matrix([
[0, A*S*beta*delta/N + I*S*beta/N,         0,                    0,         0,       ␣
↪            0],
[0,                              0, L*kappa*p, -L*kappa*p + L*kappa,         0,       ␣
↪            0],
[0,                              0,         0,                    0, I*alpha*f, -
↪I*alpha*f + I*alpha],
[0,                              0,         0,                    0, A*epsilon,       ␣
↪            0],
[0,                              0,         0,                    0,         0,       ␣
↪            0],
[0,                              0,         0,                    0,         0,       ␣
↪            0]])

In [9]: ode2.get_ode_eqn()
Out[9]:
Matrix([
[         -A*S*beta*delta/N - I*S*beta/N],
[A*S*beta*delta/N + I*S*beta/N - L*kappa],
[                   -I*alpha + L*kappa*p],
[       -A*epsilon - L*kappa*p + L*kappa],
[                  A*epsilon + I*alpha*f],
[                   -I*alpha*f + I*alpha]])
```

After unrolling the odes, we have the following transition graph

```
In [10]: ode2.get_transition_graph()
Out[10]: <graphviz.dot.Digraph at 0x7f1578ad7400>

In [11]: plt.close()

In [12]: print(sum(ode.get_ode_eqn() - ode2.get_ode_eqn()).simplify()) # difference
0
```



which is exactly the same apart from slightly weird arrangement of symbols in some of the equations. The last line with a value of zero also reaffirms the result.

1.5 Solving Boundary Value Problems

In addition to finding solutions for an IVP and estimate the unknown parameters, this package also allows you to solve BVP with a little bit of imagination. Here, we are going to show how a BVP can be solved by treating it as a parameter estimation problem. Essentially, a shooting method where the first boundary condition defines the initial condition of an IVP and the second boundary condition is an observation. Two examples, both from MATLAB[1], will be shown here.

1.5.1 Simple model 1

We are trying to find the solution to the second order differential equation

$$\nabla^2 y + |y| = 0$$

[1] http://uk.mathworks.com/help/matlab/ref/bvp4c.html

subject to the boundary conditions $y(0) = 0$ and $y(4) = -2$. Convert this into a set of first order ODE

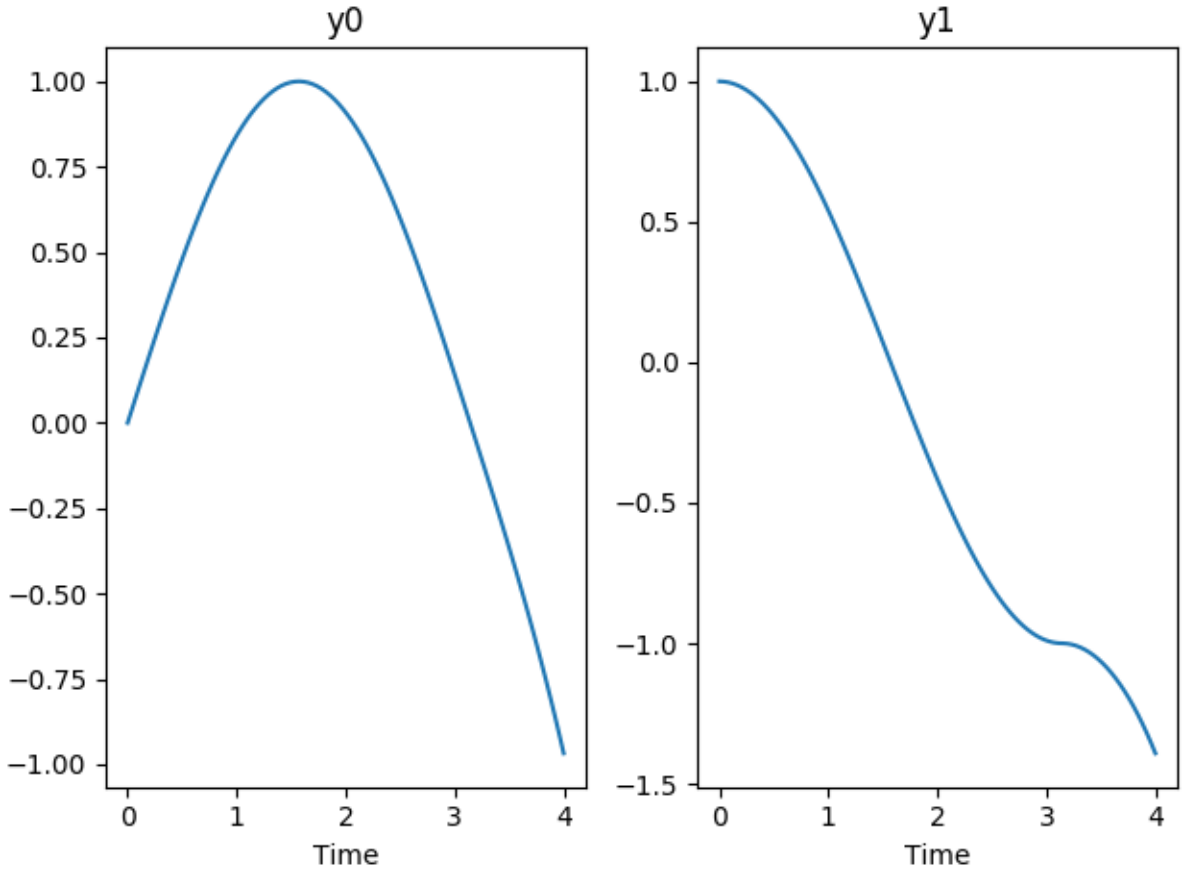
$$\frac{dy_0}{dt} = y_1$$
$$\frac{dy_1}{dt} = -|y_0|$$

using an auxiliary variable $y_1 = \nabla y$ and $y_0 = y$. Setting up the system below

```
In [1]: from pygom import Transition, TransitionType, DeterministicOde, SquareLoss

In [2]: import matplotlib.pyplot as plt

In [3]: stateList = ['y0', 'y1']

In [4]: paramList = []

In [5]: ode1 = Transition(origin='y0',
   ...:                   equation='y1',
   ...:                   transition_type=TransitionType.ODE)
   ...:

In [6]: ode2 = Transition(origin='y1',
   ...:                   equation='-abs(y0)',
   ...:                   transition_type=TransitionType.ODE)
   ...:

In [7]: model = DeterministicOde(stateList,
   ...:                          paramList,
   ...:                          ode=[ode1, ode2])
   ...:

In [8]: model.get_ode_eqn()
Out[8]:
Matrix([
[      y1],
[-Abs(y0)]])
```

We check that the equations are correct before proceeding to set up our loss function.

```
In [9]: import numpy

In [10]: from scipy.optimize import minimize

In [11]: initialState = [0.0, 1.0]

In [12]: t = numpy.linspace(0, 4, 100)

In [13]: model.initial_values = (initialState, t[0])

In [14]: solution = model.integrate(t[1::])

In [15]: f = plt.figure()

In [16]: model.plot()

In [17]: plt.close()
```



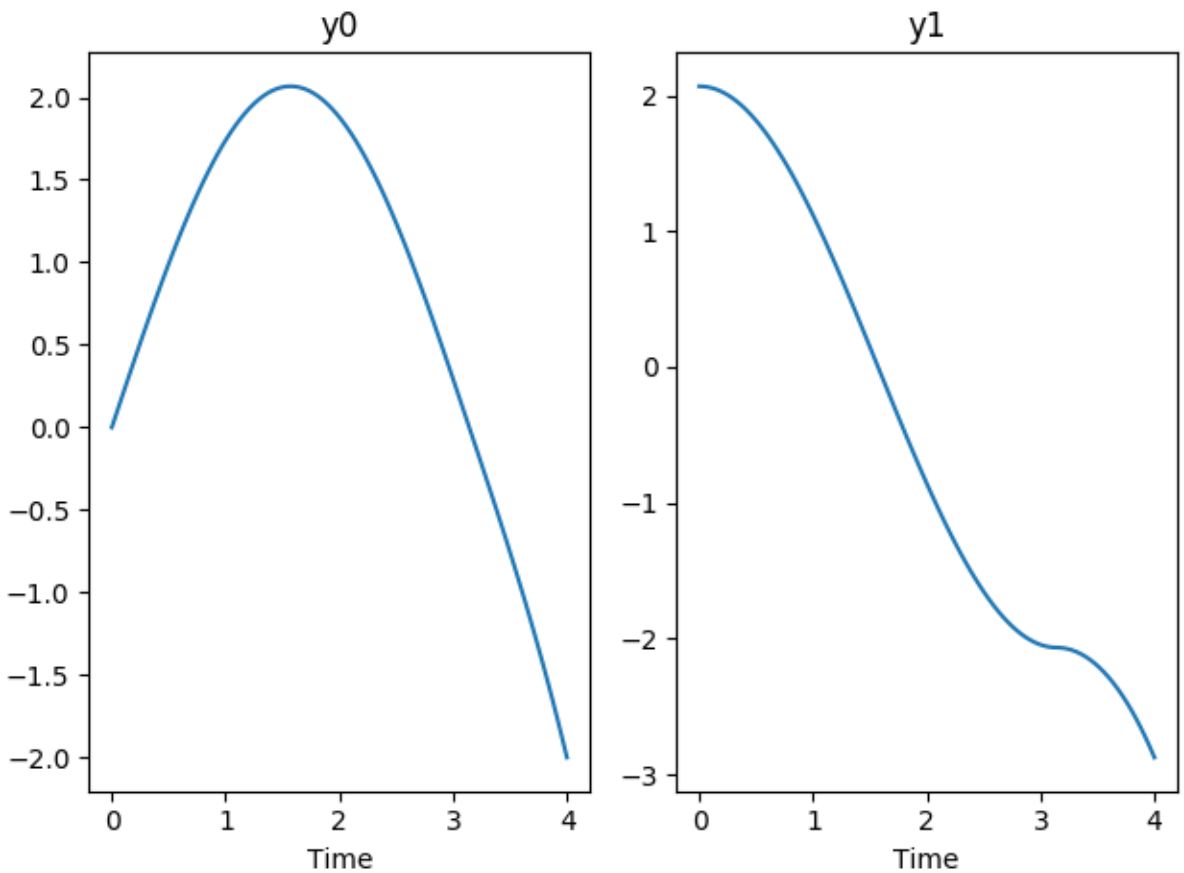
Setting up the second boundary condition $y(4) = -2$ is easy, because that is just a single observation attached to the state y_1. Enforcing the first boundary condition requires us to set it as the initial condition. Because the condition only states that $y(0) = 0$, the starting value of the other state y_1 is free. We let our loss object know that it is free through the targetState input argument.

```
In [18]: theta = [0.0]

In [19]: obj = SquareLoss(theta=theta,
   ....:                  ode=model,
   ....:                  x0=initialState,
   ....:                  t0=t[0],
   ....:                  t=t[-1],
   ....:                  y=[-2],
   ....:                  state_name=['y0'],
   ....:                  target_state=['y1'])
   ....:

In [20]: thetaHat = minimize(fun=obj.costIV, x0=[0.0])

In [21]: print(thetaHat)
      fun: 1.8896300643324946e-11
 hess_inv: array([[0.51403884]])
      jac: array([-8.6096985e-06])
  message: 'Optimization terminated successfully.'
     nfev: 18
```

(continues on next page)

(continued from previous page)

```
      nit: 5
     njev: 6
   status: 0
  success: True
        x: array([2.06657723])

In [22]: model.initial_values = ([0.0] + thetaHat['x'].tolist(), t[0])

In [23]: solution = model.integrate(t[1::])

In [24]: f = plt.figure()

In [25]: model.plot()

In [26]: plt.close()
```



We are going to visualize the solution, and also check the boundary condition. The first became our initial condition, so it is always satisfied and only the latter is of concern, which is zero (subject to numerical error) from thetaHat.

1.5.2 Simple model 2

Our second example is different as it involves an actual parameter and also time. We have the Mathieu's Equation

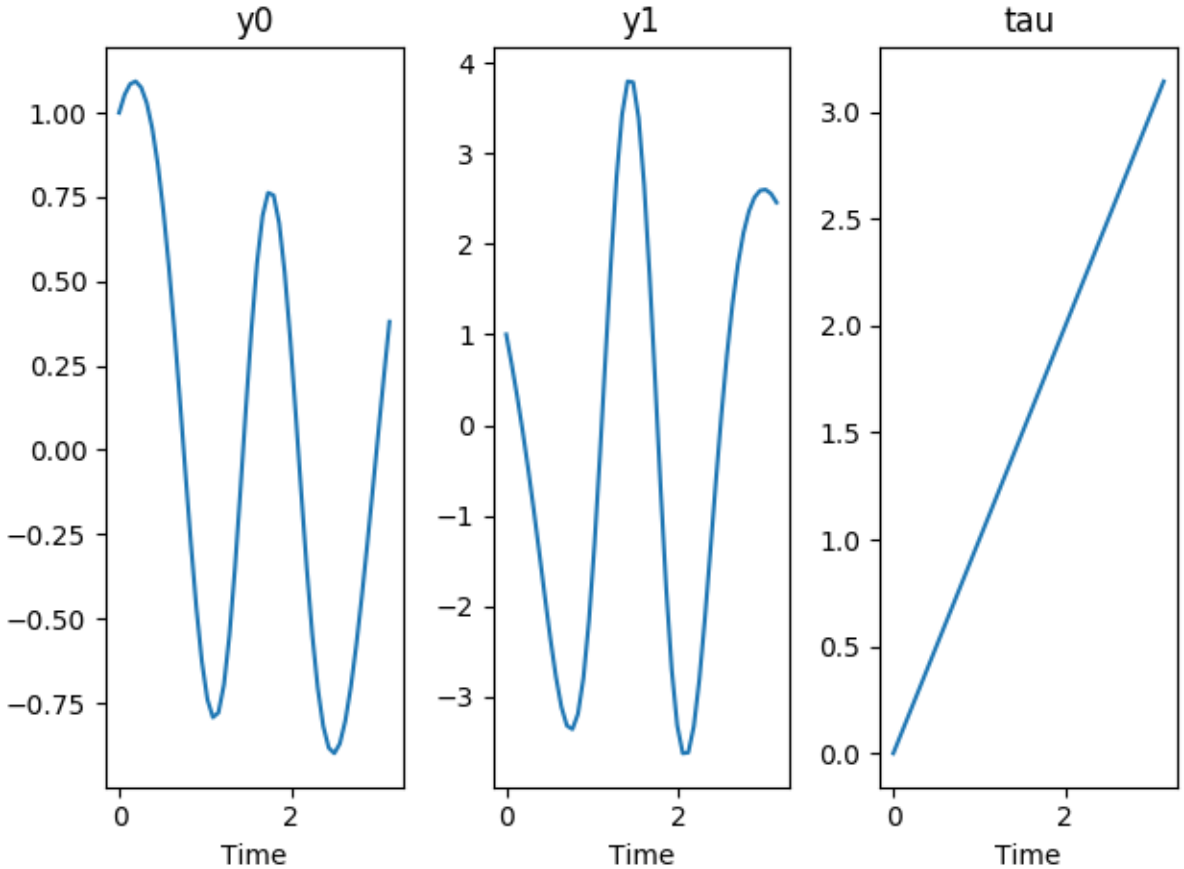
$$\nabla^2 y + (p - 2q\cos(2x))\, y = 0$$

and the aim is to compute the fourth eigenvalue $q = 5$. There are three boundary conditions

$$\nabla y(0) = 0, \quad \nabla y(\pi) = 0, \quad y(0) = 1$$

and we aim to solve it by converting it to a first order ODE and tackle it as an IVP. As our model object does not allow the use of the time component in the equations, we introduce a anxiliary state τ that replaces time t. Rewrite the equations using $y_0 = y, y_1 = \nabla y$ and define our model as

