Summer Project
Report

Brownian Motion

Integrated Masters in Sciences
in
Physics

Submitted by

P181211 - Gaurav Agarwal

Under the guidance of

Prof. Tridib Sadhu, TIFR Mumbai

Department of Physical Sciences,
University of Mumbai - dept. Atomic Energy
Centre for Excellence in Basic Sceinces
Mumbai

Summer 2021

Acknowledgement

I would like to thank Prof. Tridib Sadhu, TIFR exclusively for this project, as he agreed to guide me through an online mode and introduced me to his field. Every email I sent was followed by recognition and assurance which boosted my spirits. I also appreciate the help I got from my friends Lokendra Meena and Tharun Krishna for helping out in minor hiccups.

Contents

1 Brownian Motion

A particle in fluid executes Brownian motion when its mass is much larger than the mass of fluid particles. This is fundamentally modeled by the Langevin Equation. Naturally, given the system, it relates well to the diffusion process and constant given by Einstein's treatment. The parameters of both treatments are related by the fluctuation-dissipation theorem. Brownian motion is one of the simplest ways one can study non-equilibrium systems.

1.1 Formal definition

Brownian motion is mathematically defined as a continuous time, stochastic process and the standard Brownian motion is also called a Wiener process.
The Wiener process is defined by following statistical properties:

- $W_o = 0$, i.e. the motion starts at origin.
- W_t is continuous (but nowhere differentiable).
- W_t has independent increments.
- $W_t - W_s \sim N(0, t-s)$, $0 \leq s \leq t$.

1.2 Statistical properties

Let W_t donate the Standard Brownian Motion (continuous time, stochastic process).

(i) W_o=0

(ii) Brownian motion W(t) is normally distributed $W_t \sim N(0,t)$.
The increment is also normally distributed, $W_t - W_s \sim N(0, t-s)$.
Expectation values: $E[W(t)] = 0$ and $E[W(t) - W(s)] = 0$.
Variance: $Var[W(t)] = t$ and $Var[W(t) - W(s)] = t - s$.

(iii) The increments are stationary and independent.
$\therefore W_t - W_s \sim N(0, t-s) \Rightarrow W_{t+a} - W_{s+a} \sim N(0, t-s)$.
and, $W_{t_1} - W_{t_2}, W_{t_2} - W_{t_3}, \ldots W_{t_n} - W_{t_{n-1}}$ are all independent.

(iv) Covariance: $Cov\{W(t), W(s)\} = min\{s, t\}$

(v) W_t is a Markov process i.e. the future of motion only depends on the current position of the particle.
$\therefore P(W_t = k | W_{t-1}, W_{t-2} \ldots W_{t_o}) = P(W_t | W_{t-1})$

(vi) Brownian Motion is a martingale process. With every new increment, the expectation value of the motion changes to the current position.
$\therefore E[W(t+s)|F_t] = W_t$

(vii) The auto-correlation function is : $< W_t W_{t'} >= min(t, t')$

Other properties:

(viii) W_t is non-differentiable at any point (though continuous).
The measure of 'jaggedness' is simply given by:

$|W_{t+\epsilon} - W_t| \leq C|\epsilon|^\beta$, where $\beta < \frac{1}{2}$
]] $\therefore$ as $\epsilon \to 0$, $\frac{|W_{t+\epsilon} - W_t|}{|\epsilon|} \to \infty$.

$\therefore$ dW almost acts as $\sqrt{dt}$.

(ix) The above is formalized in Ito calculus, which states for a Wiener Process that the product of two time intervals is equal to the overlap of the two intervals.
$\therefore (W_{t_1} - W_{t_2}) \times (W_{t_3} - W_{t_4}) =$ overlap between $(t1, t2)$ & $(t3, t4)$
$\Rightarrow dW\ dW = dt$

(x) **Law of iterated logs:** The law states that the Wiener process falls in the ϵ range of $\pm\sqrt{2t \log\log \frac{1}{t}}$ when $\frac{1}{t} \to \infty$.

1.3 Background

1.3.1 Limit of random walk

Brownian motion can be derived as a limit of Gaussian random walks. In one dimension, let a particle move Δx left/right after Δt time. Let the random variable

$$X_i = \begin{cases} +1, & \text{if particle moves right} \\ -1, & \text{if particle moves left} \end{cases}$$

All $X_i\,'s$ are independently distributed. Let $P[X_i = 1] = p$ and $P[X_i = -1] = 1 - p$. Assuming $n = t/\Delta t$ is an integer, the particle's position with time is $Y(t) = \Delta x(X_1 + X_2 \ldots + X_n)$, and therefore,

$$E[Y(t)] = n\Delta x(2p - 1)$$
$$Var[Y(t)] = n(\Delta x)^2(1 - (2p - 1)^2)$$

Since the probability is Gaussian, and Brownian motion is a continuous process, if we make the parameters here continuous, i.e. $\Delta x = \sigma\sqrt{\Delta t}$ and $p = [1 + (\mu/\sigma)\sqrt{t}]/2$ we get:

$$E[Y(t)] = n\Delta x(2p - 1) \to \mu t$$
$$Var[Y(t)] = n(\Delta x)^2(1 - (2p - 1)^2) \to \sigma^2 t$$

as $\Delta t \to 0$.

Therefore, random walk is simply a version of discretized Brownian motion, on setting $\mu = 0$ and variance 1.

1.3.2 Einstein's treatment (brief)

Einstein put forward the formulation of a diffusion equation for Brownian particles, containing a parameter called the diffusion constant which is related to measurable physical quantities. An interesting application was to verify the size of atoms.

Considering a Brownian particle suspended in a fluid of other smaller particles we can say that th

The relation is: $D = \frac{k_B T}{6\pi\eta a}$
(ratio between the thermal energy and Stokes drag)

1.4 Langevin Model

Langevin Model is used to study Brownian motion with the help of calculus. The model is considered to have a large particle (of interest) immersed in a fluid of much smaller particles (atoms). The smaller particles constantly collide with the Brownian particle. These collisions are random and rapid due to density fluctuations within the fluid. The time scale of the smaller particles is around 10^{-12} seconds, the relaxation time τ_B of the particle velocity is around 10^{-3} and the relaxation time of the Brownian particle (time to diffuse it's own radius) τ_r is much higher in seconds.
Considering the difference in these time frames, we can easily consider the collisions to be totally instantaneous random fluctuations in our model. Let the fluctuating force be denoted by $\eta(t)$. We get

$$m\frac{dv}{dt} = \eta(t)$$

which implies that the particle can acquire infinite energy, and the system is over-driven(result can be obtained by solving the differential equation above and calculating rms velocity). Therefore, it is intuitive to add a friction force that is proportional to the velocity. This gives us:

$$m\frac{dv}{dt} = -\gamma m v + \eta(t) \tag{1}$$

In a real system, fluctuating force cannot be true, however, for modeling it is our best bet. Effects of fluctuating force can be summarized by giving its first and second moments as time averages over an infinitesimal time interval.

$$< \eta(t) >= 0$$

$$< \eta(t)\eta(t') >= \Gamma\delta(t - t')$$

This is to say that $\eta(t)$ is a delta correlated (i.e. no relations between collisions at time t and t'), Gaussian stationary white noise. Here, Γ is a constant, a measure of strength of the fluctuating force and has the dimensions of time inverse.

On solving the equation [1], we get

$$v(t) = v_o e^{-\gamma t} + \int_0^t e^{-\gamma(t-t')} \times \frac{\eta(t')}{m} dt' \tag{2}$$

1.4.1 Fluctuation - Dissipation balance

Since, equation (2) doesn't give us any great results to verify the model, we will find the rms velocity of the system. Now, equation 2 squared on both sides will have three terms on the right side:
$\boldsymbol{(v(t))^2}$:

1st term : $v_o^2 e^{-2\gamma t}$, which decays over time.

2nd term : $\int_0^t dt' \frac{\eta(t')}{m} e^{-\gamma(t-t')} \int_0^t dt" \frac{\eta(t")}{m} e^{-\gamma(t-t")}$

3rd term (cross term) : $2v_o e^{-\gamma t} \int_0^t e^{-\gamma t} \times \frac{\eta(t')}{m} dt'$

On averaging over time, the third term vanishes as $< \eta(t) >= 0$. The second term becomes:

$= \frac{1}{m^2} \int_0^t dt' e^{-\gamma(t-t')} \int_0^t dt" e^{-\gamma(t-t")} \Gamma \delta(t' - t")$

$= \frac{\Gamma}{2m^2\gamma}(1 - \exp(-2\gamma t))$

Hence, time average of square of the velocity becomes:

$$< (v(t))^2 >= v_o^2 e^{-2\gamma t} + \frac{\Gamma}{2m^2\gamma}(1 - \exp(-2\gamma t)) \tag{3}$$

In the long time limit, $t \to \infty$, the first term vanishes, and

$$< (v(t))^2 >= \frac{\Gamma}{2m^2\gamma}$$

However, a long time average is also the equilibrium ensemble average. And employing kinetic theory, we have $< v_{eq}^2 >= \frac{k_B T}{m}$.

$$\Rightarrow < (v(t))^2 >=< v_{eq}^2 >= \frac{\Gamma}{2m^2\gamma} = \frac{k_B T}{m}$$

$$\Rightarrow \boldsymbol{\Gamma = 2mk_B T\gamma} \tag{4}$$

Result (4) establishes a relation between the strength of the fluctuation force (Γ) and drag (γ). It serves as a necessary condition for the system to be neither decaying nor over-driven. Due to this it is also named as the *fluctuation-dissipation theorem.*

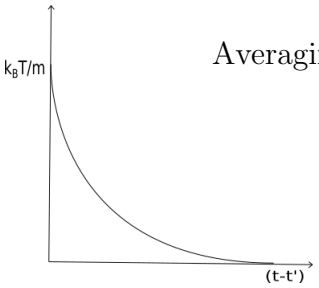
1.4.2 Velocity correlation

We had from equation (2):

$$\begin{aligned} v(t) &= v_o e^{-\gamma t} + \int_0^t e^{-\gamma t} \times \frac{\eta(t')}{m} dt' \\ \Rightarrow \overline{v(t)v(t')} &= v_0^2 e^{-2\gamma(t+t')} + \frac{\Gamma}{m^2} \int_0^t dt_1 \int_0^{t'} dt_2 e^{-\gamma(t-t_1)e^{-\gamma(t-t_2)}\delta(t_1-t_2)} \\ \Rightarrow \overline{v(t)v(t')} &= v_0^2 e^{-2\gamma(t+t')} + \frac{\Gamma}{2m^2\gamma} e^{-\gamma(t+t')}(e^{2\gamma t'} - 1) \end{aligned}$$

Substituting Γ from equation (4):

$$\begin{aligned} \overline{v(t)v(t')} &= v_0^2 e^{-2\gamma(t+t')} + \frac{k_B T}{m} e^{-\gamma(t+t')}(e^{2\gamma t'} - 1) \\ \Rightarrow \overline{v(t)v(t')} &= (v_0^2 - \frac{k_B T}{m}) e^{-\gamma(t+t')} + \frac{k_B T}{m} e^{-\gamma(t-t')} \end{aligned}$$



Averaging over long time, the velocity correlation becomes:

$$< \boldsymbol{v(t)v(t')} >= \frac{\boldsymbol{k_B T}}{\boldsymbol{m}} \boldsymbol{e^{-\gamma(t-t')}} \tag{5}$$

Here, velocity comes out to be exponentially correlated, unlike the random fluctuation $\eta(t)$.

$\eta(t) \Rightarrow$ Gaussian white noise, stationary, delta correlated, Markov process.

$$\Downarrow$$

$v(t) \Rightarrow$ Gaussian, stationary, *exponentially correlated*, stationary process.

A Gaussian, continuous, stationary, Markov process which is exponentially correlated (velocity in this case) is called a ***Ornstein–Uhlenbeck process***.

