Numerical Simulations of Linear Stochastic Oscillators

driven by Wiener and Poisson processes

André Berglund

May 12, 2017
Examensarbete, 30hp, VT 2017
Masterexamen i matematik, 120hp
Institutionen för matematik och matematisk statistik
ABSTRACT

The main component of this essay is the numerical analysis of stochastic differential equations driven by Wiener and Poisson processes. In order to do this, we focus on two model problems, the geometric Brownian motion and the linear stochastic oscillator, studied in the literature for stochastic differential equations only driven by a Wiener process.

This essay covers theoretical as well as numerical investigations of jump - or more specifically, Poisson - processes and how they influence the above model problems.

SAMMANFATTNING


Den här uppsatsen täcker teoretiska samt numeriska undersökningar av hopp - eller mer specifikt, Poisson - processer och hur de påverkar de ovan nämnda modellproblemen.
## Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abstract</td>
<td>1</td>
</tr>
<tr>
<td>Sammanfattning</td>
<td>3</td>
</tr>
<tr>
<td>1. Introduction</td>
<td>5</td>
</tr>
<tr>
<td>2. Acknowledgements</td>
<td>7</td>
</tr>
<tr>
<td>3. Glossary</td>
<td>7</td>
</tr>
<tr>
<td>4. Probability theory, random variables and stochastic processes</td>
<td>7</td>
</tr>
<tr>
<td>4.1. Probability theory</td>
<td>7</td>
</tr>
<tr>
<td>4.2. Random variables</td>
<td>7</td>
</tr>
<tr>
<td>4.3. Stochastic processes</td>
<td>9</td>
</tr>
<tr>
<td>5. Stochastic differential equations driven by Wiener processes</td>
<td>11</td>
</tr>
<tr>
<td>5.1. The Itô integral and the Itô formula</td>
<td>12</td>
</tr>
<tr>
<td>5.2. Additional theorems concerning Wiener processes</td>
<td>13</td>
</tr>
<tr>
<td>6. Stochastic differential equations driven by Wiener and Poisson processes</td>
<td>15</td>
</tr>
<tr>
<td>6.1. The Itô integral and the Itô formula for Poisson processes</td>
<td>20</td>
</tr>
<tr>
<td>6.2. Additional theorems concerning Poisson processes</td>
<td>21</td>
</tr>
<tr>
<td>7. General numerical schemes</td>
<td>25</td>
</tr>
<tr>
<td>7.1. Concepts of convergence</td>
<td>26</td>
</tr>
<tr>
<td>8. The geometric Brownian motion</td>
<td>27</td>
</tr>
<tr>
<td>8.1. Geometric Brownian motion - Background and definitions</td>
<td>27</td>
</tr>
<tr>
<td>8.2. GBPM - Model specific numerical schemes</td>
<td>27</td>
</tr>
<tr>
<td>8.3. GBPM - Exact solutions</td>
<td>28</td>
</tr>
<tr>
<td>8.4. Additional comments</td>
<td>33</td>
</tr>
<tr>
<td>9. The linear stochastic oscillator</td>
<td>35</td>
</tr>
<tr>
<td>9.1. LSO - Background and definitions</td>
<td>35</td>
</tr>
<tr>
<td>9.2. LSO - Model specific numerical schemes</td>
<td>35</td>
</tr>
<tr>
<td>9.3. Linear stochastic oscillator - Preparatory lemmas</td>
<td>37</td>
</tr>
<tr>
<td>9.4. The linear stochastic oscillator driven by a Wiener process</td>
<td>38</td>
</tr>
<tr>
<td>9.5. The linear stochastic oscillator driven by a Wiener process and either a CPoPr or a CCPoPr</td>
<td>45</td>
</tr>
<tr>
<td>10. Final words</td>
<td>67</td>
</tr>
<tr>
<td>10.1. Summary of the numerical results</td>
<td>67</td>
</tr>
<tr>
<td>10.2. Theoretical summary</td>
<td>67</td>
</tr>
<tr>
<td>10.3. Further research</td>
<td>69</td>
</tr>
<tr>
<td>References</td>
<td>71</td>
</tr>
<tr>
<td>Appendices</td>
<td>73</td>
</tr>
<tr>
<td>A. The Lambert W function</td>
<td>73</td>
</tr>
<tr>
<td>B. Strong error estimations of the linear stochastic oscillator - Driven by a CPoPr</td>
<td>75</td>
</tr>
<tr>
<td>C. Strong error estimations of the linear stochastic oscillator - Driven by a CCPoPr</td>
<td>79</td>
</tr>
<tr>
<td>D. Energy weak error estimations of the linear stochastic oscillator - Driven by a CPoPr</td>
<td>83</td>
</tr>
<tr>
<td>E. Energy weak error estimations of the linear stochastic oscillator - Driven by a CCPoPr</td>
<td>91</td>
</tr>
<tr>
<td>F. Code - Process batch simulators</td>
<td>97</td>
</tr>
<tr>
<td>G. Code - General step functions</td>
<td>99</td>
</tr>
<tr>
<td>H. Code - Geometric differential equation step functions, as used in the GBPM</td>
<td>103</td>
</tr>
<tr>
<td>I. Code - Matrix differential equation step functions, as used in the LBPO</td>
<td>105</td>
</tr>
<tr>
<td>J. Code - Illustrations</td>
<td>111</td>
</tr>
<tr>
<td>K. Code - Plot and print functions</td>
<td>157</td>
</tr>
<tr>
<td>L. Code - Miscellaneous functions and scripts</td>
<td>161</td>
</tr>
</tbody>
</table>
1. Introduction