```
In [27]: stateList = ['y0', 'y1', 'tau']

In [28]: paramList = ['p']

In [29]: ode1 = Transition('y0', 'y1', TransitionType.ODE)

In [30]: ode2 = Transition('y1', '-(p - 2*5*cos(2*tau))*y0', TransitionType.ODE)

In [31]: ode3 = Transition('tau', '1', TransitionType.ODE)

In [32]: model = DeterministicOde(stateList, paramList, ode=[ode1, ode2, ode3])

In [33]: theta = [1.0, 1.0, 0.0]

In [34]: p = 15.0

In [35]: t = numpy.linspace(0, numpy.pi)

In [36]: model.parameters = [('p',p)]

In [37]: model.initial_values = (theta, t[0])

In [38]: solution = model.integrate(t[1::])

In [39]: f = plt.figure()

In [40]: model.plot()

In [41]: plt.close()
```



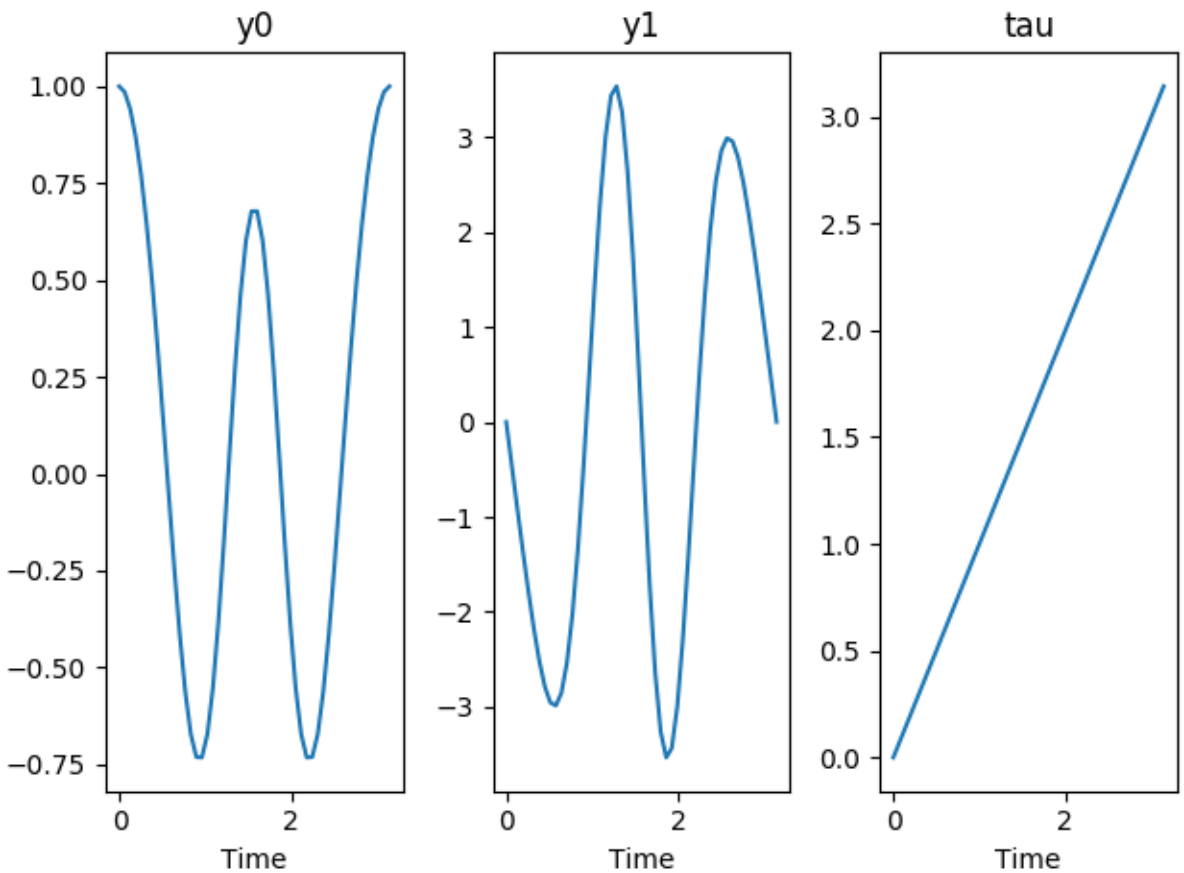
Now we are ready to setup the estimation. Like before, we setup the second boundary condition by pretending that it is an observation. We have all the initial conditions defined by the first boundary condition

```
In [42]: obj = SquareLoss(15.0, model, x0=[1.0, 0.0, 0.0], t0=0.0, t=numpy.pi, y=0.0,␣
↪state_name='y1')

In [43]: xhatObj = minimize(obj.cost,[15])

In [44]: print(xhatObj)
      fun: 8.620622377669915e-17
 hess_inv: array([[0.41085766]])
      jac: array([-2.35234683e-09])
  message: 'Optimization terminated successfully.'
     nfev: 27
      nit: 5
     njev: 9
   status: 0
  success: True
        x: array([17.09658168])

In [45]: model.parameters = [('p', xhatObj['x'][0])]

In [46]: model.initial_values = ([1.0, 0.0, 0.0], t[0])

In [47]: solution = model.integrate(t[1::])
```

(continues on next page)

(continued from previous page)

```
In [48]: f = plt.figure()

In [49]: model.plot()

In [50]: plt.close()
```



The plot of the solution shows the path that satisfies all boundary condition. The last subplot is time which obvious is redundant here but the `DeterministicOde.plot()` method is not yet able to recognize the time component. Possible speed up can be achieved through the use of derivative information or via root finding method that tackles the gradient directly, instead of the cost function.

Reference

1.6 Example: Parameter Estimation 1

1.6.1 Estimation under square loss

To ease the estimation process when given data, a separate module `ode_loss` has been constructed for observations coming from a single state. We demonstrate how to do it via two examples, first, a standard SIR model, then the Legrand SEIHFR model from [Legrand2007] used for Ebola in estimate2.

1.6.1.1 SIR Model

We set up an SIR model as seen previously in sir.

```
In [1]: from pygom import SquareLoss, common_models

In [2]: import numpy

In [3]: import scipy.integrate

In [4]: import matplotlib.pyplot

In [5]: # Again, standard SIR model with 2 parameter.  See the first script!

In [6]: # define the parameters

In [7]: paramEval = [('beta',0.5), ('gamma',1.0/3.0)]

In [8]: # initialize the model

In [9]: ode = common_models.SIR(paramEval)
```

and we assume that we have perfect information about the R compartment.

```
In [10]: x0 = [1, 1.27e-6, 0]

In [11]: # Time, including the initial time t0 at t=0

In [12]: t = numpy.linspace(0, 150, 1000)

In [13]: # Standard.  Find the solution.

In [14]: solution = scipy.integrate.odeint(ode.ode, x0, t)

In [15]: y = solution[:,1:3].copy()
```

Initialize the class with some initial guess

```
In [16]: # our initial guess

In [17]: theta = [0.2, 0.2]

In [18]: objSIR = SquareLoss(theta, ode, x0, t[0], t[1::], y[1::,:], ['I','R'])
```

Note that we need to provide the initial values, x_0 and t_0 differently to the observations y and the corresponding time t. Additionally, the state which the observation lies needs to be specified. Either a single state, or multiple states are allowed, as seen above.

1.6.1.2 Difference in gradient

We have provided two different ways of obtaining the gradient, these are explained in gradient in a bit more detail. First, lets see how similar the output of the two methods are

```
In [19]: objSIR.sensitivity()
Out[19]: array([-0.95621274,  0.87448359])
```

(continues on next page)

(continued from previous page)

```
In [20]: objSIR.adjoint()
Out[20]: array([-0.95498053,  0.87325191])
```

and the time required to obtain the gradient for the SIR model under $\theta = (0.2, 0.2)$, previously entered.

```
In [21]: %timeit objSIR.sensitivity()
17.9 ms +- 517 us per loop (mean +- std. dev. of 7 runs, 100 loops each)

In [22]: %timeit objSIR.adjoint()
648 ms +- 17.5 ms per loop (mean +- std. dev. of 7 runs, 1 loop each)
```

Obviously, the amount of time taken for both method is dependent on the number of observations as well as the number of states. The effect on the adjoint method as the number of observations differs can be quite evident. This is because the adjoint method is under a discretization which loops in Python where as the forward sensitivity equations are solved simply via an integration. As the number of observation gets larger, the affect of the Python loop becomes more obvious.

Difference in gradient is larger when there are less observations. This is because the adjoint method use interpolations on the output of the ode between each consecutive time points. Given solution over the same length of time, fewer discretization naturally leads to a less accurate interpolation. Note that the interpolation is currently performed using univaraite spline, due to the limitation of python packages. Ideally, one would prefer to use an (adaptive) Hermite or Chebyshev interpolation. Note how we ran the two gradient functions once before timing it, that is because we only find the properties (Jacobian, gradient) of the ode during runtime.

1.6.1.3 Optimized result

Then standard optimization procedures with some suitable initial guess should yield the correct result. It is important to set the boundaries for compartmental models as we know that all the parameters are strictly positive. We put a less restrictive inequality here for demonstration purpose.

```
In [23]: # what we think the bounds are

In [24]: boxBounds = [(0.0,2.0),(0.0,2.0)]
```

Then using the optimization routines in `scipy.optimize`, for example, the *SLSQP* method with the gradient obtained by forward sensitivity.

```
In [25]: from scipy.optimize import minimize

In [26]: res = minimize(fun=objSIR.cost,
   ....:                jac=objSIR.sensitivity,
   ....:                x0=theta,
   ....:                bounds=boxBounds,
   ....:                method='SLSQP')
   ....:

In [27]: print(res)
     fun: 6.107148148593866e-07
     jac: array([ 0.16105372, -0.17475277])
 message: 'Optimization terminated successfully.'
    nfev: 18
     nit: 10
    njev: 10
  status: 0
```

(continues on next page)

(continued from previous page)

```
 success: True
       x: array([0.5000679 , 0.33337421])
```

Other methods available in `scipy.optimize.minimize()` can also be used, such as the *L-BFGS-B* and *TNC*. We can also use methods that accepts the exact Hessian such as *trust-ncg* but that should not be necessary most of the time.

1.7 Pre-defined Example common_models

We have defined a set of models `common_models`, most of them commonly used in epidemiology. They are there as examples and also save time for end users. Most of them are of the compartmental type, and we use standard naming conventions i.e. **S** = Susceptible, **E** = Exposed, **I** = Infectious, **R** = Recovered. Extra state symbol will be introduced when required.

CHAPTER 2

Code Documentation and FAQ

2.1 Frequent asked questions

2.1.1 Code runs slowly

This is because the package is not optimized for speed. Although the some of the main functions are lambdified using `sympy` or compiled against `cython` when available, there are many more optimization that can be done. One example is the lines:

in `DeterministicOde.evalSensitivity()`. The first two operations can be inlined into the third and the third line itself can be rewritten as:

and save the explicit copy operation by `numpy` when making A. If desired, we could have also made used of the `numexpr` package that provides further speed up on elementwise operations in place of numpy.

2.1.2 Why not compile the numeric computation form sympy against Theano

Setup of the package has been simplified as much as possible. If you look closely enough, you will realize that the current code generation only uses `cython` and not `f2py`. This is because we are not prepared to do all the system checks, i.e. does a fortran compiler exist, is gcc installed, was python built as a shared library etc. We are very much aware of the benefit, especially considering the possibility of GPU computation in `theano`.

2.1.3 Why not use mpmath library throughout?

This is because we have a fair number of operations that depends on `scipy`. Obviously, we can solve ode using `mpmath` and do standard linear algebra. Unfortunately, optimization and statistics packages and routine are mostly based on `numpy`.

2.1.4 Computing the gradient using `SquareLoss` is slow

It will always be slow on the first operation. This is due to the design where the initialization of the class is fast and only find derivative information/compile function during runtime. After the first calculation, things should be significantly faster.

Why some of my code is not a fortran object?

When we detec either a $\exp$ or a $\log$ in the equations, we automatically force the compile to use mpmath to ensure that we obtain the highest precision. To turn this on/off will be considered as a feature in the future.

2.1.5 Can you not convert a non-autonumous system to an autonomous system for me automatically

Although we can do that, it is not, and will not be implemented. This is to ensure that the end user such as yourself are fully aware of the equations being defined.

2.1.6 Getting the sensitivities from `SquareLoss` did not get a speed up when I used a restricted set of parameters

This is because we currently evaluate the full set of sensitivities before extracting them out. Speeding this up for a restrictive set is being considered. A main reason that stopped us from implementing is that we find the symbolic gradient of the ode before compiling it. Which means that one function call to the compiled file will return the full set of sensitivities and we would only be extracting the appropriate elements from the matrix. This only amounts to a small speed up. The best method would be to compile only the necessary elements of the gradient matrix, but this would require much more work both within the code, and later on when variables are being added/deleted as all these compilation are perfromed in runtime.

2.1.7 Why do not have the option to obtain gradient via complex differencing

It is currently not implemented. Feature under consideration.

2.2 Code documentations

2.2.1 model

2.2.1.1 common_models

A set of commonly used models

`pygom.model.common_models.`**`FitzHugh`**(*param=None*)

The standard FitzHugh model without external input [FitzHugh1961]

$$\frac{dV}{dt} = c(V - \frac{V^3}{3} + R)$$
$$\frac{dR}{dt} = -\frac{1}{c}(V - a + bR).$$

Examples