1.4.3 Mean and variance of velocity

We know from equation (2)

$$v(t) = v_o e^{-\gamma t} + \int_0^t e^{-\gamma t} \times \frac{\eta(t')}{m} dt'$$

and since $< \eta(t) > = 0$ we have:

$$\overline{\boldsymbol{v(t)}} = \boldsymbol{v_o e^{-\gamma t}} \tag{6}$$

and from equation (3) we have:

$$\overline{(v(t))^2} = \frac{k_B T}{m} + (v_o^2 - \frac{k_B T}{m}) e^{-2\gamma t}$$

Hence variance of velocity is simply:

$$\boldsymbol{Var[v(t)] = \overline{(v(t))^2} - (\overline{v(t)})^2 = \frac{k_B T}{m}(1 - e^{-2\gamma t})} \tag{7}$$

1.4.4 Mean and variance of position

Using $\dot{x(t)} = v(t)$ we can get from (2):

$$x(t) = x_o + \frac{1}{\gamma} v_o (1 - e^{-\gamma t}) + \int_0^t e^{-\gamma t'} \int_0^{t'} \frac{\eta(t'')}{m} e^{\gamma t''} dt''$$

$$\Rightarrow x(t) = x_o + \frac{1}{\gamma} v_o (1 - e^{-\gamma t}) + \frac{1}{\gamma} \int_0^t \eta(t')[1 - e^{-\gamma(t-t')}] dt' \tag{8}$$

On averaging the random fluctuations vanish, and we get:

$$\boldsymbol{\overline{x(t)} = x_0 + \frac{1}{\gamma} v_o(1 - e^{-\gamma t})} \tag{9}$$

For variance, squaring (8) we get:

$$x^2(t) = x_0^2 + 2\frac{x_o v_o}{\gamma}(1-e^{-\gamma t}) + \frac{v_o^2}{\gamma^2}(1-e^{-\gamma t})^2 + \frac{1}{\gamma^2}\int_0^t \frac{\eta(t')}{m}(1-e^{-\gamma(t-t')})dt' \int_0^t \frac{\eta(t'')}{m}(1-e^{-\gamma(t-t'')})dt''$$

$$\Rightarrow < (x^2(t) > = x_0^2 + 2\frac{x_o v_o}{\gamma}(1-e^{-\gamma t}) + \frac{v_o^2}{\gamma^2}(1-e^{-\gamma t})^2 + \frac{\Gamma}{2m^2\gamma^3}[2\gamma t - 3 + 4e^{-\gamma t} - e^{-2\gamma t}]$$

$$\boldsymbol{Var[x(t)] = \overline{x^2(t)} - \overline{x(t)}^2 = \frac{\Gamma}{2m^2\gamma^3}[2\gamma t - 3 + 4e^{-\gamma t} - e^{-2\gamma t}]} \tag{10}$$

Displacement:

We define the displacement of the Brownian particle as:

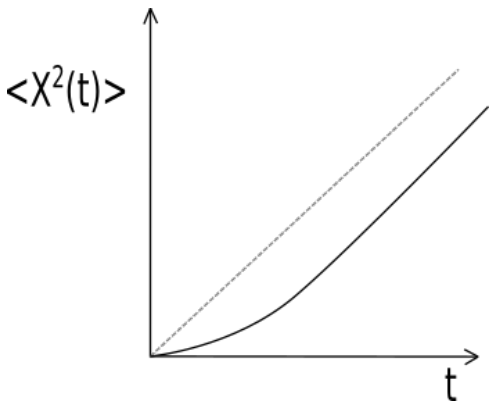
$$x(t) - x(0) = \int_0^t dt_1 v(t_1)$$

$$X(t) \equiv x(t) - x(0)$$

then,

$$< (X(t))^2 > \quad = \int_0^t dt_1 \int_0^t dt_2 < v(t_1)v(t_2) > \quad = \int_0^t dt_1 \int_0^t dt_2 \frac{k_B T}{m} e^{-\gamma|t-t'|}$$

$$\Rightarrow \quad < (X(t))^2 > \quad = \frac{2k_B T}{m\gamma^2}[\gamma t - 1 + e^{-\gamma t}]$$



Hence,

as $\gamma t \to 0$;

$$< (X(t))^2 > \to \frac{k_B T}{m} t^2 =< v^2 >_{eq} t^2$$

as $\gamma t >> 1$;

$$< (X(t))^2 > \to \frac{2k_B T}{m\gamma} t$$

Thus, initially the mean square displacement behaves very predictably, i.e. equilibrium velocity multiplied by the time, however with time, the mean square displacement becomes linear.

Also, $\overline{v(t)} = v_o e^{-\gamma t}$

$\Rightarrow \overline{X(t)} = \frac{v_0}{\gamma}(1 - e^{-\gamma t})$

1.4.5 In three dimensions

We just convert the same equation into 3 dimensions as follows:

$$\dot{\vec{v}(t)} = \gamma \vec{v(t)} + \frac{\vec{\eta(t)}}{m}$$

where, $< \eta_i(t) > = 0$ and $< \eta_i(t)\eta_j(t') > = \Gamma\delta_{ij}\delta(t-t')$, where i=(1,2,3).
(This assumes that the coordinates are completely uncorrelated).

$$\Rightarrow < v_i(t)v_j(t') > = \delta_{ij}\frac{k_BT}{m}e^{-\gamma|t-t'|}$$

If the coordinates are correlated - for example in case of a presence of a magnetic field - then the above equations change.

1.4.6 Probability density function of velocity

Considering velocity to be Gaussian at all times (except at the start where it is a dirac delta), we can substitute the mean velocity and variance of the velocity from results (6) and (7) in a normal distribution to get a conditional probability distribution (for one dimension) as follows:

$$\boldsymbol{\varrho(v,t|v_o) = \left[\frac{m}{2\pi k_B T(1-e^{-2\gamma t})}\right]^{1/2} \exp\left\{\frac{-m(v-v_o e^{-\gamma t})^2}{k_B T(1-e^{-2\gamma t})}\right\}} \tag{11}$$

1.4.7 Probability density function of position

Position again is assumed to be a Gaussian at all times, except at the initial moment where it is a dirac delta. Using results from (8) and (10) we have:

$$\begin{aligned}\boldsymbol{\varrho(x,t|x_o)} = &\left[\frac{2m^2\gamma^3}{2\pi\Gamma[2\gamma t - 3 + 4e^{-\gamma t} - e^{-2\gamma t}]}\right]^{1/2} \\ &\times \exp\left\{\frac{-m^2\gamma^3[x - x_0 + \frac{1}{\gamma}v_o(1-e^{-\gamma t})]^2}{\Gamma[2\gamma t - 3 + 4e^{-\gamma t} - e^{-2\gamma t}]}\right\}\end{aligned} \tag{12}$$

1.5 Fokker Planck Equation

A Langevin equation for a random variable ζ :

$$\dot{\zeta} = f(\zeta) + g(\zeta)\xi(t)$$

where $g(\zeta)$ is multiplicative noise and $\xi(t)$ is a delta correlated Gaussian white noise, then it has a probability distribution function $p(\zeta, t|\zeta_0, 0)$ satisfying the following equation (which is called the *Fokker Planck equation*):

$$\frac{\partial p}{\partial t} = -\frac{\partial(f(\zeta)p)}{\partial t} + \frac{1}{2}\frac{\partial^2((g(\zeta))^2 p)}{\partial \zeta^2}$$

where $p(\zeta, 0) = \delta(\zeta, \zeta_0)$.

For the Brownian motion case (taking $\sqrt{\Gamma}$ as the unit strength of noise) ,

$$\dot{v} = -\gamma v + \frac{\sqrt{\Gamma}}{m}\xi(t) \tag{13}$$

We have the following Fokker Planck equation:

$$\frac{\partial p}{\partial t} = \gamma \frac{\partial (vp)}{\partial v} + \frac{\Gamma}{2m^2}\frac{\partial^2 p}{\partial v^2}$$

and on solving we get back equation (11) exactly as in our previous assumption.
Infact, we can also get the fluctuation dissipation result from equation (4) just by taking the limit to the probability distribution at $t \to \infty$.
(the left part becomes 0, as $\frac{\partial p_{eq}}{\partial t} = 0$ and we are just left with a second order differential equation in one variable.)

1.5.1 High friction limit

Starting from the equation,

$$\dot{v} = -\gamma v + \frac{\sqrt{\Gamma}}{m}\xi(t)$$

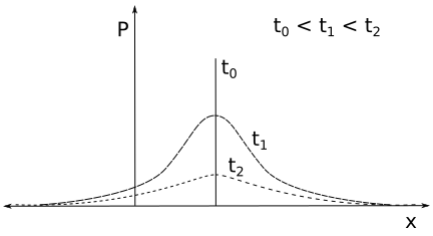
In high friction limit, $\gamma t >> 1$, therefore, we can ignore $\dot{v}$ term:

$$\gamma v = \frac{\sqrt{\Gamma}}{m}\xi(t)$$

$$\Rightarrow \dot{x} = \frac{\sqrt{\Gamma}}{m\gamma}\xi(t)$$

$$\Rightarrow \dot{x} = \sqrt{2D}\xi(t)$$

On comparing with the generalized Langevin Equation, $f(\zeta) = 0$ and $g(\zeta) = \sqrt{2D}$. The corresponding Fokker Planck equation is:



$$\frac{\partial p(x,t)}{\partial t} = D\frac{\partial^2 p}{\partial x^2}$$

which is the standard diffusion equation giving us:

$$p(x,t) = \frac{1}{\sqrt{4\pi D t}} e^{-\frac{(x-x_0)^2}{4Dt}}$$

(We can add a drift term here by adding a driving force/potential.)

2 Simulations of Brownian Motion

2.1 Random walk

As Brownian motion can be thought of as an extended limit of a random walk (in continuous time and space), the following is a simple simulation of a random walk in discretized time and continuous space in 2 dimensions.
We can simply choose a starting point and draw increments in position independently from both x and y direction from a normal distribution.

`The code is at:  5.1` Github

The code gives us the above graph of the random walk, initiated at $x, y = 0$ at $t = 0$ seconds and progressing in both x an y axis.

When we let the random walk evolve for a longer time, such that each unit time becomes less and less apparent, the motion as a whole assumes the properties of Brownian motion.

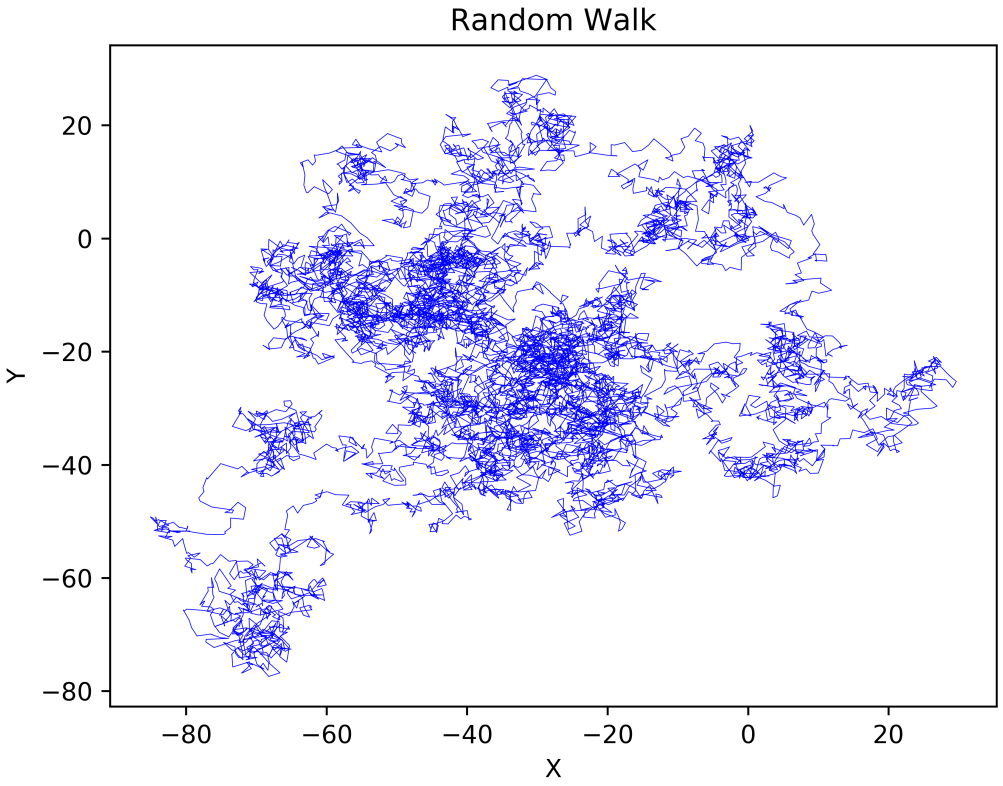


Figure 1: Random Walk (discrete time and continuous space)

2.2 Standard Brownian Motion

The standard Brownian motion or the Wiener process can be simply simulated with adding random normal noise with mean 0 and variance dt to the system.