The subject of differential equations came from the need to solve physical and astronomical problems. Early examples of such problems were vibrations of strings or the swinging of pendulums. Many famous physicists and mathematicians throughout the later history were drawn to those problems. We see names such as Leibniz, the Bernoulli brothers, Euler, Lagrange and Laplace to mention a few, appear repeatedly as they develop a multitude of models and methods to deal with said problems. Interested readers are referred to [12, Volume 2, Chapter 21].

The late 17th century is where the earliest work concerning differential equations can be recognized. Some general theory can be tracked back to as early as 1676, where Newton stated that, for a function \( f \), the solution \( y \) to

\[
\frac{d^n y}{dx^n} = f(x)
\]

was indifferent to an addition of \((n - 1)\) degree polynomials. In 1691 we see the study of second order ordinary differential equations, where Jakob Bernoulli tried to describe the shape of a sail in the wind, began. However it took until 1728 for Euler to start the systematic investigation of second order differential equations. In 1743, Euler explains the solution to the homogeneous equation

\[
C_1 y + C_2 \frac{dy}{dx} + C_3 \frac{d^2 y}{dx^2} + \ldots + C_{n+1} \frac{d^n y}{dx^n} = 0.
\]

From [1] we see that in the early 20th century the concept of the Wiener process, also called the standard Brownian motion, came into being. As early 1900 the French mathematician Louis Bachelier introduced a process to describe bond price fluctuations, which was later recognized as equivalent to the Wiener process. During the period 1905 – 1909 the Brownian motion was studied by Albert Einstein and the Polish physicist Marian Smoluchowski as a physical phenomenon. Their work was later expanded upon by the French physicist Paul Langevin, who has been attributed the first stochastic differential equations (SDE’s).

The present essay will initially assume that the reader is familiar with probability theory, random variables and stochastic processes. This is done in order to not go into details which do not clearly fall within the scope of this essay. When needed, we will introduce necessary definitions and references. This will mainly be done within the introductory sections, Section 4, Section 5 and Section 6. A flowchart over these sections can be seen in Figure 1.1. No similar flowchart will be included for the rest of the sections as no simple, planar, layout was found.

Due to the lengthy nature of the underlying theory regarding stochastic processes a lot of details have been overlooked here, in order to promote brevity. A thorough investigation of the standard Wiener process properties and continuing theory can be found in [13]. Likewise, for theory concerning Poisson processes, the reader is referred to [17]. We see in [17, Chapter 1.4] how they tie the Itô integrals to the concept of semi-martingales and finally to jump measures and Poisson processes.

It is usual to study a problem of an less complex nature before studying challenging problems. We have done that here, starting with the geometric Brownian motion, seen in Section 8, before moving on to the linear stochastic oscillator, seen in Section 9. We will look at those driven by both a Wiener and a Poisson process (or a variant thereof). The focus will be on a numerical study of those models, using schemes and error concepts given in Section 7.

Relevant theorems, regarding the linear stochastic oscillator driven by a Wiener process, can be seen in both [2] and [22]. This essay will take a number of these theorems and generalize them to the linear stochastic oscillator driven by both a Wiener and a Poisson process (or a variant thereof). A special note regarding [22] is that they observe the special case with a trivial periodicity, a practice which we will not continue. The expected value of the energy behaves differently depending on whether the linear stochastic oscillator is driven by a compound Poisson process or a compensated compound Poisson process.

The numerical confirmations can be reproduced by the code which is included in the appendix. It is commented and written in such a way that the reader can instantly reproduce and, with minimal effort, modify it to suit computational time restrictions.
Figure 1.1. Flowchart of Section 4, Section 5 and Section 6.
2. Acknowledgements

This essay would have turned out very differently without the help of a few persons in particular. First, my dear Veronica Gustavsson, without whom each day would lack its luster. Secondly, David Cohen, whose efforts and feedback gave this essay a quality far beyond what it would have had. Thirdly, Per Åhag, for the discussions and gossip which always entertain and energize. Lastly, Anton Vernersson, for his valuable opposition of this essay.
3. Glossary

Characteristic function:
   Given a set $X$ and a subset $A \subseteq X$, a function $\mathbb{1} : X \rightarrow \{0, 1\}$ such that
   \[
   \mathbb{1}_A(x) = \begin{cases}
   1, & x \in A \\
   0, & x \not\in A
   \end{cases}
   .
   
Real numbers:
   The set of real numbers, or the real line, is denoted by $\mathbb{R}$.

Integers:
   The set of integers, or $0, \pm1, \pm2, \ldots$, is denoted by $\mathbb{Z}$.

Natural numbers:
   The set of natural numbers (including 0), or $0, 1, 2, \ldots$, is denoted by $\mathbb{N}$.

