INSTITUTO
UNIVERSITÁRIO
DE LISBOA

UNIVERSIDADE
DE LISBOA

Exact Monte Carlo Sampling of Jump Diffusions

Luís Simão Almeida Ferreira

Master's in Mathematical Finance

Supervisor:
PhD in Finance, José Carlos Gonçalves Dias, Associate Professor with Aggregation,
ISCTE Business School

October, 2021

Department of Finance

Department of Mathematics

Exact Monte Carlo Sampling of Jump Diffusions

Luís Simão Almeida Ferreira

Master's in Mathematical Finance

Supervisor:
PhD in Finance, José Carlos Gonçalves Dias, Associate Professor with Aggregation,
ISCTE Business School

October, 2021

Dedicated to my parents and siblings, for all the support and for believing in me.

Acknowledgment

I would like to thank my supervisor, professor Doutor José Carlos Dias, for introducing me to the world of Monte Carlo simulation and for the discussions, availability and support provided during this undertaking.

Resumo

O principal objetivo desta tese é explorar os fundamentos teóricos relativos ao método proposto em [20] por Kay Giesecke e Dmitry Smelov e implementá-lo de modo a comparar a sua performance face a métodos mais tradicionais de elementos finitos, que geram amostras enviesadas. O método aplica-se a uma grande parte dos modelos definidos por um processo de difusão com saltos unidimensional, permitindo gerar simulações de Monte Carlo exatas de um esqueleto, tempos de paragem e outros funcionais do mesmo, com finalidades como a avaliação de path-dependent options, derivados de taxa de juro ou outros instrumentos financeiros.

Abstract

The main objective of this thesis is to explore the theoretical foundations of the exact method for sampling jump diffusions proposed in [20] by Kay Giesecke and Dmitry Smelov, and implement it in order to compare the performance of the algorithm for pricing purposes against more traditional finite element methods, which generate biased samples. The method applies to a large class of models defined by a one-dimensional jump diffusion process, allowing us to generate exact simulations of a skeleton, a hitting time and other functionals of it, used for purposes like path-dependent option or interest rate derivatives pricing.

Contents

CHAPTER 1

Introduction

Monte Carlo methods are a broad class of computational algorithms that rely on repeated sampling of random variables to obtain numerical results, and are mainly used in three problem classes: optimization, numerical integration and sampling from given probability distributions. In this thesis, we discuss the sampling problem for jump diffusions with applications to derivatives pricing, and implement the algorithm proposed by Kay Giesecke and Dmitry Smelov in [20].

1.1. Literature review

We present here historical notes and references to the relevant literature leading us to the analysis of [20].

The first landmark work in option pricing goes back to 1973, belonging to Fisher Black and Myron Scholes [7] and Robert Merton [32]. In these papers, they provided the first explicit equilibrium solution for simple puts and calls, with the underlying assumption that the stock price follows a Geometric Brownian Motion. Even though some of the assumptions are known not to hold, the Black-Scholes-Merton model opened a path to understanding more complex contingent claims and built a language and framework for option pricing theory and practice, which is still currently used.

Some of the shortfalls have since been addressed by a large class of extensions. For instance, the Constant Elasticity of Variance models - CEV for short -, which were introduced by Cox [13], try to incorporate the fact that the volatility tends to be inversely proportional to the stock price. An alternative approach was given by Heston [22], in 1993, with a stochastic volatility model, where the variance of the asset is driven by a square-root process. These new models exhibited desired features observed on the market, such as a 'volatility smile' across multiple strikes or volatility clustering and heteroscedasticity. Under these assumptions, the processes are almost-surely path-continuous, and extreme events are still unlikely. However, evidence shows shocks in prices do occur and that the tails of the distributions are heavier than the ones implied.

Thus, models were extended to include the possibility of jumps, under a larger class of Lévy processes. Carr and Linetsky [11] developed a Jump-to-Default Extended CEV model (or JDCEV), where the stock price follows a CEV process up to a stochastic default time with state-dependent intensity, in which the firm defaults. Other jump-diffusion approaches to individual or portfolio credit risk include those of Arnsdorf and Halperin [4], Ding et al. [14], Duffie and Singleton [18], among many others. We refer to [5] for theoretical notes on Lévy processes.

With the need for more accurate and faster results and the advance in complexity of both the option payoffs themselves and of the models used to price them, such that closed form solutions are not feasible anymore, numerical methods have become widely popular. For instance, American options (and early-exercise options in general), due to their implicit value, rarely lent themselves to analytical solutions. Unless the exercise frontier is determined beforehand, partial differential equation based approaches are efficient at dealing with this specification when the number of factors is low.

On the other hand, despite being a relatively recent field, the Monte Carlo approach has grown in popularity. Even though Monte Carlo methods can also be extended to provide estimators for early-exercise options, e.g. via least squares fitting [29], their strength lies in the handling of path-dependent payoffs and multi-factor models. For a classical introduction to the field, we refer to [21], whereas more recent developments can be found in [19].

The most commonly used techniques for simulating paths of Lévy processes following a pre-specified stochastic differential equation are Euler methods, which discretize the equation and sample for the increments, analogously to the case of deterministic ordinary differential equations. Nonetheless, these methods do not come without their drawbacks. For instance, while certain stochastic differential equations have almost-surely non-negative solutions, a naive discretization may return negative values, and if there is state dependency, the fixed time increments and non-linearity may lead to biased results.

Exact simulation methods have since been developed in order to provide unbiased estimates, especially for the case where estimation of the bias is not straightforward (if there are no analytical solutions, for example). However, most of these rely on special structures of the jump diffusions. The schemes of Beskos and Roberts [6], and Chen and Huang [12] provide exact samples of a skeleton between jump times when the diffusion has state dependent drift and volatility. For the cases where jump intensity is constant, Ruf and Scherer [35] consider the exact sampling of hitting times, and Broadie and Kaya [9] develop an alternative exact scheme for the two-dimensional Heston model with jumps in both price and volatility.

In general, as stated in [20], if the jump intensity is state dependent, then the jump times cannot be generated independently of the diffusion component. The acceptance/rejection scheme the authors provide, valid under nonrestrictive assumptions on the process and for a large class of expectations, efficiently generates unbiased samples. We will further discuss this method in this thesis.

1.2. Structure of the thesis

This thesis is structured as follows.

In section 2, we will introduce the needed notions to understand the approach shown in [20]. We begin by presenting the tools from Stochastic Calculus, starting from the Brownian Motion and up to jump processes and Lévy processes. We will also cover several Monte Carlo techniques in chapter 3, including sampling from distributions, sampling

jump processes and jump times, and Euler methods for stochastic differential equations needed for later comparison.

In chapter 4, we will analyse the conditions needed for the algorithm, building up from simpler circumstances until we are able to state it in full generality, while in chapter 5 we will deal with questions pertaining computational efficiency and the actual implementation of the algorithm.

Finally, in chapter 6 we will compare the performance of the exact method against more traditional Euler methods under distinct scenarios and for different assets, and present conclusions.

An appendix is also available at the end, containing Python implementations of most methods used.

CHAPTER 2

Stochastic calculus and risk neutral pricing

2.1. Stochastic processes

We begin by recalling definitions and tools regarding stochastic processes, which we will make most use of later on. Our setting is a probability space $(\Omega, \mathcal{F}, \mathbb{P})$, equipped with a filtration $\mathcal{F}_t$.

DEFINITION 2.1.1. *A standard Brownian motion is a stochastic process W_t such that:*

(1) $W_0 = 0$*;*

(2) The increments $W_t - W_s$, $W_{t'} - W_{s'}$, *for* $0 \leq s' < t' \leq s < t$, *are independent;*

(3) $W_t - W_s \sim \mathcal{N}(0, |t-s|)$.

Brownian motion has several other important properties; namely, it admits an almost-surely continuous version. It is also useful to note how it behaves by re-scaling by $\theta > 0$:

$$W_t \overset{d}{=} \theta W_{t/\theta^2}. \tag{2.1}$$

Throughout this thesis, we will also make extensive use of martingale theory, mainly due to the two results presented next.

DEFINITION 2.1.2. *A martingale (in continuous time) is an integrable stochastic process, adapted to the filtration $\mathcal{F}_t$, such that for each $s < t$*

$$\mathbb{E}[X_t | \mathcal{F}_s] = X_s. \tag{2.2}$$

PROPOSITION 2.1.1. *(Novikov's Condition) Assume X_t is a stochastic process adapted to the filtration $\mathcal{F}_t$. If the condition*

$$\mathbb{E}\left[\exp\left(\frac{1}{2} \int_0^T |X_t|^2 dt \right) \right] < +\infty, \tag{2.3}$$

holds, then the process

$$Z_t := \exp\left(\int_0^t X_s dW_s - \frac{1}{2} \int_0^t X_s^2 ds \right) \tag{2.4}$$

is a martingale under $\mathbb{P}$.

THEOREM 2.1.2. *(Girsanov) Let Z_t be the exponential defined in (2.4). If Z_t is a strictly positive martingale, then there exists a measure $\mathbb{Q}$, equivalent to $\mathbb{P}$, such that*

$$\frac{d\mathbb{Q}}{d\mathbb{P}} \Big| \mathcal{F}_t = Z_t. \tag{2.5}$$

Furthermore, the process

$$\tilde{W}_t = W_t - [W, W]_t \tag{2.6}$$

is a Brownian motion under $\mathbb{Q}$.

Novikov's condition is our main tool to show that the condition on Girsanov's theorem holds. Finally, we now recall a fundamental result in Stochastic Calculus, Itô's lemma.

THEOREM 2.1.3. *(Itô's Lemma) Suppose* X_t *is an Itô diffusion process satisfying*

$$dX_t = \mu_t dt + \sigma_t dW_t. \tag{2.7}$$

Then, for any twice-differentiable function $f(t,x)$ *and* $Y_t = f(t, X_t)$*, one has*

$$dY_t = \left(\frac{\partial f}{\partial t} + \mu_t \frac{\partial f}{\partial x} + \frac{\sigma_t^2}{2}\frac{\partial^2 f}{\partial^2 x}\right)dt + \sigma_t \frac{\partial f}{\partial x} dW_t. \tag{2.8}$$

One final notion worth remembering, of major importance in stochastic calculus, is that of *stopping times.*

DEFINITION 2.1.3. *(Stopping time) Let* τ *be a random variable in the filtered probability space* $(\Omega, (\mathcal{F}_t)_{t\in I}, \mathbb{P})$*. We say that* τ *is a stopping time, with respect to the filtration* $\mathcal{F}$*, if for any* $t \in I$ *we have that*

$$\{\tau \leq t\} \in \mathcal{F}_t. \tag{2.9}$$

This definition encompasses the idea of a random time where we stop the process, according to some rule. A nice related example is that of *hitting times* (which may not be stopping times, but this verifies in most cases), that track the instant a stochastic process hits a certain barrier, i.e.

$$\tau := \inf\{t > 0 | X_t = B\}. \tag{2.10}$$

2.2. Jump processes

In order to introduce the concept of jumps, we present the Poisson distribution, which can be seen as a discrete equivalent of the normal distribution, due to the properties of its generating function.

DEFINITION 2.2.1. *We say a random variable* X *has a Poisson distribution with parameter* $\lambda > 0$*,* $X \sim Poisson(\lambda)$*, if its probability mass function is given by*

$$f_X(k;\lambda) = \frac{\lambda^k e^{-\lambda}}{k!}, \tag{2.11}$$

and we have that

$$\lambda = \mathbb{E}[X] = Var(X). \tag{2.12}$$

DEFINITION 2.2.2. *(Poisson process) Let* $(T_i)_{i=1}^{+\infty}$ *be independent exponential random variables with parameter* λ*. Let* $T_n := t_1 + ... + t_n$*, with* $T_0 = 0$*, and define*

$$N_s = \max\{n : T_n < s\}. \tag{2.13}$$

N_s *is called a Poisson process.*

Note that no assumptions are made on the sum $T_n := t_1 + ... + t_n$. Most of the time, we will be assuming that N_t is *non-explosive*, that is, $T_\infty = +\infty$ almost surely. The

importance of Poisson processes, also called counting processes, stems from the following proposition:

PROPOSITION 2.2.1. *N_s is a Poisson process if, and only if,*

(1) $N_0 = 0$*;*
(2) $N_{t+s} - N_t \sim Poisson(\lambda s)$*;*
(3) N_t *has independent increments.*

Thus, a Poisson process can be thought of as a discrete analogue of the Brownian motion, in the sense that it is memory-less and the increments are independent. We will now define two extensions to this type of processes, which allow for vast generalization and will be referred to later.

DEFINITION 2.2.3. *Let $\lambda(t)$ be a deterministic integrable function. We say that N_t is a non-homogeneous Poisson process with local intensity function $\lambda(t)$ if*

(1) $N_0 = 0$*;*
(2) N_t *has independent increments;*
(3) $N_{t+s} - N_t \sim Poisson\left(\int_t^{t+s} \lambda(z)dz \right)$.

DEFINITION 2.2.4. *Let $\lambda_t \geq 0$ be an adapted stochastic process. We say that N_t is a doubly stochastic Poisson process if*

(1) $N_0 = 0$*;*
(2) N_t *has independent increments;*
(3) $N_t - N_s|(\lambda_z)_{z=s}^{z=t} \sim Poisson\left(\int_s^t \lambda_z dz \right)$.

Moreover,

$$\mathbb{P}(N_t - N_s = k) = \mathbb{E}\left[\frac{1}{k!} \left(\int_s^t \lambda_z dz \right)^k \exp\left(- \int_s^t \lambda_z dz \right) \right]. \tag{2.14}$$

DEFINITION 2.2.5. *(Compound Poisson Process) Let N_t be a Poisson process with parameter λ and $X_i \sim X$ be a sequence of i.i.d. random variables with parent distribution X. We say that a stochastic process J_t is a compound Poisson process (or Poisson jump process) with intensity λ and jump size distribution X if*

$$J_t = \sum_{n=1}^{N_t} X_n. \tag{2.15}$$

We will also consider more general compound Poisson processes, where the jump size variables X_n are not i.i.d. but sampled locally, for instance, depending on some function of another random process.

Another concept to have in mind is the *compensator process*

$$C_t = \int_0^t \lambda_s ds. \tag{2.16}$$

Let τ_1 be the first jump time of a doubly stochastic Poisson process as above, and $\mathcal{E}$ be an independent standard exponential random variable. Meyer [33] shows that through a

time change given by C_t, the process becomes a standard Poisson process. In turn, this implies that

$$\tau_1 \stackrel{d}{=} \inf\{t \geq 0 | C_t \geq \mathcal{E}\}, \tag{2.17}$$

a fact that will be useful for us later to run simulations.

2.3. Lévy processes

Next, we will recall some definitions and important results in Lévy process theory, in order to motivate our main object of study, jump diffusions. We will not delve in many technical details, explaining only how we can characterize this type of processes. A full exposition on Lévy processes can be found, for instance, in [36] and [5], where measure theoretic results are properly treated.

DEFINITION 2.3.1. *A stochastic process X_t is a Lévy process if*

(1) For any choice of $n \geq 1$, and $0 \leq t_0 < t_1 < ... < t_n$, the random variables X_{t_0}, $X_{t_1} - X_{t_0}$, ..., $X_{t_n} - X_{tn-1}$ are independent;
(2) $X_0 = 0$ a.s.;
(3) The distribution of $X_{t+s} - X_s$ does not depend on s;
(4) It is stochastically continuous[1];
(5) There is $\Omega_0 \in \mathcal{F}$ with $\mathbb{P}[\Omega_0] = 1$ such that, for every $\omega \in \Omega_0$, $X_t(\omega)$ is right-continuous and has left limits.

DEFINITION 2.3.2. *(Markov processes) An adapted continuous time stochastic process X_t is said to be Markov if, for any Borel set B and each $s < t$, we have that*

$$\mathbb{P}[X_t \in B | \mathcal{F}_s] = \mathbb{P}[X_t \in B | X_s]. \tag{2.18}$$

DEFINITION 2.3.3. *(Strong Markov property) For a stopping time τ, define*

$$\mathcal{F}_\tau := \{A \in \mathcal{F} | \forall t \geq 0 \{\tau \leq t\} \cap A \in \mathcal{F}_t\}. \tag{2.19}$$

Then, X_t is said to have the strong Markov property if, for each stopping time τ conditioned on $\{\tau < +\infty\}$, $X_{\tau+t} | X_\tau$ is independent of $\mathcal{F}_\tau$

Taking τ to be deterministic, we can easily see that the strong Markov property implies that a process is Markov. In essence, the Markov property tries to capture memoryless behavior, so that what happens in the future depends only on the present state, independently of past information.

It is clear from the definitions that the Brownian motion and the Poisson process are Lévy processes satisfying the strong Markov property. Thus, we can also later define processes that behave like diffusions with jumps.

[1] A process X_t is said to be stochastically continuous if, for any $\varepsilon > 0$ and all $t_0 > 0$, we have that $\lim_{t \to t_0} \mathbb{P}\{|X_t - X_{t_0}| > \varepsilon\} = 0$

DEFINITION 2.3.4. *(Infinite divisibility) A random variable Y is said to have an infinitely divisible distribution if for every $n \geq 0$, we can write*

$$Y \sim Y_1^{(n)} + ... + Y_n^{(n)}, \tag{2.20}$$

with $\{Y_j^{(m)}\}_{j=1,...,n}$ independent and identically distributed.

Fortunately, Lévy processes can be completely characterized via the Fourier transform, or their *characteristic function*, as it is called in probabilistic language.

THEOREM 2.3.1. *(Lévy-Khintchine) A real valued random variable X has an infinitely divisible distribution if there are parameters $a \in \mathbb{R}$, $\sigma^2 \geq 0$ and a measure ν on $\mathbb{R} \setminus \{0\}$, with $\int_{-\infty}^{+\infty} (1 \wedge x^2)\nu(dx) < +\infty$, such that*

$$\mathbb{E}\left[e^{i\lambda X}\right] = e^{\psi(\lambda)}, \tag{2.21}$$

with

$$\psi(\lambda) = -ia\lambda + \frac{1}{2}\sigma^2\lambda^2 - \int_{-\infty}^{+\infty} (e^{i\lambda x} - 1 - i\lambda x \mathbf{1}_{\{|x|\leq 1\}})\nu(dx). \tag{2.22}$$

DEFINITION 2.3.5. *We call the parameters (a, σ^2, ν) defining a Lévy process its Lévy-Khintchine characteristics.*

The theorem above means that we can characterize a Lévy process by specifying the drift, volatility and a Lévy measure ν, which characterizes the jumps. In our context, we will not need the full generality of a Lévy measure, which can be decomposed into 'small' and 'big' jumps and lead to a countable amount of jumps in any interval, and will only make use of 'well behaved' jump processes.

2.4. One-dimensional jump diffusions

We are now able to present our main object of study. Consider a process X_t, with a connected state space $D_X \subseteq \mathbb{R}$. Then, we call X_t a jump diffusion if, up to the hitting time of the boundary ∂D_X, it is the unique weak solution to the stochastic differential equation

$$dX_t = \mu(X_t)dt + \sigma(X_t)dW_t + dJ_t, \tag{2.23}$$

where J_t is a jump process,

$$J_t = \sum_{n=1}^{N_t} \Delta(X_{T_n^-}, Z_n). \tag{2.24}$$

Here, N_t is a non-explosive counting process with event times T_n and intensity $\lambda_t = \Lambda(X_{t^-})$. Additionally, the variables Z_n encode additional non-state dependent information for the jumps, and are sampled from a given distribution Π. We also assume that $(\mu, \sigma, \Lambda, \Delta, \Pi)$ satisfy suitable conditions for X_t to exist (uniquely), either as a weak or strong solution. Necessary conditions can be found in [25], and the models we are concerned about satisfy these.

Another useful tool to have in mind is the adaption of Itô's Lemma to account for jumps.

THEOREM 2.4.1. *(Itô for jump diffusions) Suppose X_t is a jump-diffusion process satisfying*

$$dX_t = \mu_t dt + \sigma_t dW_t + dJ_t \tag{2.25}$$

Then, for any twice-differentiable function $f(t,x)$ and $Y_t = f(t, X_t)$, one has

$$dY_t = \left(\frac{\partial f}{\partial t} + \mu_t \frac{\partial f}{\partial x} + \frac{\sigma_t^2}{2}\frac{\partial^2 f}{\partial^2 x}\right)dt + \sigma_t \frac{\partial f}{\partial x} dW_t + (f(X_t) - f(X_{t^-}))dN_t. \tag{2.26}$$

Note that the last 'differential' is only in terms of N_t instead of the full jump process J_t, as the jump size information is encoded in the difference $f(X_t) - f(X_{t^-})$. One can also formally build a differential multiplication table, similarly to classical Itô calculus, where we have

$$(dN_t)^2 = dN_t, \tag{2.27}$$

and allows us to generalize the product rule accordingly. This is true because N_t changes only in increments of either 0 or 1.

In order to simplify the problem, we will consider jump diffusions with unit volatility. This will not cause any loss of generality under our assumptions, as we make use of the Lamperti transform, as it is called in [20]. There are not many references about the origin of this transform or how it acquired its name.

DEFINITION 2.4.1. *The Lamperti transform of X_t is the process $Y_t = F(X_t)$, with*

$$F(x) = \int_{X_0}^{x} \frac{1}{\sigma(u)} du.$$

Furthermore, we implicitly assume that it is well defined (and this is true for most relevant models, as the volatility stays away from zero), so that the following proposition holds:

PROPOSITION 2.4.2. *If X_t solves the stochastic differential equation (2.23), then its Lamperti transform $Y_t = F(X_t)$ solves*

$$dY_t = \mu_Y(Y_t)dt + dW_t + dJ_t^Y, \tag{2.28}$$

with $Y_0 = 0$ and drift function

$$\mu_Y(y) = \frac{\mu(F^{-1}(y))}{\sigma(F^{-1}(y))} - \frac{1}{2}\sigma'(F^{-1}(y)), \tag{2.29}$$

provided that σ is differentiable. The jump process $J_t^Y = \sum_{n=1}^{N_t} \Delta_Y(Y_{T_n^-}, Z_n)$ has the same jump times as J_t, but jump size function

$$\Delta_Y(y,z) = F(F^{-1}(y) + \Delta(F^{-1}(y), z)) - y. \tag{2.30}$$

If ∂D_X is absorbing, then so is ∂D_Y, and if it is only attainable through a jump, the same is true for Y.

Equations (2.29) and (2.30) follow directly from Itô-Levi (2.4.1). With this in mind, we now pay special attention to the two technical assumptions underlying the proposed sampling method. Due to the nature of the technique, these are mostly *local* in nature, and distinguish it from more commonly used ones.

Our first assumption is that, on the interior of D_X, μ is C^1, σ is C^2 and Λ is locally bounded[2]. While the first two points ensure that the previous proposition is valid, the local boundedness of the jump intensity allows for the treatment of unbounded jumps, as is the case of simple ones like in the JDCEV model.

Secondly, we assume that the boundary of D_X is either unattainable or attainable only through a jump, in which case we will also assume the boundary to be absorbing. This is reasonable and in line with most equity models, with a state space $[0, +\infty)$: we want the diffusion component to be positivity preserving (ensuring the stock price stays above zero), and the value may only hit 0 after a default event.

This is the class of processes we will be studying. One last useful definition is the integrated drift, which we will be referring to as a function A,

$$A(y) := \int_0^y \mu_Y(x)dx. \tag{2.31}$$

2.5. Risk neutral asset pricing

For the sake of brevity, we assume that the reader has some experience with the basic notions of mathematical finance. For an overview, we refer to [24]. Let us, nevertheless, recall some principles which will be of most use.

First, we expect our market models to be arbitrage free. That is, if you can replicate (or hedge) a derivatives contract, then the value of the contract ought to be equal to the value of the strategy that replicates it. This forms the basis of modern financial markets nowadays, and is the principle by which market makers operate.

Secondly, we require that our (discounted) asset prices are martingales under a *risk neutral* measure, which differs from the real world measure. This restriction allows us to interpret the current value of a contract as an expected value of its terminal value and, in turn, forms the basis of Monte Carlo financial simulations: if we want to compute the value of an option, we 'only' need to simulate enough paths under the risk neutral measure and average over all of these.

It is also desirable that our markets are complete, which essentially means that we have as many securities as sources of randomness (i.e. driving Brownian motions) and allows us, in principle, to hedge every such source. This also implies that the so called risk neutral measure is uniquely defined. Unfortunately, the presence of jumps in the stochastic differential equations defining the models make our markets incomplete, making our choice of risk neutral measure non-unique, and careful economic analysis is needed in order to defined a reasonable 'price of risk'.