We follow the equation (with strength of noise = 1):

$$\dot{x} = \eta(t)$$

$$\Rightarrow dx = dW(t)$$

where dW is the Wiener process. On simulating this we get the following results: `The code is at: 5.2` Github

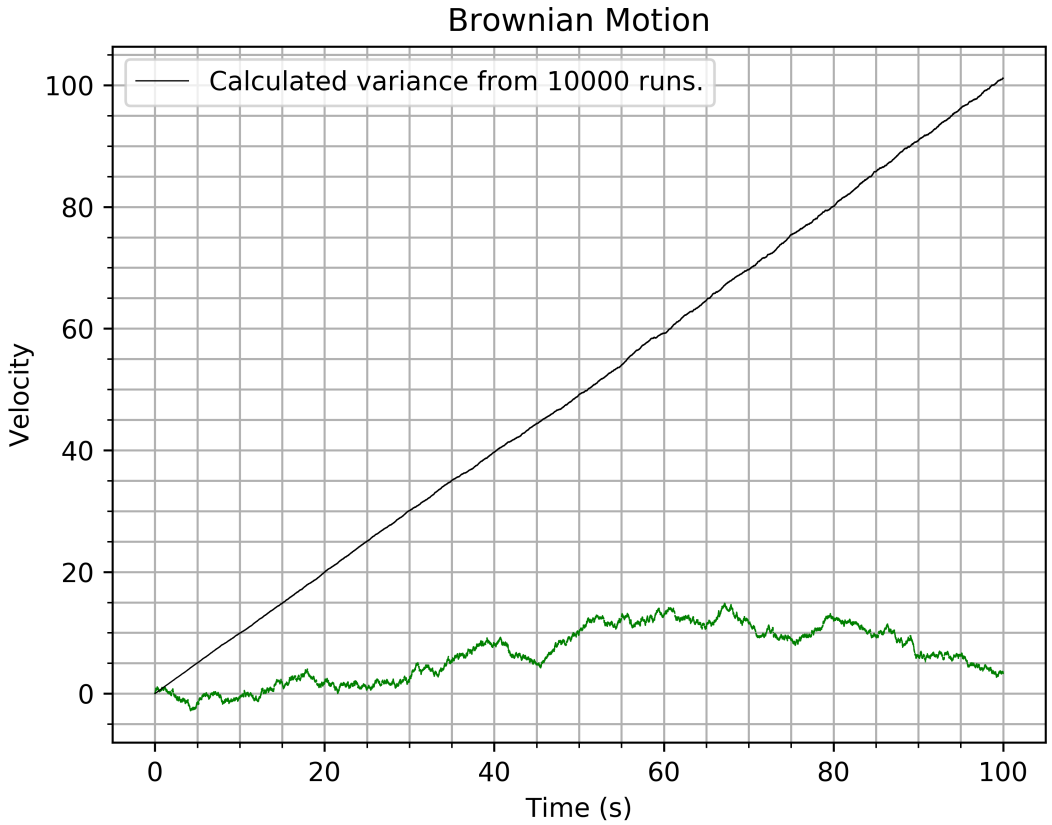


Figure 2: In the figure, variance calculated from 10^4 runs is plotted. One of the samples is also shown.

In the above image, variance σ^2 clearly goes as 't'.

2.2.1 Khinchin's Law of Iterated Logarithm

The law of iterated log describes the magnitude of the fluctuations of a random walk. Khinchin generalised it to the sums of independent and identically distributed random variables with zero mean and bounded increments.
The law states that the lim sup of absolute of the sum of n random variables approaches $\sqrt{2n\log(\log n))}$ i.e. if X_n is a random variable, then for $B_n = X_1 + X_2 \ldots + X_n$ we have:

$$\limsup_{n\to\infty} \frac{\pm S_n}{\sqrt{2n\log(\log n))}} = 1 \qquad (a.s.)$$

This also implies that the standard Brownian motion should not cross the boundary set by the iterated log as $t \to 0$.
We can demonstrate this using code. On simulating multiple instances of the BM for long times, and keeping track of the instances that cross the boundary, we see a decrease in the number of motions that have position higher than the Khinchin boundary as time progresses significantly.
`Click here to go to code : 5.2.1` Github

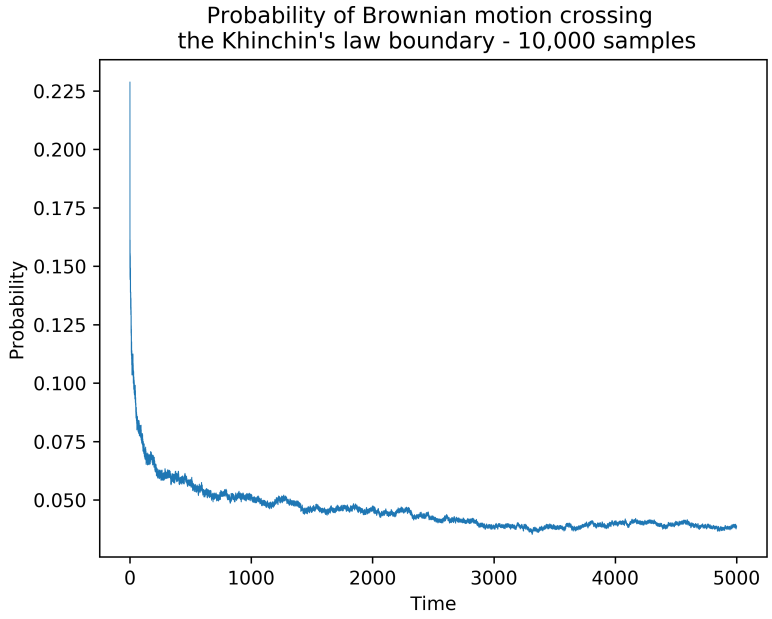


Figure 3: The probability of a BM having position greater than that of Khinchin's boundary decreases with time approaching zero. The above plot simulated 10,000 BM(s) till 5000 seconds with a time-step of 10^{-3} seconds.

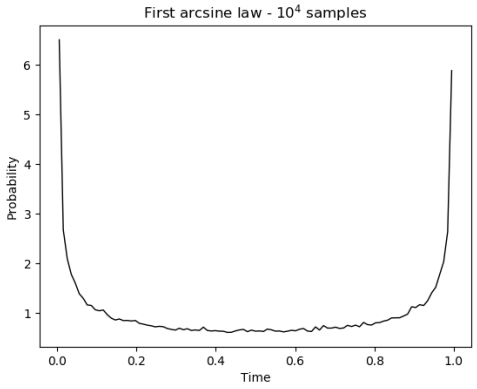
2.2.2 Arc-Sine Laws

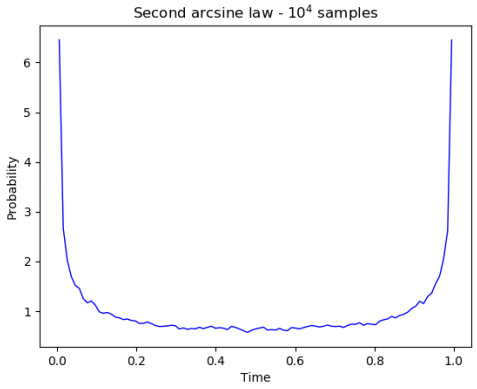
Arc sine laws relate the path properties of the Wiener process to the arc-sine distribution.
If we assume that $(W_t)_{0\leq t\leq 1}$ is the one dimensional Wiener process then the following laws hold:

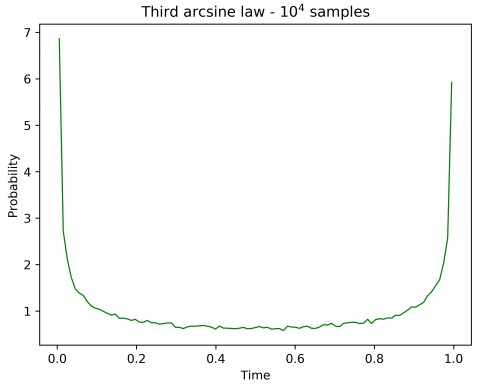
- **First Arcsine law:**
  The proportion of time that the Wiener process is positive follows an arcsine distribution.
  $$T_+ = \{t \in [0,1] : W_t > 0\}$$
- **Second Arcsine law:**
  The last time that the Wiener process changes sign follows an arcsine distribution.
  $$L = \sup\{t \in [0,1] : W_t = 0\}$$
- **Third Arcsine law:**
  The time at which the Wiener process reaches its maximum follows an arcsine distribution.
  $$W_{max} = \sup\{W_s : s \in [0,1]\}$$

`Click here to go to code : 5.2.1` Github

We use the above code, and find the following results:





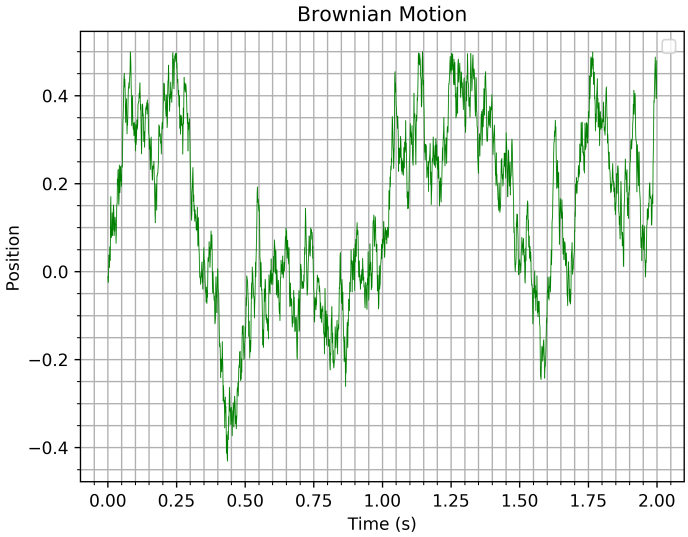


These distributions were obtained for the corresponding parameters of positive time, last time of sign change, and time of maximum position for the first, second and third arcsine laws respectively. Distribution was calculated over 10^4 sample BM runs.

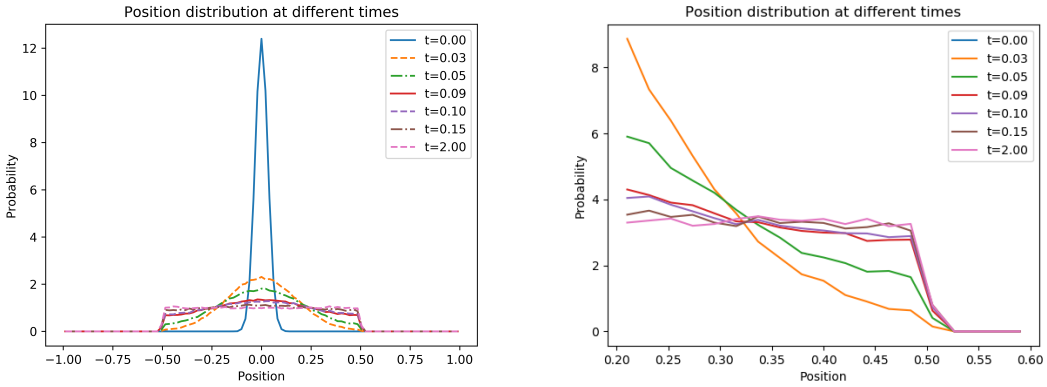
2.2.3 S.B.M. with reflecting walls

On implementation of reflecting walls with standard BM, we restrict the particle in only a continuous set of allowed positions. This can be done by folding the space over at the walls, such that the Brownian particle moves undisturbed.
`Click here to go to code : 5.2.2` Github

We obtain the following results:



This represents the BM trapped inside walls at -0.5 and 0.5.
The BM is simply folded over after if reaches any of the walls.



(a) (b)

As soon as the Brownian particles diffuse and touch the boundary, the probability distribution flattens out and is not Gaussian anymore. The flattening is shown with probability distribution at different times in (a). (b) shows the flattening magnified at boundaries.

2.3 Euler-Mayurama method

Euler Mayurama method is a method to approximate the numerical solution of a stochastic differential equation.
Considering the equation:

$$dX_t = a(X_t, t)dt + b(X_t, t)dW_t$$

given an initial condition on X, and where W_t stands for a Wiener process, then the Euler-Mayurama approximation of X is the Markov chain as follows:

- Divide the total time into ordered intervals of small times Δt.
- Recursively define $X_{n+1} = X_n + a(X_n, t_n)\Delta t + b(X_n, t_n)\Delta W_n$ where ΔW is a random normal I.I.D variable with zero mean and variance Δt.

