CTMC+PCTMC_lab_TotallySolved

May 15, 2020

```
In [1]: import numpy as np
        import scipy.linalg
```

1 A) CTMC

Continuous-time stochastic process $\{X(t), t \geq 0\}$ on a finite state space with Markov property at each time $t \geq 0$. Finite-state CTMC spends an **exponentially distributed** amount of time in a given state before jumping out of it. Thus *exponential distributions* play an important role in CTMCs.

1.1 1. Exponential distribution

Recall that for an exponential random variable $T \sim Exp(\lambda)$: - $E[T] = \frac{1}{\lambda}$ and - $Var[T] = \frac{1}{\lambda^2}$.

Exercise 1.1. Time to failure: Suppose a new machine is put into operation at time zero. Its lifetime is known to be an $Exp(\lambda)$ random variable with $\lambda = 0.1(1/hour)$. What is the probability that the machine will give trouble-free service continuously for 1 day? Suppose the machine has not failed by the end of the first day. What is the probability that it will give trouble-free service for the whole of the next day?

```
In [2]: # Theoretical exercise
        # Pr(X > 24) = exp(-lambda 24)
```

1.1.1 The minimum of independent exponential r.v.

Let T_i be an $Exp(\lambda_i)$ random variable $(1 \leq i \leq k)$, we can think of T_i as the time when an event of type i occurs and suppose $T_1, T_2, \ldots, T_k$ are independent. The r.v. $T = \min\{T_1, T_2, \ldots, T_k\}$, representing the time when the first of these k events occurs, is an exponential r.v. with $\lambda = \sum_{i=1}^{k} \lambda_i$.

Exercise 1.2. Hospital: A hospital currently has seven pregnant women waiting to give birth. Three of them are expected to give birth to boys, and the remaining four are expected to give birth to girls. From prior experience, the hospital staff knows that a mother spends on average 6 hours in the hospital before delivering a boy and 5 hours before delivering a girl. Assume that these times are independent and exponentially distributed. What is the probability that the first baby born is a boy and is born in the next hour?

```
In [3]: # Theoretical exercise
        # T_i ~ Exp(lambda_i) time at which every event of type i occurs
        # T = min{T_1, ... , T_k}time of first event with rate la,bda = sum_i (lambda_i)

        # Boys: 1, 2, 3 Girls: 4, 5, 6, 7
        # B = time of the first boy born
        # G = time of first girl born
        # P(B < G) = rate B / (rate B + rate G)
```

1.2 2. CTMC

Let $X(t)$ be the state of a system at time t.

Random evolution of the system: suppose the system starts in state i. It stays there for an $Exp(q_i)$ amount of time, called the *sojourn time in state i*. At the end of the sojourn time in state i, the system makes a sudden transition to state $j \neq i$ with probability $\pi_{i,j}$, independent of how long the system has been in state i. In case state i is absorbing we set $q_i = 0$.

Analogously to DTMCs, a CTMC can also be represented graphically by means of a directed graph: the directed graph has one node for each state. There is a directed arc from node i to node j if $\pi_{i,j} > 0$. The quantity $q_{i,j} = q_i \pi_{i,j}$, called the transition rate from i to j, is written next to this arc. Note that there are no self-loops. Note that we can recover the original parameters q_i and $\pi_{i,j}$ from the rates $q_{i,j}$:

$$q_i = \sum_j q_{i,j}$$

$$\pi_{i,j} = \frac{q_{i,j}}{q_i}.$$

In each state i, there's a race condition between k edges, each exponentially distributed with rate q_{ij}, the exit rate is $q_{i,i} = -q_i$. The matrix $Q = (q_{ij})_{ij}$ is called **infinitesiaml generator** of the CTMC. Each row sum up to 0.

