Projeto Computational 2

Métodos Computacionais em Finanças

2024/2025

Afonso da Conceição Ribeiro (ist1102763)

Mestrado em Engenharia e Ciência de Dados

Instituto Superior Técnico – Universidade de Lisboa

Índice

1.

In order to allow the implementation of the Matlab functions for generating realizations of following distributions $U([a,b])$, $Exp(\theta)$ and $N(0,1)$ using Box-Muller method, the function `lcg_uniform` was created, corresponding to the implementation of the linear congruential generator, which will be needed in the following functions.

Implementation of `lcg_uniform`:

Input: `N` size of the sample to be generated

`seed` the seed to be used

Output: A vector of size `N` with realizations of the uniform distribution `U([0,1])`. The choices of `M`, `a` and `b` are according to the common choice: $M = 2^{31} - 1$, $a = 16807$ and $b = 0$.

```
function U = lcg_uniform(N, seed)
    M = 2^31 - 1;
    a = 16807;
    b = 0;

    m = zeros(N,1);
    m(1) = mod(seed, M); % garante 0 < m1 < M

    for k = 2:N
        m(k) = mod(a * m(k-1) + b, M);
    end

    U = m / M;
end
```

Implementation of `rand_uniform.m`:

Input: `a` lower bound of the uniform distribution

`b` upper bound of the uniform distribution

`N` size of the sample to be generated

`seed` the seed to be used

Output: A vector of size `N` with realizations of the uniform distribution `U([a,b])`.

```
function X = rand_uniform(a,b,N,seed)
    if nargin < 4, seed = 12345; end
    U = lcg_uniform(N, seed);
    X = a + (b-a) .* U;
end
```

Implementation of `rand_exponential.m`:

Input: `theta` parameter of the exponential distribution

`N` size of the sample to be generated

`seed` the seed to be used

Output: A vector of size `N` with realizations of the exponential distribution $Exp(\theta)$.

```
function X = rand_exponential(theta,N,seed)
    if nargin < 3, seed = 12345; end
    U = lcg_uniform(N, seed);
    X = -theta * log(U);
end
```

Implementation of `randn_boxmuller.m`:

Input: `N` size of the sample to be generated

`seed` the seed to be used

Output: A vector of size `N` with realizations of the normal distribution `N(0,1)`.

```
function Z = randn_boxmuller(N, seed)
    if nargin < 2, seed = 12345; end
    U = lcg_uniform(N, seed);
    U1 = U(1:floor(N/2));
    U2 = U(floor(N/2)+1:end);

    R     = sqrt(-2 .* log(U1));
    Theta = 2*pi .* U2;

    Z = [R .* cos(Theta); R .* sin(Theta)];

end
```

Implementation of `halton2d.m`:

Input: `N` size of the sample to be generated

Output: A vector of size `N` with realizations of the normal distribution `N(0,1)`.

```
function H = halton2d(N)
    base = [2,3];
    H = zeros(N,2);
    for i = 1:N
        n = i;
        for d = 1:2
            f = 1/base(d);
            x = 0;
            while n > 0
                x = x + mod(n, base(d)) * f;
                n = floor(n/base(d));
                f = f/base(d);
            end
            H(i,d) = x;
            n = i;
        end
    end
end
```

2.

In the Monte Carlo and quasi-Monte Carlo methods, we will evaluate the function $\chi(X, Y)$ at random/Halton points, which, in general, do not coincide with any node of the mesh. The function `chi_mandelbrot` solves this problem: uses bilinear interpolation over the four nodes of the mesh that surround the point (x, y).

Implementation of `chi_mandelbrot`:

Input: **`x`** vector of real numbers

`y` vector of real numbers

`M` matrix of boolean values, loaded from the file `mandelbrot.mat`

Output: A vector of size `N` with realizations of the uniform distribution `U([0,1])`. The choices of `M`, `a` and `b` are according to the common choice: $M = 2^{31} - 1$, $a = 16807$ and $b = 0$.

```matlab
function v = chi_mandelbrot(x, y, M)
    n = size(M,1);
    xi = x * (n-1) + 1;
    yi = y * (n-1) + 1;
    i  = floor(xi);
    j  = floor(yi);
    i(i < 1) = 1;
    i(i > n - 1) = n - 1;
    j(j < 1) = 1;
    j(j > n - 1) = n - 1;

    dx = xi - i;
    dy = yi - j;

    v00 = M( sub2ind([n n], i    , j    ) );
    v10 = M( sub2ind([n n], i + 1, j    ) );
    v01 = M( sub2ind([n n], i    , j + 1) );
    v11 = M( sub2ind([n n], i + 1, j + 1) );

    v = (1-dx).*(1-dy).*v00 + dx.*(1-dy).*v10 + (1-dx).*dy.*v01 + dx.*dy.*
   v11;
end
```