2.3.1 Applied to Langevin equation without drift

The Langevin equation (Ornstein-Uhlenbeck solution) we had in equation (13) is;

$$\dot{v} = -\gamma v + \frac{\sqrt{\Gamma}}{m}\xi(t)$$

$$\Rightarrow dv = -\gamma v dt + \frac{\sqrt{\Gamma}}{m}\xi(t)dt$$

where $\xi(t)dt = dW_t$ (a Weiner process). Hence,

$$\Rightarrow dv = -\gamma v dt + \frac{\sqrt{\Gamma}}{m}dW_t$$

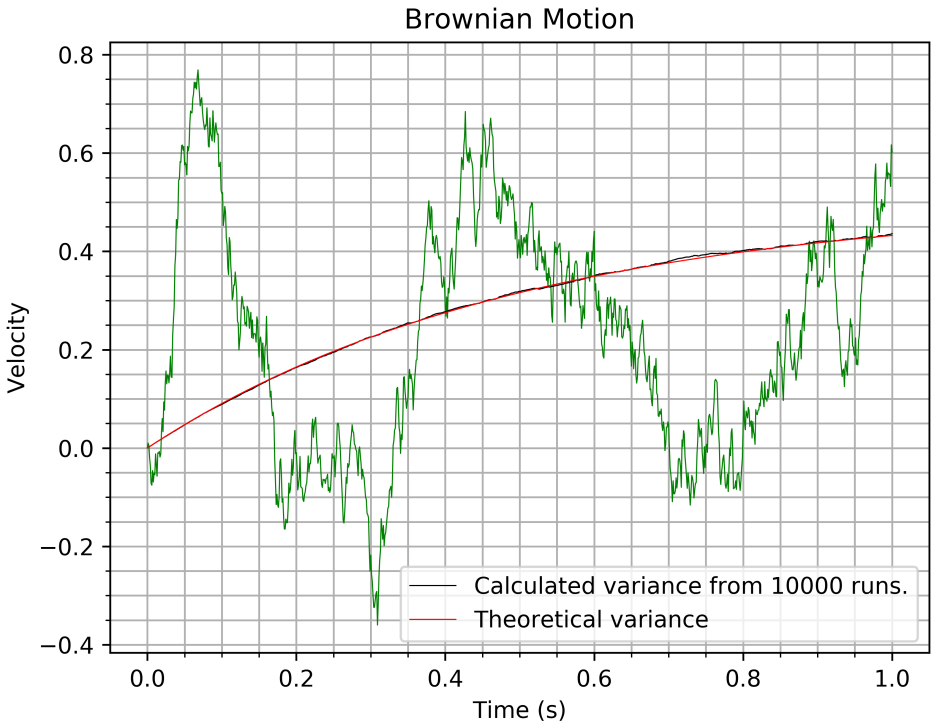
We can simulate the above using the following python code:

`Click here to go to code : 5.3.1` Github

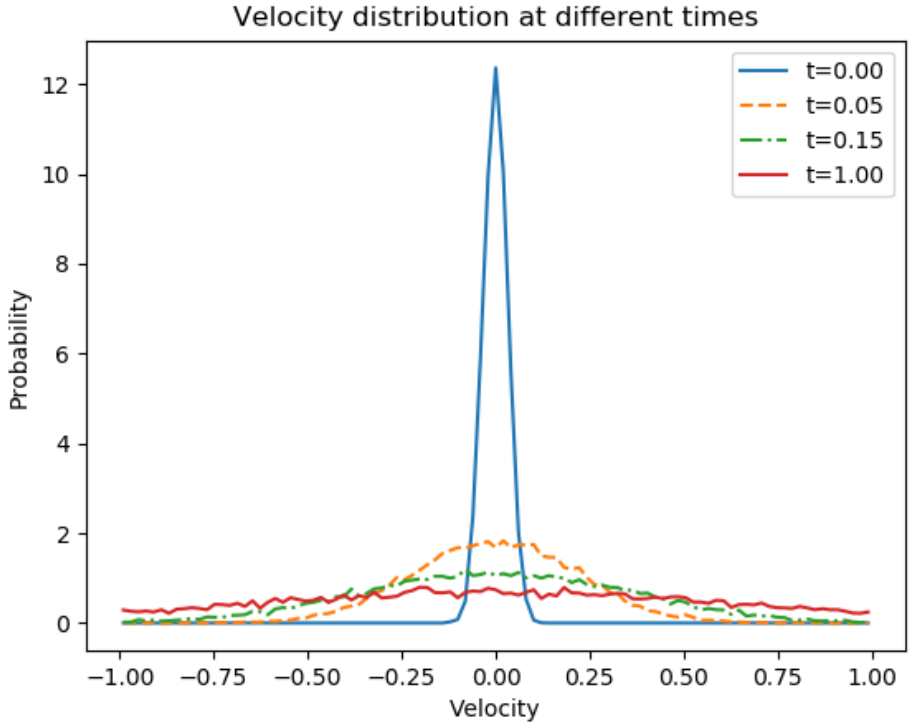
As apparent in the code, we have set our physical constants (Γ, γ, m) all as 1, and set initial condition as $v_o = 0$, and evolve 10^4 Brownian motions till 1 second in the intervals of 10^{-3} seconds to get a good result in the statistics.

It's easy to see in figure 6, the model is evolving correctly, with the variance being almost a perfect match from the analytical calculations in equation (7) ($Var[v(t)] == \frac{k_B T}{m}(1 - e^{-2\gamma t})$).

On evaluating the probability density of the velocity distribution at different times we get the results in figure 6 which come out to be exactly as expected i.e. gaussian at all times.



(a) One of the runs of Langevin Equation (green) and associated variance



(b) Probability distribution from 10^4 runs at different times

Figure 6: Results from simulation of Langevin Equation with Euler Mayurama method without drift term.

2.3.2 Applied to Langevin equation with drift

We can directly use the results from above, with a minor difference;

$$a(X_t, t) = -\gamma(v - \mu)$$

where $\mu = \frac{k_B T}{2}$ is the average velocity. Since $\Gamma = 2k_B T m\gamma$ is already defined, we use $\Gamma = 4\mu\gamma$ and have the following equation:

$$dv = -\gamma(v - \mu)dt + \frac{\sqrt{\Gamma}}{m} dW_t$$

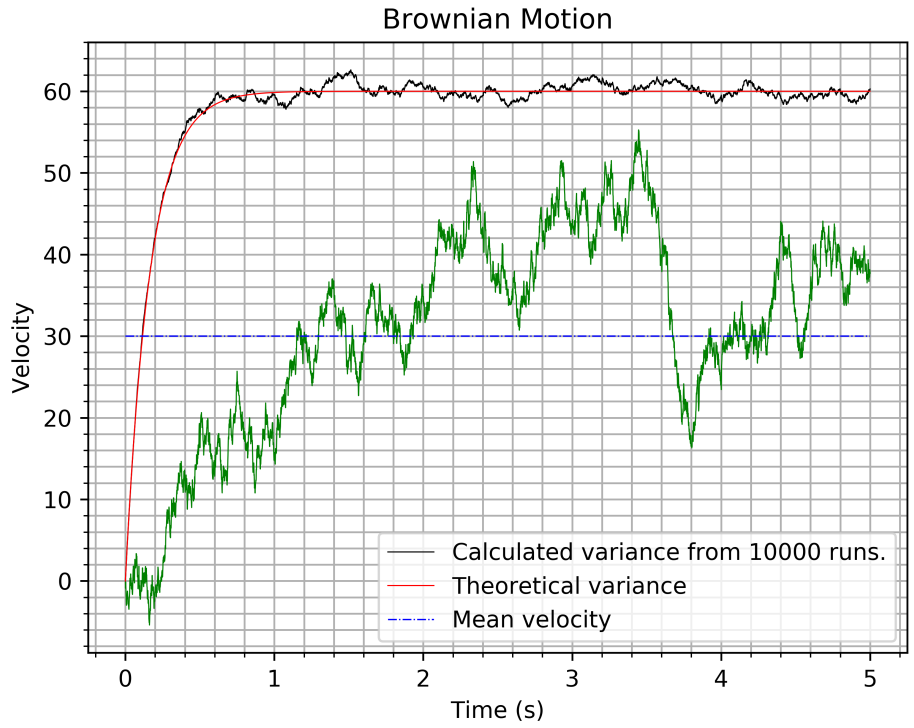
We can simulate the above using the following python code:

`Click here to go to code : 5.3.2` Github

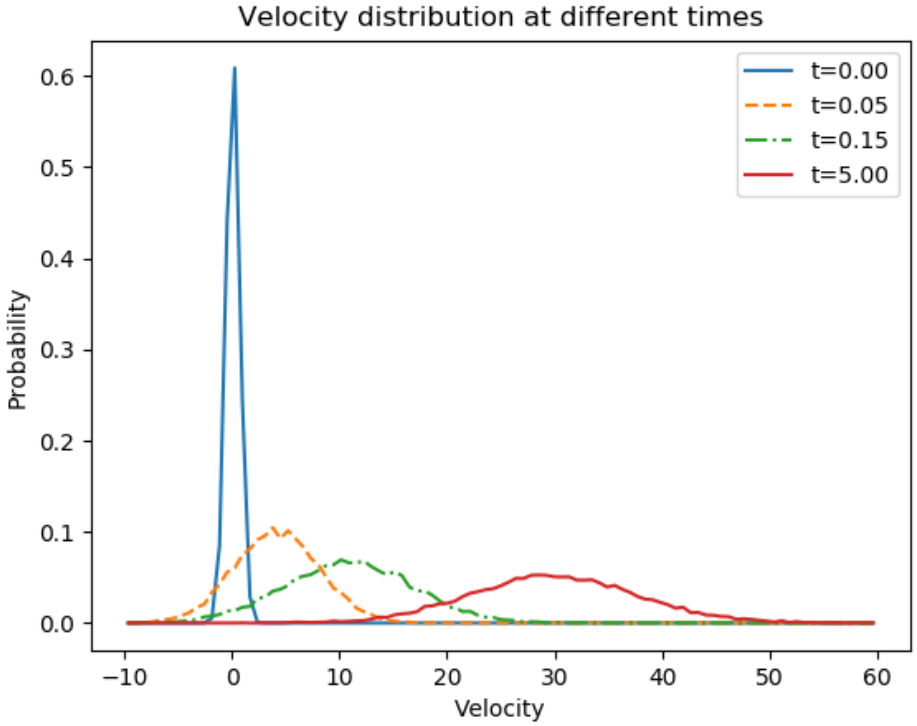
In the code, physical constants are $(\mu = 30, \gamma = 3, m = 1)$ and initial condition as $v_o = 0$. We evolved 10^4 Brownian motions till 5 seconds in the intervals of 10^{-3} seconds.
We get the following results :

It's easy to see in figure 7, the model is evolving correctly, the motion assumes values near the average velocity, with the variance being almost a perfect match from the analytical calculations in equation
(7) ($Var[v(t)] == \frac{k_B T}{m}(1 - e^{-2\gamma t})$).

On evaluating the probability density of the velocity distribution at different times we get the results in figure 7 which come out to be exactly as expected i.e. gaussian at all times and gradually shifting towards the mean of $v = 30$.



(a) One of the runs of Langevin Equation (green) and associated variance



(b) Probability distribution from 10^4 runs at different times (t in seconds)

Figure 7: Results from simulation of Langevin Equation with Euler Mayurama method with the drift term.

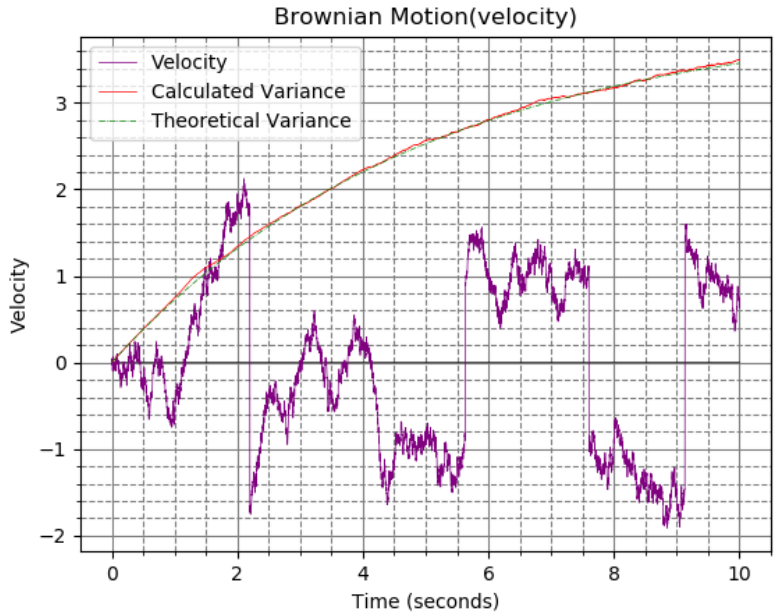
2.3.3 With reflecting walls

To implement walls we have to keep track of the position, and simply reversing the velocity if particle crosses the wall. The code used above can be modified to do so.

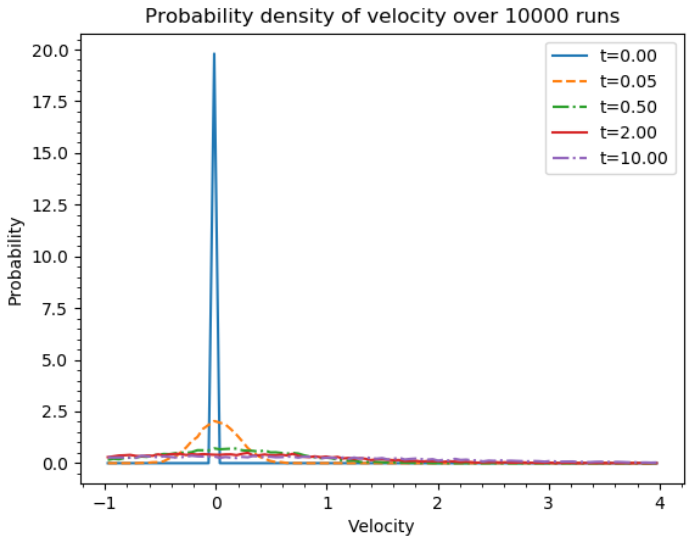
`Click here to go to code : 5.3.3` Github

The logic is to construct a boolean array containing the check if the particle's position is outside of the walls. If so, the boolean array is multiplied element-wise by the velocity array (having the information of velocities for all the number of trials at a particular time). This gives us an array with only the velocities that need to be reversed by multiplying -1. Rest of the velocities are simply evolved normally obeying the Langevin equation. Then the reversed velocities are added to the next iteration of velocities, to combine all the reversed and evolved velocities into one array.With Numba and Numpy, the code is fast enough to do 500 seconds of evolution with a 0.001 second time step for 10,000 particles in about 190 seconds.