Exercise 2.1. CTMC class: Define a class CTMC, with a method `__init__()` to initialize the infinitesiamal generator

```
In [4]: import numpy as np
        class CTMC(object):
            '''Continuous-Time Markov Chain class'''

            def __init__(
                    self,
                    Q # infinitesimal generator
            ):

                self.Q = Q
```

Exrcise 2.2. Consider the SIR model and define a function `infinitesimal_generator_SIR()` that takes the rates q_S, q_I, q_R as inputs and returns the infinitesimal generator `Q` after checking that it is well-defined.

```
In [5]: def infinitesimal_generator_SIR(qs, qi, qr, qv):
            #check they are all > 0
            return np.array([[-qi-qv,qi,qv],[0,-qr, qr],[qs,0,-qs]])

        Qsir = infinitesimal_generator_SIR(0.1, 0.2, 0.3, 0.08)
        print(Qsir)
```

```
[[-0.28  0.2   0.08]
 [ 0.   -0.3   0.3 ]
 [ 0.1   0.   -0.1 ]]
```

1.2.1 Jump chain and holding times

We can factorize a CTMC $X(t)$ in a DTMC Y_n called **jump chain**, with probability matrix Π where $\pi_{i,j} = \frac{q_{i,j}}{-q_{i,i}}$ if $i \neq j$ and $\pi_{ii} = 0$.

Exercise 2.3. Infinitesimal generator: Define a function `jump_chain_matrix()` which takes the infinitesimal generator `Q` as input and returns the transition matrix Π.

```
In [6]: def jump_chain_matrix(Q):
            n = Q.shape[0]
            J = np.zeros((n,n))
            for i in range(n):
                for j in range(n):
                    if j != i and Q[i,j] != 0:
                        J[i,j] = -Q[i,j]/Q[i,i];

            return J

        Jsir = jump_chain_matrix(Qsir)
        print(Jsir)
```

```
[[0.         0.71428571 0.28571429]
 [0.         0.         1.        ]
 [1.         0.         0.        ]]
```

Exercise 2.4. Plot trajectories: Recall the Kolmogorov equation and define a function `continuous_trajectories()` taking as input Q, the initial distribution p_0 and a time t. Use a solver for differential equations and return the solution.

Consider one of the models define before and plot the trajectory of each state against time.

```
In [7]: from scipy.integrate import odeint
        import matplotlib.pyplot as plt

        def continuous_trajectories(Q,p0,t):
```

```
        # solving the Kolmogorov diff eq with odeint
        dpdt = lambda p, t: np.dot(p,Q)
        tspan = np.linspace(0,t)
        p = odeint(dpdt, p0, tspan)
        plt.plot(tspan,p)
        plt.xlabel('time')
        plt.ylabel('p(t)')
        plt.show()
        return tspan, p

    t,p = continuous_trajectories(Qsir,[0.5,0.5,0],20)
```

```
<Figure size 640x480 with 1 Axes>
```

```
In [8]: import random
        def sample_discrete(p):
            n = len(p);
            u = np.random.rand(1,1);#usare il punto uno
            for i in range(n):
                S = np.sum(p[0:i+1]);
                if S > u:
                    sampled_index = i
                    break
            return int(sampled_index)
```