The following script is used to estimate the area of the Mandelbrot set using the Monte Carlo and quasi-Monte Carlo methods.

```matlab
clear; clc;

%% (i) Load Mandelbrot
D   = load('mandelbrot.mat');
fn  = fieldnames(D);
M   = D.(fn{1});
nGrid = size(M,1);

%% (ii) visualize fractal
figure('Color','w');
imagesc([0 1],[0 1],flipud(M'));
set(gca,'YDir','normal');
axis image;
colormap(flipud(gray)); clim([0 1]);
grid on;
```

```

%% (iii) parameters
N       = 1e5;
seedMC = 54321;

%% (iv) Monte Carlo - uniform points via LCG
U = lcg_uniform(2*N, seedMC);
P = reshape(U, [N 2]);
chiMC    = chi_mandelbrot(P(:,1), P(:,2), M);
areaMC   = mean(chiMC);
stderrMC = sqrt(var(chiMC)/N);

%% (v) Quasi-Monte Carlo - Halton sequence
H = halton2d(N);
chiQMC   = chi_mandelbrot(H(:,1), H(:,2), M);
areaQMC  = mean(chiQMC);

%% (vi) Results in the prompt
fprintf('--- Estimate of the Mandelbrot area ---\n');
fprintf('Monte Carlo Estimate   : %.6f    (   %.6f)\n', areaMC, stderrMC);
fprintf('Quasi-Monte Carlo Estimate (Halton): %.6f\n', areaQMC);
```

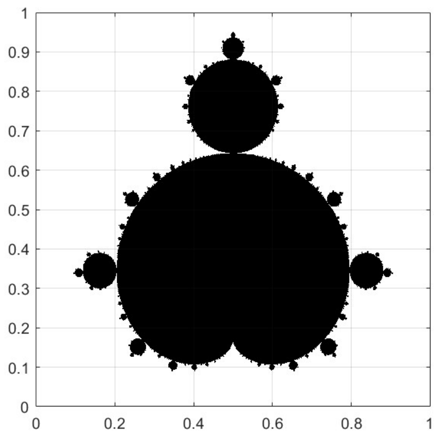


Figura 1: Mandelbrot fractal

The output obtained for the area of the Mandelbrot fractal in the file `mandelbrot.mat` is:

```
--- Estimate of the Mandelbrot area ---
Monte Carlo Estimate   : 0.325765    (± 0.001481)
Quasi-Monte Carlo Estimate (Halton): 0.326508
```

3.

Implementation of `euler_maruyama.m`

Input: `a` drift-handle `@(t,x)` ...

`b` diffusion-handle `@(t,x)` ...

`X0` initial value $X(0)$

`T` final time

`N` number of steps

Output: `t_grid` vector $(0, h, \dots, T)$

`X` Euler–Maruyama path of size $N+1$

```
function [t_grid,X] = euler_maruyama(a,b,X0,T,N,dW)
    if nargin < 6
        dW = sqrt(T/N)*randn_boxmuller(N);
    end

    h      = T/N;
    t_grid = linspace(0,T,N+1).';
    X      = zeros(N+1,1);   X(1)=X0;

    for n = 1:N
        t        = t_grid(n);
        X(n+1)   = X(n) + a(t,X(n))*h + b(t,X(n))*dW(n);
    end
end
```

Implementation of `milstein.m`

Input: identical to `euler_maruyama.m` with an extra `b_deriv` - handle for $\partial b/\partial x$.

Output: grid and Milstein path on the same nodes.

```
function [t_grid,X] = milstein(a,b,b_deriv,X0,T,N,dW)
    if nargin < 7
        dW = sqrt(T/N)*randn_boxmuller(N);
    end

    h      = T/N;
    t_grid = linspace(0,T,N+1).';
    X      = zeros(N+1,1);   X(1)=X0;

    for n = 1:N
        t      = t_grid(n);
        x      = X(n);
        dw     = dW(n);
        X(n+1) = x + a(t,x)*h + b(t,x)*dw ...
                   + 0.5*b(t,x)*b_deriv(t,x)*(dw^2-h);
    end
end
```

4.