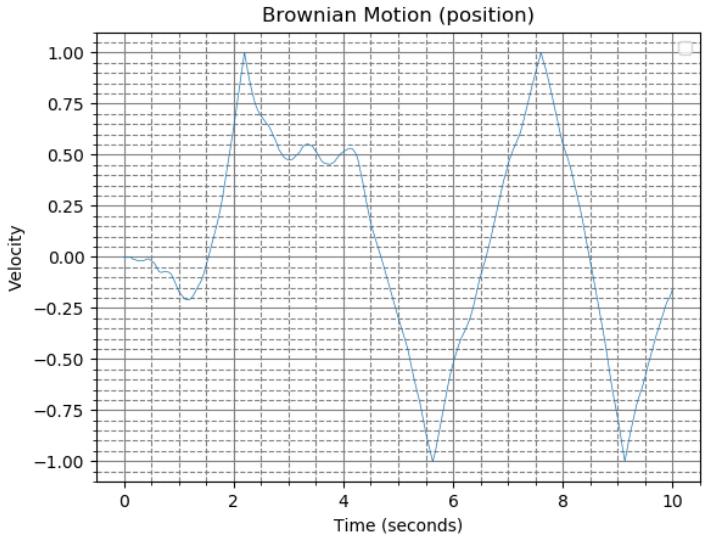
Setting the physical constants, $\gamma = 0.01, \mu = 2$ and $m = 1$, setting the walls to be at $x = \pm 1$ and evolving 10,000 particles for 10 seconds we have the following results:



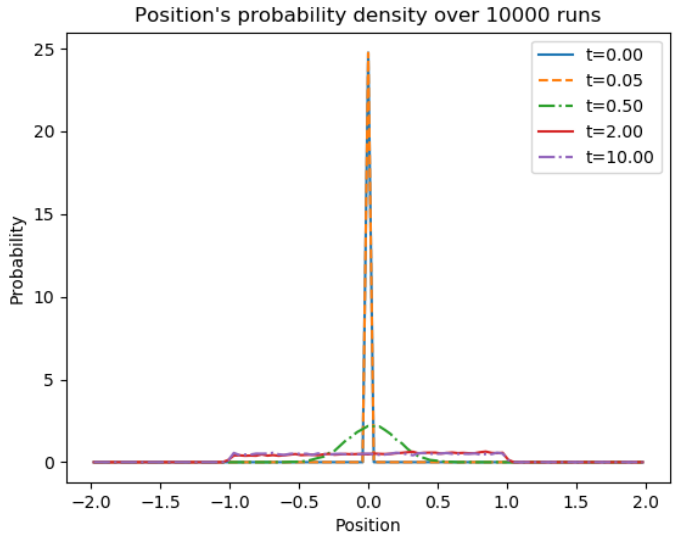
Purple line depicts the Brownian motion of one of the particles. The jumps seen in the velocity is when the particle hits a wall. As seen, the variance (calculated over 10,000 runs) matches perfectly with the theoretical value.



The graph depicts the distribution of velocity over ten seconds of run.

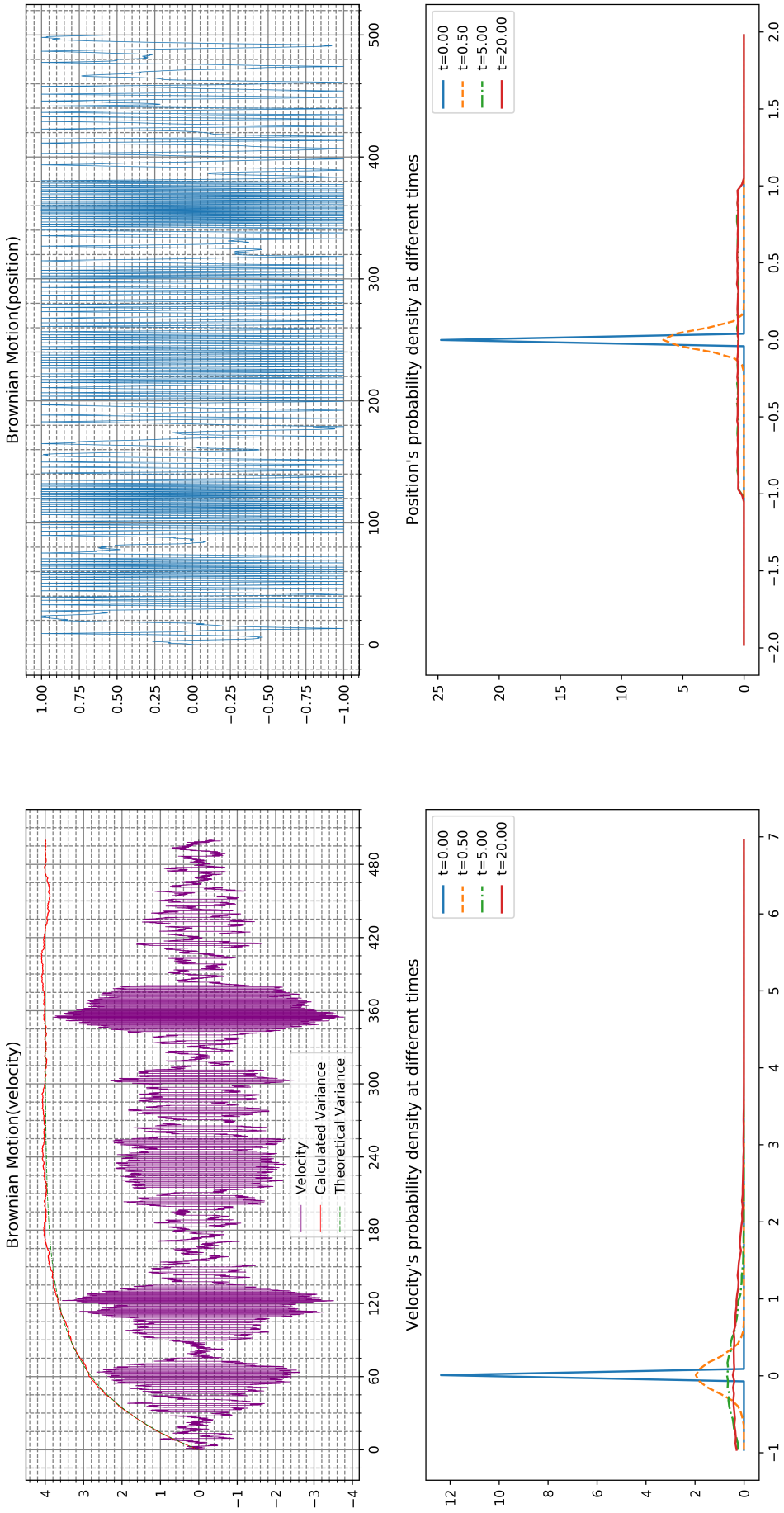


Blue line represents the position of particle with the velocity in **??**. The reflections can be seen clearly.



The probability density of the position of the particle trapped between the walls at x=-1 and x=1.

There is not much change if the simulation is run for 500 seconds:



3 Fractional Brownian Motion

A fractional Brownian motion (fBm) is a centered Gaussian process $\{B_t^H, t \geq 0\}$ defined by the following covariance function:

$$E[B_t^H B_s^H] = \frac{1}{2}(t^{2H} + s^{2H} - |t-s|^{2H}) \tag{14}$$

where $H \in (0,1)$ is a parameter called the Hurst index.

Since specifying a distribution of a Gaussian process only needs defining its mean and covariance, therefore, for a particular H, we have a uniquely determined B^H.
The covariance function defined above has to be non-negative definite if fBm exists. Also, for $H = 1/2$, the covariance :

$$E[B_t^{1/2} B_s^{1/2}] = \frac{1}{2}(t + s - |t-s|) = \min(s,t)$$

which defines the standard Brownian Motion. Therefore, *fBm serves as a generalization of the standard Brownian motion.*

The increments of fBm are defined as $B_t^H - B_s^H$ for $t > s$. The covariance functions between increments comes out to be:

$$E[(B_{t_1}^H - B_{s_1}^H)(B_{t_2}^H - B_{s_2}^H)] = E[B_{t_1}^H B_{t_2}^H + B_{s_1}^H B_{s_2}^H - B_{s_1}^H B_{t_2}^H - B_{s_2}^H B_{t_1}^H]$$

using (14) we have:

$$E[(B_{t_1}^H - B_{s_1}^H)(B_{t_2}^H - B_{s_2}^H)] = \frac{1}{2}(|t_1 - s_2|^{2H} + |t_2 - s_1|^{2H} - |t_1 - t_2|^{2H} - |s_1 - s_2|^{2H}) \tag{15}$$

This gives us the variance of increments as ($t_1 = t_2$ & $s_1 = s_2$):

$$\sigma^2 = E[(B_{t_1}^H - B_{s_1}^H)^2] = |t_1 - s_1|^{2H}$$

which again corresponds to standard Brownian motion at $H = 1/2$.

Properties:

- B_t^H has stationary increments. (evident from the variance above)
- $B_0^H = 0$, $E[B_t^H] = 0$ for all $t \geq 0$.
- $E[(B_t^H)^2] = t^{2H}$ for $t \geq 0$.

- B_t^H has a Gaussian distribution for $t > 0$.
- fBm is neither Markov nor Martingale (unlike standard B.M.).

Dependence of increments: From equation (15) we can have the covariance negative or positive.
For $H \in (0, 1/2)$; we have $E[(B_{t_1}^H - B_{s_1}^H)(B_{t_2}^H - B_{s_2}^H)] > 0$.
This essentially means that if the fBm was increasing in the past, the trend is likely to continue.
However, for $H \in (1/2, 1)$; we have $E[(B_{t_1}^H - B_{s_1}^H)(B_{t_2}^H - B_{s_2}^H)] < 0$.
This essentially means that the fBm is likely to break it's current trends in the future. As a result, the motion looks more jagged.
Hence, higher the H, smoother the fBm becomes. For H=1, fBm simply becomes a linear function of the Gaussian noise.

4 Simulation of fBm

Fractional Brownian motion can be simulated in discrete time. Since it is the cumulative sum of Fractional Gaussian noise, our aim should be to calculate 'fGn'.
Notation:
$Y_0, Y_1 \dots Y_n$ represents the fBm.
$X_0, X_1 \dots X_n$ represent the samples of fGn.

We can get samples of fGn using three well known exact methods : Hosking's, Cholesky's and Davies & Harte's method. Below are the first two:

4.1 Hosking's Method

This method is used to generate X_n given $X_{n-1} \dots X_o$ and can be used ot generate any stationary Gaussian process (in this case fGn).
Let $\gamma(k) \coloneqq E[X_n X_{n+k}]$ be the covariance function corresponding to the equation (15).
Then the $(n+1) \times (n+1)$ covariance matrix is:

$$\Gamma(n) = \begin{pmatrix} \gamma(0) & \gamma(-1) & \gamma(-2) & \dots & \gamma(-n) \\ \gamma(1) & \gamma(0) & \gamma(-1) & \dots & \gamma(-n+1) \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \gamma(n) & \gamma(n-1) & \gamma(n-2) & \dots & \gamma(0) \end{pmatrix}_{(n+1)\times(n+1)} = (\gamma(i-j))_{i,j=0,1\dots n}$$

Assume a $(n+1)$ column vector :

$$c(n) = \begin{pmatrix} \gamma(1) \\ \gamma(2) \\ \vdots \\ \gamma(n+1) \end{pmatrix} = (\gamma(k+1))_{k=0,\dots n}$$

Then the matrix $\Gamma(n+1)$ can be written in block form as follows:

$$\Gamma(n+1) = \begin{pmatrix} 1 & c(n)' \\ c(n) & \Gamma(n) \end{pmatrix}$$

Then, the inverse of $\Gamma(n+1)$ can be calculated as follows:

$$\Gamma(n+1)^{-1} = \frac{1}{|\Gamma(n+1)|} \begin{pmatrix} \Gamma(n) & -c(n) \\ -c(n) & 1 \end{pmatrix}$$

$$\Rightarrow \Gamma(n+1)^{-1} = \frac{1}{\Gamma(n)\{1 - c(n)'\Gamma(n)^{-1}c(n)\}} \begin{pmatrix} \Gamma(n) & -c(n) \\ -c(n) & 1 \end{pmatrix}$$

$$\Rightarrow \Gamma(n+1)^{-1} = \frac{1}{\sigma_n^2}\begin{pmatrix} 1 & -d(n)' \\ -d(n) & \sigma_n^2\Gamma(n)^{-1} + d(n)d(n)' \end{pmatrix}$$

where $\sigma_n^2 = 1 - c(n)'\Gamma(n)^{-1}c(n)$ and $d(n) = \Gamma(n)^{-1}c(n)$.

Using the above equation, we can get the following:

$$\begin{pmatrix} X_{n+1} & x \end{pmatrix}\Gamma(n+1)^{-1}\begin{pmatrix} X_{n+1} \\ x' \end{pmatrix} = \frac{(X_{n+1} - d(n)'x)^2}{\sigma_n^2} + x'\Gamma(n)^{-1}x$$

where $x = \begin{pmatrix} X_n & X_{(n-1)} & \dots & X_0 \end{pmatrix}$.