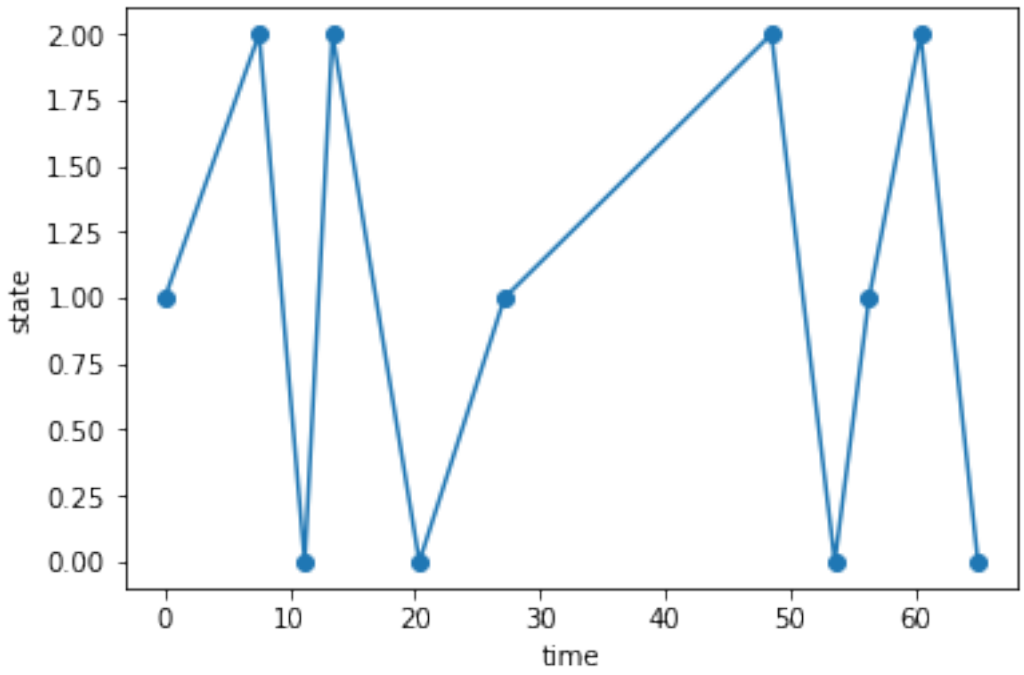
```
In [9]: def simulation_CTMC(Q,p0,n_iter):
            ind = sample_discrete(p0)

            P = jump_chain_matrix(Q)
            traj = np.zeros(n_iter+1) # vector of indexes (ie. states)
            traj[0] = ind
            time = np.zeros(n_iter+1)
            time[0] = 0
            i = 0
            while (i < n_iter):
                exit_rate = - Q[ind,ind]
                ind = sample_discrete(P[ind,:])
                tau = -np.log(np.random.rand(1,1))/exit_rate # holding times
                i += 1
                traj[i] = ind
                time[i] = time[i-1] + tau

            return traj, time
```

```
In [10]: traj,time = simulation_CTMC(Qsir,[0.5,0.5,0],10)
         plt.plot(time,traj,'o-')
         plt.xlabel('time')
```

```
plt.ylabel('state')
plt.show()
```



1.3 3. Poisson Process

Consider systems whose state transitions are triggered by streams of events that occur one at a time and assume that the successive inter-event times are iid exponential random variables. $N_\lambda(0,t)$ is a process that counts how many times an exponential distribution with rate λ has fired from time 0 to time t. It can be seen as a CTMC with $S = \mathbb{N}$ and Q is: $q_{i,i+1} = \lambda$ and zero elsewhere. The time $t_i = t_{i-1} + D_i$, with $D_i \sim Exp(\lambda)$.

Exercise 3.1. Repairing machines: A machine shop consists of N machines and M repair persons ($M \leq N$). The machines are identical, and the lifetimes of the machines are independent $Exp(\mu)$ random variables. When the machines fail, they are serviced in the order of failure by the M repair persons. Each failed machine needs one and only one repair person, and the repair times are independent $Exp(\lambda)$ random variables. A repaired machine behaves like a new machine. Let $X(t)$ be the number of machines that are functioning at time t.

Draw the diagram for $N = 4$ and $M = 2$ and define a function that takes N, M, λ and μ as input and return the infitesimal generator.

```
In [11]: def machine_shop(N, M, repair, lifetime):
             Q1 = np.diag(repair*np.array([2, 2, 2, 1]), k=1)
             Q2 = np.diag(lifetime*np.arange(1,5), k=-1)
             Q = Q1 + Q2
```

```
            for i in range(Q.shape[0]):
                Q[i, i] = - np.sum(Q[i])

            return Q
```

```
In [12]: N = 4
         M = 2

         repair = 0.1
         lifetime = 0.5

         Q_machine = machine_shop(N, M, repair, lifetime)
         print(Q_machine)
```

```
[[-0.2  0.2  0.   0.   0. ]
 [ 0.5 -0.7  0.2  0.   0. ]
 [ 0.   1.  -1.2  0.2  0. ]
 [ 0.   0.   1.5 -1.6  0.1]
 [ 0.   0.   0.   2.  -2. ]]
```

```
In [ ]:
```