```
>>> import numpy as np
>>> from pygom import common_models
>>> ode = common_models.FitzHugh({'a':0.2, 'b':0.2, 'c':3.0})
>>> t = np.linspace(0, 20, 101)
>>> x0 = [1.0, -1.0]
>>> ode.initial_values = (x0, t[0])
>>> solution = ode.integrate(t[1::])
>>> ode.plot()
```

pygom.model.common_models.**Influenza_SLIARN**(*param=None*)

A simple influenza model from [Brauer2008], page 323.

$$\frac{dS}{dt} = -S\beta(I + \delta A)$$
$$\frac{dL}{dt} = S\beta(I + \delta A) - \kappa L$$
$$\frac{dI}{dt} = p\kappa L - \alpha I$$
$$\frac{dA}{dt} = (1 - p)\kappa L - \eta A$$
$$\frac{dR}{dt} = f\alpha I + \eta A$$
$$\frac{dN}{dt} = -(1 - f)\alpha I$$

pygom.model.common_models.**Legrand_Ebola_SEIHFR**(*param=None*)

The Legrand Ebola model [Legrand2007] with 6 compartments that includes the H = hospitalization and F = funeral state. Note that because this is an non-autonomous system, there are in fact a total of 7 states after conversion. The set of equations that describes the model are

$$\frac{dS}{dt} = -(\beta_I SI + \beta_H SH + \beta_F SF)$$
$$\frac{dE}{dt} = (\beta_I SI + \beta_H SH + \beta_F SF) - \alpha E$$
$$\frac{dI}{dt} = \alpha E - (\gamma_H \theta_1 + \gamma_I(1 - \theta_1)(1 - \delta_1) + \gamma_D(1 - \theta_1)\delta_1)I$$
$$\frac{dH}{dt} = \gamma_H \theta_1 I - (\gamma_{DH}\delta_2 + \gamma_{IH}(1 - \delta_2))H$$
$$\frac{dF}{dt} = \gamma_D(1 - \theta_1)\delta_1 I + \gamma_{DH}\delta_2 H - \gamma_F F$$
$$\frac{dR}{dt} = \gamma_I(1 - \theta_1)(1 - \delta_1)I + \gamma_{IH}(1 - \delta_2)H + \gamma_F F.$$

Examples

```
>>> import numpy as np
>>> from pygom import common_models
>>> x0 = [1.0, 3.0/200000.0, 0.0, 0.0, 0.0, 0.0, 0.0]
>>> t = np.linspace(0, 25, 100)
>>> ode = common_models.Legrand_Ebola_SEIHFR([('beta_I',0.588),('beta_H',0.794),(
↪'beta_F',7.653),('omega_I',10.0/7.0),('omega_D',9.6/7.0),('omega_H',5.0/7.0),(
↪'omega_F',2.0/7.0),('alphaInv',7.0/7.0),('delta',0.81),('theta',0.80),('kappa',
↪300.0),('interventionTime',7.0)])
```

(continues on next page)

(continued from previous page)

```
>>> ode.initial_values = (x0, t[0])
>>> solution = ode.integrate(t[1::])
>>> ode.plot()
```

pygom.model.common_models.**Lorenz**(*param=None*)

Lorenz attractor define by three parameters, β, σ, ρ as per [Lorenz1963].

$$\frac{dx}{dt} = \sigma(y - x)$$
$$\frac{dy}{dt} = x(\rho - z) - y$$
$$\frac{dz}{dt} = xy - \beta z$$

Examples

```
>>> import matplotlib.pyplot as plt
>>> import numpy
>>> from pygom import common_models
>>> t = numpy.linspace(0, 20, 101)
>>> params = {'beta':8.0/3.0, 'sigma':10.0, 'rho':28.0}
>>> ode = common_models.Lorenz(params)
>>> ode.initial_values = ([1., 1., 1.], t[0])
>>> solution = ode.integrate(t[1::])
>>> plt.plot(solution[:,0], solution[:,2])
>>> plt.show()
```

pygom.model.common_models.**Lotka_Volterra**(*param=None*)

Standard Lotka-Volterra model with two states and four parameters [Lotka1920]

$$\frac{dx}{dt} = \alpha x - cxy$$
$$\frac{dy}{dt} = -\delta y + \gamma xy$$

Examples

```
>>> import numpy as np
>>> from pygom import common_models
>>> params = {'alpha':1, 'delta':3, 'c':2, 'gamma':6}
>>> ode = common_models.Lotka_Volterra(params)
>>> ode.initial_values = ([2.0, 6.0], 0)
>>> t = np.linspace(0.1, 100, 10000)
>>> ode.integrate(t)
>>> ode.plot()
```

pygom.model.common_models.**Lotka_Volterra_4State**(*param=None*)

The four state Lotka-Volterra model [Lotka1920]. A common interpretation is that a = Grass, x = rabbits, y =

foxes and b is the death of foxes.

$$\frac{da}{dt} = k_0 a x$$
$$\frac{dx}{dt} = k_0 a x - k_1 x y$$
$$\frac{dy}{dt} = k_1 x y - k_2 y$$
$$\frac{db}{dt} = k_2 y$$

Examples

```
>>> import numpy as np
>>> from pygom import common_models
>>> x0 = [150.0, 10.0, 10.0, 0.0]
>>> t = np.linspace(0, 15, 100)
>>> params = [0.01, 0.1, 1.0]
>>> ode = common_models.Lotka_Volterra_4State(params)
>>> ode.initial_values = (x0, t[0])
>>> ode.integrate(t[1::])
>>> ode.plot()
```

`pygom.model.common_models.`**`Robertson`**(*param=None*)

The so called Robertson problem [Robertson1966], which is a standard example used to test stiff integrator.

$$\frac{dy_1}{dt} = -0.04 y_1 + 1 \cdot 10^4 y_2 y_3$$
$$\frac{dy_2}{dt} = 0.04 y_1 - 1 \cdot 10^4 y_2 y_3 - 3 \cdot 10^7 y_2^2$$
$$\frac{dy_3}{dt} = 3 \cdot 10^7 y_2^2$$

Examples

```
>>> from pygom import common_models
>>> import numpy
>>> t = numpy.append(0, 4*numpy.logspace(-6, 6, 1000))
>>> ode = common_models.Robertson()
>>> ode.initial_values = ([1.0,0.0,0.0], t[0])
>>> solution = ode.integrate(t[1::])
>>> ode.plot() # note that this is not being plotted in the log scale
```

`pygom.model.common_models.`**`SEIR`**(*param=None*)

A standard SEIR model [Brauer2008], defined by the ode

$$\frac{dS}{dt} = -\beta S I$$
$$\frac{dE}{dt} = \beta S I - \alpha E$$
$$\frac{dI}{dt} = \alpha E - \gamma I$$
$$\frac{dR}{dt} = \gamma I$$

See also:

SEIR_Birth_Death()

Examples

```
>>> import numpy as np
>>> from pygom import common_models
>>> ode = common_models.SEIR({'beta':1800, 'gamma':100, 'alpha':35.84})
>>> t = np.linspace(0, 50, 1001)
>>> x0 = [0.0658, 0.0007, 0.0002, 0.0]
>>> ode.initial_values = (x0, t[0])
>>> solution,output = ode.integrate(t[1::], full_output=True)
>>> ode.plot()
```

pygom.model.common_models.**SEIR_Birth_Death**(*param=None*)

A standard SEIR model with birth and death [Aron1984], defined by the ode

$$\frac{dS}{dt} = \mu - \beta SI - \mu S$$
$$\frac{dE}{dt} = \beta SI - (\mu + \alpha)E$$
$$\frac{dI}{dt} = \alpha E - (\mu + \gamma)I$$
$$\frac{dR}{dt} = \gamma I$$

See also:

SEIR()

Examples

Uses the same set of parameters as the examples in `SEIR()` apart from μ which is new.

```
>>> import numpy as np
>>> from pygom import common_models
>>> params = {'beta':1800, 'gamma':100, 'alpha':35.84, 'mu':0.02}
>>> ode = common_models.SEIR_Birth_Death(params)
>>> t = np.linspace(0, 50, 1001)
>>> x0 = [0.0658, 0.0007, 0.0002, 0.0]
>>> ode.initial_values = (x0, t[0])
>>> solution,output = ode.integrate(t[1::], full_output=True)
>>> ode.plot()
```

pygom.model.common_models.**SEIR_Birth_Death_Periodic**(*param=None*)

A SEIR birth death model with periodic contact [Aron1984], defined by the ode

$$\frac{dS}{dt} = \mu - \beta(t)SI - \mu S$$
$$\frac{dE}{dt} = \beta(t)SI - (\mu + \alpha)E$$
$$\frac{dI}{dt} = \alpha E - (\mu + \gamma)I$$
$$\frac{dR}{dt} = \gamma I$$

where

$$\beta(t) = \beta_0(1 + \beta_1 \cos(2\pi t)).$$

An extension of an SEIR birth death model by varying the contact rate through time.

See also:

SEIR(), SEIR_Birth_Death(), SIR_Periodic()

Examples

Uses the same set of parameters as the examples in `SEIR_Birth_Death()` but now we have two beta parameters instead of one.

```
>>> import matplotlib.pyplot as plt
>>> import numpy as np
>>> from pygom import common_models
>>> params = {'beta0':1800, 'beta1':0.2, 'gamma':100, 'alpha':35.84, 'mu':0.02}
>>> ode = common_models.SEIR_Birth_Death_Periodic(params)
>>> t = np.linspace(0, 50, 1001)
>>> x0 = [0.0658, 0.0007, 0.0002, 0.0]
>>> ode.initial_values = (x0, t[0])
>>> solution,output = ode.integrate(t[1::], full_output=True)
>>> ode.plot()
>>> plt.plot(np.log(solution[:,0]), np.log(solution[:,1]))
>>> plt.show()
>>> plt.plot(np.log(solution[:,0]), np.log(solution[:,2]))
>>> plt.show()
```

`pygom.model.common_models.`**`SEIR_Multiple`**(*n=2*, *param=None*)

An SEIR model that describe spatial heterogeneity [Brauer2008], page 180. The model originated from [Lloyd1996] and notations used here follows [Brauer2008].

$$\begin{aligned}
\frac{dS_i}{dt} &= dN_i - dS_i - \lambda_i S_i \\
\frac{dE_i}{dt} &= \lambda_i S_i - (d + \epsilon) E_i \\
\frac{dI_i}{dt} &= \epsilon E_i - (d + \gamma) I_i \\
\frac{dR_i}{dt} &= \gamma I_i - dR_i
\end{aligned}$$

where

$$\lambda_i = \sum_{j=1}^{n} \beta_{i,j} I_j (1\{i \neq j\} p)$$

with n being the number of patch and p the coupled factor.

Examples

Use the initial conditions that were derived from the stationary condition specified in [Brauer2008].

```
>>> import numpy as np
>>> from pygom import common_models
>>> paramEval = {'beta_00':0.0010107, 'beta_01':0.0010107,
>>>              'beta_10':0.0010107, 'beta_11':0.0010107,
>>>              'd':0.02,'epsilon':45.6, 'gamma':73.0,
>>>              'N_0':10**6,'N_1':10**6,'p':0.01}
>>> x0 = [36139.3224081278, 422.560577637822,
>>>       263.883351688369, 963174.233662546]
>>> ode = common_models.SEIR_Multiple()
>>> t = np.linspace(0, 40, 100)
>>> x01 = []
>>> for s in x0:
>>>     x01 += [s]
>>>     x01 += [s]
>>> ode.parameters = paramEval
>>> ode.initial_values = (x01, t[0])
>>> solution, output = ode.integrate(t[1::], full_output=True)
>>> ode.plot()
```

`pygom.model.common_models.SIR(param=None)`

A standard SIR model as per [Brauer2008]

$$\frac{dS}{dt} = -\beta SI$$
$$\frac{dI}{dt} = \beta SI - \gamma I$$
$$\frac{dR}{dt} = \gamma I$$

Examples

The model that produced top two graph in Figure 1.3 of the reference above. First, when everyone is susceptible and only one individual was infected.