The above equation clearly is in the form of the Central Limit Theorem, and implies that X_{n+1} is a random variable that is Gaussian distributed with mean μ_n and variance σ_n^2 as follows:

$$\mu_n = d(n)'x = c(n)'\Gamma(n)^{-1}x$$
$$\sigma_n^2 = 1 - c(n)'\Gamma(n)^{-1}c(n)$$

The above can be done recursively to get the distribution parameters for the next random variable. To do this, one needs to calculate $d(n+1)$ (for mu) and σ_{n+1}^2 recursively. With a bit more algebra, one obtains the following recursive relations:

$$\sigma_{n+1}^2 = \sigma_n^2 - \frac{(\gamma(n+2) - \tau_n)^2}{\sigma_n^2}$$
$$d(n+1) = \begin{pmatrix} d(n) - \phi_n F(n)d(n) \\ \phi_n \end{pmatrix}$$

Where,

$$\phi_n = \frac{\gamma(n+2) - \tau_n}{\sigma_n^2}$$
$$\tau_n = c(n)'F(n)d(n)$$
$$F(n) = \begin{pmatrix} 0 & 0 & \dots & 1 \\ 0 & \dots & 1 & 0 \\ \vdots & & \vdots & \vdots \\ 1 & 0 & \dots & 0 \end{pmatrix}_{(n+1)\times(n+1)}$$

For the simulation, we have to assume a initial sample X_0 to start off. From this we can get the following:

$$\mu_0 = \gamma(1)X_0$$

$$\sigma_0^2 = 1 - \gamma(1)^2$$

$$d(0) = c(0) = \gamma(1)$$

and we can sample the Gaussian random variable X_1 from this, and then get μ_1, σ_1^2 and so on.

On simulating we obtain the following results:

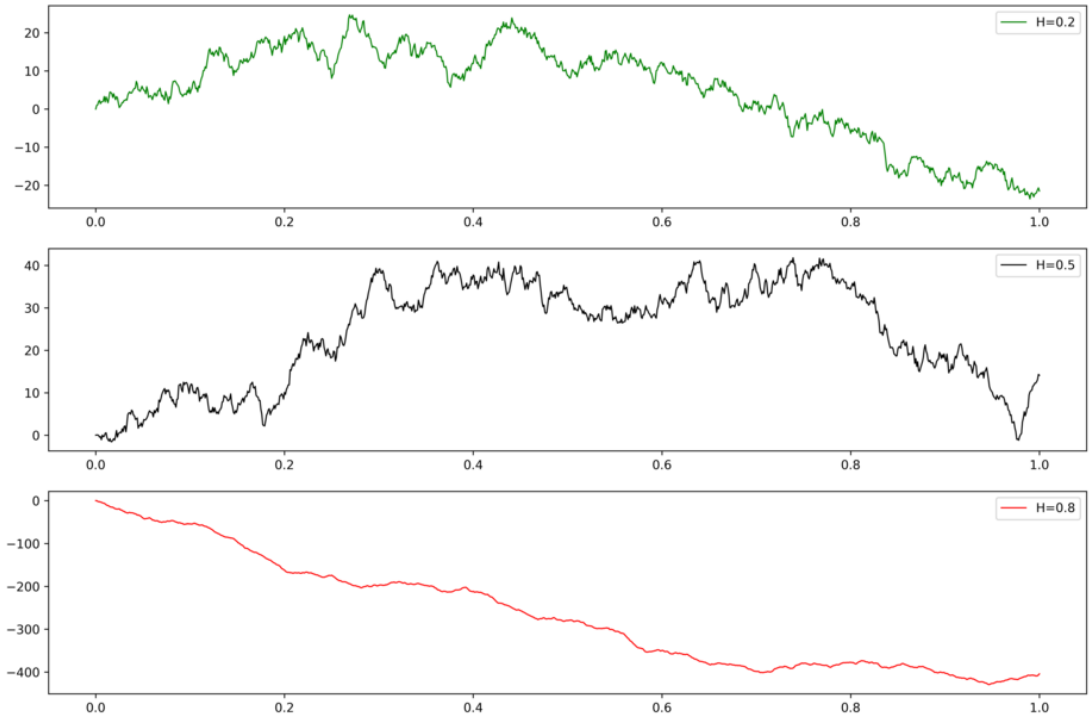


Figure 11: fBm simulated from Hosking's recursion method for different Hurst values (0.2,0.5,0.8).

It is easy to see that the motion for $H < 0.5$ is jagged, whereas $H > 0.5$ follows it's previous trend and is positively correlated.

`Click here to go to code : 5.4` Github

4.2 Cholesky Method

Extending from the Hosking's covariance matrix itself, we can decompose the Γ matrix in the form of $L(n)L(n)'$ such that $L(n)$ is a lower triangluar matrix. This is called the Cholesky decomposition, and exists whenever the matrix is symmetric and positive definite (which Γ is).

Once the Γ matrix is broken down and $L(n)$ is found out, we can easily find X_{n+1}:

$$X_{n+1} = \sum_{k=0}^{n+1} = l_{n+1,k} V_k$$

`Click here to go to code : 5.4` Github

The only problem with this method is that it is slower than Hosking method. We obtain the following results:

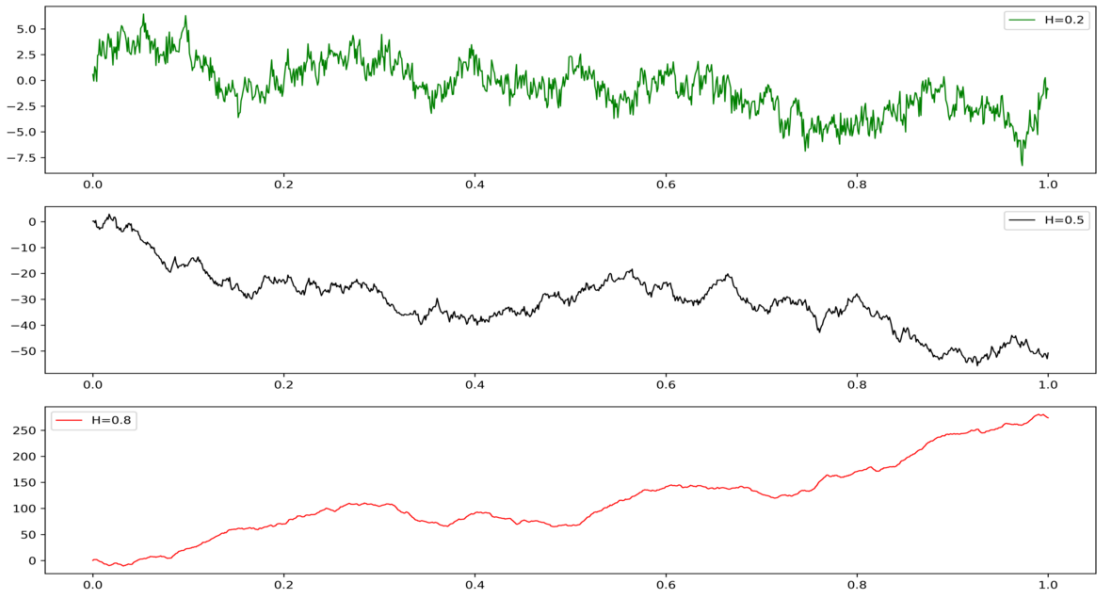


Figure 12: fBm simulated from Cholesky's decomposition method for different Hurst values (0.2,0.5,0.8).

5 Code

5.1 Random Walk

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


samples=int(1) # number of motions to simulate
time_limit=int(10000) # number of steps each brownian motion is evolved for
variance=1 #the variance of the normal distribution from which the increments are taken
           # since the increments ~ N(0,t-s); here t-s is 1 unit
initial_x, initial_y = [0],[0]


SAMPLEX = np.empty((samples,time_limit+1)) #store the position of the particle
SAMPLEY = np.empty((samples,time_limit+1))

for i in range(0,samples):

    x_inc = np.random.normal(0,variance,time_limit)  #get increments from a normal distribution
    y_inc = np.random.normal(0,variance,time_limit)

    SAMPLEX[i] = np.append(initial_x, np.cumsum(x_inc)) #set starting point to zero, then append to it
    SAMPLEY[i] = np.append(initial_y, np.cumsum(y_inc)) #final position of particle at each point

    plt.plot(SAMPLEX[i],SAMPLEY[i], c='blue', linewidth=0.3)

plt.title('Random Walk')
plt.xlabel("X")
plt.ylabel("Y")
plt.savefig('randomwalk.png',dpi=600)
```

5.2 Standard Brownian Motion

```
import numpy as np
import matplotlib.pyplot as plt
from numba import jit
num_sims = 10000  # Number of runs to evaluate

t_init = 0
t_end  = 0.2
```

```
dt      = 0.0001
N       = int((t_end - t_init)/dt)  #this many grid points will be calculated
y_init = 0



ts = np.arange(t_init, t_end+dt, dt) #timestep array
ys = np.zeros(N + 1)
print(np.shape(ts),np.shape(ys))
ys[0] = y_init
variance = np.zeros(N+1)
allmotions = np.zeros((num_sims,N+1))

#numba is used to compile code into machine language, as python interpreter is slow
@jit(nopython=True,fastmath=True)
def loop(num_sims,ts,t_init, dt,ys,allmotions):
    for j in range(num_sims):
        for i in range(1, ts.size):
            t = t_init + (i - 1) * dt
            y = ys[i - 1]
            ys[i] = y + np.random.normal(loc=0.0, scale=np.sqrt(dt))
        allmotions[j] = ys
    return ys, allmotions

ys, allmotions = loop(num_sims,ts,t_init,dt,ys,allmotions)

#plot one of the motions:
plt.plot(ts, ys, lw=0.5, c='green')

#calculation of variance using all the runs
for j in range(1,ts.size):
    variance[j] = np.var(allmotions[:,j])

#--- below variance is plotted ----- #
maxt= int(0.04243/dt)
print(maxt)
iteratedlog = np.sqrt( 2*np.multiply(ts[1:maxt],np.log(np.log(np.reciprocal(ts[1:maxt])))) )
iteratedlog2 = np.sqrt(np.multiply( 2*ts , np.log(np.log(ts)) ))

plt.plot(ts,variance, lw=0.5, c='black', label=f'Calculated variance from {num_sims} runs.')
plt.plot(ts[1:maxt], iteratedlog, 'b--', lw=0.7, label =r'$\pm \sqrt{2n\log{\log{\frac{1}{t}}}}$')
plt.plot(ts[1:maxt], -iteratedlog, 'b--', lw=0.7)
#plt.plot(ts, iteratedlog2, 'b--', lw=0.7)
```

```
51
52 plt.legend()
53 plt.title("Brownian Motion")
54 plt.xlabel("Time (s)")
55 plt.ylabel("Velocity", rotation='vertical')
56 plt.minorticks_on()
57 plt.grid(which='both')
58 plt.savefig('stdbrown1',dpi=600)
59
60 #---- plot probability density below ---- #
61 plt.clf()
62 prdens_time = (1, 50, 150, N )
63 styledict = ('-','--','-.','-')
64 bins = np.linspace(-1, 1, 100)
65
66 for j in prdens_time: #plotting prob density for velocity at times
       in prdens_time
67     hist, _ = np.histogram(np.reshape(allmotions[:,j],-1), bins=
      bins, density =True)
68     plt.plot((bins[1:] + bins[:-1]) / 2, hist,dict(zip(prdens_time
      ,styledict))[j],label=f"t={j * dt:.2f}")
69
70 plt.legend()
71 plt.title("Velocity distribution at different times")
72 plt.xlabel("Velocity")
73 plt.ylabel("Probability", rotation='vertical')
74 plt.savefig('stdbrown2')
```

5.2.1 Khinchin's Law

```
1 import numpy as np
2 import matplotlib.pyplot as plt
3 from numba import jit
4 import time
5 import gc
6
7 num_sims = 10000  # Number of runs to evaluate
8 length = (10**4)    #number of time iterations to process in one
      chunk
9 t_init = 0
10 t_end  =1000
11 dt     = 0.001
12 N      = int((t_end - t_init)/dt)  #this many grid points will be
      calculated
13 y_init = 0
14 print(f"{N} iterations for {num_sims} trials will being processed.
      ")
15 print(f"{int(N/length)} chunks will be processed. Each chunk has {
      length} timesteps.")
```

```
ts = np.arange(t_init, t_end, dt) #timestep array

if (N%length != 0):
        raise Exception("Length not compatible. Change either N or length such that N is divisible by length.")
def mainfunction(dt,N,num_sims,y_init,length):
    if (N%length != 0):
        raise Exception("Length not compatible. Change either N or length such that N is divisible by length.")
    tempprob, tempposmaxvalue = np.zeros(length),np.zeros((length,num_sims))
    np.savetxt('test.txt', [])
    probability=np.array([])
    y = y_init
    with open('test.txt', 'a') as posmaxvaluefile:
        for i in range(int(N/length)):
            tempprob, tempposmaxvalue,y = loop(dt,length,num_sims,y,i)
            np.savetxt(posmaxvaluefile, tempposmaxvalue, delimiter=",")
            probability = np.hstack((probability,tempprob))
            del(tempprob,tempposmaxvalue)
            gc.collect()
            print(f"{(i+1)*100/(int(N/length))}% done.")
    return probability