Non-negative subset:
   The non-negative subset of a set $X$, where applicable, is denoted by $X^+$.

Cartesian product:
   For an integer $n$, the Cartesian product of a set $X$ to the $n$’th power is denoted by $X^n$.

Complement:
   For a set $X$, the complement of the set is denoted by $X^C$.

Index:
   For $x \in \mathbb{R}^d$ the $i$’th index of $x$ is denoted by $x^i$.

Probability measure:
   $P$ will be reserved for probability measures.

Probability density function:
   Referred to as pdf, $f_X$ will be reserved for the pdf.

Cumulative probability density function:
   Referred to as cdf, $F_X$ will be reserved for the cdf.

Time indexing:
   Depending on context we will use subscript as a discretization index counter or a time counter.
   E.g. for a $d$-dimensional stochastic process $X$ we can refer to the random variable at a specific time
   as $X_t$.

Independent and identically distributed:
   Random variables which are independent and from the same distribution are referred to as iid.

Expected value:
   For a random variable $X$ the expected value is written as $E[X]$.

Variance:
   For a random variable $X$ the variance is written as $\text{Var}[X]$.

SDE:
   A stochastic differential equation is referred to as an SDE.

Derivative:
   Newton’s notation for differentiation, or dot notation, will be used within this essay. E.g., for a
   function $f: \mathbb{R} \rightarrow \mathbb{R}$, we would write
   \[
   \frac{df(t)}{dt} = f'(t).
   \]

The Poisson process:
   Omitting the intensity, it is referred to as the PoPr, $\mathcal{N}$ will be reserved for the Poisson process.

The compound Poisson process:
   Omitting the intensity and the distribution, it is referred to as the CPoPr, $\hat{\mathcal{N}}$ will be reserved for
   the compound Poisson process.

The compensated compound Poisson process:
   Omitting the intensity and the distribution, it is referred to as the CCPoPr, $\tilde{\mathcal{N}}$ will be reserved
   for the compensated compound Poisson process.
Either a CPoPr or a CCPoPr:
\( N^* \) will be reserved for when a process is either a CPoPr or a CCPoPr.

The geometric motion:
Referred to as the GM.

The GM driven by a Wiener process:
Referred to as the GBM. Also known as the geometric Brownian motion.

The GM driven by a Wiener process and either a CPoPr or a CCPoPr:
Referred to as the GBPM.

The linear stochastic oscillator:
Referred to as the LSO.

The LSO driven by a Wiener process:
Referred to as the LBO.

The LSO driven by a Wiener process and either a CPoPr or a CCPoPr:
Referred to as the LBPO.
This section will quickly walk through what preliminary theory we will require for this essay. Should the reader wish to read more of this we refer them to e.g. [7], [10] or [20].

4. Probability theory. We will define a few terms required to properly define what a random variable is and we begin with the following.

Definition 4.1. Let $\Omega$ be a set. A collection $\mathcal{A}$ of subsets of $\Omega$ such that
- $\Omega \in \mathcal{A}$
- $A \in \mathcal{A} \Rightarrow A^C \in \mathcal{A}$
- $A_1, A_2, \ldots \in \mathcal{A} \Rightarrow \bigcup_{i=1}^{\infty} A_i \in \mathcal{A}$

is called a $\sigma$-algebra.

Definition 4.2. For a set $\Omega$ and a $\sigma$-algebra $\mathcal{A}$ of subsets of $\Omega$, the ordered pair $(\Omega, \mathcal{A})$ is called a measurable space.

Definition 4.3. Assume a measurable space $(\Omega, \mathcal{A})$ and a sequence $A_1, A_2, \ldots \in \mathcal{A}$ such that $A_i \cap A_j = \emptyset$ for $i \neq j$ and $i, j = 1, 2, \ldots$. Then a non-negative set function $\mu$ satisfying
- $\mu(\emptyset) = 0$
- $\mu\left(\bigcup_{i=1}^{\infty} A_i\right) = \sum_{i=1}^{\infty} \mu(A_i)$

is called a measure.

An example would be the Lebesgue measure, see e.g. [4] or [23]. However, we will focus on the following type of measures.

Definition 4.4. A measure $P$ on a measurable space $(\Omega, \mathcal{A})$ such that $P(\Omega) = 1$ is called a probability measure.

Notation 4.5. Throughout this essay $P$ will be reserved for probability measures.

A good example of different probability measures would be those used for e.g. discrete or continuous random variables.

Definition 4.6. Given a measurable space $(\Omega, \mathcal{A})$ with a corresponding probability measure $P$, the ordered triplet $(\Omega, \mathcal{A}, P)$ is called a probability space.

Definition 4.7. Given a probability space $(\Omega, \mathcal{A}, P)$ and an event $A \in \mathcal{A}$ such that $P(A^C) = 0$ we say that $A$ happens almost surely, or, in short, a.s.

4.2. Random variables. Albeit this being a special case, we will only use real valued random variables throughout this essay. Therefore we will present the following definition.

Definition 4.8. For a probability space $(\Omega, \mathcal{A}, P)$, the function $X : \Omega \to \mathbb{R}$ such that

$\{\omega \in \Omega : X(\omega) \leq a\} \in \mathcal{A}, \quad \text{for each } a \in \mathbb{R}$

is called a random variable.

