Stochastic Simulations

Autumn Semester 2024

Prof. Fabio Nobile *Assistant: Matteo Raviola*

Lab 04 – 3 October 2024

Stochastic process generation

Exercise 1

Let $X = [X_1, X_2, ..., X_n] \overset{\text{i.i.d}}{\sim} \mathcal{U}([-1,1]^n)$ be a random vector uniformly distributed over the n-dimensional square $\Gamma = [-1,1]^n$, and define the random variable $Z = \mathbb{1}_{\|X\|_{l^2}<1}$. Observe that

$$I = \mathbb{E}[Z] = \int_\Gamma \mathbb{1}_{\|x\|_{l^2}<1} p(x)\mathrm{d}x = \frac{1}{|\Gamma|}\left|B(0,1)\right|,$$

where $p(x)$ is the PDF of $\mathcal{U}([-1,1]^n)$, and $|B(0,1)|$ is the volume of the n-dimensional sphere with center 0 and radius 1.

1. Let $n = 2$. Use Monte Carlo to approximate the value of I:

$$\overline{I}_N := \frac{1}{N}\sum_{k=1}^{N} Z_k,$$

   For $N = 10, 100, 1000, 10000$, compute $\overline{I}_N$ as well as an approximate confidence interval and compare with the exact value I. In addition, plot the relative error $\frac{|\overline{I}_N - I|}{I}$ versus N in logarithmic scale and verify the convergence rate.

2. (On the choice of N). By *a priori* analysis (knowing that $Z \sim$ Bernoulli(p) with $p = \pi/4$), determine three lower bounds for $N(\alpha, \epsilon)$ with $\epsilon = 10^{-2}$ and $\alpha = 10^{-4}$ for ensuring that

$$\mathbb{P}\left|\overline{I}_N - \pi/4\right| > \epsilon < \alpha$$

   using Chebycheff's inequality (rigorous), the Berry-Esseen Theorem (rigorous) and the leap of faith

$$\frac{\overline{I}_N - \pi/4}{\sqrt{\mathrm{Var}(Z)/N}} \sim N(0,1).$$

   Discuss the advantages and disadvantages of using each bound.

3. An important property of the MC method is that, under very weak regularity assumptions, an $O(N^{-1/2})$ convergence rate holds independently of the dimensionality of the underlying problem. To illustrate this, consider approximating $\mathbb{E}[Z]$ as in the first point, for $n = 6$.

Solution

1. A possible `python` implementation is attached. The results are summarized in Fig. 1

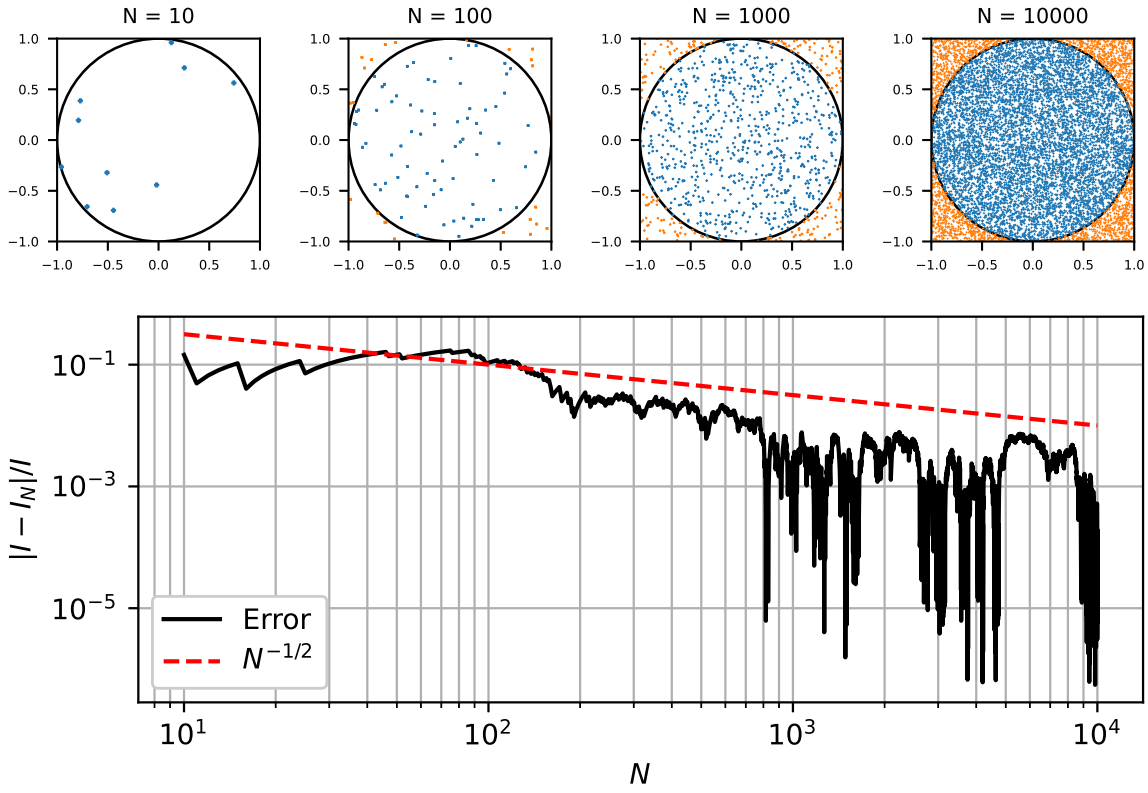

Figure 1: **Top:** Computing the area of a circle using Monte Carlo. **Bottom:** Monte Carlo error as a function of N, for N ranging from 10 to 10000. As we can see, even though there is some *randomness* associated to the rate (which is to be expected), on average, the rate decreases as $N^{-1/2}$

.

In addition, Fig. 2 shows the correct value of I compared with its 95% confidence interval for different values of N.

Python code:

```
import numpy as np
import matplotlib.pyplot as plt
import scipy.stats as st

##### Generate circle plots to illustrate Monte Carlo
Ns = [10,100,1000,10000]
markersizes=[2,1,0.5,0.25]
n = 2
I = np.pi/4 # Area of circle divided by area of square
fig, ax = plt.subplots(1,len(Ns))

for i in range(len(Ns)):
```

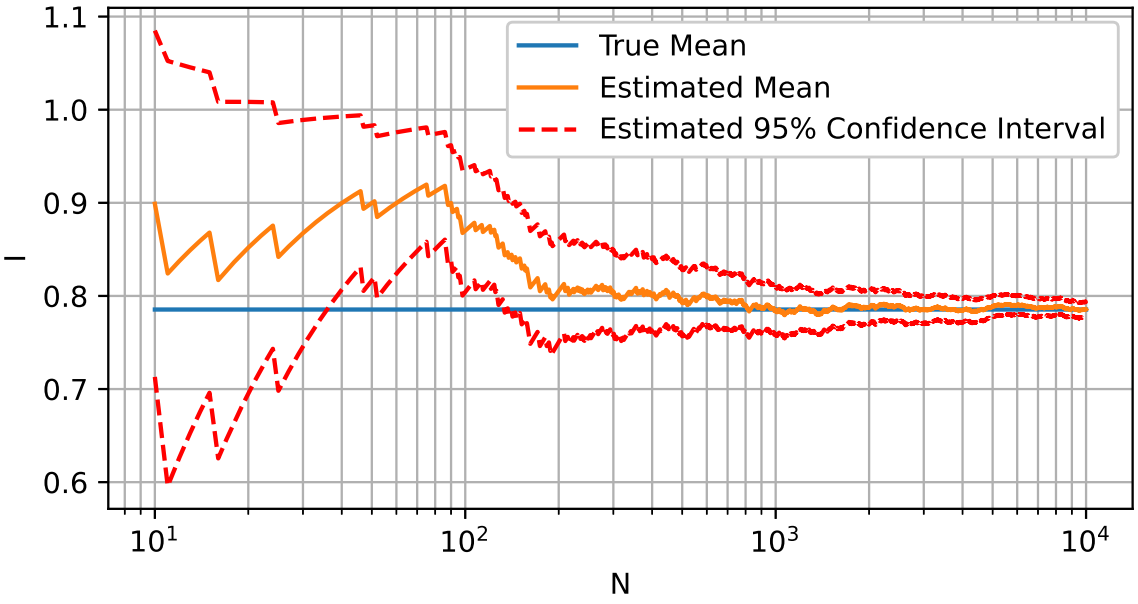

Figure 2: Estimated mean and confidence interval for different sample sizes

```
    circle = plt.Circle((0,0),1,color='black',fill=False)
    N = Ns[i]
    ms = markersizes[i]
    print('Running for N =',N)

    # Generate N samples of n-dimension vectors
    X = st.uniform.rvs(loc=-1,scale=2,size=(N,n))

    # Compute Z, I_N and relative error
    Z = [int(np.linalg.norm(x)<1) for x in X]
    I_N = np.mean(Z)

    # Collect points that are inside and outside
    Xin = [x for (x,z) in zip(X,Z) if z==1]
    Xout = [x for (x,z) in zip(X,Z) if z==0]

    # Plotting
    ax[i].plot([x[0] for x in Xin], [x[1] for x in Xin],'+',markersize=ms)
    ax[i].plot([x[0] for x in Xout], [x[1] for x in Xout],'+',markersize=ms)
    ax[i].set_xlim((-1,1))
    ax[i].set_ylim((-1,1))
    titleString = 'N = %d' %N
    ax[i].set_title(titleString,fontsize=8)
    ax[i].set(aspect='equal')
    ax[i].add_patch(circle)
    ax[i].tick_params(axis='x',labelsize=5)
    ax[i].tick_params(axis='y',labelsize=5)

plt.tight_layout()
plt.savefig('../figures/EX_1_1_MC.eps',bbox_inches='tight')

# Generate additional data for the convergence graph
```

```
Ns_all = list(range(10,10001))
Xs_all = st.uniform.rvs(loc=-1,scale=2,size=(Ns_all[-1],n))
Zs_all = [int(np.linalg.norm(x)<1) for x in Xs_all]
mean = np.mean(Zs_all[:9])
var = np.var(Zs_all[:9])
means = []
variances = []
true_errors = []

for N in Ns_all:
    mean_new = N/(N+1)*mean + Zs_all[N-1]/(N+1)
    var_new = (N-1)/N*var + (Zs_all[N-1]-mean)**2/(N+1)
    means.append(mean_new)
    variances.append(var_new)
    true_errors.append(abs(mean_new-I)/I)

    mean = mean_new
    var = var_new

plt.figure()
plt.loglog(Ns_all, true_errors,color='black',label='Error')
plt.loglog(Ns_all, [N**-0.5 for N in Ns_all],'--',color='red',label=r'$N^{-1/2}$')
plt.xlabel(r'$N$')
plt.ylabel(r'$|I-I_N|/I$')
plt.legend()
plt.grid(which='both')
plt.gca().set_aspect(0.2)
plt.savefig('../figures/EX_1_1_CONV.eps',bbox_inches='tight')

# Plot the confidence intervals
plt.figure()
plt.semilogx([Ns_all[0], Ns_all[-1]], [I,I], label=r'True Mean')
plt.semilogx(Ns_all, means, label=r'Estimated Mean')
upperlim = [(m + st.norm.ppf(0.975)*np.sqrt(v/N)) for m,v,N in zip(means, variances, Ns_all)]
lowerlim = [(m - st.norm.ppf(0.975)*np.sqrt(v/N)) for m,v,N in zip(means, variances, Ns_all)]
plt.semilogx(Ns_all, upperlim,'--',color='red',label=r'Estimated $95\%$ Confidence Interval')
plt.semilogx(Ns_all, lowerlim,'--',color='red')
plt.xlabel(r'N')
plt.ylabel(r'I')
plt.legend()
plt.grid(which='both')
plt.gca().set_aspect(3.0)
plt.savefig('../figures/EX_1_1_CONF.eps',bbox_inches='tight')
plt.show()
```

2. For determining the respective lower bounds for $N(\alpha, \epsilon)$, note first that since $Z \sim$ Bernoulli$(p = \pi/4)$, we have

$$\mathrm{Var}(Z) = \sigma^2 = \mathbb{E}(Z - p)^2 = p(1 - p),$$

and

$$\mathbb{E}|Z - p|^3 = \gamma^3 = (1 - p)^3 p + p^3(1 - p) = p(1 - p)\left[(1 - p)^2 + p^2\right].$$

When using the central limit theorem relation (called "leap of faith" as we are in the

non-asymptotic setting), we may bound the probability of failure by

$$\mathbb{P}\left|\overline{I}_N - I\right| \geq \epsilon = \mathbb{P}\sqrt{N}\frac{\left|\overline{I}_N - I\right|}{\sigma} > \frac{\sqrt{N}\epsilon}{\sigma} = 2\left(1 - \Phi\left(\frac{\sqrt{N}\epsilon}{\sigma}\right)\right).$$

To ensure that

$$2\left(1 - \Phi\left(\frac{\sqrt{N}\epsilon}{\sigma}\right)\right) < \alpha,$$

one must use at least

$$N_{CLT}(\alpha, \epsilon) = \left\lceil \left(\frac{\sigma c_{1-\alpha/2}}{\epsilon}\right)^2 \right\rceil,$$

samples, where $\lceil x \rceil := \min\{z \in \mathbb{Z} \mid z \geq x\}$ and $c_{1-\alpha/2}$ is the $1 - \alpha/2$ quantile of the standard normal distribution, i.e., $\Phi(c_{1-\alpha/2}) = 1 - \alpha/2$.

When using the Chebycheff bound,

$$\mathbb{P}\left|\overline{I}_N - I\right| > \epsilon \leq \mathbb{E}\frac{|Z - \pi|^2}{N\epsilon^2} = \frac{\sigma^2}{N\epsilon^2}.$$

To ensure that $\sigma^2/N\epsilon^2 < \alpha$, we have that

$$N_{CHEB}(\alpha, \epsilon) = \left\lceil \frac{\sigma^2}{\alpha\epsilon^2} \right\rceil,$$

ensures that the probability of failure is bounded by α.

When relying on the Berry–Essén theorem, the probability of failure is controlled by

$$\begin{aligned}\mathbb{P}\left|\overline{I}_N - I\right| > \epsilon = \mathbb{P}\sqrt{N}\frac{\left|\overline{I}_N - \pi\right|}{\sigma} &> \frac{\sqrt{N}\epsilon}{\sigma} \\ &\leq 2\left(1 - \Phi\left(\frac{\sqrt{N}\epsilon}{\sigma}\right)\right) + \frac{2}{\sqrt{N}}\beta K,\end{aligned}$$

where $K \approx 0.4878$ is the constant in the Berry-Essen inequality, and $\beta = \gamma^3/\sigma^3$.

The lower bound solution can then be viewed as the smallest integer which is larger or equal to the zero of the equation

$$f(x) := \left(1 - \Phi\left(\frac{\sqrt{x}\epsilon}{\alpha}\right)\right) + \frac{\beta K}{\sqrt{x}} - \frac{\alpha}{2}.$$

Since $f' < 0$ for all $x > 0$, the zero is unique. We determine the value of N_{BE} in `python` using the bisection method as follows:

```
import numpy as np
import scipy.stats as st
import matplotlib.pyplot as plt
import scipy.optimize as so

p = np.pi/4
```

```
sigma = ( p*(1-p) )**(1/2)
gamma = p*(1-p)*((1-p)**2 + p**2)**(1/3)
K_BE = 0.4748

alpha = 1e-4
eps = 1e-2

# Central Limit theorem bound
N_CLT = int(np.ceil( ( sigma*st.norm.ppf(1-alpha/2)/eps )**2 ))
print("N_CLT \t= ", N_CLT)

# Berry Essen bound
# Note - this interval needs to contain the root of f, otherwise the bisection method
# iterations will fail as coded here.
Nmin = 100000
Nmax = 300000
maxIter = 1000
Ns = [Nmin, Nmax]
f = lambda N : 1-alpha/2-st.norm.cdf(np.sqrt(N)*eps/sigma) \
        + K_BE*gamma**3/sigma**3/np.sqrt(N)
i=1
while True and i < maxIter:
    Nmid = int(np.ceil((Nmin+Nmax)/2))
    if Nmid==Nmax-1 or Nmid==Nmin+1 or Nmid==Nmax or Nmid==Nmin:
        break
    if f(Nmin)*f(Nmid) < 0:
        Nmax = Nmid-1
    elif f(Nmax)*f(Nmid) < 0:
        Nmin = Nmid+1

    else:
        import sys
        sys.exit('Error during bisection method')
    i = i+1

if f(Nmid)<0:
    Nmid=Nmid+1

N_BE = Nmid
print("N_BE \t= ", N_BE)

# Chebyshev bound
N_CHEB = int(np.ceil(  sigma**2/(alpha*eps**2) ) )
print("N_CHEB \t= ", N_CHEB)
```