```
>>> import numpy as np
>>> from pygom import common_models
>>> ode = common_models.SIR({'beta':3.6, 'gamma':0.2})
>>> t = np.linspace(0, 730, 1001)
>>> N = 7781984.0
>>> x0 = [1.0, 10/N, 0.0]
>>> ode.initial_values = (x0, t[0])
>>> solution = ode.integrate(t[1::])
>>> ode.plot()
```

Second model with a more *realistic* scenario

```
>>> import numpy as np
>>> from pygom import common_models
>>> ode = common_models.SIR({'beta':3.6, 'gamma':0.2})
>>> t = np.linspace(0, 730, 1001)
>>> N = 7781984.0
>>> x0 = [0.065, 123*(5.0/30.0)/N, 0.0]
>>> ode.initial_values = (x0, t[0])
>>> solution = ode.integrate(t[1::])
>>> ode.plot()
```

`pygom.model.common_models.`**`SIR_Birth_Death`**(*param=None*)

Extension of the standard SIR model [Brauer2008] to also include birth and death

$$\frac{dS}{dt} = B - \beta SI - \mu S$$
$$\frac{dI}{dt} = \beta SI - \gamma I - \mu I$$
$$\frac{dR}{dt} = \gamma I$$

See also:

`SIR()`

Examples

The model that produced bottom graph in Figure 1.3 of the reference above.

```
>>> import numpy as np
>>> from pygom import common_models
>>> B = 126372.0/365.0
>>> N = 7781984.0
>>> params = {'beta':3.6, 'gamma':0.2, 'B':B/N, 'mu':B/N}
>>> ode = common_models.SIR_Birth_Death(params)
>>> t = np.linspace(0, 35*365, 10001)
>>> x0 = [0.065, 123.0*(5.0/30.0)/N, 0.0]
>>> ode.initial_values = (x0, t[0])
>>> solution,output = ode.integrate(t[1::], full_output=True)
>>> ode.plot()
```

`pygom.model.common_models.`**`SIR_N`**(*param=None*)

A standard SIR model [Brauer2008] with population N. This is the unnormalized version of the SIR model.

$$\frac{dS}{dt} = -\beta SI/N$$
$$\frac{dI}{dt} = \beta SI/N - \gamma I$$
$$\frac{dR}{dt} = \gamma I$$

Examples

The model that produced top two graph in Figure 1.3 of the reference above. First, when everyone is susceptible and only one individual was infected.

```
>>> import numpy as np
>>> from pygom import common_models
>>> ode = common_models.SIR({'beta':3.6, 'gamma':0.2})
>>> t = np.linspace(0, 730, 1001)
>>> N = 7781984.0
>>> x0 = [N, 1.0, 0.0]
>>> ode.initial_values = (x0, t[0])
>>> solution = ode.integrate(t[1::])
>>> ode.plot()
```

Second model with a more *realistic* scenario

```
>>> import numpy as np
>>> from pygom import common_models
>>> ode = common_models.SIR({'beta':3.6, 'gamma':0.2})
>>> t = np.linspace(0, 730, 1001)
>>> N = 7781984.0
>>> x0 = [int(0.065*N), 21.0, 0.0]
>>> ode.initial_values = (x0, t[0])
>>> solution = ode.integrate(t[1::])
>>> ode.plot()
```

pygom.model.common_models.**SIS**(*param=None*)

A standard SIS model

$$\frac{dS}{dt} = -\beta SI + \gamma I$$
$$\frac{dI}{dt} = \beta SI - \gamma I$$

Examples

```
>>> import numpy as np
>>> from pygom import common_models
>>> ode = common_models.SIS({'beta':0.5, 'gamma':0.2})
>>> t = np.linspace(0, 20, 101)
>>> x0 = [1.0, 0.1]
>>> ode.initial_values = (x0, t[0])
>>> solution = ode.integrate(t[1::])
>>> ode.plot()
```

pygom.model.common_models.**SIS_Periodic**(*param=None*)

A SIS model with periodic contact, defined by the ode as per [Hethcote1973]

$$\frac{dI}{dt} = (\beta(t)N - \alpha)I - \beta(t)I^2$$

where

$$\beta(t) = 2 - 1.8\cos(5t).$$

As the name suggests, it achieves a (stable) periodic solution.

Examples

```
>>> from pygom import common_models
>>> import numpy as np
>>> ode = common_models.SIS_Periodic({'alpha':1.0})
>>> t = np.linspace(0, 10, 101)
>>> x0 = [0.1, 0.0]
>>> ode.initial_values = (x0, t[0])
>>> solution = ode.integrate(t[1::])
>>> ode.plot()
```

pygom.model.common_models.**vanDelPol**(*param=None*)

The van der Pol equation [vanderpol1926], a second order ode

$$y'' - \mu(1 - y^2)y' + y = 0$$

where $\mu > 0$. This can be converted to a first order ode by equating $x = y'$

$$x' - \mu(1 - y^2)x + y = 0$$

which result in a coupled ode

$$\begin{aligned} x' &= \mu(1 - y^2)x - y \\ y' &= x \end{aligned}$$

and this can be solved via standard method.

Examples

