INSA INSTITUT NATIONAL DES SCIENCES APPLIQUÉES TOULOUSE

Decomposing the standard Brownian Motion along the non-normalized Faber-Schauder System

Valentin Debarnot, Léo Martire

Institut National des Sciences Appliquées de Toulouse
4ème année, Génie Mathématique et Modélisation

Monday, 11th of September, 2017

Table of Contents

1 Preface

First of all, we would like to thank professor Frédéric de Gournay (University of Toulouse and National Institute of Applied Sciences, France). Indeed, this article would not have been possible without his original idea and the guidance he provided during this developement.

"Omne tulit punctum qui miscuit utile dulci.", Quintus Horatius Flaccus, Ars Poetica, l. 343.

2 Introduction

2.1 Notations

Let us be on a probability space denoted $(\Omega, \mathcal{F}, \mathbb{P})$, where Ω is the set of all possible outcomes, $\mathcal{F}$ is a σ-algebra included in $\mathcal{P}(\Omega)$, and $\mathbb{P}$ is the measure of probability over Ω.

For all function f defined on a set X, we shall denote: $\text{supp}(f) = \{x \in X \,|\, f(x) \neq 0\}$.

For given random variables X and Y, expected value shall be denoted $\mathbb{E}(X)$, variance $\text{Var}(X)$ and covariance $\text{Cov}(X, Y)$. These quantities are defined as follows.

Definition 2.1 (Expected value)**.** *Let X be a $\mathbb{P}$-integrable random variable of $(\Omega, \mathcal{F}, \mathbb{P})$. Its expected value is given by:*

$$\mathbb{E}(X) = \int_\Omega X d\mathbb{P} = \int_\Omega X(\omega)\mathbb{P}(d\omega).$$

If X admits a probability density function f_X, its expected value is given by:

$$\mathbb{E}(X) = \int_{\mathbb{R}} \omega f_X(\omega) d\omega.$$

Definition 2.2 (Covariance)**.** *Let X and Y be random variables of $(\Omega, \mathcal{F}, \mathbb{P})$, admitting a joint probability density function $f_{X,Y}$. Their covariance is given by:*

$$\text{Cov}(X, Y) = \int_{\mathbb{R}^2} (x - \mathbb{E}(X))\,(y - \mathbb{E}(Y))\, f_{X,Y}(x, y) dx dy.$$

Definition 2.3 (Variance)**.** *Let X be a random variable of $(\Omega, \mathcal{F}, \mathbb{P})$, admitting a probability density function f_X. Its variance is given by* $\text{Cov}(X, X)$*:*

$$\text{Var}(X) = \text{Cov}(X, X) = \int_{\mathbb{R}} (\omega - \mathbb{E}(X))^2 f_X(\omega) d\omega,$$

or equivalently by:

$$\text{Var}(X) = \mathbb{E}(X^2) - \mathbb{E}(X)^2 = \int_{\mathbb{R}} \omega^2 f_X(\omega) d\omega - \mathbb{E}(X)^2.$$

Calculation rules relative to the use of these probability operators shall be admitted.

Definition 2.4 (Almost surely)**.** *Let $(\Omega, \mathcal{F}, \mathbb{P})$ be a probability space.*

An event $E \in \mathcal{F}$ happens "$\mathbb{P}$-almost surely" if $\mathbb{P}(E) = 1$.

For a more natural enunciation, we often shorten "$\mathbb{P}(E) = 1$" by "$\mathbb{P}$-almost surely over $\omega \in \Omega$, E happens" since E can have a complicated definition.

2.2 Definitions

We study the stochastic process often called "Wiener process" or "standard Brownian motion" and shortened to only "Brownian motion".

Definition 2.5 (Stochastic process)**.** *A stochastic process $X = (X_t)_{t\in T}$ is a family of random variables X_t indexed by a parameter t, where t belongs to some set index T.*

Definition 2.6 (Standard Brownian motion)**.** *The d-dimension process $(B_t)_{t\in T} \in L^2(\Omega, \mathcal{F}, \mathbb{P})$ is a Wiener process or standard Brownian motion if:*

1. $\mathbb{P}(B_0 = 0) = 1$,
2. $\forall\, (t_0, \ldots, t_p) \in T^p$ *such that* $0 = t_0 < t_1 < ... < t_p$, *we have:* $\forall j \in [\![1, p]\!] \left[B_{t_j} - B_{t_{j-1}}\right]$ *are independent,* $\mathcal{N}(0, (t_j - t_{j-1}))$*-distributed random variables,*
3. $\mathbb{P}$*-almost surely over* $\omega \in \Omega$, *the function* $t \mapsto B_t(\omega)$ *is continuous (where* $B_t(\omega)$ *is a realisation of the stochastic process).*

Other types of Brownian motions will not be discussed here, but some of the given results are generalizable to the more general Brownian motion.

Hence, for the sake of the seriousness of the notations, note that when the "Brownian motion" is quoted in this article, we are in fact talking about the standard Brownian motion.

2.3 Motivation and Setting of the Problem

In our first approach, we will only consider the 1-dimensional Brownian motion.

The Brownian motion is used in applied mathematics, physics and economics. The well known construction of the Brownian motion is usually done with a random walk, but we are going to show how to represent it using a wavelet system.

To build an actually continuous Brownian motion using the random walk, we need to construct the limit of a sequence of random variables. The fact that the sequence is not necessarily of decreasing magnitude is an issue to implement.

That being said, we describe here a construction of the Brownian motion using a series of Gaussian variables and triangle functions where the first terms are of most important magnitude. This representation implies numerous other advantages, as developed in section 3.4.3.

First is demonstrated that the Brownian motion admits at least one decomposition, using the Karhunen-Loève theorem.

Then is explained how the Brownian motion can be expressed in terms of an other decomposition, using triangle functions, and that it is an optimal decomposition. Hence, with only a finite number of terms from the sum, we can approach the Brownian motion in the best way possible.

Moreover, we are going to expose that this decomposition does verify the key properties of the Brownian motion.

Finally, the objective consisting in giving an implementation of said Brownian motion in 1 dimension, we discretize the aforementioned decomposition and we implement it. The algorithm

is supposed to give us a good modelling of the Brownian motion, with as few iterations and coefficients as possible.

3 The Schauder Wavelets Decomposition and Representation of the Brownian Motion

We chose to study the Brownian motion over $[0,1]$, a closed and bounded interval. We will see later that it is possible to scale it over other type of intervals (see section 3.5).

3.1 Existence of a decomposition

Recall that we use the probability space defined earlier, *ie* $(\Omega, \mathcal{F}, \mathbb{P})$. The main goal of this section is to show that the Brownian motion admits at least a decomposition.

First, let us state a fundamental theorem which is primordial in obtaining a decomposition of the Brownian motion.

Theorem 3.1 (Karhunen-Loève)**.** *Let $(X_t)_{t\in[a,b]}$ be a centred stochastic process for with covariance function $K_X(t,s)$ continuous in t and s. Let T_{K_X} be defined by:*

$$T_{K_X} : \left| \begin{array}{rcl} L^2([a,b]) & \to & L^2([a,b]) \\ \Phi & \mapsto & T_{K_X}\Phi = \displaystyle\int_a^b K_X(t,s)\Phi(t)ds \end{array} \right. .$$

Let $(e_i(t))_{i\in I}$ be an orthonormal basis of $L^2([a,b])$ formed by its eigenvectors and let $(\lambda_i(t))_{i\in I}$ be its eigenvalues.

We define, $\forall \omega \in \Omega$:

$$Z_i(\omega) = \int_0^1 X_t(\omega)e_i(t)dt.$$

We have the following results:

1. *Z_i are centered orthogonal random variables,*
2. *$\forall \omega \in \Omega$:*

$$\forall t \in [0,1], \;\; X_t(\omega) = \sum_{i\in I} Z_i(\omega)e_i(t),$$

where the convergence is in $L^2([0,1])$ and uniform in t.

As underlined in the proof that follows (see proof 3.2), the fact that hypothesis are met (in the case of the Brownian motion in particular) is given by Mercer's theorem. Let us begin by stating it in a general case, because it is needed for the proof.

Theorem 3.2 (Mercer)**.** *Let K be a continuous symmetric non-negative definite kernel. Then there is an orthonormal basis $(e_i(t))_{i\in\mathbb{N}^*}$ on $L^2([0,1])$ consisting of eigenfunctions of T_K (as defined in theorem 3.1) such that the corresponding sequence of eigenvalues $(\lambda_i)_{i\in\mathbb{N}^*}$ is non negative.*

The eigenfunctions corresponding to non-zero eigenvalues are continuous on $[0,1]$ *and* K *has the following representation:*

$$K(s,t) = \sum_{i=1}^{\infty} \lambda_i e_i(s) e_i(t),$$

where the convergence is absolute and uniform.

Proof 3.1 (Mercer)**.** *We admit this theorem for our article, but see page 60 in [3] for a proof.* ⊡

Proof 3.2 (Karhunen-Loève)**.** *The covariance function* K_X *satisfies the definition of a Mercer kernel. By Mercer's theorem, there consequently exists a set* $\{\lambda_i,\ e_i(t)\}_{i\in\mathbb{N}^*}$ *of eigenvalues and eigenfunctions of* T_{K_X} *(as defined in theorem 3.1) forming an orthonormal basis of* $L^2([a,b])$ *(we keep generality on the interval used), and* K_X *can be expressed as:*

$$K_X(s,t) = \sum_{k=1}^{\infty} \lambda_k e_k(s) e_k(t).$$

$\mathbb{P}$*-almost surely over* $\omega \in \Omega$, $X_t(\omega)$ *can be expanded in terms of the eigenfunctions* e_k*. Hence, the process* X_t *can also be, as:*

$$X_t = \sum_{k=1}^{\infty} Z_k e_k(t),$$

where the coefficients (random variables) Z_k *are given by the projection of* X_t *on the respective eigenfunctions:*

$$Z_k = \int_a^b X_t e_k(t) dt.$$

Recall that X_t *is centered. We may now write:*

$$\begin{aligned} \mathbb{E}[Z_k] &= \mathbb{E}\left[\int_a^b X_t e_k(t)\, dt\right] \\ &= \int_a^b \mathbb{E}[X_t] e_k(t) dt \\ &= 0. \end{aligned}$$

Note that we apply Fubini's theorem, because $t \mapsto \mathbb{E}[X_t] e_k(t) \in L^2([a,b])$*.*

Moreover:

$$\begin{aligned} \mathbb{E}[Z_i Z_j] &= \mathbb{E}\left[\int_a^b \int_a^b X_t X_s e_j(t) e_i(s) dt\, ds\right] \\ &= \int_a^b \int_a^b \mathbb{E}\left[X_t X_s\right] e_j(t) e_i(s) dt\, ds \\ &= \int_a^b \int_a^b K_X(s,t) e_j(t) e_i(s) dt\, ds \\ &= \int_a^b e_i(s) \left(\int_a^b K_X(s,t) e_j(t) dt\right) ds. \end{aligned}$$

Recall that e_j *is an eigenfunction of* T_{K_X}*, meaning that* $T_{K_X} e_j = \langle K_X, e_j \rangle_{L^2([a,b])} = \lambda_j e_j$*. From this result ensues that:*

$$\mathbb{E}[Z_i Z_j] = \lambda_j \int_a^b e_i(s) e_j(s) ds.$$

Since e_k functions are eigenfunctions of T_{K_X} and are orthonormal, we finally have:

$$\mathbb{E}[Z_i Z_j] = \delta_{ij}\lambda_i,$$

where λ_i is the eigenvalue associated to e_i and δ_{ij} is the Kronecker symbol ($\delta_{ij} = 1$ if $i = j$ and 0 if $i \neq j$).

That proves the first result of the theorem.

Let us now show that the convergence is in L^2. For all t, let $S_N(t) = \sum_{k=1}^{N} Z_k e_k(t)$. We shorten $S_N(t)$ to S_N for the continuation.

Then write the convergency:

$$\begin{aligned}
\mathbb{E}\left[|X_t - S_N|^2\right] &= \mathbb{E}\left[X_t^2\right] + \mathbb{E}\left[S_N^2\right] - 2\mathbb{E}\left[X_t S_N\right] \\
&= K_X(t,t) + \mathbb{E}\left[\sum_{k=1}^{N}\sum_{l=1}^{N} Z_k Z_l e_k(t) e_l(t)\right] - 2\mathbb{E}\left[X_t \sum_{k=1}^{N} Z_k e_k(t)\right].
\end{aligned}$$

Recall that Z_k is given by the projection of X_t on e_k:

$$\begin{aligned}
\mathbb{E}\left[|X_t - S_N|^2\right] &= K_X(t,t) + \sum_{k=1}^{N} \lambda_k e_k(t)^2 - 2\mathbb{E}\left[\sum_{k=1}^{N}\int_a^b X_t X_s e_k(s) e_k(t) ds\right] \\
&= K_X(t,t) + \sum_{k=1}^{N} \lambda_k e_k(t)^2 - 2\sum_{k=1}^{N}\int_a^b \mathbb{E}\left[X_t X_s\right] e_k(s) e_k(t) ds \\
&= K_X(t,t) + \sum_{k=1}^{N} \lambda_k e_k(t)^2 - 2\sum_{k=1}^{N}\int_a^b K_X(t,s) e_k(s) e_k(t) ds.
\end{aligned}$$

As before, recall that e_k is an eigenfunction of T_{K_X}. Finally:

$$\begin{aligned}
\mathbb{E}\left[|X_t - S_N|^2\right] &= K_X(t,t) + \sum_{k=1}^{N} \lambda_k e_k(t)^2 - 2\sum_{k=1}^{N}\int_a^b \lambda_k e_k(t) e_k(t) ds \\
&= K_X(t,t) - \sum_{k=1}^{N} \lambda_k e_k(t)^2 \\
&\underset{N\to+\infty}{\longrightarrow} 0
\end{aligned}$$

by Mercer's theorem. Hence the series converges in L^2 and that proves the second result of the theorem and concludes the proof. ⊡

It is easy to see that the Mercer's theorem hypothesis are met when we take K as the covariance function of the Brownian motion, *ie* defined by $K : (t,s) \mapsto \mathrm{Cov}(B_t, B_s)$. It is:

1. continuous (below, see proposition 3.1 and corresponding proof),
2. symmetric (we clearly have $\mathrm{Cov}(B_{t_1}, B_{t_2}) = \mathrm{Cov}(B_{t_2}, B_{t_1}), \forall t_1, t_2 \in [0,1]$),
3. and non-negative definite ($\mathrm{Cov}(B_t, B_t) = \mathrm{Var}(B_t) > 0$ as long as $t \neq 0$).