With this we can define the following well known function.

Definition 4.9. For a random variable $X$, the function $F_X : \mathbb{R} \to [0,1]$ such that

$F_X(x) = P(X \leq x)$

is called the cumulative distribution function, or, in short, cdf.

With this we need to consider two cases, whether the random variable is discrete or continuous.
Definition 4.10. Take a random variable $X$ with the cdf $F_X$. If $X$ is a discrete random variable, define the function $f_X : \mathbb{R} \to [0, 1]$ such that
\[ f_X(x) = P(X = x). \]
If $X$ is a continuous random variable, define the function $f_X : \mathbb{R} \to [0, \infty]$ such that
\[ f_X(x) = \frac{dF_X(x)}{dx}. \]
The function $f$ is then called the probability density function, or, in short, pdf.

Notation 4.11. As with the probability measure $P$, for a random variable $X$ we will reserve the notation $F_X$ and $f_X$ for the cdf and the pdf respectively.

The reader should note that when the random variable $X$ is discrete the pdf is mapped onto $[0, 1]$ instead of $[0, \infty]$. This is because it is necessary for the sum of all probabilities to be one instead of the integral of all probabilities.

Much like for lone random variables we can describe behaviors for collections of random variables.

Definition 4.12. Given $n$ random variables $\{X_i\}_{i=1}^n$, the function $F_{X_1, X_2, \ldots, X_n} : \mathbb{R}^n \to [0, 1]$ such that
\[ F_{X_1, X_2, \ldots, X_n}(x_1, x_2, \ldots, x_n) = P(X_1 \leq x_1, X_2 \leq x_2, \ldots, X_n \leq x_n) \]
is called the joint cdf.

The joint pdf also translates into two different cases, depending on whether we have discrete or continuous random variables.

Definition 4.13. Take $n$ random variables $\{X_i\}_{i=1}^n$ with the joint cdf $F_{X_1, X_2, \ldots, X_n}$. If they are discrete, define the function $f_{X_1, X_2, \ldots, X_n} : \mathbb{R}^n \to [0, 1]$ such that
\[ f_{X_1, X_2, \ldots, X_n}(x_1, x_2, \ldots, x_n) = P(X_1 = x_1, X_2 = x_2, \ldots, X_n = x_n). \]
If they are continuous, assuming joint cdf $F_{X_1, X_2, \ldots, X_n}$, define the function $f_{X_1, X_2, \ldots, X_n} : \mathbb{R}^n \to [0, \infty]$ such that
\[ f_{X_1, X_2, \ldots, X_n}(x_1, x_2, \ldots, x_n) = \frac{\partial^n F_{X_1, X_2, \ldots, X_n}(x_1, x_2, \ldots, x_n)}{\partial X_1 \partial X_2 \cdots \partial X_n}. \]
The function $f$ is then called the joint pdf.

Given the recently defined terms it is a short step to give the following definition.

Definition 4.14. Given two random variables $X$ and $Y$. If, for the joint cdf $F_{X,Y}$,
\[ F_{X,Y}(x, y) = F_X(x)F_Y(y) \]
holds, we say that $X$ and $Y$ are independent.

Concerning the following two theorems, we will assume that the reader is comfortable with them, though they will be stated for the sake of completeness. We will not refer to the following two theorems each time they are used.

Theorem 4.15. Given two independent random variables $X_1, X_2$, the following equality holds
\[ E[X_1X_2] = E[X_1]E[X_2]. \]

Proof. Assuming continuous random variables, this follows analogously for the discrete case. We get, thanks to the independence, that
\[
E[X_1X_2] = \int_{-\infty}^{\infty} x_1x_2 \frac{\partial^2 F_{X_1, X_2}(x_1, x_2)}{\partial X_1 \partial X_2} dx_1 dx_2 = \int_{-\infty}^{\infty} x_1x_2 \frac{\partial^2 F_{X_1}(x_1)F_{X_2}(x_2)}{\partial X_1 \partial X_2} dx_1 dx_2 = \int_{-\infty}^{\infty} x_1 \frac{\partial F_{X_1}(x_1)}{\partial X_1} dx_1 \int_{-\infty}^{\infty} x_2 \frac{\partial F_{X_2}(x_2)}{\partial X_2} dx_2 = E[X_1]E[X_2]. \]

Theorem 4.16. Given a random variable $X$, the following equality holds
\[ \text{Var}[X] = E[X^2] - E[X]^2. \]
Proof. It follows directly from the definition of the variance.

\[ \text{Var}[X] = \text{E}[(X - \text{E}[X])^2] = \text{E}[X^2 - 2XE[X] + E[X]^2] = E[X^2] - E[X]^2. \]

\[ \square \]

An important result concerning distribution functions is the following theorem, which will come into play in Section 6.