```
>>> from pygom import common_models
>>> import numpy
>>> t = numpy.linspace(0, 20, 1000)
>>> ode = common_models.vanDelPol({'mu':1.0})
>>> ode.initial_values = ([2.0,0.0], t[0])
>>> solution = ode.integrate(t[1::])
>>> ode.plot()
```

2.2.1.2 transition

All classes required to define a transition that is inserted into the ode model

class `pygom.model.transition.`**`Transition`**(*origin*, *equation*, *transition_type='ODE'*, *destination=None*, *ID=None*, *name=None*)

This class carries the information for transitions defined for an ode, which includes the ode itself, a birth death process where only one state is involved and also a transition between two states

Parameters

origin: str Origin state.

equation: str Equation defining the transition

transition_type: enum or str, optional of type `TransitionType` or one of ('ODE', 'T', 'B', 'D') defaults to 'ODE'

destination: str, optional Destination State. If the transition is not between state, such as a birth or death process, then this is is not required. If it is stated as a birth, death or an ode then it throws an error

`destination`

Return the destination state

Returns

string The destination state

`equation`

Return the transition _equation

Returns

string The transition _equation

`is_between_state`()

Return whether it is a transition between two state

Returns

bool True if it is a transition between two state False if it is only related to the origin state

origin

Return the origin state

Returns

string The origin state

transition_type

Return the type of transition

Returns

transition_type One of the four type available from `transition_type`

class `pygom.model.transition.`**TransitionType**

This is an Enum describing the four feasible type of transitions use to define the ode model `BaseOdeModel`

The following four types of transitions are available.

B = Birth process

D = Death process

T = Transition between states

ODE = ODE _equation

2.2.1.3 deterministic

This module is defined such that operation on ode are all gathered in one place. Future extension of operations should be added here

class `pygom.model.deterministic.`**DeterministicOde**(*state=None*, *param=None*, *derived_param=None*, *transition=None*, *birth_death=None*, *ode=None*)

This contains the interface and operation built above the already defined set of ode

Parameters

state: list A list of states (string)

param: list A list of the parameters (string)

derived_param: list A list of the derived parameters (tuple of (string,string))

transition: list A list of transition (`Transition`)

birth_death: list A list of birth or death process (`Transition`)

ode: list A list of ode (`Transition`)

adjoint(*state*, *t*, *state_param*, *func=None*)

Compute the adjoint given the adjoint vector, time, state variable and the objective function. Note that this function is very restrictive in the sense that the (original) state variable changes through time but this assumes it is a constant, i.e. we assume that the original system is linear.

Parameters

state: array like The current value of lambda, where lambda's are the Lagrangian multipliers of the differential equation.

t: double The current time.

state_param: array like The state vector that is (or maybe) required to evaluate the jacobian of the original system

func: callable This should take inputs similar to an ode, i.e. of the form func(y,t). If j(y,t) is the cost function, then func is a function that calculates $\frac{\partial j}{\partial x}$.

Returns

numpy.ndarray output of the same length as the ode

Notes

The size of lambda should be the same as the state. The integral should be starting from T, the final time of the original system and is integrated backwards (for stability).

adjoint_T(*t*, *state*, *state_param*, *func=None*)
Same as `adjoint()` but with t as first parameter

adjoint_interpolate(*state*, *t*, *interpolant*, *func=None*)
Compute the adjoint given the adjoint vector, time, the functions which was used to interpolate the state variable

Parameters

state: array like The current value of lambda, where lambda's are the Lagrangian multipliers of the differential equation.

t: double The current time.

interpolant: list list of interpolating functions of the state

func: callable This should take inputs similar to an ode, i.e. of the form func(y,t). If j(y,t) is the cost function, then func is a function that calculates $\frac{\partial j}{\partial x}$.

Returns

numpy.ndarray output of the same length as the ode

adjoint_interpolate_T(*t*, *state*, *interpolant*, *objInput=None*)
Same as `adjoint_interpolate()` but with t as first parameter

adjoint_interpolate_jacobian(*state*, *t*, *interpolant*, *func=None*)
Compute the Jacobian of the adjoint given the adjoint vector, time, function of the interpolation on the state variables and the objective function. This is simply the same as the negative Jacobian of the ode transposed.

Parameters

state: array like The current value of lambda, where lambda's are the Lagrangian multipliers of the differential equation.

t: double The current time.

interpolant: list list of interpolating functions of the state

func: callable This should take inputs similar to an ode, i.e. of the form func(y,t). If j(y,t) is the cost function, then func is a function that calculates $\frac{\partial j}{\partial x}$.

Returns

numpy.ndarray output of is a two dimensional array of size [number of state x number of state]

See also:

adjoint_jacobian()

Notes

Same as `adjoint_jacobian()` but takes a list of interpolating function instead of a single (vector) value

adjoint_interpolate_jacobian_T(*t*, *state*, *interpolant*, *func=None*)
Same as `adjoint_interpolate_jacobian()` but with t as first parameter

adjoint_jacobian(*state*, *t*, *state_param*, *func=None*)
Compute the jacobian of the adjoint given the adjoint vector, time, state variable and the objective function. This is simply the same as the negative jacobian of the ode transposed.

Parameters

state: array like The current value of lambda, where lambda's are the Lagrangian multipliers of the differential equation.

t: double The current time.

state_param: array like The state vector that is (or maybe) required to evaluate the jacobian of the original system

func: callable This should take inputs similar to an ode, i.e. of the form func(y,t). If j(y,t) is the cost function, then func is a function that calculates $\frac{\partial j}{\partial x}$.

Returns

numpy.ndarray output of is a two dimensional array of size [number of state x number of state]

See also:

adjoint()

Notes

It takes the same number of argument as the adjoint for simplicity when integrating.

adjoint_jacobian_T(*t*, *state*, *state_param*, *func=None*)
Same as `adjoint_jacobian_T()` but with t being the first parameter

diff_jacobian(*state*, *t*)
Evaluate the differential of jacobian given state and time

Parameters

state: array like The current numerical value for the states which can be `numpy.ndarray` or `list`

t: double The current time

Returns

numpy.ndarray Matrix of dimension [number of state x number of state]

diff_jacobian_T(*t*, *state*)
Same as `diff_jacobian()` but with t as first parameter

eval_diff_jacobian(*parameters=None*, *time=None*, *state=None*)

Evaluate the differential of the jacobian given parameters, state and time. An extension of `diff_jacobian()` but now also include the parameters.

Parameters

parameters: list see `parameters()`

time: double The current time

state: array list The current numerical value for the states which can be `numpy.ndarray` or `list`

Returns

`numpy.matrix` or `mpmath.matrix` Matrix of dimension [number of state x number of state]

See also:

`jacobian()`

Notes

Name and order of state and time are also different.

eval_forwardforward(*FF*, *S*, *state*, *t*)

Evaluate a single f(x) of the forward-forward sensitivities

Parameters

FF: array like this is in fact a 3rd order Tensor, aka 3d array

S: array like sensitivities in matrix form

state: array like the current state

t: numeric time

Returns

`numpy.ndarray` f(x) of size [number of state * (number of parameters * number of parameters)]

eval_grad(*parameters=None*, *time=None*, *state=None*)

Evaluate the gradient given parameters, state and time. An extension of `grad()` but now also include the parameters.

Parameters

parameters: list see `parameters()`

time: double The current time

state: array list The current numerical value for the states which can be `numpy.ndarray` or `list`

Returns

`numpy.matrix` or `mpmath.matrix` Matrix of dimension [number of state x number of state]

See also:

`grad()`

Notes

Name and order of state and time are also different.

eval_grad_jacobian (*parameters=None*, *time=None*, *state=None*)

Evaluate the jacobian of the gradient given parameters, state and time. An extension of `grad_jacobian()` but now also include the parameters.

Parameters

parameters: list see `parameters()`

time: double The current time

state: array list The current numerical value for the states which can be `numpy.ndarray` or `list`

Returns

`numpy.matrix` or `mpmath.matrix` Matrix of dimension [number of state x number of state]

See also:

`grad_jacobian()`, `get_grad_jacobian_eqn()`

Notes

Name and order of state and time are also different.

eval_hessian (*parameters=None*, *time=None*, *state=None*)

Evaluate the hessian given parameters, state and time. An extension of `hessian()` but now also include the parameters.

Parameters

parameters: list see `parameters()`

time: double The current time

state: array list The current numerical value for the states which can be `numpy.ndarray` or `list`

Returns

list list of dimension number of state, each with matrix [number of parameters x number of parameters] in `sympy.matricies.matricies`

See also:

`grad()`, `eval_grad()`

eval_jacobian (*parameters=None*, *time=None*, *state=None*)

Evaluate the jacobian given parameters, state and time. An extension of `jacobian()` but now also include the parameters.

Parameters

parameters: list see `parameters()`

time: double The current time

state: array list The current numerical value for the states which can be `numpy.ndarray` or `list`

Returns

numpy.matrix or mpmath.matrix Matrix of dimension [number of state x number of state]

See also:

jacobian()

Notes

Name and order of state and time are also different.

eval_ode(*parameters=None*, *time=None*, *state=None*)

Evaluate the ode given time, state and parameters. An extension of `ode()` but now also include the parameters.

Parameters

parameters: list see `parameters()`

time: numeric The current time

state: array like The current numerical value for the states which can be `numpy.ndarray` or `list`

Returns

numpy.matrix or mpmath.matrix output of the same length as the ode.

See also:

ode()

Notes

There are differences between the output of this function and `ode()`. Name and order of state and time are also different.

eval_sens_jacobian_state(*time=None*, *state=None*, *sens=None*)

Evaluate the jacobian of the sensitivities w.r.t the states given parameters, state and time. An extension of `sens_jacobian_state()` but now also include the parameters.

Parameters

parameters: list see `parameters()`

time: double The current time

state: array list The current numerical value for the states which can be `numpy.ndarray` or `list`

Returns

numpy.matrix or mpmath.matrix Matrix of dimension [number of state x number of state]

See also:

sens_jacobian_state()

Notes

Name and order of state and time are also different.

eval_sensitivity(*S*, *t*, *state*, *by_state=False*)

Evaluate the sensitivity given state and time

Parameters

S: array like Which should be `numpy.ndarray`. The starting sensitivity of size [number of state x number of parameters]. Which are normally zero or one, depending on whether the initial conditions are also variables.

t: double The current time

state: array like The current numerical value for the states which can be `numpy.ndarray` or `list`

by_state: bool how we want the output to be arranged. Default is True so that we have a block diagonal structure

Returns

numpy.ndarray

See also:

sensitivity()

Notes

It is different to `eval_ode()` and `eval_jacobian()` in that the extra input argument is not a parameter

eval_sensitivityIV(*S*, *IV*, *t*, *state*)

Evaluate the sensitivity with initial values given state and time

Parameters

S: array like Which should be `numpy.ndarray`. The starting sensitivity of size [number of state x number of parameters]. Which are normally zero or one, depending on whether the initial conditions are also variables.

IV: array like sensitivities for the initial values

t: double The current time

state: array like The current numerical value for the states which can be `numpy.ndarray` or `list`

Returns

numpy.ndarray $f(s(x, \theta))$ and $f(s(x_0))$

See also:

sensitivityIV()

Notes

It is different to `eval_ode()` and `eval_jacobian()` in that the extra input argument is not a parameter.

forwardforward(*ff*, *t*, *state*, *s*)

Evaluate a single $f(x)$ of the forward-forward sensitivities

Parameters

ff: array like the forward-forward sensitivities in vector form

t: numeric time

state: array like the current state

s: array like forward sensitivities in vector form

Returns

numpy.ndarray $f(x)$ of size [number of state * (number of parameters * number of parameters)]

forwardforward_T(*t*, *ff*, *s*, *state*)

Same as `forwardforward()` but with t as the first parameter

get_diff_jacobian_eqn()

Returns the jacobian differentiate w.r.t. states in algebraic form

Returns

list list of size (num of state,) each with `sympy.matrices.matrices` of dimension [number of state x number of state]

get_grad_eqn()

Return the gradient of the ode in algebraic form

Returns

sympy.matrices.matrices A matrix of dimension [number of state x number of parameters]

get_grad_jacobian_eqn()

Return the jacobian of the gradient in algebraic form

Returns

sympy.matrices.matrices A matrix of dimension [number of state * number of parameters x number of state]

See also:

get_grad_eqn()

get_hessian_eqn()

Return the Hessian of the ode in algebraic form

Returns

list list of dimension number of state, each with matrix [number of parameters x number of parameters] in `sympy.matricies.matricies`

Notes

We deliberately return a list instead of a 3d array of a tensor to avoid confusion

get_jacobian_eqn()

Returns the jacobian in algebraic form

Returns

sympy.matrices.matrices A matrix of dimension [number of state x number of state]

get_ode_eqn(*param_sub=False*)

Find the algebraic equations of the ode system.

Returns

sympy.matrices.matrices ode in matrix form

get_transition_graph(*file_name=None*, *show=True*)

Returns the transition graph using graphviz

Parameters

file_name: str, optional name of the output file, defaults to None

show: bool, optional If the graph should be plotted, defaults to True

Returns

graphviz.Digraph

grad(*state*, *time*)

Evaluate the gradient given state and time

Parameters

state: array like The current numerical value for the states which can be `numpy.ndarray` or `list`

t: numeric The current time

Returns

numpy.ndarray Matrix of dimension [number of state x number of parameters]

grad_T(*t*, *state*)

Same as `grad_T()` but with t as first parameter

grad_jacobian(*state*, *time*)

Evaluate the Jacobian of the gradient given state and time

Parameters

state: array like The current numerical value for the states which can be `numpy.ndarray` or `list`

t: numeric The current time

Returns

numpy.ndarray Matrix of dimension [number of state x number of parameters]

See also:

grad()

grad_jacobianT(*t*, *state*)
Same as `grad_jacobian()` but with t as first parameter

hessian(*state*, *time*)
Evaluate the hessian given state and time

Parameters

state: array like The current numerical value for the states which can be `numpy.ndarray` or `list`

t: double The current time

Returns

list list of dimension number of state, each with matrix [number of parameters x number of parameters] in `sympy.matricies.matricies`

initial_state
Return the initial state values

initial_time
Return the initial time

initial_values
Returns the initial values, both time and state as a tuple (x0, t0)

integrate(*t*, *full_output=False*)
Integrate over a range of t when t is an array and a output at time t

Parameters

t: array like the range of time points which we want to see the result of

full_output: bool if we want additional information

integrate2(*t*, *full_output=False*, *method=None*)
Integrate over a range of t when t is an array and a output at time t. Select a suitable method to integrate when method is None.

Parameters

t: array like the range of time points which we want to see the result of

full_output: bool if we want additional information

method: str, optional the integration method. All those available in `ode` are allowed with 'vode' and 'ivode' representing the non-stiff and stiff version respectively. Defaults to None, which tries to choose the integration method via eigenvalue analysis (only one) using the initial conditions

is_stiff(*state=None*, *t=None*)
Test on the eigenvalues of the jacobian. We classify the problem as stiff if any of the eigenvalues are positive

Parameters

state: array like The current numerical value for the states which can be `numpy.ndarray` or `list`

t: double The current time

Returns

`numpy.ndarray` eigenvalues of the system given input

jacobian(*state*, *t*)

Evaluate the jacobian given state and time

Parameters

state: array like The current numerical value for the states which can be `numpy.ndarray` or `list`

t: double The current time

Returns

`numpy.ndarray` Matrix of dimension [number of state x number of state]

jacobian_T(*t*, *state*)

Same as `jacobian()` but with t as first parameter

