PCTMC_methods

April 28, 2022

[1]:
```python
import sympy as sp
import numpy as np
import time
from scipy.integrate import odeint
import matplotlib.pyplot as plt
import numpy.random as rnd
from math import ceil
```

[2]:
```python
class Symbols:

    def __init__(self):
        # initialize
        self.reference = []
        self.values = []
        # initialize map from string to int and back
        self.reference2id = {}
        self.names2id = {}
        self.id2reference = {}
        self.dimension = 0

    #Adds a new symbol to the symbol array, creating a sympy object
    def add(self, name, value):
        symbol = sp.symbols(name)
        index = len(self.reference)
        self.reference.append(symbol)
        self.values.append(value)
        self.reference2id[symbol] = index
        self.names2id[name] = index
        self.id2reference[index] = symbol
        self.dimension += 1
        return symbol

    # sets the value of a symbol
    def set(self, name, value):
        try:
            index = self.names2id[name]
            self.values[index] = value
```

```
        except:
            print("Symbol " + name + " is not defined")

    def get_value(self,name):
        try:
            index = self.names2id[name]
            return self.values[index]
        except:
            print("Symbol " + name + " is not defined")

    #finalizes the symbol array, generating a numpy array for values
    def finalize(self):
        self.values = np.array(self.values)

    def get_id(self, name):
        return self.names2id[name]
```

[3]:
```
class Transition:

    def __init__(self, update, rate, sympy_symbols):
        #this will convert rate into a sympy expression.

        self.update = None
        self.update_dictionary = update
        try:
            self.rate = sp.sympify(rate, sympy_symbols, evaluate=False)
        except sp.SympifyError as e:
            print("An error happened while parsing expression", rate,":",e)

    # finalizes transition by turning the update list into a numpy array
    def finalize(self, variables):
        self.update = np.zeros(variables.dimension)
        for var_name in self.update_dictionary:
            index = variables.get_id(var_name)
            self.update[index] = self.update_dictionary[var_name]
        self.update = np.reshape(self.update, (1,variables.dimension))
```

[5]:
```
class Model:

    def __init__(self):
        #init variables and parameters
        self.variables = Symbols()
        self.parameters = Symbols()
        # this contains a map of names to sympy variables, to be used later for␣
↪parsing expressions
        self.names2sym = {}
        # init transition list
```

```
        self.transitions = []
        self.transition_number = 0
        self.system_size = 0;
        self.system_size_reference = None
        self.system_size_name = ''
        self.variables_names = []

    def set_system_size(self, name, value):
        self.add_parameter(name, value)
        self.system_size_reference = self.names2sym[name]
        self.system_size_name = name
        self.system_size = value


    def add_variable(self, name, value):
        if name in self.names2sym:
            raise ModelError("Name " + name + " already defined!")
        var = self.variables.add(name, value)
        self.variables_names.append(name)
        self.names2sym[name] = var

    def add_parameter(self, name, value):
        if name in self.names2sym:
            raise ModelError("Name " + name + " already defined!")
        par = self.parameters.add(name, value)
        self.names2sym[name] = par

    # Changes the initial value of a variable
    def set_variable(self, name, value):
        self.variables.set(name, value)

    # Changes the value of a parameter
    def set_parameter(self, name, value):
        if self.system_size_name == name:
            self.parameters.set(name, value)
            self.system_size = value
        else:
            self.parameters.set(name, value)

    def get_parameter_value(self, name):
        return self.parameters.get_value(name)

    # Adds a transition to the model
    def add_transition(self, update, rate):
        t = Transition(update, rate, self.names2sym)
        self.transitions.append(t)
        self.transition_number += 1
```

```
    # Finalizes the initialization
    def finalize_initialization(self):
        self.variables.finalize()
        self.parameters.finalize()
        for t in self.transitions:
            t.finalize(self.variables)

        self.__generate_vector_field()
        self.__generate_diffusion()
        self.__generate_jacobian()
        self.__generate_numpy_functions()


    #generates the mean field vector field
    def __generate_vector_field(self):
        self._vector_field_sympy = np.zeros(self.transitions[0].update.shape,␣
↪dtype=object)

        '''

        CODE HERE: fluid equation

        '''

        self._vector_field_sympy = sp.simplify(self._vector_field_sympy)


    #generates the diffusion term
    def __generate_diffusion(self):
        self._diffusion_sympy = 0

        '''

        CODE HERE: diffusion term

        '''

    # computes symbolically the jacobian of the vector field
    def __generate_jacobian(self):
        n = self.variables.dimension
        f = self._vector_field_sympy
        x = self.variables.reference
        J = np.zeros((n, n), dtype=object)

        '''
```

```
        CODE HERE: javobian (sympy.diff)

        '''

        self._jacobian_sympy = J



    # generate numpy expressions and the mean field VF
    def __generate_numpy_functions(self):
        sympy_ref = self.variables.reference + self.parameters.reference
        self.rates = sp.lambdify(sympy_ref, [t.rate for t in self.transitions],␣
↪"numpy")
        self.vector_field = sp.lambdify(sympy_ref, self._vector_field_sympy,␣
↪"numpy")
        self.diffusion = sp.lambdify(sympy_ref, self._diffusion_sympy, "numpy")
        self.jacobian = sp.lambdify(sympy_ref, self._jacobian_sympy, "numpy")


    #evaluates and returns vector field, diffusion, ...
    def evaluate_all_vector_fields(self, var_values):
        f = self.vector_field(*var_values, *self.parameters.values)
        D = self.diffusion(*var_values, *self.parameters.values)
        J = self.jacobian(*var_values, *self.parameters.values)
        return np.asarray(f), np.asarray(D), np.asarray(J)

    def evaluate_MF_vector_field(self, var_values):
        f = self.vector_field(*var_values, *self.parameters.values)
        return np.asarray(f)


    def evaluate_rates(self, var_values):
        r = self.rates(*var_values, *self.parameters.values)
        return np.asarray(r)
```