Exercise 3.2. Engines of a jet airplane: Draw the diagram for a commercial jet airplane has four engines, two on each wing. Each engine lasts for a random amount of time that is an exponential random variable with parameter λ and then fails. If the failure takes place in flight, there can be no repair. The airplane needs at least one engine on each wing to function properly in order to fly safely. Model this system so that we can predict the probability of a trouble-free flight.

```
In [13]: '''
         S = (00, 01, 02, 10, 11, 12, 20, 21, 22)
         '''

         def airplane(lifetime):
             Q1 = np.diag(lifetime*np.array([1,2,0,1,2,0,1,2]), k = -1)
             Q2 = np.diag(lifetime*np.array([1,1,1,2,2,2]), k = -3)
             Q = Q1 + Q2
             for i in range(Q.shape[0]):
                 Q[i, i] = - np.sum(Q[i])
             return Q
```

```
In [14]: Q_airplane = airplane(0.3)
         print(Q_airplane)
```

```
[[-0.   0.   0.   0.   0.   0.   0.   0.   0. ]
 [ 0.3 -0.3  0.   0.   0.   0.   0.   0.   0. ]
 [ 0.   0.6 -0.6  0.   0.   0.   0.   0.   0. ]
 [ 0.3  0.   0.  -0.3  0.   0.   0.   0.   0. ]
 [ 0.   0.3  0.   0.3 -0.6  0.   0.   0.   0. ]
```

```
[ 0.   0.   0.3  0.   0.6 -0.9  0.   0.   0. ]
[ 0.   0.   0.   0.6  0.   0.  -0.6  0.   0. ]
[ 0.   0.   0.   0.   0.6  0.   0.3 -0.9  0. ]
[ 0.   0.   0.   0.   0.   0.6  0.   0.6 -1.2]]
```

1.4 4. Birth-death process and queues

General class of CTMC on $S = \mathbb{N}$ with birth rate λ_i (from i to $i+1$) and death rate μ_i (from i to $i-1$). When a birth occurs, the process goes from state i to $i+1$. When a death occurs, the process goes from state i to state $i1$.

The birth–death process is the most fundamental example of a **queueing model**. This is a queue with Poisson arrivals, drawn from an infinite population, and C servers with exponentially distributed service time with K places in the queue.

Exercise 4.1. BirthDeath class: Define a general `BirthDeath` class inheriting from the class CTMC in order to initialize the number of states, the rates of forward and backward transitions.

```
In [15]: class BirthDeath(CTMC):
             ''' Birth-Death Process '''

             def __init__(
                     self,
                     num_states,
                     forward,  # forward rate
                     backward,  # backward rate
             ):


                 # set the final element of the forward array and the first element of the bac
                 self.forward = np.append(np.asarray(forward)[:-1], 0)
                 self.backward = np.append(0, np.asarray(backward)[1:])

                 if (self.forward < 0).any() or (self.backward < 0).any():
                     raise ValueError('forward and backward may not be negative.')

                 Q = - np.diag(np.append(self.forward[:-1], 0)) \
                     - np.diag(np.append(0, self.backward[1:])) \
                     + np.diag(self.forward[:-1], 1) \
                     + np.diag(self.backward[1:], -1)

                 super(BirthDeath, self).__init__(
                     Q=Q
                     )
```

Exercise 4.2. Telefone switch: A telephone switch can handle K calls at any one time. Calls arrive according to a Poisson process with rate λ. If the switch is already serving K calls when a new call arrives, then the new call is lost. If a call is accepted, it lasts for an $Exp(\mu)$ amount of time

and then terminates. All call durations are independent of each other. Let $X(t)$ be the number of calls that are being handled by the switch at time t. Model $X(t)$ as a CTMC.