jacobian_eigenvalue(*state=None*, *t=None*)

Find out the eigenvalues of the jacobian given state and time. If None is given, the initial values are used.

Parameters

state: array like The current numerical value for the states which can be `numpy.ndarray` or `list`

t: double The current time

Returns

bool True if any eigenvalue is positive

linear_ode()

To check whether the input ode is linear

Returns

bool True if it is linear, False otherwise

ode(*state*, *t*)

Evaluate the ode given state and time

Parameters

state: array like The current numerical value for the states which can be `numpy.ndarray` or `list`

t: double The current time

Returns

`numpy.ndarray` output of the same length as the ode

ode_T(*t*, *state*)

Same as `ode()` but with t as the first parameter

ode_and_forwardforward(*state_param*, *t*)

Evaluate a single f(x) of the ode and the forward-forward sensitivities

Parameters

state_param: array like state and forward-forward sensitivities in vector form

t: numeric time

Returns

`numpy.ndarray` same size as the state_param input

ode_and_forwardforward_T(*t*, *state_param*)
Same as `odeAndForwardForward()` but with time as the first input

ode_and_forwardforward_jacobian(*state_param*, *t*)
Return the jacobian after evaluation given the input of the state and the forward forward sensitivities

Parameters

state_param: array like state and forward-forward sensitivities in vector form

t: numeric time

Returns

numpy.ndarray size of (a,a) where a is the length of the state_param input

ode_and_forwardforward_jacobian_T(*t*, *state_param*)
Same as `ode_and_forwardforward_jacobian()` but with t being the first parameters

ode_and_sensitivity(*state_param*, *t*, *by_state=False*)
Evaluate the sensitivity given state and time

Parameters

state_param: array like The current numerical value for the states as well as the sensitivities values all in one. We assume that the state values comes first.

t: double The current time

by_state: bool Whether the output vector should be arranged by state or by parameters. If False, then it means that the vector of output is arranged according to looping i,j from Sensitivity_{i,j} with i being the state and j the param. This is the preferred way because it leds to a block diagonal Jacobian

Returns

list concatenation of 2 element. First contains the ode, second the sensitivity. Both are of type `numpy.ndarray`

See also:

sensitivity(), ode()

ode_and_sensitivityIV(*state_param*, *t*)
Evaluate the sensitivity given state and time

Parameters

state_param: array like The current numerical value for the states as well as the sensitivities values all in one. We assume that the state values comes first.

t: double The current time

Returns

list concatenation of 3 element. First contains the ode, second the sensitivity, then the sensitivity of the initial value. All of them are of type `numpy.ndarray`

See also:

sensitivity(), ode()

ode_and_sensitivityIV_T(*t*, *state_param*)
Same as `ode_and_sensitivityIV()` but with t as first parameter

ode_and_sensitivityIV_jacobian(*state_param*, *t*)
Evaluate the sensitivity given state and time. Output a block diagonal sparse matrix as default.

Parameters

state_param: array like The current numerical value for the states as well as the sensitivities values all in one. We assume that the state values comes first.

t: double The current time

byState: bool How the output is arranged, according to the vector of output. It can be in terms of state or parameters, where by state means that the jacobian is a block diagonal matrix.

Returns

numpy.ndarray output of a square matrix of size: number of ode + 1 times number of parameters

See also:

ode_and_sensitivity()

ode_and_sensitivityIV_jacobian_T(*t*, *state_param*)
Same as `ode_and_sensitivityIV_jacobian()` but with t as first parameter

ode_and_sensitivity_T(*t*, *state_param*, *by_state=False*)
Same as `ode_and_sensitivity()` but with t as first parameter

ode_and_sensitivity_jacobian(*state_param*, *t*, *by_state=False*)
Evaluate the sensitivity given state and time. Output a block diagonal sparse matrix as default.

Parameters

state_param: array like The current numerical value for the states as well as the sensitivities values all in one. We assume that the state values comes first.

t: double The current time

by_state: bool How the output is arranged, according to the vector of output. It can be in terms of state or parameters, where by state means that the jacobian is a block diagonal matrix.

Returns

numpy.ndarray output of a square matrix of size: number of ode + 1 times number of parameters

See also:

ode_and_sensitivity()

ode_and_sensitivity_jacobian_T(*t*, *state_param*, *by_state=False*)
Same as `ode_and_sensitivity_jacobian()` but with t as first parameter

plot()
Plot the results of the integration

Notes

If we have 3 states or more, it will always be arrange such that it has 3 columns. Uses the operation from `odeutils`

print_ode(*latex_output=False*)

Prints the ode in symbolic form onto the screen/console in actual symbols rather than the word of the symbol.

Parameters

latex_output: bool, optional Defaults to false which prints the equation in terms of symbols, if set to yes then the formula in terms of latex equations will be printed onto the screen.

sens_jacobian_state(*state_param*, *t*)

Evaluate the jacobian of the sensitivity w.r.t. the state given state and time

Parameters

state_param: array like The current numerical value for the states as well as the sensitivities, which can be `numpy.ndarray` or `list`

t: double The current time

Returns

numpy.ndarray Matrix of dimension [number of state * number of parameters x number of state]

sens_jacobian_state_T(*t*, *state*)

Same as `sens_jacobian_state_T()` but with t as first parameter

sensitivity(*sens*, *t*, *state*, *by_state=False*)

Evaluate the sensitivity given state and time. The default is to output the values by parameters, i.e. $s_i, \ldots, s_{i+n}$ are partial derivatives w.r.t. the states for $i \in 1, 1+p, 1+2p, 1+3p, \ldots, 1+(n-1)p$. This is to take advantage of the fact that we have a block diagonal jacobian that was already evaluated

Parameters

sens: array like The starting sensitivity of size [number of state x number of parameters]. Which are normally zero or one, depending on whether the initial conditions are also variables.

t: double The current time

state: array like The current numerical value for the states which can be `numpy.ndarray` or `list`

by_state: bool how we want the output to be arranged. Default is True so that we have a block diagonal structure

Returns

numpy.ndarray

sensitivityIV(*sensIV*, *t*, *state*)

Evaluate the sensitivity which include the initial values as our parameters given state and time. The default is to output the values by parameters, i.e. $s_i, \ldots, s_{i+n}$ are partial derivatives w.r.t. the states for $i \in 1, 1+p, 1+2p, 1+3p, \ldots, 1+(n-1)p$. This is to take advantage of the fact that we have a block diagonal Jacobian that was already evaluated.

Parameters

sensIV: array like The starting sensitivity of size [number of state x number of parameters] + [number of state x number of state] for the initial condition. The latter is an identity matrix at time zero.

t: double The current time

state: array like The current numerical value for the states which can be `numpy.ndarray` or `list`

Returns

`numpy.ndarray` output of the same length as the ode

`sensitivityIV_T`(*t*, *sensIV*, *state*)
Same as `sensitivityIV()` but with t as first parameter

`sensitivity_T`(*t*, *sens*, *state*, *by_state=False*)
Same as `sensitivity()` but with t as first parameter

2.2.1.4 stochastic

Module/class that carries out different type of simulation on an ode formulation

class `pygom.model.simulate.`**`SimulateOde`**(*state=None*, *param=None*, *derived_param=None*, *transition=None*, *birth_death=None*, *ode=None*)
This builds on top of `DeterministicOde` which we simulate the outcome instead of solving it deterministically

Parameters

state: list A list of states (string)

param: list A list of the parameters (string)

derived_param: list A list of the derived parameters (tuple of (string,string))

transition: list A list of transition (`Transition`)

birth_death: list A list of birth or death process (`Transition`)

ode: list A list of ode (`Transition`)

`birth_death_rate`(*state*, *t*)
Evaluate the birth death rates given state and time

Parameters

state: array like The current numerical value for the states which can be `np.ndarray` or `list`

t: double The current time

Returns

`numpy.ndarray` an array of size (M,M) where M is the number of birth and death actions

`cle`(*x0*, *t0*, *t1*, *output_time=False*)
Stochastic simulation using the CLE approximation starting from time t0 to t1 with the starting state values of x0. The CLE approximation is performed using a simple Euler-Maruyama method with step size h. We assume that the input parameter transition_func provides $f(x,t)$ while the CLE is defined as $dx = x + V * h * f(x,t) + \sqrt(f(x,t)) * Z * \sqrt(h)$ with Z being standard normal random variables.

Parameters

x: array like state vector

t0: double start time

t1: double final time

eval_birth_death_rate (*parameters=None*, *time=None*, *state=None*)
Evaluate the birth and death rates given parameters, state and time.

Parameters

parameters: list see `setParameters()`

time: double The current time

state: array list The current numerical value for the states which can be `numpy.ndarray` or `list`

Returns

`numpy.matrix` or `mpmath.matrix` Matrix of dimension [number of birth and death rates x 1]

eval_transition_matrix (*parameters=None*, *time=None*, *state=None*)
Evaluate the transition matrix given parameters, state and time. Note that the output is not in sparse format

Parameters

parameters: list see `setParameters()`

time: double The current time

state: array list The current numerical value for the states which can `numpy.ndarray` or `list`

Returns

`numpy.matrix` or `mpmath.matrix` Matrix of dimension [number of state x number of state]

eval_transition_mean (*parameters=None*, *time=None*, *state=None*)
Evaluate the transition mean given parameters, state and time.

Parameters

parameters: list see `setParameters()`

time: double The current time

state: array list The current numerical value for the states which can be `numpy.ndarray` or `list`

Returns

`numpy.matrix` or `mpmath.matrix` Matrix of dimension [number of state x number of state]

eval_transition_var (*parameters=None*, *time=None*, *state=None*)
Evaluate the transition variance given parameters, state and time.

Parameters

parameters: list see `setParameters()`

time: double The current time

state: array list The current numerical value for the states which can be `numpy.ndarray` or `list`

Returns

`numpy.matrix` or `mpmath.matrix` Matrix of dimension [number of state x number of state]

eval_transition_vector(*parameters=None*, *time=None*, *state=None*)

Evaluate the transition vector given parameters, state and time. Note that the output is not in sparse format

Parameters

parameters: list see `setParameters()`

time: double The current time

state: array list The current numerical value for the states which can `numpy.ndarray` or `list`

Returns

`numpy.matrix` or `mpmath.matrix` vector of dimension [total number of transitions]

exact(*x0*, *t0*, *t1*, *output_time=False*)

Stochastic simulation using an exact method starting from time t0 to t1 with the starting state values of x0

Parameters

x: array like state vector

t0: double start time

t1: double final time

get_bd_from_ode(*A=None*)

Returns a list of:class:*Transition* from this object by unrolling the odes. All the elements are of TransitionType.B or TransitionType.D

get_birth_death_rate()

Find the algebraic equations of birth and death processes

Returns

`sympy.matrices.matrices` birth death process in matrix form

get_transition_matrix()

Returns the transition matrix in algebraic form.

Returns

`sympy.matrices.matrices` A matrix of dimension [number of state x number of state]

get_transition_vector()

Returns the set of transitions in a single vector, transitions between state to state first then the birth and death process

Returns

`sympy.matrices.matrices` A matrix of dimension [total number of transitions x 1]

get_transitions_from_ode()

Returns a list of `Transition` from this object by unrolling the odes. All the elements are of TransitionType.T

get_unrolled_obj()

Returns a `SimulateOde` with the same state and parameters as the current object but with the equations defined by a set of transitions and birth death process instead of say, odes

hybrid(*x0*, *t0*, *t1*, *output_time=False*)

Stochastic simulation using an hybrid method that uses either the first reaction method or the τ-leap depending on the size of the states and transition rates. Starting from time t0 to t1 with the starting state values of x0.

Parameters

x: array like state vector

t0: double start time

t1: double final time

plot(*sim_X=None*, *sim_T=None*)

Plot the results of a simulation

Takes the output of a function like *simulate_jump*

Parameters

sim_X: list of length iteration each with (len(t),len(state)) if t is a vector, else it outputs unequal shape that was record of all the jumps

sim_T: list or :class:'numpy.ndarray' if t is a single value, it outputs unequal shape that was record of all the jumps. if t is a vector, it outputs t so that it is a `numpy.ndarray` instead

Notes

If either sim_X or sim_T are None the this function will attempt to plot the deterministic ODE

If we have 3 states or more, it will always be arrange such that it has 3 columns. Uses the operation from `odeutils`

simulate_jump(*t*, *iteration*, *parallel=False*, *exact=False*, *full_output=False*)

Simulate the ode using stochastic simulation. It switches between a first reaction method and a τ-leap algorithm internally. When a parallel backend exists, then a new random state (seed) will be used for each processor. This is due to a lack of appropriate parallel seed random number generator in python.

Parameters

t: array like the range of time points which we want to see the result of or the final time point

iteration: int number of iterations you wish to simulate

parallel: bool, optional Defaults to True

exact: bool, optional True if exact simulation is desired, defaults to False

full_output: bool, optional if we want additional information, sim_T

Returns

sim_X: list of length iteration each with (len(t),len(state)) if t is a vector, else it outputs unequal shape that was record of all the jumps

sim_T: list or `numpy.ndarray` if t is a single value, it outputs unequal shape that was record of all the jumps. if t is a vector, it outputs t so that it is a `numpy.ndarray` instead

simulate_param(*t*, *iteration*, *parallel=False*, *full_output=False*)

Simulate the ode by generating new realization of the stochastic parameters and integrate the system deterministically.

Parameters

t: array like the range of time points which we want to see the result of

iteration: int number of iterations you wish to simulate

parallel: bool, optional Defaults to True

full_output: bool, optional if we want additional information, Y_all in the return, defaults to false

Returns

Y: numpy.ndarray of shape (len(t), len(state)), mean of all the simulation

Y_all: np.ndarray of shape (iteration, len(t), len(state))

total_transition(*state*, *t*)

Evaluate the total transition rate given state and time

Parameters

state: array like The current numerical value for the states which can be numpy.ndarray or list

t: double The current time

Returns

float total rate

transition_matrix(*state*, *t*)

Evaluate the transition matrix given state and time

Parameters

state: array like The current numerical value for the states which can be numpy.ndarray or list

t: double The current time

Returns

numpy.ndarray a 2d array of size (M,M) where M is the number of transitions

transition_mean(*state*, *t*)

Evaluate the mean of the transitions given state and time. For m transitions and n states, we have

$$
\begin{aligned}
f_{j,k} &= \sum_{i=1}^{n} \frac{\partial a_j(x)}{\partial x_i} v_{i,k} \\
\mu_j &= \sum_{k=1}^{m} f_{j,k}(x) a_k(x) \\
\sigma_j^2(x) &= \sum_{k=1}^{m} f_{j,k}^2(x) a_k(x)
\end{aligned}
$$

where $v_{i,k}$ is the state change matrix.

Parameters

state: array like The current numerical value for the states which can be numpy.ndarray or list

t: double The current time

Returns

numpy.ndarray an array of size m where m is the number of transition

transition_var(*state*, *t*)

Evaluate the variance of the transitions given state and time

Parameters

state: array like The current numerical value for the states which can be `numpy.ndarray` or `list`

t: double The current time

Returns

`numpy.ndarray` an array of size M where M is the number of transition

`transition_vector`(*state*, *t*)

Evaluate the transition vector given state and time

Parameters

state: array like The current numerical value for the states which can be `numpy.ndarray` or `list`

t: double The current time

Returns

`numpy.ndarray` a 1d array of size K where K is the number of between states transitions and the number of birth death processes

2.2.1.5 epi_analysis

Module containing functions that performs epidemiology based analysis via algebraic manipulation, such as the basic reproduction number

`pygom.model.epi_analysis.`**`DFE`**(*ode*, *disease_state*)

Returns the disease free equilibrium from an ode object

Parameters

ode: :class:'.BaseOdeModel' a class object from pygom

diseaseState: array like name of the disease states

Returns

e: array like disease free equilibrium

`pygom.model.epi_analysis.`**`R0`**(*ode*, *disease_state*)

Returns the basic reproduction number, in symbolic form when the parameter values are not available