**Theorem 4.17.** Take \( n \in \mathbb{N} \), let \( X = \{X_i\}_{i=1}^n \) be iid continuous random variables with the pdf \( f_X \) and let \( Y = \{Y_i\}_{i=1}^n \) be the ordered set of \( X \), such that \( Y_1 < Y_2 < \ldots < Y_n \). Then the joint pdf \( g_{Y_1,\ldots,Y_n} \) of \( Y \) is given by

\[ g_{Y_1,\ldots,Y_n}(x_1,x_2,\ldots,x_n) = \frac{n!}{f_X(x_1)f_X(x_2)\ldots f_X(x_n)}. \]

**Proof.** See e.g. [10, Chapter 4.6]. \( \square \)

We move onto the random variables which also will be used later throughout this essay. Note that three of them are continuous random variables and one is discrete.

**Definition 4.18.** Given an interval \([a, b] \subset \mathbb{R}\), a random variable \( X \) with pdf

\[ f_X(x) = \begin{cases} \frac{1}{b-a}, & x \in [a, b] \\ 0, & x \notin [a, b] \end{cases} \]

is said to be uniformly distributed over \([a, b]\), and we denote it as \( X \sim U(a,b) \).

**Definition 4.19.** Given the constants \( \mu \in \mathbb{R} \) and \( \sigma \in \mathbb{R}^+ \), a random variable \( X \) with the pdf

\[ f_X(x) = \frac{1}{\sqrt{2\sigma^2\pi}}e^{-\frac{(x-\mu)^2}{2\sigma^2}}, \quad x \in \mathbb{R} \]

is said to be normally distributed with mean \( \mu \) and variance \( \sigma^2 \), and we denote it as \( X \sim N(\mu,\sigma^2) \).

**Definition 4.20.** Given a constant \( \lambda \in \mathbb{R}^+ \), a random variable \( X \) with pdf

\[ f_X(x) = \begin{cases} \lambda e^{-\lambda x}, & x \in \mathbb{R}^+ \\ 0, & x \notin \mathbb{R}^+ \end{cases} \]

is said to be exponentially distributed with parameter \( \lambda \), and we denote it as \( X \sim \exp(\lambda) \).

**Definition 4.21.** Given a constant \( \lambda \in \mathbb{R}^+ \), a random variable \( X \) with pdf

\[ f_X(x) = \frac{\lambda x e^{-\lambda}}{x!}, \quad x \in \mathbb{N} \]

is said to be Poisson distributed with parameter \( \lambda \), and we denote it as \( X \sim \text{Po}(\lambda) \).

4.3. **Stochastic processes.** The concept of a time dependent process might be close at hand to give examples of, but as we want to observe time dependent processes with random elements we give the following definition.

**Definition 4.22.** Given a time set \( T \) and a space set \( S \), the family of random variables \( X(T,S) = \{X_t \in S, t \in T\} \) is called a stochastic process.

We will work with the special case where \( T = \mathbb{R}^+ \) and \( S = \mathbb{R}^d \). Further, take note that when we mention the process as a whole we do not use the time index. I.e. \( X \) would refer to the process as a whole while \( X_t \) would refer to the process at time \( t \).

**Definition 4.23.** A stochastic process \( X(\mathbb{R}^+,\mathbb{R}^d) \) is called a \( d \)-dimensional stochastic process.

**Notation 4.24.** Depending on context we will use subscript as a discretization index counter or a time counter. E.g. for a \( d \)-dimensional stochastic process \( X \) we can refer to the random variable at a specific time as \( X_t \).
The main focus of this essay will be limited to a narrow set of dimensions, though expanding the linear stochastic oscillator, to observe higher dimensional movement, is a short step. The MATLAB implementations will be constructed, as much as possible, to be independent of choice of dimension, or at least be easily modifiable.

We now proceed with defining a special type of stochastic process which behaves similarly, no matter what time we would choose to inspect it at.

**Definition 4.25.** Take a \(d\)-dimensional stochastic process \(X\). If its joint cdf is invariant under time displacements, that is if
\[
F_{X_{t_1+h}, X_{t_2+h}, \ldots, X_{t_n+h}} = F_{X_{t_1}, X_{t_2}, \ldots, X_{t_n}}
\]
\(\forall h > 0, t_i \geq 0, i \in \{1, 2, \ldots, n\}, n \in \mathbb{N}\), we say that \(X\) is stationary.

The details concerning the following concept will be heavily omitted, giving an intuitive rather than mathematical definition, and we encourage any reader intent on understanding this well to look into e.g. [13, Chapter 2.3].

**Definition 4.26.** Take a \(d\)-dimensional stochastic process \(X\). Then if, for any realization
\[
\mathcal{X}((\omega, R^+)) = \{X_t(\omega), t \in R^+\}
\]
and time \(T \in \mathbb{R}^+\), the information given by \(\mathcal{X}(\omega, [0, T]) = \{X_t(\omega), t \in [0, T]\}\) does not, for any \(s > T\), reveal \(X_s(\omega)\) we say that \(X\) is a non-anticipating process.
5. Stochastic differential equations driven by Wiener processes

This section will provide a brief introduction to the Wiener process, SDE’s driven by Wiener processes, and some theoretical or numerical results concerning them. For a more in depth and complete account of stochastic differential equations we refer the reader to e.g. [11], [13], [15] or [19]. We begin by defining what a Wiener process is.