What if the caller can put a maximum of H callers on hold?

```
In [16]: K = 5 #max number of calls

         arrival = 0.1
         service = 0.2

         forward = arrival*np.ones(K)
         backward = service*np.arange(K)
         telephone_switch = BirthDeath(K,forward, backward)

         print(telephone_switch.Q)
```

```
[[-0.1  0.1  0.   0.   0. ]
 [ 0.2 -0.3  0.1  0.   0. ]
 [ 0.   0.4 -0.5  0.1  0. ]
 [ 0.   0.   0.6 -0.7  0.1]
 [ 0.   0.   0.   0.8 -0.8]]
```

```
In [17]: hold = 3
         Kcc = K + hold
         forward_cc = arrival*np.ones(Kcc)

         backward_cc = []
         for i in range(Kcc):
             backward_cc.append(min(i,K))
         backward_cc = service*np.array(backward_cc)

         call_center = BirthDeath(Kcc,forward_cc, backward_cc)

         print(call_center.Q)
```

```
[[-0.1  0.1  0.   0.   0.   0.   0.   0. ]
 [ 0.2 -0.3  0.1  0.   0.   0.   0.   0. ]
 [ 0.   0.4 -0.5  0.1  0.   0.   0.   0. ]
 [ 0.   0.   0.6 -0.7  0.1  0.   0.   0. ]
 [ 0.   0.   0.   0.8 -0.9  0.1  0.   0. ]
 [ 0.   0.   0.   0.   1.  -1.1  0.1  0. ]
 [ 0.   0.   0.   0.   0.   1.  -1.1  0.1]
 [ 0.   0.   0.   0.   0.   0.   1.  -1. ]]
```

Extra (Inventory management): A retail store manages the inventory of washing machines as follows. When the number of washing machines in stock decreases to a fixed number k, an order is placed with the manufacturer for m new washing machines. It takes a random amount of time

for the order to be delivered to the retailer. If the inventory is at most k when an order is delivered (including the newly delivered order), another order for m items is placed immediately. Suppose the delivery times are iid $Exp(\lambda)$ and that the demand for the washing machines occurs according to a $PP(\mu)$. Demands that cannot be immediately satisfied are lost. Model this system as a CTMC.

```
In [18]: delivery = 0.1 # exp(lambda)
         demand = 0.2 #PP(mu)

         def inventory(delivery, demand):
             Q = np.diag(delivery*np.ones(4), k = 2)+np.diag(demand*np.ones(5), k = -1)
             for i in range(Q.shape[0]):
                 Q[i, i] = - np.sum(Q[i])

             return Q

         Q_inventory = inventory(delivery, demand)
         print(Q_inventory)
```

```
[[-0.1  0.   0.1  0.   0.   0. ]
 [ 0.2 -0.3  0.   0.1  0.   0. ]
 [ 0.   0.2 -0.3  0.   0.1  0. ]
 [ 0.   0.   0.2 -0.3  0.   0.1]
 [ 0.   0.   0.   0.2 -0.2  0. ]
 [ 0.   0.   0.   0.   0.2 -0.2]]
```

```
In [ ]:
```

2 B) Population CTMC (PCTMC)

Consider a CTMC model of a population in which each of N individuals can be in a state of state space S. Firing rate may depends on the density of individuals in certain states.