Parameters

ode: :class:'.BaseOdeModel' a class object from pygom

diseaseStateIndex: array like name of the disease states

Returns

e: array like R0

See also:

`getDiseaseProgressionMatrices()`, `getR0GivenMatrix()`

`pygom.model.epi_analysis.`**`R0_from_matrix`**(*F*, *V*, *disease_state=None*)

Returns the symbolic form of the basic reproduction number. This will include the states symbols which is different from `getR0()` where the states is replaced by the values of the disease-free equilibrium.

Parameters

F: :class:'sympy.matrices.MatrixBase' secondary infection rates

V: :class:'sympy.matrices.MatrixBase' disease progression rates

disease_state: list like, optional list of the disease state as `sympy.Symbol`. Defaults to None which assumes that F, V had been differentiated

Returns

e: `sympy.matrices.MatrixBase` the eigenvalues of FV^{-1} for the disease states

See also:

`getDiseaseProgressionMatrices()`, `getR0()`

`pygom.model.epi_analysis.`**`disease_progression_matrices`**(*ode*, *disease_state*, *diff=True*)

Returns (F,V), the secondary infection rates and disease progression rate respectively.

Parameters

ode: :class:'.BaseOdeModel' an ode class in pygom

diseaseStates: array like the name of the disease states

diff: bool, optional if the first derivative of the matrices are return, defaults to true

Returns

(F, V): tuple The progression matrices. If diff=False, then we return the F_i and V_i matrices as per [Brauer2008].

2.2.1.6 ode_utils

Utilities used throughout the package.

class `pygom.model.ode_utils.`**`shapeAdjust`**(*numState*, *numParam*, *numTarget=None*)

A class that change vector into matrix and vice versa for vectors used in `DeterministicOde`

Parameters

numState: int number of states

numParam: int number of parameters

numTarget: int, optional number of targeted states, default assumes that this is the same as numState

`kronParam`(*A*, *pre=False*)

A sparse multiplication with an identity matrix of size equal to the number of parameters as initialized

Parameters

A: array like a 2d array

pre: bool, optional If True, then returns $I \otimes A$. If False then $A \otimes I$, where A is the input matrix, I is the identity matrix and $\otimes$ is the kron operator

`kronState`(*A*, *pre=False*)

A sparse multiplication with an identity matrix of size equal to the number of state as initialized

Parameters

A: array like a 2d array

pre: bool, optional If True, then returns $I \otimes A$. If False then $A \otimes I$, where A is the input matrix, I is the identity matrix and $\otimes$ is the kron operator

matToVecFF(*FF*)
Transforms the forward forward sensitivity matrix to a vector

matToVecSens(*S*)
Transforms the sensitivity matrix to a vector

vecToMatFF(*ff*)
Transforms the forward forward sensitivity vector to a matrix

vecToMatSens(*s*)
Transforms the sensitivity vector to a matrix

pygom.model.ode_utils.**integrate**(*ode*, *x0*, *t*, *full_output=False*)
A wrapper on top of `odeint` using `DeterministicOde`.

Parameters

ode: object of type `DeterministicOde`

t: array like the time points including initial time

full_output: bool, optional If the additional information from the integration is required

pygom.model.ode_utils.**integrateFuncJac**(*func*, *jac*, *x0*, *t0*, *t*, *args=()*, *includeOrigin=False*, *full_output=False*, *method=None*, *nsteps=10000*)
A replacement for `scipy.integrate.odeint` which performs integration using `scipy.integrate.ode`, tries to pick the correct integration method at the start through eigenvalue analysis

Parameters

func: callable the ode $f(x)$

jac: callable jacobian of the ode, $J_{i,j} = \nabla_{x_j} f_i(x)$

x0: 'numpy.ndarray' or list of numeric initial value of the states

t0: float initial time

args: tuple, optional additional arguments to be passed on

includeOrigin: bool, optional if the output should include the initial states x0

full_output: bool, optional if additional output is required

method: str, optional the integration method. All those availble in `ode` are allowed with 'vode' and 'ivode' representing the non-stiff and stiff version respectively. Defaults to None, which tries to choose the integration method via eigenvalue analysis (only one) using the initial conditions

nstep: int, optional number of steps allowed between each time point of the integration

Returns

solution: array like a `np.ndarray` of shape (len(t), len(x0)) if includeOrigin is False, else an extra row with x0 being the first.

output [dict, only returned if full_output=True] Dictionary containing additional output information

key	meaning
'ev'	vector of eigenvalues at each t
'maxev'	maximum eigenvalue at each t
'minev'	minimum eigenvalue at each t
'suc'	list whether integration is successful
'in'	name of integrator

class pygom.model.ode_utils.**compileCode**(*backend=None*)

A class that compiles an algebraic expression in sympy to a faster numerical file using the appropriate backend.

compileExpr(*inputSymb*, *inputExpr*, *backend=None*, *compileType=False*)

Compiles the expression given the symbols. Determines the backend if required.

Parameters

inputSymb: list the set of symbols for the input expression

inputExpr: expr expression in sympy

backend: optional the backend we want to use to compile

compileType: optional defaults to False. If True, return an extra output that informs the end user of the method used to compile the equation, can be one of (np, mpmath, sympy)

Returns

Compiled function taking arguments of the input symbols

compileExprAndFormat(*inputSymb*, *inputExpr*, *backend=None*, *modules=None*, *outType=None*)

Compiles the expression given the symbols and determine which type of output is it. Transforms the output appropriately into numpy

Parameters

inputSymb: list the set of symbols for the input expression

inputExpr: expr expression in sympy

backend: optional the backend we want to use to compile

modules: optional in the event that f2py and Cython fails, which modules do we want to try and compile against

Returns

Function determined from the input using closures.

class pygom.model.ode_utils.**CompileCanary**

Hold the need for (re-)compilation for various functions

A subclass of this should specify the states to watch

They may all be tripped to True using the trip() method An individual may be reset with the reset() method or with a direct assignment (they may not be tripped in this way).

reset(*name*)

Reset a canary

Parameters

name: string the name of the canary to reset

Returns

None

trip()
Trip all the canaries Returns ——- None

pygom.model.ode_utils.**plot_det**(*solution*, *t*, *stateList=None*, *y=None*, *yStateList=None*)
Plot the results of the integration

Parameters

solution: :class:'numpy.ndarray' solution from the integration

t: array like the vector of time where the integration output correspond to

stateList: list name of the states, if available

Notes

If we have 5 states or more, it will always be arrange such that it has 3 columns.

pygom.model.ode_utils.**plot_stoc**(*solution*, *t*, *stochastic_model*)
Plot the results of a stocastic simulation

Parameters

solution: :class: list results of the stochastic simulation

t: array like the vector of time where the integration output correspond to

stochastic_model: :class: 'pygom.SimulateOde' the model from which this simulation was generated

Notes

If we have 5 states or more, it will always be arrange such that it has 3 columns.

pygom.model.ode_utils.**check_array_type**(*x*)
Check to see if the type of input is suitable. Only operate on one or two dimension arrays

Parameters

x: array like which can be either a `numpy.ndarray` or list or tuple

Returns

x: `numpy.ndarray` checked and converted array

pygom.model.ode_utils.**check_dimension**(*x*, *y*)
Compare the length of two array like objects. Converting both to a numpy array in the process if they are not already one.

Parameters

x: array like first array

y: array like second array

Returns

x: `numpy.array` checked and converted first array

y: `numpy.array` checked and converted second array

pygom.model.ode_utils.**is_list_like**(*x*)
Test whether the input is a type that behaves like a list, such as (list,tuple,np.ndarray)

Parameters

x: anything

Returns

bool: True if it belongs to one of the three expected type (list,tuple,np.ndarray)

`pygom.model.ode_utils.str_or_list(x)`

Test to see whether input is a string or a list. If it is a string, then we convert it to a list.

Parameters

x: str or list

Returns

x: x in list form

2.2.2 loss

2.2.2.1 ode_loss

These are basically the interfaces for `pygom.loss.BaseLoss`

The loss functions that "implements" the class BaseLoss, if Python has such a thing. Overrides the method _setLossType

class `pygom.loss.ode_loss.SquareLoss`(*theta*, *ode*, *x0*, *t0*, *t*, *y*, *state_name*, *state_weight=None*, *target_param=None*, *target_state=None*)

The square loss function

class `pygom.loss.ode_loss.NormalLoss`(*theta*, *ode*, *x0*, *t0*, *t*, *y*, *state_name*, *sigma=None*, *target_param=None*, *target_state=None*)

Realizations from a Normal distribution

class `pygom.loss.ode_loss.PoissonLoss`(*theta*, *ode*, *x0*, *t0*, *t*, *y*, *state_name*, *target_param=None*, *target_state=None*)

Realizations from a Poisson distribution

2.2.2.2 calculations

The base class which contains has all the calculation implemented

To place everything about estimating the parameters of an ode model under square loss in one single module. Focus on the standard local method which means obtaining the gradient and Hessian.

class `pygom.loss.base_loss.BaseLoss`(*theta*, *ode*, *x0*, *t0*, *t*, *y*, *state_name*, *state_weight=None*, *target_param=None*, *target_state=None*)

This contains the base that stores all the information of an ode.

Parameters

theta: array like input value of the parameters

ode: :class:'DeterministicOde' the ode class in this package

x0: numeric initial time

t0: numeric initial value

t: array like time points where observations were made

y: array like observations

state_name: str the state which the observations came from

state_weight: array like weight for the observations

target_param: str or array like parameters that are not fixed

target_state: str or array like states that are not fixed, applicable only when the initial values are also of interest

adjoint(*theta=None*, *full_output=False*)

Obtain the gradient given input parameters using the adjoint method. Values of state variable are found using an univariate spline interpolation between two observed time points where the internal knots are explicitly defined.

Parameters

theta: array like input value of the parameters

full_output: bool if True, also output the full set of adjoint values (over time)

Returns

grad: numpy.ndarray array of gradient

infodict [dict, only returned if full_output=True] Dictionary containing additional output information

key	meaning
'resid'	residuals given theta
'diff_loss'	derivative of the loss function
'gradVec'	gradient vectors
'adjVec'	adjoint vectors
'interpolateInfo'	info from integration over the interpolating points
'solInterpolate'	solution from the integration over the interpolating points
'tInterpolate'	interpolating time points

See also:

sensitivity()

cost(*theta=None*)

Find the cost/loss given time points and the corresponding observations.

Parameters

theta: array like input value of the parameters

Returns

numeric sum of the residuals squared

See also:

diff_loss()

Notes

Only works with a single target (state)

costIV(*theta=None*)

Find the cost/loss given the parameters. The input theta here is assumed to include both the parameters as well as the initial values

Parameters

theta: array like parameters and guess of initial values of the states

Returns

numeric sum of the residuals squared

See also:

residualIV()

diff_loss(*theta=None*)

Find the derivative of the loss function given time points and the corresponding observations, with initial conditions

Parameters

theta: array like input value of the parameters

Returns

numpy.ndarray an array of residuals

See also:

cost()

diff_lossIV(*theta=None*)

Find the derivative of the loss function w.r.t. the parameters given time points and the corresponding observations, with initial conditions.

Parameters

theta: array like parameters and initial values of the states

Returns

numpy.ndarray an array of result

See also:

costIV(), diff_loss()

fisher_information(*theta=None*, *full_output=False*, *method=None*)

Obtain the Fisher information

Parameters

theta: array like input value of the parameters

full_output: bool if additional output is required

method: str, optional what method to use in the integrator

Returns

I: numpy.ndarray $I(\theta)$ of the objective function

infodict [dict, only returned if full_output=True] Dictionary containing additional output information

key	meaning
'state'	intermediate values for the state (original ode)
'sens'	intermediate values for the sensitivities by state, parameters, i.e. $x_{(i-1)p+j}$ is the element for state i and parameter j with a total of p parameters
'resid'	residuals given theta
'info'	output from the integration

See also:

`sensitivity()`, `jtj()`

fit(*x*, *lb=None*, *ub=None*, *A=None*, *b=None*, *disp=False*, *full_output=False*)

Find the estimates given the data and an initial guess x. Note that there is no guarantee that the estimation procedure is successful. It is recommended to at least supply box constraints, i.e. lower and upper bounds

Parameters

x: array like an initial guess

lb: array like the lower bound elementwise $lb_i <= x_i$

ub: array like upper bound elementwise $x_i <= ub_i$

A: array like matrix A for the inequality $Ax <= b$

b: array like vector b for the inequality $Ax <= b$

Returns

xhat: `numpy.ndarray` estimated value

gradient(*theta=None*, *full_output=False*)

Returns the gradient calculated by solving the forward sensitivity equation. Identical to `sensitivity()` without the choice of integration method

See also:

`sensitivity()`

hessian(*theta=None*, *full_output=False*, *method=None*)

Obtain the Hessian using the forward forward sensitivities.

Parameters

theta: array like input value of the parameters

full_output: bool if additional output is required

method: str, optional what method to use in the integrator

Returns

Hessian: `numpy.ndarray` Hessian of the objective function

infodict [dict, only returned if full_output=True] Dictionary containing additional output information

key	meaning
'state'	intermediate values for the state (original ode)
'sens'	intermediate values for the sensitivities by state, parameters, i.e. $x_{(i-1)p+j}$ is the element for state i and parameter j with a total of p parameters
'hess'	intermediate values for the hessian by state, parameter, parameter, i.e. $x_{(i-1)p^2+j+k}$ is the element for state i, parameter j and parameter k
'resid'	residuals given theta
'info'	output from the integration

See also:

`sensitivity()`

`jac`(*theta=None*, *sens_output=False*, *full_output=False*, *method=None*)

Obtain the Jacobian of the objective function given input parameters using forward sensitivity method.

Parameters

theta: array like, optional input value of the parameters

sens_output: bool, optional whether the full sensitivities is required; full_output overrides this option when true

full_output: bool, optional if additional output is required

method: str, optional Choice between lsoda, vode and dopri5, the three integrator provided by scipy. Defaults to lsoda.

Returns

grad: `numpy.ndarray` Jacobian of the objective function

infodict [dict, only returned if full_output=True] Dictionary containing additional output information

key	meaning
'sens'	intermediate values over the original ode and all the sensitivities, by state, parameters
'resid'	residuals given theta
'diff_loss'	derivative of the loss function

See also:

`sensitivity()`

`jacIV`(*theta=None*, *sens_output=False*, *full_output=False*, *method=None*)

Obtain the Jacobian of the objective function given input parameters which include the current guess of the initial value using forward sensitivity method.

Parameters

theta: array like, optional input value of the parameters

sens_output: bool, optional whether the full sensitivities is required; full_output overrides this option when true

full_output: bool, optional if additional output is required

method: str, optional Choice between lsoda, vode and dopri5, the three integrator provided by scipy. Defaults to lsoda

Returns

grad: `numpy.ndarray` Jacobian of the objective function

infodict [dict, only returned if full_output=True] Dictionary containing additional output information

key	meaning
'sens'	intermediate values over the original ode and all the sensitivities, by state, parameters
'resid'	residuals given theta
'info'	output from the integration

See also:

`sensitivityIV()`

`jtj`(*theta=None*, *full_output=False*, *method=None*)

Obtain the approximation to the Hessian using the inner product of the Jacobian.

Parameters

theta: array like input value of the parameters

full_output: bool if additional output is required

method: str, optional what method to use in the integrator

Returns

jtj: `numpy.ndarray` $J^\top J$ of the objective function

infodict [dict, only returned if full_output=True] Dictionary containing additional output information

key	meaning
'state'	intermediate values for the state (original ode)
'sens'	intermediate values for the sensitivities by state, parameters, i.e. $x_{(i-1)p+j}$ is the element for state i and parameter j with a total of p parameters
'resid'	residuals given theta
'info'	output from the integration

See also:

`sensitivity()`

`plot`()

Plots the solution of all the states and the observed y values

`residual`(*theta=None*)

Find the residuals given time points and the corresponding observations, with initial conditions