[2]We say that a function f is of class C^k if it is differentiable k times with continuous derivatives and locally bounded if, for any x in its domain, there is an interval containing x where f is bounded.

2.5.1. Pricing derivatives and exotic options

Overall, a risk neutral specification allows us to compute the value of complex derivatives, in terms of their payoffs and the value of the underlying S_t. Take, for instance, a European vanilla call, a contract with maturity T and strike K, paying at time T the difference $S_T - K$, if positive. At this moment, we know exactly how much the option is worth, as the value $(S_T - K)^+ | F_T$ is deterministic. In this framework, we can then make use of this knowledge and, in essence, propagate the price backwards, discounting and averaging over all possible results, and write the present value of the option V_t as

$$V_t = e^{-r(T-t)}\mathbb{E}[(S_T - K)^+ | \mathcal{F}_t], \tag{2.32}$$

and we can interpret $\mathcal{F}_t$ as the history up to time t. Note that, in most circumstances, the Markov property is verified and this will be a function of only the parameters defining the process and the present value.

More generally for equity models, if at time T a contract has a payoff given by a function of the value of the underlying asset $\varphi(S_T)$, then we can compute the present value as

$$V_t = \mathbb{E}\left[\frac{\varphi(S_T)}{B_T}\middle| \mathcal{F}_t\right], \tag{2.33}$$

with B_T being our numeraire, commonly just the discounted time value of money. Recall that a numeraire is defined as any tradeable asset with price process B_t such that $B_t > 0$, for all times t. In other words, the relative price of an asset is its price expressed in the units of the numeraire.

We will be interested in other types of derivatives, in order to replicate Giesecke and Smelov's experiments.

While a call option has a positive payoff when the asset value is above a certain strike, a put option pays if it finishes below the strike. Another major important distinction is between European and American options, as the latter allow for early exercise and thus have an implicit value tied to them. However, these are far more complex to treat and are not worth exploring much for our purposes. We can, however, find useful complexity in Asian options.

First, we say that an option is *path-dependent*, and, hence, *exotic*, if its payoff depends on the asset value at times before maturity. For instance, an *Asian option* is a contract which depends on the *average* of the price over a certain time period. While continuous time averages can be considered, even though the integral $\int_0^T X_u du$ is much harder to evaluate, it is more common for Asian options to be discretely monitored.

Thus, a *discretely monitored Asian put* can have its value computed as

$$V_t = e^{-r(T-t)}\mathbb{E}\left[\left(K - \frac{1}{N}\sum_{i=1}^{N} X_{t_i}\right)^+ \middle| \mathcal{F}_t\right]. \tag{2.34}$$

Another large class of option contracts suitable for the exact sampling method are the barrier options, which require the tracking of a barrier hitting time. These can come in

many different flavors, where the barriers can be above or below the current underlying value and can serve as a knock-in (the option only has value if the barrier is hit) or knock-out (the option is terminated and loses its value if the barrier is crossed) barriers. For instance, if $B < X_0$ is our down barrier, a *down-and-out call* can be priced as

$$V_t = e^{-r(T-t)}\mathbb{E}[(X_T - K)^+ \mathbf{1}_{\{\inf_{0\leq t\leq T} X_t > B\}} | \mathcal{F}_t] \tag{2.35}$$

As for interest rate models, even though the 'numeraire' is not deterministic, the time value of money can still be computed by an expectation to avoid arbitrage opportunities and the price of a *risk-less zero coupon bond* is given by

$$B_0 = \mathbb{E}\bigg[\exp\bigg(-\int_0^T X_s ds \bigg) \bigg| \mathcal{F}_0 \bigg]. \tag{2.36}$$

More interesting payoffs can be treated this way as well. We will consider the cap, which is nothing more than multiple calls on an interest rate, with different maturities and same strike:

$$\mathbb{E}\bigg[\sum_{i=1}^{N} \exp\bigg(-\int_0^{t_i} X_s ds \bigg) (X_{t_i} - K)^+ \bigg| \mathcal{F}_0 \bigg]. \tag{2.37}$$

What is left for us to do is to define the dynamics and transition probability distributions of the asset price process X_t, so that the expectations can be computed. Our attention now turns to the two main models of interest that we will later use as benchmarks for the exact method: the JDCEV and AJD models.

2.5.2. JDCEV model

In this section, we describe the jump-to-default extended CEV (JDCEV) model, introduced by Carr and Linestky in 2006 [11].

We say X_t is a CEV process if its dynamics follow the stochastic differential equation

$$dX_t = rX_t dt + aX_t^{\beta+1} dW_t, \tag{2.38}$$

with r and σ positive parameters, and $\beta \in \mathbb{R}$. The major feature of such model, introduced by Cox in 1975 [13], is the existence of a local volatility function, and the elasticity parameter β determines its relation with the state X_t. When $\beta < 0$, we observe the so called leverage effect, where the volatility is inversely proportional to the spot price. In commodity markets, however, it is usual to observe the inverse. Note that $\beta = 0$ yields precisely the Black-Scholes model.

The JDCEV model goes a step further, modelling the evolution of the spot price of e.g. a stock as a CEV process, including the possibility of a default by an instantaneous jump to 0. Thus, assuming an affine jump (default) intensity function depending on the volatility, in accordance with experimental data,

$$\Lambda(x) = b + ca^2 x^{2\beta} \tag{2.39}$$

and jump size function $\Delta(x,z) = -x$, with $b \geq 0$, $c \geq \frac{1}{2}$, the risk neutral dynamics become

$$dX_t = (r + \Lambda(X_t))X_t dt + aX_t^{\beta+1} dW_t + dJ_t, \tag{2.40}$$

where r is the continuous interest rate and the extra term $\Lambda(X_t)X_t dt$ compensates the drift for the jump process J_t and ensures that our discounted stock price is still a martingale. Note that if the stock pays a dividend yield q, compounded continuously, we can adapt the model by simply replacing r with $r - q$.

This model is a clear candidate to test the sampling algorithm. On one hand, it surprisingly allows rather analytical solutions: the theory of squared Bessel processes allows us to solve the CEV component of the process, and conditioning on default or no-default yields the complete solution, for European vanilla options at least. Thus, we can directly check how accurate our results are.

On the other hand, the unbounded default intensity leaves the model outside the scope of most Monte Carlo methods, which motivates the work of Giesecke and Smelov. Due to the way we define the default intensity, inversely proportional to the stock price as we only consider $\beta < 0$, it can be proven that 0 is attainable only through a jump.

Regarding its specification as a unit volatility process, we note that

$$F(x) = \frac{X_0^{-\beta} - x^{-\beta}}{a\beta} \tag{2.41}$$

for $x \in (0, +\infty)$ and $F(0) = X_0^{-\beta}/(a\beta)$, so that

$$F^{-1}(y) = (X_0 - ya\beta)^{-1/\beta}. \tag{2.42}$$

As before we had $D_X = [0, +\infty)$, we have that $D_Y = [X_0^{-\beta}/(a\beta), +\infty)$. According to (2.29) and (2.30), the drift coefficient becomes

$$\mu_Y(y) = \frac{1}{a}(r+b)(X_0^{-\beta} - ya\beta) + a\frac{c - (\beta+1)/2}{X_0^{-\beta} - ya\beta} \tag{2.43}$$

and the jump size function

$$\Delta_Y(y,z) = F(0) - y. \tag{2.44}$$

Finally, we have that

$$A(y) = \frac{1}{a}(r+b)\left(X_0^{-\beta}y - \frac{y^2}{2}a\beta\right) - \frac{c-(\beta+1)/2}{\beta}\log\left(1 - ya\beta X_0^{\beta}\right). \tag{2.45}$$

2.5.3. Affine Jump Diffusion models

In this section, we describe the affine jump diffusion interest rate models (or simply AJD), first presented by Duffie [18], [17]. Under a risk-neutral specification, the short rate process X_t satisfies the stochastic differential equation

$$dX_t = \kappa(\theta - X_t)dt + \sigma\sqrt{X_t}dW_t + dJ_t, \tag{2.46}$$

with positive parameters initial condition X_0, mean θ, mean-reversion speed κ and volatility σ. The drift and diffusion terms of the equation form what is known as a Feller diffusion, the same process that the volatility in the Heston model follows.

Furthermore, the jump intensity function is given by

$$\Lambda(x) = \Lambda_0 + \Lambda_1 x. \tag{2.47}$$

Note that the model obtains its name from the affine structure in both the drift, the volatility and the jump intensity in terms of X_t. Finally, the jump size is simply

$$\Delta(x, z) = z, \tag{2.48}$$

with z being sampled from a distribution of our choice, without much technical restriction. A common choice is the exponential distribution, but the authors of [20] choose to follow the approach of Zhou [38], where a uniform distribution in a small interval is considered and the parameters are estimated from weekly observations of the US federal funds rate between 1954 and 1999.

As for the specification under the exact method, the Lamperti transform of the AJD process is

$$F(x) = \frac{2(\sqrt{x} - \sqrt{X_0})}{\sigma}, \tag{2.49}$$

so that

$$F^{-1}(y) = \left(\frac{\sigma}{2}y + \sqrt{X_0}\right)^2 \tag{2.50}$$

As we previously had $D_X = (0, +\infty)$, we now have $D_Y = (-2\sqrt{X_0}/\sigma, +\infty)$ and the drift function is given by

$$\mu_Y(y) = \frac{4\kappa\theta - \sigma^2}{2\sigma^2(y + 2\sqrt{X_0}/\sigma)} - \frac{\kappa}{2}(y + 2\sqrt{X_0}/\sigma). \tag{2.51}$$

Finally, the jump size function takes the form

$$\Delta_Y(y, z) = \frac{2}{\sigma}\left[\left((\sqrt{X_0} + \sigma y/2)^2 + z\right)^{\frac{1}{2}} - \sqrt{X_0}\right] - y \tag{2.52}$$

and compute

$$A(y) = \frac{4\kappa\theta - \sigma^2}{2\sigma^2}\log\left(1 + y\sigma/(2\sqrt{X_0})\right) - \frac{\kappa}{2}\left(\frac{y^2}{2} + 2y\sqrt{X_0}/\sigma\right). \tag{2.53}$$

CHAPTER 3

Monte Carlo methods

Monte Carlo methods (MCM) were first based on the correspondence between volume and probability. Consider the quantity

$$\alpha = \int_0^1 f(x)dx, \tag{3.1}$$

for some integrable function f. If U is a random variable with uniform distribution in the $[0, 1]$ interval, then we can write the expression above as $\alpha = \mathbb{E}_{\mathbb{P}}[f(U)]$. As such, if we generated n independent samples of U, $(U_1, ..., U_n)$, by the central limit theorem, the estimator

$$\hat{\alpha} = \frac{1}{n}\sum_{j=0}^{n} f(U_j), \tag{3.2}$$

is asymptotically normally distributed, with mean α and standard deviation $\frac{\sigma_U}{\sqrt{n}}$. While the square-root convergence is relatively slow, making MCM inefficient in few variables, it is in fact independent of the number of dimensions and allows for high-dimensional numerical integration. This type of error bounds hold in general for most MCM.

Generating uniform random variables is the basis for Monte Carlo simulation. Even though true random number generation is not possible, pseudo-random generators like linear congruential generators work very well for most purposes, granted that their drawbacks and lattice structure is taken into consideration. We refer to [21] for an extensive treatment of such methods.

3.1. Bias and error

Let us explain possible origins of bias in Monte Carlo sampling.

Bias is a well known phenomenon in statistical theory. Recall first that an estimator of an unknown parameter θ, which may be, for instance, the mean or standard deviation of a distribution, is a function $\hat{\theta} = \hat{\theta}_n = \hat{\theta}(X_1, ..., X_n)$ of n samples of our true random variable X that *somehow* allows us to infer the value of θ, with increasing accuracy as n grows to infinity. We are being intentionally vague, as this 'definition' encompasses many different estimation methods, and we may omit the dependency in n.

More precisely, we say that an estimator is *centered* if

$$\mathbb{E}[\hat{\theta}] = \theta, \tag{3.3}$$

and it is biased if this does not hold. An estimator is asymptotically centered if

$$\lim_{n\to+\infty} \mathbb{E}[\hat{\theta}_n] = \theta. \tag{3.4}$$

The main causes of bias are non-linearity and approximation errors. While the latter may come, for instance, from the discretization of integrals, which comes up when dealing with stochastic interest rate models, people are usually more familiar with the former. Suppose that X is a normal random variable with standard deviation σ and mean zero, for the sake of simplicity. As usual, consider the following estimator for the variance σ^2

$$\hat{\sigma}^2 = \frac{1}{n}\sum_{k=1}^{n} X_n^2, \tag{3.5}$$

which is centered. However, if we wish to estimate the standard deviation itself, we already have a bias, as a consequence of Jensen's inequality:

$$\mathbb{E}[\hat{\sigma}] = \mathbb{E}\left[\sqrt{\frac{1}{n}\sum_{k=1}^{n} X_n^2}\right] \leq \sqrt{\mathbb{E}\left[\frac{1}{n}\sum_{k=1}^{n} X_n^2\right]} = \sigma. \tag{3.6}$$

In this 'toy' example, we know that the bias can be corrected by a factor of $\sqrt{\frac{n}{n-1}}$, but we are not so lucky in most situations. We refer to [34] for more information on this topic, as our goal with exact sampling is to avoid these computations entirely.

In general, our measurement of error will be the root mean square error (RMSE), defined as

$$RMSE = \sqrt{SE^2 + Bias^2}, \tag{3.7}$$

with SE, the standard error of the sample, being the standard deviation of the sample output divided by the square root of the number of trials.

3.2. General sampling

In this section, the two most general techniques for generating samples from given distributions are presented, assuming we can generate a sequence U_1, U_2, ..., of i.i.d. standard uniform random variables - most MCM can be broke down up to this point.

We begin with the *inverse transform method*. Recall that an α-quantile of a distribution is the value x_α such that $\mathbb{P}[X \leq x_\alpha] = \alpha$. Under suitable assumptions, it is easy to see that $x_\alpha = F^{-1}(\alpha)$. By definition, the sample quantiles are uniformly distributed - that is, the probability that a sample is less then the α-quantile is exactly α.

Formally, let U be a uniform random variable and $Y := F^{-1}(U)$. We then compute

$$\mathbb{P}[Y \leq x] = \mathbb{P}[F^{-1}(U) \leq x] = \mathbb{P}[U \leq F(x)] = F(x) \tag{3.8}$$

and the proposition below follows:

PROPOSITION 3.2.1. *(Inverse transform method) Suppose $U \sim U(0,1)$, $F(x) = \mathbb{P}[X \leq x]$ for some random variable X, and let $Y = F^{-1}(U)$. Then we have that $\mathbb{P}[Y \leq x] = F(x)$, that is, Y has the same distribution as X.*

Thus, if we know the distribution of X, and we can invert it (even if numerically), then we can generate samples of X. The applications are immediate. For instance, if we

consider the exponential distribution, that is,

$$\mathbb{P}[X \leq x] = 1 - e^{-\lambda x}, \tag{3.9}$$

then the inverse transform yields

$$X = -\frac{1}{\lambda}\log(1-U). \tag{3.10}$$

For discrete distributions, the method is even simpler, since no inversion is needed and we only need to lookup the table of discrete values of F.

One benefit of this approach is that it also allows us to sample from conditional distributions. By setting $V = F(a) + (F(b) - F(a))U$, the inverse transform on V has the same distribution as $X \sim F$, conditional on $a < X \leq b$.

The second technique, which we will make the most use of, is called the *acceptance-rejection method*, and allows us to sample from a given distribution by sampling from another one, hopefully less complex, rejecting a random subset.

Let f be the density we want to sample from, defined in some Ω, and g be such that $f(x) \leq cg(x)$, $\forall x \in \Omega$, for some constant c. The idea is as follows: we start by sampling from the 'larger' distribution - the inequality assures us that we can re-scale the distributions such that g is always 'more likely'. We can then turn to point-wise comparison, and use the ratio $f(X)/cg(X) \leq 1$ as a measure of how likely it is that our initial sample X from g is a sample from f. In practice, we can proceed as follows

Algorithm 1: Acceptance/Rejection

Result: Generates X from f

while *no sample is accepted* **do**
- Sample X from g;
- Sample $U \sim U(0,1)$;
- **if** $U \leq f(X)/cg(X)$ **then**
  - Accept X as a sample from f;
- **else**
  - Reject X;

The following proposition assures us that this method returns a sample from the required distribution.

Proposition 3.2.2. *Let Y be the sample returned by the acceptance-rejection algorithm. Then $Y \sim f$.*

Proof. Given an event A, we first observe that

$$\mathbb{P}[Y \in A] = \mathbb{P}[X \in A | U \leq f(X)/cg(X)] = \frac{\mathbb{P}[X \in A, U \leq f(X)/cg(X)]}{\mathbb{P}[U \leq f(X)/cg(X)]} \tag{3.11}$$

and

$$\mathbb{P}[U \leq f(X)/cg(X)] = \int_\Omega \frac{f(x)}{cg(x)} g(x)dx = \frac{1}{c}. \tag{3.12}$$

Combining both equations, we have that

$$\mathbb{P}[Y \in A] = c\mathbb{P}[X \in A, U \leq f(X)/cg(X)] = c\int_A \frac{f(x)}{cg(x)} g(x)dx = \int_A f(x)dx \tag{3.13}$$

which proves that Y has f as its density. □

3.3. Sampling from specific distributions

As we have saw in equation (3.10), we already know how to sample directly from an exponential distribution. While logarithms are computationally expensive and should in general be avoided, since the sample is immediate we will not be concerned about this.

However, most distributions do not allow for fast inversion, and we need alternative methods, adapted to the density in question. This is a very deep topic with immense material for discussion, so we will discuss only some of the most common methods.

3.3.1. Normal distribution

We first consider the normal distribution, which is effectively one of the building blocks of financial simulation. Given that we have an expression for its density, there are a wide variety of methods relying on both techniques shown previously. However, specific methods available for this distribution are usually more efficient, as is the case of the Box-Müller method:

Algorithm 2: Box-Müller method

Result: Z_1, Z_2 i.i.d. standard normal random variables

1 Generate independent uniform random variables U_1, $U_2 \sim U(0,1)$;

2 Set $Z_1 = \sqrt{-2\log U_1}\sin(2\pi U_2)$, $Z_2 = \sqrt{-2\log U_1}\cos(2\pi U_2)$.

Another slightly better alternative is Marsaglia's polar method, which avoids the computation of the sine/cosine transcendental functions and essentially adapts the Box-Müller method with acceptance-rejection on the unit square.

In practice, however, we will make use of the Numpy package in Python to generate Gaussian samples with the polar method, as its implementation favours efficiency. One could, in theory, implement even faster Gaussian generators with adaptions of the Ziggurat algorithm, which we will not discuss. We refer to [31] for an overview on the topic.

3.3.2. Gamma distribution

One other distribution worth focusing on is the Gamma distribution, and generating samples is much more involved in this case. As before, we will make use of the Numpy package to generate Gamma distributed random variables, but we will briefly discuss how this can be implemented.

First, recall that a random variable X is said to be Gamma distributed with shape parameter $\alpha > 0$ and rate parameter β (or equivalently a scale parameter $\theta = 1/\beta$) if its probability density function is given by

$$\rho(x;\alpha,\beta) := \frac{\beta^\alpha x^{\alpha-1}e^{-\beta x}}{\Gamma(\alpha)} \tag{3.14}$$

By the scaling property of the Gamma distribution, we only need to focus on the standard case. That is, given $\beta > 0$ and $X \sim \text{Gamma}(\alpha, 1)$, then $\frac{1}{\beta}X \sim \text{Gamma}(\alpha, \beta)$. Now, let $\alpha = n + \delta$, with n an integer and $0 < \delta < 1$.

If U_k is uniformly distributed in $[0, 1]$, then $-\log U_k$ is Gamma$(1, 1)$ (or, equivalently, standard exponential) distributed. By the additive property, we have that

$$-\sum_{k=1}^{n} \log U_k \sim \text{Gamma}(n, 1). \tag{3.15}$$

Thus, what is left for us to generate is a sample of Gamma$(\delta, 1)$, which is considerably harder. For this purpose, we can use acceptance-rejection via the Ahrens-Dieter method [1].

Algorithm 3: Ahrens-Dieter acceptance-rejection method

Result: ξ Gamma$(\delta, 1)$-distributed

1 Generate independent uniform random variables $U, V, W \sim U(0, 1)$;
2 **if** $U \leq \frac{e}{e+\delta}$ **then**
3 | Set $\xi = V^{1/\delta}$, $\eta = W\xi^{\delta-1}$;
4 **else**
5 | Set $\xi = 1 - \log V$, $\eta = We^{-\xi}$;
6 **if** $\eta > \xi^{\delta-1}e^{-\xi}$ **then**
7 | Return to step 1;
8 Return ξ.

Overall, we then have that

$$\frac{1}{\beta}\left(-\sum_{k=1}^{n} \log U_k + \xi\right) \sim \text{Gamma}(n + \delta, \beta). \tag{3.16}$$

3.4. Brownian sampling

As we saw that $W_t - W_s \sim \mathcal{N}(0, |t - s|)$, sampling the path of a Brownian motion can be reduce to simply sampling from a normal distribution.

A closely related and more complicated process of interest is the Brownian bridge B_t on $[0, T]$, defined by

$$B_t := (W_t | W_T = 0). \tag{3.17}$$

In essence, it is a Brownian motion that begins and ends at 0, and it admits several representations in terms of paths of Brownian motions, reducing the problem of sampling from a Brownian bridge to sampling from a normal distribution. In particular, we have that

$$B_t = W_t - \frac{t}{T}W_T. \tag{3.18}$$

Thus, if we need to generate values of a Brownian bridge along several points, we only need to simulate a Brownian path and apply the transformation above.

3.4.1. Exit times

Consider the exit time $\tau := \inf\{t : |W_t| = 1\}$. Note that, by property (2.1), we can re-scale our Brownian motion, with $\tau_\theta \overset{d}{=} \theta^2\tau$, and do not lose any generality.

Burq and Jones [10] show, using the Laplace transform and the martingale stopping theorem, that the density $h(t)$ of τ is given by

$$h(t) := \frac{e^{-\frac{1}{2t}}}{\sqrt{2\pi t^3}} + \sum_{j=1}^{+\infty} \frac{(-1)^j}{\sqrt{2\pi t^3}} \left[(2j+1) \exp\left(-\frac{(2j+1)^2}{2t} \right) - (2j-1) \exp\left(-\frac{(2j-1)^2}{2t} \right) \right], \tag{3.19}$$

and we can use acceptance/rejection to sample from it, as it is bounded above by a Gamma distribution:

$$h(t) \leq a g(t; b; \gamma) := \frac{a\gamma^b}{\Gamma(b)} t^{b-1} e^{-\gamma t}, \tag{3.20}$$

with Γ being the Gamma function, $a = 1.243707$, $b = 1.088870$ and $\gamma = 1.233701$.

Thus, we sample ν from $g(t; b; \gamma)$ and accept it as from τ if, given $U \sim U(0, 1)$:

$$aUg(\nu; b; \gamma) \leq h(\nu). \tag{3.21}$$

We cannot explicitly compute the value of $h(\nu)$ as it is defined as an infinite sum. However, due to the oscillating nature of the sum, Burk and Jones proved that, if $aUg(\nu; b; \gamma) < h^{n+1}(\nu) < h^n(\nu)$ for some n, then $aUg(\nu; b; \gamma) < h(\nu)$, where h^n denotes the n-th partial sum. Analogously, if $aUg(\nu; b; \gamma) > h^{n+1}(\nu) > h^n(\nu)$, then $aUg(\nu; b; \gamma) > h(\nu)$.

A sequence a_n is called oscillating if there exists a positive N such that, for $i \geq N$,

$$0 < -\frac{a_{i+1} - a_i}{a_i - a_{i-1}} < 1. \tag{3.22}$$

Granted that the partial sums satisfy this relation, the condition to terminate the sum early follows. The authors also concluded that only a finite number of terms has to be computed to determine this N.