Proposition 3.1 (Continuity of the covariance function)**.** *Let $t_1, t_2 \in [0,1]$ such that $t_1 < t_2$. The Brownian motion covariance function,* $\mathrm{Cov}\left(B(t_1), B(t_2)\right)$ *is continuous.*

Proof 3.3 (Continuity of the covariance function)**.** *Let $t_1, t_2 \in [0,1]$ such that $t_1 < t_2$. We have:*

$$\begin{aligned}\mathrm{Cov}\left(B(t_1), B(t_2)\right) &= \mathbb{E}\left[\left(B(t_1) - \mathbb{E}\left[B(t_1)\right]\right)\left(B(t_2) - \mathbb{E}\left[B(t_2)\right]\right)\right] \\ &= \mathbb{E}\left[B(t_1)B(t_2)\right] \\ &= \mathbb{E}\left[B(t_1)\left(B(t_2) - B(t_1) + B(t_1)\right)\right] \\ &= \mathbb{E}\left[B(t_1)\left(B(t_2) - B(t_1)\right)\right] + \mathbb{E}\left[B(t_1)^2\right]\end{aligned}$$

Since $B(t_1)$ and $(B(t_2) - B(t_1))$ are idenpendent (by a property of the Brownian motion), we have:

$$\begin{aligned}\mathrm{Cov}\left(B(t_1), B(t_2)\right) &= \mathbb{E}\left[B(t_1)\right]\mathbb{E}\left[B(t_2) - B(t_1)\right] + \mathbb{E}\left[B(t_1)^2\right] \\ &= \mathbb{E}\left[B(t_1)^2\right] \\ &= t_1\end{aligned}$$

(because $B(t_1)$ is zero-mean and because $\mathbb{E}\left[B(t_1)^2\right] = t_1$).

It is equivalent to show that a bilinear form is continuous over $[0,1]^2$ or to show that it is continuous in $(0,0)$, which is easy to see in our case with the equality above. Hence, the covariance function is continuous. ⊡

It is known that the Brownian motion $(B(t))_{t\in[0,1]}$ is a zero-mean square-integrable stochastic process. We put ourselves in a closed bounded interval and the covariance function is continuous (as shown before). Mercer's theorem's (3.2) hypothesis and thus Karhunen-Loève theorem's (3.1) hypothesis are met, hence we have the following result.

Corollary 3.1. *The Brownian motion $(B_t)_{t\in[0,1]}$ admits the following representation:*

$$B_t(\omega) = \sum_{i\in I} Z_i(\omega) e_i(t),\ \forall \omega \in \Omega. \tag{1}$$

where:

- *I is an index set,*
- *the convergence is in L^2 and uniform in t,*
- *$(Z_i(\omega))_i$ are realisations of random variables $(Z_i)_i$ that are:*
  - *are zero-mean,*
  - *have a joint Gaussian distribution,*
  - *and are stochastically independent,*
- *the vectors $(e_i)_{i\in I}$ form an orthonormal basis of $L^2[0,1]$.*

This result give the essence of the fact that the Brownian motion is decomposable.

3.2 Learning to express the Brownian motion

The Brownian motion can be decomposed, as proved in 3.1. Let us now focus on two interesting representations. The construction of these is not developed here, but the Brownian motion properties are verified in 3.4.1.

Proposition 3.2. *(Brownian motion representation) Antoine Ayache [1] gives two different representations for the Brownian motion defined on* $[0,1]$*:*

1. *Trigonometric formula:*

$$\forall t \in [0,1], \forall \omega \in \Omega, \;\; B_t(\omega) = g_0(\omega)t + \sqrt{2}\sum_{k=1}^{+\infty} g_k(\omega)\frac{sin(\pi k t)}{\pi k}.$$

2. *Faber-Schauder formula (introduced by Paul Lévy in 1948):*

$$\forall t \in [0,1], \forall \omega \in \Omega, \;\; B_t(\omega) = g_0(\omega)t + \sum_{j=0}^{+\infty}\sum_{k=1}^{2^j-1} g_{j,k}(\omega)2^{-j/2}\tau(2^j t - k). \tag{2}$$

Where:

- g_0 *and* $g_{j,k}$ *are independent real-valued* $\mathcal{N}(0,1)$*-distributed random variables,*
- τ *is the triangle function based on* $[0,1]$*,*
- *the series is almost surely uniformly convergent in* $t \in [0,1]$*.*

Remark 3.1. *The trigonometric formula comes directly from the decomposition induced by the Karhunen-Loève theorem (see 3.1). This will not be proved here.*

Remark 3.2. *It is to be remarked that the series expansion of the Brownian motion in Faber-Schauder system is optimal. See the proof in [1] (starting at slide number 27).*

Remark 3.3. τ *is the triangle function based on* $[0,1]$*. We denote:*

$$\forall n \in \mathbb{N}, \;\; \forall k \in [\![0, 2^n - 1]\!], \;\; \tau_{n,k} : t \mapsto 2^{-\frac{n}{2}}\tau\left(2^n t - k\right).$$

$\tau_{n,k}$ *is a "little tent" that is:*

- *of height* $2^{-1-\frac{n}{2}}$*,*
- *of width* $\dfrac{2}{2^{n+1}}$*,*
- *and centered on* $\dfrac{2k+1}{2^{n+1}}$*.*

We can illustrate it:

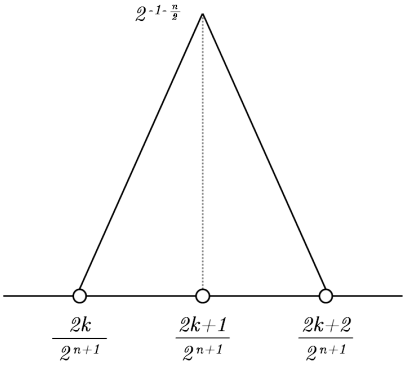


Figure 1 – $\tau_{n,k}(t)$: a "little tent".

See some examples of $\tau_{n,k}$ for low values of n and k:

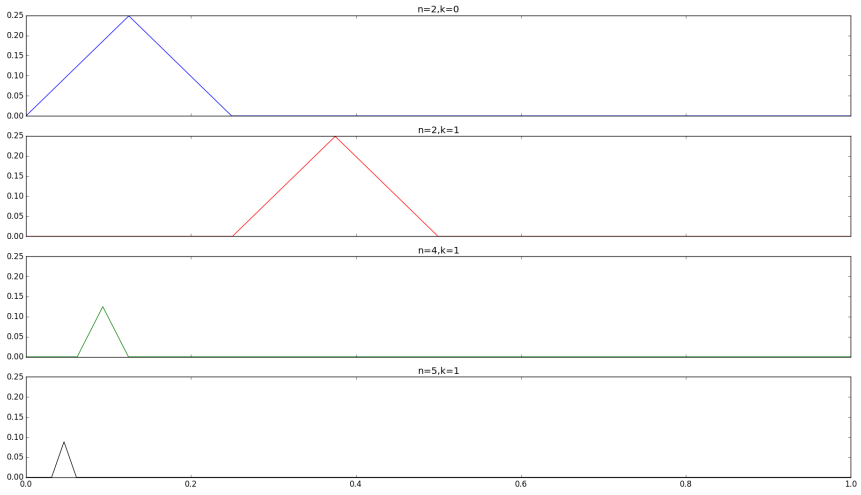


Figure 2 – $\tau_{2,0}(t)$, $\tau_{2,1}(t)$, $\tau_{4,1}(t)$ and $\tau_{5,1}(t)$ from $t = 0$ to $t = 1$.

3.3 The Decomposition along the non-normalized Faber-Schauder System

Let us start by a definition of the Faber-Schauder System.

Definition 3.1 (Faber-Schauder system)**.** *The Faber–Schauder system is the family of continuous functions on $[0, 1]$ consisting of the constant function 1, and of multiples of indefinite integrals of the functions in the Haar system on $[0, 1]$, chosen to have norm 1 in the maximum norm:*

$$\forall t \in [0,1], \quad \begin{cases} s_0(t) = 1, \\ s_1(t) = \displaystyle\int_0^t \phi(u)du = t, \\ \forall n \in \mathbb{N}, \quad \forall k \in [\![0, 2^n - 1]\!], \quad s_{n,k}(t) = 2^{1+n/2} \displaystyle\int_0^t \psi_{n,k}(u)du, \end{cases}$$

where the Haar system functions ϕ and $(\psi_{n,k})_{(n,k)\in\mathbb{Z}^2}$ are defined below.

Definition 3.2 (Scaling function)**.** *The scaling function ϕ of the Haar system is defined by:*

$$\phi : t \mapsto \begin{cases} 1 & \text{if } 0 \leq t < 1, \\ 0 & \text{otherwise.} \end{cases}$$

Definition 3.3 (Haar wavelet)**.** *The Haar base wavelet ψ is defined by:*

$$\psi : t \mapsto \begin{cases} 1 & \text{if } 0 \leq t < \frac{1}{2}, \\ -1 & \text{if } \frac{1}{2} \leq t < 1, \\ 0 & \text{otherwise.} \end{cases}$$

Definition 3.4 (Haar function)**.** *For every $(n,k) \in \mathbb{Z}^2$, the Haar function $\psi_{n,k}$ is defined by:*

$$\psi_{n,k} : t \mapsto 2^{\frac{n}{2}} \psi\left(2^n t - k\right).$$

$(\tau_{n,k})_{(n,k)\in\mathbb{N}\times[\![0,2^n-1]\!]}$ functions used in the decomposition (2) are closely linked to the Faber-Schauder system. Indeed, we have the following result:

Proposition 3.3. *$(\tau_{n,k})_{(n,k)\in\mathbb{N}\times[\![0,2^n-1]\!]}$ functions are primitives of Haar functions, ie:*

$$\forall n \in \mathbb{N},\ \ \forall k \in [\![0, 2^n - 1]\!],\ \ \forall t \in \mathbb{R},\ \ \tau_{n,k}(t) = \langle \mathbb{1}_{[0,t]}, \psi_{n,k} \rangle_{L^2(\mathbb{R})}. \tag{3}$$

Proof 3.4. *Let $(n,k) \in \mathbb{N} \times [\![0, 2^n - 1]\!]$.*

$$\begin{aligned} \int_0^t \psi_{n,k}(x)dx &= \int_0^t 2^{\frac{n}{2}} \psi(2^n x - k)dx \\ &= 2^{\frac{n}{2}} \int_{-k}^{2^n t - k} \psi(u) \frac{du}{2^n} \quad \textit{(we set } u = 2^n x - k\textit{)} \\ &= 2^{-\frac{n}{2}} \int_{-k}^{2^n t - k} \psi(u)du. \end{aligned}$$

Suppose first that $k \geq 0$ (the proof follows the same pattern for $k \leq 0$).
We have $\mathrm{supp}(\psi) \cap [-k, 2^n t - k] = [0, 2^n t - k]$*. We distinguish 4 cases.*

- *Case 1:* $2^n t - k \geq 1$*, ie* $t \in \left[\frac{2k+2}{2^{n+1}}, +\infty\right]$.

  We have $\mathrm{supp}(\psi) \cap [0, 2^n t - k] = [0, 1]$*, hence:*

  $$\int_0^t \psi_{n,k}(x)dx = 0.$$

- *Case 2:* $2^n t - k \in \left[\frac{1}{2}, 1\right]$*, ie* $t \in \left[\frac{2k}{2^{n+1}}, \frac{2k+2}{2^{n+1}}\right]$.

We have $\text{supp}(\psi) \cap [0, 2^n t - k] = \left[0, \frac{1}{2}\right] \cup \left[\frac{1}{2}, 2^n t - k\right]$, *hence:*

$$\begin{aligned}\int_0^t \psi_{n,k}(x)dx &= 2^{-\frac{n}{2}} \left(\frac{1}{2} + \int_{\frac{1}{2}}^{2^n t-k} \psi(u)du\right) \\ &= 2^{-\frac{n}{2}} \left(\frac{1}{2} + [-u]_{\frac{1}{2}}^{2^n t-k}\right) \\ &= 2^{-\frac{n}{2}} \left(\frac{1}{2} - 2^n t + k + \frac{1}{2}\right) \\ &= 2^{-\frac{n}{2}} \left(-2^n t + 1 + k\right).\end{aligned}$$

- *Case 3:* $2^n t - k \in \left[0, \frac{1}{2}\right]$, *ie* $t \in \left[\frac{2k+1}{2^{n+1}}, \frac{2k}{2^{n+1}}\right]$.

  We have $\text{supp}(\psi) \cap [0, 2^n t - k] = [0, 2^n t - k]$, *hence:*

$$\begin{aligned}\int_0^t \psi_{n,k}(x)dx &= 2^{-\frac{n}{2}} \left(\int_0^{2^n t-k} \psi(u)du\right) \\ &= 2^{-\frac{n}{2}} \left([u]_0^{2^n t-k}\right) \\ &= 2^{-\frac{n}{2}} \left(2^n t - k\right) \\ &= 2^{-\frac{n}{2}} \left(2^n t - k\right).\end{aligned}$$

- *Case 4:* $2^n t - k \leq 0$, *ie* $t \in \left[-\infty, \frac{2k+1}{2^{n+1}}\right]$.

  We have $\text{supp}(\psi) \cap [0, 2^n t - k] = \varnothing$, *hence:*

$$\int_0^t \psi_{n,k}(x)dx = 0.$$

We find that $\int_0^t \psi_{n,k}(x)dx$ *follows perfectly the four expressions of* $\tau_{n,k}$ *as we described it in remark 3.3. Also, the demonstration is exactly the same by "symmetrising" it along the y-axis for* $k \leq 0$.

Finally, we have:

$$\int_0^t \psi_{n,k}(x)dx = 2^{-\frac{n}{2}} \tau(2^n t - k) = \tau_{n,k}(t).$$

Allowing us to write:

$$\forall (n,k) \in \mathbb{Z}^2, \ \ \forall t \in \mathbb{R}, \ \ \tau_{n,k}(t) = \langle \mathbb{1}_{[0,t]}, \psi_{n,k} \rangle_{L^2(\mathbb{R})}.$$

⊡

We can write a clear relation between $(\tau_{n,k})_{(n,k)\in\mathbb{N}\times[\![0,2^n-1]\!]}$ functions and those of the Faber-Schauder system $(s_{n,k})_{(n,k)\in\mathbb{N}\times[\![0,2^n-1]\!]}$:

$$\forall n \in \mathbb{N}, \ \ \forall k \in [\![0, 2^n - 1]\!], \ \ \tau_{n,k} = 2^{-1-n/2} s_{n,k}.$$

That is, our $(\tau_{n,k})_{(n,k)\in\mathbb{N}\times[\![0,2^n-1]\!]}$ functions are only non-normalized Faber-Schauder functions.

We can clearly see that the expression given in (2) uses from the Faber-Schauder system only s_1 and the non-normalized $(s_{n,k})_{(n,k)\in\mathbb{N}\times[\![0,2^n-1]\!]}$. Using a system that is not normalized is essential in obtaining the decrement of the series' tail. Not using the first element of the Faber-Schauder system s_0 (or setting its component to 0) guaranties that the Brownian Motion will be zero-meaned, but this can be modified to the user's convenience.

3.3.1 Interesting results between Haar wavelets and Schauder wavelets

We focus in this subsection on Haar functions because they make up a satisfaying Hilbert basis of $L^2(\mathbb{R})$. Note that we are using the definitions at the beginning of section 3.3 for Haar functions.

Let us recall the definition of a Hilbert basis.

Definition 3.5 (Hilbert basis)**.** *Let H be a Hilbert set and $(\phi_i)_{i\in I}$ be a set of vectors of H where I is a set index. $(\phi_i)_{i\in I}$ is a Hilbert basis of H if:*

- *$(\phi_i)_{i\in I}$ is orthonormal,*
- *$(\phi_i)_{i\in I}$ is complete, ie $\forall x \in H,\ \exists(\lambda_i)_{i\in I}$ such that $x = \sum_{i\in I} \lambda_i\phi_i$.*

Proposition 3.4. *Haar functions define a Hilbert basis of $L^2(\mathbb{R})$.*

Proof 3.5 (Haar functions define a Hilbert basis of $L^2(\mathbb{R})$)**.** *Such a basis must be orthogonal (1), normalized (2) and complete (3).*
We remind that for every $(n,k)\in\mathbb{Z}^2$, the Haar function $\psi_{n,k}$ is defined by:

$$\psi_{n,k} : t \mapsto 2^{\frac{n}{2}}\psi\left(2^n t - k\right).$$

1. *Let $n, n', k, k' \in \mathbb{Z}$.*

   We note that $\mathrm{supp}(\psi_{n,k}) = I_{n,k}$, *where* $I_{n,k} = \left[2^{-n}k, 2^{-n}(1+k)\right]$.

   (a) *If $k \neq k'$ and $n = n'$:* $I_{n,k} \cap I_{n,k'} = \emptyset$.

   $$\begin{aligned}\langle\psi_{n,k}, \psi_{n,k'}\rangle_{L^2(\mathbb{R})} &= \int_{\mathbb{R}} \psi_{n,k}(t)\psi_{n',k'}(t)dt \\ &= 0.\end{aligned}$$

   (b) *If $k = k'$ and $n \neq n'$. Without loss of generality we considere the case $n > n'$. We have 3 cases:*

   i. *If $n \leq n'\dfrac{\log_2(k)}{\log_2(k+1)}$:* $I_{n,k} \cap I_{n,k'} = \emptyset$.

   $$\langle\psi_{n,k}, \psi_{n',k}\rangle_{L^2(\mathbb{R})} = 0.$$

ii. *If* $n < n'\dfrac{\log_2(k)}{\log_2(k+1)}$*:* $I_{n,k} \cap I_{n,k'} = \left[2^{-n'}k, 2^{-n}(1+k)\right]$.

$$\begin{aligned}
\langle \psi_{n,k}; \psi_{n',k} \rangle_{L^2(\mathbb{R})} &= \int_{\mathbb{R}} \psi_{n,k}(t)\psi_{n',k}(t)dt \\
&= \int_{\mathbb{R}} 2^{\frac{n}{2}} \psi\left(2^n t - k\right) 2^{\frac{n'}{2}} \psi\left(2^{n'} t - k\right) dt \\
&= 2^{\frac{n+n'}{2}} \int_{2^{-n'}k}^{2^{-n}(k+1)} \psi\left(2^n t - k\right) \psi\left(2^{n'} t - k\right) dt \qquad \textit{(we set } u = 2^n t - k\textit{)} \\
&= 2^{\frac{n+n'}{2}} \int_{(2^{n-n'}-1)k}^{1} \psi\left(u\right) \psi\left(2^{n'-n} u + (2^{n'-n} - 1)k\right) du \\
&= 0
\end{aligned}$$

Because $(2^{n-n'} - 1)k \geq 1$*, so* $\psi(u) = 0$ *on the integral set.*

(c) If $k \neq k'$ *and* $n \neq n'$.

i. *If* $I_{n,k} \cap I_{n',k'} = \emptyset$*: obvious.*

ii. *If* $I_{n,k} \cap I_{n',k'} = I_{n,k}$*, with* $I_{n',k'}$ *center in* $\dfrac{2k'+1}{2^{n'+1}}1$.

$$\begin{aligned}
\langle \psi_{n,k}; \psi_{n',k'} \rangle_{L^2(\mathbb{R})} &= \int_{I_{n,k}} \psi_{n,k}(t)\psi_{n',k'}(t)dt \\
&= \pm \int_{I_{n,k}} \psi_{n,k}(t)dt \\
&= 0
\end{aligned}$$

Because supp $\psi_{n,k} \in \left[2^{-n'}k'; 2^{-n'}k' + 2^{-n'-1}\right]$*, or supp* $\psi_{n,k} \in \left[2^{-n'}k' + 2^{-n'-1}; 2^{-n'}(k'+1)\right]$*, ie* $\psi_{n,k}\psi_{n',k'} = \pm\psi_{n,k}$*, since,* $\dfrac{2k'+1}{2^{n'+1}} \notin \overset{\circ}{I}_{n,k}$ *where* $\dfrac{2k'+1}{2^{n'+1}}$ *is the middle of the interval* $I_{n',k'}$*, ie where the sign of* $\psi_{n',k'}$ *change.*

iii. *If* $I_{n,k} \cap I_{n',k'} = I_{n',k'}$*: same as above.*

iv. *If* $I_{n,k} \cap I_{n',k'} = \left[2^{-n'}k', 2^{-n}(1+k)\right]$*: without loss of generality we considere the case* $n > n'$.

$$\begin{aligned}
\langle \psi_{n,k}; \psi_{n',k'} \rangle_{L^2(\mathbb{R})} &= \int_{\mathbb{R}} \psi_{n,k}(t)\psi_{n',k'}(t)dt \\
&= \int_{\mathbb{R}} 2^{\frac{n}{2}} \psi\left(2^n t - k\right) 2^{\frac{n'}{2}} \psi\left(2^{n'} t - k'\right) dt \\
&= 2^{\frac{n+n'}{2}} \int_{2^{-n'}k'}^{2^{-n}(k+1)} \psi\left(2^n t - k\right) \psi\left(2^{n'} t - k'\right) dt \qquad \textit{(we set } u = 2^n t - k\textit{)} \\
&= 2^{\frac{n+n'}{2}} \int_{2^{n-n'}k-k'}^{1} \psi\left(u\right) \psi\left(2^{n'-n} u + 2^{n'-n}k - k'\right) du \\
&= 0
\end{aligned}$$

Since $n > n'$*,* $n - n' \geq 1$*, so* $2^{n-n'} \geq 2$*, then* $2^{n-n'}k - k' \geq k - k' \geq 1$*, that prove the latest equality.*

v. *If* $I_{n,k} \cap I_{n',k'} = \left[2^{-n}k, 2^{-n'}(1+k')\right]$*: same as above.*

2. Let $n, k \in \mathbb{Z}$.

$$\begin{aligned}
\|\psi_{n,k}\|^2_{L^2(\mathbb{R})} &= \int_{\mathbb{R}} |\psi_{n,k}(t)|^2 dt \\
&= \int_{\mathbb{R}} \left(2^{\frac{n}{2}} \psi\left(2^n t - k\right)\right)^2 dt \\
&= 2^n \int_{\mathbb{R}} \psi\left(2^n t - k\right)^2 dt \\
&= 2^n \int_{\mathbb{R}} \psi\left(u\right)^2 \frac{du}{2^n} \qquad \textit{(we set } u = 2^n t - k\textit{)} \\
&= \int_0^1 1 du \\
&= 1
\end{aligned}$$

3. In order to prove the completeness of the Haar system, we remark that any step function on a dyadic interval is a linear combination of the Haar system and that any L^2 function can be approximated by a sequence of such step functions defined on dyadic intervals of decreasing length by taking the average of the target function on the corresponding interval. In interest of time we do not lengthen this point.

Hence, the Haar system $(\psi_{n,k})_{(n,k)\in\mathbb{Z}^2}$ (as defined in 3.4) makes up a Hilbert basis of $L^2(\mathbb{R})$. ⊡

3.4 Viability of the Faber-Schauder Formula

We now focus on studying the Faber-Schauder formula, given by, recall (2):

$$\forall t \in [0,1], \forall \omega \in \Omega, \;\; B_t(\omega) = g_0(\omega)t + \sum_{j=0}^{+\infty} \sum_{k=1}^{2^j-1} g_{j,k}(\omega)\tau_{j,k}(t),$$

where g_0 and $g_{j,k}$ are independent real-valued $\mathcal{N}(0,1)$-distributed random variables and $\tau_{j,k}$ is the hat function described in remark 3.3.

As it has been said, this representation is optimal and presents multiple advantages (see section 3.4.3).

3.4.1 Properties verified by the Faber-Schauder representation

Let us now prove that the Faber-Schauder formula defines a Brownian motion according to the definition 2.6.

Proof 3.6 (The Faber-Schauder formula defines a Brownian motion)**.** *Let us enumerate the properties the formula verifies. Let $\omega \in \Omega$ be a realisation.*

1. We have:

$$\begin{aligned}
B_0(\omega) &= g_0(\omega) \cdot 0 + \sum_{j=0}^{+\infty} \sum_{k=1}^{2^j-1} g_{j,k}(\omega)\tau_{j,k}(0) \\
&= \sum_{j=0}^{+\infty} \sum_{k=1}^{2^j-1} g_{j,k}(\omega)\tau_{j,k}(0).
\end{aligned}$$

But we can easily see from (3) *that* $\forall j \in \mathbb{N},\ \forall k \in [\![0, 2^n - 1]\!],\ \tau_{j,k}(0) = 0$. *Hence:*

$$B_0(\omega) = 0,$$

and trivially:

$$\mathbb{P}(B_0 = 0) = 1.$$

Which proves the first property the Brownian motion must fulfil.

2. *Let* $t_1, t_2, t_3, t_4 \in [0, 1]$ *such that* $t_1 < t_2 < t_3 < t_4$. *Denote* $B_{2-1}(\omega) = [B(t_2)(\omega) - B(t_1)(\omega)]$ *and* $B_{4-3}(\omega) = [B(t_4)(\omega) - B(t_3)(\omega)]$. *We have:*

$$\begin{cases} B_{2-1}(\omega) = g_0(\omega)(t_2 - t_1) + \displaystyle\sum_{j=0}^{+\infty} \sum_{k=1}^{2^j-1} g_{j,k}(\omega) \left(\tau_{j,k}(t_2) - \tau_{j,k}(t_1)\right), \\ B_{4-3}(\omega) = g_0(\omega)(t_3 - t_4) + \displaystyle\sum_{j=0}^{+\infty} \sum_{k=1}^{2^j-1} g_{j,k}(\omega) \left(\tau_{j,k}(t_3) - \tau_{j,k}(t_4)\right). \end{cases}$$

We easily see that B_{2-1} *and* B_{4-3} *are normally distributed zero-mean vectors as they both are linear combinations of* $\mathcal{N}(0, 1)$*-distributed random variables. Let us now determine their variance and prove that they are independent.*

First, we know that g_0 *and all* $g_{j,k}$ *are independent and of variance 1. We deduce:*

$$\begin{aligned} \mathrm{Var}\left(B_{2-1}\right) &= (t_2 - t_1)^2 \mathrm{Var}\left(g_0\right) + \sum_{j=0}^{+\infty} \sum_{k=1}^{2^j-1} \left(\tau_{j,k}(t_2) - \tau_{j,k}(t_1)\right)^2 \mathrm{Var}\left(g_{j,k}\right) \\ &= (t_2 - t_1)^2 + \sum_{j=0}^{+\infty} \sum_{k=1}^{2^j-1} \left(\tau_{j,k}(t_2) - \tau_{j,k}(t_1)\right)^2 \\ &= (t_2 - t_1)^2 + \sum_{j=0}^{+\infty} \sum_{k=1}^{2^j-1} \left(\langle \mathbb{1}_{[0,t_2]}, \psi_{j,k} \rangle_{L^2(\mathbb{R})} - \langle \mathbb{1}_{[0,t_1]}, \psi_{j,k} \rangle_{L^2(\mathbb{R})}\right)^2 \\ &= (t_2 - t_1)^2 + \sum_{j=0}^{+\infty} \sum_{k=1}^{2^j-1} \langle \mathbb{1}_{[t_1,t_2]}, \psi_{j,k} \rangle^2_{L^2(\mathbb{R})}. \end{aligned} \tag{4}$$

Notice that the Var *operator can be shifted under the infinite sum thanks to Lebesgue's dominated convergence theorem. Indeed,* $t \mapsto g_{j,k}(\omega)\left(\tau_{j,k}(t_2) - \tau_{j,k}(t_1)\right)$ *is continuous (almost surely), and can be majored by its supremum which is fast-decreasing (with remark 3.3, we have* $\forall (n, k) \in \mathbb{Z}^2,\ \forall (t_1, t_2) \in [0, 1],\ 0 \leq \tau_{j,k}(t_2) - \tau_{j,k}(t_1) \leq 2^{-j/2}$*).*

The Haar functions Hilbert basis of $L^2(\mathbb{R})$ *allows us to write:*

$$\mathbb{1}_{[t_1,t_2]} = \langle \mathbb{1}_{[t_1,t_2]}, \phi \rangle_{L^2(\mathbb{R})} \phi + \sum_{j=0}^{+\infty} \sum_{k=1}^{2^j-1} \langle \mathbb{1}_{[t_1,t_2]}, \psi_{j,k} \rangle_{L^2(\mathbb{R})} \psi_{j,k},$$

where ϕ is the scaling function (see definition 3.2). Take the $L^2(\mathbb{R})$-norm of this:

$$\begin{aligned}
\langle \mathbb{1}_{[t_1,t_2]}, \mathbb{1}_{[t_1,t_2]} \rangle_{L^2(\mathbb{R})} &= \langle \mathbb{1}_{[t_1,t_2]}, \phi \rangle_{L^2(\mathbb{R})} \langle \phi, \mathbb{1}_{[t_1,t_2]} \rangle_{L^2(\mathbb{R})} \\
&\quad + \sum_{j=0}^{+\infty} \sum_{k=1}^{2^j-1} \langle \mathbb{1}_{[t_1,t_2]}, \psi_{j,k} \rangle_{L^2(\mathbb{R})} \langle \psi_{j,k}, \mathbb{1}_{[t_1,t_2]} \rangle_{L^2(\mathbb{R})} \\
\Leftrightarrow \quad \left\| \mathbb{1}_{[t_1,t_2]} \right\|^2_{L^2(\mathbb{R})} &= \langle \mathbb{1}_{[t_1,t_2]}, \phi \rangle^2_{L^2(\mathbb{R})} + \sum_{j=0}^{+\infty} \sum_{k=1}^{2^j-1} \langle \mathbb{1}_{[t_1,t_2]}, \psi_{j,k} \rangle^2_{L^2(\mathbb{R})} \\
\Leftrightarrow \quad (t_2 - t_1) &= (t_2 - t_1)^2 + \sum_{j=0}^{+\infty} \sum_{k=1}^{2^j-1} \langle \mathbb{1}_{[t_1,t_2]}, \psi_{j,k} \rangle^2_{L^2(\mathbb{R})} \\
\Leftrightarrow \quad \sum_{j=0}^{+\infty} \sum_{k=1}^{2^j-1} \langle \mathbb{1}_{[t_1,t_2]}, \psi_{j,k} \rangle^2_{L^2(\mathbb{R})} &= (t_2 - t_1) - (t_2 - t_1)^2 .
\end{aligned} \tag{5}$$

With (4) and (5), we can finally write:

$$\begin{aligned}
\mathrm{Var}\left(B_{2-1}\right) &= (t_2 - t_1)^2 + (t_2 - t_1) - (t_2 - t_1)^2 \\
&= (t_2 - t_1) .
\end{aligned}$$

Hence, we have that B_{2-1} is a $\mathcal{N}(0, (t_2 - t_1))$-distributed random variable and we can prove that B_{4-3} is a $\mathcal{N}(0, (t_4 - t_3))$-distributed random variable by following a similar proof.

Finally, let us prove that B_{2-1} and B_{4-3} are independent, ie show that $\mathbb{E}\left(B_{2-1}B_{4-3}\right) = \mathbb{E}\left(B_{2-1}\right)\mathbb{E}\left(B_{4-3}\right)$. Since $\mathbb{E}\left(B_{2-1}\right) = \mathbb{E}\left(B_{4-3}\right) = 0$, this amounts to show that $\mathbb{E}\left(B_{2-1}B_{4-3}\right) = 0$. Let us begin:

$$\begin{aligned}
B_{2-1}(\omega)B_{4-3}(\omega) &= g_0^2(\omega)(t_2 - t_1)(t_3 - t_4) \\
&\quad + \sum_{j=0}^{+\infty} \sum_{k=1}^{2^j-1} g_0(\omega) g_{j,k}(\omega) \left(\tau_{j,k}(t_4) - \tau_{j,k}(t_3)\right)(t_2 - t_1) \\
&\quad + \sum_{j=0}^{+\infty} \sum_{k=1}^{2^j-1} g_0(\omega) g_{j,k}(\omega) \left(\tau_{j,k}(t_2) - \tau_{j,k}(t_1)\right)(t_4 - t_3) \\
&\quad + \left(\sum_{j=0}^{+\infty} \sum_{k=1}^{2^j-1} g_{j,k}(\omega) \left(\tau_{j,k}(t_2) - \tau_{j,k}(t_1)\right) \right) \left(\sum_{j=0}^{+\infty} \sum_{k=1}^{2^j-1} g_{j,k}(\omega) \left(\tau_{j,k}(t_3) - \tau_{j,k}(t_4)\right) \right)
\end{aligned} \tag{6}$$

Recall that g_0 and all $g_{j,k}$ are independent and zero-mean. Hence $\mathbb{E}\left(g_0 g_{j,k}\right) = \mathbb{E}\left(g_0\right)\mathbb{E}\left(g_{j,k}\right) = 0$. We deduce from this that:

$$\mathbb{E}\left(B_{2-1}B_{4-3}\right) = 0$$

Indeed: all the terms are null in expected value. The first term of (6) is only dependent of g_0, and the two following sums of $g_0 g_{j,k}$. For the last term, when we develop the product we obtain a sum of terms of type $g_{j,k}g_{j',k'}$, which expected value is also null.

Hence $\mathbb{E}(B_{2-1}B_{4-3}) = 0$, which proves that B_{2-1} and B_{4-3} are independent random variables.

Since this demonstration is valid for all $t_1, t_2, t_3, t_4 \in [0, 1]$, we have proven the second property the Brownian motion must fulfill (see 2.6).

3. Recall the Brownian motion representation in the Faber Schauder system (see (2)):

$$\forall t \in [0,1], \forall \omega \in \Omega \;\; B_t(\omega) = g_0(\omega)t + \sum_{j=0}^{+\infty} \sum_{k=1}^{2^j-1} g_{j,k}(\omega)\tau_{j,k}(t),$$

and (3):

$$\forall n \in \mathbb{N}, \;\; \forall k \in [\![0, 2^n - 1]\!], \;\; \forall t \in \mathbb{R}, \;\; \tau_{n,k}(t) = \langle \mathbb{1}_{[0,t]}, \psi_{n,k} \rangle_{L^2(\mathbb{R})}.$$

First, it is necessary to show that the series converge uniformly over $[0,1]$, for all $\omega \in \Omega$, except for ω in $\mathcal{A} \in \mathcal{F}$ where $\mathcal{A}$ is of null probability.

Notice, as said in remark 3.3, $0 \leq \tau_{j,k} \leq 2^{-j/2}$. Moreover, for j fixed and $k \in [\![0, 2^j - 1]\!]$, all $\tau_{j,k}$ functions have disjoint supports.

Hence, $\forall t \in [0,1], \; \forall \omega \in \Omega$:

$$\sup_{t\in[0,1]} \left| \sum_{k=0}^{2^j-1} g_{j,k}(\omega)\tau_{j,k}(t) \right| \leq 2^{-j/2} \sup_{0\leq k\leq 2^j-1} |g_{j,k}(\omega)| .$$

Let us state a short lemma.

Lemma 3.1. *If $N \sim \mathcal{N}(0,1)$, then $\forall a \geq 1$:*

$$\mathbb{P}(|N| \geq a) \leq e^{-a^2/2}.$$

Proof 3.7.

$$\mathbb{P}(|N| \geq a) = \frac{2}{\sqrt{2\pi}} \int_a^\infty e^{-x^2/2} dx \leq \frac{2}{\sqrt{2\pi}} \int_a^\infty \frac{x}{a} e^{-x^2/2} dx = \frac{2}{a\sqrt{2\pi}} e^{-a^2/2}$$

⊡

Since $g_{j,k}$ are all Gaussian random variables we may use the lemma. We have:

$$\begin{aligned} \mathbb{P}\left(\sup_{0\leq k\leq 2^j-1} |g_{j,k}(\omega)| > 2^{j/4} \right) &\leq \sum_{k=0}^{2^j-1} \mathbb{P}\left(|g_{j,k}(\omega)| > 2^{j/4} \right) \\ &\leq 2^j \exp\left(-2^{j/2-1} \right). \end{aligned} \tag{7}$$

Let us define:

$$\forall j \in \mathbb{N}, \;\; \mathcal{A}_j = \left\{ \sup_{0\leq k\leq 2^j-1} |g_{j,k}(\omega)| > 2^{j/4} \right\}.$$

Let us state the Borel-Cantelli lemma:

Lemma 3.2 (Borel-Cantelli). *Let A_n be a sequence of events in $(\Omega, \mathcal{F}, \mathbb{P})$ such that:*

$$\sum_{n=1}^{\infty} \mathbb{P}(A_n) < \infty.$$

Then:

$$\mathbb{P}\left(\limsup_{n\to\infty} A_n\right) = 0,$$

where the operator lim sup *is defined in definition 3.6.*

Proof 3.8 (Borel-Cantelli). *We admit this theorem for our article, but see [10] for a proof.* ⊡

Definition 3.6 (lim sup operator). *Let X be a set and let $\{X_n\}_{n\in N}$ be a sequence of subsets of X. The* lim sup *of the sequence is defined as:*

$$\begin{aligned}\limsup_{n\to\infty} X_n &= \inf_{n\in\mathbb{N}}\left\{\sup_{m>=n}\{X_m\}\right\} \\ &= \bigcap_{n=1}^{+\infty}\left(\bigcup_{m=n}^{+\infty} X_m\right).\end{aligned}$$

The hypothesis for the Borel-Cantelli lemma is immediately verified by equation (7)*:*

$$\sum_{j=1}^{+\infty} \mathbb{P}\left(\mathcal{A}_j\right) < +\infty.$$

Hence, from the Borel-Cantelli lemma, we have:

$$\mathbb{P}\left(\limsup_{j\to+\infty} \mathcal{A}_j\right) = 0.$$

Take $\mathcal{A} = \limsup_{j\to\infty} \mathcal{A}_j$. Then $\mathbb{P}(\mathcal{A}) = 0$.

Using the set $\mathcal{A}$ built before, two cases appear.

(a) If $\omega \in \mathcal{A}$, convergency of the series does not matter (because $\mathbb{P}(\mathcal{A}) = 0$).

(b) If $\omega \notin \mathcal{A}$, for j high enough, we have:

$$\sup_{0\le k\le 2^j-1} |g_{j,k}(\omega)| \le 2^{j/4},$$

hence:

$$\sup_{t\in[0,1]}\left|\sum_{k=0}^{2^j-1} g_{j,k}(\omega)\tau_{j,k}(t)\right| \le 2^{-n/4},$$

and the series converges.

Thus, the convergence is almost surely uniform over $[0,1]$.

Moreover, $\forall j \in \mathbb{N}$ *and* $\forall k \in [\![0, 2^j - 1]\!]$, $\tau_{j,k}$ *functions are continuous over* $[0,1]$. *By the uniform convergence theorem, the series* $\sum_{j=0}^{+\infty} \sum_{k=1}^{2^j-1} g_{j,k}(\omega)\tau_{j,k}(t)$ *is continuous over* $[0,1]$.

From there, the continuity of the Faber-Schauder representation for $t \in [0,1]$ *ensues, for all* $\omega \in \Omega$.

Hence, the Faber-Schauder formula does defines a Brownian motion on $[0,1]$. ⊡

Remark 3.4. *The continuity of* $(\tau_{n,k})_{(n,k)\in\mathbb{N}\times[\![0,2^n-1]\!]}$ *functions (hat functions described in 3.3) is essential in order to obtain the continuity of the Brownian motion, as it has been used in the proof. We can deduce from this that the decomposition must be done along a basis made of continuous vectors (functions).*

3.4.2 Optimality

In this section, we prove that the Brownian motion defined by the Faber-Schauder representation is optimal, meaning that the tail of the series tends to zero as fast as possible. To do so, we want to evaluate the quantity:

$$l_N(X) = \inf\left\{ \left(\mathbb{E}\left\| \sum_{k=N+1}^{\infty} \sum_{k=0}^{2^j-1} g_{j,k}(\omega)\tau_{j,k} \right\|^2 \right)^{1/2} \;;\; X = \sum_{j=0}^{\infty} \sum_{k=0}^{2^j-1} g_{j,k}(\omega)\tau_{j,k} \right\}.$$

It has been proven by Maiorov and Wasilkowski in the theory of so-called average linear widths that for the Brownian motion B we have:

$$l_N(B) \approx \left(\left(2^{N+1}\right)^{-1} \log\left(2^{N+1}\right) \right)^{1/2}.$$

Proposition 3.5 (Optimality of the Faber-Schauder representation)**.** *The Brownian motion defined by the Faber-Schauder representation is optimal.*

To prove (3.5), we need the two folowing lemmas.

Lemma 3.3. *There is a constant* $c > 0$ *such that for any integer* $N \geq 1$ *and for each centered Gaussian real-valued sequence* $\tau_1, \cdots, \tau_N$ *one has:*

$$\mathbb{E}\left(\sup_{1\leq k\leq N} |g_k| \right) \leq c\,(1+\log N)^{1/2} \sup_{1\leq k\leq N} \left(\mathbb{E}|g_k|^2\right)^{1/2}.$$

Lemma 3.4. *For any* $t \in [0,1]$ *and for each integer* $j \geq 0$, *there is at most one integer* k, *such that* $0 \leq k < 2^j$ *and* $\tau(2^j - k) \neq 0$.

Proof 3.9 (Optimality of the Faber-Schauder representation)**.** *With lemma 3.4 we have, for all* $\omega \in \Omega$, *for all* $j \geq 0$ *and for* $t \in [0,1]$*:*

$$\sum_{k=0}^{2^j-1} |g_{j,k}(\omega)||\tau(2^j t - k)| \leq \left(\sup_{0\leq k\leq 2^j-1} \right) \|\tau\|_\infty.$$

And, for all $N \in \mathbb{N}$, with lemma 3.3 we have:

$$\begin{aligned}
& \mathbb{E}\left(\sup_{t\in[0,1]} |B(t) - g_0(\omega)t - \sum_{j=0}^{N}\sum_{k=1}^{2^j-1} 2^{-j/2} g_{j,k}(\omega)\tau(2^j t - k)|\right) \\
\leq\ & \mathbb{E}\left(\sup_{t\in[0,1]} |\sum_{j=N+1}^{N}\sum_{k=1}^{2^j-1} 2^{-j/2} g_{j,k}(\omega)\tau(2^j t - k)|\right) \\
\leq\ & c_1 \sum_{j=N+1}^{\infty} 2^{-j/2}\mathbb{E}\left(\sup_{0\leq k\leq 2^j} |g_{j,k}(\omega)|\right) \\
\leq\ & c_2 \sum_{j=N+1}^{\infty} 2^{-j/2}(1+j)^{1/2}\left(\sup_{0\leq k\leq 2^j} \mathbb{E}|g_{j,k}(\omega)|^2\right) \\
=\ & c_2 \sum_{j=N+1}^{\infty} 2^{-j/2}(1+j)^{1/2} \\
\leq\ & c_3 2^{-N/2}(1+N)^{1/2} \\
=\ & c_3 (2^N)^{-1/2}(1+log(2^N))^{1/2} \\
\leq\ & c_3 \left(\frac{log(2^{N+1})}{2^{N+1}}\right)^{1/2} \\
\approx\ & \mathcal{O}\left(((2^{N+1})^{-1} log(2^{N+1})^{1/2}\right) \\
\approx\ & l_N(B).
\end{aligned}$$

That prove the Faber-Schauder representation is optimal for the Brownian motion. ⊡

3.4.3 Other Advantages of such a Representation

First, triangle functions $(\tau_{n,k})_{(n,k)\in\mathbb{N}\times[\![0,2^n-1]\!]}$ can be written as a scalar product between $\mathbb{1}_{[0,t]}$ and Haar functions, and Haar function are "located" perfectly in time (not in frequency, but that is less important).

A good precision in frequency is not needed since the amplitude is not important to reconstruct the Brownian motion, the error in frequency is "offset" by the random part of the separation.

Another advantage of such a representation is that Schauder wavelets have compact support, making computations and beforementioned proofs fast.

3.5 Scaled process

Proposition 3.6. *Let B_t be a Brownian motion for $t \in [0,1]$. The following stochastic process:*

$$\sqrt{c}\, B_{\left(\frac{t}{c}\right)}$$

is a Brownian motion for $t \in [0,c]$.

This result is given with B_t a Brownian motion for $t \in [0,1]$, and thus does not depend on the chosen representation. In particular, it works with the Faber-Schauder representation, and the proof is almost immediate: follow the pattern given in section 3.4.1 with the aforementioned

process, and the result ensues.

The discretization of the scaled process is detailed in section 4.5.

4 Discretization and Implementation

4.1 Discretization of the formula

The Brownian motion can be written with the Faber-Schauder formula given in (2).

As said in remark 3.2, Antoine Ayache showed in [1] that this decomposition of the Brownian motion is optimal, meaning that the tail of the series is going to zero as fast as possible.
This property (which we do not demonstrate) incite us to use this decomposition to implement the Brownian motion.

Hence, for a viable distretization, write the Brownian motion following the Faber-Schauder formula (2), and instead of taking the double sum all the way to infinity, take only a finite number of element N from the sum. Thus, leaving us with:

$$\forall t \in [0,1], \forall \omega \in \Omega, \quad B_t(\omega) \approx g_0(\omega)t + \sum_{j=0}^{N} \sum_{k=1}^{2^j-1} g_{j,k}(\omega)\tau_{j,k}(t), \tag{8}$$

This enable to implement the formula, which should give a good approximation of the Brownian motion.

The continuity of the Brownian motion is preserved with this representation, meaning that the simulation can be refined efficiently only by taking more terms in the discretized interval.

4.2 Implementation in Python Language

Firstly, implement the 1-dimensional Brownian motion, using the approximation given above in (8). Note that the full Python code is attached in section 7.2.

The primary function, `Brownian`, returns the Brownian motion in each point of the discretization of the interval $[0, 1]$. This function uses another function, which is detailed below.

Our main script is designed to use this function, but also to perform some tests, which are needed as we develop below. Instructions on how to use it are all written as commentaries in the code itself.

4.2.1 The function `Triangle_Generator`

The function `Triangle_Generator` takes 3 parameters:

1. `n` corresponds to the index j for the function $\tau_{j,k}$ in the formula (8),
2. `k` corresponds to the index k for the function $\tau_{j,k}$ in the formula (8),

3. `nbPoint` corresponds to the number of points we want for the discretization of the interval $[0, 1]$.

As its name says it all, it generates an approximation of $\tau_{j,k}$ over the discretized interval.

4.2.2 The function `Brownian`

The function `Brownian` takes only one parameter: `nbPoint`, which is the number of points wanted for the discretization of the interval $[0, 1]$.

In this function, an important variable is `N`. It corresponds to the truncation of the sum we talked about earlier (see section 4.1). It can be set at any value the user finds acceptable. Note that the higher it is, the more precise the approximation will be. Also, the higher it is, the longer computation time will be.

For information, in this approximation, one is adding around 2^N terms and calling each time the function `Triangle_Generator`. Note that the calculation complexity of the realisations of the $g_{j,k}$ are as reduced as possible: the $(N + 1)(2^N - 1)$ realisations of the Gaussian random variables $g_{j,k}$ are "asked" only one time by the function and are stored in a $(N+1)\times(2^N-1)$ vector.

We chose to set $N = 15$ for the continuation. With this choice, the infinite norm of the last triangle function $\tau_{n,k}$ used in the truncated sum is worth only 0.015, which is found to be acceptable.

4.2.3 Higher-dimension Brownian motions

The 2-dimensional Brownian motion is composed of two independent standard Brownian motions, one along the abscissa, and one along the ordinate.

And the same is true for higher-dimension Brownian motions. Hence, one can use our function as multiple times as needed in order to achieve multi-dimensional Brownian motions.

In particular, we implemented a function for the 2-dimensional Brownian motion (`Brownian2D()`, see 7.2).

4.3 Precision

Usually when using an approximation, one studies the error made out of this approximation. Since we are working with a random process we cannot do that the classic way.

However, we are able to plot the quantity of information that every triangle function is bringing.

To do so we take the infinite norm of the discretized vector. We can see in next figure that the infinite norm is higher for small values of n and k.

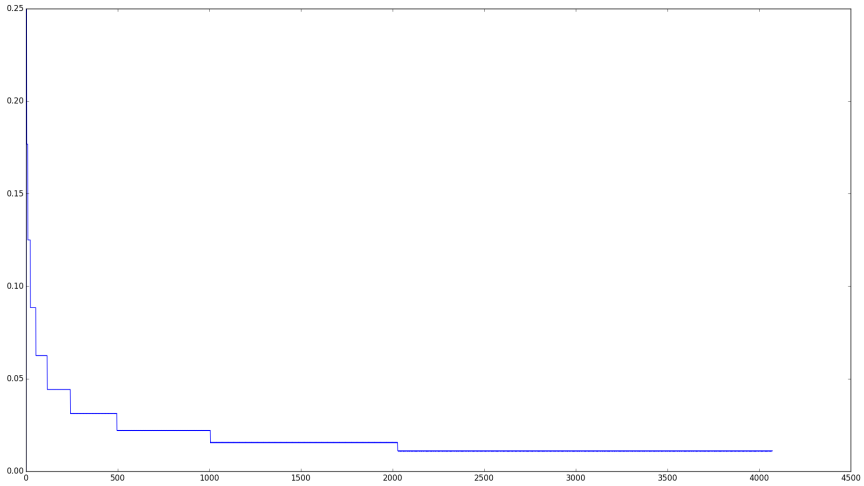


Figure 3 – $\|\tau_{n,k}\|_\infty$

The abscissa corresponds to the index `j` of the triangle function, given by:

```
j=0
for n in range(0, N):
    for k in range(1, 2**n-1):
        j=j+1
        # Compute the error of the triangle function number j.
```

Notice that it is coherent to index the error in that way, because in the Brownian function we compute the N first triangle functions following this method (where $N \in \mathbb{N}$).

Remark 4.1. *It seems like the infinity norm is equal to 0 as we take n and k big enough. This result is caused by the lack of points to discretize our interval. However, if we increase this number of points we would still have a decreasing norm.*

With $N = 15$, the infinite norm of the last triangle $\tau_{n,k}$ of the truncated sum worth 0.015. That means the truncated term is a sum of Gaussian times a number smaller than 0.015. This is satisfactory regarding to the first value of $\|\tau_{n,k}\|_\infty$ which are quite high in comparison.

4.4 Illustrations

We give in this section illustrations of our implementation of the Brownian motion following the Faber-Schauder formula, as described earlier. We set $t \in [0, 1]$ for all this illustations: the only variable that changes is `nbPoint`.

Figure 4 – Example of 6 1-d Brownian motions for `nbPoint` = 1000

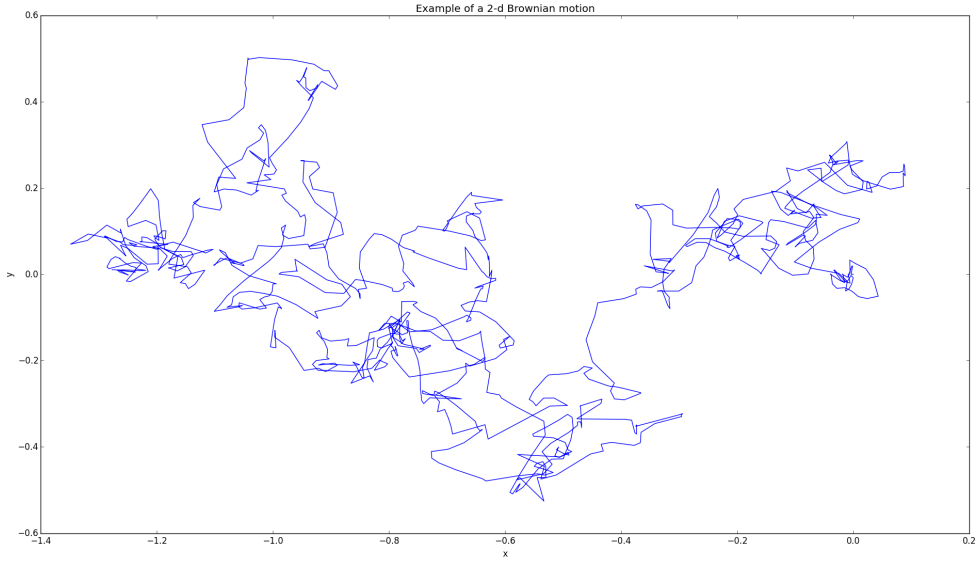
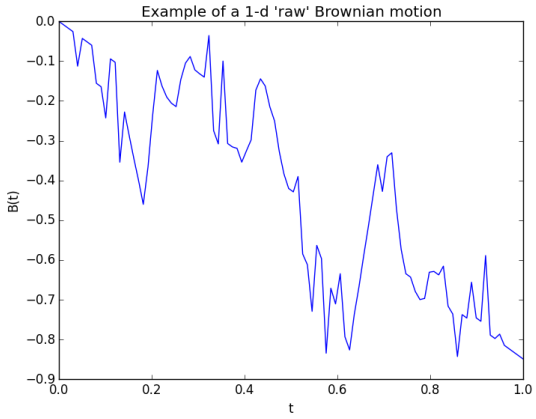
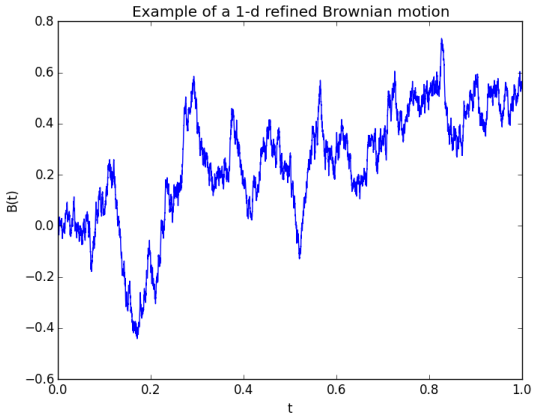


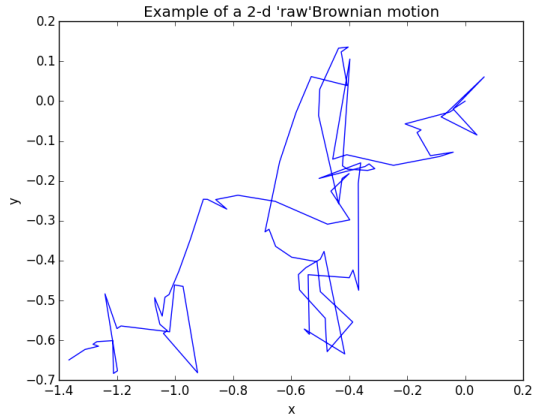
Figure 5 – Example of a 2-d Brownian motion for `nbPoint` = 1000



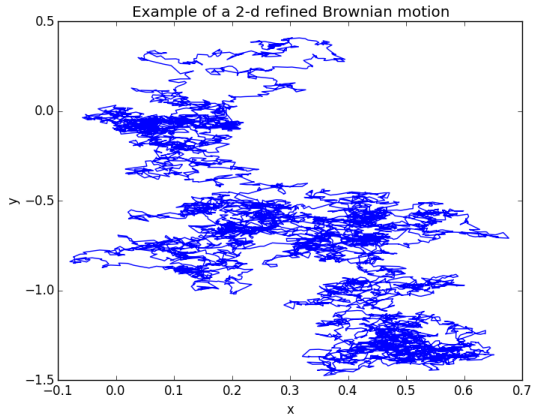
(a) 1-d Brownian motion, `nbPoint` = 100



(b) 1-d Brownian motion, `nbPoint` = 10000



(c) 2-d Brownian motion, `nbPoint` = 100



(d) 2-d Brownian motion, `nbPoint` = 10000

Figure 6 – Refinement of the Brownian motion

As the number of points requested increases, the Brownian motion becomes less abrupt. However, this refinement requires a longer computation time. Recall what is said in section 3.4.3: the continuity of the Brownian motion is preserved with this reprensentation, meaning that the simulation can be refined efficiently only by increasing the `nbPoints` variable.

4.5 Scaled Process

Suppose one wants to simulate a Brownian motion over $t \in [0, c]$. Let us use the formula given in section 3.5. Our algorithm does not rely on the chosen interval, but rather on a number of points.

Thus, to get a Brownian motion on the wanted interval, it is sufficient to choose a number of points over the interval, compute a Brownian motion with the algorithm and multiply the result by the square root of the length of the interval.

A possible algorithm using our functions may be (interval given by $[t_{\min}, t_{\max}]$):

```
nbPoint=1000
t=np.linspace(t_min, t_max, nbPoint)
B=np.sqrt(t_max-t_min)*Brownian(nbPoint)
```

5 Conclusions

We started by observing that the construction of the Brownian motion in the classic way, *ie* with the random walk, is using a limit of random variables that are not "sorted", which is an issue to implement.

Then, the Karhunen-Loève theorem gives another way to write a random process, using an orthonormal basis and centered orthogonal random variables.

Furthermore, the Faber-Schauder expansion gives us a series which is sorted, meaning that when truncating the series, the tail of the series is as small as wanted.

Moreover, the Faber-Schauder system is also an equivalent way to write the Brownian motion, and we use this representation to implement an algorithm which return a random process of the Brownian motion over $[0, 1]$.

With an algorithm returning a approximation of one Brownian motion, we are able to plot a Brownian motion in any dimension.

With a relatively low number of iterations, we are able to return a random process very close to a veritable Brownian motion, even if we cannot plot the error since we are working in a probability set.

Finally, with our algorithm, it is possible to plot a Brownian motion over any interval $[a, b]$ by scaling the process (see sections 3.5 and 4.5).

5.1 Limitations of the Approximation

Recall that this article concerns the Wiener process, *ie* the standard Brownian motion, which increments follow a specific distribution (see definition 2.6).

For now, it is not clear if a parameterization of our implementation of the Brownian motion over mean and variance is possible. Further work and demonstrations are needed for this.

5.2 Comparison with existing Brownian Motion Approximations

When searching for it, most approximations of the Brownian motion are base on methods based on the random walk formula over small steps dt.

For example, the SciPy Cookbook [11] proposes an implementation using the cumulative sum of random samples.

On one hand, this type of implementation raises the aforementioned problems (see section 2.3), but can be a good compromise for a basic use.

On the other hand, if one wants to be able to refine a Brownian motion or wants to get one precise enough over a long interval, our implementation will give better results.

5.3 Possible Ameliorations

The performance of the algorithm might be improved. As seen in section 4.2.2, the 2^N calls to the `Triangle_Generator` function is doubtlessly slowing down the determination of the approximation.

Also, a good amelioration to the overall performance of the algorithm would be to let the user choose the value of N according to his needs, although this amelioration would require further tests on precision.

Finally, d-dimension Brownian motion calculation might be improved by designing dedicated functions rather than calling d times our basic `Brownian` function.

6 Java Toolbox

We created a Java toolbox capable of generating efficiently multidimensionnal Brownian motions over any type of index interval.

The aim of this section is to show the most efficient way to implement the approximation of the Brownian motion, given in (8).

Let us begin by stating the general principle (in section 6.1). Then, the degrees of freedom of such Brownian implementations will be detailed (in section 6.2), as well as their influence on the base implementaiton. Finally, the full code and explanations will be given (in section 7.1).

6.1 Principle

Let d be the dimension of B, the desired Brownian motion ($d \in \mathbb{N}^*$). Each component B_i of B is an independant Brownian motion. The one dimensionnal expression of the approximation, given in (8), can be generalized:

$$\forall t \in [0,1], \forall i \in [\![1,d]\!], \forall \omega \in \Omega, \;\; B_{i,t}(\omega) \approx g_{i,0}(\omega)t + \sum_{j=0}^{N} \sum_{k=1}^{2^j-1} g_{i,j,k}(\omega)\tau_{j,k}(t),$$

that means:

$$\forall t \in [0,1], \forall \omega \in \Omega, \;\; B_t(\omega) \approx G_0(\omega)t + \sum_{j=0}^{N} \sum_{k=1}^{2^j-1} G_{j,k}(\omega)\tau_{j,k}(t), \tag{9}$$

where G_0 and $(G_{j,k})_{(n,k)\in\mathbb{N}\times[\![0,2^n-1]\!]}$ are Gaussian vectors of dimension d (and $(\tau_{n,k})_{(n,k)\in\mathbb{N}\times[\![0,2^n-1]\!]}$ the non-normalized Faber-Schauder functions).

Hence, we create two Java objects, `Brownian` and `SchauderWavelet`.

`SchauderWavelet`:
It implements the non-normalised Faber-Schauder system and is built to follow the ordering needed in formula (9). It has attributes encoding the wavelet needed, starting when initialised at the wavelet s_1 (s_0 being useless in our implementation of the Brownian motion). It also has

a method allowing any user to "increment" the wavelet, *ie* transforming the current one into the following one, that is:

	current wavelet	following wavelet
	s_1	$\tau_{0,0}$
$\forall j \in \mathbb{N}, \forall k \in [\![0, 2^j - 2]\!]$	$\tau_{j,k}$	$\tau_{j,k+1}$
$\forall j \in \mathbb{N}, k = 2^j - 1$	$\tau_{j,k}$	$\tau_{j+1,0}$

With the definition of every wavelet of the non-normalized Faber-Schauder system, this class is able to return their starting point, the length of their support and their values on their support.

All of this is being discretised with a given number of points over the $[0, 1]$ interval (which will be K_t, when used with the `Brownian` class). See section 6.2.2 for further details on the discretisation implementation.

`Brownian`:

Its main attribute, alongside with others (see 7.1.2), is a double index table: the first dimension being d (the dimension of the Brownian motion) and the second being K_t (the wanted number of points for the discretisation of the interval).

Using the formula (9) implies the following scheme of implementation:

Listing 1 – Brownian motion approximation pseudo-code.

```
# @param d the dimension of the wanted Brownian motion
# @param N the cut parameter of the approximation
w = # initialise a SchauderWavelet object
while(not([stopping criterion])){
  tmp  = # Get values of the wavelet over its support.
  wSt  = # Get start index of the support.
  wEnd = # Get end index of the support.
  g    = # Get Gaussian vector of dimension d.
  for(int i=wSt; i<=wEnd; i++){
    # Run through the discretised support of the wavelet (over time index
    # discretisation).
    for(int j=0; j<d; j++){
      # At each time, run through coordinates.
      B[j][i]+=g[j]*tmp[i-wSt];
    }
  }
  # increment the wavelet
}
w.reinitialize();
```

Keep in mind that this implementation is already being considered over a discretised interval and that it is designed to minimize the number of calls to the generator of $(\tau_{n,k})_{(n,k)\in\mathbb{N}\times[\![0,2^n-1]\!]}$ functions.

Moreover, note that the stopping criterion does rely both on a cut parameter (N) and a condition on the wavelet. Indeed, once the width of the support of the wavelet is too small compared to the wanted discretisation (typically, when the length is under 3 indexes), it is no use to try to iterate more.

Also, that implies that using a cutting parameter N is useless when $K_t < 2^N + 1$, as the decreasing size of the wavelets supports will reach the critical value sooner than N.

6.2 Degrees of Freedom of the Algorithm

The degrees of freedom of the algorithm are:

- K_t, number of discretisation points for the time interval (mandatory),
- d, dimension of the wanted Brownian motion (optional, default at 1),
- c, length of the wanted Brownian (optional, default at 1),
- N, cut parameter (optional, default at 15).

The K_t parameter is the most important parameter and will be choosed by the user. We propose two implementations, depending on the form of K_t. See section 6.2.2 for further details on the discretisation implementation.

We design a complete constructor that accepts all degrees of freedom as parameters, and simpler constructors for more basic needs.

The most basic constructor only needs a number of discretisation points and will return an approximation of the one-dimensionnal Brownian motion over the $[0, 1]$ interval, and cut at $N = 15$. The next one adds a parametrisation of the dimension.

Then, one adds the possibility to scale the Brownian motion with a parameter c (the length of the wanted time interval, see 6.2.1), and cuts at $N = 15$. A different one adds the possibility to cut at a certain N, but only on an interval of length 1.

The last one is the complete one.

6.2.1 Scaling

Suppose one wants to simulate a Brownian motion over $[t_{\min}, t_{\max}]$. Let us use the formula given in section 3.5.

Recall that it is important to note that the Brownian motion does not depend on its starting point, but the length of the time interval has an impact on it.

Our base algorithm does not rely on the chosen interval, but rather on a number of points. We create a function to scale the wanted process that, after the fashion of the Brownian motion, only depends on the length of the wanted interval.

Thus, to get a Brownian motion on the wanted interval, it is sufficient to choose a number of points over the interval, compute a Brownian motion with the algorithm and multiply the result by the square root of the length of the interval. Its header is:

Listing 2 – `scale` function header.

```
private void scale(double c)
```

Its code, in the `Brownian` class, implements the principle described aforementionned.

6.2.2 The Number of Points in the Discretisation

We distinguish two cases:

- Case 1: $\exists p \in \mathbb{N}$ such that $K_t = 2^p + 1$,
- Case 2: and all other cases.

Case 1 ($\exists p \in \mathbb{N} \mid K_t = 2^p + 1$):
We remark that, for all (j,k) indexes, the start point, the middle point, and the end point of the $\tau_{j,k}$ function "drops" exactly on an index of the table enconding the discretised interval.

Let us denote `tab` this table. Recalling the description of the $(\tau_{n,k})_{(n,k)\in\mathbb{N}\times[\![0,2^n-1]\!]}$ given in remark 3.3, that means:

$$\exists (i_s, i_m, i_e) \in [\![0, K_t - 1]\!]^3 \text{ such that } \begin{cases} \texttt{tab}[i_s] &= \dfrac{2k}{2^{n+1}}, \\ \texttt{tab}[i_m] &= \dfrac{2k+1}{2^{n+1}}, \\ \texttt{tab}[i_e] &= \dfrac{2k+2}{2^{n+1}}. \end{cases}$$

This peculiar but not very surprising fact allows us to encode explicitely the method returning the values of the wavelet on its support.

The code of the function `getValuesPowerOf2()` in the `SchauderWavelet` class illustrates by itself this implementation.

Case 2 (general case):
When this condition is not verified, we use as strategy a sampling of all the non-normalised Faber-Schauder functions over the discretised interval. That means more computations, but allows a more flexible parametrisation of the decomposition.

The code of the function `getValuesGeneral()` in the `SchauderWavelet` class implements this sampling, using diverse utilitary functions in the `Util` class.

Final strategy:
When initialised, the constructor of `SchauderWavelet` objects will detect in which case the used K_t parameter is, and save it.

Then, the `Brownian` class access the public `getValues()` method, which will use either `getValuesGeneral()` or `getValuesPowerOf2()` depending on the mode (these are private methods).

The Brownian motion builder will also use public methods `getStart()` and `getEnd()` to get respectively start and end indexes of the wavelet support. These are not dependent on the mode (the case).

6.3 Usage

This Java Toolbox is packed in a .jar archive.

6.3.1 Package Usage

The source classes can be copied besides any other source code, and compiled alongside them. This enables the user to call the methods directly in his program.

6.3.2 Command Line Usage

Java is needed in order to execute the .jar directly, the command line format being the following:

```
java -jar BrownianGenerator.jar kt=... [d=...] [c=...] [n=...] [start=...]
   [output=...]
```

Where option names are directly linked to the degrees of freedom detailed before (see 6.2). In section 6.3.3 are further explanations on the options possibilities.

`kt`, `d`, `c`, `n` and `start` options are pretty self-explanatory.

`output` allows the user to write the generated Brownian motion in a text file.
If this option is not specified, the Brownian motion will be displayed on the standard output, *ie* the console or the terminal.

If the user has specified an output file, the program will write all points of the Brownian motion in it, as well as information on it (K_t, d, c, N and its starting point) in order to be retrievable by anyone able to parse this file.
All characters will be encoded in UTF-8, and coordinates written in scientific format with 16 decimals. For example, a file containing a default 1000-point 2-D Brownian motion will start like so:

```
kt=1000; d=2; c=1.0; n=15; startPoint=[0.0, 0.0]
 0.0000000000000000e+00  0.0000000000000000e+00
 3.8088021391987150e-02 -2.9328389173551024e-02
 7.5191443043892860e-02 -5.5503525835590165e-02
 9.3114861759012070e-02 -2.0253303573377730e-02
```

6.3.3 Command Line Option Guide

Option	Mandatory ?	Values
`kt`	yes	integers
`d`	no	integers
`c`	no	doubles
`n`	no	integers
`start`	no	table of doubles
`output`	no	any filename

Examples of usage:

```
java -jar BrownianGenerator.jar kt=10
java -jar BrownianGenerator.jar kt=10 d=2
java -jar BrownianGenerator.jar kt=10 c=14.5
java -jar BrownianGenerator.jar kt=10 n=5
java -jar BrownianGenerator.jar kt=10 start=[1]
java -jar BrownianGenerator.jar kt=10 d=2 start=[1,1]
java -jar BrownianGenerator.jar kt=10 start=[10.45]
java -jar BrownianGenerator.jar kt=10 d=3 start=[10.8,4.6,1.4e6]
```

```
java -jar BrownianGenerator.jar kt=10 d=2 output=test.txt
java -jar BrownianGenerator.jar kt=10 d=2 c=10 n=7 start=[0.0,1.47]
   output=brownian.txt
```

References

[1] Antoine Ayache, 2012, "Optimal series representation of continuous Gaussian random fields", http://math.univ-lille1.fr/~ayache/Vanderbilit-Univ-April2012.pdf

[2] Jean-Christophe Breton, 2013, "Processus stochastique", https://perso.univ-rennes1.fr/jean-christophe.breton/Fichiers/processus_M2.pdf

[3] Felipe Cucker and Ding Xuan Zhou, 2017, "Learning Theory: An Approximation Theory Viewpoint"

[4] Mark H. A. Davis, 2004, "Construction of Brownian Motion", http://wwwf.imperial.ac.uk/~mdavis/course_material/sp1/BROWNIAN_MOTION.PDF

[5] Jean Francois Le Gall, 2006, "Intégration, Probabilités et Processus Aléatoire", http://www2.fimfa.ens.fr/telecharger_fichier.php?fichier=117

[6] Monique Jeanblanc, 2007, "Mouvement Brownien", http://www.maths.univ-evry.fr/pages_perso/jeanblanc/cours/M2IF_1.pdf

[7] Sham Kakade and Greg Shakhnarovich, 2009, "Dimensionality Reduction", http://ttic.uchicago.edu/~gregory/courses/LargeScaleLearning/lectures/kl_pca.pdf

[8] María Cristina Pereyra & Lesley A. Ward, 2012, "Harmonic Analysis: From Fourier to Wavelets"

[9] Mark A. Pinsky, 2002, "Introduction to Fourier Analysis and Wavelets", https://books.google.fr/books?id=PyISCgAAQBAJ

[10] Proof Wiki, https://proofwiki.org/wiki/Borel-Cantelli_Lemma

[11] SciPy Cookbook, http://scipy-cookbook.readthedocs.org/items/BrownianMotion.html

[12] J. Michael Steele, 2001, "Stochastic Calculus and Financial Applications"

[13] Wei Zang, 2008, "Haar-based Multi-resolution Stochastic Processes", https://books.google.fr/books?id=9JC2rI7Pq7IC

7 Annexes

7.1 Java Code

The complete .jar archive, executable and containing the source files, is available at the following address:

http://etud.insa-toulouse.fr/~martire/files/2016_05_BrownianGenerator.jar

If you have any question concerning the code, or any bug to report, do not hesitate to send me a mail at:

leo.martire@outlook.com .

7.1.1 Generator

```
// Title       : Generator.java
// Description : This class implements the decomposition along
//               the non-normalised Faber-Schauder system
//               (often called the Lévy decomposition) of the
//               Brownian motion.
// Author      : Léo Martire.
// Date        : 2016.
// Notes       : None.

package BrownianGenerator;

import java.util.Arrays; // affichage de tableaux (à virer plus tard)

public class Generator{
  public static void main(String[] args){
    System.out.println(Arrays.toString(args));
  }
}
```

7.1.2 Brownian

```
// Title       : Brownian.java
// Description : This class implements the decomposition along
//               the non-normalised Faber-Schauder system
//               (often called the Lévy decomposition) of the
//               Brownian motion.
// Author      : Léo Martire.
// Date        : 2016.
// Notes       : None.

package BrownianGenerator;

import java.util.Random;
import java.util.Arrays; // affichage de tableaux (à virer plus tard)

public class Brownian{
  // Attributes //-------------------------------------------
  private int len; // number of time index discretisation points
  private int dim; // Brownian motion dimension
  private double[] start; // Brownian motion starting point
```

```
private double[][] BrownianPath; // Brownian motion values
//-------------------------------------------------------------

// Constructors //-----------------------------------------
public Brownian(int Kt){
  // Constructor : basic.
  this(Kt, 1, 1.0, 15);
}
public Brownian(int Kt, int dimension){
  // Constructor : default.
  this(Kt, dimension, 1.0, 15);
}
public Brownian(int Kt, int dimension, double c){
  // Constructor : scale.
  this(Kt, dimension, c, 15);
}
public Brownian(int Kt, int dimension, int N){
  // Constructor : cut.
  this(Kt, dimension, 1.0, N);
}
public Brownian(int Kt, int dimension, double c, int N){
  // Constructor : complete.
  // Errors :
  if(Kt<=0){
    System.err.println("Error : interval discretisation is impossible
       (Kt="+Kt+").");
    System.exit(-1);
  }
  if(c<=0){
    System.err.println("Error : dimension is incorrect
       (d="+dimension+").");
    System.exit(-1);
  }
  if(c<=0){
    System.err.println("Error : interval length is incorrect (c="+c+").");
    System.exit(-1);
  }
  if(c<=0){
    System.err.println("Error : cut index is incorrect (N="+N+").");
    System.exit(-1);
  }

  // Warnings :
  if(Kt<=10){
    System.err.println("Warning : interval discretisation is very small
       (Kt="+Kt+"), beware.");
  }
  if(c<=1e-12){
    System.err.println("Warning : interval length is very small
       (c="+c+"), numerical singularities may occur.");
```

```
    }
    if(N<=5){
      System.err.println("Warning : cut point is very small (N="+N+"),
         beware.");
    }

    // Construction :
    this.len=Kt;
    this.dim=dimension;
    this.start=new double[dimension];
    SchauderWavelet sw=new SchauderWavelet(this.len);
    this.BrownianPath=this.getBrownian(sw, this.dim, N);
    for(int i=0; i<this.dim; i++){ // ensure the starting point is 0
      this.start[i]=0;
    }
    this.scale(c);
  }
  //-------------------------------------------------------

  // Methods //-----------------------------------------------
  private void scale(double c){
    // Scales the [0, 1]-based Brownian motion.
    // @param c the scaling parameter
    // @return void.
    double coef=Math.sqrt(c);
    for(int i=0; i<this.len; i++){
      for(int j=0; j<this.dim; j++){
        this.BrownianPath[j][i]=coef*this.BrownianPath[j][i];
      }
    }
  }

  private double[][] getBrownian(SchauderWavelet w, int dimension, int N){
    // Generates an approximation of the Brownian motion using the
       non-normalised Faber-Schauder system.
    // @param w the SchauderWavelet object used in the approximation
    // @param dimension the dimension of the wanted Brownian motion
    // @param N the cut parameter of the approximation
    // @return a double-index table representing the wanted Brownian motion
       approximation
    double[][] B=new double[dimension][this.len];
    double[] tmp;
    double[] g;
    int wSt;
    while(!(w.hasToStop() || w.getOrder()>N)){
      tmp=w.getValues();
      g=getGaussians(dimension);
      wSt=w.getStart();
      for(int i=wSt; i<=w.getEnd(); i++){
        for(int j=0; j<dimension; j++){
```

```
          B[j][i]+=g[j]*tmp[i-wSt];
        }
      }
      //w.printStatus(); // Debugging : show all used wavelets when they
         are used.
      w.iterate();
    }
    w.reinitialize();
    return(B);
  }

  private double[] getGaussians(int number){
    // Builds a Gaussian vector.
    // @param number size of the wanted Gaussian vector
    // @return a Gaussian vector
    if(number<=0){
      System.err.println("Error : invalid number of Gaussians requested
         ("+number+" requested).");
      System.exit(-1);
    }
    double[] g=new double[number];
    Random r=new Random();
    for(int i=0; i<number; i++){
      g[i]=r.nextGaussian();
    }
    return(g);
  }

  public void changeStartPoint(double[] coordinates){
    // Changes the starting point of the Brownian motion.
    // @param coordinates a table of coordinates for the new starting point
    // @return void
    if(coordinates.length!=this.dim){
      System.err.println("Error : dimension of requested starting point is
         invalid (dimension : "+coordinates.length+", Brownian motion
         dimension : "+this.dim+").");
      System.exit(-1);
    }
    for(int i=0; i<this.len; i++){
      for(int j=0; j<this.dim; j++){
        this.BrownianPath[j][i]-=this.start[j]; // remove the previous
           starting point
        this.BrownianPath[j][i]+=coordinates[j]; // add the new starting
           point
      }
    }
    this.start=coordinates;
  }

  public double[] get(int t){
```

```
    // Returns the t-th coordinate.
    if(t<0 || t>=this.len){
      System.err.println("Error : index requested is invalid (t="+t+",
         Kt="+this.len+").");
      System.exit(-1);
    }
    double[] r=new double[this.dim];
    for(int i=0; i<this.dim; i++){
      r[i]=this.BrownianPath[t][i];
    }
    return(r);
  }
  //-------------------------------------------------------------

  // toString redefinition //---------------------------------
  public String toString(){
    // toString redefinition.
    String str="";
    for(int i=0; i<this.len; i++){
      str+="[";
      for(int j=0; j<this.dim; j++){
        str+=this.BrownianPath[j][i];
        if(j<this.dim-1){str+=", ";}
      }
      str+="]\n";
    }
    return(str);
  }
  //-------------------------------------------------------------
}
```

7.1.3 SchauderWavelet

```
// Title       : SchauderWavelet.java
// Description : This class implements a discretisation of the
//               functions in the non-normalized Faber-Schauder
//               system.
// Author      : Léo Martire.
// Date        : 2016.
// Notes       : None.

package BrownianGenerator;

import java.util.Arrays;

public class SchauderWavelet{
  // Attributes //--------------------------------------------
  private boolean s1; // encodes the use of the s_1 wavelet
  private boolean powerOf2; // encodes the mode
```

```
private int len; // number of [0, 1] interval discretisation points
private int j; // j index of the non-normalised Faber-Schauder function
private int k; // k index of the non-normalised Faber-Schauder function
private int width; // width of the wavelet support (only used in powerOf2
   mode)
private double step; // step of the discretisation
//-------------------------------------------------------------

// Constructors //-----------------------------------------
public SchauderWavelet(int Kt){
  this.len=Kt;
  this.step=1/(double)(this.len-1);
  if(Math.abs(Math.log(Kt-1)/Math.log(2)-(int)(Math.log(Kt-1)/Math.log(2)))==0){
     // checks if powerOf2 mode must be activated
    this.powerOf2=true;
  }
  else{
    this.powerOf2=false;
  }
  reinitialize();
}
//-------------------------------------------------------------

// Methods //---------------------------------------------
public void reinitialize(){
  // Reinitialises the wavelet.
  // @param void
  // @return void
  this.j=0;
  this.k=0;
  this.width=this.len; // initial width of the support (only used in
     powerOf2 mode)
  this.s1=true; // the first used wavelet is s_1
}

private void updateWidth(){
  // Calculates width of the support, based on the previous width (only
     used in powerOf2 mode).
  // @param void
  // @return void
  this.width=(int)((this.width-1)/2+1);
}

public void iterate(){
  // Iterates the wavelet. That means :
  //   If the current wavelet is s_1, make the boolean false ((j, k)
     indexes are already initialised).
  //   If k is between 0 and 2^j-1-1, add 1 to k and translate support by
     the width of it.
```

```
    //   If k=2^n-1, add 1 to j bring back k to 0 : start index gets to 0
       and height is updated.
    // @param void
    // @return void
    if(this.s1){
      this.s1=false;
    }
    else{
      if(this.k<Math.pow(2, this.j)-1){
        this.k+=1;
      }
      else{
        if(k==Math.pow(2, this.j)-1){
          this.k=0;
          this.j+=1;
          updateWidth(); // (only used in powerOf2 mode)
        }
        else{ // k >= 2^n
          System.err.println("Warning : bad wavelet iteration.");
        }
      }
    }
  }

  public boolean hasToStop(){
    // If the width of the support of the wavelet is too small compared to
       the wanted discretisation (typically, when the length is under 3
       indexes), it is no use to try to iterate more.
    // @param void
    // @return true if the width of the support is too small, false if not
    return(this.width<3);
  }

  public int getOrder(){
    // Returns the order, that means the j index (encoding the height of
       the wavelet).
    // @param void
    // @return j index
    return(this.j);
  }

  public double[] getValues(){
    // Returns the values of the current wavelet on its discretised
       support. Lengths of returned tables vary depending on the indexes.
       If the number of discretisation points is a power of two plus one,
       getValuesPowerOf2 is used. If not, getValuesGeneral is used.
    // @param void
    // @return the values of the wavelet on its current discretised support.
    if(this.powerOf2){
      return(getValuesPowerOf2());
```

```
    }
    else{
      return(getValuesGeneral());
    }
  }

  public int getStart(){
    return((int)Math.ceil(2*this.k/Math.pow(2, this.j+1)/this.step));
  }
  public int getEnd(){
    return((int)Math.floor(2*(this.k+1)/Math.pow(2, this.j+1)/this.step));
  }

  public double[] getValuesPowerOf2(){
    // Returns the values of the current wavelet on its discretised support
       when the number of discretisation points is a power of two plus one
       by using the exact values of the triangle slopes on the indexes.
    // @param void
    // @return the values of the wavelet on its current discretised support.
    double[] vals=new double[this.width];
    if(this.s1){
      for(int i=0; i<this.width; i++){
        vals[i]=(double)(i)/(this.width-1);
      }
    }
    else{
      double height=Math.pow(2, -1-(double)(this.j)/2);
      int indice_milieu=(int)((this.width-1)/2);
      for(int i=0; i<=indice_milieu; i++){ // montée
        vals[i]=i*height/indice_milieu;
      }
      for(int i=indice_milieu+1; i<this.width; i++){ // descente
        vals[i]=(this.width-1-i)*height/indice_milieu;
      }
    }
    return(vals);
  }

  public double[] getValuesGeneral(){
    // Returns the values of the current wavelet on its discretised support
       when the number of discretisation points is not a power of two plus
       one by sampling over the discretised interval of the two slopes of
       the triangle.
    // @param void
    // @return the values of the wavelet on its current discretised support.
    double[] vals;

    if(this.s1){
      vals=Util.sample(Util.fillWithStep(this.len, 0, this.step),
                       1.0,
```

```
                        0.0);
    }
    else{
      double[] up, down, coefs;
      int upperStart, lowerMid, upperMid, lowerEnd;
      upperStart=this.getStart(); // Upper integer value of start point,
         when projected on the discretised interval.
      lowerMid=(int)Math.floor((2*this.k+1)/(Math.pow(2,
         this.j+1)*this.step));  // Lower integer value of middle point,
         when projected on the discretised interval.
      upperMid=(int)Math.ceil((2*this.k+1)/(Math.pow(2,
         this.j+1)*this.step)); // Upper integer value of middle point,
         when projected on the discretised interval.
      lowerEnd=this.getEnd(); // Lower integer value of start point, when
         projected on the discretised interval.
      coefs=Util.getTriangleCoefs(this.j, this.k);
      up=Util.sample(Util.fillWithStep(lowerMid-upperStart+1,
                                       upperStart*this.step,
                                       this.step),
                     coefs[0],
                     coefs[1]);
      down=Util.sample(Util.fillWithStep(lowerEnd-upperMid+1,
                                         upperMid*this.step,
                                         this.step),
                       coefs[2],
                       coefs[3]);
      if(lowerMid!=upperMid){ // If the middle of the triangles drops
         between two indexes, just concatenate the two arrays.
        vals=Util.concat(up, down);
      }
      else{ // If the middle of the triangles drops exaclty on an index, do
         not duplicate the vertex' value.
        vals=Util.concat(up, Arrays.copyOfRange(down, 1, down.length));
      }
    }
    return(vals);
  }

  public void printStatus(){
    // Prints the status of the current wavelet, using the toString method.
    // @param void
    // @return void
    System.out.println(this);
  }
  //-------------------------------------------------------------

  // toString redefinition //---------------------------------
  public String toString(){
    return("\n> SchauderWavelet "+this.hashCode()+" (mode =
       "+(this.powerOf2?"powerOf2":"general")+") :"+
```

```
            "\n>> index"+(this.s1?"":"es")+" :
               "+(this.s1?"s1":"(j="+this.j+", k="+this.k+")")+","+
            "\n>> support (table indexes) : width "+this.width+", start
               "+this.getStart()+","+
            "\n>> height : "+(Math.pow(2, -1-(double)(this.j)/2)+","+
            "\n>> values : "+Arrays.toString(this.getValues())+".\n"));
  }
  //-------------------------------------------------------------
}
```

7.1.4 Util

```
// Title       : Util.java
// Description : This class contains useful static methods
//               regarding discretisations of affine functions.
// Author      : Léo Martire.
// Date        : 2016.
// Notes       : None.

package BrownianGenerator;

import java.util.Arrays;

public class Util{
  // Attributes //-------------------------------------------
  //-------------------------------------------------------------
  // Constructors //-----------------------------------------
  //-------------------------------------------------------------
  // Methods //---------------------------------------------
  public static double[] concat(double[] a, double[] b){
    // Concatenate two double arrays.
    // @param a the first array
    // @param b the second array
    // @return an array containing a and b at the end of a
    double[] c= new double[a.length+b.length];
    System.arraycopy(a, 0, c, 0, a.length);
    System.arraycopy(b, 0, c, a.length, b.length);
    return c;
  }

  public static double[] fillWithStep(int size, double startPoint, double
     stp){
    // Fills an array of given size with values starting at a certain one
       and incrementing by a fixed step.
    // @param size the wanted size
    // @param startPoint the first value of the array
    // @param stp the step by which to increment
    // @return an array of wanted size with values starting at the wanted
       one and incrementing by the given step
```

```
    double[] arr=new double[size];
    arr[0]=startPoint;
    for(int i=1; i<size; i++){
      arr[i]=arr[i-1]+stp;
    }
    return(arr);
  }

  public static double[] getTriangleCoefs(int j, int k){
    // Get the two coefficients encoding the two affine lines encoding the
       (j, k) non-normalised Faber-Schauder function.
    // @param j j index of the non-normalised Faber-Schauder function
    // @param k k index of the non-normalised Faber-Schauder function
    // @return [a1, b1, a2, b2] where the first slope of the (j, k)
       non-normalised Faber-Schauder function is given by (a1 * x + b1) and
       the second one by (a2 * x + b2)
    double[] r={Math.pow(2, (double)(j)/2),
                -k*Math.pow(2, -(double)(j)/2),
                -Math.pow(2, (double)(j)/2),
                (k+1)*Math.pow(2, -(double)(j)/2)};
    return(r);
  }

  public static double[] sample(double[] abscissas, double a, double b){
    // Samples over a given array of abcsissas an affine function given by
       its characteristic coefficients.
    // @param abscissas table containing the wanted abscissas
    // @param a slope of the affine function
    // @param b offset at x=0 of the affine function
    // @return a table containing (a * x + b) evaluated for all x in the
       abscissas table
    double[] r=new double[abscissas.length];
    for(int i=0; i<abscissas.length; i++){
      r[i]=a*abscissas[i]+b;
    }
    return(r);
  }
  //-------------------------------------------------------------
}
```

7.2 Full Python code

The Python code is not finalized as well as the Java Toolbox, but remains more practical when it comes to fiddling with our functions.

```
# -*- coding: utf-8 -*-
import numpy as np
import matplotlib.pyplot as plt
plt.close('all')
```

```
# Triangle functions generator ################################
def Triangle_Generator(n, k, nbPoint):
    # @param n index j of the desired triangle function
    #          tau_{j, k}
    # @param k index k of the desired triangle function
    #          tau_{j, k}
    # @param nbPoint number of points of the discretization
    # @return vect the approximation of the desired triangle
    #              function
    vect=np.zeros(nbPoint)
    if(n==0 and k==0):
        # Base triangle function: height 1, centered on 0.5, ##
        #                         width 0.5.
        debut=np.floor(0.25*nbPoint)
        fin=np.floor(0.75*nbPoint)
        milieu=np.floor((debut+fin)*0.5)
        vect[debut:milieu]=(np.arange(debut, milieu, 1)-debut)/(debut)
        vect[milieu]=1
        vect[milieu+1:fin+1]=((np.ones(fin-milieu)-np.arange(milieu+2,
           fin+2, 1))+fin)/(debut)
        return vect
    else:
        # Adjustement of length and centering #################
        debut=np.maximum(np.floor((2*k)/(2.**(n+1))*nbPoint), 1)
        fin=np.maximum(np.floor((2*k+2)/(2.**(n+1))*nbPoint), 1)
        milieu=np.floor((debut+fin)*0.5)
        vect[debut:milieu+1]=(np.arange(debut, milieu+1,
           1)-debut)/np.maximum((milieu-debut), 1)
        vect[milieu:fin]=((np.ones(fin-milieu)-np.arange(milieu+1, fin+1,
           1))+fin-1)/np.maximum((milieu-debut), 1)
        # Adjustement of height ###############################
        vect=vect*2**(-n*0.5-1)
        return vect
##############################################################

# 1-d Brownian motion #########################################
def Brownian(nbPoint):
    # @param nbPoint number of points of the discretization
    # @return B the approximation of the 1-d Brownian motion
    N=15 # sum troncature
    gauss=np.random.randn(N+1, 2**N-1)
    gauss_0=np.random.randn(1)
    temps=np.linspace(0, 1, nbPoint)
    B=np.zeros(nbPoint)
    dblSum=np.zeros(nbPoint)
    for n in range(0, N):
        for k in range(1, 2**n-1):
            dblSum=dblSum+Triangle_Generator(n, k, nbPoint)*gauss[n, k]
    B=temps*gauss_0+dblSum
```

```
    return B
#############################################################

# 2-d Brownian motion ######################################
def Brownian2D(nbPoint):
    # @param nbPoint number of points of the discretization
    # @return B the approximation of the 1-d Brownian motion
    B=np.zeros((nbPoint, 2))
    B[:, 0]=Brownian(nbPoint)
    B[:, 1]=Brownian(nbPoint)
    return B
#############################################################

# Main program #############################################

plt.close('all')
nbPoint=1000 # number of points of the discretization
k=6 # for case 2 : the number of different Brownian motions to plot

case=5
# 1: draw triangle functions examples
# 2: draw 1-d Brownian motions
# 3: draw a 2-d Brownian motion
# 4: draw the error (in infinite norm) comitted on the
#    approximation of the triangle functions
# 5: draw illustrations

if case==1:
    x=np.linspace(0, 1, nbPoint)
    f, (ax0, ax1, ax2, ax3)=plt.subplots(4, sharex=True, sharey=True)
    v=Triangle_Generator(2, 0, nbPoint)
    ax0.plot(x, v)
    ax0.set_title('n=2, k=0')
    v=Triangle_Generator(2, 1, nbPoint)
    ax1.plot(x, v, color='r')
    ax1.set_title('n=2, k=1')
    v=Triangle_Generator(4, 1, nbPoint)
    ax2.plot(x, v, color='g')
    ax2.set_title('n=4, k=1')
    v=Triangle_Generator(5, 1, nbPoint)
    ax3.plot(x, v, color='k')
    ax3.set_title('n=5, k=1')
    f.subplots_adjust(hspace=0.2)
    plt.setp([a.get_xticklabels() for a in f.axes[:-1]], visible=False)

if case==2:
    x=np.linspace(0, 1, nbPoint)
    for i in range(1, k+1):
        B=Brownian(nbPoint)
        plt.plot(x, B)
```

```
    #plt.title('Example of '+str(k)+' Brownian motions')
    plt.title('Example of a Brownian motion')
    plt.xlabel('t')
    plt.ylabel('B(t)')

if case==3:
    B=Brownian2D(nbPoint)
    plt.plot(B[:, 0], B[:, 1])
    plt.title('Example of a 2-d Brownian motion')
    plt.xlabel('x')
    plt.ylabel('y')

if case==4:
    N=15
    w=0
    norme_inf=np.zeros(((N+1)*2**N, 1))
    for n in range(0, N):
        for k in range(1, 2**n-1):
            w=w+1
            norme_inf[w]=np.max(Triangle_Generator(n, k, nbPoint))
            if norme_inf[w]<10**-8:
                break
    x=np.linspace(1, w, w)
    plt.plot(x, norme_inf[:w])

if case==5:
    k=6
    nbPoint=1000
    x=np.linspace(0, 1, nbPoint)
    plt.figure()
    for i in range(1, k+1):
        B=Brownian(nbPoint)
        plt.plot(x, B)
    plt.title('Example of '+str(k)+' Brownian motions')
    plt.xlabel('t')
    plt.ylabel('B(t)')

    plt.figure()
    B=Brownian2D(nbPoint)
    plt.plot(B[:, 0], B[:, 1])
    plt.title('Example of a 2-d Brownian motion')
    plt.xlabel('x')
    plt.ylabel('y')

    nbPoint=100
    x=np.linspace(0, 1, nbPoint)
    plt.figure()
    B=Brownian(nbPoint)
    plt.plot(x, B)
    plt.title('Example of a 1-d \'raw\' Brownian motion')
```

```
    plt.xlabel('t')
    plt.ylabel('B(t)')
    plt.figure()
    B=Brownian2D(nbPoint)
    plt.plot(B[:, 0], B[:, 1])
    plt.title('Example of a 2-d \'raw\'Brownian motion')
    plt.xlabel('x')
    plt.ylabel('y')

    nbPoint=10000
    x=np.linspace(0, 1, nbPoint)
    plt.figure()
    B=Brownian(nbPoint)
    plt.plot(x, B)
    plt.title('Example of a 1-d refined Brownian motion')
    plt.xlabel('t')
    plt.ylabel('B(t)')
    plt.figure()
    B=Brownian2D(nbPoint)
    plt.plot(B[:, 0], B[:, 1])
    plt.title('Example of a 2-d refined Brownian motion')
    plt.xlabel('x')
    plt.ylabel('y')

plt.show()
```