Parameters

theta: array like input value of the parameters

Returns

`numpy.ndarray` an array of residuals

See also:

`cost()`

Notes

Makes a direct call to initialized loss object which has a method called residual

residualIV(*theta=None*)

Find the residuals given time points and the corresponding observations, with initial conditions.

Parameters

theta: array like parameters and initial values of the states

Returns

`numpy.ndarray` an array of residuals

See also:

`costIV()`, `residual()`

Notes

Makes a direct call to `residual()` using the initialized information

sens_to_grad(*sens*, *diff_loss*)

Forward sensitivites to the gradient.

Parameters

sens: :class:'numpy.ndarray' forward sensitivities

diff_loss: array like derivative of the loss function

Returns

g: `numpy.ndarray` gradient of the loss function

sens_to_jtj(*sens*, *resid=None*)

forward sensitivites to $J^\top J$ where J is the Jacobian. The approximation to the Hessian.

Parameters

sens: :class:'numpy.ndarray' forward sensitivities

resid: :class:'numpy.ndarray', optional the residuals corresponding to the input sens

Returns

JTJ: `numpy.ndarray` An approximation to the Hessian using the inner product of the Jacobian

sensitivity(*theta=None*, *full_output=False*, *method=None*)

Obtain the gradient given input parameters using forward sensitivity method.

Parameters

theta: array like input value of the parameters

full_output: bool if additional output is required

method: str, optional what method to use in the integrator

Returns

grad: `numpy.ndarray` array of gradient

infodict [dict, only returned if full_output=True] Dictionary containing additional output information. Same output as `jac()`

Notes

It calculates the gradient by calling `jac()`

sensitivityIV(*theta=None*, *full_output=False*, *method=None*)

Obtain the gradient given input parameters (which include the current guess of the initial conditions) using forward sensitivity method.

Parameters

theta: array like, optional input value of the parameters

full_output: bool, optional if additional output is required

method: str, optional what method to use in the integrator

Returns

grad: `numpy.ndarray` array of gradient

infodict [dict, only returned if full_output=True] Dictionary containing additional output information

key	meaning
'sens'	intermediate values over the original ode and all the sensitivities, by state, parameters
'resid'	residuals given theta
'info'	output from the integration

Notes

It calculates the gradient by calling `jacIV()`

thetaCallBack(*x*)

Print x, the parameters

thetaCallBack2(*x*, *f*)

Print x and f where x is the parameter of interest and f is the objective function

Parameters

x: parameters

f: f(x)

2.2.2.3 confidence_interval

Module that is used to calculate the confidence interval given the estimated parameters

`pygom.loss.confidence_interval.`**asymptotic**(*obj*, *alpha=0.05*, *theta=None*, *lb=None*, *ub=None*)

Finds the confidence interval at the α level under the $\mathcal{X}^2$ assumption for the likelihood

Parameters

obj: ode object an object initialized from `BaseLoss`

alpha: numeric, optional confidence level, $0 < \alpha < 1$. Defaults to 0.05.

theta: array like, optional the MLE parameters. Defaults to None which then theta will be inferred from the input obj

lb: array like, optional expected lower bound

ub: array like, optional expected upper bound

Returns

l: array like lower confidence interval

u: array like upper confidence interval

`pygom.loss.confidence_interval.`**`profile`**(*obj*, *alpha*, *theta=None*, *lb=None*, *ub=None*, *full_output=False*)

Finds the profile confidence interval at the α level under the $\mathcal{X}^2$ assumption for the likelihood

Parameters

obj: ode object an object initialized from `BaseLoss`

alpha: numeric confidence level, $0 < \alpha < 1$

theta: array like, optional the MLE parameters. When None given, it tries to estimate the optimal using methods provided by obj

lb: array like, optional expected lower bound

ub: array like, optional expected upper bound

full_output: bool, optional if more output is desired

Returns

l: array like lower confidence interval

u: array like upper confidence interval

`pygom.loss.confidence_interval.`**`bootstrap`**(*obj*, *alpha=0.05*, *theta=None*, *lb=None*, *ub=None*, *iteration=0*, *full_output=False*)

Finds the confidence interval at the α level via bootstrap

Parameters

obj: ode object an object initialized from `BaseLoss`

alpha: numeric, optional confidence level, $0 < \alpha < 1$. Defaults to 0.05.

theta: array like, optional the MLE parameters

lb: array like, optional upper bound for the parameters

ub: array like, optional lower bound for the parameters

iteration: int, optional number of bootstrap samples, defaults to 0 which is interpreted as $min(2n, 100)$ where n is the number of data points.

full_output: bool if the full set of estimates is required.

Returns

l: array like lower confidence interval

u: array like upper confidence interval

`pygom.loss.confidence_interval.`**`geometric`**(*obj*, *alpha=0.05*, *theta=None*, *method='jtj'*, *geometry='o'*, *full_output=False*)

Finds the geometric confidence interval under profiling at the α level the normal approximation

Parameters

obj: ode object an object initialized from `BaseLoss`

alpha: numeric confidence level, $0 < \alpha < 1$

theta: array like, optional the MLE parameters. When None given, it tries to estimate the optimal using methods provided by obj

method: string construction of the covariance matrix. jtj is the $J^\top$ where J is the Jacobian of the ode. 'hessian' is the hessian of the ode while 'fisher' is the fisher information found by $cov(\nabla_\theta \mathcal{L})$.

geometry: string the two types of geometry defined in [Moolgavkar1987]. c geometry uses the covariance at the maximum likelihood estimate $\hat{\theta}$, while the 'o' geometry is the covariance defined at point θ.

full_output: bool, optional If True then both the l_path and u_path will be outputted, else only the point estimates of l and u

Returns

l: array like lower confidence interval

u: array like upper confidence interval

l_path: list path from $\hat{\theta}$ to the lower $1 - \alpha/2$ point for all parameters

u_path: list same as l_path but for the upper confidence interval

2.2.2.4 loss_type

The different loss types. Such as thought based on parametric distributions.

class `pygom.loss.loss_type.`**`Square`**(*y*, *weights=None*)

Square loss object

Parameters

y: array like observations

`diff2Loss`(*yhat*)

Twice derivative of the square loss. Which is simply 2.

Parameters

yhat: array like observations

Returns

array with values of 2: either a scalar, vector or matrix depending on the shape of of the input yhat

`diff_loss`(*yhat*)

Derivative under square loss. Assuming that we are solving the minimization problem i.e. our objective function is the negative of the log-likelihood

Parameters

yhat: array like observation

Returns

$-2(y_i - \hat{y}_i)$

loss (*yhat*)

Loss under square loss. Not really saying much here

Parameters

yhat: array like observation

Returns

$\sum_{i=1}^{n}(\hat{y} - y)^2$

residual (*yhat*)

Raw residuals if no weights was initialized, else the weighted residuals

Parameters

yhat: array like observation

Returns

$y_i - \hat{y}_i$

class pygom.loss.loss_type.**Normal** (*y*, *sigma=1.0*)

Normal distribution loss object

Parameters

y: array like observation

sigma: float standard deviation

diff2Loss (*yhat*)

Twice derivative of the normal loss.

Parameters

yhat: array like observations

Returns

s: array like inverse of the variance with shape = yhat.shape

diff_loss (*yhat*)

Derivative of the loss function which is $\sigma^{-1}(y - \hat{y})$

Parameters

yhat: array like observation

Returns

r: array like $\nabla\mathcal{L}(\hat{y}, y)$

loss (*yhat*)

The loss under a normal distribution. Defined as the negative log-likelihood here.

Parameters

yhat: array like observation

Returns

negative log-likelihood, $\mathcal{L}(\hat{y}, y)$

residual(*yhat*)
Residuals under a normal loss

Parameters

yhat: array like observation

Returns

r: array like residuals

class pygom.loss.loss_type.**Poisson**(*y*)
Poisson distribution loss object

Parameters

y: array like observation

diff2Loss(*yhat*)
Twice derivative of the Poisson loss.

Parameters

yhat: array like observations

Returns

s: array like $\frac{y}{\hat{y}^2}$ with shape = yhat.shape

diff_loss(*yhat*)
Derivative of the loss function, $1 - y\hat{y}^{-1}$

Parameters

yhat: array like observation

Returns

$\nabla\mathcal{L}(\hat{y}, y)$

loss(*yhat*)
The loss under a Poisson distribution. Defined as the negative log-likelihood here.

Parameters

yhat: array like observation

Returns

negative log-likelihood, $\mathcal{L}(\hat{y}, y)$

residual(*yhat*)
Raw residuals

Parameters

yhat: array like observation

Returns

r: array like residuals

2.2.2.5 get_init

pygom.loss.get_init.**cost_grad_interpolant**(*ode*, *spline_list*, *t*, *theta*)
Returns the cost (sum of squared residuals) and the gradient between the first derivative of the interpolant and the function of the ode

Parameters

ode: :class:'.DeterministicOde' an ode object

spline_list: list list of `scipy.interpolate.UnivariateSpline`

t: array like time

theta: array list parameter value

Returns

cost: double sum of squared residuals

g: gradient of the squared residuals

`pygom.loss.get_init.`**`cost_interpolant`**(*ode*, *spline_list*, *t*, *theta*, *vec=True*, *aggregate=True*)

Returns the cost (sum of squared residuals) between the first derivative of the interpolant and the function of the ode

Parameters

ode: :class:'.DeterministicOde' an ode object

spline_list: list list of `scipy.interpolate.UnivariateSpline`

t: array like time

theta: array list paramter value

vec: bool, optional if the matrix should be flattened to be a vector

aggregate: bool, optional sum the vector/matrix

Returns

cost: double sum of squared residuals

`pygom.loss.get_init.`**`cost_sample`**(*ode*, *fxApprox*, *xApprox*, *t*, *theta*, *vec=True*, *aggregate=True*)

Returns the cost (sum of squared residuals) between the first derivative of the interpolant and the function of the ode using samples at time points t.

Parameters

ode: :class:'.DeterministicOde' an ode object

fxApprox: list list of approximated values for the first derivative

xApprox: list list of approximated values for the states

t: array like time

theta: array list parameter value

vec: bool, optional if the matrix should be flattened to be a vector.

aggregate: bool/str, optional sum the vector/matrix. If this is equals to 'int' then the Simpsons rule is applied to the samples. Also changes the behaviour of vec, where True outputs a vector where the elements contain the values of the integrand on each of the dimensions of the ode. False returns the sum of this vector, a scalar.

Returns

r: array list the cost or the residuals if vec is True

See also:

`residual_sample()`

`pygom.loss.get_init.`**`get_init`**(*y*, *t*, *ode*, *theta=None*, *full_output=False*)

Get an initial guess of theta given the observations y and the corresponding time points t.

Parameters

y: :array like observed values

t: array like time

ode: :class:'.DeterministicOde' an ode object

theta: array like parameter value

full_output: bool, optional True if the optimization result should be returned. Defaults to False.

Returns

theta: array like a guess of the parameters

`pygom.loss.get_init.`**`grad_sample`**(*ode*, *fxApprox*, *xApprox*, *t*, *theta*, *vec=False*, *output_residual=False*)

Returns the gradient of the objective value using the state values of the interpolant given samples at time points t. Note that the parameters taken here is chosen to be same as `cost_sample()` for convenience.

Parameters

ode: :class:'.DeterministicOde' an ode object

fxApprox: list list of approximated values for the first derivative

xApprox: list list of approximated values for the states

t: array like time

theta: array list parameter value

vec: bool, optional if the matrix should be flattened to be a vector

output_residual: bool, optional if True, then the residuals will be returned as an additional argument

Returns

g: `numpy.ndarray` gradient of the objective function

See also:

`jac_sample()`

`pygom.loss.get_init.`**`interpolate`**(*solution*, *t*, *s=0*)

Interpolate the solution of the ode given the time points and a suitable smoothing vector using univariate spline

Parameters

solution: :class:'numpy.ndarray' f(t) of the ode with the rows correspond to time

t: array like time

s: smoothing scalar, optional greater or equal to zero

Returns

splineList: list of `scipy.interpolate.UnivariateSpline`

`pygom.loss.get_init.`**`jac_sample`**(*ode*, *fxApprox*, *xApprox*, *t*, *theta*, *vec=True*)

Returns the Jacobian of the objective value using the state values of the interpolant given samples at time points t. Note that the parameters taken here is chosen to be same as `cost_sample()` for convenience.

Parameters

ode: :class:'.DeterministicOde' an ode object

fxApprox: list list of approximated values for the first derivative

xApprox: list list of approximated values for the states

t: array like time

theta: array list parameter value

vec: bool, optional if the matrix should be flattened to be a vector

Returns

r: array list the residuals

See also:

cost_sample()

pygom.loss.get_init.**residual_interpolant**(*ode*, *spline_list*, *t*, *theta*, *vec=True*)

Returns the residuals between the first derivative of the interpolant and the function of the ode

Parameters

ode: :class:'.DeterministicOde' an ode object

spline_list: list list of `scipy.interpolate.UnivariateSpline`

t: array like time

theta: array list parameter value

vec: bool, optional if the matrix should be flattened to be a vector

aggregate: bool, optional sum the vector/matrix

Returns

r: array list the residuals

pygom.loss.get_init.**residual_sample**(*ode*, *fxApprox*, *xApprox*, *t*, *theta*, *vec=True*)

Returns the residuals between the first derivative of the interpolant and the function of the ode using samples at time points t.

Parameters

ode: :class:'.DeterministicOde' an ode object

fxApprox: list list of approximated values for the first derivative

xApprox: list list of approximated values for the states

t: array like time

theta: array list parameter value

vec: bool, optional if the matrix should be flattened to be a vector

Returns

r: array list the residuals

See also:

cost_sample()

CHAPTER 3

References

3.1 References

CHAPTER 4

Indices and tables

- genindex
- modindex

Bibliography

[Aron1984] Seasonality and period-doubling bifurcations in an epidemic model, Joan L. Aron and Ira B. Schwartz, Journal of Theoretical Biology, Volume 110, Issue 4, page 665-679, 1984

[Brauer2008] Mathematical Epidemiology, Lecture Notes in Mathematics, Fred Brauer, Springer 2008

[Cao2006] Efficient step size selection for the tau-leaping simulation method, Yang Cao et el., The Journal of Chemical Physics, Volume 124, Issue 4, page 044109, 2006

[Finnie2016] EpiJSON: A unified data-format for epidemiology, Thomas Finnie et al., Epidemics, Volume 15, page 20-26, 2016

[FitzHugh1961] Impulses and Physiological States in Theoretical Models of Nerve Membrane, Richard FitzHugh, Biophysical Journal, Volume 1, Issue 6, page 445-466, 1961

[Gillespie1977] Exact stochastic simulation of coupled chemical reactions, Danial T. Gillespie, The Journal of Physical Chemistry, Volume 81, Issue 25, page 2340-2361, 1977

[Girolami2011] Riemann manifold Langevin and Hamiltonian Monte Carlo methods, Mark Girolami and Ben Calderhead, Journal of the Royal Statistical Society Series B, Volume 73, Issue 2, page 123-214, 2011.

[Hethcote1973] Asymptotic behavior in a deterministic epidemic model, Herbert W. Hethcote, Bulletin of Mathematical Biology, Volume 35, page 607-614, 1973

[Legrand2007] Understanding the dynamics of Ebola epidemics, J. Legrand et al. Epidemiology and Infection, Volume 135, Issue 4, page 610-621, 2007

[Lloyd1996] Spatial Heterogeneity in Epidemic Models, A.L. Lloyd and R.M. May, Journal of Theoretical Biology, Volume 179, Issue 1, page 1-11, 1996

[Lorenz1963] Deterministic Nonperiodic Flow, Edward N. Lorenz, Journal of the Atmospheric Sciences, Volume 20, Issue 2, page 130-141, 1963

[Lotka1920] Analytical Note on Certain Rhythmic Relations in Organic Systems, Alfred J. Lotka, Proceedings of the National Academy of Sciences of the United States of America, Volume 7, Issue 7, page 410-415, 1920

[Moolgavkar1987] Confidence Regions for Parameters of the Proportional Hazards Model: A Simulation Study, S.H. Moolgavkar and D.J. Venzon, Scandianvia Journal of Statistics, Volume 14, page 43-56, 1987

[Press2007] Numerical Recipes 3rd Edition: The Art of Scientific Computing, W.H. Press et al., Cambridge University Press, 2007

[Ramsay2007] Parameter estimation for differential equations: a generalized smoothing approach, Journal of the Royal Statistical Society Series B, James O. Ramsay et al., Volume 69, Issue 5, page 741-796, 2007

[Raue2009] Structural and Practical Identifiability Analysis of Partially Observed Dynamical Models by Exploiting the Profile Likelihood, A. Raue et al., Bioinformatics, Volume 25, Issue 15, page 1923-1929, 2009

[Robertson1966] The solution of a set of reaction rate equations, H.H. Robertson, Academic Press, page 178-182, 1966

[vanderpol1926] On Relaxed Oscillations, Balthasar van der Pol, The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science, Volume 2, Issue 11, page 978-992, 1926

[Venzon1988] A Method for Computing Profile-Likelihood-Based Confidence Intervals, D.J. Venzon and S.H. Moolgavkar, Journal of the Royal Statistical Society Series C (Applied Statistics), Volume 37, Issue 1, page 87-94, 1988