Build a class `PCTMC()` defining: - variables and parameters - initial state x_0 - system size - transitions - transition number;

```
In [19]: from ipynb.fs.full.PCTMC_methods import *
```

```
In [20]: def lotka_volterra():
             lv = Model()
             # Adding variables
             lv.add_variable("A", 50)
             lv.add_variable("B", 100)
             # Adding parameters
             lv.add_parameter("k1", 1)
             lv.add_parameter("k2", 0.005)
             lv.add_parameter("k3", 0.6)
             # setting the system size N
```

```
            lv.set_system_size("N", 1)
            # Adding transitions, using a dictionary to represent the update vector
            lv.add_transition({"A": 1}, "k1*A")
            lv.add_transition({"B": +1, "A": -1}, "k2*A*B")
            lv.add_transition({"B": -1}, "k3*B")
            # adding observables
            lv.add_observable("A", "A")  # syntax: name (to plot) and expression
            lv.add_observable("B", "B")  # syntax: name (to plot) and expression

            # remember to finalize
            lv.finalize_initialization()

            return lv
```

```
In [21]: def gene_regulation():
            grn = Model()
            # variables are generated in the following order
            grn.add_variable("G", 1)
            grn.add_variable("M", 0)
            grn.add_variable("P", 0)
            grn.add_variable("GP", 0)
            # Adding parameters
            grn.add_parameter("km", 1.0)
            grn.add_parameter("kp", 0.005)
            grn.add_parameter("kb", 0.6)
            grn.add_parameter("ku", 0.001)
            grn.add_parameter("dm", 0.0001)
            grn.add_parameter("dp", 0.005)
            # setting the system size N
            grn.set_system_size("N", 1)
            # Adding transitions, using a dictionary to represent the update vector
            grn.add_transition({"M": 1}, "km*G")
            grn.add_transition({"P": 1}, "kp*M")
            grn.add_transition({"G": -1, "P": -1, "GP": 1}, "kb*G*P")
            grn.add_transition({"GP": -1, "G": 1, "P": 1}, "ku*GP")
            grn.add_transition({"M": -1}, "dm*M")
            grn.add_transition({"P": -1}, "dp*P")

            # adding observables: these are the values tracked by the tool
            grn.add_observable("mRNA", "M")  # syntax: name (to plot) and expression
            grn.add_observable("protein", "P")
            grn.finalize_initialization()
            return grn
```

```
In [22]: def client_server_model():
            network = Model()
            # variables are generated in the following order
            network.add_variable("C", 17)
```

```
        network.add_variable("Q", 3)

        # Adding parameters
        network.add_parameter("rt", 1.0)
        network.add_parameter("rs", 20.0)

        # setting the system size N
        network.set_system_size("N", 1)

        # Adding transitions, using a dictionary to represent the update vector
        network.add_transition({"C": -1, "Q": +1}, "rt*C")
        network.add_transition({"Q": -1, "C": +1}, "rs*Q/(0.01 + Q)")

        # adding observables: these are the values tracked by the tool
        network.add_observable("clients", "C")  # syntax: name (to plot) and expression
        network.add_observable("queue", "Q")  # syntax: name (to plot) and expression
        network.finalize_initialization()

        return network
```

Define a class Simulator() with the methods for SSA simulation of trajectories.