**Definition 5.1.** The non-anticipating, 1-dimensional stochastic process \( W \) with stationary independent normally distributed increments and continuous sample paths where

1. \( W_0 = 0 \) a.s.,
2. \( E[W_t] = 0, t \in \mathbb{R}^+ \),
3. \( \text{Var}[W_t - W_s] = |t - s|, t, s \in \mathbb{R}^+ \),

is called the **standard Wiener process**.

Limitations imposed upon us from the world prompts us to seek numerical approximations to many problems. These approximations can take many forms, one of which is discretization, which we will do to the time intervals on which we simulate instances of the Wiener processes. Let \( T \in \mathbb{R}^+ \) and \( n \in \mathbb{N} \), if we are to fix a partition \( 0 = t_0 < t_1 < \ldots < t_n = T \) we can simulate each increment \( W_{t_{k+1}} - W_{t_k} \) with normally distributed random variables, since \( \Delta W_{t_k} = W_{t_{k+1}} - W_{t_k} \sim N(0, \Delta t_k) \) per definition.

Quite often it is beneficial if we hold a fixed time stepping size, \( \Delta t_k = \Delta t \), \( k = 0, 1, \ldots, n - 1 \), which gives us that \( \Delta W_{t_k} \sim N(0, \Delta t) \). Due to the independence of the increments we can therefore construct a discretized standard Wiener process by providing Algorithm 1 with the proper constants. For an alternative implementation, see e.g. [8].

**Algorithm 1. Wiener process simulation**

1. Get \( T, n \)
2. Set \( W_0 = 0 \)
3. \( \Delta t = T/n \)
4. for \( k \in \{0, \ldots, n - 1\} \) do
   5. Simulate \( X_k \sim N(0,1) \)
   6. \( W_{k+1} = W_k + \sqrt{\Delta t} X_k \)
5. end for
6. Return \( \{W_i\}_{i=0}^n \)

Here a MATLAB implementation can be seen in [WienerProc.m](#). The two scripts [dtdprocplot.m](#) and [ThreeProcs.m](#) respectively, produces a number of plots illustrating a few properties of the different processes. As for the Wiener process, in [Figure 5.1A](#) we can see three simulated instances done with a relatively small \( \Delta t \) in order to give a good view of the fractal appearance of the trajectories. In [Figure 5.1B](#) we have only one instance with line \( \Delta t \) and an illustration of how an increment could become given a coarser time grid, i.e. a larger \( \Delta t \).
5.1. The Itô integral and the Itô formula. Much like the Riemann integral, we define the Itô integral as the limit of a sum, though we cannot use the ordinary limit as there is much work required to properly deal with the non-differentiability of the Wiener process. For more details, see e.g. [13, Chapter 3.1]. First we give the definition of the limit in question. The reader should note that this concept can be generalized into a convergence in higher dimensions as well.

**Definition 5.2.** Take a sequence of random variables $X = \{X_n\}_{n \geq 1}$ and a random variable $X$. If

$$\lim_{n \to \infty} E[(X_n - X)^2] = 0$$

$X$ is said to converge to $X$ in the mean square sense. We denote this by

$$\text{ms-lim}_{n \to \infty} X_n = X.$$

An informal interpretation would be that the mean square limit gives us a way to guarantee that the variation of the random variable obtained by the limit becomes manageable. With this we proceed to the definition of the stochastic integral. This have some limitations upon the function which we integrate and for more details we refer to e.g. [11], [13], [15] or [19].

**Definition 5.3.** Take a finite interval $[a, b] \subset \mathbb{R}$ with partitioning $a = t_0 < t_1 < \ldots < t_{n-1} < t_n = b$ and set, for a Wiener process $W$, $\Delta W_{t_k} = W_{t_{k+1}} - W_{t_k}$. Then the Itô integral of a given process $f : \mathbb{R}^+ \to \mathbb{R}$ over $[a, b]$ is

$$\int_a^b f(t) dW_t = \text{ms-lim}_{n \to \infty} \sum_{k=0}^{n-1} f(t_k) \Delta W_{t_k}.$$

The stochastic integral has certain properties which can change depending on where the integrand evaluations are taken. For the Itô integral we have the left-most point but an alternative would be to evaluate the integrand at both end points and take an average. The latter case is called a Stratonovich integral and can be seen in e.g. [8].

Given the definition of the Itô integral we can, similar to the ODE integral representation, define Itô processes. The step from one-dimensional to $d$-dimensional is rather short and can be seen in e.g. [13, Chapter 3.4].

**Definition 5.4.** Given that $W$ is a $m$-dimensional Wiener process, $\mu : \mathbb{R}^+ \times \mathbb{R}^d \to \mathbb{R}^d$, $\sigma : \mathbb{R}^+ \times \mathbb{R}^d \to \mathbb{R}^{d \times m}$ and $X_0 \in \mathbb{R}^d$, the non-anticipating $d$-dimensional stochastic process $X$ where

$$X_t = X_0 + \int_0^t \mu(s, X_s) ds + \int_0^t \sigma(s, X_s) dW_s,$$
is called an Itô process. Or equivalently, written in differential form,
\[ dX_t = \mu(t, X_t)dt + \sigma(t, X_t)dW_t. \]

This works well with the discretization we mentioned earlier, as the Itô integral can be approximated by a finite sum of function evaluations and Wiener increments, as seen in [Definition 5.3]. The deterministic integral is handled as usual.

We can now give the Itô formula. The Itô formula is a standard tool to investigate both functions and differential equations. We present the 1-dimensional case here, though we omit some requirements posed on the functions in question.

**Theorem 5.5.** Assume that \( W \) is a 1-dimensional Wiener process, nice enough functions \( f, \mu, \sigma : \mathbb{R}^+ \times \mathbb{R} \to \mathbb{R} \) and that \( X \) is a 1-dimensional Itô process which satisfies
\[ dX_t = \mu(t, X_t)dt + \sigma(t, X_t)dW_t. \]

Then the 1-dimensional Itô formula states that
\[ df = \left( \frac{\partial f}{\partial t} + \mu(t, X_t) \frac{\partial f}{\partial x} + \frac{\sigma(t, X_t)^2}{2} \frac{\partial^2 f}{\partial x^2} \right) dt + \sigma(t, X_t) \frac{\partial f}{\partial x} dW_t. \]

**Proof.** See e.g. [13, Chapter 3.3].

5.2. **Additional theorems concerning Wiener processes.** When we arrive at calculating expected values later we will need to be able to tackle the expected value of stochastic integrals as well as squared stochastic integrals. This motivates the following two theorems, of which the latter is known as the Itô isometry.

**Theorem 5.6.** Given \( T \in \mathbb{R}^+ \), a nice enough function \( f : \mathbb{R}^+ \times \mathbb{R} \to \mathbb{R} \), a non-anticipating stochastic process \( X \) and the Wiener process \( W \), the following equality holds.
\[ E \left[ \int_0^T f(t, X_t)dW_t \right] = 0. \]

**Proof.** See e.g. [13] Theorem 5.8.

**Theorem 5.7.** The Itô isometry states that for \( T \in \mathbb{R}^+ \), a given nice enough function \( f : \mathbb{R}^+ \times \mathbb{R} \to \mathbb{R} \), a non-anticipating stochastic process \( X \) and the Wiener process \( W \), the following equality holds.
\[ E \left[ \left( \int_0^T f(t, X_t)dW_t \right)^2 \right] = E \left[ \int_0^T (f(t, X_t))^2 dt \right]. \]

**Proof.** See e.g. [13] Theorem 5.8.
6. Stochastic differential equations driven by Wiener and Poisson processes

In this section we will cover the concepts of Poisson processes which, for more details, can be seen in e.g. [5] and [18], and stochastic processes or SDE’s driven by said processes. For more details of the latter, the reader is referred to e.g. [4], [16], [17] and [18].

The Poisson process have a few equivalent definitions which have different advantages, and from a visual standpoint it could be argued that it is beneficial to follow the definition, as provided by e.g. [4, Definition 2.17], here seen below as Definition 6.1. Aside from the visual benefit, it also provides a good basis to easily transition into the compound Poisson process, here seen below as Definition 6.5.

Definition 6.1. Given \( \lambda \in \mathbb{R}^+ \) and a set of iid random variables \( \tau_i \sim \exp(\lambda) , i \in \mathbb{N} \), define

\[
T_n = \sum_{i=1}^{n} \tau_i , \quad n \in \mathbb{N}.
\]

Then the 1-dimensional stochastic process \( N \) where

\[
N_t = \sum_{n \geq 1} \mathbb{1}_{(t \geq T_n)}(t)
\]

is called the Poisson process with intensity \( \lambda \).

In the glossary we introduced the following notation, but we will repeat it here to stress its importance, as it will be similar to the other variants of the Poisson process.

Notation 6.2. A Poisson process with intensity \( \lambda \) will be referred to as a PoPr(\( \lambda \)). Or, omitting the intensity, as a PoPr.

Definition 6.1 have some impracticalities regarding simulating large numbers of processes, which is the strength of the following, equivalent, definition:

Definition 6.3. Given \( \lambda \in \mathbb{R}^+ \), the 1-dimensional stochastic process \( N \) where

1. \( N_0 = 0 \) a.s.,
2. \( N_t - N_s \sim \text{Po}(\lambda(t-s)), t \geq s, t, s \in \mathbb{R}^+ \),
3. \( N_{t_1} - N_{s_1} \) independent of \( N_{t_2} - N_{s_2} \) if \( (s_1, t_1] \cap (s_2, t_2] = \emptyset \), \( \forall s_1 < t_1, s_2 < t_2 \in \mathbb{R}^+ \),

is called the Poisson process with intensity \( \lambda \).

To show that these two definitions are equivalent is a task in itself and it clearly falls outside the scope of this essay.

Theorem 6.4. [Definition 6.1] and [Definition 6.3] are equivalent.

Proof. See e.g. [7, Chapter 8, Theorem 1.4]. \( \square \)

The latter definition quite closely follows the thinking presented in the definition of the Wiener process, [Definition 5.1] and therefore have a similar method of approaching the simulation. Again we will take a discretization of the time interval \([0, T]\) with a fixed time step \( \Delta t \), though instead of simulating increments with the normal distribution we will use the Poisson distribution, yielding \( \Delta N_{t_k} \sim \text{Po}(\lambda \Delta t) \). We get Algorithm 2.