[6]:
```
class Simulator:
    def __init__(self, model):
        self.model = model
        self.t0 = 0
        self.x0 = model.variables.values

    def _unpack(self, z):
        n = self.model.variables.dimension
        phi = z[0:n]  # mean field
        c = np.reshape(z[n:], (n, n))  # c term
        return phi, c
```

```python
    def _pack(self, f, dc):
        z = np.concatenate((f.flatten(), dc.flatten()))
        return z

    # computes the full vector field for the linear noise ODE
    def _linear_noise_ODE(self, z, t):
        x_t, c_t = self._unpack(z)

        '''

        CODE HERE: linear noise eq

        '''

        dz = self._pack(dx_dt, dc_dt)
        return dz

    # computes the vector field for the classic mean field
    def _mean_field_ODE(self, x, t):

        '''

        CODE HERE: mean field eq

        '''

        return dx.flatten()


    def _generate_time_stamp(self, final_time, points):
        """
        Generates a time stamp from time self.t0 to final_time,
        with points+1 number of points.

        :param final_time: final time of the simulation
        :param points: number of points
        :return: a time stamp numpy array
        """
        step = (final_time - self.t0) / points
        time = np.arange(self.t0, final_time + step, step)
        return time

    def _SSA_single_simulation(self, final_time, time_stamp, model_dimension,␣
↪trans_number):
        """
        A single SSA simulation run
```

```
        :param final_time: final simulation time
        :param time_stamp: time array containing time points to save
        :param model_dimension: dimension of the model
        :param trans_number: transitions' number
        :return: the variables computed along the trajectory
        """
        # tracks simulation time and state
        time = 0
        state = self.x0
        # tracks index of the time stamp vector, to save the array
        print_index = 1
        x = np.zeros((len(time_stamp), model_dimension))
        # save initial state
        x[0, :] = self.x0
        # main SSA loop
        trans_code = range(trans_number)
        while time < final_time:
            # compute rates and total rate

            # sanity check, to avoid negative numbers close to zero


            # COMPLETE HERE


            # check if total rate is non zero.

            # store values in the output array
            while print_index < len(time_stamp) and time_stamp[print_index] <=␣
↪time:
                x[print_index, :] = state
                print_index += 1

        return x


    def SSA_simulation(self, final_time, runs=100, points=1000, update=1):
        """
        Runs SSA simulation for a given number of runs and returns the average

        :param final_time: final simulation time
        :param runs: number of runs, default is 100
        :param points: number of points to be saved, default is 1000
        :param update: percentage step to update simulation time on screen
        :return: a Trajectory object, containing the average
        """
```

```
        time_stamp = self._generate_time_stamp(final_time, points)
        n = self.model.variables.dimension
        m = self.model.transition_number
        average = np.zeros((len(time_stamp), n))
        # LOOP ON RUNS, count from 1
        update_runs = ceil(runs * update / 100.0)
        c = 0
        for i in range(1, runs + 1):
            c = c + 1
            # updates every 1% of simulation time
            if c == update_runs:
                print(ceil(i * 100.0 / runs), "% done")
                c = 0
            y = self._SSA_single_simulation(final_time, time_stamp, n, m)
            # WARNING, works with python 3 only.
            # updating average
            average = (i - 1) / i * average + y / i
        time_stamp = np.reshape(time_stamp, (len(time_stamp), 1))
        trajectory = Trajectory(time_stamp, average, "SSA average", self.model.
↪variables_names)
        return trajectory

    def MF_simulation(self, final_time, points=1000):
        """
        Numerically integrates standard mean field equations

        :param final_time: final simulation time
        :param points: number of points to be saved
        :return:  a trajectory object for model observables
        """
        t = self._generate_time_stamp(final_time, points)

        '''

        CODE HERE: mean field solution

        '''

        t = np.reshape(t, (len(t), 1))
        trajectory = Trajectory(t, x, "Mean Field", self.model.variables_names)
        return trajectory

    def LN_simulation(self, final_time, points=1000):
        """
        Numerivally integrates the linear noise equations

        :param final_time: final simulation time
```

```
        :param points: number of points to be saved
        :return:  a trajectory object for corrected model observables
        """
        n = self.model.variables.dimension
        t = self._generate_time_stamp(final_time, points)

        '''

        CODE HERE: linear noise solution

        '''

        return
```