@jit(nopython=True ,fastmath=True, parallel=True)
def loop(dt,N,num_sims,y_init,i):
    posmaxvalue = np.zeros((N,num_sims))
    probability = np.zeros(N)
    y=np.zeros(num_sims) + y_init
    Dt = np.sqrt(dt)
    k=2.719+i*N*dt
    for j in range(N):
        y += np.random.normal(0,Dt,num_sims)
        temp = np.absolute(posmaxvalue[j-1,:])>np.absolute(y)
        posmaxvalue[j,:] = np.absolute(np.multiply(~temp,y) + np.multiply(temp,posmaxvalue[j-1,:]))
        k=k+dt
        probability[j] = np.sum(np.absolute(y)>np.sqrt(2*k*np.log(np.log(k))))/num_sims

    return probability,posmaxvalue,y

time0 = time.time()
probability = mainfunction(dt,N,num_sims,y_init,length)
print("Simulation done in: ",time.time()-time0, ". Proceeding with plotting.")
```

```

#----------- PLOT KHINCHIN's PROB w/ TIME -------#
plt.xlabel("Time")
plt.ylabel("Probability", rotation='vertical')
plt.title(f"Probability of Brownian motion crossing \n the
    Khinchin's law boundary - {num_sims} samples", name='CMU Sans
    Serif')
plt.plot(ts[:N],probability,lw=0.5)
plt.savefig('stdbrownitlog', dpi=600)


#--------- PLOT LIM SUP OF POSITION ------------#

cols=50
plt.clf()
gc.collect()
for i in range(int(num_sims/cols)):
    j=[i*cols+k for k in range(cols)]
    temp = np.loadtxt('test.txt', usecols=j, delimiter=',')
    for k in range(cols):
        plt.plot(ts[:N], temp[:N,k], lw=0.5)
    del(temp)
    gc.collect()
    print(f"{(i+1)*cols} / {num_sims} done.", '\r')
iteratedlog2 = np.sqrt(np.multiply( 2*(ts+2.719) , np.log(np.log(
    ts+2.719)) ))
plt.plot(ts, iteratedlog2, 'k--', lw=0.9, label =r'$\pm \sqrt{2n\
    log{\log{t}}}$')
plt.xlabel("Time (sec)")
plt.ylabel("Position")
plt.title("Brownian Motion - 200 samples \n lim sup of absolute
    position")
plt.savefig('iteratedlogruns', dpi=600)
gc.collect()
```

```
import numpy as np
import matplotlib.pyplot as plt
from numba import jit
num_sims = 100000  # Number of runs to evaluate

t_init = 0
t_end  =1.0001
dt     = 0.0001
N      = int((t_end - t_init)/dt)  #this many grid points will be
    calculated
y_init = 0
prdens_time = (1, 50, 150, N )

```

```
13
14 ts = np.arange(t_init, t_end, dt) #timestep array
15 #@jit(nopython=True ,fastmath=True)
16 def loop(dt,N,num_sims,y_init):
17     pos = np.zeros(num_sims) +y_init
18     Dt=np.sqrt(dt)
19    # positivepos=np.zeros(num_sims)
20     positivetime= np.zeros(num_sims)
21     maxpostime = np.zeros(num_sims)
22     maxpostn=np.zeros(num_sims)
23     signchangetime = np.zeros(num_sims)
24     lastiter = np.zeros(num_sims).astype(bool)
25
26     for j in range(N):
27         pos += np.random.normal(0,Dt,num_sims)
28         thisiter = pos>0
29        # temp = (lastiter-thisiter).astype(bool)
30         temp = np.logical_xor(lastiter,thisiter)
31         signchangetime = np.multiply(signchangetime,~temp) + temp*
    j
32         lastiter = thisiter
33         positivetime += thisiter
34
35         temp = maxpostn > pos
36         maxpostime = np.multiply(maxpostime,temp) + ~temp*j
37         maxpostn = np.multiply(maxpostn,temp) + np.multiply(pos,~
    temp)
38
39     return positivetime,maxpostime,signchangetime
40
41 positivetime,maxpostime,signchangetime = loop(dt,N,num_sims,y_init
    )
42 positivetime = positivetime*dt
43 maxpostime = maxpostime*dt
44 signchangetime = signchangetime*dt
45
46 plt.title(r"First arcsine law - $10^4$ samples")
47 plt.xlabel("Time")
48 plt.ylabel('Probability')
49 bins = np.linspace(0,1,100)
50 hist,_ = np.histogram(positivetime, bins = bins, density=True )
51 plt.plot((bins[1:] + bins[:-1]) / 2, hist, 'k-', lw=1)
52 plt.savefig('arcsinepositivetime')
53
54
55 plt.clf()
56 plt.title(r"Second arcsine law - $10^4$ samples")
57 plt.xlabel("Time")
58 plt.ylabel('Probability')
```

```
bins = np.linspace(0,1,100)
hist,_ = np.histogram(maxpostime, bins = bins, density=True )
plt.plot((bins[1:] + bins[:-1]) / 2, hist, 'r-', lw=1)
plt.savefig('arcsinemaxpostime')

plt.clf()
plt.title(r"Third arcsine law - $10^4$ samples")
plt.xlabel("Time")
plt.ylabel('Probability')
bins = np.linspace(0,1,100)
hist,_ = np.histogram(signchangetime, bins = bins, density=True )
plt.plot((bins[1:] + bins[:-1]) / 2, hist, 'g-', lw=1)
plt.savefig('arcsinesignchangetime', dpi=600)
```

5.2.2 with reflecting walls

```
import numpy as np
import matplotlib.pyplot as plt
from numba import jit
import time

time0=time.time()

num_sims = 100000  # Number of runs to evaluate

t_init = 0
t_end  =2
dt     = 0.001
wall1 , wall2 = 0.5,-0.5
N      = int((t_end - t_init)/dt)  #this many grid points will be
    calculated
y_init = 0
prdens_time = (0, int(0.03/dt), int(0.05/dt),int(0.09/dt),int(0.1/
    dt), int(0.15/dt), N-1)
styledict = ('-','--','-.','-','--','-.', '--' )

ts = np.arange(t_init, t_end, dt) #timestep array
probdenspos = np.zeros((np.size(prdens_time),num_sims))
@jit(nopython=True ,fastmath=True)
def loop(dt,N,num_sims,y_init,prdens_time,probdenspos):
    posonerun = np.zeros(N) + y_init
    variance = np.zeros(N)

    y=np.zeros(num_sims)
    Dt = np.sqrt(dt)
    i=0
    for j in range(N):
        y += np.random.normal(0,Dt,num_sims)
        temp = (y<wall2)
```

```
        temp2 = np.multiply((2*wall2-np.multiply(y,temp)),temp)
        y = np.multiply(~temp,y) + temp2
        temp = (y>wall1)
        temp2 = np.multiply((2*wall1 - np.multiply(y,temp)),temp)
        y = np.multiply(~temp,y) + temp2
        variance[j] = np.var(y)
        posonerun[j] = y[0]
        if j in prdens_time:
            probdenspos[i,:] = y
            i+=1

    return posonerun,variance,probdenspos

posonerun,variance,probdenspos = loop(dt,N,num_sims,y_init,
    prdens_time,probdenspos)


# #plot one of the motions:
print(np.shape(ts),np.shape(posonerun))
plt.plot(ts[0:N], posonerun,'g-' ,lw=0.5)


# #--- below variance is plotted ----- #

iteratedlog2 = np.sqrt(np.multiply( 2*ts , np.log(np.log(ts)) ))
# maxt = int(40/dt)
# plt.plot(ts[:maxt],variance[:maxt], lw=0.5, c='black', label=f'
    Calculated variance from {num_sims} runs.')
plt.plot(ts, iteratedlog2, 'b--', lw=0.7, label =r'$\pm \sqrt{2t\
    log{\log{t}}}$')
# plt.plot(ts, -iteratedlog2, 'b--', lw=0.7)

plt.legend()
plt.title("Brownian Motion")
plt.xlabel("Time (s)")
plt.ylabel("Position", rotation='vertical')
plt.minorticks_on()
plt.grid(which='both')
plt.savefig('teststd1',dpi=600)

#---- plot probability density below ---- #
plt.clf()

bins = np.linspace(-1, 1, 100)

i=0
for j in prdens_time: #plotting prob density for velocity at times
    in prdens_time
    hist, _ = np.histogram(np.reshape(probdenspos[i,:],-1), bins=
```

```
    bins, density =True)
77     plt.plot((bins[1:] + bins[:-1]) / 2, hist,dict(zip(prdens_time
    ,styledict))[j],label=f"t={j * dt:.2f}")
78     i+=1
79
80 plt.legend()
81 plt.title("Position distribution at different times")
82 plt.xlabel("Position")
83 plt.ylabel("Probability", rotation='vertical')
84 plt.savefig('teststd2',dpi=600)
85
86 plt.clf()
87
88 bins = np.linspace(0.2, 0.6, 20)
89
90 i=0
91 for j in prdens_time: #plotting prob density for velocity at times
     in prdens_time
92     hist, _ = np.histogram(np.reshape(probdenspos[i,:],-1), bins=
    bins, density =True)
93     plt.plot((bins[1:] + bins[:-1]) / 2, hist,label=f"t={j * dt:.2
    f}")
94     i+=1
95
96 plt.legend()
97 plt.title("Position distribution at different times")
98 plt.xlabel("Position")
99 plt.ylabel("Probability", rotation='vertical')
100 plt.savefig('teststd3')
101
102 print(f"done in {time.time()-time0}")
```

5.3 Euler-Mayurama

5.3.1 without drift

```
1 import numpy as np
2 import matplotlib.pyplot as plt
3 from numba import jit
4 num_sims = 10000  # Number of runs to evaluate
5
6 t_init = 0
7 t_end  = 1
8 dt     = 0.001
9 N      = int((t_end - t_init)/dt)  #this many grid points will be
    calculated
10 y_init = 0
11
12 #--- All physical constants we need for the simulation ---#
```

```
mass = 1
gamma = 1
Gamma = 1


ts = np.arange(t_init, t_end + dt, dt) #timestep array
ys = np.zeros(N + 1)
ys[0] = y_init
variance = np.zeros(N+1)
allmotions = np.zeros((num_sims,N+1))


#numba is used to compile code into machine language, as python interpreter is slow
@jit(nopython=True,fastmath=True)
def loop(num_sims,ts,t_init, dt,ys,allmotions, gamma, Gamma, mass):
    for j in range(num_sims):
        for i in range(1, ts.size):
            t = t_init + (i - 1) * dt
            y = ys[i - 1]
            ys[i] = y + -gamma*(y) * dt + np.sqrt(Gamma)/mass * np.random.normal(loc=0.0, scale=np.sqrt(dt))
        allmotions[j] = ys
    return ys, allmotions

ys, allmotions = loop(num_sims,ts,t_init,dt,ys,allmotions, gamma, Gamma, mass)

#plot one of the motions:
plt.plot(ts, ys, lw=0.5, c='green')

#calculation of variance using all the runs
for j in range(1,ts.size):
    variance[j] = np.var(allmotions[:,j])

#--- below variance is plotted ----- #
plt.plot(ts,variance, lw=0.5, c='black', label=f'Calculated variance from {num_sims} runs.')
plt.plot(ts,Gamma/(2*mass*mass*gamma) * (1-np.exp(-2*gamma*ts)), lw=0.5, c="red", label='Theoretical variance')

plt.legend()
plt.title("Brownian Motion")
plt.xlabel("Time (s)")
plt.ylabel("Velocity", rotation='vertical')
plt.minorticks_on()
plt.grid(which='both')
plt.savefig('images/stbrownmot.png',dpi=600)
```

```
56
57 #---- plot probability density below ---- #
58 plt.clf()
59 prdens_time = (1, 50, 150, N )
60 styledict = ('-','--','-.',':')
61 bins = np.linspace(-1, 1, 100)
62
63 for j in prdens_time: #plotting prob density for velocity at times
       in prdens_time
64     hist, _ = np.histogram(np.reshape(allmotions[:,j],-1), bins=
      bins, density =True)
65     plt.plot((bins[1:] + bins[:-1]) / 2, hist,dict(zip(prdens_time
      ,styledict))[j],label=f"t={j * dt:.2f}")
66
67 plt.legend()
68 plt.title("Velocity distribution at different times")
69 plt.xlabel("Velocity")
70 plt.ylabel("Probability", rotation='vertical')
71 plt.savefig('images/stdbrown-probdist')
```

5.3.2 with drift