3.4.2. Brownian meanders

Suppose now we have sampled an exit time τ and correspondingly W_τ, which is just θ or $-\theta$ with equal probability, and wish to sample for a value of W_t, with $t < \tau$, conditional on (τ, W_τ). As τ is the first time the Brownian process hits $\pm\theta$, we want to sample from a Brownian process that stays away from θ until τ. For this purpose, consider the following definition:

Definition 3.4.1. *Let $\tau' := \sup\{t \in [0, 1] : W_t = 0\}$ be the last time before $t = 1$ when W_t visits 0. A Brownian meander W_t^+ is defined by:*

$$W_t^+ := \frac{1}{\sqrt{1-\tau'}} |W_{\tau' + t(1-\tau')}|. \tag{3.23}$$

In essence, a Brownian meander results from ignoring the trajectory before τ', the last time the Brownian walk hits 0, and re-scaling the remaining part. Thus, W_t^+ is a

Brownian walk that stays away from 0. Transition density functions are known for this process, but we will not focus on them.

Williams [37] shows that, given (τ, W_τ), W_t behaves like a time-reversed Brownian meander, while Imhof [26] proves that a Brownian meander can be decomposed as a function of 3 independent Brownian bridges over $[0, \tau]$. We will follow the acceptance/rejection approach described in Chen and Huang [12], where they make use of both of these facts.

Again, by the re-scaling property of Brownian motion (2.1), we will consider the case where $\theta = 1$, and τ_1 is the corresponding hitting time. Now, given (τ_1, W_{τ_1}), let

$$V_t := \begin{cases} 1 - W_{\tau_1 - t}, & \text{if } W_{\tau_1} = 1 \\ 1 + W_{\tau_1 - t}, & \text{if } W_{\tau_1} = -1 \end{cases} \tag{3.24}$$

and

$$B_t := \sqrt{(t/\tau_1 + B^1_{\tau_1 - t})^2 + (B^2_{\tau_1 - t})^2 + (B^3_{\tau_1 - t})^2}, \tag{3.25}$$

the decomposition suggested by Imhof, with B^i_t three independent Brownian bridges. Chen and Huang show that the likelihood ratio between these two variables satisfies

$$\frac{\mathbb{P}(V_{t_1} \in dy_1, ..., V_{t_n} \in dy_n | \tau_1, W_{\tau_1})}{\mathbb{P}(B_{t_1} \in dy_1, ..., B_{t_n} \in dy_n)} \propto \prod_{i=1}^{n} p(t_i, y_i; t_{i+1}, y_{i+1}) q(t_1, y_1) \mathbf{1}_{(0,2)}(y_i), \tag{3.26}$$

with

$$p(s, x; t, y) := \frac{1 - \sum_{j=1}^{+\infty} (\theta_j(s, x; t, y) - \vartheta_j(s, x; t, y))}{1 - \exp(-2xy/(t-s))}, \tag{3.27}$$

$$q(s, x) = 1 - \frac{1}{x} \sum_{j=1}^{+\infty} (\rho_j(s, x) - \varrho_j(s, x)), \tag{3.28}$$

and

$$\theta_j(s, x; t, y) := \exp\left(- \frac{2(2j - x)(2j - y)}{t - s} \right) + \exp\left(- \frac{2(2(j-1) + x)(2(j-1) + y)}{t - s} \right), \tag{3.29}$$

$$\vartheta_j(s, x; t, y) := \exp\left(- \frac{2j(4j + 2(x - y))}{t - s} \right) + \exp\left(- \frac{2j(4j - 2(x - y))}{t - s} \right), \tag{3.30}$$

$$\rho_j(s, x) := (4j - x) \exp\left(- \frac{4j(2j - x)}{s} \right), \tag{3.31}$$

$$\varrho_j(s, x) := (4j + x) \exp\left(- \frac{4j(2j + x)}{s} \right). \tag{3.32}$$

As discussed in the previous section, the authors take advantage of the oscillating nature of these sums in order to perform the required acceptance test with a finite number of summands, under the same condition as before, and prove that this can be done with probability one. With these facts in mind, the algorithm for sampling $(W_{t_1}, ..., W_{t_n})$ given (τ, W_τ) can be stated as:

Algorithm 4: Brownian meander sampling

Result: Skeleton $(W_{t_1}, ..., W_{t_n})$ given (τ, W_τ)

while *no skeleton is accepted* **do**

  Sample τ_1 as in section 3.4.1, and W_{τ_1}, $\mathbb{P}(W_{\tau_1} = 1) = \mathbb{P}(W_{\tau_1} = -1) = \frac{1}{2}$;

  Sample $(B^i_{t_1}, ..., B^i_{t_n})_{i=1,2,3}$, independent Brownian bridges from 0 to 0 on $[0, \tau_1]$, as in section 3.4;

  Transform these samples into samples of $(B_{\tau_1 - t_1}, ..., B_{\tau_1 - t_n})$;

  if

$$U_j \leq p(\tau_1 - t_j, B_{\tau_1 - t_j}, \tau_1 - t_{j+1}, B_{\tau_1 - t_{j+1}}), \qquad 1 \leq j \leq n \tag{3.33}$$

$$U_{n+1} \leq q(\tau_1 - t_n, B_{\tau_1 - t_n}). \tag{3.34}$$

  then

    Accept $(B_{\tau_1 - t_1}, ..., B_{\tau_1 - t_n})$ as a sample of $(V_{\tau_1 - t_1}, ..., V_{\tau_1 - t_n})$;

    Transform $V_{\tau_1 - t}$ to W_t, and re-scale by θ;

  else

    Reject the proposal skeleton;

One might ask if we could not simply sample Brownian paths and reject those that cross the intended boundary. While this method would certainly yield Brownian meander paths, it does not do so with the right frequency or probability and is *biased*. To understand why we need a more convoluted method, picture a Brownian bridge which we sample at discrete times and suppose that at two consecutive instants it is close to the barrier. While our sample is an exact sample of the Brownian path at these points, we have no information about what happens between them, and there is a small probability that it did cross the barrier in the meanwhile. An acceptance-rejection scheme is thus needed, so that we reject the right amount of paths that are 'too close' to the barrier.

3.5. Jump process sampling

Since the Poisson process has independent increments, inter arrival times are independent and exponentially distributed with parameter λ. Thus, we can sequentially generate arrival times by sampling the increments, using, for instance, the inverse transform method.

If, however, we want to sample from non-homogeneous or even doubly stochastic Poisson processes, we can still do this exactly through a so called thinning procedure. Thinning a Poisson process refers to classifying each random point independently, into one of a finite number of different types. The random points of a given type also form Poisson processes, and these processes are independent. We call $\tilde{N}_t$ a p-thinning of N_t

if points from N_t are accepted with probability $0 < p < 1$. In general, the following proposition holds:

PROPOSITION 3.5.1. *Let N_t be a Poisson process with intensity $\lambda > 0$ and $\tilde{N}_t$ be the p-thinning of N_t. Then, $\tilde{N}_t$ is a Poisson process with intensity λp.*

This fact can be proven using generating functions, and more general versions hold analogously when p is either a deterministic function or an adapted process. We omit the proof and statements here, which can be found in [28].

We may proceed as follows. Let V_t be a doubly stochastic Poisson process with intensity λ_t, satisfying $0 \leq \lambda_t \leq \lambda$, for all t a.s.. We obtain the samples of jump times of V_t by first sampling jump times τ_j of a Poisson process with intensity λ, and accepting/rejecting each one with probability equal to the ratio λ_{τ_j}/λ.

It is important to remind ourselves that this method relies on a boundedness assumption over the jump rates, which is a barrier we will overcome later.

3.6. Discretization methods for stochastic differential equations

Discretization methods approximate the evolution of a stochastic process over a discrete grid, considering small increments instead of continuous integration. Instead of developing general approximation theory for stochastic differential equations, we will focus on particular methods for the two models described before in section 2.5.

3.6.1. Discretization of the JDCEV model

To approximate the stochastic differential equation (2.40) over an interval $[0, T]$, we first consider a discrete time grid of N segments of length $h = \frac{T}{N}$.

Before a jump time, the evolution of the process is described only by the diffusive and drift terms of the equation. By integrating the equation over a small time step and approximating the integrals up to first order, we obtain a first order approximation of the diffusion equation:

$$\hat{X}_{i+1} = \hat{X}_i + (r + \Lambda(\hat{X}_i))\hat{X}_i h + a\hat{X}_i^{\beta+1}\mathcal{N}_i, \tag{3.35}$$

with $\hat{X}$ being our estimator, and $\{\mathcal{N}_i\}_i$ a sequence of i.i.d. $\mathcal{N}(0, h)$ random variables, for $i \in \{0, ..., N-1\}$.

Higher order methods may be obtained by interpolating the integrals over a larger number of points, which may also lead into implicit methods. We will not discuss them here, and refer to [23] for an exposition.

We now need to determine the jump time. By discretizing the compensator of the jump process

$$C_t = \int_0^t \Lambda(X_s)ds \tag{3.36}$$

up to first order, we obtain the estimator

$$\hat{C}_{i+1} = h\sum_{k=0}^{i} \Lambda(\hat{X}_k). \tag{3.37}$$

Thus, recalling section 2.2, we can sample an independent standard exponential random variable $\mathcal{E}$, and take as a jump time the index i when we first have $\mathcal{E} \leq \hat{C}_{i+1}$. However, as the process hits 0 only through default, we have to avoid negative values and take the moment the process turns negative through diffusion as a jump time.

We can then write our estimate for the first jump time as

$$\hat{T}_1 = \inf\{ih : \mathcal{E} \leq \hat{C}_{i+1}\} \wedge \inf\{ih : \hat{X}_{i+1} \leq 0\}. \tag{3.38}$$

Note that a jump implies the default of the firm and the value stays at 0 from then on.

3.6.2. Discretization of the AJD model

Previously, we did not need to pay much attention to the positivity of the process, as the process 'died' when reaching 0. However, to simulate the AJD model we need to be careful about how we ensure that the process stays positive, and there are several issues to address.

Discretizing as before, we could obtain the following scheme

$$\hat{X}_{i+1} = \hat{X}_i + \kappa(\theta - \hat{X}_i)h + \sigma\sqrt{\hat{X}_i}\mathcal{N}_i. \tag{3.39}$$

Unfortunately, our estimator $\hat{X}_i$ can turn negative with positive probability, and it would leave the next time step undefined. A good first step to take is avoiding the negative square root by substituting $\hat{X}_i^+$ for $\hat{X}_i$, with $\hat{X}_i^+ := \max(0, \hat{X}_i)$.

Once again, there is still a positive probability that the estimator turns negative, even though it will deterministically drift up afterwards. Due to the affine structure of the jump intensity, this may lead to negative intensities, which is not well defined. Resampling when we get negative values is not helpful eather, as that introduces more bias, and a simple approach is to take $\Lambda(x) = \Lambda_0 + \Lambda_1 x^+$.

One final substitution leads us to the scheme

$$\hat{X}_{i+1} = \hat{X}_i + \kappa(\theta - \hat{X}_i^+)h + \sigma\sqrt{\hat{X}_i^+}\mathcal{N}_i. \tag{3.40}$$

In the context of the Heston model, where the volatility process follows a Feller diffusion equal to the AJD model, Lord et al. [30] observe that (3.40) leads to the smallest discretization error among several other schemes, reliant on truncating or reflecting the process to deal with negative values, and converges strongly as well.

To avoid all the nuances above, one might settle for an alternative first order scheme such as

$$\hat{X}_{i+1} = \left[(1 - \kappa h/2)\sqrt{\hat{X}_i} + \frac{\sigma\mathcal{N}_i}{2(1 - \kappa h/2)}\right]^2 + (k\theta - \sigma^2/4)h, \tag{3.41}$$

which ensures positivity and converges strongly if $4k\theta > \sigma^2$ and $kh \neq 2$, or even a second order one, for higher precision:

$$\hat{X}_{i+1} = e^{-\kappa h/2}\left\{\left((\theta\kappa - \sigma^2/4)\frac{1 - e^{-\kappa h/2}}{\kappa} + e^{-\kappa h/2}\hat{X}_i\right)^{\frac{1}{2}} + \frac{\sigma\mathcal{N}_i}{2}\right\}^2 + (\kappa\theta - \sigma^2/4)\frac{1 - e^{-\kappa h/2}}{\kappa}. \tag{3.42}$$

Both of these methods were introduced by Alfonsi [2], [3], which we refer to for in depth details related to the derivation.

The jump times may be treated using the compensator method described before, and we need only to re-sample a mark variable $\mathcal{E}$ and reset the compensator after jumping.

Interest rate models and related payoffs also may force us to compute the time-integrated exponential of the process,

$$\exp\left(\int_0^T X_t dt\right). \tag{3.43}$$

For this purpose, we will approximate $\int_0^T X_t dt$ using the trapezoidal rule, a second order method of numerical integration, given by

$$\int_0^T X_t dt \approx \sum_{n=0}^{N-1} \frac{X_{nT/N} + X_{(n+1)T/N}}{2}. \tag{3.44}$$

CHAPTER 4

Building up the algorithm

This chapter presents step by step, alongside [20], how to sample from jump diffusions via a rejection sampling mechanism, with increasing complexity until we meet the full generality of section 2.4. Our objective is to generate samples of a *skeleton* Σ of the process,

$$\Sigma = ((X_t)_{t\in S}, (N_t, J_t)_{t\in S}, \xi \wedge T, X_{\xi\wedge T}), \tag{4.1}$$

with $T > 0$ being the time horizon, S a set of fixed times up to T and ξ an exit time. If our samples are exact - that is, do not rely on approximations or 'cheat' non-linearities, we will also have unbiased Monte Carlo estimates of expectations

$$V = \mathbb{E}[v(\Sigma)], \tag{4.2}$$

for some function of interest v. Note that this encompasses cases where V is a statistic of our process, the value of a contingent claim or probability of default. This method also allows for unbiased estimation of more general expectations of the form

$$\mathbb{E}\left[\exp\left(-\int_0^T X_s ds\right) u(\Sigma)\right], \tag{4.3}$$

which are of interest in credit rates and fixed-income securities.

Our goal for this chapter is to make it clear how on each step the difficulties arising from relaxed assumptions are handled.

4.1. Rejection sampling for diffusions

We now begin building the acceptance/rejection mechanism that will allow us to draw exact samples from the jump diffusion, by first presenting a method for obtaining exact samples of Y_T, when Y_t is a unit-volatility diffusion with no jumps.

Let

$$A(y) := \int_0^y \mu_Y(s) ds, \tag{4.4}$$

and consider the proposal density $g(y) := \exp(A(y) - y^2/2T)/C$, with

$$C := \int_{-\infty}^{+\infty} \exp(A(y) - y^2/2T) dy < +\infty, \tag{4.5}$$

under suitable assumptions on μ_Y. Suppose as well that μ_Y satisfies Novikov's condition, that is

$$\mathbb{E}\left[\exp\left(\frac{1}{2}\int_0^T \mu_Y^2(W_t) dt\right)\right] < +\infty. \tag{4.6}$$

Beskos and Roberts [6] show that, under the conditions stated above, $g(y)$ satisfies

$$\frac{f_{Y_T}(y)}{g(y)} \propto \mathbb{E}\left[\exp\left(-\int_0^T \phi(W_s)ds\right)\middle| W_T = y\right] := H(y), \tag{4.7}$$

with

$$\phi(y) := \frac{1}{2}(\mu_Y'(y) + \mu_Y^2(y)). \tag{4.8}$$

While we might have no tools to compute the transition density f_{Y_T}, since the ratio satisfies $H(y) \leq 1$, after generating a sample Y from g, we can then accept or reject it based on a Bernoulli random variable with probability $H(Y)$ to obtain a sample from f. The expectation is not straightforward to compute, and trying to numerically compute the integral will lead to biased results. However, this can be avoided by a clever interpretation of this value.

Consider a doubly stochastic Poisson process V_t, with intensity $\phi(W_t)$, assuming $\phi \geq 0$. It is clear that

$$\mathbb{P}[V_T = 0 | W_T = y] = \mathbb{E}\left[\exp\left(-\int_0^T \phi(W_s)ds\right)\middle| W_T = y\right] = H(y). \tag{4.9}$$

The required Bernoulli indicator can then be generated by sampling jump times of V_t conditioned on $W_T = Y$, and accepting the sample if no jump occurs before the instant T. We will see later that this method is also valid under more general assumptions on ϕ.

Even though the sampling of arrival times for a doubly stochastic Poisson process is itself another challenge, this can be done exactly if $0 \leq \phi(x) \leq \lambda$ for some constant λ, by thinning a Poisson process with intensity λ, as we explained in section 3.5.

Candidate arrival times ν are then accepted with probability $\phi(W_\nu)/\lambda$, with W_ν sampled from a Brownian bridge with endpoint $W_T = Y$.

Summarizing, the algorithm for generating exact samples of Y_T can be stated as:

Algorithm 5: Diffusion sampling

Result: Generates Y_T from f_{Y_T}

while *no sample is accepted* **do**
- Sample Y from g;
- Generate candidate jump times $\nu_1, ..., \nu_l < T$ from a Poisson process with intensity λ;
- Sample $(W_{\nu_1}, ..., W_{\nu_l})$ from a Brownian bridge with endpoint $W_T = Y$;
- Set $i = 1$;
- **while** $i \leq l$ **do**
  - Generate $U \sim U(0, 1)$;
  - **if** $U \leq \phi(W_{\nu_i})/\lambda$ **then**
    - Accept ν_i as a jump time and reject Y as a sample from f_{Y_T};
  - **else**
    - Set $i = i + 1$;
- Accept Y as a sample from f_{Y_T}.

4.2. Localization

The previous section is built upon the assumptions of boundedness for μ_Y and ϕ, and Novikov's condition, which are quite stringent and limit the scope of models suitable for the algorithm. In order to lift these restrictions, Chen and Huang [12] have developed a localization technique, which has much milder conditions on local boundedness and integrability.

The idea is as follows: instead of trying to sample for the horizon T, we will decompose the range of Y into bounded segments, ensuring that no explosion occurs. Sampling for exit times using the acceptance/rejection mechanism developed previously is possible with minor changes to the algorithm. This is one of the most versatile insights provided, as it vastly simplifies the treatment of, for instance, barrier options, as hitting times are obtained directly.

To illustrate, we present the steps for the first segment, as the process can be continued with simple adjustments. We first choose the level to consider, selecting $\theta > 0$ such that $[-\theta, \theta] \subset \text{int } D_Y$, and consider the exit time $\zeta := \inf\{t > 0 : |Y_t| > \theta\}$. Further details about level selection can be found in section 5.1, which develops on the relation between this choice and performance.

Our goal is then to generate the pair (ζ, Y_ζ) via an appropriate acceptance-rejection mechanism. For this purpose, we will consider now a proposal Brownian pair (τ, W_τ), with $\tau := \inf\{t > 0 : |W_t| > \theta\}$. By the symmetry of Brownian motion, we have that W_τ is either θ or $-\theta$ with equal probability, while τ can be generated by the method we present in section 3.4.1.

The required likelihood ratio can be derived by means of Girsanov's theorem, as this gives us the weight under which our diffusion Y_t is a Brownian motion. We will not develop the details here and instead provide the formula for this ratio, leaving the computations to be carried out later on for the general case:

$$\exp(A(W_\tau))\mathbb{E}\left[\exp\left(-\int_0^\tau \phi(W_s)ds\right)\middle|\tau, W_\tau\right]. \tag{4.10}$$

Once again, assuming $\phi > 0$, we can think of the expectation as the probability that no arrivals occur until time τ for a Poisson process with intensity $\phi(W_t)$, given τ and W_τ, and we can use a thinning method as before.

Thus, a proposal pair (τ, W_τ) is accepted as a sample of (ζ, Y_ζ) if no arrivals of V_t occur during the interval $[0, \tau]$, and if $U < \exp(A(W_\tau))/K$, with U uniformly distributed in $[0, 1]$.

Once a proposal pair is accepted, we select a new level θ, this time such that $[Y_\zeta - \theta, Y_\zeta + \theta] \subset \text{int } D_Y$ and repeat the previous steps. We continue until $\tau \geq T$, where we will instead consider a proposal pair (τ, W_T) for (ζ, Y_T), with likelihood ratio

$$\exp(A(W_T))\mathbb{E}\left[\exp\left(-\int_0^T \phi(W_s)ds\right)\middle|\tau, W_T\right]. \tag{4.11}$$

4.3. Extension to jump diffusions

The extension of the algorithm to diffusions with jumps is based on the following observation:

LEMMA 4.3.1. *Let $\bar{Y}_t$ be a solution of*

$$d\bar{Y}_t = \mu_Y(\bar{Y}_t)dt + dW_t, \tag{4.12}$$

and let T_1 be the first jump time of Y_t. Then $\bar{Y}_t \overset{d}{=} Y_t$ for $0 \leq t < T_1$. Additionaly, we have that

$$Y_{T_1} = \bar{Y}_{T_1} + \Delta_Y(\bar{Y}_{T_1}, Z_1). \tag{4.13}$$

Define $\zeta := \inf\{t \geq 0 : |\bar{Y}_t| \geq \theta\}$. If $\zeta < T_1$, then Y_ζ and $\bar{Y}_\zeta$ are equal in distribution, and we can sample directly from $\bar{Y}$ instead. On the other hand, if a jump occurs before ζ, equation (4.13) allows us to compute Y_{T_1} from $\bar{Y}_{T_1}$. Thus, we only need to be more incisive on the sampling of jump times.

Suppose we generated pair (τ, W_τ), with $\tau \leq T$, for some level θ, as in section 4.1. To determine if a jump occurs before τ, we use the same thinning argument as we did in section 4.2.

By localizing as before, we guarantee the existence of $\lambda > 0$ such that $\Lambda(F^{-1}(y)) \leq \lambda$ for all $y \in [-\theta, \theta]$, as Λ is assumed locally bounded. Thus, we can generate candidate jump times ν_1, ν_2, ..., ν_a from a Poisson process with intensity λ, and accept a candidate ν_n with probability $\Lambda(F^{-1}(W_{\nu_n}))/\lambda$, with W_{ν_n} drawn from a Brownian meander given (τ, W_τ).

If no jump time is accepted, we generated a skeleton $(\tau, W_{\nu_1}, ..., W_{\nu_a}, W_\tau)$; otherwise, if ν_l is the first jump time, we consider $(\nu_l, W_{\nu_1}, ..., W_{\nu_l})$ instead. These skeletons, composed of multiple frames, fall outside of the scope of the previously presented acceptance tests, and we show on the next section how one such test can be derived.

4.4. Acceptance test for Brownian skeletons

As the acceptance test we want to derive has Brownian paths as candidates, we are interested in the likelihood ratio between the true measure and a martingale measure. We will consider the more general scenario where, fixing $Y_s = y$ for some $s \geq 0$, we want to sample $\bar{Y}_t$, from $t = s$ up to the exit time of the interval $[y - \theta, y + \theta]$. For simplicity sake, we take $D_Y = [\underline{y}, +\infty)$ for some $\underline{y}$, θ is such that $\underline{y} < y - \theta$ and the exit time is $\zeta := \inf\{t \geq 0 : |\bar{Y}_t - y| \geq \theta\}$. Notice that, for $u \in [s, t \wedge \zeta]$, we have that $\bar{Y}_u \in [y - \theta, y + \theta] \subset D_Y$. Thus, by the local boundedness of μ_Y, there is $M_Y > 0$ such that $|\mu_Y| \leq M_Y$ in this interval.

In order to illustrate the usefulness and versatility of Girsanov's theorem, we need to endure some more formality. Let $(\bar{\mathcal{F}}_t)_{t\geq 0}$ be the filtration generated by $\bar{Y}_t$, and let $\bar{\mathbb{P}} = \bar{\mathbb{P}}_{(y;s,t)}$ be the probability measure induced by $\{\bar{Y}_{u\wedge\zeta}, s \leq u \leq t \wedge \zeta\}$ on the stopped sigma-algebra $\bar{\mathcal{F}}_{t\wedge\zeta}$. By Girsanov's theorem, there exists an equivalent measure $\mathbb{Q} = \mathbb{Q}_{(y;s,t)}$ under which $\{\bar{Y}_{u\wedge\zeta}, s \leq u \leq t \wedge \zeta\}$ is the path of a standard Brownian motion.

In fact, consider the supermartingale Z_t, defined by

$$Z_t = \exp\left(-\int_s^{t\wedge\zeta} \mu_Y(\bar{Y}_u)dW_u - \frac{1}{2}\int_s^{t\wedge\zeta} \mu_Y^2(\bar{Y}_u)du \right). \tag{4.14}$$

Again, by the local boundedness assumption, $Z_t > 0$ almost surely and Novikov's condition (2.1.1) holds, since

$$\mathbb{E}\left[\frac{1}{2}\exp\left(\int_s^t \mu_Y^2(\bar{Y}_u)du\right)\right] \leq \exp\left(\frac{1}{2}M_Y^2(t-s)\right) < +\infty, \tag{4.15}$$

and we conclude that Z_t is indeed a martingale. Now let $\mathbb{Q}$ be the equivalent measure to $\bar{\mathbb{P}}$ defined by $d\mathbb{Q} = Z_t d\bar{\mathbb{P}}$. By Girsanov's theorem, the process

$$\bar{W}^{\mathbb{Q}}_{t\wedge\bar{\tau}} = W_{t\wedge\zeta} + \int_s^{t\wedge\zeta} \mu_Y(\bar{Y}_u)du = \bar{Y}_{t\wedge\zeta} \tag{4.16}$$

is a standard Brownian motion under $\mathbb{Q}$, started at $\bar{W}^{\mathbb{Q}}_s = Y_s = y$, and with

$$\bar{\tau} = \inf\{t \geq s : |\bar{W}^{\mathbb{Q}}_t - y| \geq \theta\} = \zeta. \tag{4.17}$$

Thus, $\mathbb{Q}$ is exactly the martingale measure we were looking for.

Now let $B \in \bar{\mathcal{F}}_{t\wedge\zeta}$. Since

$$\frac{d\bar{\mathbb{P}}}{d\mathbb{Q}} = \frac{1}{Z_t}, \tag{4.18}$$

we have that

$$\bar{\mathbb{P}}(B) = \mathbb{E}^{\bar{\mathbb{P}}}[\mathbf{1}_B] = \mathbb{E}^{\mathbb{Q}}\left[\mathbf{1}_B \frac{1}{Z_t}\right] = \mathbb{E}^{\mathbb{Q}}\left[\frac{1}{Z_t}\middle| B\right]\mathbb{Q}(B), \tag{4.19}$$

giving us

$$\frac{\bar{\mathbb{P}}(B)}{\mathbb{Q}(B)} = \mathbb{E}^{\mathbb{Q}}\left[\frac{1}{Z_t}\middle| B\right]. \tag{4.20}$$

What is left for us to compute is an expression for $1/Z_t$ in terms of $\bar{W}^{\mathbb{Q}}_{t\wedge\zeta}$, as this is what we will be sampling first.

From equation (4.16), we have

$$\frac{1}{Z_t} = \exp\left(\int_s^{t\wedge\bar{\tau}} \mu_Y(\bar{W}^{\mathbb{Q}}_u)(d\bar{W}^{\mathbb{Q}}_u - \mu_Y(\bar{W}^{\mathbb{Q}}_u)du) + \frac{1}{2}\int_s^{t\wedge\zeta} \mu_Y^2(\bar{W}^{\mathbb{Q}}_u)du \right) \tag{4.21}$$

$$= \exp\left(\int_s^{t\wedge\bar{\tau}} \mu_Y(\bar{W}^{\mathbb{Q}}_u)d\bar{W}^{\mathbb{Q}}_u - \frac{1}{2}\int_s^{t\wedge\bar{\tau}} \mu_Y^2(\bar{W}^{\mathbb{Q}}_u)du \right). \tag{4.22}$$

The stochastic integral against $d\bar{W}^{\mathbb{Q}}_u$ poses another challenge, as it is not easily computable. However, by applying Itô's lemma to $A(\bar{W}^{\mathbb{Q}}_{t\wedge\bar{\tau}})$, again with $A(y) = \int_0^y \mu_Y(u)du$, we notice that

$$dA(\bar{W}^{\mathbb{Q}}_{t\wedge\bar{\tau}}) = \frac{1}{2}\mu_Y'(\bar{W}^{\mathbb{Q}}_{t\wedge\bar{\tau}})dt + \mu_Y(\bar{W}^{\mathbb{Q}}_{t\wedge\bar{\tau}})d\bar{W}^{\mathbb{Q}}_{t\wedge\bar{\tau}}, \tag{4.23}$$

from which we derive

$$A(\bar{W}^{\mathbb{Q}}_{t\wedge\bar{\tau}}) = A(\bar{W}^{\mathbb{Q}}_{s\wedge\bar{\tau}}) + \int_s^{t\wedge\bar{\tau}} \mu_Y(\bar{W}^{\mathbb{Q}}_u)d\bar{W}^{\mathbb{Q}}_u + \frac{1}{2}\int_s^{t\wedge\bar{\tau}} \mu_Y'(\bar{W}^{\mathbb{Q}}_u)du. \tag{4.24}$$

Finally, with ϕ defined as in equation (4.8), we have

$$\frac{1}{Z_t} = \exp\left(A(\bar{W}^{\mathbb{Q}}_{t\wedge\bar{\tau}}) - A(\bar{W}^{\mathbb{Q}}_s) - \int_s^{t\wedge\bar{\tau}} \phi(\bar{W}^{\mathbb{Q}}_u)du\right). \tag{4.25}$$

By the translation property, $W^{\mathbb{Q}}_t := \bar{W}^{\mathbb{Q}}_t - y$ is also a Brownian motion under $\mathbb{Q}$, starting at $W^{\mathbb{Q}}_s = 0$, and $\tau = \inf\{t \geq s : |W^{\mathbb{Q}}_t| \geq \theta\} = \bar{\tau}$ holding almost surely. We have then proved the following:

PROPOSITION 4.4.1. *Suppose X_t is a jump diffusion satisfying the assumptions described in section 2.4. Then, for any event $B \in \bar{\mathcal{F}}_{t\wedge\zeta}$ we have the formula*

$$\frac{\bar{\mathbb{P}}(B)}{\mathbb{Q}(B)} = \mathbb{E}^{\mathbb{Q}}\left[\exp\left(A(y + W^{\mathbb{Q}}_{t\wedge\tau}) - A(y) - \int_s^{t\wedge\tau} \phi(y + W^{\mathbb{Q}}_u)du\right)\middle| B\right], \tag{4.26}$$

with $W^{\mathbb{Q}}_t$ being a Brownian motion under $\mathbb{Q}$ starting at $W^{\mathbb{Q}}_s = 0$, and $\tau = \inf\{t \geq s : |W^{\mathbb{Q}}_t| > \theta\}$.

As in section 4.3, suppose we have generated either a skeleton $(\tau, W_{\nu_1}, ..., W_{\nu_a}, W_\tau)$, if $\tau > \nu_l$, or $(\nu_l, W_{\nu_1}, ..., W_{\nu_l})$ otherwise. By letting $\eta = \min\{\tau, \nu_l, T - s\}$, we can just write $(\eta, W_{\nu_1}, ..., W_\eta)$ for the proposal skeleton, and our target skeleton is now $(\eta, Y_{\nu_1} - Y_s, ..., Y_{s+\eta^-} - Y_s)$. By the previous proposition, the likelihood ratio between our pair is proportional to

$$\mathcal{L} := \exp(A(Y_s + W_\eta))\mathbb{E}\left[\exp\left(-\int_0^\eta \phi(Y_s + W_u)du\right)\middle|\eta, W_{\nu_1}, ..., W_\eta\right], \tag{4.27}$$

where the expectation can again be interpreted as a probability of no arrivals. This requires that $\phi > 0$; however, we can ignore this restriction by rescaling. By our assumptions on the coefficients, $\phi(y)$ is bounded in $[Y_s - \theta, Y_s + \theta]$, and a minimum m and a maximum M exist. Thus, we can instead consider

$$\mathcal{L} = \exp(A(Y_s + W_\eta))\exp(-m\eta)\mathbb{E}\left[\exp\left(-\int_0^\eta (\phi(Y_s + W_u) - m)du\right)\middle|\eta, W_{\nu_1}, ..., W_\eta\right]. \tag{4.28}$$

This factorization of the likelihood vastly simplifies the rejection sampling. Notice that we can interpret it as three independent Bernoulli events. First, as we have

$$0 \leq \phi(Y_s + W_u) - m \leq M - m, \tag{4.29}$$

we can use the same previously used thinning principle to avoid computing the expectation, and reject the sample if arrivals do occur. Additionally, as the other two factors are bounded by $K := \max_{u\in[-\theta,\theta]} \exp(A(Y_s + u))$ and $S := \max\{\exp(-m(T - s)), 1\}$, we can directly generate the three Bernoulli variables with the required probabilities.

4.5. General algorithm for jump diffusion sampling

We are now in conditions of stating the general sampling algorithm for a one-dimensional jump diffusion Y_t. Suppose its domain is $D_Y = (\underline{y}, \bar{y})$.

Algorithm 6: Jump Diffusion sampling

Result: Generates Y_T from a unit-volatility jump diffusion

Set $n = 1$, $y = Y_0 = 0$, $s_0 = 0$;
while $s_n < T$ **do**
  Choose $\theta_n > 0$ such that $\underline{y} + \theta_n < y < \bar{y} - \theta_n$;
  Generate $\tau = \inf\{t : |W_t| \geq \theta_n\}$;
  Choose $\lambda > 0$ such that $\lambda > \Lambda(F^{-1}(y+z))$, for $|z| \leq \theta_n$;
  Generate jump times $\nu_1 < ... < \nu_a \leq \tau \wedge (T - s_{n-1})$ of a Poisson process with rate λ;
  Generate jump times $\kappa_1 < ... < \kappa_b \leq \tau \wedge (T - s_{n-1})$ of a Poisson process with rate $M - m$, with M and m being the maximum and the minimum of $\phi(y+z)$, $|z| \leq \theta_n$;
  Sample $(W_{\nu_1}, ..., W_{\nu_a}, W_{\kappa_1}, ..., W_{\kappa_b}, W_\tau, W_{\tau \wedge (T - s_{n-1})})$;
  Set $i = 1$;
  while $i \leq a$ **do**
    Draw $U_i \sim U(0,1)$;
    if $U_i \leq \Lambda(F^{-1}(y + W_{\nu_i}))/\lambda$ **then**
      Set $l = i$;
      Leave while loop;
    else
      Set $i = i + 1$;

(17)

(18) **if** $i = a + 1$ **then**

(19) Accept/reject the proposal skeleton $(s_{n-1} + \tau, y + W_{\nu_1}, ..., y + W_{\nu_a}, y + W_{\tau\wedge(T-s_{n-1})})$ as a sample of the skeleton $(\zeta_n, Y_{s_{n-1}+\nu_1}, ..., Y_{s_{n-1}+\nu_a}, Y_{(s_{n-1}+\tau)\wedge T})$ using the arrival times κ_j;

(20) **if** *the proposal is accepted and* $T \leq s_{n-1} + \tau$ **then**

(21) Return Y_T;

(22) **if** *the proposal is accepted and* $s_{n-1} + \tau < T$ **then**

(23) Set $y = Y_{s_{n-1}+\tau}$, $s_n = s_{n-1} + \tau$, $n = n + 1$;

(24) **else**

(25) Return to step 2;

(26) **else**

(27) Accept/reject the proposal skeleton $(s_{n-1} + \tau, y + W_{\nu_1}, ..., y + W_{\nu_l})$ as a sample of the skeleton $(s_{n-1} + \nu_l, Y_{s_{n-1}+\nu_1}, ..., Y_{s_{n-1}+\nu_{l-1}}, Y_{(s_{n-1}+\nu_l)^-})$ using the arrival times κ_j;

(28) **if** *the proposal is accepted* **then**

(29) Sample Z from Π;

(30) Set $s_n = s_{n-1} + \nu_l$, $y = Y_{(s_{n-1}+\nu_l)^-} + \Delta_Y(Y_{(s_{n-1}+\nu_l)^-}, Z)$, $n = n + 1$;

(31) **else**

(32) Return to step 2;

CHAPTER 5

Computational efficiency and implementation

This chapter is dedicated to discussing topics relating to computational efficiency and implementation of the algorithm. More precisely, we will first specify a method for picking θ in order to maximize efficiency, and will then discuss extensions of the algorithm in order to adapt it to a number of situations.

5.1. Level selection

The constraints on θ in the previous algorithms are very non-stringent, and we thus have some freedom in this choice. The authors of [20] propose the number of skeletons generated before reaching the desired horizon T as a measure of efficiency, as minimizing the number of skeletons might lead to a very short time increment, and maximizing the time increment might require many tries at generating skeletons until one is accepted. As such, at each step n, we pick a value of θ that maximizes the time increment per skeleton generated instead.

While this measure might seem challenging to compute, we can establish favourable lower bounds, and the tower property of iterated expectations allows us to vastly simplify the problem.

First, let $\mathcal{K} = \mathcal{K}_n(\theta)$ be the number of proposal skeletons generated before one is accepted during step n. This a random variable that, given Y_s, depends on θ and s, but not on previous level choices θ_j, $j \neq n$, by the strong Markov property of Y, and the same is true for the time increment η. Furthermore, given τ, ν_l, W_η and Y_s, $\mathcal{K}$ has a simple geometric distribution with success parameter $p(\theta, s; \tau, \nu_l, W_\eta, Y_s)$, the conditional probability of accepting a skeleton, and the normalized likelihood (4.28) we computed in section 4.4 gives us the expression for this parameter, conditional on τ, ν_l, W_{ν_1}, ..., W_η and Y_s:

$$\begin{aligned} p(\theta, s; \tau, \nu_l, W_{\nu_1}, ..., W_\eta, Y_s) =& \frac{\exp(A(Y_s + W_\eta))}{K} \frac{\exp(-m\eta)}{S} \\ & \times \mathbb{P}(V_\eta - V_0 = 0 | \tau, \nu_l, W_{\nu_1}, ..., W_\eta, Y_s), \end{aligned} \tag{5.1}$$

with V being a doubly stochastic Poisson process with intensity $\phi(Y_s + W_t) - m$.

In these terms, our efficiency measure $\mathcal{M}$ becomes

$$\mathcal{M}(\theta, s, Y_s) := \mathbb{E}[\eta / \mathcal{K} | Y_s]. \tag{5.2}$$

Unfortunately, the expectation above is hard to compute, especially without further assumptions on the dynamics of Y. As our goal is to maximize this measure, let us derive a lower bound on this value, more explicitly computable.

Now fix θ and s. By the properties of the geometric distribution, we have that

$$\mathbb{E}[\mathcal{K}|\tau, \nu_l, W_\eta, Y_s] = 1/p(\theta, s; \tau, \nu_l, W_\eta, Y_s), \tag{5.3}$$

and, by the tower property of conditional expectations, we also have

$$\mathcal{M}(\theta, s, Y_s) = \mathbb{E}[\eta \mathbb{E}[1/\mathcal{K}|\tau, \nu_l, W_\eta, Y_s]|Y_s]. \tag{5.4}$$

With an application of Jensen's inequality, due to the convexity of $\frac{1}{x}$, we obtain our first lower bound:

$$\mathcal{M}(\theta, s, Y_s) \geq \mathbb{E}[\eta/\mathbb{E}[\mathcal{K}|\tau, \nu_l, W_\eta, Y_s]|Y_s] = \mathbb{E}[\eta p(\theta, s; \tau, \nu_l, W_\eta, Y_s)|Y_s]. \tag{5.5}$$

On the other hand, as in the thinning argument we have used before, V is dominated by a Poisson process with intensity $M - m$, giving us

$$\begin{aligned} p(\theta, s; \tau, \nu_l, W_{\nu_1}, ..., W_\eta, Y_s) &\geq \frac{\exp(A(Y_s + W_\eta))}{K} \frac{\exp(-m\eta)}{S} \exp(-(M-m)\eta) \\ &= \exp(A(Y_s + W_\eta) - M\eta)/(KS). \end{aligned} \tag{5.6}$$

Thus, once again making use of the tower property to introduce more information, we have

$$\begin{aligned} \mathcal{M}(\theta, s, Y_s) \geq& \mathbb{E}[\eta p(\theta, s; \tau, \nu_l, W_\eta, Y_s)|Y_s] \\ =& \mathbb{E}[\eta p(\theta, s; \tau, \nu_l, W_{\nu_1}, ..., W_\eta, Y_s)|Y_s] && (5.7) \\ \geq& \mathbb{E}[\eta \exp(A(Y_s + W_\eta) - M\eta)|Y_s]/(KS) && (5.8) \\ =& \frac{1}{KS} \int \min(t, x, T-s) \exp(A(Y_s + \omega) - M \min(t, x, T-s)) \\ & \times \mathbb{P}(\tau \in dt, \nu_l \in dx, W_\eta \in d\omega|Y_s). && (5.9) \end{aligned}$$

The inequality of most interest is (5.9), and we will now turn to the problem of estimating this lower bound. Note that this choice of lower bound and approximations introduced hereafter *do not* induce any bias, as they pertain only to the choice of a more efficient value of θ.

In order to approximate the joint distribution of (τ, ν_l, W_η) given Y_s, we first notice that we can factor this measure as

$$\begin{aligned} \mathbb{P}(\tau \in dt, \nu_l \in dx, W_\eta \in d\omega|Y_s) &= \mathbb{P}(W_\eta \in d\omega|\tau = t, \nu_l = x, Y_s)\mathbb{P}(\tau \in dt, \nu_l \in dx|Y_s) \\ &= \mathbb{P}(W_\eta \in d\omega|\tau = t, \nu_l = x, Y_s)\mathbb{P}(\nu_l \in dx|Y_s)\mathbb{P}(\tau \in dt). \end{aligned} \tag{5.10}$$

While the distribution for the exit time of a Brownian motion, $\mathbb{P}(\tau \in dt) = h(t)dt$, was discussed in section 3.4.1, the distribution of the jump time $\mathbb{P}(\nu_l \in dx|Y_s)$ is not easily computable. However, if θ is not very large, we assume that Y jumps approximately as a Poisson process with the largest intensity in this segment:

$$c(Y_s) := \max_{|z|\leq\theta} \Lambda(F^{-1}(Y_s + z)), \tag{5.11}$$

which gives

$$\mathbb{P}(\nu_l \in dx|Y_s) \approx c\exp(-cx)dx. \tag{5.12}$$

Finally, as for the distribution of W_η, if $\min(t, x, T-s) = t$, we have that $\mathbb{P}(W_\tau = \theta) = \mathbb{P}(W_\tau = -\theta) = 1/2$. If $\min(t, x, T-s)$ is either x or $T-s$, the conditional distribution is more convoluted, and thus we approximate it by the true mean of the distribution, $\delta(\omega)d\omega$. All together, we then write

$$\mathbb{P}(W_\eta \in d\omega|\tau = t, \nu_l = x, Y_s) = \begin{cases} \frac{1}{2}(\delta(\omega+\theta) + \delta(\omega-\theta))d\omega, & \text{if } \min(t, x, T-s) = t \\ \approx \delta(\omega)d\omega, & \text{if } \min(t, x, T-s) = x, T-s \end{cases} \tag{5.13}$$

With these facts in mind, and recalling equation (5.9), our estimator $\tilde{\mathcal{M}}_L(\theta, s, Y_s)$ for the lower bound can be defined as

$$\begin{aligned} \tilde{\mathcal{M}}_L(\theta, s, Y_s) =& \frac{c}{KS}\int_0^{T-s}\left(\int_0^t x\exp(A(Y_s) - Mx - cx)dx\right)h(t)dt \\ &+\frac{c}{KS}\int_0^{T-s}\left(\int_t^{+\infty}\frac{1}{2}(e^{A(Y_s+\theta)} + e^{A(Y_s-\theta)})e^{-cx}dx\right)e^{-Mt}th(t)dt \\ &+\frac{c}{KS}\int_{T-s}^{+\infty}\left(\int_0^{T-s} x\exp(A(Y_s) - Mx - cx)dx\right)h(t)dt \\ &+\frac{c}{KS}\int_{T-s}^{+\infty}\left(\int_{T-s}^{+\infty}\exp(A(Y_s) - M(T-s) - cx)dx\right)(T-s)h(t)dt. \end{aligned} \tag{5.14}$$

It is important to note that most of the dependency in θ is hidden in the above expression, as M, c and τ are functions of this choice. Apart from this fact, the estimation involves only computing the above integrals and maximizing with respect to θ. The authors suggest to ignore the dependency in s, as this computation is costly and we avoid performing it at the beginning of each step, and compute only for a range of values $y \in D_Y$, at $s = 0$. During the algorithm execution, we can then interpolate between the optimal values $\theta^*(y)$ previously determined. The integration itself can be performed efficiently with some quadrature method.

We now turn to implementing the procedure above. Here, we used h to denote the probability density function of $\tau = \tau_\theta$, whereas in section 3.4.1 we only evaluated the density of τ_1. If we now let h denote this density previously studied, by the change of variables formula and using the fact that $\tau_\theta \overset{d}{=} \theta^2\tau_1$ we see that

$$\mathbb{P}(\tau \in dt) = \frac{1}{\theta^2}h(t/\theta^2)dt. \tag{5.15}$$

Setting $Y_s = y$ and $s = 0$, notice that we can factor some terms and compute all of the integrals in terms of x exactly. With some manipulation, we then have that

$$\begin{aligned}\tilde{\mathcal{M}}_L(\theta, 0, y) =& \frac{c\exp\left(A(y)\right)}{KS(M+c)^2\theta^2}\left(\int_0^T h\left(\frac{t}{\theta^2}\right)dt - (M+c)\int_0^T te^{-(M+c)t}h\left(\frac{t}{\theta^2}\right)dt\right. \\ &\left. - \int_0^T e^{-(M+c)t}h\left(\frac{t}{\theta^2}\right)dt\right) \\ &+ \frac{\exp(A(y+\theta)) + \exp(A(y-\theta))}{2KS\theta^2}\int_0^T e^{-Mt}th\left(\frac{t}{\theta^2}\right)dt \\ &+ \frac{c\exp(A(y))}{KS}\left(1 - \frac{1}{\theta^2}\int_0^T h\left(\frac{t}{\theta^2}\right)dt\right)\left(\frac{1-e^{-(M+c)T}}{M+c} + \frac{Te^{-(M+c)T}}{c}\right).\end{aligned} \tag{5.16}$$

This reduces the amount of integrals we have to compute, and they are all now defined over a finite interval. For practical purposes, we employ the `scipy` package, which allows us to compute both the required optimizers via the `optimize.minimize_scalar` function and integrals using `integrate.quad` at each step, as well as the optimal θ.

All that is left to do is optimize the target function above over a list of values of y, spread out through the domain D_Y. Linear interpolation for values of y outside this list suffices to determine a choice of θ.

5.1.1. Convergence

While we now have a way to quantify the computational efficiency of our algorithm and maximize the expected time increment per skeleton generated, we have to be cautious about the nature of $\eta_1 + \eta_2 + ...$, and ensure that there is some kind of convergence. Fortunately, in [20] the authors manage to provide conditions that guarantee that any finite horizon can be reached in a finite amount of steps.

PROPOSITION 5.1.1. *Suppose the jump-diffusion X satisfies the assumptions described in 2.4. In addition, suppose that $\theta_n = \theta(Y_{s_{n-1}})$ for a deterministic function θ, and that one of the following holds:*

- *$D_Y = (-\infty, +\infty)$ and there are $\underline{\theta} < \overline{\theta}$ such that $\theta_n \in [\underline{\theta}, \overline{\theta}]$ for all n; or*
- *$D_Y = (\underline{y}, +\infty)$ and there exist $\varepsilon > 0$, $0 < \alpha < 1$, $\underline{\theta}$, $\overline{\theta}$ such that $\theta_n(y) = \alpha(y - \underline{y})$ for $y \in (\underline{y}, \underline{y} + \varepsilon)$, $\theta_n(y) \leq \alpha(y - \underline{y})$ for $y \geq \underline{y} + \varepsilon$, $\theta_n(y) \in [\underline{\theta}, \overline{\theta}]$ for $y \geq \underline{y} + \varepsilon$.*

Then, for any finite time horizon T,

$$\mathbb{P}\left[\sum_{n=1}^{+\infty} \eta_n < T\right] = 0. \tag{5.17}$$

The proof is shown in the appendix of the original paper due to its length.

Let us, however, examine the assumptions of the proposition regarding the choice function. As in the method described before, we assume that the choice at step n depends only on the value of Y at the previous time s_{n-1}.

In the case where the domain is the real line, we only have to further assume that our choice function is uniformly bounded, that is, there is a minimum and maximum step size, independent of n.

If our domain is restricted by some lower bound (say, zero when our process is strictly positive), we need to restrict the behaviour near the boundary as well. Close enough to $\underline{y}$, the step size choice has to be a fixed fraction of the current distance from it.

Now consider the set $E = \{\omega \in \Omega | \sum_{n=1}^{+\infty} \eta_n(\omega) < T\}$, for a fixed time horizon T. Since, for each $\omega \in \Omega$, we have that $\eta_n(\omega) < T - s_{n-1}$ for all n, otherwise the sum would be larger than T. Thus, we assume without loss of generality that

$$\eta_n = \min(\tau_n(\theta_n), \nu_l, T - s_{n-1}) = \min(\tau_n(\theta_n), \nu_l^n). \tag{5.18}$$

The goal is then to show that $\mathbb{P}(E) = 0$. To do this, the authors define the sets

$$E_k = \left\{\omega \in \Omega : \eta_k(\omega) < \frac{1}{k}\right\}, \tag{5.19}$$

so that $\mathbb{E} \subseteq \limsup_k E_k$, and thus $\mathbb{P}(E) = \mathbb{P}(E \cap \limsup_k E_k)$. Furthermore, by using a localization argument, it also suffices to show that

$$\mathbb{P}(E \cap \limsup_k E_k \cap B_m) = 0 \tag{5.20}$$

for each $m \in \mathbb{N}$, where B_m is the set of events where Y_t is contained in the localized interval $[l_m, r_m]$, converging monotonously to the entire domain D_Y. This observation allows us to use the local boundedness assumptions on the function parameters defining the process. The final observation needed before the proof becomes largely computational, is that

$$\mathbb{P}(E \cap \limsup_k E_k \cap B_m) = \mathbb{P}(\limsup_k (E \cap E_k \cap B_m)) \leq \mathbb{P}\left(\bigcup_{n=k}^{+\infty} (E \cap E_n \cap B_m)\right), \tag{5.21}$$

for each k. Thus, if

$$\mathbb{P}\left(\bigcup_{n=k}^{+\infty} (E \cap E_n \cap B_m)\right) \xrightarrow[k \to +\infty]{} 0, \tag{5.22}$$

the result is proven. As previously noted, we refer to the appendix of the original paper for this computation.

5.2. Extensions

In this section, we will discuss how the algorithm can be further extended in order to encompass a larger amount of situations.

Before we move on to more delicate questions regarding the algorithm itself, note that we omitted many dependencies for the sake of simplicity, but due to the sequential nature of the algorithm have no influence and could easily be included, with immediate adaptations. For instance, the jump intensity may also be a function of time, the current state of the process and the number and size of previous jumps. This allows to model conditions like seasonal behaviour or trading restrictions.

5.2.1. Sampling a skeleton

Let S be a discrete set of fixed times in $[0, T]$, and suppose that we wish to generate the values $(X_t)_{t\in S}$ (or $(Y_t)_{t\in S}$, equivalently), e.g. when dealing with discretely monitored options.

To do this, we only have to additionally sample the values $(W_t)_{t\in S}$ alongside with the W_{ν_j} and W_{κ_j}. These values do not interfere with the likelihood (4.28), as no $t \in S$ is equal to a proposed jump time a.s., and thus we can use the same acceptance test as before.

Therefore, there is little to none extra computational burden when sampling for a single value at the horizon T or for a range of values in-between.

5.2.2. Sampling hitting times

There are times when it is useful to keep track of certain hitting times, such as when considering the possibility of a defaulting firm or when dealing with barrier options. The versatility of the algorithm makes it very easy to adapt to this type of problem, with an adequate choice of θ.

Suppose we want to generate a sample of $\xi \wedge T$, for $\xi = \xi(x_d, x_u) = \inf\{t \geq 0 : X_t \notin (x_u, x_d)\}$. If $\theta_n(y)$ is the choice of θ described in 5.1, then at each step n we can take

$$\theta_n^*(y) := \min(\theta_n(y), y - x_d, x_u - y). \tag{5.23}$$

If the process hits one of the barriers via drift, we immediately obtain the hitting time by the algorithm; if the process jumps through one of the barriers, we take the hitting time as the last jump time $T_n = s_{n-1} + \nu_l$.

5.2.3. Exponential of time-integrated jump diffusion

The algorithm may also be modified to allow the treatment of expectations of the type

$$B(T) = \mathbb{E}\left[\exp\left(-\int_0^T X_s ds\right) u((X_t)_{t\in S}, (J_t)_{t\leq T})\right], \tag{5.24}$$

under some interest rate model, for instance. Unfortunately, it is not possible for us to simulate a complete path of X_t, and using some discrete approximation to the integral $\int_0^T X_s ds$ will induce a bias.

Nevertheless, we can still obtain exact samples by making use of the special exponential structure. For this, take $u = 1$, so that we can interpret the expectation as the probability that no arrivals occur for a doubly stochastic Poisson process with intensity X_t. Note that X need not be positive, as we can re-scale the exponential, neither uniformly bounded.

In this case, if we have (localized) bounds $\underline{x}$, $\bar{X}$, we can sample jump times ε_k of a Poisson process with intensity $\bar{X} - \underline{x}$. Reasoning similarly to the thinning argument we previously used, we can show that

$$e^{-\underline{x}T} \prod_{k=1}^{n} \left(1 - \frac{X_{\varepsilon_k} - \underline{x}}{\bar{X} - \underline{x}}\right) \tag{5.25}$$

is an exact estimator for $\exp\left(-\int_0^T X_s ds\right)$. To see this, note that $\frac{X_{\varepsilon_k}-\underline{x}}{\bar{X}-\underline{x}}$ is the probability of accepting the k-th jump as a jump from our doubly stochastic process, and thus the product $\prod_{k=1}^{n}\left(1-\frac{X_{\varepsilon_k}-\underline{x}}{\bar{X}-\underline{x}}\right)$ is the probability of rejecting every jump, and thus that no arrivals occur, *for this particular path*.

We conclude that, for a general function u, our estimator of the value inside the expectation is simply

$$e^{-\underline{x}T}\prod_{k=1}^{n}\left(1-\frac{X_{\varepsilon_k}-\underline{x}}{\bar{X}-\underline{x}}\right)u((X_t)_{t\in S},(J_t)_{t\leq T}). \tag{5.26}$$

The algorithm can easily be adapted to account for these jump times ε_k, as they are generated independently from the other considered times, and have no part in the acceptance/rejection. We only have to be careful regarding our stopping time (whether it is an exit or jump time is irrelevant), and include the ε_k up to this instant at each step until we reach the horizon T.

5.3. Implementation notes

There are still a few details needed to be cleared up before we implement this method. One such question is how to determine the optimizing bounds needed to do the acceptance tests. In both cases, we need to compute $K := \max_{u\in[-\theta,\theta]}\exp(A(Y_s+u))$, $S := \max\{\exp(-m(T-s)),1\}$, constants m, M and λ such that $0\leq\phi(Y_s+z)-m\leq M-m$, and $\Lambda(F^{-1}(Y_s+z))\leq\lambda$ for all $z\in[-\theta,\theta]$.

For the JDCEV case, noting that F^{-1} is increasing and Λ decreasing, we can take λ as $\Lambda(F^{-1}(Y_s-\theta))$. Furthermore, as μ_Y is always positive, $A(y)$ is increasing, and we then have $K=\exp(A(Y_s+\theta))$.

The behaviour of the function $\phi=\frac{1}{2}(\mu_Y'+\mu_Y^2)$ is non-trivial but one can determine that it is monotonically increasing. Note that, close to the boundary of D_Y, the value of $M-m$ might become too large due to the singularity present and slow down computations. We use smaller values of θ near this point. Note that, in the JDCEV case, the default intensity becomes very large and we are more likely to accept a default and, in the AJD case, the drift ensures we stay away from the boundary.

For the AJD model, we can only skip the optimization of the value of λ, as its the only one we determine explicitly. The `scipy` package would allow us to determine the optimizers through the `optimize.minimize_scalar` function, we can make use of the `method="bounded"` option for an improve in performance. We may, however, avoid this computation. By observing that the functions $A(y)$ and $\phi(y)$ have a single extrema (a maximum and a minimum, respectively), we can efficiently determine optimizers in an interval $[y-\theta,y+\theta]$.

CHAPTER 6

Numerical results and conclusions

Now with a good theoretical understanding of the method, we move on to the numerical results. In this section, we implement the exact and discretization methods discussed previously, in order to replicate in Python the results obtained by Giesecke and Smelov. All code was programmed in Python 3.8 and computations were performed in a Intel(R) Core(TM) i9-10900 CPU computer, with 64 GB of RAM (63.7 accessible).

The exact method always has a bias of 0, while the biases of the discretization methods were computed by the authors of the method using either closed-form solutions when available, as is the case of the European put under the JDCEV model, or a large number of trials of the exact method to determine the true value, and performing 10 million trials of the discrete methods, given a number of steps, to estimate their expectations.

The choice of number of steps for the discrete methods follows as well the approach described by Duffie and Glynn in [16]. These methods rely on a good choice of number of steps. While increasing the number of trials reduces the standard error of the sample, the step size determines how good of an approximation the discrete grid is to the true process, and a smaller step size induces less bias. However, there is a trade-off between performance and bias, which does not happen with the exact method, and the authors determine that the optimal trade-off breakpoint is when the number of steps is equal to the square-root of the number of trials, for first order methods, and the fourth root for second order methods.

6.1. JDCEV

We present the results we obtained from our simulations, beginning with the JDCEV model. The chosen parameters, which we will also perturb, are the same as in [11]: $X_0 = 50$, $\beta = -1$, $r = 0.05$, $a = 50/4$, $b = 0$, and $c = 1/2$.

6.1.1. European option

First, we examine the behaviour of the algorithm when pricing a European put option with strike $K = 5$ and maturity $T = 1$ year. The true value of this contract is 0.1491. Table 1 showcases the obtained data under the previously defined parameters. While we were able to obtain a much better (absolute) performance for the exact method, the RMSE's converge in a similar fashion to Giesecke and Smelov's implementation.

Furthermore, as we expected, the exact method converges with the optimal square-root rate, while the discrete method is far slower. Note that, while we need to increase the computational budget non-linearly to increase precision for the discrete methods, the

exact method's computation time is linear in the number of simulations, so it will always eventually outperform.

Method	Trials	Steps	Value	Bias	SE	RMSE	Time (sec)
Exact	10K	N/A	0.1560	0	0.0089	0.0089	3.4
	20K	N/A	0.1603	0	0.0064	0.0064	6.77
	40K	N/A	0.1503	0	0.0044	0.0044	13.38
	100K	N/A	0.1552	0	0.0028	0.0028	34.11
	500K	N/A	0.1499	0	0.0012	0.0012	175.22
Discretization	10K	100	0.1379	0.0019	0.0084	0.0086	6.34
	20K	140	0.1523	0.0018	0.0062	0.0065	17.53
	40K	200	0.1519	0.0008	0.0044	0.0045	49.34
	100K	310	0.1523	0.0005	0.0028	0.0028	194.71
	500K	707	0.1516	0.0004	0.0012	0.0013	2187.45

TABLE 1. Simulation results under the JDCEV model for a European put option with strike price $K = 5$ and expiration date $T = 1$ year.

We may now observe how different parameter values impact performance. Figure 1 showcases our results for the convergence of RMSE's under different parameter values, perturbing only one at a time, while figure 2 contains Giesecke and Smelov's results for comparison purposes. [1]

The profiles obtained are very similar, and the square root convergence of the exact method is verified. The differences in performance impact are worth being discussed. For $c = 1$, $X_0 = 25$ or $\beta = -0.5$, the changes are noticeable, but the exact method still outperforms at (almost) every point in time.

However, for $b = 0.2$, the exact method is no longer able to overtake the discrete method over the tested numbers of iterations. This seems to imply that the improvement in performance from hardware is not strong enough to overcome the loss from increasing the value of b.

One possible explanation for this is as follows. On one hand, the discrete method is rather stable over parameter changes in terms of computational burden. There might be an impact in the error, but goes largely unnoticed. The exact method is much more sensitive, and the function ϕ has the most impact in performance, as the likelihood ratio is proportional to $e^{-\phi}$. Larger values of ϕ lead to a large amount of rejected samples, wasting a lot of computing time.

The value of 0.2 is also very large in relative terms, leading to such a large performance loss. Recall that, in the formula for μ_Y, b shows up added to the interest rate $r = 0.05$ and multiplied by some 'larger' factor. When it changes from 0 to 0.2, this factor is now 5 times bigger than before. This impact is much bigger than when we consider $X_0 = 25$,

[1]Note that the initial volatility does not remain constant. Given that $\sigma(X) = aX^\beta$, we first have a volatility of 25%. Increases in volatility lead to larger sample standard deviations and thus larger RMSE's.

where we are much closer to the singularity. This is not as big of a concern in practice, as this value is fairly unrealistic and inferred values range over much lower values (see, for instance, [15]).

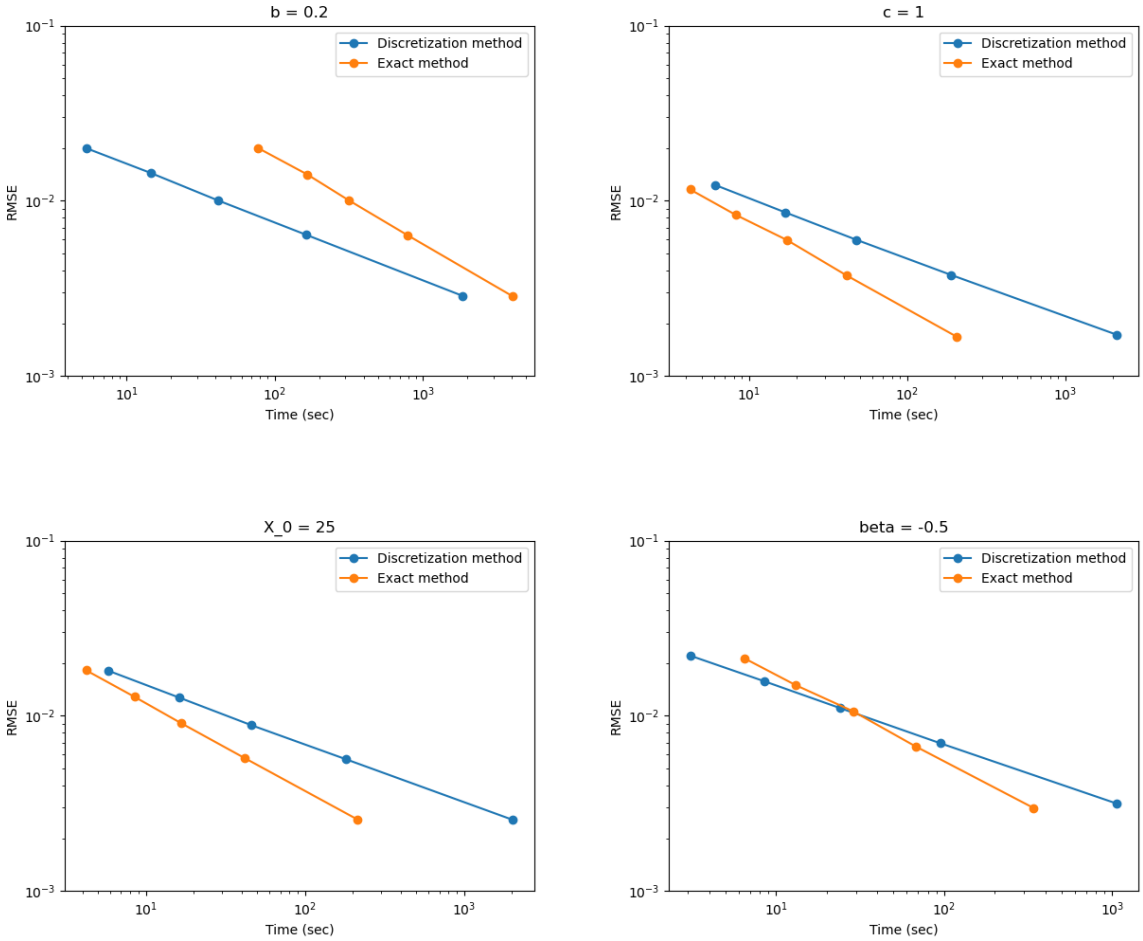

FIGURE 1. Convergence of RMSE's for the discrete and exact methods when one parameter is perturbed. The title of each figure indicates which parameter was changed and to which value.

6.1.2. Exotic options

We turn our attention to the pricing of exotic options, which is one of the main strengths of the exact method. If we move slightly away from simpler payoffs similar to those of European vanilla options, closed-form solutions are no longer available and controlling for bias is significantly more expensive.

Table 2 contains the results relative to the pricing of an Asian put with semiannual monitoring, strike $K = 5$ and maturity $T = 1$, whose true value is 0.0745, while table 3 shows the case of a down-and-out call with strike $K = 65$, down barrier $B = 5$ and maturity $T = 1$, whose true value is 1.2794.

We achieve performances similar to those of the European put, being much faster than the discrete method and obtaining RMSE's comparable to or somewhat smaller than the ones of the discretization.

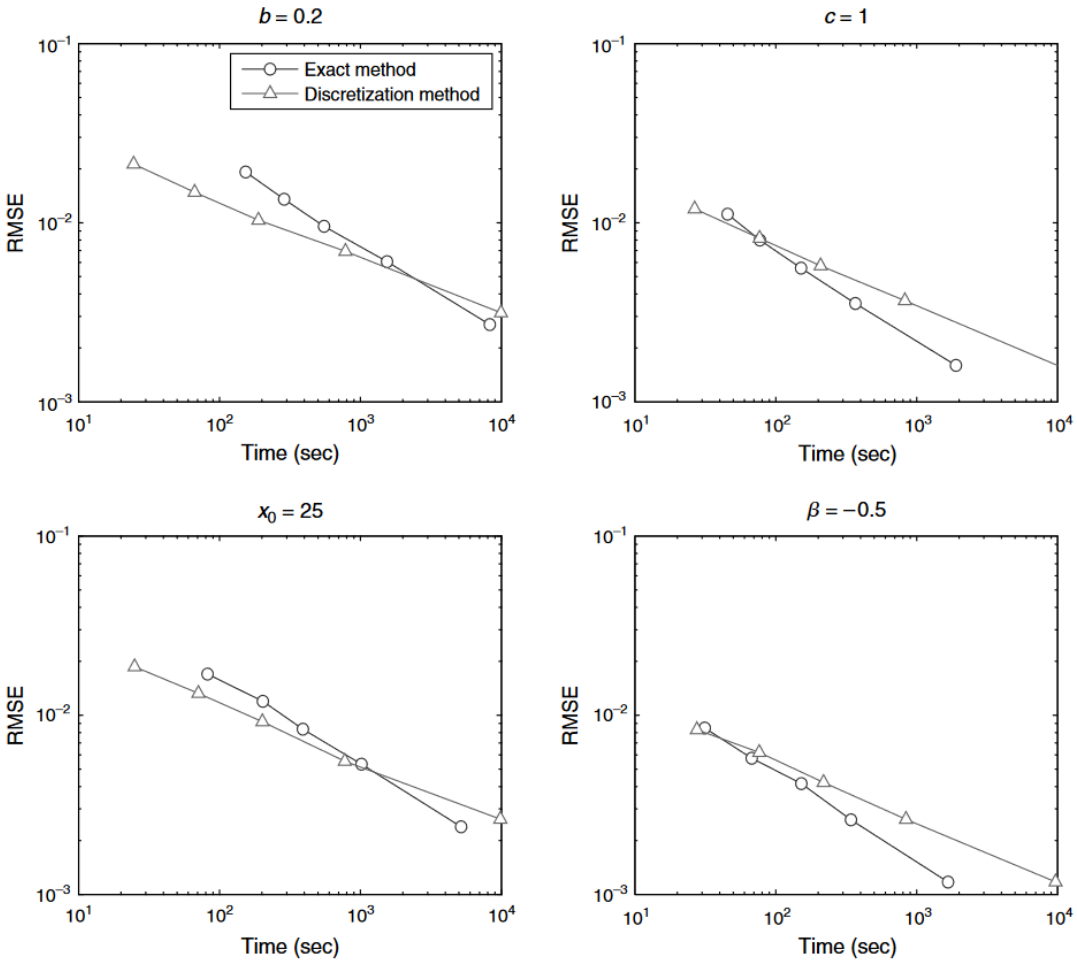

FIGURE 2. RMSE convergence profiles obtained by Giesecke and Smelov, taken directly from [20].

This fact, together with the previous sensitivity tests, showcase the importance of the method for efficient pricing of exotic products and complex payoffs.

Method	Trials	Steps	Value	Bias	SE	RMSE	Time (sec)
Exact	10K	N/A	0.0694	0	0.0060	0.0060	4.4
	20K	N/A	0.0749	0	0.0044	0.0044	9.69
	40K	N/A	0.076	0	0.0019	0.0019	18.92
	100K	N/A	0.0734	0	0.0028	0.0028	47.2
	500K	N/A	0.0744	0	0.0009	0.0009	235.8
Discrete	10K	100	0.0723	0.0014	0.0061	0.0063	6.46
	20K	140	0.0763	0.0008	0.0044	0.0045	18.26
	40K	200	0.0763	0.0005	0.0031	0.0031	50.94
	100K	310	0.0754	0.0004	0.0020	0.0020	197.4
	500K	707	0.0748	0.0002	0.0009	0.0009	2172.99

TABLE 2. Simulation results under the JDCEV model for a semiannually monitored Asian put option with strike price $K = 5$ and expiration date $T = 1$ year.

Method	Trials	Steps	Value	Bias	SE	RMSE	Time (sec)
Exact	10K	N/A	1.3104	0	0.0388	0.0388	3.72
	20K	N/A	1.301	0	0.0272	0.0272	7.47
	40K	N/A	1.2917	0	0.0191	0.0191	14.99
	100K	N/A	1.2924	0	0.0121	0.0121	37.42
	500K	N/A	1.2836	0	0.0054	0.0054	188.05
Discrete	10K	100	1.2691	0.0221	0.0384	0.0443	6.37
	20K	140	1.2356	0.0144	0.0257	0.0295	18.09
	40K	200	1.2814	0.0094	0.0189	0.0211	49.43
	100K	310	1.2862	0.0058	0.0120	0.0133	194.91
	500K	707	1.2793	0.0013	0.0053	0.0055	2180.75

TABLE 3. Simulation results under the JDCEV model for a down-and-out call option with strike price $K = 65$, down barrier $B = 5$ and expiration date $T = 1$ year.

6.2. Affine Jump Diffusions

We now analyze how the method performs when dealing with interest rate models, namely the AJD model, and how implementation can be adapted to account for time-integrated exponentials.

6.2.1. Zero coupon bond

First, we observe how the method behaves against the pricing of a zero coupon bond with maturity $T = 3$, with a true value of 0.879872. We will use again the same parameters as Giesecke and Smelov, estimated by Zhou from weekly observations of the US federal funds rate via a multivariate weighted nonlinear least square for jump diffusion (MWNLS-JD) estimator [38]. These values are $X_0 = \theta = 0.0422$, $\kappa = 0.0117$, $\sigma = 0.0130$, $\Lambda_0 = 0.0110$, $\Lambda_1 = 0.1000$, and the jump size is drawn from a uniform distribution $U(0.0113, 0.0312)$. The outcome of our experiments is presented in table 4.

The increase in performance is extremely noticeable, even when comparing to a fast second order method, and we thus achieve much lower values of RMSE for any given duration.

Method	Trials	Steps	Value	Bias	SE	RMSE	Time (sec)
Exact	10K	N/A	0.880702	0	1.312e-03	1.312e-03	2.12
	20K	N/A	0.879736	0	9.374e-04	9.374e-04	4.19
	40K	N/A	0.880501	0	6.600e-04	6.600e-04	8.31
	100K	N/A	0.879894	0	4.204e-04	4.204e-04	20.45
	500K	N/A	0.879712	0	1.886e-04	1.886e-04	103.5
Discrete I	10K	100	0.878773	1.137e-03	9.997e-05	1.141e-03	5.6
	20K	140	0.879017	8.080e-04	7.060e-05	8.111e-03	15.6
	40K	200	0.879271	5.637e-04	4.984e-05	5.659e-04	44.1
	100K	310	0.879485	3.659e-04	3.114e-05	3.672e-04	175.03
	500K	707	0.879733	1.677e-04	1.390e-05	1.683e-04	1941.23
Discrete II	10K	100	0.878851	1.141e-03	9.811e-05	1.145e-03	6.41
	20K	140	0.879061	8.056e-04	7.029e-05	8.087e-04	17.84
	40K	200	0.879198	5.673e-04	5.032e-05	5.695e-04	50.12
	100K	310	0.879514	3.661e-04	3.121e-05	3.674e-04	197.94
	500K	707	0.879715	1.652e-04	1.390e-05	1.658e-04	2209.05
Discrete III	10K	100	0.870707	1.131e-02	1.148e-04	1.131e-02	1.19
	20K	140	0.871682	9.435e-03	7.842e-05	9.435e-03	2.54
	40K	200	0.873880	8.094e-03	5.362e-05	8.094e-03	6.27
	100K	310	0.875158	6.300e-03	3.385e-05	6.300e-03	18.74
	500K	707	0.877427	4.206e-03	1.455e-05	4.206e-03	139.55

TABLE 4. Simulation results under the AJD model for a zero coupon bond with maturity $T = 3$ years.

