Stochastic Simulation
Professor: Prof. Fabio Nobile
Assistant: Eliott Van Dieren

Week of 2025-09-22 to 2025-09-26

Lab 3 of Thursday 25th September 2025

Exercice 1.

Consider a multivariate Gaussian random variable $X = (X_1, X_2, \dots, X_n)^T \sim \mathcal{N}(\mu, \Sigma)$, with mean $\mu \in \mathbb{R}^n$ and covariance matrix $\Sigma \in \mathbb{R}^{n\times n}$.

1) Generate a sample of $N = 10^6$ independent random vectors X_i, $1 \leqslant i \leqslant N$, each X_i following a $\mathcal{N}(\mu, \Sigma)$ distribution with

$$\mu = \begin{pmatrix} 2 \\ 1 \end{pmatrix} \quad \text{and} \quad \Sigma = \begin{pmatrix} 1 & 2 \\ 2 & 5 \end{pmatrix}.$$

   Specifically, use the Cholesky decomposition (`numpy.linalg.cholesky()` in Python[1]) to compute the factor A, such that $\Sigma = AA^t$. Generate the standard normal vectors $Y_i \sim \mathcal{N}(0, I_{2\times 2}), 1 \leqslant i \leqslant N$, possibly using Numpy's `np.random.randn()` function[2]. Then generate the X_i as $X_i = \mu + AY_i$. Assess the quality of the samples by plotting a bivariate histogram[3].

2) Propose a method for generating Gaussian random variables $X \sim \mathcal{N}(\mu, \Sigma)$ with covariance matrix $\Sigma = \left(\begin{smallmatrix} 1 & 2 \\ 2 & 4 \end{smallmatrix}\right)$ and mean μ as before. Test your method by generating $N = 10^6$ independent copies of the random vector and plot a bivariate histogram. Compare the outcomes with the previous point and explain the differences.

Solution

The difference between part 1 and part 2 of this exercise is that the covariance matrix Σ in part 1 is regular, while it is singular in part 2. Consequently, a Cholesky decomposition via `numpy.linalg.cholesky` is not possible in part 2 and an eigenvalue decomposition (e.g. via `numpy.linalg.svd`) has to be used. A possible `Python` code is attached.

[1] https://docs.scipy.org/doc/numpy/reference/generated/numpy.linalg.cholesky.html
[2] https://docs.scipy.org/doc/numpy/reference/generated/numpy.random.randn.html
[3] https://docs.scipy.org/doc/numpy/reference/generated/numpy.histogram2d.html

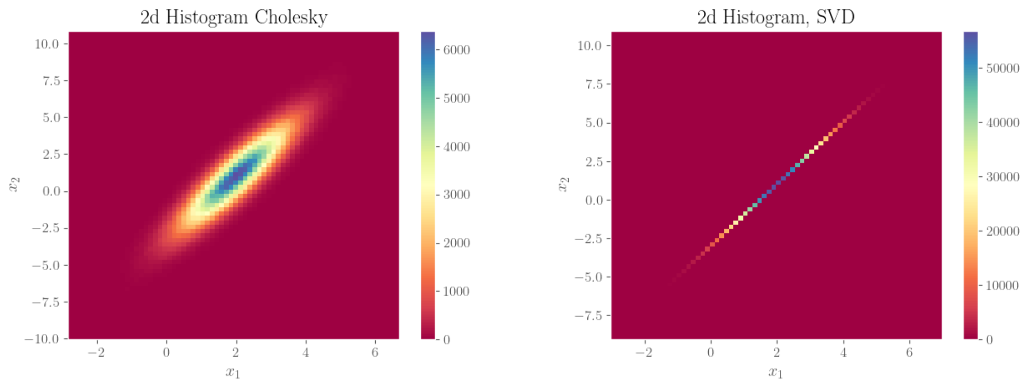

Figure 1: (Left) 2D histogram of the samples obtained with the Cholesky factorization.(Right) 2D histogram of the samples obtained with the SVD

Python code:

```
import numpy as np
import matplotlib.pyplot as plt
import scipy.linalg as linalg
import time
from matplotlib import rc

# graphical parameters
##################################################################################
plt.style.use("ggplot")
rc("font", **{"family": "serif", "serif": ["Computer Modern Roman"], "size": "12"})
rc("text", usetex=True)
rc("lines", linewidth=2)
plt.rcParams["axes.facecolor"] = "w"
##################################################################################

np.random.seed(42)

N = int(1e6)
mu = np.array([2, 1])
Sigma = np.array([[1, 2], [2, 5]])
p = 2

# ### Part 1
#
# Generates samples based on the Cholesky factorization

mu = np.ones((p, N))
mu[0, :] = 2.0 * np.ones(N)
mu[1, :] = 1.0 * np.ones(N)

tstart = time.time()
A = np.linalg.cholesky(Sigma)
xi = np.zeros([2, N])
xi[0, :] = np.random.standard_normal(N)
xi[1, :] = np.random.standard_normal(N)
X = mu + A @ xi
telapsed = time.time() - tstart
print("time needed  " + str(telapsed) + " " + "seconds")
```

```python
# Computes a 2D histogram

plt.figure(1)
plt.hist2d(X[0, :], X[1, :], bins=70, cmap="Spectral")
plt.colorbar()
plt.xlabel("$x_1$")
plt.ylabel("$x_2$")
plt.title("2d Histogram Cholesky")
plt.show()
# ### part 2
#
#
# Now we try to the same with the second problem. Note that the Cholesky
# decomposition for $\Sigma=\begin{bmatrix}1 & 2\\2 & 4\end{bmatrix}$ fails.
# Thus, we do an eigenvalue decomposition (e.g. via svd):

Sigma = np.array([[1, 2], [2, 4]])
tstart = time.time()
# Computes the SVD
U, s, Vh = linalg.svd(Sigma)
A = U * np.sqrt(s)
xi = np.zeros([2, N])
xi[0, :] = np.random.standard_normal(N)
xi[1, :] = np.random.standard_normal(N)
X = mu + A @ xi
telapsed = time.time() - tstart
print("time needed  " + str(telapsed) + " " + "seconds")

plt.figure(2)
plt.hist2d(X[0, :], X[1, :], bins=70, cmap="Spectral")
plt.colorbar()
plt.xlabel("$x_1$")
plt.ylabel("$x_2$")
plt.title("2d Histogram, SVD")
plt.show()
```