(a)

We consider the geometric Brownian motion (GBM)

$$dS(t) = \mu\, S(t)\, dt + \sigma\, S(t)\, dB(t), \qquad S(0) = S_0,$$

whose exact solution is

$$S(t) = S_0 \exp\Big(\big(\mu - \tfrac{1}{2}\sigma^2\big)t + \sigma B(t)\Big).$$

Throughout we use

$$\mu = 0.5, \quad \sigma = 0.3, \quad S_0 = 1, \quad T = 1.$$

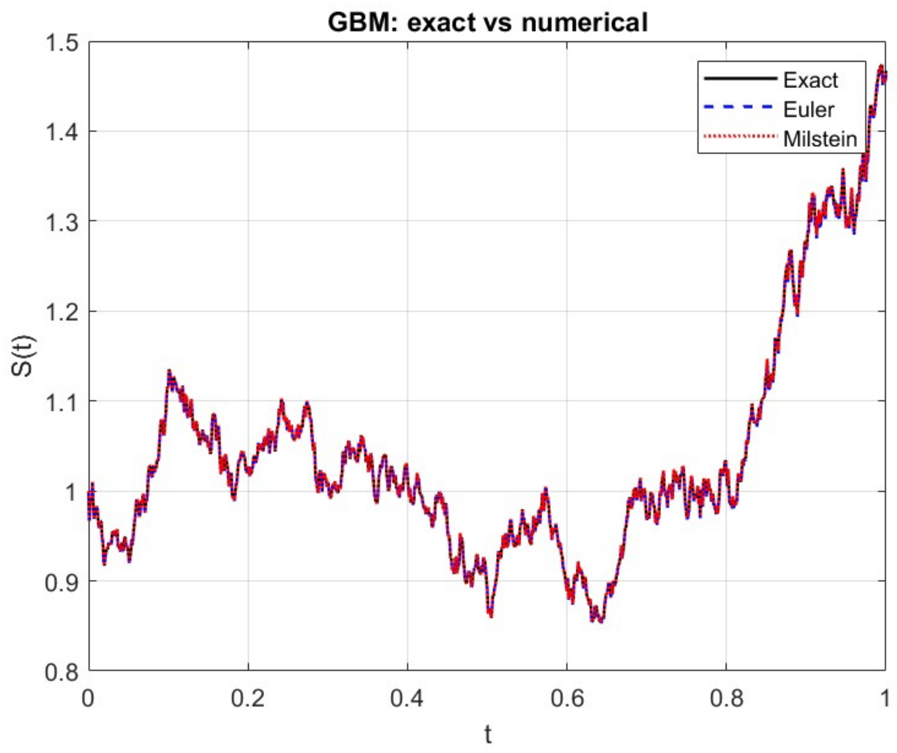
With step-size $h = 10^{-3}$ ($N = 1000$ steps) we simulate one GBM trajectory and two numerical approximations driven by the *same* Brownian increments $\Delta W_n \sim \mathcal{N}(0, h)$:

- **Euler-Maruyama** (order 1/2 strong): $S_{n+1} = S_n + \mu S_n h + \sigma S_n \Delta W_n$.
- **Milstein** (order 1 strong): $S_{n+1} = S_n + \mu S_n h + \sigma S_n \Delta W_n + \frac{1}{2}\sigma^2 S_n \big(\Delta W_n^2 - h\big)$.

The following script is used to plot the exact solution and the numerical solutions obtained by Euler-Maruyama and Milstein methods.

```matlab
clear; clc; close all;

T = 1;
h = 1e-3;
N = T/h;
mu = 0.5;
sigma = 0.3;
S0 = 1;

a = @(t,x) mu*x;
b = @(t,x) sigma*x;
b_x = @(t,x) sigma;

dW = sqrt(h) * randn_boxmuller(N,12345);

[ t, SE ] = euler_maruyama(a,b,S0,T,N,dW);
[ ~, SM ] = milstein(a,b,b_x,S0,T,N,dW);

W      = cumsum([0; dW]);                    % W_0=0
SExact = S0*exp((mu-0.5*sigma^2)*t + sigma*W);

plot(t,SExact,'k', t,SE,'b--', t,SM,'r:','LineWidth',1.2);
xlabel('t'); ylabel('S(t)');
legend('Exact','Euler','Milstein');
title('GBM: exact vs numerical');
grid on;
```

The output obtained is:



Because h is very small, the Milstein path (red dotted) is visually indistinguishable from the exact solution (black), while Euler (blue dashed) deviates only slightly, fully consistent with the different strong orders.

(b)

We estimate strong and weak convergence orders for Euler-Maruyama and Milstein using

$$h_k = 0.005\,(1/2)^k, \qquad k = 0, 1, 2, 3$$

and 5×10^5 Monte-Carlo paths for every h_k.
Error definitions:

$$\text{strong error} = \mathbb{E}\Big[|S_T - S_T^{(h)}|\Big], \qquad \text{weak error} = \Big|\mathbb{E}[S_T] - \mathbb{E}[S_T^{(h)}]\Big|.$$

The convergence study is produced by the following script:

```
clear; clc;

T = 1;
mu = 0.5;
sigma = 0.3;
S0 = 1;
Nsim = 5e5;
hs = 0.005*(0.5).^(0:3);

a = @(t,x) mu*x;
b = @(t,x) sigma*x;
b_x = @(t,x) sigma;
batch = 1e4;

errS_E = zeros(size(hs));  errS_M = errS_E;
errW_E = errS_E;           errW_M = errS_E;

for k = 1:numel(hs)
    h = hs(k);
  M = round(T/h);
    sumAbsE=0; sumAbsM=0; sumSE=0; sumSM=0; sumSX=0;
    sims = 0;

    while sims < Nsim
        m   = min(batch,Nsim-sims);
        Z   = randn_boxmuller(M*m, 12345+17*k+13*sims);
        Z   = reshape(Z,M,m);
        dW  = sqrt(h).*Z;

        SE  = zeros(m,1);  SM = SE;  SX = SE;
        for j=1:m
            [~,SEj] = euler_maruyama(a,b,S0,T,M,dW(:,j));
            [~,SMj] = milstein     (a,b,b_x,S0,T,M,dW(:,j));
            W       = sum(dW(:,j));
            SX(j)   = S0*exp((mu-0.5*sigma^2)*T + sigma*W);
            SE(j)   = SEj(end);   SM(j)=SMj(end);
        end

        sumAbsE  = sumAbsE+sum(abs(SE-SX));
        sumAbsM  = sumAbsM+sum(abs(SM-SX));
        sumSE    = sumSE+sum(SE);  sumSM=sumSM+sum(SM); sumSX=sumSX+sum(SX);
        sims     = sims+m;
    end
    errS_E(k)=sumAbsE/Nsim;   errS_M(k)=sumAbsM/Nsim;
    meanX    = sumSX/Nsim;
    errW_E(k)=abs(sumSE/Nsim-meanX);
    errW_M(k)=abs(sumSM/Nsim-meanX);
end
```

```

ord   = @(e) -diff(log(e))./log(2);
fprintf('\n    h          strong-E      strong-M       weak-E        weak-M\n');
for k=1:numel(hs)
    fprintf('%8.5f   %10.4e %10.4e %10.4e %10.4e\n', ...
            hs(k),errS_E(k),errS_M(k),errW_E(k),errW_M(k));
end
fprintf('\nEstimated orders (using last three hs):\n');
fprintf('Euler-Maruyama      strong %.3f   |   weak %.3f\n',mean(ord(errS_E)),
    mean(ord(errW_E)));
fprintf('Milstein   strong %.3f   |   weak %.3f\n',mean(ord(errS_M)),mean(ord(
    errW_M)));
```

The numerical results are:

h	strong-E	strong-M	weak-E	weak-M
0.00500	6.0185×10^{-3}	1.3125×10^{-3}	1.0318×10^{-3}	1.0321×10^{-3}
0.00250	4.2198×10^{-3}	6.5753×10^{-4}	5.1630×10^{-4}	5.1696×10^{-4}
0.00125	2.9634×10^{-3}	3.2726×10^{-4}	2.5902×10^{-4}	2.5689×10^{-4}
0.00063	2.0919×10^{-3}	1.6387×10^{-4}	1.2911×10^{-4}	1.2859×10^{-4}

Estimated orders (ratio between successive h):

$$\begin{aligned} \text{Euler-Maruyama:} &\quad \hat{p}_{\text{strong}} \approx 0.508,\ \hat{p}_{\text{weak}} \approx 0.999; \\ \text{Milstein:} &\quad \hat{p}_{\text{strong}} \approx 1.001,\ \hat{p}_{\text{weak}} \approx 1.002. \end{aligned}$$

Discussion:

- Euler-Maruyama converges with empirical strong order $\simeq 1/2$ and weak order $\simeq 1$, exactly as predicted by theory.
- Milstein attains order 1 in both strong and weak senses, again matching the stochastic Taylor expansion.
- The slopes were recovered from $\log_2\!\big(\text{error}(h_k)/\text{error}(h_{k+1})\big)$ and remain stable once h is sufficiently small.