6.2.2. Cap

Finally, we will see how the algorithm handles the treatment of a cap with annual payments and maturity $T = 3$, whose true value is 0.001196324. According to the results in table 5, our implementation loses some performance when compared to the evaluation of a zero coupon bond, but still performs incredibly well.

There is a detail regarding this implementation that is worth discussing. As the payoff of a multiple payment cap can be broke down in calls with different maturities, one simple way to do this would be running the algorithm sequentially up to each maturity ($T = 1, 2, 3$ in our case). This leads to a performance about 3 times worse than the bond case, as we need to essentially run 3 times. Most of the time, the first sampled exit time τ is larger than some of these stop points and we can treat them simultaneously in the same execution.

Method	Trials	Steps	Value	Bias	SE	RMSE	Time (sec)
Exact	10K	N/A	0.001234	0	6.467e-05	6.467e-05	2.12
	20K	N/A	0.001223	0	4.534e-05	4.534e-05	4.49
	40K	N/A	0.001213	0	3.191e-05	3.191e-05	9.12
	100K	N/A	0.001202	0	1.982e-05	1.982e-05	18.21
	500K	N/A	0.001192	0	8.801e-06	8.801e-06	236.96
Discrete I	10K	100	0.001256	2.301e-05	6.542e-05	6.935e-05	5.74
	20K	140	0.001159	1.223e-05	4.376e-05	4.544e-05	15.95
	40K	200	0.001208	7.181e-06	3.104e-05	3.186e-05	45.37
	100K	310	0.001180	5.544e-06	1.953e-05	2.030e-05	179.23
	500K	707	0.001196	4.765e-06	8.774e-06	9.984e-06	1997.96
Discrete II	10K	100	0.001133	2.121e-05	6.030e-05	6.392e-05	6.45
	20K	140	0.001136	1.452e-05	4.318e-05	4.556e-05	18.13
	40K	200	0.001120	1.151e-05	2.917e-05	3.136e-05	50.92
	100K	310	0.001183	7.445e-06	1.941e-05	2.079e-05	202.52
	500K	707	0.001203	7.473e-07	8.833e-06	8.865e-06	2260.38
Discrete III	10K	100	0.001144	2.409e-04	6.411e-05	2.493e-04	1.13
	20K	140	0.001072	1.461e-04	4.194e-05	1.520e-04	2.47
	40K	200	0.001146	1.717e-04	3.137e-05	1.745e-04	6.1
	100K	310	0.001088	9.891e-05	1.880e-05	1.007e-04	18.4
	500K	707	0.001076	6.508e-05	8.423e-06	6.562e-05	138.02

TABLE 5. Simulation results under the AJD model for a caplet with maturity $T = 3$ years, a strike $K = 0.05$ and yearly payments.

6.3. Conclusions

We began by exploring the needed tools, and built an exact method for progressively more general models defined by stochastic differential equations, eventually arriving at full generality.

The exact method's convergence lives up to the theoretical expectations, and we were able to replicate the results obtained by Giesecke and Smelov in 2013. It achieved the optimal convergence rate, while we also illustrated that the existence of bias significantly lowers the performance of classical discretization methods.

In terms of implementation, our results suggest that the exact method may be more sensitive to hardware changes. Furthermore, while MATLAB is very efficient at dealing with number arrays and linear algebra, it loses out to Python when the programs become more complex and involve several loops, comparisons and calls. Thus, the performance improvement for the exact method is larger than that of the discretization methods. The `numpy` package also provides efficient linear algebra tools, rivaling those of MATLAB.

As prospects of future research, finding more optimal choices of θ, improving general implementation and extending to multiple dimensions are among the possibilities. In this approach, a particular measure for efficiency was chosen, and there were several approximations and lower bounds leading to our final choice of θ. Our implementations also favored the discrete methods, as they are far more straightforward, and there is a lot of room for improvement in terms of code for the exact method, leading to possible stronger results.

Ultimately, extending the result to multiple dimensions is of large interest. Many jump models used nowadays are either multi-factor, allowing us to consider multiple dynamics simultaneously, or include, for instance, stochastic volatility, where it is itself driven by a mean-reverting process. One may also consider various combinations of models, where we have simultaneously a stochastic interest rate and asset price.

A recent paper by Blanchet and Zhang provides the first generic exact simulation algorithm for multivariate Itô diffusions, introducing new methods as the Lamperti transform only works in one dimension [8]. There is, however, no extension yet to multivariate jumps.

APPENDIX A

Python code

A.1. Sampling methods

Includes all of the needed methods, developed in the Monte Carlo section.

```
1  import sys
2  from matplotlib import pyplot as plt
3  import numpy as np
4  from scipy.special import gamma
5
6
7  def generate_exp(lamb=1):
8      # generates an exponential random variable with the inverse transform
           method
9      u = np.random.rand()
10     sample = - np.log(u)/lamb
11     return sample
12
13
14 def generate_poisson_jumps(lamb, T):
15     # generates poisson jump times (up to T) with rate lamb
16     jump_times = []
17     t = 0
18     while t < T:
19         tau = generate_exp(lamb)
20         t += tau
21         if t < T:
22             jump_times.append(t)
23
24     return jump_times
25
26
27 def generate_bm(times):
28     # generates a finite sample path of Brownian motion at the requested
           times
29
30     n = len(times)
31
32     z = np.random.normal()
33     new_step = np.sqrt(times[0]) * z
34     path = [new_step]
35
```

```
36      prev_step = new_step
37      i = 1
38      while i < n:
39          z = np.random.normal()
40          dt = times[i] - times[i-1]
41          new_step = prev_step + np.sqrt(dt)*z
42
43          path.append(new_step)
44          prev_step = new_step
45
46          i += 1
47
48      return path
49
50
51  def generate_bridge(times):
52      # generates a sample path of a Brownian bridge from a sample path of
            brownian motion
53
54      n = len(times)
55      T = times[-1]
56
57      bm = generate_bm(times)
58      bridge = []
59
60      for i in range(n):
61          new_step = bm[i] - times[i] * bm[-1] / T
62          bridge.append(new_step)
63
64      return bridge
65
66
67  def gamma_dens(t, b, y):
68      # pdf of the gamma distribution
69
70      g = y**b * t**(b-1) * np.exp(-y*t) / gamma(b)
71      return g
72
73
74  def generate_exit_time():
75      # generates a sample of Brownian motion exit time following the
            approach of
76
77      a = 1.243707
78      b = 1.088870
79      y = 1.233701
80
```

```
81      tau = 0
82      rejected = True
83
84      while rejected:
85          v = np.random.gamma(shape=b, scale=1 / y)
86          u = np.random.rand()
87          test_value = a * u * gamma_dens(v, b, y)
88
89          h = 1 / np.sqrt(2 * np.pi * v ** 3) * np.exp(-1 / (2 * v))
90          terminated = False
91          j = 1
92
93          while not terminated:
94
95              p_term = (2 * j + 1) * np.exp(-(2 * j + 1) ** 2 / (2 * v))
96              n_term = (2 * j - 1) * np.exp(-(2 * j - 1) ** 2 / (2 * v))
97              h_next = h + (-1) ** j / np.sqrt(2 * np.pi * v ** 3) * (p_term
                    - n_term)
98
99              if test_value < h_next <= h:
100                 terminated = True
101                 rejected = False
102
103                 tau = v
104
105             elif h <= h_next < test_value:
106                 terminated = True
107
108             h = h_next
109             j += 1
110
111     return tau
112
113
114 def generate_brownian_meander(times):
115     # generates a sample path of a Brownian meander at the given times,
116     # where the last element is taken as the exit time
117
118     tau = times[-1]
119
120     meander = []
121     rejected = True  # set to True so that we go through the loop at least
            once
122     invalid = False
123
124     # if only the exit time is given, we return the final value (-1 or 1
            with equal probability)
```

```
125     if len(times) == 1:
126         w_tau = np.random.choice([-1, 1])
127         return [w_tau]
128 
129     while rejected or invalid:
130         invalid = False
131         rejected = False
132 
133         bridges = [[], [], []]
134         candidate = []
135         w_tau = np.random.choice([-1, 1])
136 
137         # generate the 3 required Brownian bridges
138         for i in range(3):
139             bridges[i] = generate_bridge(times)
140 
141         # transform them into a test sample
142         for i in range(len(times)):
143             t = times[i]
144             b = np.sqrt(((tau - t) / tau + bridges[0][i]) ** 2 + bridges
                    [1][i] ** 2 + bridges[2][i] ** 2)
145 
146             if b >= 2:
147                 # the sample is invalid, and we break out of the for loop
148                 invalid = True
149                 break
150 
151             candidate.append(b)
152 
153         # if our sample is invalid, we return to the beginning of the while
                loop
154         if invalid:
155             continue
156 
157         # first part of the test (acceptance/rejection against p)
158         for i in range(len(times) - 2):
159             u = np.random.rand()
160 
161             s = tau - times[i]
162             x = candidate[i]
163             t = tau - times[i + 1]
164             y = candidate[i + 1]
165 
166             denom = 1 - np.exp(2 * x * y / (t - s))
167             p = 1
168             terminated = False
169             j = 1
```

```
170
171             # only a finite number of steps is needed for our verification
                    to terminate
172             while not terminated:
173
174                 if j % 2 == 0:
175                     jj = j // 2
176                     nu = np.exp(2 * jj * (4 * jj + 2 * (x - y)) / (t - s))
                            + np.exp(
177                         2 * jj * (4 * jj - 2 * (x - y)) / (t - s))
178                     p_next = p + nu
179                 else:
180                     jj = j // 2 + 1
181                     theta = np.exp(2 * (2 * jj - x) * (2 * jj - y) / (t - s
                            )) + np.exp(
182                         2 * (2 * (jj - 1) + x) * (2 * (jj - 1) + y) / (t -
                                s))
183                     p_next = p - theta
184
185                 if denom * u < p_next <= p:
186                     # the sum terminates and we don't reject the sample
187                     terminated = True
188
189                 elif p <= p_next < u:
190                     # the sum terminates and we reject the sample
191                     rejected = True
192                     terminated = True
193
194                 p = p_next
195                 j += 1
196
197             # if the sample was already rejected, we break out of the for
                    loop
198             if rejected:
199                 break
200
201         # if we reject the sample in the first tests, we return to the
                beginning of the while loop
202         if rejected:
203             continue
204
205         # second part of the test (acceptance/rejection against q)
206         u = np.random.rand()
207         t = tau - times[-2]
208         x = candidate[-2]
209
210         q = 1
```

```
211         j = 1
212         terminated = False
213 
214         while not terminated:
215 
216             if j % 2 == 0:
217                 jj = j // 2
218                 rho2 = (4 * jj + x) * np.exp(-4 * jj * (2 * jj + x) / t)
219                 q_next = q + rho2 / x
220             else:
221                 jj = j // 2 + 1
222                 rho1 = (4 * jj - x) * np.exp(-4 * jj * (2 * jj - x) / t)
223                 q_next = q - rho1 / x
224 
225             if u < q_next <= q:
226                 # the sum terminates and we do accept the sample
227                 terminated = True
228             elif q <= q_next < u:
229                 # the sum terminates and we reject the sample
230                 terminated = True
231                 rejected = True
232 
233             q = q_next
234             j += 1
235 
236     if w_tau == 1:
237         meander = [1 - b for b in candidate]
238     else:
239         meander = [b - 1 for b in candidate]
240 
241     return meander
```

A.2. Level selection

```
1  import numpy as np
2  import sampling
3  from scipy.optimize import minimize_scalar
4  from scipy import integrate
5  import time
6  import matplotlib.pyplot as plt
7  import sys
8  import csv
9 
10 T = 1
11 strike = 5
12 X_0 = 50
13 beta = -1
```

```
14 r = 0.05
15 sigma = 50/4
16 b = 0
17 c = 1/2
18
19
20 def f(x):
21     return (X_0 ** (-beta) - x ** (-beta)) / (beta * sigma)
22
23
24 # inverse Lamperti transform
25 def f_inv(x):
26     return (X_0 ** (-beta) - x * sigma * beta) ** (-1 / beta)
27
28
29 # drift function
30 def mu_y(x):
31     return ((r + b) / sigma) * (X_0 ** (-beta) - x * sigma * beta) + sigma
       * (c - (beta + 1) / 2) / (
32                 X_0 ** (-beta) - x * sigma * beta)
33
34
35 def jump_int(x):
36     return b + c * sigma ** 2 * x ** (2 * beta)
37
38
39 def delta_y(x):
40     return X_0 ** (-beta) / (sigma * beta) - x
41
42
43 def phi_y(x):
44     return 0.5 * (-(r + b) * beta + sigma ** 2 * beta * (c - (beta + 1) /
       2) / (
45                 X_0 ** (-beta) - x * sigma * beta) ** 2 + mu_y(x) ** 2)
46
47
48 # function A(y) needed for the acceptance tests
49 def integrated_drift(x):
50     return (r + b) / sigma * (X_0 ** (-beta) * x - x ** 2 / 2 * sigma *
       beta) - \
51             (c - (beta + 1) / 2) / beta * np.log(1 - x * sigma * beta * X_0
                ** beta)
52
53
54 def h(t, precision=1e-10, max_iter=100):
55
56     h_value = 1 / np.sqrt(2 * np.pi * t ** 3) * np.exp(-1 / (2 * t))
```

```
57      j = 1
58
59      p_term = (2 * j + 1) * np.exp(-(2 * j + 1) ** 2 / (2 * t))
60      n_term = (2 * j - 1) * np.exp(-(2 * j - 1) ** 2 / (2 * t))
61      summand = (-1) ** j / np.sqrt(2 * np.pi * t ** 3) * (p_term - n_term)
62
63      h_value += summand
64
65      while abs(summand) > precision and j < max_iter:
66
67          p_term = (2 * j + 1) * np.exp(-(2 * j + 1) ** 2 / (2 * t))
68          n_term = (2 * j - 1) * np.exp(-(2 * j - 1) ** 2 / (2 * t))
69          summand = (-1) ** j / np.sqrt(2 * np.pi * t ** 3) * (p_term -
                n_term)
70          h_value += summand
71
72          j += 1
73
74      return h_value
75
76
77  def objective_function2(y, theta):
78      phi_max = -minimize_scalar(lambda x: -phi_y(x), bounds=(y - theta, y +
            theta), method='bounded').fun
79      phi_min = minimize_scalar(phi_y, bounds=(y - theta, y + theta), method=
            'bounded').fun
80      c = -minimize_scalar(lambda x: -jump_int(f_inv(x)), bounds=(y - theta,
            y + theta), method='bounded').fun
81      K = -minimize_scalar(lambda x: -np.exp(integrated_drift(x)), bounds=(y
            - theta, y + theta), method='bounded').fun
82      S = max(np.exp(-phi_min*T), 1)
83
84      val1 = integrate.quad(lambda t: h(t/theta**2), 0, T)[0]
85      val2 = integrate.quad(lambda t: t*h(t/theta**2)*np.exp(-phi_max*t), 0,
            T)[0]
86      val3 = integrate.quad(lambda t: t*h(t/theta**2)*np.exp(-(phi_max+c)*t),
             0, T)[0]
87      val4 = integrate.quad(lambda t: h(t/theta**2)*np.exp(-(phi_max+c)*t),
            0, T)[0]
88
89      term1 = c*np.exp(integrated_drift(y))*(val1 - (phi_max+c)*val3 - val4)
            /(K*S*theta**2*(phi_max+c)**2)
90      term2 = (np.exp(integrated_drift(y+theta)) + np.exp(integrated_drift(y-
            theta)))*val2/(2*K*S*theta**2)
91      term3 = c*np.exp(integrated_drift(y))*(1-val1/theta**2)*((1-np.exp(-(
            phi_max+c)*T))/(phi_max+c)
```

```
92                                                                  + (T*np.exp(-(
                                                                        phi_max+c)*
                                                                        T))/c)/(K*S
                                                                        )
93
94      return -(term1+term2+term3)
95
96
97  def main():
98      theta_values = []
99      y_range = np.linspace(-3.5, 20, 100)
100     for y in y_range:
101         if y < -2:
102             u_bound = 4 + y
103         else:
104             u_bound = 2
105         theta = minimize_scalar(lambda x: objective_function2(y, x), bounds
                =(0.05, u_bound), method='Bounded').x
106         theta_values.append(theta)
107         print(theta)
108     with open('jdcev_thetas.txt', 'w') as text_file:
109         for theta in theta_values:
110             text_file.write("%s\n" % theta)
111
112
113 main()
```

```
1   import numpy as np
2   import sampling
3   from scipy.optimize import minimize_scalar
4   from scipy import integrate
5   import time
6   import matplotlib.pyplot as plt
7   import sys
8   import csv
9
10  lamb0 = 0.0110
11  lamb1 = 0.1000
12
13  min_jump = 0.0113
14  max_jump = 0.0312
15
16  X_0 = 0.0422
17  mean = 0.0422
18  kappa = 0.0117
19  sigma = 0.0130
20
```

```
21 T = 3
22 T_max = 3
23 strike = 0.05
24 periods = 3
25 cap_limits = [1, 2, 3]
26
27 # model functions
28
29
30 # lamperti transform
31 def f(x):
32     return 2*(np.sqrt(x) - np.sqrt(X_0))/sigma
33
34
35 # inverse lamperti transform
36 def f_inv(x):
37     return (sigma*x/2 + np.sqrt(X_0))**2
38
39
40 # drift function
41 def mu_y(x):
42     return (4*kappa*mean - sigma**2)/(2*sigma**2)/(x+2*np.sqrt(X_0)/sigma)
           - kappa/2 * (x+2*np.sqrt(X_0)/sigma)
43
44
45 def jump_int(x):
46     return lamb0 + lamb1 * x
47
48
49 def delta_y(x, z):
50     return 2 * (np.sqrt((np.sqrt(X_0) + x*sigma/2)**2 + z) - np.sqrt(X_0))/
           sigma - x
51
52
53 def phi_y(x):
54     return 0.5 * (-(4*kappa*mean - sigma**2)/(2*sigma**2)/(x+2*np.sqrt(X_0)
           /sigma)**2
55                   - kappa/2 + mu_y(x)**2)
56
57
58 # function A(y) needed for the acceptance tests
59 def integrated_drift(x):
60     return (4*kappa*mean - sigma**2)/(2*sigma**2) * np.log(1 + x*sigma/(2*
           np.sqrt(X_0))) \
61             - kappa/2 * (x**2/2 + 2*np.sqrt(X_0)*x/sigma)
62
63
```

```
64 def h(t, precision=1e-10, max_iter=100):
65
66     h_value = 1 / np.sqrt(2 * np.pi * t ** 3) * np.exp(-1 / (2 * t))
67     j = 1
68
69     p_term = (2 * j + 1) * np.exp(-(2 * j + 1) ** 2 / (2 * t))
70     n_term = (2 * j - 1) * np.exp(-(2 * j - 1) ** 2 / (2 * t))
71     summand = (-1) ** j / np.sqrt(2 * np.pi * t ** 3) * (p_term - n_term)
72
73     h_value += summand
74
75     while abs(summand) > precision and j < max_iter:
76
77         p_term = (2 * j + 1) * np.exp(-(2 * j + 1) ** 2 / (2 * t))
78         n_term = (2 * j - 1) * np.exp(-(2 * j - 1) ** 2 / (2 * t))
79         summand = (-1) ** j / np.sqrt(2 * np.pi * t ** 3) * (p_term -
               n_term)
80         h_value += summand
81
82         j += 1
83
84     return h_value
85
86
87 def objective_function2(y, theta):
88     phi_max = -minimize_scalar(lambda x: -phi_y(x), bounds=(y - theta, y +
           theta), method='bounded').fun
89     phi_min = minimize_scalar(phi_y, bounds=(y - theta, y + theta), method=
           'bounded').fun
90     c = -minimize_scalar(lambda x: -jump_int(f_inv(x)), bounds=(y - theta,
           y + theta), method='bounded').fun
91     K = -minimize_scalar(lambda x: -np.exp(integrated_drift(x)), bounds=(y
           - theta, y + theta), method='bounded').fun
92     S = max(np.exp(-phi_min*T), 1)
93
94     val1 = integrate.quad(lambda t: h(t/theta**2), 0, T)[0]
95     val2 = integrate.quad(lambda t: t*h(t/theta**2)*np.exp(-phi_max*t), 0,
           T)[0]
96     val3 = integrate.quad(lambda t: t*h(t/theta**2)*np.exp(-(phi_max+c)*t),
            0, T)[0]
97     val4 = integrate.quad(lambda t: h(t/theta**2)*np.exp(-(phi_max+c)*t),
           0, T)[0]
98
99     term1 = c*np.exp(integrated_drift(y))*(val1 - (phi_max+c)*val3 - val4)
           /(K*S*theta**2*(phi_max+c)**2)
100    term2 = (np.exp(integrated_drift(y+theta)) + np.exp(integrated_drift(y-
           theta)))*val2/(2*K*S*theta**2)
```

```
    term3 = c*np.exp(integrated_drift(y))*(1-val1/theta**2)*((1-np.exp(-(
        phi_max+c)*T))/(phi_max+c)
                                                        + (T*np.exp(-(
                                                            phi_max+c)*
                                                            T))/c)/(K*S
                                                            )

    return -(term1+term2+term3)


def main():
    theta_values = []
    y_range = np.linspace(-29, 40, 100)
    for y in y_range:
        if y < -20:
            u_bound = 30 + y
        else:
            u_bound = 10
        theta = minimize_scalar(lambda x: objective_function2(y, x), bounds
            =(0.005, u_bound), method='Bounded').x
        theta_values.append(theta)
        print(theta)
    with open('ajd_thetas.txt', 'w') as text_file:
        for theta in theta_values:
            text_file.write("%s\n" % theta)


main()
```

A.3. JDCEV implementation

We provide the implementation only for the put case, as the algorithm is easily adaptable to the other tests.

A.3.1. Exact method

```
import numpy as np
import sampling
from scipy.optimize import minimize_scalar
import time
import matplotlib.pyplot as plt


def monte_carlo(n_sim=1000, sim_frames=[1000], T=1, strike=5, X_0=50, beta
    =-1, r=0.05, sigma=50/4, b=0, c=1/2):

    # model functions
    # Lamperti transform
```

```
12      def f(x):
13          return (X_0**(-beta) - x**(-beta))/(beta*sigma)
14
15      # inverse Lamperti transform
16      def f_inv(x):
17          return (X_0**(-beta) - x*sigma*beta)**(-1/beta)
18
19      # drift function
20      def mu_y(x):
21          return ((r+b)/sigma)*(X_0**(-beta) - x*sigma*beta) + sigma * (c - (
                beta+1)/2)/(X_0**(-beta) - x*sigma*beta)
22
23      def jump_int(x):
24          return b + c * sigma**2 * x**(2*beta)
25
26      def delta_y(x):
27          return X_0**(-beta)/(sigma*beta) - x
28
29      def phi_y(x):
30          return 0.5 * (-(r+b)*beta + sigma**2 * beta * (c-(beta+1)/2)/(X_0
                **(-beta)-x*sigma*beta)**2 + mu_y(x)**2)
31
32      # function A(y) needed for the acceptance tests
33      def integrated_drift(x):
34          return (r+b)/sigma * (X_0**(-beta)*x - x**2/2 * sigma * beta) - \
35                  (c - (beta+1)/2)/beta * np.log(1 - x*sigma*beta*X_0**beta)
36
37      final_samples = []
38      default_times = []
39      sim_data = []
40      optimal_thetas = [float(theta.strip()) for theta in open("jdcev_thetas.
            txt", 'r')]
41      start_time = time.time()
42
43      sim_count = 0
44      while sim_count < n_sim:
45          # print(sim_count)
46          y = 0
47          t = 0
48
49          # main while loop iterating over time segments
50          while t < T:
51              # choice of theta
52              y_range = np.linspace(-3.5, 20, 100)
53              if -3.5 < y < 19:
54                  for i in range(len(y_range)):
55                      if y_range[i] < y:
```

```
56                      theta_opt = (optimal_thetas[i] * (y_range[i + 1] -
                            y) + optimal_thetas[i + 1] * (
57                                  y - y_range[i])) \
58                                  / (y_range[i + 1] - y_range[i])
59                      theta = min(theta_opt, (y - X_0 ** (-beta) / (sigma
                            * beta)) / 2)
60                      break
61              else:
62                  theta = min(1.5, (y - X_0 ** (-beta) / (sigma * beta)) / 2)
63
64              # sampling and acceptance while loop
65              rejected = True
66              while rejected:
67
68                  # determination of the minimum and maximum of phi
69                  # phi_max = -minimize_scalar(lambda x: -phi_y(x), bounds=(y
                        -theta, y+theta), method='bounded').fun
70                  # phi_min = minimize_scalar(phi_y, bounds=(y-theta, y+theta
                        ), method='bounded').fun
71
72                  phi_max = phi_y(y+theta)
73                  phi_min = phi_y(y-theta)
74
75                  lamb = jump_int(f_inv(y - theta))
76
77                  # generate the exit time for the interval [y-theta, y+theta
                        ]
78                  tau = theta**2 * sampling.generate_exit_time()
79
80                  # generate the required Poisson jump times
81                  candidate_times = sampling.generate_poisson_jumps(lamb, min
                        (tau, T-t))
82                  test_times = sampling.generate_poisson_jumps(phi_max-
                        phi_min, min(tau, T-t))
83
84                  all_times = sorted(candidate_times + test_times)
85                  if T-t < tau:
86                      all_times.append(T-t)
87                  all_times.append(tau)
88                  # generate a Brownian meander with exit time tau at all the
                         required times and we rescale
89                  candidate_bridge_values = sampling.
                        generate_brownian_meander([i/theta**2 for i in all_times
                        ])
90                  candidate_bridge_values = [x*theta for x in
                        candidate_bridge_values]
91
```

```
92                  candidate_bridge = {all_times[i]: candidate_bridge_values[i
                        ] for i in range(len(all_times))}
93
94                  i = 0
95                  a = len(candidate_times)
96                  while i < a:
97                      u = np.random.rand()
98                      if u*lamb < jump_int(f_inv(y + candidate_bridge[
                            candidate_times[i]])):
99                          # a default time is accepted and we break the loop
100                         break
101                     else:
102                         i += 1
103
104                 if i == a:
105
106                     # no jump time is accepted and we test the complete
                            skeleton
107                     stopping_time = t + min(tau, T - t)
108
109                     u, v = np.random.rand(2)
110
111                     # first Bernoulli variable
112                     K = integrated_drift(y+theta)
113                     test_factor_1 = (np.log(u) < integrated_drift(y +
                            candidate_bridge[min(tau, T-t)]) - K)
114                     # second Bernoulli variable
115                     S = max(np.exp(-phi_min*(T-t)), 1)
116                     test_factor_2 = (v*S < np.exp(-phi_min*min(tau, T-t)))
117
118                     # thinning test process
119                     test_factor_3 = True
120                     j = 0
121                     while j < len(test_times):
122
123                         w = np.random.rand()
124                         if w * (phi_max-phi_min) < phi_y(y +
                                candidate_bridge[test_times[j]])-phi_min:
125
126                             # one of the test jump times \kappa_j is
                                    accepted and we reject the skeleton
127                             test_factor_3 = False
128                             break
129                         j += 1
130
131                     if test_factor_1 and test_factor_2 and test_factor_3:
132
```

```
                    # the sample is accepted
                    rejected = False
                    if stopping_time < T:
                        y += candidate_bridge_values[-1]
                        t = stopping_time

                    else:
                        y += candidate_bridge[T-t]
                        final_samples.append(f_inv(y))
                        t = T

            else:

                # one of the jump times was accepted and we test the
                    skeleton up to the default time
                stopping_time = t + candidate_times[i]

                u, v = np.random.rand(2)
                K = integrated_drift(y+theta)
                S = max(np.exp(-phi_min * min(tau, (T-t))), 1)
                # first Bernoulli variable
                test_factor_1 = (np.log(u) < integrated_drift(y +
                    candidate_bridge[candidate_times[i]]) - K)
                # second Bernoulli variable
                test_factor_2 = (v*S < np.exp(-phi_min*candidate_times[
                    i]))

                test_factor_3 = True
                j = 0
                while j < len(test_times):

                    # we only need to test up to default time
                    if test_times[j] > candidate_times[i]:
                        break

                    w = np.random.rand()
                    if w * (phi_max-phi_min) < phi_y(y +
                        candidate_bridge[test_times[j]])-phi_min:
                        test_factor_3 = False
                        break

                    j += 1

                if test_factor_1 and test_factor_2 and test_factor_3:

                    rejected = False
                    final_samples.append(0)
```

```
                        default_times.append(stopping_time)
                        t = T
        sim_count += 1
        if sim_count in sim_frames:
            option_results = [max(strike - x, 0) for x in final_samples] *
                np.exp(-r)
            option_price = np.mean(option_results)
            sample_std = np.std(option_results)
            std_error = sample_std / np.sqrt(sim_count)
            time_spent = time.time() - start_time

            print(sim_count)
            sim_data.append([sim_count, option_price, std_error, time_spent
                ])

    return sim_data
```

A.3.2. Discretization method

```
import numpy as np
import sampling
import time


def monte_carlo(sim_frames=[1000], T=1, strike=5, X_0=50, beta=-1, r=0.05,
    sigma=50/4, b=0, c=1/2):

    def jump_int(x):
        return b + c * sigma**2 * x**(2*beta)

    sim_data = []

    for n_sim in sim_frames:
        results = []
        N = int(np.sqrt(n_sim))
        h = T / N

        start_time = time.time()
        sim_count = 0
        while sim_count < n_sim:
            i = 0
            X = np.zeros(N+1)
            X[0] = X_0

            compensator = h * jump_int(X[0])
            default_breakpoint = sampling.generate_exp()

```

```
            while i < N:
                u = np.random.normal()
                X[i+1] = X[i] + (r + jump_int(X[i])) * X[i] * h + sigma*(X[
                    i]**(beta+1)) * np.sqrt(h) * u

                compensator += h * jump_int(X[i+1])

                if X[i+1] < 0 or compensator > default_breakpoint:
                    X[i+1] = 0
                    break

                i += 1

            results.append(X[-1])
            sim_count += 1

        print(n_sim)
        option_results = [max(strike-x, 0)*np.exp(-r*T) for x in results]
        option_price = np.mean(option_results)
        sample_std = np.std(option_results)
        std_error = sample_std/np.sqrt(n_sim)
        time_spent = time.time() - start_time

        sim_data.append([n_sim, option_price, std_error, time_spent])

    return sim_data
```

A.4. AJD Implementation

A.4.1. Exact method

A.4.1.1. *Zero coupon bond*

```
import numpy as np
from scipy.optimize import minimize_scalar
import sampling

import time


# params

lamb0 = 0.0110
lamb1 = 0.1000

min_jump = 0.0113
max_jump = 0.0312

X_0 = 0.0422
```

```
17 mean = 0.0422
18 kappa = 0.0117
19 sigma = 0.0130
20
21 T = 3
22
23 # model functions
24
25
26 # lamperti transform
27 def f(x):
28     return 2*(np.sqrt(x) - np.sqrt(X_0))/sigma
29
30
31 # inverse lamperti transform
32 def f_inv(x):
33     return (sigma*x/2 + np.sqrt(X_0))**2
34
35
36 # drift function
37 def mu_y(x):
38     return (4*kappa*mean - sigma**2)/(2*sigma**2)/(x+2*np.sqrt(X_0)/sigma)
           - kappa/2 * (x+2*np.sqrt(X_0)/sigma)
39
40
41 # jump intensity
42 def jump_int(x):
43     return lamb0 + lamb1 * x
44
45 # jump size
46 def delta_y(x, z):
47     return 2 * (np.sqrt((np.sqrt(X_0) + x*sigma/2)**2 + z) - np.sqrt(X_0))/
           sigma - x
48
49
50 # phi = 0.5*(mu' + mu^2)
51 def phi_y(x):
52     return 0.5 * (-(4*kappa*mean - sigma**2)/(2*sigma**2)/(x+2*np.sqrt(X_0)
           /sigma)**2
53                   - kappa/2 + mu_y(x)**2)
54
55
56 # function A(y) needed for the acceptance tests
57 def integrated_drift(x):
58     return (4*kappa*mean - sigma**2)/(2*sigma**2) * np.log(1 + x*sigma/(2*
           np.sqrt(X_0))) \
59             - kappa/2 * (x**2/2 + 2*np.sqrt(X_0)*x/sigma)
```

```
60
61
62  def monte_carlo(n_sim=1000, sim_frames=[1000]):
63      sim_count = 0
64      sim_data = []
65      bond_results = []
66      optimal_thetas = [float(theta.strip()) for theta in open("ajd_thetas.
            txt", 'r')]
67
68      # we start the timer
69      start_time = time.time()
70
71      # main simulation loop
72      while sim_count < n_sim:
73
74          # initialize the simulation variables
75          y = 0
76          t = 0
77          g = 1  # estimator of the time-integrated exponential
78
79          # loop until we reach our horizon T
80          while t < T:
81
82              # choice of theta
83              y_range = np.linspace(-29, 40, 100)
84              if -29 < y < 40:
85                  for i in range(len(y_range)):
86                      if y_range[i] < y:
87                          theta_opt = (optimal_thetas[i] * (y_range[i + 1] -
                                y) + optimal_thetas[i + 1] * (
88                                      y - y_range[i])) \
89                                          / (y_range[i + 1] - y_range[i])
90                          theta = min(theta_opt, (y - f(0)) / 4)
91                          break
92              else:
93                  theta = min(3, (y - f(0)) / 4)
94
95              # determination of the minimum and maximum of phi
96
97              # phi_max = -minimize_scalar(lambda x: -phi_y(x), bounds=(y -
                    theta, y + theta), method='bounded').fun
98              # phi_min = minimize_scalar(phi_y, bounds=(y - theta, y + theta
                    ), method='bounded').fun
99
100             theta_min = -2.9081943013559806
101             if y - theta < theta_min < y + theta:
102                 phi_min = -0.006002
```

```
103                 phi_max = max(phi_y(y - theta), phi_y(y + theta))
104             elif y + theta < theta_min:
105                 phi_max = phi_y(y - theta)
106                 phi_min = phi_y(y + theta)
107             elif theta_min < y - theta:
108                 phi_max = phi_y(y + theta)
109                 phi_min = phi_y(y - theta)
110
111             lamb = jump_int(f_inv(y + theta))
112
113             # bounds for sampling of estimator
114             x_up = f_inv(y + theta)
115             x_down = f_inv(y - theta)
116
117             # main A/R scheme loop
118             rejected = True
119             while rejected:
120
121                 # generate the exit time for the interval [y-theta, y+theta
                        ]
122                 tau = theta ** 2 * sampling.generate_exit_time()
123
124                 # generate the required Poisson jump times
125                 candidate_times = sampling.generate_poisson_jumps(lamb, min
                        (tau, T - t))
126                 test_times = sampling.generate_poisson_jumps(phi_max -
                        phi_min, min(tau, T - t))
127                 exponential_times = sampling.generate_poisson_jumps(x_up-
                        x_down, min(tau, T-t))
128
129                 # sort all times and add T-t if needed
130                 all_times = sorted(candidate_times + test_times +
                        exponential_times)
131                 if T - t < tau:
132                     all_times.append(T - t)
133                 all_times.append(tau)
134
135                 candidate_bridge_values = sampling.
                        generate_brownian_meander([i / theta ** 2 for i in
                        all_times])
136                 candidate_bridge_values = [x * theta for x in
                        candidate_bridge_values]
137
138                 candidate_bridge = {all_times[i]: candidate_bridge_values[i
                        ] for i in range(len(all_times))}
139
140                 i = 0
```

```
141                 a = len(candidate_times)
142
143                 while i < a:
144                     u = np.random.rand()
145                     w = candidate_bridge[candidate_times[i]]
146
147                     if u * lamb < jump_int(f_inv(y + w)):
148                         break
149                     else:
150                         i += 1
151
152                 if i == a:
153
154                     # no jump time was accepted
155                     stopping_time = min(t + tau, T)
156
157                     u, v = np.random.rand(2)
158
159                     # first Bernoulli variable
160                     # K = -minimize_scalar(lambda x: - integrated_drift(x),
161                     #                      bounds=(y-theta, y+theta),
                            method='bounded').fun
162
163                     # previously computed location of global maximum
164                     K_max = 1.382436647017447
165
166                     if y - theta < K_max < y + theta:
167                         K = 0.011016
168                     else:
169                         K = max(integrated_drift(y - theta),
                                integrated_drift(y + theta))
170
171                     w = candidate_bridge[min(tau, T - t)]
172                     test_factor_1 = (np.log(u) < integrated_drift(y + w) -
                            K)
173
174                     # second Bernoulli variable
175                     S = max(np.exp(-phi_min * (T - t)), 1)
176                     test_factor_2 = (v * S < np.exp(-phi_min * min(tau, T -
                            t)))
177
178                     # thinning test process
179                     test_factor_3 = True
180                     j = 0
181                     while j < len(test_times):
182
183                         u = np.random.rand()
```

```
184                     w = candidate_bridge[test_times[j]]
185                     if u * (phi_max - phi_min) < phi_y(y + w) - phi_min
                            :
186
187                         # one of the test jump times \kappa_j is
                                accepted and we reject the skeleton
188                         test_factor_3 = False
189                         break
190
191                     j += 1
192
193                 if test_factor_1 and test_factor_2 and test_factor_3:
194
195                     # the sample is accepted
196                     rejected = False
197
198                     if stopping_time < T:
199                         for jump in exponential_times:
200                             w = candidate_bridge[jump]
201                             g = g * (1 - (f_inv(y + w) - x_down)/(x_up
                                    - x_down))
202
203                         g = g * np.exp(-x_down * tau)
204
205                         y += candidate_bridge_values[-1]
206                         t = stopping_time
207
208                     else:
209                         for jump in exponential_times:
210                             if T - t < jump:
211                                 break
212                             w = candidate_bridge[jump]
213                             g = g * (1 - (f_inv(y + w) - x_down)/(x_up
                                    - x_down))
214
215                         g = g * np.exp(-x_down * (T-t))
216                         bond_results.append(g)
217                         w = candidate_bridge[T-t]
218                         y += w
219                         t = T
220
221             else:
222
223                 # one of the jump times was accepted and we test the
                        skeleton up to the jump time
224                 stopping_time = t + candidate_times[i]
225
```

```
                u, v = np.random.rand(2)

                # first Bernoulli variable

                # first optimization approach
                # K = -minimize_scalar(lambda x: - integrated_drift(x),
                #                      bounds=(y-theta, y+theta),
                    method='bounded').fun

                # second optimization approach
                K_max = 1.382436647017447  # previously computed
                    location of global maximum

                if y - theta < K_max < y + theta:
                    K = 0.011016
                else:
                    K = max(integrated_drift(y - theta),
                        integrated_drift(y + theta))

                w = candidate_bridge[candidate_times[i]]
                test_factor_1 = (np.log(u) < integrated_drift(y + w) -
                    K)

                # second Bernoulli variable
                S = max(np.exp(-phi_min * (T - t)), 1)
                test_factor_2 = (v * S < np.exp(-phi_min *
                    candidate_times[i]))

                test_factor_3 = True
                j = 0
                while j < len(test_times):

                    # we only need to test up to default time
                    if test_times[j] > candidate_times[i]:
                        break

                    u = np.random.rand()
                    w = candidate_bridge[test_times[j]]

                    if u * (phi_max - phi_min) < phi_y(y + w) - phi_min
                        :
                        test_factor_3 = False
                        break

                    j += 1

                if test_factor_1 and test_factor_2 and test_factor_3:
```

```
                        # we accept the skeleton up to jump time
                        rejected = False

                        for jump in exponential_times:
                            if candidate_times[i] < jump:
                                break
                            w = candidate_bridge[jump]
                            g = g * (1 - (f_inv(y + w) - x_down) / (x_up -
                                x_down))

                        g = g * np.exp(- x_down * candidate_times[i])

                        # compute Y immediately before the jump
                        w = candidate_bridge[candidate_times[i]]
                        y += w
                        # compute Y after the jump
                        u = np.random.rand()
                        z = min_jump + (max_jump - min_jump) * u
                        y += delta_y(y, z)
                        t = stopping_time

        sim_count += 1
        if sim_count in sim_frames:
            print(sim_count)
            bond_price = np.mean(bond_results)
            sample_std = np.std(bond_results)
            std_error = sample_std / np.sqrt(sim_count)
            time_spent = time.time() - start_time
            sim_data.append([sim_count, bond_price, std_error, time_spent])

    return sim_data
```

A.4.1.2. *Cap*

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

from scipy.optimize import minimize_scalar
import sampling

import time


# params

lamb0 = 0.0110
lamb1 = 0.1000
```

```
15
16 min_jump = 0.0113
17 max_jump = 0.0312
18
19 X_0 = 0.0422
20 mean = 0.0422
21 kappa = 0.0117
22 sigma = 0.0130
23
24 T = 3
25 strike = 0.05
26 periods = 3
27 cap_limits = [1, 2, 3]
28
29 # model functions
30
31
32 # lamperti transform
33 def f(x):
34     return 2*(np.sqrt(x) - np.sqrt(X_0))/sigma
35
36
37 # inverse lamperti transform
38 def f_inv(x):
39     return (sigma*x/2 + np.sqrt(X_0))**2
40
41
42 # drift function
43 def mu_y(x):
44     return (4*kappa*mean - sigma**2)/(2*sigma**2)/(x+2*np.sqrt(X_0)/sigma)
        - kappa/2 * (x+2*np.sqrt(X_0)/sigma)
45
46
47 def jump_int(x):
48     return lamb0 + lamb1 * x
49
50
51 def delta_y(x, z):
52     return 2 * (np.sqrt((np.sqrt(X_0) + x*sigma/2)**2 + z) - np.sqrt(X_0))/
        sigma - x
53
54
55 def phi_y(x):
56     return 0.5 * (-(4*kappa*mean - sigma**2)/(2*sigma**2)/(x+2*np.sqrt(X_0)
        /sigma)**2
57                   - kappa/2 + mu_y(x)**2)
58
```

```
59
60 # function A(y) needed for the acceptance tests
61 def integrated_drift(x):
62     return (4*kappa*mean - sigma**2)/(2*sigma**2) * np.log(1 + x*sigma/(2*
       np.sqrt(X_0))) \
63         - kappa/2 * (x**2/2 + 2*np.sqrt(X_0)*x/sigma)
64
65
66 def monte_carlo(n_sim=1000, sim_frames=[1000]):
67
68     sim_count = 0
69     sim_data = []
70     cap_results = []
71     optimal_thetas = [float(theta.strip()) for theta in open("ajd_thetas.
       txt", 'r')]
72
73     # we start the timer
74     start_time = time.time()
75
76     # main simulation loop
77     while sim_count < n_sim:
78         print(sim_count)
79         y = 0
80         t = 0
81         g = 1
82         cap_value = 0
83         cap_counter = 0
84
85         # loop until we reach our horizon T
86         while t < T:
87             # choice of theta
88             y_range = np.linspace(-29, 40, 100)
89             if -29 < y < 40:
90                 for i in range(len(y_range)):
91                     if y_range[i] < y:
92                         theta_opt = (optimal_thetas[i] * (y_range[i + 1] -
                           y) + optimal_thetas[i + 1] * (
93                                 y - y_range[i])) \
94                                     / (y_range[i + 1] - y_range[i])
95                         theta = min(theta_opt, (y - f(0)) / 4)
96                         break
97             else:
98                 theta = min(3, (y - f(0)) / 4)
99
100            # determination of the minimum and maximum of phi
101            # phi_max = -minimize_scalar(lambda x: -phi_y(x), bounds=(y -
               theta, y + theta), method='bounded').fun
```

```
102             # phi_min = minimize_scalar(phi_y, bounds=(y - theta, y + theta
                    ), method='bounded').fun
103
104             theta_min = -2.9081943013559806
105             if y - theta < theta_min < y + theta:
106                 phi_min = -0.006002
107                 phi_max = max(phi_y(y - theta), phi_y(y + theta))
108             elif y + theta < theta_min:
109                 phi_max = phi_y(y - theta)
110                 phi_min = phi_y(y + theta)
111             elif theta_min < y - theta:
112                 phi_max = phi_y(y + theta)
113                 phi_min = phi_y(y - theta)
114
115             lamb = jump_int(f_inv(y + theta))
116
117             # bounds for sampling of estimator
118             x_up = f_inv(y + theta)
119             x_down = f_inv(y - theta)
120
121             # main A/R scheme loop
122             rejected = True
123             while rejected:
124
125                 # generate the exit time for the interval [y-theta, y+theta
                        ]
126                 tau = theta ** 2 * sampling.generate_exit_time()
127
128                 # generate the required Poisson jump times
129                 candidate_times = sampling.generate_poisson_jumps(lamb, min
                        (tau, T - t))
130                 test_times = sampling.generate_poisson_jumps(phi_max -
                        phi_min, min(tau, T - t))
131                 exponential_times = sampling.generate_poisson_jumps(x_up -
                        x_down, min(tau, T - t))
132
133                 # sort all times and add T-t if needed
134                 all_times = candidate_times + test_times +
                        exponential_times
135
136                 if T - t < tau:
137                     all_times.append(T - t)
138                 # we add the necessary cap checkpoints
139                 cap_times = [t_cap - t for t_cap in cap_limits if t < t_cap
                        < min(T, t + tau)]
140                 all_times = all_times + cap_times
141                 all_times.append(tau)
```

```
142                 all_times = sorted(all_times)
143
144                 # generate a Brownian meander with exit time tau at all the
                        required times and we rescale
145                 candidate_bridge_values = sampling.
                        generate_brownian_meander([i / theta ** 2 for i in
                        all_times])
146                 candidate_bridge_values = [x * theta for x in
                        candidate_bridge_values]
147
148                 candidate_bridge = {all_times[i]: candidate_bridge_values[i
                        ] for i in range(len(all_times))}
149
150                 i = 0
151                 a = len(candidate_times)
152
153                 # iterating over the jump times to thin the jump process
154                 while i < a:
155                     u = np.random.rand()
156                     w = candidate_bridge[candidate_times[i]]
157
158                     if u * lamb < jump_int(f_inv(y + w)):
159                         # one of the jump times was accepted
160                         break
161                     else:
162                         i += 1
163
164                 if i == a:
165
166                     # no jump time was accepted
167                     stopping_time = min(t + tau, T)
168
169                     u, v = np.random.rand(2)
170
171                     # first Bernoulli variable
172                     # K = -minimize_scalar(lambda x: - integrated_drift(x),
173                     #                      bounds=(y-theta, y+theta),
                            method='bounded').fun
174
175                     # previously computed location of global maximum
176                     K_max = 1.382436647017447
177
178                     if y-theta < K_max < y+theta:
179                         K = 0.011016
180                     else:
181                         K = max(integrated_drift(y-theta), integrated_drift
                                (y+theta))
```

```
182
183                     w = candidate_bridge[min(tau, T - t)]
184                     test_factor_1 = (np.log(u) < integrated_drift(y + w) -
                            K)
185
186                     # second Bernoulli variable
187                     S = max(np.exp(-phi_min * (T - t)), 1)
188                     test_factor_2 = (v * S < np.exp(-phi_min * min(tau, T -
                            t)))
189
190                     # thinning test process
191                     test_factor_3 = True
192                     j = 0
193                     while j < len(test_times):
194
195                         u = np.random.rand()
196                         w = candidate_bridge[test_times[j]]
197                         if u * (phi_max - phi_min) < phi_y(y + w) - phi_min
                                :
198
199                             # one of the test jump times \kappa_j is
                                    accepted and we reject the skeleton
200                             test_factor_3 = False
201                             break
202
203                         j += 1
204
205                     if test_factor_1 and test_factor_2 and test_factor_3:
206
207                         # the sample is accepted
208                         rejected = False
209
210                         if stopping_time < T:
211                             current_t = t
212                             for jump in exponential_times:
213
214                                 while cap_counter < len(cap_limits) - 1:
215                                     t_cap = cap_limits[cap_counter] - t
216                                     if current_t < t + t_cap < t + jump:
217                                         w = candidate_bridge[t_cap]
218                                         cap_value += g * max(f_inv(y + w) -
                                                strike, 0) * np.exp(-x_down *
                                                t_cap)
219                                         cap_counter += 1
220                                     else:
221                                         break
222
```

```
                    w = candidate_bridge[jump]
                    g = g * (1 - (f_inv(y + w) - x_down)/(x_up
                        - x_down))
                    current_t += jump

                while cap_counter < len(cap_limits) - 1:
                    if cap_limits[cap_counter] < stopping_time:
                        t_cap = cap_limits[cap_counter] - t
                        w = candidate_bridge[t_cap]
                        cap_value += g * max(f_inv(y + w) -
                            strike, 0) * np.exp(-x_down * t_cap)
                        cap_counter += 1
                    else:
                        break

                g = g * np.exp(-x_down * tau)

                y += candidate_bridge_values[-1]
                t = stopping_time

            else:
                current_t = t

                for jump in exponential_times:

                    if T - t < jump:
                        break

                    # the while loop covers multiple cap
                        payments between jumps
                    while cap_counter < len(cap_limits) - 1:
                        t_cap = cap_limits[cap_counter] - t
                        if current_t < t + t_cap < t + jump:
                            w = candidate_bridge[t_cap]
                            cap_value += g * max(f_inv(y + w) -
                                strike, 0) * np.exp(-x_down *
                                t_cap)
                            cap_counter += 1
                        else:
                            break

                    w = candidate_bridge[jump]
                    g = g * (1 - (f_inv(y + w) - x_down)/(x_up
                        - x_down))
                    current_t += jump

                while cap_counter < len(cap_limits) - 1:
```

```
                        # we iterate over the missing cap payments between the last jump and the stopping time
                        t_cap = cap_limits[cap_counter] - t
                        w = candidate_bridge[t_cap]

                        cap_value += g * max(f_inv(y + w) - strike, 0) * np.exp(-x_down * t_cap)
                        cap_counter += 1

                    w = candidate_bridge[T-t]
                    y += w
                    g = g * np.exp(-x_down * (T - t))
                    t = T

                    cap_value += g * max(f_inv(y)-strike, 0)

            else:
                # one of the jump times was accepted and we test the skeleton up to the jump time
                stopping_time = t + candidate_times[i]

                u, v = np.random.rand(2)

                # first Bernoulli variable
                # K = -minimize_scalar(lambda x: - integrated_drift(x),
                #                      bounds=(y-theta, y+theta),
                #                      method='bounded').fun

                # previously computed location of global maximum
                K_max = 1.382436647017447

                if y - theta < K_max < y + theta:
                    K = 0.011016
                else:
                    K = max(integrated_drift(y - theta),
                            integrated_drift(y + theta))

                w = candidate_bridge[candidate_times[i]]
                test_factor_1 = (np.log(u) < integrated_drift(y + w) - K)

                # second Bernoulli variable
                S = max(np.exp(-phi_min * (T - t)), 1)
                test_factor_2 = (v * S < np.exp(-phi_min * candidate_times[i]))

```

```
303                     test_factor_3 = True
304                     j = 0
305                     while j < len(test_times):
306
307                         # we only need to test up to default time
308                         if test_times[j] > candidate_times[i]:
309                             break
310
311                         u = np.random.rand()
312                         w = candidate_bridge[test_times[j]]
313
314                         if u * (phi_max - phi_min) < phi_y(y + w) - phi_min
                                :
315                             test_factor_3 = False
316                             break
317
318                         j += 1
319
320                     if test_factor_1 and test_factor_2 and test_factor_3:
321                         # we accept the skeleton up to jump time
322                         rejected = False
323
324                         current_t = t
325                         for jump in exponential_times:
326                             if candidate_times[i] < jump:
327                                 break
328
329                             while cap_counter < len(cap_limits) - 1:
330                                 t_cap = cap_limits[cap_counter] - t
331                                 if current_t < t + t_cap < t + jump:
332                                     w = candidate_bridge[t_cap]
333                                     cap_value += g * max(f_inv(y + w) -
                                            strike, 0) * np.exp(-x_down * t_cap)
334                                     cap_counter += 1
335                                 else:
336                                     break
337
338                             w = candidate_bridge[jump]
339                             g = g * (1 - (f_inv(y + w) - x_down) / (x_up -
                                    x_down))
340                             current_t += jump
341
342                         while cap_counter < len(cap_limits) - 1:
343                             if cap_limits[cap_counter] < stopping_time:
344                                 t_cap = cap_limits[cap_counter] - t
345                                 w = candidate_bridge[t_cap]
```

```
346                                 cap_value += g * max(f_inv(y + w) - strike,
                                        0) * np.exp(-x_down * t_cap)
347                                 cap_counter += 1
348                             else:
349                                 break
350
351                         g = g * np.exp(- x_down * candidate_times[i])
352
353                         # compute Y immediately before the jump
354                         w = candidate_bridge[candidate_times[i]]
355                         y += w
356
357                         # compute Y after the jump
358                         u = np.random.rand()
359                         z = min_jump + (max_jump - min_jump) * u
360                         y += delta_y(y, z)
361                         t = stopping_time
362
363         cap_results.append(cap_value)
364         sim_count += 1
365         if sim_count in sim_frames:
366             print(sim_count)
367             cap_price = np.mean(cap_results)
368             sample_std = np.std(cap_results)
369             std_error = sample_std / np.sqrt(sim_count)
370             time_spent = time.time() - start_time
371             sim_data.append([sim_count, cap_price, std_error, time_spent])
372     return sim_data
```

A.4.2. Discretization methods

A.4.2.1. *Zero coupon bond* Discretization method 1:

```
1  import numpy as np
2  import sampling
3  import time
4
5
6  # model parameters
7
8  lamb0 = 0.0110
9  lamb1 = 0.1000
10
11 min_jump = 0.0113
12 max_jump = 0.0312
13
14 X_0 = 0.0422
15 mean = 0.0422
```

```
16 kappa = 0.0117
17 sigma = 0.0130
18
19
20 def jump_int(x):
21     return lamb0 + lamb1 * max(x, 0)
22
23
24 T = 3
25
26
27 def monte_carlo(sim_frames=[1000]):
28
29     sim_data = []
30
31     for n_sim in sim_frames:
32         sim_count = 0
33         results = []
34
35         N = int(np.sqrt(n_sim))
36         h = T/N
37
38
39         start_time = time.time()
40
41         while sim_count < n_sim:
42             i = 0
43             g = 0
44
45             compensator = 0
46             tau = sampling.generate_exp()
47
48             X = X_0
49
50             while i < N+1:
51                 u = np.random.normal()
52                 X_next = X + kappa*(mean - max(X, 0))*h + sigma*np.sqrt(h*
                       max(X, 0))*u
53
54                 compensator += h * jump_int(X_next)
55                 if compensator > tau:
56                     u = np.random.rand()
57                     z = min_jump + (max_jump - min_jump)*u
58                     X_next += z
59
60                     tau += sampling.generate_exp()
61
```

```
                g += h*(X + X_next)/2
                X = X_next
                i += 1

            bond_result = np.exp(-g)
            results.append(bond_result)
            sim_count += 1

        print(n_sim)
        bond_price = np.mean(results)
        sample_std = np.std(results)
        std_error = sample_std/np.sqrt(n_sim)
        time_spent = time.time() - start_time

        sim_data.append([n_sim, bond_price, std_error, time_spent])

    return sim_data
```

Discretization method 2:

```
import numpy as np
import sampling
import time


# model parameters

lamb0 = 0.0110
lamb1 = 0.1000

min_jump = 0.0113
max_jump = 0.0312

X_0 = 0.0422
mean = 0.0422
kappa = 0.0117
sigma = 0.0130


def jump_int(x):
    return lamb0 + lamb1 * max(x, 0)


T = 3


def monte_carlo(sim_frames=[1000]):

```

```
29      sim_data = []
30
31      for n_sim in sim_frames:
32          sim_count = 0
33          results = []
34
35          N = int(np.sqrt(n_sim))
36          h = T/N
37
38          start_time = time.time()
39
40          while sim_count < n_sim:
41              i = 0
42              g = 0
43
44              compensator = 0
45              tau = sampling.generate_exp()
46
47              X = X_0
48
49              while i < N+1:
50                  u = np.random.normal()
51                  X_next = ((1 - kappa*h/2)*np.sqrt(X) + sigma*np.sqrt(h)*u
                        /(2*(1 - kappa*h/2)))**2 + h*(kappa*mean - sigma**2/4)
52
53                  compensator += h * jump_int(X_next)
54                  if compensator > tau:
55                      u = np.random.rand()
56                      z = min_jump + (max_jump - min_jump)*u
57                      X_next += z
58
59                      tau += sampling.generate_exp()
60
61                  g += h*(X + X_next)/2
62                  X = X_next
63                  i += 1
64
65              bond_result = np.exp(-g)
66              results.append(bond_result)
67              sim_count += 1
68
69          print(sim_count)
70          bond_price = np.mean(results)
71          sample_std = np.std(results)
72          std_error = sample_std/np.sqrt(n_sim)
73          time_spent = time.time() - start_time
74
```

```
75          sim_data.append([n_sim, bond_price, std_error, time_spent])
76
77      return sim_data
```

Discretization method 3:

```
1  import numpy as np
2  import sampling
3  import time
4
5  # model parameters
6
7  lamb0 = 0.0110
8  lamb1 = 0.1000
9
10 min_jump = 0.0113
11 max_jump = 0.0312
12
13 X_0 = 0.0422
14 mean = 0.0422
15 kappa = 0.0117
16 sigma = 0.0130
17
18
19 def jump_int(x):
20     return lamb0 + lamb1 * max(x, 0)
21
22
23 T = 3
24
25
26 def monte_carlo(sim_frames=[1000]):
27
28     sim_data = []
29     for n_sim in sim_frames:
30         results = []
31         N = int(n_sim**(1/4))
32         h = T / N
33
34         start_time = time.time()
35         sim_count = 0
36         while sim_count < n_sim:
37             i = 0
38             g = 0
39
40             compensator = 0
41             tau = sampling.generate_exp()
42
```

```
            X = X_0

            while i < N + 1:
                u = np.random.normal()
                X_next = np.exp(-kappa * h / 2) * (np.sqrt((mean * kappa -
                    sigma ** 2 / 4) *
                                                           (1 - np.exp(-
                                                               kappa * h /
                                                               2)) / kappa +
                                                          np.exp(-kappa *
                                                               h / 2) * X) +
                                                            sigma * np.
                                                           sqrt(h) * u /
                                                            2) ** 2 \
                        + (kappa*mean - sigma**2/4)*(1 - np.exp(kappa*h/2)
                            )/kappa

                compensator += h * jump_int(X_next)
                if compensator > tau:
                    u = np.random.rand()
                    z = min_jump + (max_jump - min_jump) * u
                    X_next += z

                    tau += sampling.generate_exp()

                g += h * (X + X_next) / 2
                X = X_next
                i += 1

            bond_result = np.exp(-g)
            results.append(bond_result)
            sim_count += 1

        print(n_sim)
        bond_price = np.mean(results)
        sample_std = np.std(results)
        std_error = sample_std / np.sqrt(n_sim)
        time_spent = time.time() - start_time

        sim_data.append([n_sim, bond_price, std_error, time_spent])

    return sim_data
```

A.4.2.2. *Cap* Discretization method 1:

```
import numpy as np
import sampling
import time
```

```
4
5
6  # model parameters
7
8  lamb0 = 0.0110
9  lamb1 = 0.1000
10
11 min_jump = 0.0113
12 max_jump = 0.0312
13
14 X_0 = 0.0422
15 mean = 0.0422
16 kappa = 0.0117
17 sigma = 0.0130
18
19
20 def jump_int(x):
21     return lamb0 + lamb1 * max(x, 0)
22
23
24 T = 3
25 strike = 0.05
26 cap_limits = [1, 2, 3]
27
28
29 def monte_carlo(sim_frames=[1000]):
30
31     sim_data = []
32     for n_sim in sim_frames:
33         results = []
34         N = int(np.sqrt(n_sim))
35         h = T/N
36
37         start_time = time.time()
38         sim_count = 0
39         while sim_count < n_sim:
40             i = 0
41             g = 0
42             cap_counter = 0
43             bond_result = 0
44             compensator = 0
45             tau = sampling.generate_exp()
46             X = X_0
47
48             while i < N:
49                 u = np.random.normal()
```

```
                X_next = X + kappa*(mean - max(X, 0))*h + sigma*np.sqrt(h*
                    max(X, 0))*u

                compensator += h * jump_int(X_next)
                if compensator > tau:
                    u = np.random.rand()
                    z = min_jump + (max_jump - min_jump)*u
                    X_next += z

                    tau += sampling.generate_exp()

                if i+1 >= N*cap_limits[cap_counter]/3:
                    bond_result += np.exp(-g)*max(X_next-strike, 0)
                    cap_counter += 1

                g += h * (X + X_next) / 2
                X = X_next
                i += 1

            results.append(bond_result)
            sim_count += 1

        print(n_sim)
        bond_price = np.mean(results)
        sample_std = np.std(results)
        std_error = sample_std/np.sqrt(n_sim)
        time_spent = time.time() - start_time

        sim_data.append([n_sim, bond_price, std_error, time_spent])

    return sim_data
```

Discretization method 2:

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


# model parameters

lamb0 = 0.0110
lamb1 = 0.1000

min_jump = 0.0113
max_jump = 0.0312

```

```
15 X_0 = 0.0422
16 mean = 0.0422
17 kappa = 0.0117
18 sigma = 0.0130
19
20
21 def jump_int(x):
22     return lamb0 + lamb1 * max(x, 0)
23
24
25 T = 3
26 strike = 0.05
27 cap_limits = [1, 2, 3]
28
29
30 def monte_carlo(sim_frames=[1000]):
31     sim_data = []
32     for n_sim in sim_frames:
33
34         sim_count = 0
35         results = []
36
37         N = int(np.sqrt(n_sim))
38         h = T/N
39
40         start_time = time.time()
41
42         while sim_count < n_sim:
43             i = 0
44             g = 0
45             cap_counter = 0
46             bond_result = 0
47             compensator = 0
48             tau = sampling.generate_exp()
49
50             X = X_0
51
52             while i < N:
53                 u = np.random.normal()
54                 X_next = ((1 - kappa*h/2)*np.sqrt(X) + sigma*np.sqrt(h)*u
                       /(2*(1 - kappa*h/2)))**2 + h*(kappa*mean - sigma**2/4)
55
56                 compensator += h * jump_int(X_next)
57                 if compensator > tau:
58                     u = np.random.rand()
59                     z = min_jump + (max_jump - min_jump)*u
60                     X_next += z
```

```
61
62                      tau += sampling.generate_exp()
63
64                  if i+1 >= N*cap_limits[cap_counter]/3:
65                      bond_result += np.exp(-g)*max(X_next-strike, 0)
66                      cap_counter += 1
67
68                  g += h * (X + X_next) / 2
69                  X = X_next
70                  i += 1
71
72              results.append(bond_result)
73              sim_count += 1
74
75          print(sim_count)
76          bond_price = np.mean(results)
77          sample_std = np.std(results)
78          std_error = sample_std/np.sqrt(n_sim)
79          time_spent = time.time() - start_time
80
81          sim_data.append([n_sim, bond_price, std_error, time_spent])
82
83      return sim_data
```

Discretization method 3:

```
1  import numpy as np
2  import matplotlib.pyplot as plt
3  import sampling
4  import time
5
6
7  # model parameters
8
9  lamb0 = 0.0110
10 lamb1 = 0.1000
11
12 min_jump = 0.0113
13 max_jump = 0.0312
14
15 X_0 = 0.0422
16 mean = 0.0422
17 kappa = 0.0117
18 sigma = 0.0130
19
20
21 def jump_int(x):
22     return lamb0 + lamb1 * max(x, 0)
```

```
23
24
25 T = 3
26 strike = 0.05
27 cap_limits = [1, 2, 3]
28
29
30 def monte_carlo(sim_frames=[1000]):
31
32     sim_data = []
33     for n_sim in sim_frames:
34         sim_count = 0
35         results = []
36
37         N = int(n_sim**(1/4))
38         h = T/N
39
40         start_time = time.time()
41
42         while sim_count < n_sim:
43             i = 0
44             g = 0
45             cap_counter = 0
46             bond_result = 0
47             compensator = 0
48             tau = sampling.generate_exp()
49
50             X = X_0
51
52             while i < N:
53                 u = np.random.normal()
54                 X_next = np.exp(-kappa * h / 2) * (np.sqrt((mean * kappa -
                       sigma ** 2 / 4) *
55                                                            (1 - np.exp(-
                                                               kappa * h /
                                                               2)) / kappa +
56                                                           np.exp(-kappa *
                                                               h / 2) * X) +
                                                                sigma * np.
                                                               sqrt(h) * u /
                                                                2) ** 2 \
57                         + (kappa * mean - sigma ** 2 / 4) * (1 - np.exp(
                               kappa * h / 2)) / kappa
58
59                 compensator += h * jump_int(X_next)
60                 if compensator > tau:
61                     u = np.random.rand()
```

```
62                      z = min_jump + (max_jump - min_jump)*u
63                      X_next += z
64
65                      tau += sampling.generate_exp()
66
67                  if i+1 >= N*cap_limits[cap_counter]/3:
68                      bond_result += np.exp(-g)*max(X_next-strike, 0)
69                      cap_counter += 1
70
71                  g += h * (X + X_next) / 2
72                  X = X_next
73                  i += 1
74
75              results.append(bond_result)
76              sim_count += 1
77
78          print(n_sim)
79          bond_price = np.mean(results)
80          sample_std = np.std(results)
81          std_error = sample_std/np.sqrt(n_sim)
82          time_spent = time.time() - start_time
83
84          sim_data.append([n_sim, bond_price, std_error, time_spent])
85
86      return sim_data
```

References

[1] Ahrens, J. H. and Dieter, U. (1982), "Generating gamma variates by a modified rejection technique", *Communications of the ACM*, 25 (1), 47-54

[2] Alfonsi, A. (2005), "On the discretization schemes for the CIR (and Bessel squared) processes", *Monte Carlo Methods Appl.*, 11 (4), 355–384

[3] Alfonsi, A. (2010), "High order discretization schemes for the CIR process: Application to affine term structure and Heston models", *Math. Comput.*, 79 (269), 209–237

[4] Arnsdorf, M. and Halperin, I. (2008), "BSLP: Markovian Bivariate Spread-Loss Model for Portfolio Credit Derivatives", *Journal of Computational Finance*, 12, 77-100

[5] Bertoin, J. (1996), "Lévy Processes", *Cambridge University Press*, 0-521-56243-0

[6] Beskos, A. and Roberts, G. O. (2005), "Exact Simulation of Diffusions", *The Annals of Applied Probability*, 15 (4), 2422-2444

[7] Black, F. and Scholes, M. (1973), "The pricing of options and corporate liabilities", *Journal of Political Economy*, 81 (3), 637-654

[8] Blanchet, J. and Zhang, F. (2020), "Exact Simulation for Multivariate Itô Diffusions ", *Advances in Applied Probability*, 52 (4), 1003-1034

[9] Broadie, M. and Kaya, O. (2006), "Exact simulation of stochastic volatility and other affine jump diffusion processes", *Operations Research*, 54 (2), 217–231

[10] Burq, Z. and Jones, O. (2008), "Simulation of Brownian motion at first-passage times", *Mathematics and Computers in Simulation*, 77, 64–81

[11] Carr, P. and Linetsky, V. (2006), "A Jump to Default Extended CEV Model: An Application of Bessel Processes", *"Finance Stoch.*, 10, 303–330

[12] Chen, N. and Huang, Z. (2013), "Localization and exact simulation of Brownian motion-driven stochastic differential equations", *Operations Research*, 38 (3), 591–616

[13] Cox, J. (1975, 1996), "Notes on Option Pricing I: Constant Elasticity of Variance Diffusions", *reprinted in The Journal of Portfolio Management*, 23, 15–17

[14] Ding, X., Giesecke, K. and Tomecek, P. (2009), "Time-changed birth processes and multiname credit derivatives", *Operations Research*, 57 (4), 990–1005

[15] Diop, S. and Pascucci, A. (2018), "CDS calibration under an extended JDCEV model", *International Journal of Computer Mathematics*, 96 (9), 1-22
Sidy Diop Andrea Pascucci

[16] Duffie, D. and Glynn, P. (1995), "Efficient Monte Carlo simulation of security prices", *Annals of Applied Probability*, 5 (4), 897-905

[17] Duffie, D., Pan, J. and Singleton K. (2000), "Transform analysis and asset pricing for affine jump-diffusions", *Econometrica*, 68, 1343–1376

[18] Duffie, D. and Singleton, K. J. (1999), "Modeling term structures of defaultable bonds", *Review of Financial Studies*, 12, 687–720

[19] Gerstner, T. and Kloeden, P. (2012), "Recent Developments In Computational Finance: Foundations, Algorithms And Applications", *World Scientific*, 978-981-4436-42-7

[20] Giesecke, K. and Smelov, D. (2013), "Exact Sampling of Jump Diffusions", *Operations Research*, 61 (4), 894-907

[21] Glasserman, P. (2003), "Monte Carlo Methods in Financial Engineering", *Springer-Verlag New York*, 978-0-387-21617-1

[22] Heston, S. (1993), "A closed-form solution for options with stochastic volatility with applications to bond and currency options", *Review of Financial Studies*, 6 (2), 327-343

[23] Higham, D.J. (2006), "An Algorithmic Introduction to Numerical Simulation of Stochastic Differential Equations", *SIAM Review*, 43 (3), 525–546

[24] Hull, J. C. (2014), "Options, Futures and Other Derivatives", *Pearson*

[25] Ikeda, N. and Watanabe, S. (1989), "Stochastic Differential Equations and Diffusion Processes", *North Holland Publishing Company, New York*

[26] Imhof, P. (1984), "Density factorizations for Brownian motion, meander and the three-dimensional Bessel process", *Journal of Applied Probability*, 21, 500–510

[27] Jeanblanc, M., Yor, M., and Chesney, M. (2009), "Recent Developments In Computational Finance: Foundations, Algorithms And Applications", *Springer Finance*, 10.1007/978-1-84628-737-48

[28] Last, G. and Penrose, M. (2017), "Lectures on the Poisson Process", *Cambridge University Press*

[29] Longstaff, F. and Schwartz, E. (2001), "Valuing American Options by Simulation: A Simple Least-Squares Approach", *Review of Financial Studies*, 14, 113-147

[30] Lord, R., Koekkoek, R. and Van Dijk, D. (2010), "A comparison of biased simulation schemes for stochastic volatility models", *Quantitative Finance*, 10 (2), 177–194

[31] Marsaglia, G. Tsang, W. W. (2000), "The Ziggurat Method for Generating Random Variable", *Journal of Statistical Software*, 5 (8)

[32] Merton, R.C. (1973), "Theory of Rational Option Pricing", *Bell Journal of Economics and Management Science* , 4 (1), 141-183

[33] Meyer, P-A. (1971), "Démonstration simplifée d'un théorème de Knight", *Séminaire de Probabilités V. Lecture Notes in Mathematics, Vol. 191 (Springer-Verlag, Berlin)*, 191–195

[34] Ribeiro, C. Webber, N. (2006), "Correcting for Simulation Bias in Monte Carlo Methods to Value Exotic Options in Models Driven by Lévy Processes", *Applied Mathematical Finance*, 13 (4) 333-352

[35] Ruf, J. and Scherer, M. (2011), "Pricing corporate bonds in an arbitrary jump-diffusion model based on an improved Brownian bridge algorithm", *Journal of Computational Finance*, 14 (3), 30–45

[36] Schoutens, W. (2003), "Lévy Processes in Finance: Pricing Financial Derivatives", *John Wiley Sons, Ltd.*, 0-470-85156-2

[37] Williams, D. (1970), "Decomposing the Brownian path", *Bulletin of the American Mathematical Society*, 76, 871–873

[38] Zhou, H. (2003), "Jump-diffusion term structure and Ito conditional moment generator", *Journal of Financial Econometrics*, 1 (2), 250–271