[7]:
```
class Trajectory:
    def __init__(self, t, x, desc, labels):
        self.time = t
        self.data = x
        self.labels = labels
        self.description = desc

    def plot(self, var_to_plot=None):
        if var_to_plot is None:
            var_to_plot = self.labels
        fig = plt.figure()
        ax = fig.add_subplot(111)
        ax.set_prop_cycle(plt.cycler('color', ['r', 'g', 'b', 'c', 'm', 'y',␣
 ↪'k']))
        handles = []
        labels = []
        for v in var_to_plot:
            try:
                i = self.labels.index(v)
                h, = ax.plot(self.time, self.data[:, i])
                handles.append(h)
                labels.append(v)
            except:
                print("Variable", v, "not found")
        fig.legend(handles, labels)
        plt.title(self.description)
        plt.xlabel('Time')
        plt.show()

    def plot_comparing_to(self, trajectory, var_to_plot=None):
        if var_to_plot is None:
            var_to_plot = self.labels
        fig = plt.figure()
```

```
        ax = fig.add_subplot(111)
        ax.set_prop_cycle(plt.cycler('color', ['r', 'r', 'g', 'g', 'b', 'b',␣
↪'c', 'c', 'm', 'm', 'y', 'y', 'k', 'k']))
        handles = []
        labels = []
        for v in var_to_plot:
            try:
                i = self.labels.index(v)
                h, = ax.plot(self.time, self.data[:, i])
                handles.append(h)
                labels.append(self.description + " " + v)
                h, = ax.plot(trajectory.time, trajectory.data[:, i], '--')
                handles.append(h)
                labels.append(trajectory.description + " " + v)
            except Exception as e:
                print("Probably variable", v, "not found")
                print("Exception is", e)
        fig.legend(handles, labels)
        plt.title(self.description + " vs " + trajectory.description)
        plt.xlabel('Time')
        plt.show()
```

[ ]:

```

```