2.0.1 Stochastic Simulation - SSA algorithm

Direct method: function taking as input the model, the final time and the max number of transitions.

```
In [27]: class Simulator:
             def __init__(self, model):
                 self.model = model
                 self.t0 = 0
                 self.x0 = model.variables.values

             def _unpack(self, x):
                 n = self.model.variables.dimension
                 phi = x[0:n]  # mean field
                 c = x[n:2 * n]  # c term
                 d = np.reshape(x[2 * n:], (n, n))  # d term
                 return phi, c, d

             def _pack(self, f, dc, dd):
                 x = np.concatenate((f.flatten(), dc.flatten(), dd.flatten()))
                 return x

             # computes observables for
             def _compute_corrected_observables(self, x):
                 p = np.size(x, 0)
                 y = np.zeros((p, self.model.observable_dimension))
```

```
        for i in range(p):
            phi, c, d = self._unpack(x[i])
            h, Dh, D2h = self.model.evaluate_all_observables(phi)
            y[i] = h + (np.matmul(c, Dh.transpose()) + np.tensordot(D2h, d,
                                                                    axes=([1, 2], [0,
        return y

    def _compute_observables(self, x):
        p = np.size(x, 0)
        y = np.zeros((p, self.model.observable_dimension))
        for i in range(p):
            y[i] = self.model.evaluate_observables(x[i])
        return y

    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_nu
        """
        A single SSA simulation run, returns the value of observables

        :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 observables 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
            rates = self.model.evaluate_rates(state)
            # sanity check, to avoid negative numbers close to zero
            rates[rates < 1e-14] = 0.0
            total_rate = sum(rates)
            # check if total rate is non zero.
            if total_rate > 1e-14:
                # if so, sample next time and next state and update state and time
                trans_index = rnd.choice(trans_number, p=rates / total_rate)
                delta_time = rnd.exponential(1 / (self.model.system_size * total_rate)
                time += delta_time
                state = state + self.model.transitions[trans_index].update.flatten() /
            else:
                # If not, stop simulation by skipping to final time
                time = final_time
            # 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
        # computes observables
        y = self._compute_observables(x)
        return y

    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), self.model.observable_dimension))
        # 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 = 0fa
            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.observa
                return trajectory
```

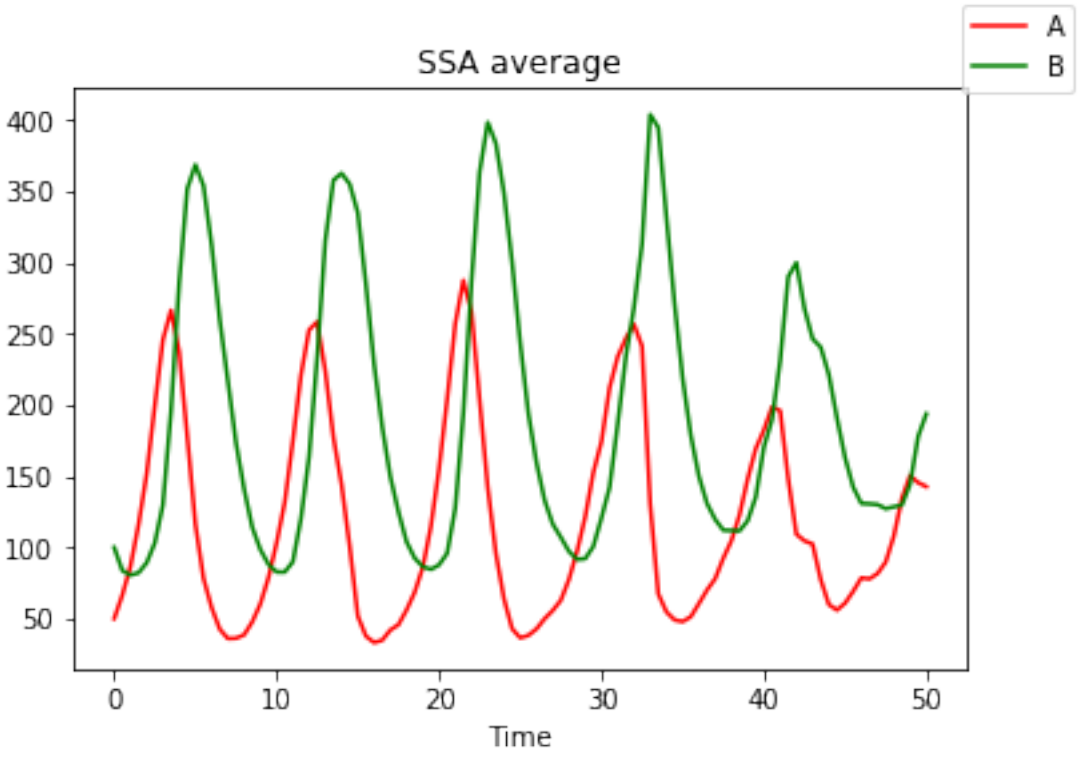
In [32]:
```
lv = lotka_volterra()

# initialize simulator
simulator = Simulator(lv)
final_time = 50
points = 100
runs = 10

# SSA simulation
print("SSA simulation...")
start = time.time()
traj_av_ssa = simulator.SSA_simulation(final_time, runs, points, 10)
end = time.time()-start
print("SSA simulation time:", (end - start), "seconds")
```

```
SSA simulation...
10 % done
20 % done
30 % done
40 % done
50 % done
60 % done
70 % done
80 % done
90 % done
100 % done
SSA simulation time: -1587465796.5131907 seconds
```