For the considered problem setting, we report the obtained numbers below and conclude

that

$$N_{CLT} = 25513 \leq N_{BE} = 189737 \ll N_{CHEB} = 16854789.$$

3. We rerun the Monte Carlo simulations for $n = 6$. The results are shown in Fig. 3 and Fig. 4.

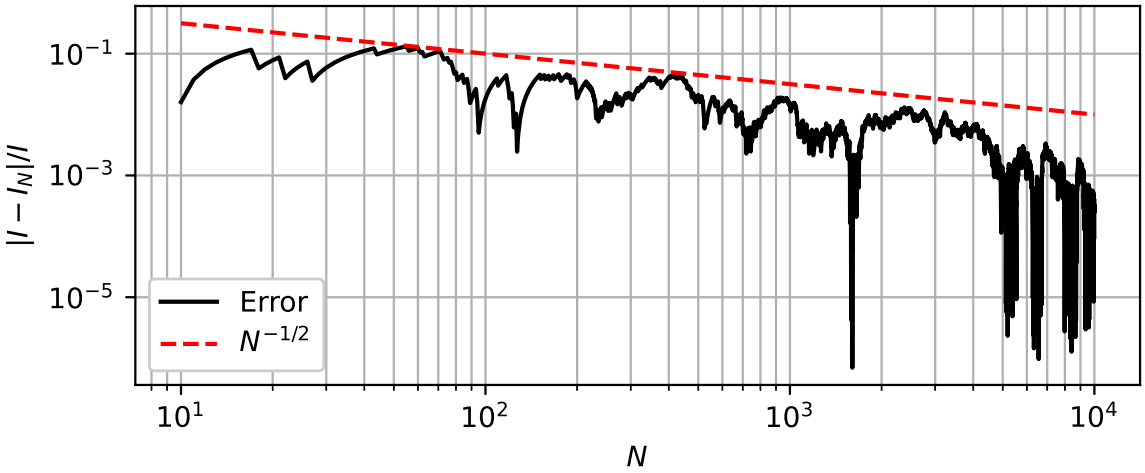

Figure 3: Numerical approximation of the 6-dimensional unit ball volume by respectively the Monte Carlo method

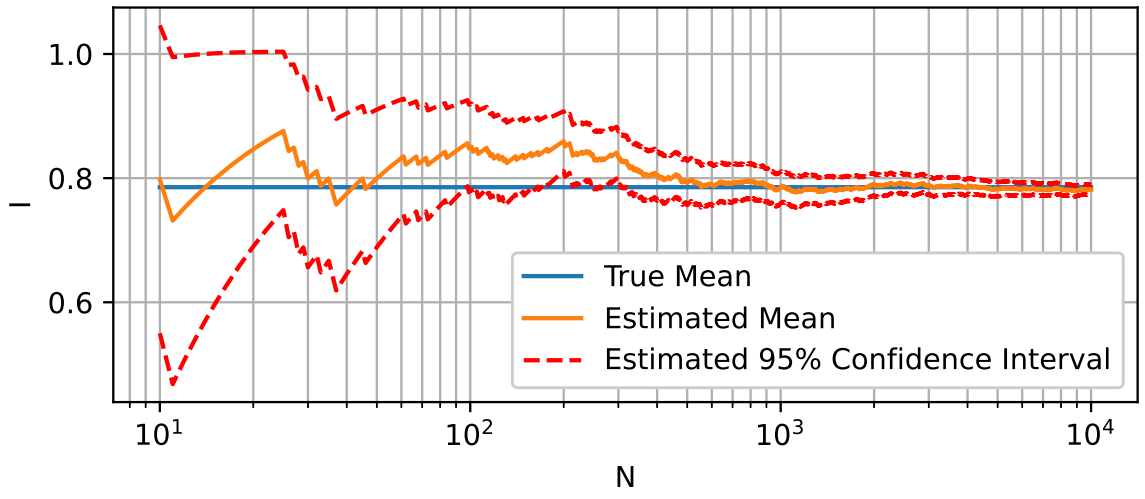

Figure 4: Estimated mean and confidence interval for different sample sizes

We observe that the Monte Carlo method approximates the reference solution with a rate 1/2 even for higher dimensions.

Exercise 2

Note: Refer to Section 4.4 of the lecture notes.
Consider the chemical reactions between three species S_1, S_2, S_3, which are determined by the following four reaction channels:

$$\begin{aligned} S_1 &\xrightarrow{c_1} \emptyset \ , \\ S_1 + S_1 &\xrightarrow{c_2} S_2 \ , \\ S_2 &\xrightarrow{c_3} S_1 + S_1 \ , \\ S_2 &\xrightarrow{c_4} S_3 \ . \end{aligned}$$

To simulate this system, consider the process $\boldsymbol{N}_t = (N_t^1, N_t^2, N_t^3) \in \mathbb{N}_0^3$, where N_t^i denotes the number of molecules of species S_i at time $t \geq 0$. In fact, this process is a time-continuous Markov chain with transition probabilities given by

$$\begin{aligned} \mathbb{P}\big(\boldsymbol{N}_{t+h} = \boldsymbol{N}_{t,1} = (N^1 - 1, N^2, N^3) \big| \boldsymbol{N}_t = = (N^1, N^2, N^3)\big) &= a_1(\boldsymbol{N}_t)h + o(h) \ , \\ \mathbb{P}\big(\boldsymbol{N}_{t+h} = \boldsymbol{N}_{t,2} = (N^1 - 2, N^2 + 1, N^3) \big| \boldsymbol{N}_t = (N^1, N^2, N^3)\big) &= a_2(\boldsymbol{N}_t)h + o(h) \ , \\ \mathbb{P}\big(\boldsymbol{N}_{t+h} = \boldsymbol{N}_{t,3} = (N^1 + 2, N^2 - 1, N^3) \big| \boldsymbol{N}_t = (N^1, N^2, N^3)\big) &= a_3(\boldsymbol{N}_t)h + o(h) \ , \\ \mathbb{P}\big(\boldsymbol{N}_{t+h} = \boldsymbol{N}_{t,4} = (N^1, N^2 - 1, N^3 + 1) \big| \boldsymbol{N}_t = (N^1, N^2, N^3)\big) &= a_4(\boldsymbol{N}_t)h + o(h) \ , \\ \mathbb{P}\big(\boldsymbol{N}_{t+h} = \boldsymbol{N}_{t,5} = (N^1, N^2, N^3) \big| \boldsymbol{N}_t = (N^1, N^2, N^3)\big) &= 1 - h\sum_{j=1}^{4} a_j(\boldsymbol{N}_t) + o(h) \ , \end{aligned}$$

for h sufficiently small, where $\boldsymbol{N}_{t,k}, k \in \{1, ..., 5\}$ indexes the possible transitions. Here, the so-called propensity functions are

$$a_1(\boldsymbol{N}) = c_1 N^1 \ , \quad a_2(\boldsymbol{N}) = c_2 \frac{N^1(N^1 - 1)}{2} \ , \quad a_3(\boldsymbol{N}) = c_3 N^2 \ , \qquad a_4(\boldsymbol{N}) = c_4 N^2 \ ,$$

with $\boldsymbol{N} = (N^1, N^2, N^3)$.

1. Try to construct the transition matrix corresponding to the above transition probabilities and note the challenges. Is it possible to simulate the chemical reaction without the explicit Q matrix? **Hint:** Think back to how you simulated the process in Exercise 2.1.

2. Utilise the following algorithm to simulate the chemical reaction system. Plot a time series for each species' number of molecules for $t \in [0, T]$, $T = 0.2$, for the reaction rates

$$c_1 = 1 \ , \quad c_2 = 5 \ , \quad c_3 = 15 \ , \quad c_4 = \frac{3}{4} \ ,$$

using $\boldsymbol{N}_0 = (400, 800, 0)$ as initial number of molecules. Repeat the simulation for the same reaction rates $c_1, \ldots, c_4$ also for $T = 5$.

Solution

Let $\big\{\boldsymbol{N}_t \in \mathbb{N}_0^3 \colon t \in [0, T]\big\}$ be the Markov jump process that describes the number of each species present in the chemical reaction system. That is, unlike the processes covered during

Algorithm 1: Reaction simulation

1: Set $\boldsymbol{N}_0 = (N_0^1, N_0^2, N_0^3)$, $J_0 = 0$
2: **for** $n = 1, 2, \ldots$ **do**
3: Compute $\lambda = \sum_{j=1}^{4} a_j(\boldsymbol{N}_{J_{n-1}})$
4: Generate $S_n \sim \text{Exp}(\lambda)$ and set $J_n = J_{n-1} + S_n$
5: Generate $I \in \{1, 2, 3, 4\}$ with probability mass function $\mathbb{P}(I = j) = \frac{a_j(\boldsymbol{N}_{J_{n-1}})}{\sum_{l=1}^{4} a_l(\boldsymbol{N}_{J_{n-1}})}$, which is the probability that the j^{th} reaction happens.
6: Set $\boldsymbol{N}_t = \boldsymbol{N}_{J_{n-1}} \forall t \in [J_{n-1}, J_n)$ and $\boldsymbol{N}_{J_n} = \boldsymbol{N}_{t,I}$
7: **end for**

the lecture, we have to deal with both a vector-valued process and the fact that the state space may be unbounded. Regardless of these differences, the Q-matrix could be constructed as usual, namely by $Q = \big(q(\boldsymbol{n}, \boldsymbol{m}),\, \boldsymbol{n}, \boldsymbol{m} \in \mathbb{N}_0^3\big)$, where

$$q(\boldsymbol{n}, \boldsymbol{m}) = \begin{cases} \lim_{h \to 0} \frac{\mathbb{P}(\boldsymbol{N}_{t+h} = \boldsymbol{n} | \boldsymbol{N}_t = \boldsymbol{m})}{h}\,, & \boldsymbol{n} \neq \boldsymbol{m}\,, \\ -\lim_{h \to 0} \frac{1 - \mathbb{P}(\boldsymbol{N}_{t+h} = \boldsymbol{n} | \boldsymbol{N}_t = \boldsymbol{n})}{h}\,, & \boldsymbol{n} = \boldsymbol{m}\,. \end{cases}$$

If one wanted to implement this matrix, then one would, of course, have to truncate the state space appropriately. However, implementing this matrix explicitly is neither needed nor advisable! In fact, the code below shows an exemplary implementation, which is used to produce the times series shown in Figure 5. There first the jump times are generated, before its is determined which reaction takes place.

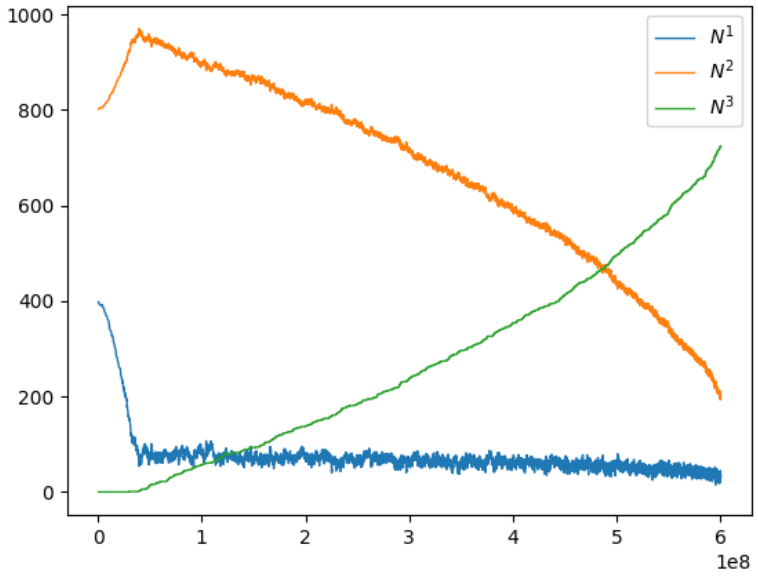

Figure 5: Species' concentration as a function of $t \in [0, 5]$.