```
1 import numpy as np
2 import matplotlib.pyplot as plt
3 from numba import jit
4 num_sims = 10000  # Number of runs to evaluate
5
6 t_init = 0
7 t_end  = 5
8 dt     = 0.001
9 N      = int((t_end - t_init)/dt)  #this many grid points will be
      calculated
10 y_init = 0
11
12 #--- All physical constants we need for the simulation ---#
13 mass = 1
14 gamma = 3
15 mu = 30
16 Gamma = 4*mu* gamma
17
18
19 ts = np.arange(t_init, t_end + dt, dt) #timestep array
20 ys = np.zeros(N + 1)
21 ys[0] = y_init
22 variance = np.zeros(N+1)
23 allmotions = np.zeros((num_sims,N+1))
24
25
26 #numba is used to compile code into machine language, as python
```

```
    interpreter is slow
@jit(nopython=True,fastmath=True)
def loop(num_sims,ts,t_init, dt,ys,allmotions, gamma, Gamma, mass):
    for j in range(num_sims):
        for i in range(1, ts.size):
            t = t_init + (i - 1) * dt
            y = ys[i - 1]
            ys[i] = y  -gamma*(y-mu) * dt + np.sqrt(Gamma)/mass * np.random.normal(loc=0.0, scale=np.sqrt(dt))
        allmotions[j] = ys
    return ys, allmotions

ys, allmotions = loop(num_sims,ts,t_init,dt,ys,allmotions, gamma, Gamma, mass)

#plot one of the motions:
plt.plot(ts, ys, lw=0.5, c='green')

#calculation of variance using all the runs
for j in range(1,ts.size):
    variance[j] = np.var(allmotions[:,j])

#--- below variance is plotted ----- #
plt.plot(ts,variance, lw=0.5, c='black', label=f'Calculated variance from {num_sims} runs.')
plt.plot(ts,Gamma/(2*mass*mass*gamma) * (1-np.exp(-2*gamma*ts)), lw=0.5, c="red", label='Theoretical variance')
print(np.size(ts))
plt.plot(ts,np.zeros(np.size(ts))+mu, 'b-.', lw=0.6,label='Mean velocity')

plt.legend()
plt.title("Brownian Motion")
plt.xlabel("Time (s)")
plt.ylabel("Velocity", rotation='vertical')
plt.minorticks_on()
plt.grid(which='both')
plt.savefig('stbrownmotwdrift.png',dpi=600)

#---- plot probability density below ---- #
plt.clf()
prdens_time = (1, 50, 150, N )
styledict = ('-','--','-.','-')
bins = np.linspace(-10, 60, 100)

for j in prdens_time: #plotting prob density for velocity at times in prdens_time
    hist, _ = np.histogram(np.reshape(allmotions[:,j],-1), bins=
```

```
    bins, density =True)
68     plt.plot((bins[1:] + bins[:-1]) / 2, hist,dict(zip(prdens_time
    ,styledict))[j],label=f"t={j * dt:.2f}")
69
70 plt.legend()
71 plt.title("Velocity distribution at different times")
72 plt.xlabel("Velocity")
73 plt.ylabel("Probability", rotation='vertical')
74 plt.savefig('stdbrown-probdistwdrift')
```

5.3.3 with walls

```
1  import numpy as np
2  import matplotlib.pyplot as plt
3  import time
4  from numba import jit
5
6  ## -----All physical variables -------#
7  mass = 1
8  gamma = 0.1
9  mu = 2                        # Mean velocity (affects drift)
10 #(don't put 0 here)
11 Gamma = 2*((mu*mass)**2) * gamma
12 ##---- All Simulation variables -----#
13 init_pos = 0
14 init_vel = 0
15 wall2, wall1 = -1,1            # Wall coordinates in x
16 dt = .001                      # Time step.
17 T = 10                        # Total time in seconds.
18 ntrials=10000                 # number of brownian motions to do
19
20 #-----------------------------------#
21 n = int(T / dt)                 # Number of time steps.
22 time0 = time.time()
23 print("number of time steps = ",n)
24 print("number of trials = ", ntrials)
25 #---------Initialisation-------------#
26
27 vel = np.zeros((1,ntrials))+init_vel
28 postn = init_pos+np.zeros((1,ntrials))
29 prdens_time = (0, int(0.005/(gamma*dt)), int(0.05/(gamma*dt)), int
    (0.2/(gamma*dt)), int(T/dt)-1 )
30 #the times at which probability density will be plotted
31
32 probdensvel = np.zeros((np.size(prdens_time),ntrials))
33 probdenspos = np.zeros((np.size(prdens_time),ntrials))
34
35 """
36 #LOOP EXPLAINED:
```

```
# tempwall returns the boolean array indicating if particle has reached out of wall
# temp holds the array of new negative velocities at the true locations indicated by tempwall, rest are zero
# new vel gets updated by normal laws, but only the ones which are within limits, i.e. false in tempwall i.e. ~tempwall
# then just add temp to new vel to reverse the required velocities
# position is updated and saved onto file
"""
@jit(nopython=True,fastmath=True)
def loop(vel,postn,probdensvel,probdenspos,gamma,Gamma,mass,n,mu,dt):
    i=0
    gammaloop = gamma*dt
    sigmaloop = np.sqrt(Gamma)/mass
    variancevel = np.zeros(n)
    variancepos = np.zeros(n)
    velonerun = np.zeros(n)
    posonerun = np.zeros(n)
    Dt=np.sqrt(dt)
    for j in range(n):
        if j in prdens_time:
            probdensvel[i,:] = (vel[0,:])
            probdenspos[i,:] = (postn[0,:])
            i+=1
        tempwall = (postn[0,:] > wall1) + (postn[0,:] < wall2)
        temp = np.multiply(vel[0,:],-1*(tempwall))
        vel[0,:] = np.multiply((vel[0,:] -(vel[0,:]-mu)*gammaloop+sigmaloop*np.random.normal(0,Dt,size=(1,ntrials))),~tempwall)
        vel[0,:] += temp
        postn[0,:] = postn[0,:]+ vel[0,:]*dt
        variancevel[j] = np.var(np.reshape(vel,-1))
        variancepos[j] = np.var(postn[0,:])
        velonerun[j] = vel[0,0]
        posonerun[j] = postn[0,0]


    return velonerun,posonerun,variancevel,variancepos,probdensvel,probdenspos

velonerun,posonerun,variancevel,variancepos,probdensvel,probdenspos = loop(vel,postn,probdensvel,probdenspos,gamma,Gamma,mass,n,mu,dt)
time0=time.time()-time0
print("Calculations done in ", time0, "seconds. Now starting plotting...")
# ---------Simulation Ended-------------------------#
time0=time.time()

```

```
# ----------- block related to plotting ----------- #
t = np.linspace(0., T, n)       # array of time for plotting x axis
fig, axs = plt.subplots(2,2 , figsize=(20,10))

axs[0,0].set_title('Brownian Motion(velocity)')
axs[0,0].grid(b=True, which='major', color='grey', linestyle='-')
axs[0,0].grid(b=True, which='minor', color='grey', linestyle='--')
axs[0,0].minorticks_on()
axs[0,1].set_title('Brownian Motion(position)')
axs[0,1].grid(b=True, which='major', color='grey', linestyle='-')
axs[0,1].grid(b=True, which='minor', color='grey', linestyle='--')
axs[0,1].minorticks_on()
axs[1,0].set_title('Velocity\'s probability density at different
    times')
axs[1,1].set_title('Position\'s probability density at different
    times')
styledict = ('-','--','-.','-','-.')
axs[0,0]
#-------------------------------------------------#

#----- plotting position and velocity wrt time ------#

axs[0,0].plot(t, np.zeros(n), c='black', lw=0.5) #refrence line
axs[0,1].plot(t,posonerun, lw=0.5)
axs[0,0].plot(t,velonerun, lw=0.5, c='purple', label=f"Velocity")
axs[0,0].plot(t,variancevel[:n], c='red', lw=0.5, label=f"
    Calculated Variance")
axs[0,0].plot(t,(mu**2)*(1-np.exp(-2*gamma*t)),'-.',c='g',lw=0.5,
    label=f"Theoretical Variance")
axs[0,0].legend()
axs[0,0].xaxis.set_major_locator(plt.MaxNLocator(10))
axs[0,0].yaxis.set_major_locator(plt.MaxNLocator(10))
axs[0,0].set_xlabel("t (seconds)")
axs[0,0].set_ylabel("Velocity", rotation=90)


#---- plotting prob density of velocity and position-----#
bins = np.linspace(-1, 4, 100)
i=0
for j in prdens_time: #plotting prob density for velocity at times
     in prdens_time
     hist, _ = np.histogram(np.reshape(probdensvel[i,:],-1), bins=
    bins, density =True)
     axs[1,0].plot((bins[1:] + bins[:-1]) / 2, hist,dict(zip(
    prdens_time,styledict))[j],label=f"t={j * dt:.2f}")
     axs[1,0].legend()
     i +=1

bins = np.linspace(-2, 2, 100)
```

```
i=0
for j in prdens_time: #only for these times create histogram
    hist, _ = np.histogram(np.reshape(probdenspos[i,:],-1), bins=
    bins, density =True)
    axs[1,1].plot((bins[1:] + bins[:-1]) / 2, hist,dict(zip(
    prdens_time,styledict))[j],label=f"t={j * dt:.2f}")
    axs[1,1].legend()
    i+=1
plt.title(f'Brownian Motion for {T} seconds.')
plt.savefig('test.png',dpi=600)
print("Time spent in plotting = ", time.time()-time0, "seconds.")
```

5.4 fBm

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

def autcovar(x,H,C=1):
    """
    Defines the covariance of fBm.
    Input -> x : index , H : hurst parameter, C : scale (default
    =1).
    Ouput -> covariance
    """
    autocovariance = C**2 /2 * ( (abs(x-1))**(2*H) + (abs(x+1))
    **(2*H) -2*(abs(x))**(2*H) )
    return autocovariance

def fGn_hosking(N,H,init_pos=0):
    """
    Generates fractional gaussian noise using hosking's recursion
    method.
    Input -> H: hurst index, N : number of samples to generate,
    init_pos : initial sample of noise.
    Output -> (N+1)x1 array of fGn
    """
    # Initialisation of all parameters
    mu = autcovar(1,H) * init_pos
    var = 1-autcovar(1,H) **2
  #  var = 1
    d = np.zeros(N)
    d[0] = autcovar(1,H)
    x = np.zeros(N+1)
    x[0] = init_pos
    x[1] = np.random.normal(mu,np.sqrt(var))
    c = [autcovar(i,H) for i in range(1,N)]

    for i in range(2,N+1):
        tau = np.dot(c[0:i-2],np.flip(d[0:i-2]))
```

```
        phi = (autcovar(i,H) - tau)/var
        var = (var - phi**2 * var)
        d[0:i-2] = d[0:i-2] - phi* np.flip(d[0:i-2])
        d[0:i-1] = phi
        mu = np.dot(d[0:i-1],np.flip(x[0:i-1]))
        x[i] = np.random.normal(mu,np.sqrt(var))

    return x

def fGn_cholesky(N,H):
    """
    Generates fractional gaussian noise using cholesky's
    decomposition method.
    Input -> H: hurst index, N : number of samples to generate,
    init_pos : initial sample of noise.
    Output -> (N+1)x1 array of fGn
    """
    Gamma = np.zeros((N,N))
   # Gamma = [ [autcovar(j-i,H) for i in range(j+1)] for j in
    range(N)]
    for i in range(N):
        for j in range(i+1): Gamma[i,j] = autcovar(i-j,H)
    cov = np.linalg.cholesky(Gamma)
    x = np.dot(cov, np.array(np.random.normal(0,1,N)).transpose())
    x = np.squeeze(x)
    return x


dt=0.001
N = int(1/dt)

fig,axs = plt.subplots(3, figsize=(15,10))


y=np.cumsum(fGn_hosking(N,0.2))
t = np.linspace(0,1,y.size)
axs[0].plot(t,y,label='H=0.2',lw=1,c='green')
axs[0].legend()

y=np.cumsum(fGn_hosking(N,0.5))
t = np.linspace(0,1,y.size)
axs[1].plot(t,y,label='H=0.5',lw=1,c='black')
axs[1].legend()

y=np.cumsum(fGn_hosking(N,0.8))
t = np.linspace(0,1,y.size)
axs[2].plot(t,y,label='H=0.8',lw=1,c='red')
axs[2].legend()

```

```
78 plt.savefig('fbmhosking.png',dpi=600)
79
80 plt.clf()
81
82 fig,axs = plt.subplots(3, figsize=(15,10))
83
84
85 y=np.cumsum(fGn_cholesky(N,0.2))
86 t = np.linspace(0,1,y.size)
87 axs[0].plot(t,y,label='H=0.2',lw=1,c='green')
88 axs[0].legend()
89
90 y=np.cumsum(fGn_cholesky(N,0.5))
91 t = np.linspace(0,1,y.size)
92 axs[1].plot(t,y,label='H=0.5',lw=1,c='black')
93 axs[1].legend()
94
95 y=np.cumsum(fGn_cholesky(N,0.8))
96 t = np.linspace(0,1,y.size)
97 axs[2].plot(t,y,label='H=0.8',lw=1,c='red')
98 axs[2].legend()
99
100 plt.savefig('fbmcholesky.png',dpi=600)
```