Exercice 2.

Consider a Gaussian process $\{X_t\ ,\ t \in I\}$ on $I = [0, 1]$ with mean function $\mu_X \colon I \to \mathbb{R}$,

$$\mu_X(t) \equiv \mathbb{E}[X_t] = \sin(2\pi t)\ ,$$

and covariance function $C_X \colon I \times I \to \mathbb{R}$,

$$C_X(t, s) \equiv \mathbb{E}[(X_t - \mu_X(t))(X_s - \mu_X(s))] = e^{-|t-s|/\rho}\ ,$$

where $\rho > 0$.

1) Generate the Gaussian process in a set of n points $t_1, \dots, t_n \in I$. Plot the resulting point values of the random process for various values of n and ρ. Implement both direct generation and circulant embedding method with FFT. Compare the costs varying n which should behave as $\mathcal{O}(n^3)$ and $\mathcal{O}(n \log n)$, respectively.

2) Generate the Gaussian process for $\rho = 1/200$ on a uniform partition of $I = [0, 1]$ with $n = 51$ points, i.e. $t_i = \frac{i-1}{n-1}$ for $i = 1 \dots, n$. Let's denote this collection of point-wise evaluations of $\{X_t,\, t \in I\}$ by Z_n. Then generate $m = n - 1 = 50$ additional point evaluations of the Gaussian process in new points $t_{n+1}, \dots, t_{n+m}$ by a uniform grid refinement (i.e. $t_{n+j} = \frac{2j-1}{2(n-1)}$

for $j = 1, \ldots, m = n-1$), denoted by Y_m, conditioned upon the previously generated ones Z_n. Specifically, use the results for conditioned multivariate Gaussian random variables discussed in the lecture notes.

Solution

A possible `Python` implementation is attached. This implementation uses the second algorithm for conditional Gaussian sampling presented during the lecture.

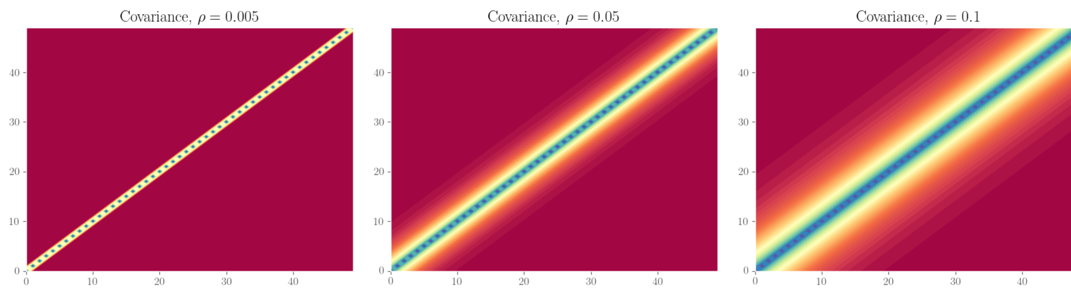

Figure 2: Different covariance matrices for different values of ρ. Notice that for $N = 50$, the covariance matrix becomes singular for relatively large ($\rho \geqslant 0.05$) values of ρ.

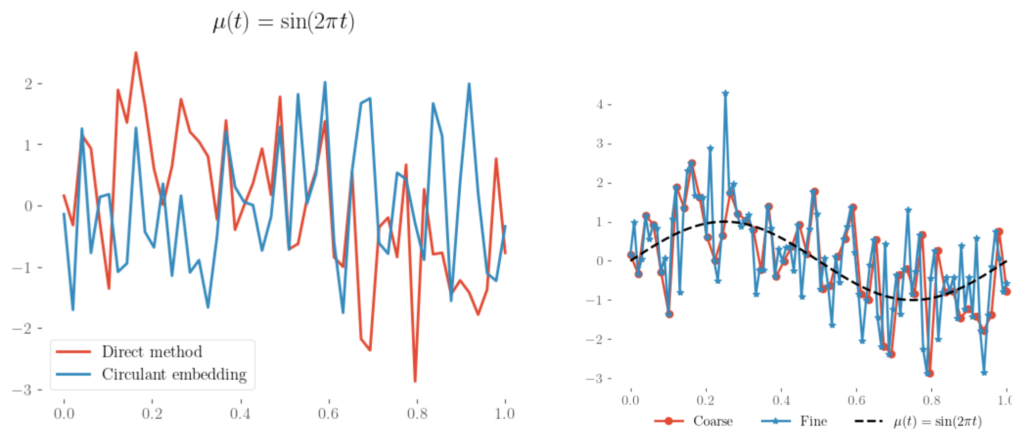

Figure 3: Samples(red) and their refinement (blue) along with mean (black).

Python code:

```
import numpy as np
import scipy as sp
import matplotlib.pyplot as plt
from matplotlib import rc
from scipy.fftpack import fft, ifft
import time

# graphical parameters
###############################################################################
plt.style.use("ggplot")
rc("font", **{"family": "serif", "serif": ["Computer Modern Roman"], "size": "12"})
```

```
rc("text", usetex=True)
rc("lines", linewidth=2)
plt.rcParams["axes.facecolor"] = "w"
##############################################################################


np.random.seed(42)

# defines some functions first
def mu(t):
    return np.sin(2 * np.pi * t)


def Cov(t, s, rho=1 / 20):
    return np.exp(-np.abs(t - s) / rho)


# defines some parameters
N = 50
dt = 1 / N
t = np.linspace(0, 1, N)
muvec = mu(t)  # gets mean
sigma = np.zeros((N, N))
rho = 0.005
# notice that if we increase rho, the covariance becomes singular,
# try changing rho to 1/10

# Populates covatiance matrix
for i in range(N):
    for j in range(N):
        sigma[i, j] = Cov(t[i], t[j], rho)

print(np.linalg.det(sigma))  # checks if its singular
plt.contourf(sigma, 50, cmap="Spectral")
plt.title(r"Covariance, $\rho=" + str(rho) + "$")
plt.show()

U, D, V = np.linalg.svd(sigma)  # we do this since depending on the value of rho,
# \Sigma can become numerically singular
A = U @ np.diag(np.sqrt(D))

# samples random process with direct generation
proc = muvec + A @ np.random.standard_normal(N)
plt.plot(t, proc, label="Direct method")

# Samples random process with circulant embedding
# Construct FFT
c = np.hstack([sigma[0, :], sigma[0, -2:0:-1]])
labda = fft(c)

# Generate iid normal vectors and do IFFT
YR = np.random.standard_normal(2 * (N - 1))
YI = np.random.standard_normal(2 * (N - 1))
Y = np.array([complex(yr, yi) for yr, yi in zip(YR, YI)])
eta = np.sqrt(2 * (N - 1)) * np.sqrt(labda)
print(eta * Y)
Xtilde = ifft(eta * Y)
proc_circ = np.real(Xtilde[1 : N + 1])

plt.plot(t, proc_circ, label="Circulant embedding")
plt.title(r" $\mu(t)=\sin(2\pi t)$")
plt.legend()
plt.show()
```

```python
# adds new data by adding one midpoint
tall = np.linspace(0, 1, N * 2)
Nall = len(tall)
M = Nall - N

# transforms to layout X=(Y,Z) where Z is the previous observation
zobs = proc
# gets the indices
idZ = np.arange(0, Nall, 2)
idY = np.arange(1, Nall, 2)
idP = np.concatenate((idY, idZ))
P = np.eye(Nall)
P = P[idP, :]
# mu for the fine process
mu_all = mu(tall)
sigma_all = np.zeros((Nall, Nall))

for i in range(Nall):
    for j in range(Nall):
        sigma_all[i, j] = Cov(tall[i], tall[j], rho)

sigmaX = P @ sigma_all @ np.transpose(P)
muX = P @ mu_all
# extracts the components
muY = muX[0:M]
muZ = muvec

# constructs sub covariance matrices
sigmaYY = sigmaX[:M, :M]
sigmaYZ = sigmaX[:M, M:]
sigmaZZ = sigmaX[M:, M:]

# computes conditional mean and variance

muYgZ = muY + sigmaYZ @ (np.linalg.solve(sigmaZZ, zobs - muZ))
sigmaYgZ = sigmaYY - sigmaYZ @ (np.linalg.solve(sigmaZZ, sigmaYZ.T))

# samples again
U, D, V = np.linalg.svd(sigmaYgZ)
A = U @ np.diag(np.sqrt(D))
Y = muYgZ + A @ np.random.standard_normal(M)
X = np.concatenate((Y, zobs))
proc_all = np.linalg.solve(P, X)

tg = np.linspace(0, 1, 10 * N)

plt.plot(t, proc, "-o")
plt.plot(tall, proc_all, "-*")
plt.plot(tg, mu(tg), "--", color="k")
plt.legend(
    ["Coarse", "Fine", r"$\mu(t)=\sin(2\pi t)$"],
    fancybox=True,
    framealpha=0.0,
    loc="upper center",
    bbox_to_anchor=(0.5, -0.05),
    shadow=False,
    ncol=3,
)
plt.savefig("../figures/Ex02.png")
plt.show()

rho = 0.005
```

```python
# Compare direct method and circulant embedding costs
N_grid = np.arange(2000, 5000, 500)
num_rep = 10
times_dir = []
times_circ = []

for N in N_grid:
    dt = 1 / N
    t = np.linspace(0, 1, N)
    muvec = mu(t)  # gets mean
    sigma = Cov(
        t.reshape(-1, 1), t.reshape(1, -1)
    )  # faster way to populate covariance matrix

    time_dir = 0
    time_circ = 0
    for i in range(num_rep):
        # Samples random process with direct generation
        start = time.time()
        U, D, V = sp.linalg.svd(sigma)
        A = U @ np.diag(np.sqrt(D))
        proc = muvec + A @ np.random.standard_normal(N)
        end = time.time()
        time_dir += end - start

        # Samples random process with circulant embedding
        c = np.hstack([sigma[0, :], sigma[0, -2:0:-1]])
        start = time.time()
        labda = fft(c)
        YR = np.random.standard_normal(2 * (N - 1))
        YI = np.random.standard_normal(2 * (N - 1))
        Y = np.array([complex(yr, yi) for yr, yi in zip(YR, YI)])
        eta = np.sqrt(2 * (N - 1)) * np.sqrt(labda)
        Xtilde = ifft(eta * Y)
        proc_circ = np.real(Xtilde[1 : N + 1])
        end = time.time()
        time_circ += end - start
    times_dir.append(time_dir / num_rep)
    times_circ.append(time_circ / num_rep)

plt.plot(N_grid, times_dir, label="Direct method")
plt.plot(N_grid, times_circ, label="Circulant embedding")
xx = np.log(N_grid)
yyd = np.log(times_dir)
yyc = np.log(times_circ)
aad, bbd = np.polyfit(xx, yyd, deg=1)
aac, bbc = np.polyfit(xx, yyc, deg=1)
plt.plot(
    np.exp(xx),
    np.exp(aad * xx + bbd),
    "-.",
    label=r"$\mathcal{O}(N^{" + str(round(aad, 2)) + "})$",
    color="gray",
)
plt.plot(
    np.exp(xx),
    np.exp(aac * xx + bbc),
    "--",
    label=r"$\mathcal{O}(N^{" + str(round(aac, 2)) + "})$",
    color="gray",
)

plt.yscale("log")
```

```
plt.xscale("log")
plt.title("Cost")
plt.legend()
plt.show()
```

Exercice 3.

The lecture notes introduce a Brownian bridge as a Wiener process $\{W_t\ ,\ t \in [0,1]\}$ conditioned upon $W_1 = b$. Derive a generalized Brownian bridge $\{X_t\ ,\ t \in [0,1]\}$ that is given as the Wiener process conditioned on $W_0 = a$ and $W_1 = b$ and generate realizations of this Brownian bridge at $0 = t_0 < t_1 < \cdots < t_n < t_{n+1} = 1$. Specifically, carry out the following exercises:

1) Show that
$$\mu_X(t) = a + (b-a)t$$
and
$$C_X(t,s) = \min\{s,t\} - st\ .$$

2) Propose and implement an iterative algorithm that generates X_{t_i} conditioned upon $X_{t_{i-1}}$ and $X_{t_{n+1}} = b$.

Solution

As usual, a possible `Python` code is attached.

1) We know from the lecture that $Y_t = W_t - t(W_{t_{n+1}} - b)$ is a Brownian bridge attaining the value b at final time $t_{n+1} = 1$, where W_t is the Wiener process. We thus simply set
$$X_t := (1-t)a + Y_t = a + W_t - t(W_{t_{n+1}} + a - b)\ .$$
In fact, it is easy to see that $\mathbb{E}(X_t) = a + (b-a)t$, while the covariance follows immediately form the observation that $X_t - \mathbb{E}(X_t) = W_t - tW_{t_{n+1}} = Y_t|_{b=0}$.

2) If we wanted to generate this generalized Brownian bridge given a certain grid $0 = t_0 < t_1 < \cdots < t_n < t_{n+1} = 1$, then we could simply add the term $(1-t)a$ to the algorithm presented in the lecture notes. In this part of the exercise however, we aim at constructing an iterative algorithm that generates X_{t_i} conditioned upon $X_{t_{i-1}}$ and $X_{t_{n+1}} = b$. That is, we start with $X_0 = a$ and we then want to iteratively compute the next value in a way that the final condition $X_{t_{n+1}} = b$ is still met. To see how this can be achieved, we first look at the process
$$Z_t := \alpha + W_{t-t_0} - \frac{t-t_0}{T-t_0}(W_{T-t_0} + \alpha - \beta)\ , \quad t \in [t_0, T]\ .$$
This process is a generalized Brownian bridge on the interval $[t_0, T]$ with initial (final) value α (β). Moreover, it has mean function $\mu(t) = \alpha + \frac{(\beta-\alpha)(t-t_0)}{T-t_0}$ and covariance $C(s,t) = \min\{s-t_0, t-t_0\} - \frac{(s-t_0)(t-t_0)}{T-t_0}$ (check this!). This observation implies that the iterative procedure can thus be realized by generating appropriate increments. In fact, the results above show that X_{t_i} conditioned upon $X_{t_{i-1}}$ is normally distributed with mean $X_{t_{i-1}} + (b - X_{t_{i-1}})\frac{t_i - t_{i-1}}{t_{n+1} - t_{i-1}}$ and variance $\frac{(t_{n+1}-t_i)(t_i-t_{i-1})}{t_{n+1}-t_{i-1}}$ for any $1 \leqslant i \leqslant n$. The iteration based on these facts is implemented in the attached `Python` code .

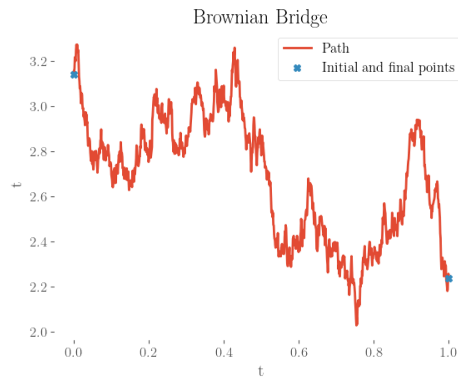

Figure 4: Brownian bridge

Python code:

```
import numpy as np
import matplotlib.pyplot as plt
from matplotlib import rc

#graphical parameters
################################################################################
plt.style.use('ggplot')
rc('font',**{'family':'serif','serif':['Computer Modern Roman'],
     'size' : '12'})
rc('text', usetex=True)
rc('lines', linewidth=2)
plt.rcParams['axes.facecolor']='w'
################################################################################

np.random.seed(42)

def brownian_bridge(t,a,b):
    n=np.size(t)-2
    X=np.zeros(n+2,)
    X[0]=a
    X[-1]=b
    tfinal=t[-1]
    Z=np.random.standard_normal(n,)
    #main loop
    for k in range(1,n+1):
        mu = X[k-1] + (b-X[k-1])*(t[k]-t[k-1])/(tfinal-t[k-1]);
        sig2 = (tfinal-t[k])*(t[k]-t[k-1])/(tfinal-t[k-1]);
        X[k] = mu +np.sqrt(sig2)*Z[k-1];

    return X

# Having defined the function, we are ready to run it

a = np.pi; #initial point
b = np.sqrt(5); #final point
n = 1000;
t = np.linspace(0,1,n);
X = brownian_bridge(t,a,b);
plt.plot(t,X,label='Path');
plt.plot([0,1],[a,b],marker="X",linestyle='None',label='Initial and final points');
```

```
plt.xlabel("t");
plt.ylabel("t");
plt.legend()
plt.title('Brownian Bridge')
plt.show()
```

Exercice 4.

Consider a fractional Brownian motion (fBM) $\{B^H(t), t \in [0,1]\}$, which is a centered Gaussian process with $B^H(0) = 0$ and covariance function

$$\mathrm{Cov}(t,s) = \frac{1}{2}(|t|^{2H} + |s|^{2H} - |t-s|^{2H}),$$

where $H \in (0,1)$ is the so-called *Hurst index*.

1) To sample such a process let us consider, for a fixed $h > 0$, the increment process $\delta B_h(t) = B^H(t+h) - B^H(t)$. Show that $\delta B_h(t)$ is a centered stationary Gaussian process.

2) If one is able to sample exactly the process $\delta B_h(t)$ on a uniform grid $t_j = jh$, then one can construct an exact sample of the fractional Brownian motion on the same grid points as $B^H(t_k) = \sum_{j=0}^{k-1} \delta B_h(t_j)$. Sample a fractional Brownian motion using FFT and circular embedding. Implement your experiment for different values of $H < 1/2$ and $H > 1/2$.

Solution

1) We begin by obtaining the covariance matrix for the incremental process δB^H. For any $t, s, h \geqslant 0$ we have

$$\begin{aligned}
&\mathrm{Cov}_{\delta B^H}(t,s) := \mathrm{Cov}[B^H(t+h) - B^H(t), B^H(s+h) - B^H(s)] \\
&= \mathbb{E}[(B^H(t+h) - B^H(t))(B^H(s+h) - B^H(s))] \\
&= \mathbb{E}[B^H(t+h)B^H(s+h)] - \mathbb{E}[B^H(t+h)B^H(s)] - \mathbb{E}[B^H(t)B^H(s+h)] + \mathbb{E}[B^H(s)B^H(t)] \\
&= \mathrm{Cov}_{\delta B^H}(t+h, s+h) - \mathrm{Cov}_{\delta B^H}(t+h, s) - \mathrm{Cov}_{\delta B^H}(t, s+h) - \mathrm{Cov}_{\delta B^H}(t,s) \\
&= \frac{1}{2}(|t+h|^{2H} + |s+h|^{2H} - |t-s|^{2H} - |t+h|^{2H} - |s|^{2H} + |t+h-s|^{2H} - |t|^{2H} \\
&- |s+h|^{2H} + |s+h-t|^{2H} + |t|^{2H} + |s|^{2H} - |t-s|^{2H}) \\
&= \frac{1}{2}(|t+h-s|^{2H} + |s+h-t|^{2H} - 2|t-s|^{2H}),
\end{aligned}$$

which is a stationary Gaussian process. Notice, moreover, that since $\mathbb{E}[B^H(t)] = 0,\ \forall t \geqslant 0$, then $\mathbb{E}[\delta B^H] = \mathbb{E}[B^H(t+h)] - \mathbb{E}[B^H(t)] = 0$, and as such, δB^H is a centered stationary Gaussian process.

2) Code is attached. The resulting simulations for different values of H ($H = 0.2, H = 0.5$ and $H = 0.9$) are shown in Figure 5.

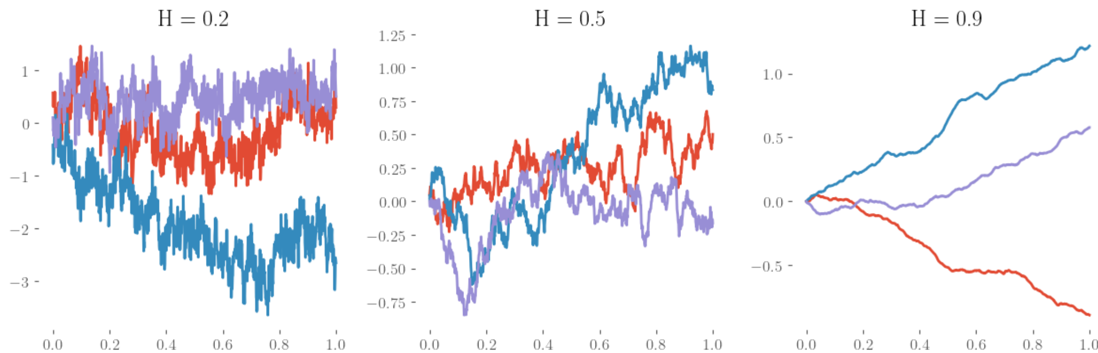

Figure 5: Three simulations of the fBM process for different values of H. Notice that the smoothness increases as H increases. In addition, for $H = 1/2$, the fBM coincides with a Brownian motion.

Python code:

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.fftpack import fft
from matplotlib import rc

#graphical parameters
################################################################################
plt.style.use('ggplot')
rc('font',**{'family':'serif','serif':['Computer Modern Roman'],
     'size' : '12'})
rc('text', usetex=True)
rc('lines', linewidth=2)
plt.rcParams['axes.facecolor']='w'
################################################################################

np.random.seed(42)

fig, axes = plt.subplots(nrows = 1, ncols = 3, figsize = (12,4))
H=[0.2,0.5,0.9]
hh=0
for ax in axes:
    for l in range(3):
        N= 1001 #number of samples
        H2= 2*H[hh] #Hurst parameter
        sigma=np.zeros(N)
        sigma[0]=1
        #constructs covariance generator
        for k in range(1,N):
            sigma[k]=0.5* (  (k+1)**H2  -2.0*(k)**H2 + (k-1)**H2     )

        # we now construct the covariance matrix generator via circular embedding
        c=np.hstack((sigma,sigma[-2::-1][:-1]))
        # notice that we don't assemble the covariance complete matrix
        labda=fft(c) # gets eigenvalues
        eta=np.sqrt(labda/(2*N))
        #samples complex normal vectors
```

```
        Z=np.random.standard_normal(len(c))+1.j*np.random.standard_normal(len(c))
        Zeta=Z*eta
        #computes fft
        X2n=fft(Zeta)
        A=X2n[:N+1]
        #gets real part
        X=np.real(A)
        c=1/N
        #rescales
        X1=c**(H2/2)*np.cumsum(X)
        #plots

        ax.plot(np.linspace(0,1,len(X1)),X1)
    ax.set_title('H = '+str(H2/2))
    plt.tight_layout()

    hh+=1
plt.show()
```

(Optional) Exercice 5.

(Lévy-Ciesielski construction). Let $\xi_{-1} \sim \mathcal{N}(0,1)$ and denote by $Y_0(t)$ the linear function on $[0,1]$ with $Y_0(0) = 0$, and $Y_0(1) = \xi_{-1}$. For $j \in \mathbb{N}$ and $N = 0, 1, 2 \dots$, let $t_j^N = 2^{-N} j$ and let $Y_N(t)$ be the piecewise linear function such that

$$Y_{N+1}(t_j^N) = Y_N(t_j^N) \tag{5.1}$$

$$Y_{N+1}(t_{2j+1}^{N+1}) = \frac{1}{2}\big(Y_N(t_j^N) + Y_N(t_{j+1}^N)\big) + \xi_{j,N}, \quad \xi_{j,N} \overset{\text{iid}}{\sim} \mathcal{N}(0, 2^{-N-2}). \tag{5.2}$$

Here N is to be understood as a "discretization level" of the interval [0,1] on equal sub-intervals of length 2^{-N}. This process is known as the *Lévy-Ciesielski construction of a Brownian motion.*

1) Simulate the previous process for different values of N.
2) Prove that for any $N \in \mathbb{N}$, $\mathbb{E}[Y_N(t_j^N)] = 0$ and $\mathrm{Cov}\big(Y_N(t_j^N), Y_N(t_k^N)\big) = \min\{t_j^N, t_k^N\}$, with $j, k = 0, \dots 2^N$.
3) For any $s_1, \dots, s_m \in [0,1]$, prove that $[Y_N(s_1), \dots, Y_N(s_m)]^T$ converges in distribution to a normal $\mathcal{N}(0, C)$, where C has entries $C_{ij} = \min\{t_j^N, t_k^N\}$, with $j, k = 0, \dots, 2^N$.
4) Write $W_n(t) := [Y_N(0), Y_N(t_1^N), \dots Y_N(t_{2N-1}^N), Y_N(1)]$. Conclude that $W_N(t) \to W(t)$, where $W(t)$ is the standard Brownian motion.

Solution

1) `Python` code is attached. Results are shown in Figure 6. Notice that a similar procedure can be followed to construct a Brownian bridge.
2) This can be shown inductively. Notice that for $N = 0$ we have

$$\mathbb{E}[Y_0(t^0)] = \mathbb{E}[Y_0(0) + t_1^0 \xi_{-1}] = t_1^0 \mathbb{E}[\xi_{-1}] = 0,$$
$$\mathbb{E}[(Y_0(t^0))^2] = \mathbb{E}[(Y_0(0) + t_1^0 \xi_{-1})^2] = (t_1^0)^2 \mathbb{E}[\xi_{-1}^2] = (t_1^0)^2 = 1.$$

Proceeding inductively, we may assume that at the Nth step it holds that $\mathbb{E}[Y_N(t_k)] = 0$ and $\text{Cov}[Y_N(t_k^N), Y_N(t_j^N)] = \min\{t_k^N, t_j^N\}$. This will be our inductive hypothesis. At the $(N+1)^{\text{th}}$ step we have that

$$\mathbb{E}[Y_{N+1}(t_j^{N+1})] = \mathbb{E}[Y_N(t_j^n)] = 0, \quad \text{(by inductive hypothesis)}$$
$$\mathbb{E}[Y_{N+1}(t_{2j+1}^{N+1})] = \frac{1}{2} \underbrace{\mathbb{E}[Y_N(t_j^N) + Y_N(t_{j+1}^N)]}_{= 0 \text{ by inductive hypothesis}} + \mathbb{E}[\xi_{j,N}] = 0,$$

which shows that $\mathbb{E}[Y_n(t_j^N)] = 0$, for any $N \in \mathbb{N}$ and any $t_j^N = 2^{-N}j,\ j = 0, \dots, 2^N$. We now focus on the covariance term. For some $k, j \in \{0, 1, 2, \dots, 2^N\}$ we have:

$$\begin{aligned}
&\text{Cov}\big[Y_{N+1}(t_{2j+1}^{N+1}), Y_{N+1}(t_{2k+1}^{N+1})\big] \\
&= \mathbb{E}\bigg[\Big(\frac{1}{2}\big(Y_N(t_j^N) + Y_N(t_{j+1}^N)\big) + \xi_{j,N}\Big)\Big(\frac{1}{2}\big(Y_N(t_k^N) + Y_N(t_{k+1}^N)\big) + \xi_{k,N}\Big)\bigg] \\
&= \frac{1}{4}\mathbb{E}\big[\big(Y_N(t_j^N) + Y_N(t_{j+1}^N)\big)\big(Y_N(t_k^N) + Y_N(t_{k+1}^N)\big)\big] + 2^{-N-2}\delta_{j,k} =: K_{j,k}^N
\end{aligned}$$

where $\delta_{j,k}$ is the Kronecker delta. Assuming without loss of generality that $j < k$ we then have from our induction hypothesis that

$$K_{j,k}^N = \frac{1}{4}\big(2t_j^N + 2t_{j+1}^N\big) = t_{2j+1}^{N+1}. \tag{5.3}$$

Similarly, if $k = j$, we then have

$$K_{j,k}^N = \frac{1}{4}\big(3t_j^N + t_{j+1}^N + 2^{-N}\big) + 2^{-N-2} = t_{2j+1}^{N+1} = t_{2j+1}^{N+1}. \tag{5.4}$$

We can proceed similarly to verify

$$\begin{aligned}
&\text{Cov}\big[Y_{N+1}(t_{2j+1}^{N+1}), Y_{N+1}(t_{k+1}^{N+1})\big] \\
&= \mathbb{E}\bigg[\Big(\frac{1}{2}\big(Y_N(t_j^N) + Y_N(t_{j+1}^N)\big) + \xi_{j,N}\Big)\big(Y_{N+1}(t_{k+1}^N)\big)\bigg] \\
&= \frac{1}{2}\mathbb{E}\big[Y_N(t_j^N)\big(Y_N(t_{k+1}^N)\big)\big] + \frac{1}{2}\mathbb{E}\big[Y_N(t_{j+1}^N)\big(Y_N(t_{k+1}^N)\big)\big] =: K_{j,k}^N.
\end{aligned}$$

Once again, assuming without loss of generality that $j \leqslant k$, we have that, from the inductive hypothesis,

$$K_{j,k}^N = \frac{1}{2}\big(t_j^N + t_{j+1}^N\big) = t_{2j+1}^{N+1}, \quad j \leqslant k,$$

and similarly for $\text{Cov}\big[Y_{N+1}(t_{2j+1}^{N+1}), Y_{N+1}(t_k^{N+1})\big]$. Thus, for any $N \in \mathbb{N}$ and $j, k \in \{0, 1, \dots, 2^N\}$, it follows by induction that $\mathbb{E}[Y_N(t_j^N)] = 0$ and $\text{Cov}\big(Y_N(t_j^N), Y_N(t_k^N)\big) = \min\{t_j^N, t_k^N\}$.

3) We now want to extend the previous result to any $t \in [0, 1]$ (i.e., not just the dyadic points of the form $t_j^N = 2^{-N}j,\ j = 0, 1, 2 \dots 2^N$). To do so, we use a continuous, piecewise linear interpolation between the $Y_N(t_j^N)$ points. For some given $N \in \mathbb{N}$, define the linear interpolant $\{\phi_j(t)\}_{j=0}^{2^N}$ as

$$\phi_j(t) = \begin{cases} 2^{(N-1)/2}(t - t_j^N), & \text{if } t \in [t_j^N, t_{2j+1}^{N+1}], \\ 2^{(N-1)/2}(t_{j+1}^N - t), & \text{if } t \in [t_{2j+1}^{N+1}, t_{j+1}^N], \\ 0 & \text{otherwise.} \end{cases}$$

Thus, we can the write the continuous, piecewise linear interpolated process as

$$X_N(t) = \sum_{j=1}^{N} Y_N(t_j^N)\phi_j(t).$$

It is not difficult to see that $\mathbb{E}[X_N(t)] = 0$. As for the covariance we have that

$$\begin{aligned}
\mathrm{Cov}_{X_N}(s,t) &= \mathbb{E}\left[\left(\sum_{j=1}^{N} Y_N(t_j^N)\phi_j(t)\right)\left(\sum_{k=1}^{N} Y_N(t_j^N)\phi_k(s)\right)\right] \\
&= \sum_{j,k} C_{j,k}^N \phi_j(t)\phi_k(s) = \sum_{j,k} \min\{t_k^N, t_j^N\}\phi_j(t)\phi_k(s) \\
&= I_N \otimes I_N \min\{s,t\}
\end{aligned}$$

Convergence to the Wiener process follows from the fact that the dyadic numbers (numbers of the form $2^{-k}, k = 1, 2, \dots$) are dense in $[0, 1]$.

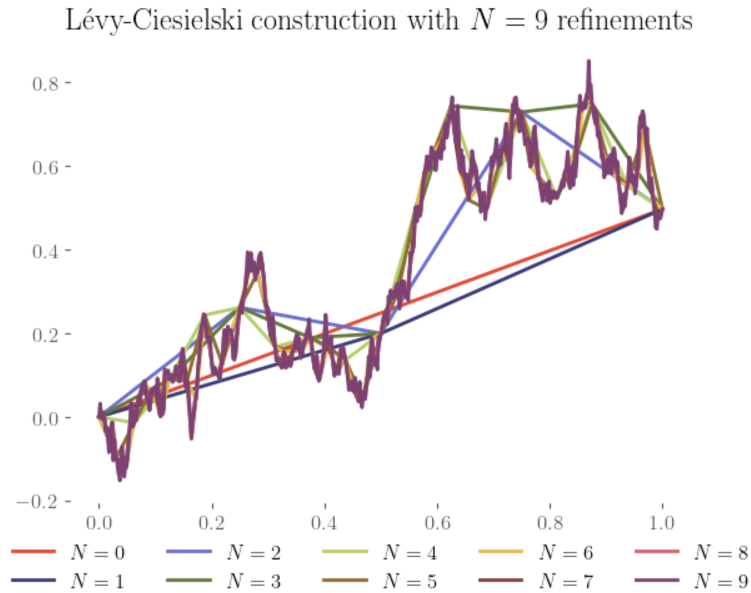

Figure 6

Python code:

```
import numpy as np
import matplotlib.pyplot as plt
from matplotlib import rc
from matplotlib.pyplot import cm
#graphical parameters
################################################################################
plt.style.use('ggplot')
rc('font',**{'family':'serif','serif':['Computer Modern Roman'],
     'size' : '12'})
rc('text', usetex=True)
```

```
rc('lines', linewidth=2)
plt.rcParams['axes.facecolor']='w'
##############################################################################

np.random.seed(42)

N=10
color=iter(cm.tab20b(np.linspace(0,1,N+1))) # for plotting

y0=np.zeros(1)
y1=np.random.standard_normal(1)
Y_old=np.concatenate([y0,y1])
plt.plot(Y_old)
#stats the construction
for i in range(1,N):
     t=np.linspace(0,1,2**i+1) #discretizes time
     Y=np.zeros(len(t))
     Y[::2]=Y_old
     for j in range(len(t)-1):
         if np.mod(j+1,2)==0:
             #interpolates
             Y[j]=0.5*(Y[j-1]+Y[j+1])+np.sqrt(2**(-i-2))*np.random.standard_normal(1)
     Y_old=Y
     #for the plotting color
     c=next(color)
     plt.plot(t,Y,c=c)

plt.title(r'L\'evy-Ciesielski construction with $N='+str(N-1)+'$ refinements')


#some plotting parameters
leg=['$N=0$']
for i in range(N):
    ln='$N='+str(i+1)+'$'
    leg.append(ln)
plt.legend(leg,fancybox=True, framealpha=0.0,
               loc='upper center', bbox_to_anchor=(0.5, -0.05), shadow=False, ncol=5)
plt.show()
```