Python code

```
import numpy as np
import scipy.stats as st
import matplotlib.pyplot as plt
#defines constant and preallocates
c = np.array([1,5,15,3./4])

x, y, z = 400, 800, 0
xx = np.array([])
yy = np.array([])
zz = np.array([])
tt = np.array([])
t1 = 600000000
t = 0
i = 0
#simmulates dynamics
while t < t1:
    v = np.array([x, x*(x - 1.)/2., y, y])
    a = c * v
    lam = np.sum(a)
    #samples
    p = a / np.sum(a)
    rand = st.uniform.rvs()
    r = np.where(np.cumsum(p) >= rand)[0].min()
    t = t + st.expon.rvs(scale=lam)
    print(t, lam)
    #changes dynamics according to r
    if r == 0:
        x = x - 1
    elif r == 1:
        x, y = x - 2, y + 1
    elif r == 2:
        x, y = x + 2, y - 1
    elif r == 3:
        y, z = y - 1, z + 1

    xx = np.append(xx,x)
    yy = np.append(yy,y)
    zz = np.append(zz,z)
    tt = np.append(tt,t)
    i = i + 1
#plots
plt.plot(tt, xx, linewidth = 1., label = r'$N^1$')
plt.plot(tt, yy, linewidth = 1., label = r'$N^2$')
plt.plot(tt, zz, linewidth = 1., label = r'$N^3$')
plt.legend()
plt.savefig('../py_figures/Ex2.png')
plt.show()
```

Exercise 3

Let $\{N_t \in \mathbb{N}_0 \colon t \geq 0,\, N_0 = 0\}$ be a Poisson process with rate λ.

1. Show that, conditional on the event $\{N_T = n\}$, the jump times $J_1, \ldots, J_n$ have joint density function

$$f_{J_1,\ldots,J_n}(j_1, \ldots, j_n) = n! T^{-n} \mathbb{I}(0 \leq j_1 \leq \cdots \leq j_n \leq T) \; .$$

   In other words, show that conditional on $\{N_T = n\}$, the jump times $J_1, \ldots, J_n$ have the same distribution as an ordered sample of size n from the uniform distribution on $[0, T]$.

   Hints: *Use the joint distribution of the holding times $S_1, \ldots, S_{n+1}$ to first derive the joint distribution of the jump times, where $S_{i+1} = J_{i+1} - J_i$. Then compute the conditional distribution of the jump times given that $N_T = n$, using the fact that $\{N_T = n\} = \{J_n \leq T < J_{n+1}\}$ a.s.*

2. Use the property above to propose an algorithm to generate the process N_t, $t \in (t_1, t_2)$, conditional upon $N_{t_1} = n_1$ and $N_{t_2} = n_2 > n_1$. Such a process is called *Poisson bridge*.

Solution

1. Since the Poisson process $\{N_t \in \mathbb{N}_0 \colon t \geq 0,\, N_0 = 0\}$ is non-decreasing, the condition $N_T = n$ implies that n jumps need to take place. Let's denote these jump times by $J_1, \ldots, J_n$. As a matter of fact, the jump times are related to the $(n+1)$ holding times $S_1, \ldots, S_{n+1}$ by $S_{i+1} = J_{i+1} - J_i$ for $i = 0, \ldots, n$, with the convention that $J_0 = 0$. We know that the holding times are i.i.d. Exp(λ) for some $\lambda > 0$. Consequently, their joint PDF reads

$$f_{S_1,\ldots,S_{n+1}}(s_1, \ldots, s_{n+1}) = \lambda^{n+1} \exp\left(-\lambda \sum_{i=1}^{n+1} s_i\right) \mathbb{I}(s_1, \ldots s_{n+1} \geq 0) \; .$$

   As $\sum_{i=1}^{n+1} S_i = \sum_{i=1}^{n+1} (J_i - J_{i-1}) = J_{n+1}$, the joint PDF of the jumping times $J_1, \ldots, J_{n+1}$ is given by

$$f_{J_1,\ldots,J_{n+1}}(t_1, \ldots, t_{n+1}) = \lambda^{n+1} e^{(-\lambda t_{n+1})} \mathbb{I}(0 \leq t_1 \leq \cdots \leq t_{n+1}) \; .$$

   For any (Borel set) $A \subseteq \mathbb{R}^n$ we thus find

$$\begin{aligned}
&\mathbb{P}\big((J_1, \ldots, J_n) \in A | N_T = n\big) = \frac{\mathbb{P}\big((J_1, \ldots, J_n) \in A \,,\, N_T = n\big)}{\mathbb{P}(N_T = n)} \\
&= \mathbb{P}((J_1, ..., J_n) \in A, J_n \leq T < J_{n+1}) \\
&= \frac{n! T^{-n}}{\lambda^n e^{-\lambda T}} \int_T^\infty \left(\int_A \lambda^{n+1} e^{-\lambda t_{n+1}} \mathbb{I}(0 \leq t_1 \leq ... \leq t_n \leq T) dt_1 dt_2 ... dt_n \right) dt_{n+1} \\
&= \frac{\lambda n! T^{-n}}{e^{-\lambda T}} \int_T^\infty e^{-\lambda t_{n+1}} dt_{n+1} \int_A \mathbb{I}(0 \leq t_1 \leq ... \leq t_n \leq T) dt_1 dt_2 ... dt_n \\
&= n! T^{-n} \int_A \mathbb{I}(0 \leq t_1 \leq \cdots \leq t_n \leq T) \, dt_1 \ldots dt_n \; ,
\end{aligned}$$

   as claimed.

2. A possible `Python` code is shown below.

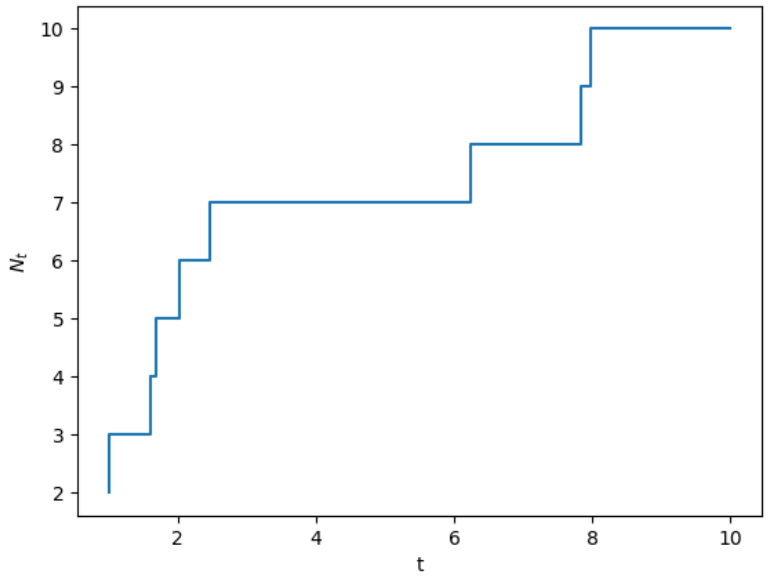

Figure 6: Simulation of the Poisson jump process.

Python code

```
import numpy as np
import scipy.stats as st
import matplotlib.pyplot as plt

#defines some parameters
t1 = 1.
t2 = 10.
dt = t2 - t1
n1 = 2
n2 = 10
#starts the simulation
n = n2 - n1
J = np.sort(dt * st.uniform.rvs(size = (n,)))
t = t1 + np.hstack([0., J])
t = np.hstack([t, t2])
N = n1 + np.array(range(n+1))
N = np.hstack([N,n2])
# Plots
plt.step(t, N)
plt.xlabel('t')
plt.ylabel(r'$N_t$')
plt.savefig('../py_figures/Ex3.png')

plt.show()
```

Exercise 4

Let $\{N_t, t \geq 0, N_0 = 0\}$ be a non-homogeneous Poisson process with rate $\lambda : [0, \infty) \mapsto \mathbb{R}_+$. In addition, define $\Lambda(t) = \int_0^t \lambda(s)\mathrm{d}s$, and let $\{\tilde{N}_t, t \geq 0, \tilde{N}_0 = 0\}$ be a homogeneous Poisson process with rate one.

1. Show that the non-homogeneous Poisson process can be obtained as $N_t = \tilde{N}_t \circ \Lambda(t)$, i.e., $N_t = \tilde{N}_{\Lambda(t)}$.
2. Simulate a non-homogeneous Poisson process with rate function $\lambda(t) = \sin^2(t)$ on the interval $[0, 50]$.

Exercise 5 (Optional)

1. Generate a random walk $\{X_n \in \mathbb{Z},\, n \in \mathbb{N}_0,\, X_0 = 0\}$ with transition probabilities

   $$\mathbb{P}(X_{n+1} = j | X_n = j-1) = \mathbb{P}(X_{n+1} = j | X_n = j+1) = a\ , \quad \mathbb{P}(X_{n+1} = j | X_n = j) = 1-2a\ ,$$

   for some $0 < a \leq 1/2$.
2. Consider the rescaled process $Y_{t_i} := \sqrt{\Delta t/(2a)} X_i$ for $i = 0, \ldots, n$ with $t_i = i\Delta t$. Compare this process with the process W_{t_i}, $i = 0, \ldots, n$, where W_t denotes a Wiener process with $W_0 = 0$. That is, show that both processes "look similar" in the limit as $\Delta t \to 0$ by plotting multiple realizations of both processes for $n = \lceil 1/\Delta t \rceil$.
3. (**Optional:**) More theoretical analysis of the observed phenomenon:
   (a) Consider the spatial mesh $x_m = m\Delta x = m\sqrt{\Delta t/(2a)}$ for $m \in \mathbb{Z}$ and the following notation for the rescaled process' probability mass function at time t_i:

   $$\bar{u}(t_i, x_m) := \mathbb{P}(Y_{t_i} = x_m | Y_0 = 0), \quad m \in \mathbb{Z}, i = 0, 1, \ldots$$

   Use the discrete Chapman–Kolmogorov formula

   $$\mathbb{P}(Y_{t_{i+1}} = x_m | Y_0 = 0) = \sum_k \mathbb{P}(Y_{t_{i+1}} = x_m | Y_{t_i} = x_k) \mathbb{P}(Y_{t_i} = x_k | Y_0 = 0) \qquad (1)$$

   to derive a difference equation for $\bar{u}(t_{i+1}, x_m)$ in terms of $\bar{u}(t_i, \cdot)$.
   (b) Show that the difference equation obtained in 3a corresponds to a finite difference approximation of the one dimensional heat equation

   $$u_t(t, x) = \frac{u_{xx}(t, x)}{2}, \quad x \in \mathbb{R}, t > 0,$$

   on a uniform grid $x_i = i\Delta x$ and $t_j = j\Delta t$ with $\Delta t = 2ax^2$, using a second order centered finite difference stencil in space and a first order forward Euler scheme in time.

(c) For the standard Wiener process with $\mathbb{P}(W_0 = 0) = 1$, we denote the probability density function at time $t > 0$ by

$$u(t, x) := \frac{e^{-x^2/(2t)}}{\sqrt{2\pi t}}, \qquad x \in \mathbb{R}.$$

For all $t > 0$ and $x \in \mathbb{R}$, show that the density satisfies the same heat equation introduced in point 3b.