In [33]:
```
traj_av_ssa.plot()
```



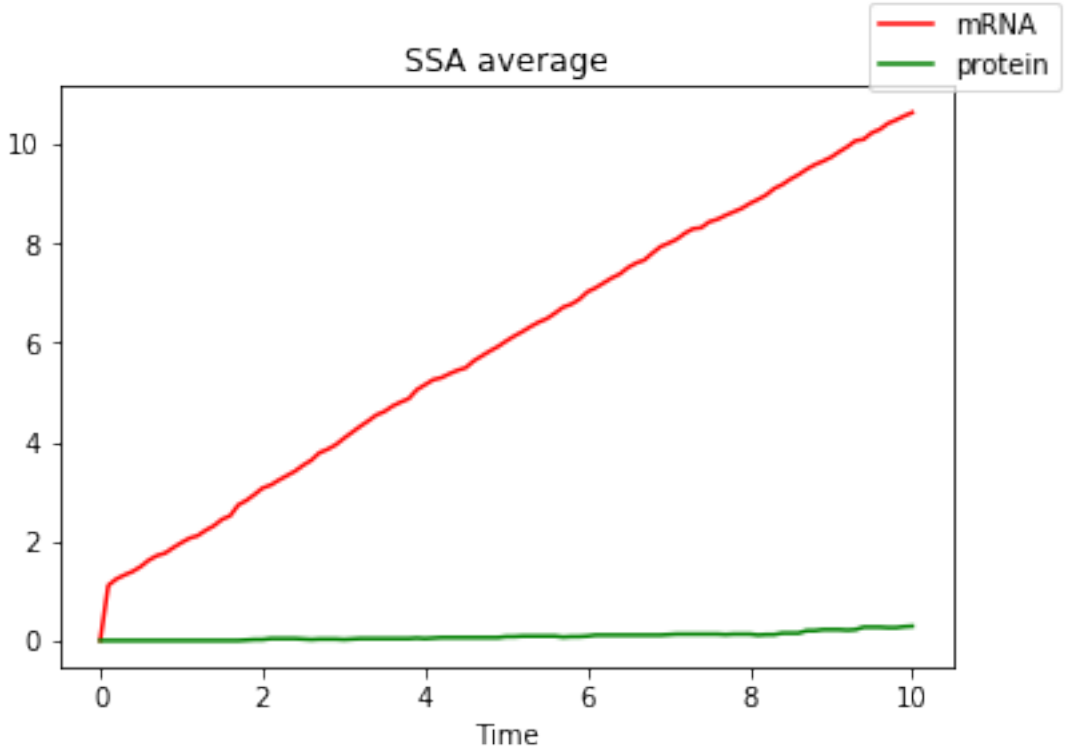

```
In [30]: grn = gene_regulation()

         # initialize simulator
         grn_simulator = Simulator(grn)
         final_time = 10
         points = 100
         runs = 100

         # SSA simulation
         print("SSA simulation...")
         start = time.time()
         traj_grn_ssa = grn_simulator.SSA_simulation(final_time, runs, points, 10)
         end = time.time()-start
         print("SSA simulation time:", (end - start), "seconds")
         traj_grn_ssa.plot()

SSA simulation...
10 % done
20 % done
30 % done
40 % done
50 % done
60 % done
```

```
70 % done
80 % done
90 % done
100 % done
SSA simulation time: -1587465778.837482 seconds
```



In [ ]:

In [ ]: