Fractional and Volterra processes in Finance

Challenge 1 - Simulation of Gaussian Volterra processes

Eduardo Abi Jaber

Challenge : Unlock the potential of Gaussian fractional processes and pave the way for more accurate simulations!

PLEASE ENTER YOUR FULL NAMES HERE:

- Dimitri Sotnikov
- Elie Jabbour
- Romain Farthoat

DEADLINE: 4 February before 10 AM to be sent by email to eduardo.abi-jaber@polytechnique.edu

The aim of the challenge is to figure out ways to efficiently simulate the Riemann-Liouville fractional Brownian motion:

$$X_t = \nu \int_0^t K(t,s) dW_s,$$

with

$$K(t,s) = \frac{1}{\Gamma(H+1/2)} (t-s)^{H-1/2} 1_{s<t}$$

and $H < 1/2$.

The covariance kernel of X is given in the following closed form

$$\Sigma_0(s,u) = \frac{\nu^2}{\Gamma(H+1/2)^2} \int_0^{s\wedge u} (s-z)^{H-1/2}(u-z)^{H-1/2} dz \tag{1}$$

$$= \frac{\nu^2}{\Gamma(\alpha)\Gamma(1+\alpha)} \frac{s^\alpha}{u^{1-\alpha}} \, {}_2F_1\left(1, 1-\alpha; 1+\alpha; \frac{s}{u}\right) \tag{2}$$

where $\alpha = H + 1/2$ and ${}_2F_1$ is the Gaussian hypergeometric function.

Guidelines

- Implement and briefly explain and comment the methods. We are interested in low regimes of H. Plot the sample paths on same gaussian increments, to compare paths by paths. You can take $T = 1.$ and $n_{steps} = 300$ time steps uniformly spaced on $[0, T]$. (set $\nu = 1$).
- Two metrics : running time (using "timeit) to simulate one trajectory and MSE error of the paths wrt to the exact path simulated using cholesky method:

$$MSE = \sqrt{\frac{1}{n_{steps}} \sum_{i=1}^{n_{steps}} \left(X_{t_i}^{\text{method}} - X_{t_i}^{\text{cholesky}} \right)^2}$$

Question: Detail the computations that lead to the covariance kernel. Is it valid for $H \leq 1/2$, $H \geq 1/2$? both?

Answer: Setting $s \leq u$, it readily follows the definitions and a change of variable that

$$\Sigma_0(s,u) = \frac{\nu^2}{\Gamma(H+1/2)^2} \int_0^{s\wedge u} (s-z)^{H-1/2}(u-z)^{H-1/2} dz \tag{3}$$

$$= \frac{\nu^2}{\Gamma(\alpha)^2} \int_0^{s} (s-z)^{\alpha-1}(u-z)^{\alpha-1} dz \tag{4}$$

$$= \frac{\nu^2 \alpha}{\Gamma(\alpha)\Gamma(\alpha+1)} \int_0^{s} (s-z)^{\alpha-1}(u-z)^{\alpha-1} dz \tag{5}$$

$$= \frac{\nu^2 \alpha}{\Gamma(\alpha)\Gamma(\alpha+1)} \frac{s^\alpha}{u^{1-\alpha}} \cdot \underbrace{\frac{\alpha}{s} \frac{u^{1-\alpha}}{s^{\alpha-1}} \int_0^{s} (s-z)^{\alpha-1}(u-z)^{\alpha-1} dz}_{= \alpha \int_0^s \left(1-\frac{z}{s}\right)^{\alpha-1} \left(1-\frac{z}{u}\right)^{\alpha-1} d\left(\frac{z}{s}\right)} \tag{6}$$

$$\overset{v=\frac{z}{s}}{=} \frac{\nu^2 \alpha}{\Gamma(\alpha)\Gamma(\alpha+1)} \frac{s^\alpha}{u^{1-\alpha}} \cdot \underbrace{\alpha \int_0^1 (1-v)^{\alpha-1} \left(1-\frac{s}{u}v\right)^{\alpha-1} dz}_{=: \, \phi\left(\frac{s}{u}\right)} \tag{7}$$

Now we have the relationship (Euler's representation for hypergeometric function)

$$B(b, c-b) \, {}_2F_1(a, b; c; z) = \int_0^1 x^{b-1}(1-x)^{c-b-1}(1-zx)^{-a} dx \tag{8}$$

where $\Re(c) > \Re(b) > 0$, $|z| < 1$, and B is the Beta function. \ It can be derived by using the series expansion of $(1-zx)^{-a}$ and integrating it term by term. \ Thus setting

$$\begin{cases} b = 1 \\ c = \alpha + 1 \\ a = 1 - \alpha \end{cases}$$

we find that

$$\phi\left(\frac{s}{u}\right) = \alpha B(1, \alpha) \, {}_2F_1\left(1, 1-\alpha; 1+\alpha; \frac{s}{u}\right) \tag{9}$$

Since $B(1,\alpha) = \int_0^1 x^{\alpha-1} dx = \frac{1}{\alpha}$, the result follows:

$$\boxed{\Sigma_0(s,u) = \frac{\nu^2}{\Gamma(\alpha)\Gamma(1+\alpha)} \frac{s^\alpha}{u^{1-\alpha}} \, {}_2F_1\left(1, 1-\alpha; 1+\alpha; \frac{s}{u}\right)} \tag{10}$$

This computation is valid both for $H \geq \frac{1}{2}$ and $H \leq \frac{1}{2}$.

Several options and suggestions detailed below:

- Cholesky
- Different Euler schemes
- multifactor euler vs exact (cholesky on factors)

1. Exact simulation using Cholesky

Warm up with Cholesky, but aim to set the simulation field on fire with your efficient methods for Gaussian fractional processes.

In [ ]:
```
import numpy as np
import matplotlib.pyplot as plt
import scipy.special as sc
from scipy.special import gamma, gammainc
from scipy.integrate import quad
```

In [2]:
```
DEFAULT_SEED = 18600503 # Volterra's birthday
```

The ${}_2F_1$ Gaussian hypergeometric function can be implemented using scipy `sc.hyp2f1`, pay close attention to the parameters, notably final parameter needs to be less than 1?

Question: Check the doc https://docs.scipy.org/doc/scipy/reference/generated/scipy.special.hyp2f1.html and explain.

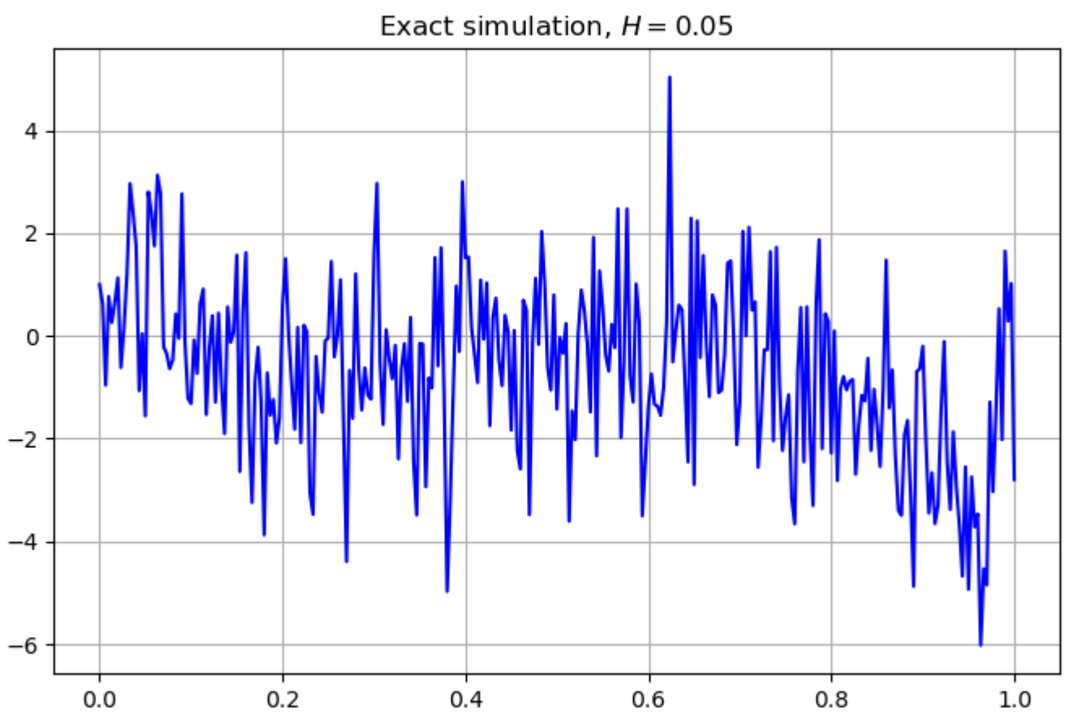
Answer: The $sc.\ hyp2f1$ function can be used to compute ${}_2F_1$, using different methods according to the value of z. \ For $|z| < 1$, the series

$$\sum_{n=0}^{\infty} \frac{(a)_n (b)_n}{(c)_n} \frac{z^n}{n!} \tag{11}$$

converge, and can be approximated without too much trouble on much of the unit disk. \ In particular, real arguments $z \in (-1, 1)$ aren't problematic. Outside the disk, other techniques are required to compute the analytic continuation of the function defined by the series. \ Poles on the real axis arise, but they are easily detected.

In [3]:
```
def covariance(H, s, u):
    a, b = np.minimum(s, u), np.maximum(s, u)
    if b == 0:
        return 0
    alpha = H + .5
    return sc.hyp2f1(1, 1 - alpha, 1 + alpha, a/b) * a**alpha / b**(1-alpha) / gamma(alpha) / gamma(1 + alpha)

cov = np.frompyfunc(covariance, 3, 1)

def exact_cholesky(n_steps, H, X0=0, T=1, rng=None):
    if rng is None:
        rng = np.random.default_rng(seed=DEFAULT_SEED)
    time = np.arange(1, n_steps + 1) * T / n_steps
    cov_matrix = cov(H, time[:, None], time[None, :]).astype('float64')
    sim = np.matmul(np.linalg.cholesky(cov_matrix), rng.normal(0, 1, n_steps).T)
    time = np.insert(time, 0, 0)
    sim = X0 + np.insert(sim, 0, 0)
    return sim
```

In [7]:
```
T = 1
H = 0.05
n_steps = 300

time = np.arange(1, n_steps + 1) * T/n_steps
time = np.insert(time, 0, 0)
sim = exact_cholesky(n_steps, H, 1)
fig = plt.figure(figsize=(8, 5))

plt.plot(time, sim, 'b')
plt.grid()
plt.title(f"Exact simulation, $H={H}$")
plt.show()
```



2. Euler Schemes

Explore the unknown and discover the hidden potential of Euler methods through a performance evaluation.

We will consider three (modified) Euler schemes after writing

$$X_{t_i} = X_0 + \sum_{j=1}^{i} \underbrace{\int_{t_{j-1}}^{t_j} K(t_i, s) dW_s}_{Y_j^i}.$$

1. **EULER 1** Naive:

$$X_{t_i} = X_0 + \nu \sqrt{dt} \sum_{j=1}^{i} K(t_i, t_{j-1}) Z_j$$

with $Z_j \sim \mathcal{N}(0, 1)$ iid.

2. **EULER 2** Write $dW_s \approx Z_j \frac{ds}{\sqrt{dt}}$ so that

$$X_{t_i} = X_0 + \nu \sum_{j=1}^{i} w_j^i Z_j$$

with

$$w_j^i = \frac{1}{\sqrt{dt}} \int_{t_{j-1}}^{t_j} K(t_i, s) ds = \frac{1}{\sqrt{dt}} \frac{1}{\Gamma(H + 0.5)(H + 0.5)} \left((t_i - t_{j-1})^{H+0.5} - (t_i - t_j)^{H+0.5} \right)$$

3. **EULER 3** Observe that $(Y_1^i, \dots, Y_i^i)$ is a centered Gaussian vector with independent components such that the std of the j-th component is

$$\tilde{w}_j^i = \sqrt{\int_{t_{j-1}}^{t_j} K(t_i, s)^2 ds} = \frac{1}{\Gamma(H + 0.5)} \sqrt{\frac{\left((t_i - t_{j-1})^{2H} - (t_i - t_j)^{2H} \right)}{2H}}$$

so that we use

$$X_{t_i} \approx X_0 + \nu \sum_{j=1}^{i} \tilde{w}_j^i Z_j.$$

Note that the simulation is not exact since

$$\mathbb{E}[Y_j^i Y_{j'}^{i'}] = \int_{t_{j-1}}^{t_j} K(t_i, s) K(t_{i'}, s) ds 1_{j=j'},$$

whereas in the approximation $\mathbb{E}[\tilde{Y}_j^i \tilde{Y}_{j'}^{i'}] = w_j^i w_j^{i'}$. (to double check)

Reference: Rambaldi, S., & Pinazza, O. (1994). An accurate fractional Brownian motion generator. Physica A: Statistical Mechanics and its Applications, 208(1), 21-30.

Compare on graphs + MSE that the Naive Euler scheme is way off for small values of $H < 0.05$. Works fine for bigger values of $H > 0.3$... etc...

(!) Please stick to the names **EULER 1**, **EULER 2**, **EULER 3**.

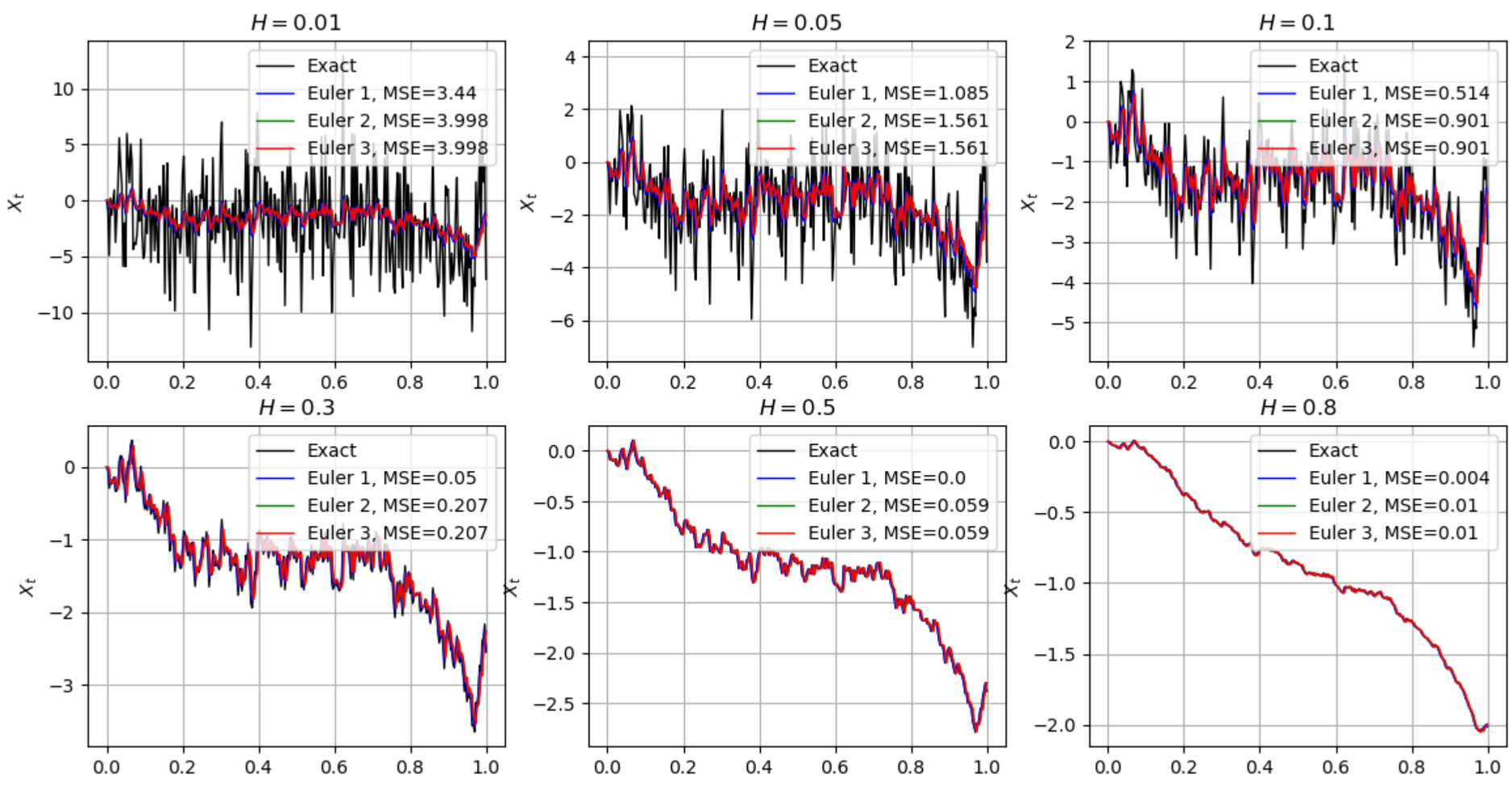
In [12]:
```
from scipy.linalg import circulant

def get_power_kernel(H):
    """
    Returns a power-law kernel with the parameter H as a function.
    """
    return lambda u: u**(H - 0.5) / gamma(H + 0.5) * (u >= 0) # we will assume everywhere that K(t, s) = K(t - s)

def mse(x, y):
    return np.sqrt(np.mean((x - y)**2))
```

In [13]:
```
def euler_scheme_1(n_steps, K, X0=0, T=1, nu=1, rng=None):
    """
    Args:
        K: kernel, a function of u.
        T: time horizon, a real number,
        n_steps: number of steps in the discretization scheme.
    """
    if rng is None:
        rng = np.random.default_rng(seed=DEFAULT_SEED)
    dt = T / n_steps
    Z = rng.normal(size=n_steps)
    K_arr = K(np.arange(1, n_steps + 1) * dt)
    K_mat = np.tril(circulant(K_arr), 0)
    X = X0 + nu * np.sqrt(dt) * K_mat @ Z
    X = np.insert(X, 0, X0)
    return X

def euler_scheme_2(n_steps, H, X0=0, T=1, nu=1, rng=None):
    """
    Args:
        K: kernel, a function of u.
        T: time horizon, a real number,
        n_steps: number of steps in the discretization scheme.
    """
    if rng is None:
        rng = np.random.default_rng(seed=DEFAULT_SEED)
    dt = T / n_steps
    Z = rng.normal(size=n_steps)
    W_arr = (np.arange(1, n_steps + 1) * dt)**(H + 0.5)
    W_arr = np.diff(W_arr, prepend=W_arr[0])
    W_mat = np.tril(circulant(W_arr), 0) / np.sqrt(dt) / gamma(H + 0.5) / (H + 0.5)
    X = X0 + nu * W_mat @ Z
    X = np.insert(X, 0, X0)
    return X

def euler_scheme_3(n_steps, H, X0=0, T=1, nu=1, rng=None):
    """
    Args:
        K: kernel, a function of u.
        T: time horizon, a real number,
        n_steps: number of steps in the discretization scheme.
    """
    if rng is None:
        rng = np.random.default_rng(seed=DEFAULT_SEED)
    dt = T / n_steps
    Z = rng.normal(size=n_steps)
    W_arr = (np.arange(1, n_steps + 1) * dt)**(2 * H)
    W_arr = np.sqrt(np.diff(W_arr, prepend=W_arr[0]) / (2 * H))
    W_mat = np.tril(circulant(W_arr), 0) / gamma(H + 0.5)
    X = X0 + nu * W_mat @ Z
    X = np.insert(X, 0, X0)
    return X
```

In [14]:
```
n_steps = 300
H_range = [[0.01, 0.05, 0.1],
           [0.3, 0.5, 0.8]]
T = 1
nu = 1

t_grid = np.linspace(0, T, n_steps + 1)
dt = T / n_steps
```

Schemes performance for different H

In [15]:
```
fig, ax = plt.subplots(2, 3, figsize=(14, 7))

for i in range(2):
    for j in range(3):
        H = H_range[i][j]
        X_scheme_1 = euler_scheme_1(
            n_steps=n_steps,
            K=get_power_kernel(H),
        )
        X_scheme_2 = euler_scheme_2(
            n_steps=n_steps,
            H=H
        )
        X_scheme_3 = euler_scheme_3(
            n_steps=n_steps,
            H=H
        )
        X_exact = exact_cholesky(n_steps, H)

        ax[i, j].plot(t_grid, X_exact, 'k', label='Exact', lw=1)
        ax[i, j].plot(t_grid, X_scheme_1, 'b', label=f'Euler 1, MSE={np.round(mse(X_exact, X_scheme_1), 3)}', lw=1)
        ax[i, j].plot(t_grid, X_scheme_2, 'g', label=f'Euler 2, MSE={np.round(mse(X_exact, X_scheme_2), 3)}', lw=1)
        ax[i, j].plot(t_grid, X_scheme_3, 'r', label=f'Euler 3, MSE={np.round(mse(X_exact, X_scheme_3), 3)}', lw=1)

        ax[i, j].set_ylabel('$X_t$')
        ax[i, j].set_title(f'$H = {H}$')
        ax[i, j].legend(loc='upper right')
        ax[i, j].grid()
```



Time performance

In [16]:
```
H = 0.01
n_steps = 300
```

In [10]:
```
%%timeit
X_scheme_1 = euler_scheme_1(
    n_steps=n_steps,
    K=get_power_kernel(H),
)
```

```
1.31 ms ± 217 µs per loop (mean ± std. dev. of 7 runs, 1,000 loops each)
```

In [62]:
```
%%timeit
X_scheme_2 = euler_scheme_2(
    n_steps=n_steps,
    H=H
)
```

```
1.37 ms ± 113 µs per loop (mean ± std. dev. of 7 runs, 1,000 loops each)
```

In [63]:
```
%%timeit
X_scheme_3 = euler_scheme_3(
    n_steps=n_steps,
    H=H
)
```

```
1.15 ms ± 183 µs per loop (mean ± std. dev. of 7 runs, 1,000 loops each)
```

In [64]:
```
%%timeit
X_exact = exact_cholesky(n_steps, H)
```

```
642 ms ± 24.7 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

Conclusions

- Pretty the same time performance for all the 3 Euler's schemes, approximately 400-600 times as faster as exact simulation for n_steps = 300. For greater n_steps this difference will be much greater due to quickly rising (at least quadratic) complexity of the exact simulation.
- The first scheme demonstrates the lowest MSE error for all considered values of H.
- In general, extremely poor performance of the Euler's schemes for rough processes ($H < 0.1$) due to the its failure to reproduce the behavior of the explosing kernel involved in the stochastic integral, i.e. in the case $t_i - t_j \ll 1$.

3. Multifactor approximations

Embrace the challenge, and push the boundaries of what's possible by making non-standard multifactor approximations work effectively.

Based on

- Abi Jaber, E., & El Euch, O. (2019). Multifactor approximation of rough volatility models. SIAM Journal on Financial Mathematics, 10(2), 309-349. https://arxiv.org/abs/1801.10359
- Abi Jaber, E. (2019). Lifting the Heston model. Quantitative Finance, 19(12), 1995-2013. https://arxiv.org/abs/1810.04868

$$X_t \approx X_0 + \nu \sum_{k=1}^{n} c_k Y_t^k$$

with

$$Y_t^k = \int_0^t e^{-x_k(t-s)} dW_s$$

$$Y_{t_i}^k = e^{-x_k h} Y_{t_{i-1}}^k + \xi_i^k, \quad \xi_i^k = \int_{t_{i-1}}^{t_i} e^{-x_k(t_i - s)} dW_s$$

with the parametrization:

$$c_i^n = \frac{(r_n^{(1-\alpha)} - 1) r_n^{(\alpha-1)(1+n/2)}}{\Gamma(\alpha)\Gamma(1-\alpha)(1-\alpha)} r_n^{(1-\alpha)i}, \quad x_i^n = \frac{1-\alpha}{2-\alpha} \frac{r_n^{2-\alpha} - 1}{r_n^{1-\alpha} - 1} r_n^{i-1-n/2},$$

where $\alpha := H + 1/2$, with a geometric repartition $\eta_i^n = r_n^i$ for some r_n such that

$$r_n \downarrow 1 \quad \text{and} \quad n \ln r_n \to \infty, \quad \text{as } n \to \infty.$$

We denote by

$$K_n(t) = \sum_{i=1}^{n} c_i e^{-x_i t}.$$

The first step is to determine a good value or r_n for a choice of n, H and T. For this, for a given H, n, T, we can choose r_n to minimize

$$\int_0^T |K_n(t) - K(t)|^2 dt$$

Question: Develop the expression (by developing the square) and show that it admits an explicit expression in terms of incomplete gamma function. Write a minimization function to find r and sanity check with the following table ($H = 0.1, T = 0.5$)

Answer: We have

$$\int_0^T |K_n(t) - K(t)|^2 dt = \int_0^T K_n(t)^2 dt + \int_0^T K(t)^2 dt - 2\int_0^T K_n(t)K(t)dt \tag{12}$$

The first term is

$$\int_0^T K_n(t)^2 dt = \int_0^T \left(\sum_{i=1}^n c_i e^{-x_i t} \right)^2 dt \tag{13}$$

$$= \sum_{i,j=1}^n c_i c_j \int_0^T e^{-(x_i+x_j)t} dt \tag{14}$$

$$\int_0^T K_n(t)^2 dt = \sum_{i,j=1}^n \frac{c_i c_j}{x_i + x_j} \left(1 - e^{-(x_i+x_j)T} \right) \tag{15}$$

The second one is

$$\int_0^T K(t)^2 dt = \frac{1}{\Gamma(H+1/2)^2} \int_0^T t^{2H-1} dt \tag{16}$$

$$= \frac{T^{2H}}{2H\,\Gamma(H+1/2)} \tag{17}$$

The last one is

$$2\int_0^T K_n(t)K(t)dt = \frac{2}{\Gamma(H+1/2)} \sum_{i=1}^n c_i \int_0^T e^{-x_i t} t^{H-1/2} dt \tag{18}$$

$$\overset{s=x_i t}{=} \frac{2}{\Gamma(H+1/2)} \sum_{i=1}^n \frac{c_i}{x_i^{H+1/2}} \underbrace{\int_0^{x_i T} e^{-s} s^{H-1/2} ds}_{=\gamma(H+1/2, x_i T)}, \tag{19}$$

where $\gamma(s,x)$ is an incomplete gamma function defined by $\gamma(s,x) = \int_0^x t^{s-1} e^{-t} dt$.

In [17]:
```
def get_c(H, n_fact, r):
    alpha = H + .5
    i = np.arange(1, n_fact + 1)
    return (r ** (1 - alpha) - 1) * r ** ( (alpha - 1) * (1 + n_fact/2) ) * r ** ( (1 - alpha) * i ) / gamma(alpha) / gamma(1 - alpha) / (1 - alpha)

def get_x(H, n_fact, r):
    alpha = H + .5
    i = np.arange(1, n_fact + 1)
    return (1 - alpha) / (2 - alpha) * ( r ** (2 - alpha) - 1) / ( r ** (1 - alpha) - 1 ) * r ** (i - 1 - n_fact/2)

def norm2(r, H, T, n):
    X = get_x(H, n, r)
    C = get_c(H, n, r)
    term1 = -2 * ( C * gammainc(H+0.5, T*X) / X**(H+0.5) ).sum() # gammainc is defined differently in scipy
    m = X[:,None] + X
    term2 = ( (C[:,None]*C) * (1-np.exp(-m*T)) / m ).sum()
    term3 = T**(2*H) / (2*H*gamma(H+0.5)**2)
    return term1 + term2 + term3

def get_r(H, T, n):
    return minimize(norm2, x0, (H,T,n), tol=1e-6).x[0]
```

In [18]:
```
from scipy.optimize import minimize

H = 0.1
T = 0.5
n = 4
N = [4, 10, 20, 40, 200]
X0 = [40, 30, 20, 10, 2]
result = []
for (x0,n) in zip(X0,N):
    res = minimize(norm2, x0, (H,T,n), tol=1e-6)
    print(f"n = {n}, r = {np.round(res.x[0], 4)}, norm^2 = {np.round(res.fun, 4)}")
    result.append( (res.x,res.fun) )
```

```
n = 4, r = 50.5435, norm^2 = 0.3699
n = 10, r = 18.0553, norm^2 = 0.1125
n = 20, r = 8.8749, norm^2 = 0.0325
n = 40, r = 4.4738, norm^2 = 0.0076
n = 200, r = 1.6945, norm^2 = 0.0001
```

Now that we know how to determine r.

3.1 Multifactor with Euler methods on factors

We will consider several Euler-type approximations for factors:

1. **Factor-Euler 1** :

$$\xi_i^k \approx e^{-x_k dt} \sqrt{dt} Z_i$$

2. **Factor-Euler 2**: writing $dW_s = Z_i ds / \sqrt{dt}$

$$\xi_i^k \approx \frac{1}{\sqrt{dt}} \int_{t_{i-1}}^{t_i} e^{-x_k(t_i - s)} ds Z_i = \frac{1}{\sqrt{dt}} \frac{1 - e^{-x_k dt}}{x_k} Z_i$$

3. **Factor-Euler 3**: using that ξ_i^k is gaussian with variance $\frac{1-e^{-2x_k h}}{2x_k}$, so that

$$\xi_i^k \approx \sqrt{\frac{1 - e^{-2x_k h}}{2x_k}} Z_i$$

4. **Factor-Euler 4**: implicit scheme as in lifting heston paper in the appendix. In our case, where $\lambda = 0$ and the is no $\sqrt{X_t}$ in the equation, so the schema is explicit:

$$Y_{t_i}^k \approx \frac{1}{1 + x_k dt} \left(Y_{t_{i-1}}^k + \sqrt{dt} Z_i \right)$$

5. **Factor-Euler 5**: modified variance:

$$X_{t_{i+1}} = X_0 + \nu \sum_k c_k e^{-x_k dt} Y_{t_i}^k + \nu \int_{t_i}^{t_{i+1}} K_n(t_{i+1}, s) dW_s$$

approximate second term by variance of original kernel K. Approximation of the second term:

$$\int_{t_i}^{t_{i+1}} K_n(t_{i+1}, s) dW_s \approx \sqrt{\int_{t_i}^{t_{i+1}} K(t_{i+1}, s)^2 ds} Z_i = \frac{(dt)^H}{\Gamma(H+0.5)\sqrt{2H}} Z_i$$

In [19]:
```
def get_xi(x, Z, dt, method):
    xi = np.zeros((n_steps, n_fact))
    if method == 1:
        xi = np.exp(-x[None, :] * dt) * np.sqrt(dt) * Z[:, None]
    elif method == 2:
        xi = (1 - np.exp(-x[None, :] * dt)) / x[None, :] / np.sqrt(dt) * Z[:, None]
    elif method == 3:
        xi = np.sqrt((1 - np.exp(-2 * x[None, :] * dt)) / x[None, :] / 2) * Z[:, None]
    return xi

def multifactor_euler(n_steps, n_fact, H, r, method, X0=0, T=1, nu=1, rng=None):
```

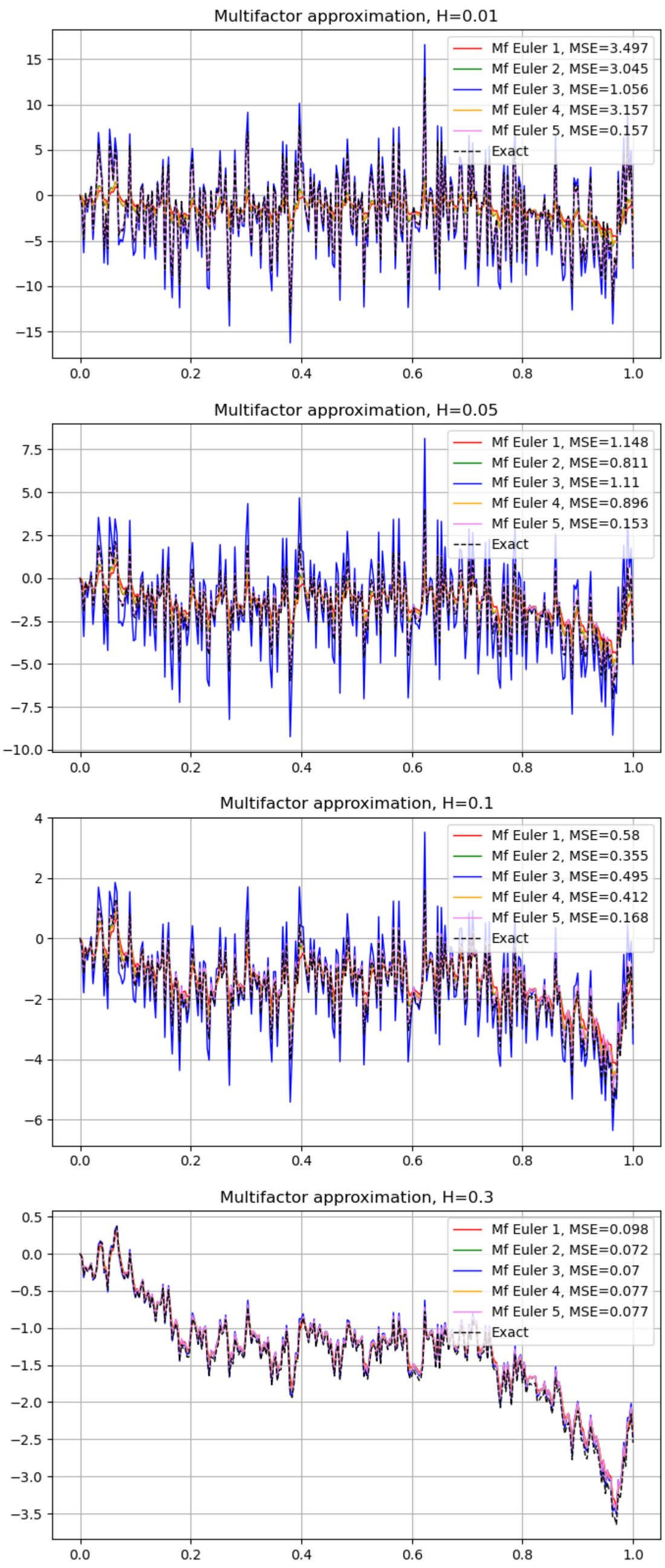
```
    """
    Implementation of the first 3 schemes. (method = 1, 2, 3 correspondingly).
    """
    if rng is None:
        rng = np.random.default_rng(seed=DEFAULT_SEED)
    c, x = get_c(H, n_fact, r), get_x(H, n_fact, r)
    dt = T / n_steps
    Z = rng.normal(size=n_steps)
    xi = get_xi(x, Z, dt, method)
    Y = np.zeros((1 + n_steps, n_fact))
    for i in range(1, n_steps + 1):
        Y[i] = np.exp(-x * dt) * Y[i - 1] + xi[i - 1]
    X = X0 + nu * Y @ c
    return X


def multifactor_euler_4(n_steps, n_fact, H, r, X0=0, T=1, nu=1, rng=None):
    if rng is None:
        rng = np.random.default_rng(seed=DEFAULT_SEED)
    c, x = get_c(H, n_fact, r), get_x(H, n_fact, r)
    dt = T / n_steps
    Z = rng.normal(size=n_steps)
    Y = np.zeros((1 + n_steps, n_fact))
    for i in range(1, n_steps + 1):
        Y[i] = (Y[i - 1] + np.sqrt(dt) * Z[i - 1]) / (1 + x * dt)
    X = X0 + nu * Y @ c
    return X


def multifactor_euler_5(n_steps, n_fact, H, r, X0=0, T=1, nu=1, rng=None):
    if rng is None:
        rng = np.random.default_rng(seed=DEFAULT_SEED)
    c, x = get_c(H, n_fact, r), get_x(H, n_fact, r)
    dt = T / n_steps
    Z = rng.normal(size=n_steps)
    xi = get_xi(x, Z, dt, method=1)
    Y = np.zeros((1 + n_steps, n_fact))
    v = dt**H / gamma(H + 0.5) / np.sqrt(2*H)
    X = np.zeros(n_steps + 1)
    for i in range(1, n_steps + 1):
        Y[i] = np.exp(-x * dt) * Y[i - 1] + xi[i - 1]
        X[i] = nu * c @ (np.exp(-x * dt) * Y[i - 1]) + nu * v * Z[i - 1]
    X = X0 + X
    return X
```

In [20]:

```
H = 0.01
n_fact = 30
n_steps = 300
T = 1

r = 8.83
```

In [21]:

```
t_grid = np.linspace(0, T, n_steps + 1)
H_range = [0.01, 0.05, 0.1, 0.3]

fig, axes = plt.subplots(4, 1, figsize=(8, 20))

for H, ax in zip(H_range, axes):
    euler_mf_1 = multifactor_euler(n_steps, n_fact, H, r, method=1)
    euler_mf_2 = multifactor_euler(n_steps, n_fact, H, r, method=2)
    euler_mf_3 = multifactor_euler(n_steps, n_fact, H, r, method=3)
    euler_mf_4 = multifactor_euler_4(n_steps, n_fact, H, r)
    euler_mf_5 = multifactor_euler_5(n_steps, n_fact, H, r)
    X_exact = exact_cholesky(n_steps, H)

    ax.plot(t_grid, euler_mf_1, 'r', lw=1, label=f'Mf Euler 1, MSE={np.round(mse(X_exact, euler_mf_1), 3)}')
    ax.plot(t_grid, euler_mf_2, 'g', lw=1, label=f'Mf Euler 2, MSE={np.round(mse(X_exact, euler_mf_2), 3)}')
    ax.plot(t_grid, euler_mf_3, 'b', lw=1, label=f'Mf Euler 3, MSE={np.round(mse(X_exact, euler_mf_3), 3)}')
    ax.plot(t_grid, euler_mf_4, 'orange', lw=1, label=f'Mf Euler 4, MSE={np.round(mse(X_exact, euler_mf_4), 3)}')
    ax.plot(t_grid, euler_mf_5, 'violet', lw=1, label=f'Mf Euler 5, MSE={np.round(mse(X_exact, euler_mf_5), 3)}')
    ax.plot(t_grid, X_exact, 'k--', lw=1, label='Exact')


    ax.legend(loc='upper right')
    ax.set_title(f'Multifactor approximation, H={H}')
    ax.grid()
```



For this example 5-th scheme shows a way much better MSE, especially, for rough processes!

Time performance

In [22]:
```
H = 0.01
n_fact = 30
n_steps = 300
T = 1

r = get_r(H, T, n)
```

In [23]:
```
%%timeit
euler_mf_1 = multifactor_euler(n_steps, n_fact, H, r, method=1)
```

```
1.2 ms ± 41.1 µs per loop (mean ± std. dev. of 7 runs, 1,000 loops each)
```

In [108…
```
%%timeit
euler_mf_2 = multifactor_euler(n_steps, n_fact, H, r, method=2)
```

```
1.24 ms ± 38 µs per loop (mean ± std. dev. of 7 runs, 1,000 loops each)
```

In [109…
```
%%timeit
euler_mf_3 = multifactor_euler(n_steps, n_fact, H, r, method=3)
```

```
1.27 ms ± 104 µs per loop (mean ± std. dev. of 7 runs, 1,000 loops each)
```

In [110…
```
%%timeit
euler_mf_4 = multifactor_euler_4(n_steps, n_fact, H, r)
```

```
1.88 ms ± 477 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

In [111…
```
%%timeit
euler_mf_5 = multifactor_euler_5(n_steps, n_fact, H, r)
```

```
3.5 ms ± 965 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

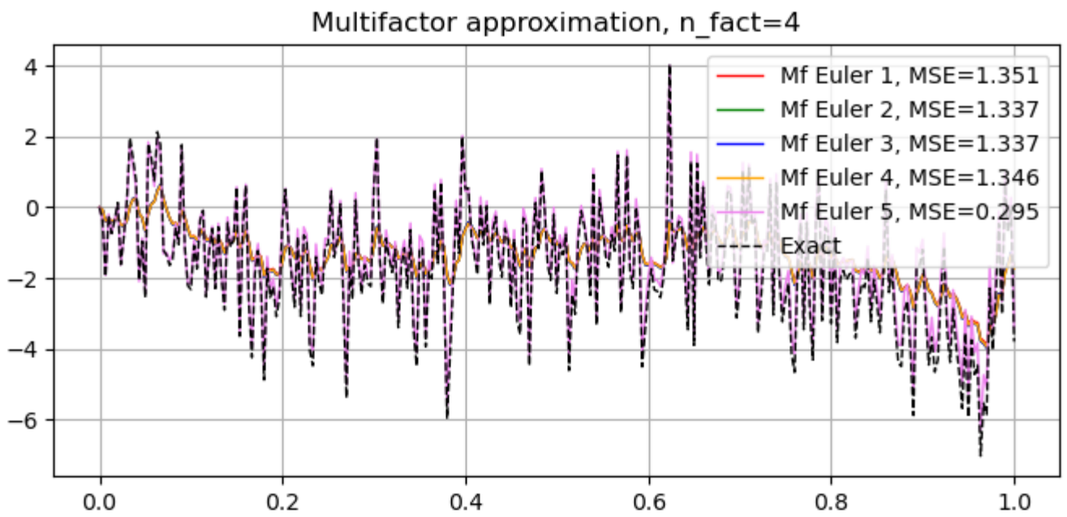
In [112…
```
%%timeit
X_exact = exact_cholesky(n_steps, H)
```

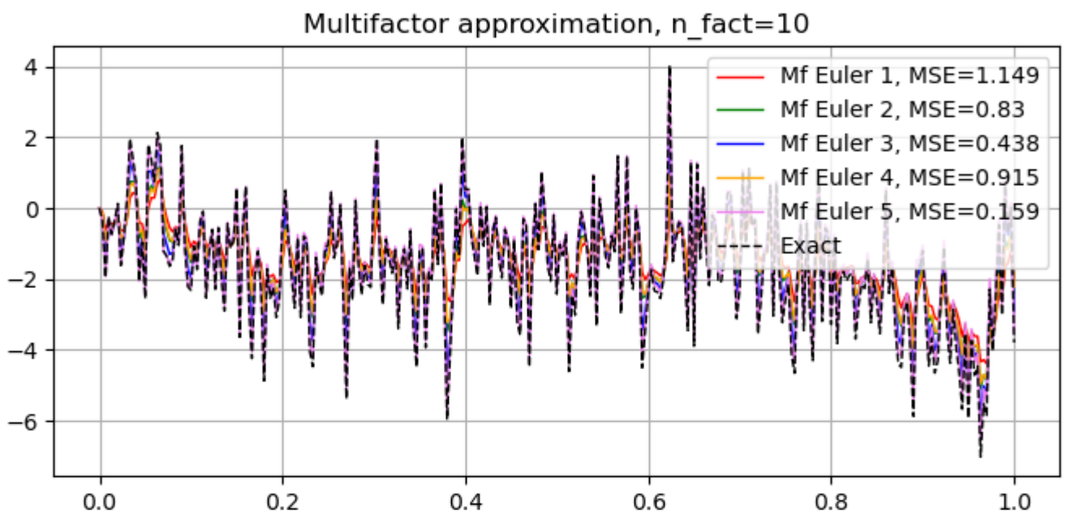
```
695 ms ± 78.4 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

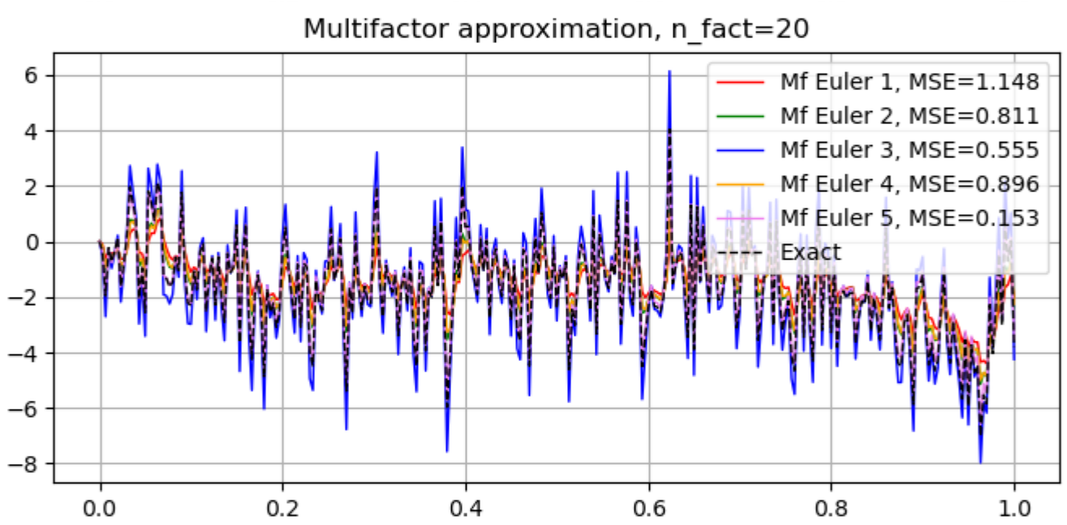
Changing the number of factors

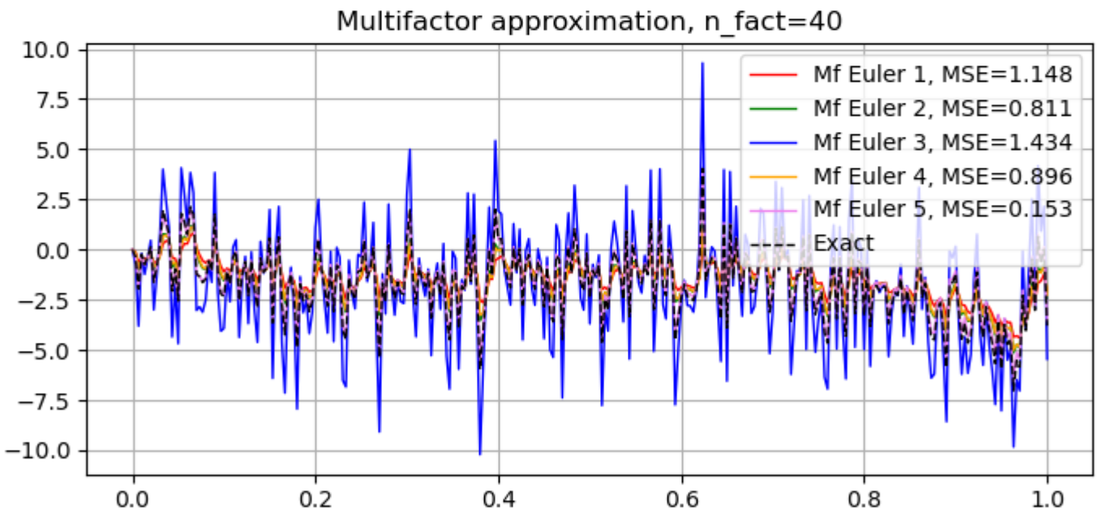
In [26]:
```
H = 0.05
n_steps = 300
T = 1

r = 8.83
```

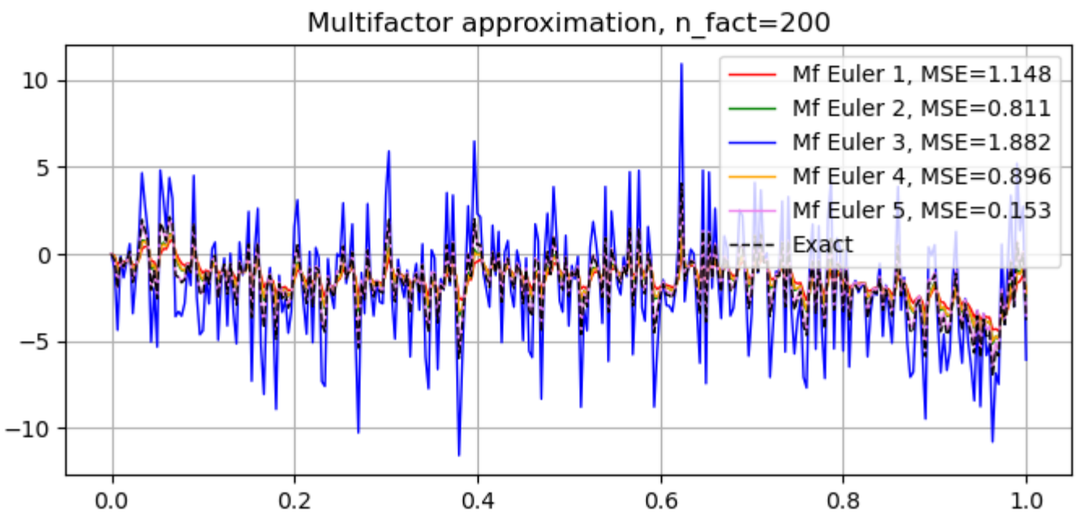
In [27]:
```
n_fact_range = [4, 10, 20, 40, 200]

fig, axes = plt.subplots(5, 1, figsize=(8, 20))

for n_fact, ax in zip(n_fact_range, axes):
    t_grid = np.linspace(0, T, n_steps + 1)

    euler_mf_1 = multifactor_euler(n_steps, n_fact, H, r, method=1)
    euler_mf_2 = multifactor_euler(n_steps, n_fact, H, r, method=2)
    euler_mf_3 = multifactor_euler(n_steps, n_fact, H, r, method=3)
    euler_mf_4 = multifactor_euler_4(n_steps, n_fact, H, r)
    euler_mf_5 = multifactor_euler_5(n_steps, n_fact, H, r)
    X_exact = exact_cholesky(n_steps, H)

    ax.plot(t_grid, euler_mf_1, 'r', lw=1, label=f'Mf Euler 1, MSE={np.round(mse(X_exact, euler_mf_1), 3)}')
    ax.plot(t_grid, euler_mf_2, 'g', lw=1, label=f'Mf Euler 2, MSE={np.round(mse(X_exact, euler_mf_2), 3)}')
    ax.plot(t_grid, euler_mf_3, 'b', lw=1, label=f'Mf Euler 3, MSE={np.round(mse(X_exact, euler_mf_3), 3)}')
    ax.plot(t_grid, euler_mf_4, 'orange', lw=1, label=f'Mf Euler 4, MSE={np.round(mse(X_exact, euler_mf_4), 3)}')
    ax.plot(t_grid, euler_mf_5, 'violet', lw=1, label=f'Mf Euler 5, MSE={np.round(mse(X_exact, euler_mf_5), 3)}')
    ax.plot(t_grid, X_exact, 'k--', lw=1, label='Exact')


    ax.legend(loc='upper right')
    ax.set_title(f'Multifactor approximation, n_fact={n_fact}')
    ax.grid()
```











Conclusions

- For time efficiency, same situation as in part 2: approximately the same time for all the Euler schemes, which are all much faster than exact simulations.
- Scheme 5 shows the best performance in terms of MSE and stability for the rough processes.
- Analysis of the number of factors shows that for most of the schemes it's enough to take ~10-20 factors. However, scheme 3 is prone to large errors due to too much oscillations for the number of factors greater than 20 (in the case $H = 0.01$).
- Multi-factor approximations can satisfyingly reproduce the fast oscillations of the rough processes, which was impossible for the standard Euler schemes of the previous part.
- Our numerical experiments showed better results when using $r = 8.8$ rather than the theoretical optimal value got by optimisation.

3.2 Multifactor exact simulation with Cholesky

$$X_t \approx X_0 + \nu \sum_{k=1}^{n} c_k Y_t^k$$

with

$$Y_t^k = \int_0^t e^{-x_k(t-s)} dW_s$$

$$Y_{t_i}^k = e^{-x_k h} Y_t^k + \xi_i^k, \quad \xi_i^k = \int_{t_{i-1}}^{t_i} e^{-x_k(t_i - s)} dW_s$$

We will use exact approximation using Cholseky to simulate $(\xi_i^1, \dots, \xi_i^n)^\top \sim \mathcal{N}(0, \Sigma)$ with

$$\Sigma_{kl} = \int_{t_i}^{t_{i+1}} e^{-(x_k + x_l)(t_{i+1} - s)} ds = \frac{1 - e^{-(x_k + x_l)dt}}{x_k + x_l}$$

For each t_i generate $Z_i = (Z_i^1, \dots, Z_i^n)^\top$ independant standard Gaussian and set

$$\xi_i = LZ_i \quad \text{with } LL^\top = \Sigma.$$

Set

$$E_{dt} = \exp(-\mathrm{diag}(x_1, \dots, x_n)dt)$$

Then,

$$X_{t_{i+1}} = X_0 + \nu * c^\top E_{dt} Y_{t_i} + \nu * c^\top L Z_i = X_0 + \nu * c^\top E_{dt} Y_{t_i} + \nu * \sqrt{c^\top \Sigma c} U_i$$

with

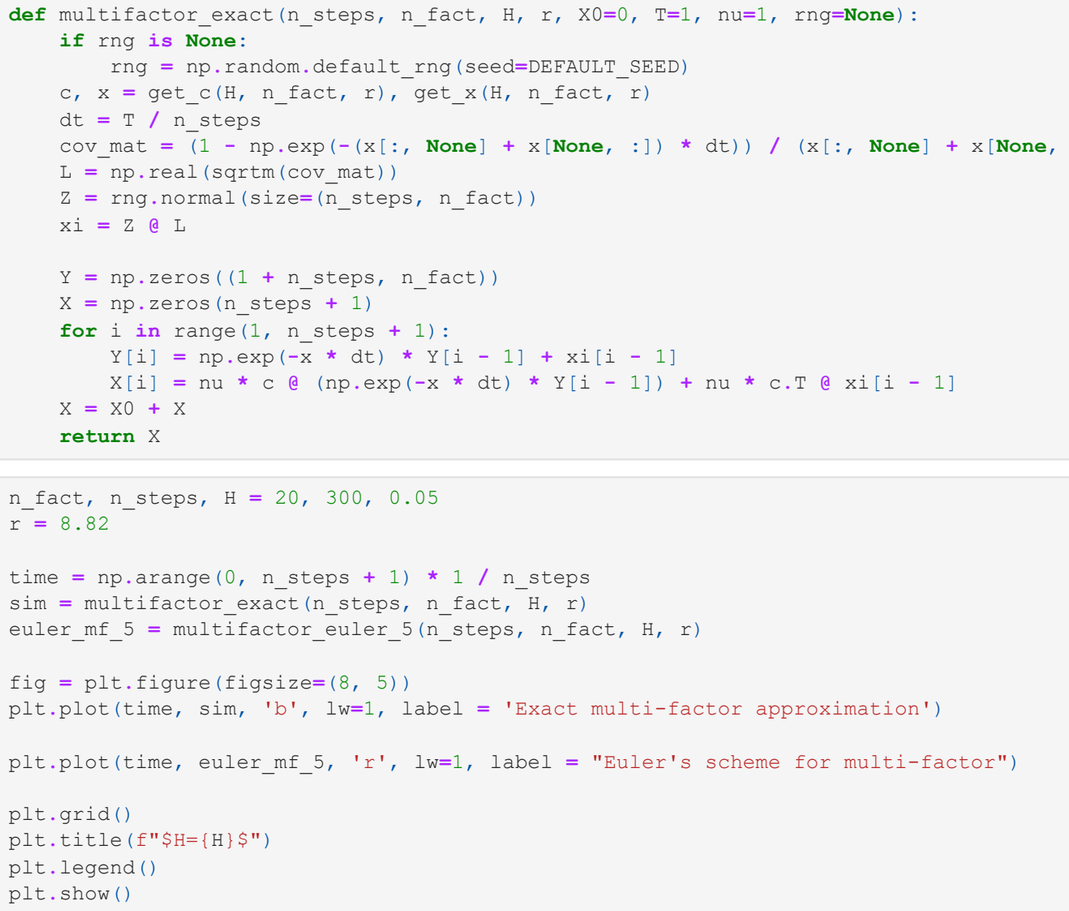
$$U_i := \frac{c^\top L Z_i}{\sqrt{c^\top \Sigma c}} \sim \mathcal{N}(0, 1)$$

Q: What is the difference and main advantage of such method compared to Cholesky of part 1?

A: At each step t_i we just simulate a gaussian vector of length n_fact, which is much smaller than n_steps. In fact, this method allows the complexity to depend *linearly* on n_steps, whereas the complexity of the first algorithm is worse than quadratic.

In [28]:
```
from scipy.linalg import sqrtm

def multifactor_exact(n_steps, n_fact, H, r, X0=0, T=1, nu=1, rng=None):
    if rng is None:
        rng = np.random.default_rng(seed=DEFAULT_SEED)
    c, x = get_c(H, n_fact, r), get_x(H, n_fact, r)
    dt = T / n_steps
    cov_mat = (1 - np.exp(-(x[:, None] + x[None, :]) * dt)) / (x[:, None] + x[None, :])
    L = np.real(sqrtm(cov_mat))
    Z = rng.normal(size=(n_steps, n_fact))
    xi = Z @ L

    Y = np.zeros((1 + n_steps, n_fact))
    X = np.zeros(n_steps + 1)
    for i in range(1, n_steps + 1):
        Y[i] = np.exp(-x * dt) * Y[i - 1] + xi[i - 1]
        X[i] = nu * c @ (np.exp(-x * dt) * Y[i - 1]) + nu * c.T @ xi[i - 1]
    X = X0 + X
    return X
```

In [29]:
```
n_fact, n_steps, H = 20, 300, 0.05
r = 8.82

time = np.arange(0, n_steps + 1) * 1 / n_steps
sim = multifactor_exact(n_steps, n_fact, H, r)
euler_mf_5 = multifactor_euler_5(n_steps, n_fact, H, r)

fig = plt.figure(figsize=(8, 5))
plt.plot(time, sim, 'b', lw=1, label = 'Exact multi-factor approximation')

plt.plot(time, euler_mf_5, 'r', lw=1, label = "Euler's scheme for multi-factor")

plt.grid()
plt.title(f"$H={H}$")
plt.legend()
plt.show()
```



Here the trajectories do not coincide as, using the same random numbers, we simulate a vector of the BM's increments for Euler and n_fact-dimensional vector of Itô integrals for the exact simulation. So, the trajectories on this graph should be approximately equal in law, but not a.s.

4. Going beyond

Simulation is not a one-size-fits-all solution. Break free from the mold and discover new methods to solve the problem at hand.

Explain how the above method can be adapted to the shifted kernel

$$K_\epsilon(t,s) = \frac{1}{\Gamma(H+1/2)}(\epsilon + t - s)^{H-1/2}1_{s<t}$$

Notice that now $H \in (-\infty, \infty)$. Why? Because now, thanks to the shift, the convolution kernel becomes integrable at 0 for any exponent, *i.e.* for any value of H.

Study the impact of $\epsilon > 0$ on the schemes. You can make epsilon vary between 0 and $1/52$, also for $\epsilon > 0$ you can test with H varying between -1 and 0.5

To go beyond, we note that the key fact used for the multi-factor approximation was the Laplace transform of the kernel:

$$K(t) = \frac{t^{H-0.5}}{\Gamma(H+0.5)} = \int_0^\infty e^{-xt}\mu(dx),$$

where $\mu(dx) = \dfrac{x^{-H-0.5}}{\Gamma(0.5-H)\Gamma(H+0.5)}$. We also notice that

$$K_\epsilon(t) = \frac{(t+\epsilon)^{H-0.5}}{\Gamma(H+0.5)} = \int_0^\infty e^{-x(t+\epsilon)}\mu(dx)$$

for any $t > 0$. So, in order to approximate the Volterra process with shifted kernel one should use OU factors with shifted exponential kernels $e^{-x_k(t-s+\epsilon)}$:

$$X_t \approx X_0 + \nu \sum_{k=1}^n c_k Y_t^{k,\epsilon}$$

with

$$Y_t^{k,\epsilon} = \int_0^t e^{-x_k(\epsilon+t-s)}dW_s$$

$$Y_{t_i}^{k,\epsilon} = e^{-x_k h}Y_{t_{i-1}}^{k,\epsilon} + \xi_i^k, \quad \xi_i^k = \int_{t_{i-1}}^{t_i} e^{-x_k(\epsilon+t_i-s)}dW_s$$

The schemes can be easily modified as following:

1. **Factor-Euler 1** :

$$\xi_i^k \approx e^{-x_k(\epsilon+dt)}\sqrt{dt}Z_i$$

2. **Factor-Euler 2**: writing $dW_s = Z_i ds/\sqrt{dt}$

$$\xi_i^k \approx \frac{1}{\sqrt{dt}}\int_{t_{i-1}}^{t_i} e^{-x_k(\epsilon+t_i-s)}ds Z_i = \frac{1}{\sqrt{dt}}\frac{e^{-x_k\epsilon} - e^{-x_k(\epsilon+dt)}}{x_k}Z_i$$

3. **Factor-Euler 3**: using that ξ_i^k is gaussian with variance $\frac{e^{-2x_k\epsilon} - e^{-2x_k(\epsilon+dt)}}{2x_k}$, so that

$$\xi_i^k \approx \sqrt{\frac{e^{-2x_k\epsilon} - e^{-2x_k(\epsilon+dt)}}{2x_k}}Z_i$$

4. **Factor-Euler 4**: implicit scheme as in lifting heston paper in the appendix.

$$Y_{t_i}^{k,\epsilon} \approx \frac{1}{1+x_k dt}(Y_{t_{i-1}} + e^{-x_k\epsilon}\sqrt{dt}Z_i)$$

5. **Factor-Euler 5**: modified variance:

$$X_{t_{i+1}} = X_0 + \nu\sum_k c_k e^{-x_k dt}Y_{t_i}^k + \nu\int_{t_i}^{t_{i+1}} K_n(t_{i+1},s)dW_s$$

approximate second term by variance of original kernel K.

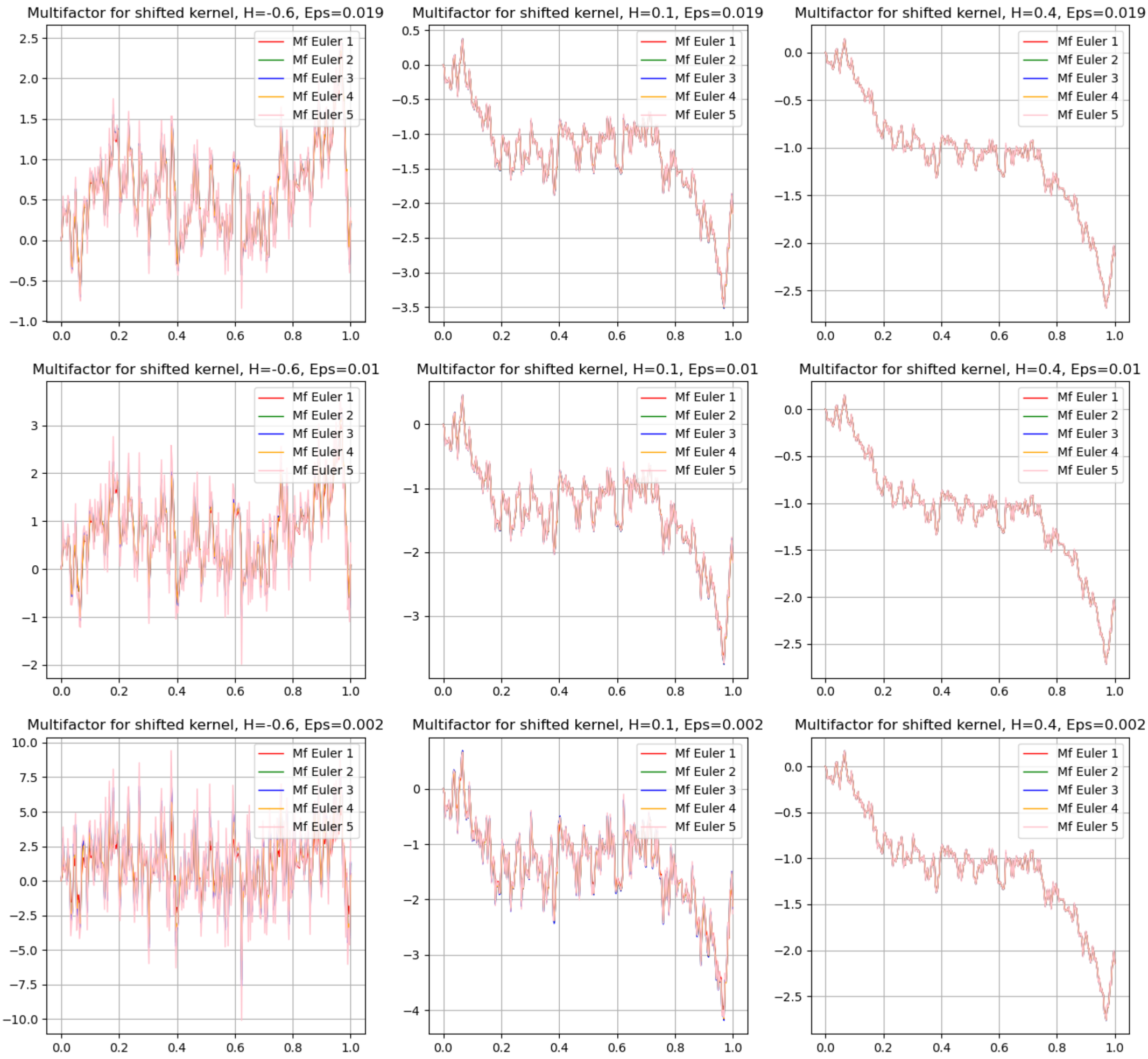
$$\int_{t_i}^{t_{i+1}} K_n(t_{i+1},s)dW_s \approx \sqrt{\int_{t_i}^{t_{i+1}} K_\epsilon(t_{i+1},s)^2 ds}Z_i = \sqrt{\frac{(\epsilon+dt)^{2H} - \epsilon^{2H}}{2H}}\frac{1}{\Gamma(H+0.5)}Z_i$$

In [30]:
```
def get_xi_shifted(x, Z, dt, eps, method):
    xi = np.zeros((n_steps, n_fact))
    if method == 1:
        xi = np.exp(-x[None, :] * (eps + dt)) * np.sqrt(dt) * Z[:, None]
    elif method == 2:
        xi = ( np.exp(-x[None, :] * eps) - np.exp(-x[None, :] * (eps + dt)) ) / x[None, :] / np.sqrt(dt) * Z[:, None]
    elif method == 3:
        xi = np.sqrt( ( np.exp(-2 * x[None, :] * eps) - np.exp(-2 * x[None, :] * (eps + dt)) ) / x[None, :] / 2 ) * Z[:, None]
    return xi

def multifactor_euler_shifted(n_steps, n_fact, H, r, method, X0=0, T=1, eps=1/52, nu=1, rng=None):
    """
    Implementation of the first 3 schemes. (method = 1, 2, 3 correspondingly).
    """
    if rng is None:
        rng = np.random.default_rng(seed=DEFAULT_SEED)
    c, x = get_c(H, n_fact, r), get_x(H, n_fact, r)
    dt = T / n_steps
    Z = rng.normal(size=n_steps)
    xi = get_xi_shifted(x, Z, dt, eps, method)
    Y = np.zeros((1 + n_steps, n_fact))
    for i in range(1, n_steps + 1):
        Y[i] = np.exp(-x * dt) * Y[i - 1] + xi[i - 1]
    X = X0 + nu * Y @ c
    return X
```

In [39]:
```
def multifactor_euler_4_shifted(n_steps, n_fact, H, r, X0=0, T=1, eps=1/52, nu=1, rng=None):
    if rng is None:
        rng = np.random.default_rng(seed=DEFAULT_SEED)
    c, x = get_c(H, n_fact, r), get_x(H, n_fact, r)
    dt = T / n_steps
    Z = rng.normal(size=n_steps)
    Y = np.zeros((1 + n_steps, n_fact))
    for i in range(1, n_steps + 1):
        Y[i] = Y[i - 1] / (1 + x * dt) + np.sqrt(dt) * np.exp(-x * eps) * Z[i - 1] / (1 + x * dt)
    X = X0 + nu * Y @ c
    return X

def multifactor_euler_5_shifted(n_steps, n_fact, H, r, X0=0, T=1, eps=1/52, nu=1, rng=None):
    if rng is None:
        rng = np.random.default_rng(seed=DEFAULT_SEED)
    c, x = get_c(H, n_fact, r), get_x(H, n_fact, r)
    dt = T / n_steps
    Z = rng.normal(size=n_steps)
    xi = get_xi_shifted(x, Z, dt, eps, method=1)
    Y = np.zeros((1 + n_steps, n_fact))
    v = np.sqrt(((eps + dt)**(2*H) - eps**(2*H)) / (2*H)) / gamma(H + 0.5)
    X = np.zeros(n_steps + 1)
    for i in range(1, n_steps + 1):
        Y[i] = np.exp(-x * dt) * Y[i - 1] + xi[i - 1]
        X[i] = nu * c @ (np.exp(-x * dt) * Y[i - 1]) + nu * v * Z[i - 1]
    X = X0 + X
    return X
```

In [39]:
```
R_range = np.ones(3) * 8.82

H = 0.1
n_fact = 20
n_steps = 300
H_range = [-0.6, 0.1, 0.4]
```

```
eps_range = np.round([1/52, 0.5 / 52, 0.1 / 52], 3)
t_grid = np.linspace(0, T, n_steps + 1)
fig, ax = plt.subplots(3,3, figsize=(16, 15))


for i in range(3):
    for j in range(3):
        euler_mf_1 = multifactor_euler_shifted(n_steps, n_fact, H_range[j], R_range[j], eps=eps_range[i], method=1)
        euler_mf_2 = multifactor_euler_shifted(n_steps, n_fact, H_range[j], R_range[j], eps=eps_range[i], method=2)
        euler_mf_3 = multifactor_euler_shifted(n_steps, n_fact, H_range[j], R_range[j], eps=eps_range[i], method=3)
        euler_mf_4 = multifactor_euler_4_shifted(n_steps, n_fact, H_range[j], R_range[j], eps=eps_range[i])
        euler_mf_5 = multifactor_euler_5_shifted(n_steps, n_fact, H_range[j], R_range[j], eps=eps_range[i])
        #X_exact = exact_cholesky(n_steps, H_range[j])

        ax[i,j].plot(t_grid, euler_mf_1, 'r', lw=1, label='Mf Euler 1')
        ax[i,j].plot(t_grid, euler_mf_2, 'g', lw=1, label='Mf Euler 2')
        ax[i,j].plot(t_grid, euler_mf_3, 'b', lw=1, label='Mf Euler 3')
        ax[i,j].plot(t_grid, euler_mf_4, 'orange', lw=1, label='Mf Euler 4')
        ax[i,j].plot(t_grid, euler_mf_5, 'pink', lw=1, label='Mf Euler 5')
        #ax[i,j].plot(t_grid, X_exact, 'k--', lw=1, label='Non shifted')

        ax[i,j].legend(loc='upper right')
        ax[i,j].set_title(f'Multifactor for shifted kernel, H={H_range[j]}, Eps={eps_range[i]}')
        ax[i,j].grid()
```



Exact simulation for multi-factor approximation

The exact simulation scheme is also modified to take into account the new covariance matrix

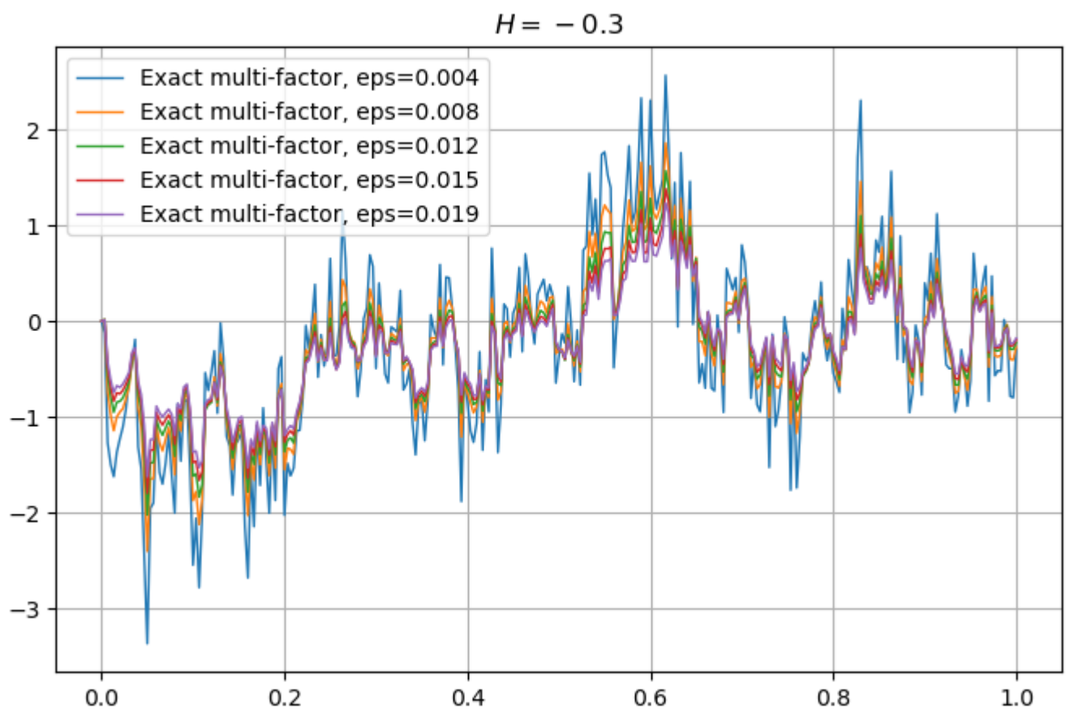
$$\Sigma^{\epsilon}_{kl} = \int_{t_i}^{t_{i+1}} e^{-(x_k+x_l)(\epsilon+t_{i+1}-s)} ds = \frac{e^{-(x_k+x_l)\epsilon} - e^{-(x_k+x_l)(\epsilon+dt)}}{x_k + x_l}$$

In [41]:
```
def multifactor_exact_shifted(n_steps, n_fact, H, r, eps, X0=0, T=1, nu=1, rng=None):
    if rng is None:
        rng = np.random.default_rng(seed=DEFAULT_SEED)
    c, x = get_c(H, n_fact, r), get_x(H, n_fact, r)
    dt = T / n_steps
    cov_mat = (np.exp(-(x[:, None] + x[None, :]) * eps) - np.exp(-(x[:, None] + x[None, :]) * (dt + eps))) / (x[:, None] + x[None, :])
    L = np.real(sqrtm(cov_mat))

    Z = rng.normal(size=(n_steps, n_fact))
    xi = Z @ L

    Y = np.zeros((1 + n_steps, n_fact))
    X = np.zeros(n_steps + 1)
    for i in range(1, n_steps + 1):
        Y[i] = np.exp(-x * dt) * Y[i - 1] + xi[i - 1]
        X[i] = nu * c @ (np.exp(-x * dt) * Y[i - 1]) + nu * c.T @ xi[i - 1]
    X = X0 + X
    return X
```

Trajectories with the same random numbers, but with different eps

In [42]:
```
n_fact, n_steps, H = 20, 300, -0.3
r = 8.82

eps_range = np.linspace(0, 1, 6)[1:] / 52

time = np.arange(0, n_steps + 1) * 1 / n_steps
```

```
fig = plt.figure(figsize=(8, 5))

for eps in eps_range:
    sim = multifactor_exact_shifted(n_steps, n_fact, H, r, eps=eps)
    plt.plot(time, sim, lw=1, label = f'Exact multi-factor, eps={np.round(eps, 3)}')

plt.grid()
plt.title(f"$H={H}$")
plt.legend()
plt.show()
```



Conclusions

- Shifted kernels lead to a more regular behavior of the trajectories (less oscillations since the kernel does not explode at $t = s$).
- All the Euler schemes generate similar trajectories.
- $H < 0$ leads to the "super-rough" behavior and the appearence of "peaks" if ϵ is small enough.