Solution

1.–2. A possible `Python`implementation for both parts is shown at end of this section. Figure 7 illustrates typical realizations of both the Wiener process W_t and the rescaled random walk Y_t for two values of Δt. We observe that, as $\Delta t \to 0$, both processes "look similar" in the sense that one could not identify the rescaled random walk Y_t, if it wasn't for the legend. Furthermore, similarly to the Wiener process, one can easily show that the increments of the rescaled Random Walk are independent with variance equal to $\sqrt{\Delta t}$.

3. (a) We have that,

$$\begin{aligned}
\bar{u}(t_{i+1}, x_m) &= \mathbb{P}(Y_{t_{i+1}} = x_m | Y_0 = 0) \\
&= \sum_{k\in\mathbb{Z}} \mathbb{P}(Y_{t_{i+1}} = x_m | Y_{t_i} = x_k)\mathbb{P}(Y_{t_i} = x_k | Y_0 = 0) \\
&= \sum_{k\in\mathbb{Z}} \mathbb{P}(Y_{t_{i+1}} = x_m | Y_{t_i} = x_k)\bar{u}(t_i, x_k) \\
&= (1 - 2a)\bar{u}(t_i, x_m) + a\bar{u}(t_i, x_{m-1}) + a\bar{u}(t_i, x_{m+1}) \qquad (2)
\end{aligned}$$

(b) Using the Taylor series for the solution of the heat equation, we have that

$$\begin{aligned}
u(t_i, x_{m-1}) &= u(t_i, x_m) - \Delta x u_x(t_i, x_m) + \frac{\Delta x^2}{2} u_{xx}(t_i, x_m) + \mathcal{O}(\Delta x^4), \\
u(t_i, x_{m+1}) &= u(t_i, x_m) + \Delta x u_x(t_i, x_m) + \frac{\Delta x^2}{2} u_{xx}(t_i, x_m) + \mathcal{O}(\Delta x^4), \\
u(t_{i+1}, x_m) &= u(t_i, x_m) + \Delta t u_t(t_i, x_m) + \mathcal{O}(\Delta t^2).
\end{aligned}$$

With the choice $\Delta x = \sqrt{\Delta t/2a}$, this leads to

$$\begin{aligned}
u(t_{i+1}, x_m) &= u(t_i, x_m) + \frac{\Delta t}{2} u_{xx}(t_i, x_m) + \mathcal{O}(\Delta t^2) \\
&= u(t_i, x_m) + \frac{\Delta t}{2} \frac{u(t_i, x_{m-1}) + u(t_i, x_{m+1}) - 2u(t_i, x_m)}{\Delta x^2} + \mathcal{O}(\Delta t^2 + \Delta x^4) \\
&= (1 - 2a)u(t_i, x_m) + au(t_i, x_{m-1}) + u(t_i, x_{m+1}) + \mathcal{O}(\Delta t^2),
\end{aligned}$$

which corresponds to Eq. (2), up to error terms that vanish as $\Delta t \to 0$.

(c) This can be seen by simply substituting the density into the standard heat equation.

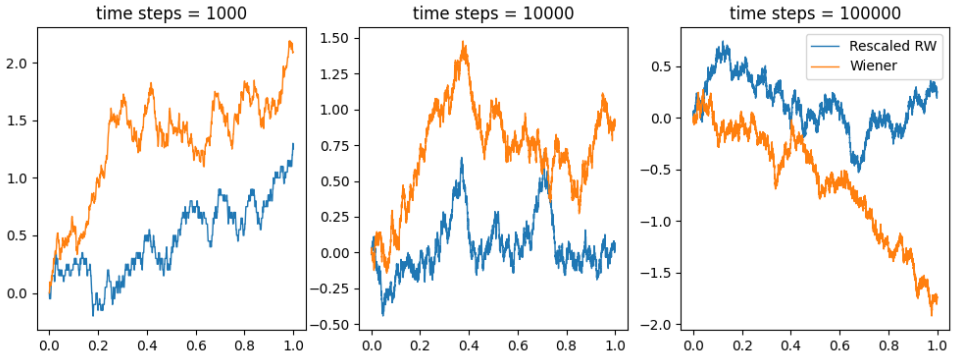

Figure 7: Realizations of Y_t and W_t for different dt.

Python code

```
import numpy as np
import scipy.stats as st
import matplotlib.pyplot as plt

#defines the standard random walk
def RW(steps = 100, a = 1./2):
    assert a <= 1./2, 'Parameter a should be at most 0.5.'
    N = steps
    X = np.zeros(N+1)
    for i in range(1, N+1):
        U = st.uniform.rvs()
        X[i] = X[i-1] + (U <= a) - (a < U) * (U <= 2*a)
    return X

#defines the rescaled random walk
def rescaledRW(T = 1., n = 100, a = 0.2):
    X = RW(n, a)
    dt = T / (n+1)
    return np.sqrt(dt / (2*a)) * X

#defines the Wiener process
def Wiener(T = 1., n = 100):
    W = np.zeros(n+1)
    dt = T / (n+1)
    for i in range(1,n+1):
        W[i] = W[i-1] + np.sqrt(dt) * st.norm.rvs()
    return W

#X = rescaledRW(n = 1000, a = 0.2)
#W = Wiener(n = 1000)
fig, axes = plt.subplots(nrows = 1, ncols = 3, figsize = (12,4))
i = 1
for ax in axes:
    n = 100*(10**i)
    print('running for n = ', n)
```

```
    X = rescaledRW(n = n, a = 0.2)
    W = Wiener(n = n)
    t = np.linspace(0., 1, n+1)
    ax.plot(t, X, linewidth = 1, label = 'Rescaled RW')
    ax.plot(t, W, linewidth = 1, label = 'Wiener')
    ax.set_title('time steps = '+str(n))
    i = i + 1
plt.legend()
plt.savefig('../py_figures/Ex1.png')
plt.show()
```