---

**Algorithm 2. Poisson process simulation**

1. Get \( \lambda, T, n \)
2. Set \( N_0 = 0 \)
3. \( \Delta t = T/n \)
4. for \( k \in \{0, \ldots, n-1\} \) do
5. Simulate \( X_k \sim \text{Po}(\lambda \Delta t) \)
6. \( N_{k+1} = N_k + X_k \)
7. end for
8. Return \( \{N_k\}_{k=0}^n \)
A MATLAB implementation of the Poisson process (as well as other processes, see below) can be seen in Code 2 PoissonProc.m. It might be worth noting, should the reader choose to inspect the code at this stage of the essay, that the MATLAB implementation does more than simply simulating the Poisson process, also including the compound Poisson process, see Definition 6.5, and the compensated compound Poisson process, see Definition 6.9. This combining of simulation is to support the implementation of exact solutions which may involve many different parts, such as we will see in Theorem 8.5 below.

As in Section 5 we use the two scripts dtdprocplot.m and ThreeProcs.m [Code 20] and [Code 21] producing the following plots using the same presentation as Figure 5.1. We see three instances of the Poisson process in Figure 6.1A and in Figure 6.1B we see an illustration of a possible increment in case of a larger \( \Delta t \).

![Figure 6.1A](image1.png) ![Figure 6.1B](image2.png)

(A) Three samples of the PoPr over \( t \in [0, 1] \).  
(B) A sample of the PoPr with a coarse increment illustrated.

**Figure 6.1.** Poisson process samples, \( N \), for \( t \in [0, 1] \) and with intensity \( \lambda = 12 \), one with a coarse increment illustrated. Processes simulated by Code 2, which is a MATLAB implementation of Algorithm 2.

Leaving the PoPr behind, using it as a basis, we construct a generalization which allows for jumps of random size, instead of a fixed size of 1.

**Definition 6.5.** Given \( \lambda \in \mathbb{R}^+ \) and a set of iid random variables \( \tau_i \sim \text{exp}(\lambda), \ i \in \mathbb{N} \), define

\[
T_n = \sum_{i=1}^{n} \tau_i, \quad n \in \mathbb{N}.
\]

Further, take a set of iid random variables \( \{X_n\}_{n=1}^{\infty} \) from a given distribution \( \mathcal{N} \), i.e. \( X_n \sim \mathcal{N}, \ n \in \mathbb{N} \). Then the 1-dimensional stochastic process \( \hat{N} \), where

\[
\hat{N}_t = \sum_{n \geq 1} X_n \mathbb{1}_{\{t \geq T_n\}}(t),
\]

is called the compound Poisson process with intensity \( \lambda \) and distribution \( \mathcal{N} \).

As before we will repeat the shorthand notation.

**Notation 6.6.** A compound Poisson process with intensity \( \lambda \) and distribution \( \mathcal{N} \) will be referred to as a CPoPr(\( \lambda, \mathcal{N} \)). Or, omitting the intensity and distribution, as a CPoPr. Note that this introduces the implicit use of \( X_n \) as the jumps.

We can see that the PoPr is a special case of the CPoPr where the jumps are \( X_n = 1, n = 1, 2, \ldots \). The simulation algorithm follows the same train of thought as Algorithm 2, but with one additional step. Before, the Poisson distributed random variable was the increment itself as well, but here it is taken as the number of random variables which needs to be simulated from a distribution \( \mathcal{N} \), which then are summed. This can be seen in Algorithm 3.
Algorithm 3. Compound Poisson process simulation

1: Get \( \lambda, T, n \)
2: Get the distribution \( \mathcal{N} \)
3: Set \( \hat{N}_0 = 0 \)
4: \( \Delta t = T/n \)
5: for \( k \in \{0, \ldots, n - 1\} \) do
6: Simulate \( m_k \sim Po(\lambda \Delta t) \)
7: Simulate \( X_1, X_2, \ldots, X_{m_k} \sim \mathcal{N} \)
8: \( \hat{N}_{k+1} = \hat{N}_k + \sum_{i=1}^{m_k} X_i \)
9: end for
10: Return \( \{\hat{N}_k\}_{k=0}^n \)

As stated before, for Algorithm 2, a MATLAB implementation can be seen in Code 2, PoissonProc.m. And again, as seen with Figure 5.1 and Figure 6.1, the two scripts dtdprocplot.m and ThreeProcs.m, Code 20 and Code 21 are used, following the same presentation. We can see three instances of the compound Poisson process in Figure 6.2A and further, in Figure 6.2B we can see an illustration of a possible increment in case of a larger \( \Delta t \).

Figure 6.2. Compound Poisson process samples, \( \hat{N} \), for \( t \in [0, 1] \), with intensity \( \lambda = 12 \) and distribution \( \mathcal{N} = N(0.5, 1) \). One with a coarse increment illustrated. Processes simulated by Code 2 which is a MATLAB implementation of Algorithm 3.

An experienced reader will perhaps have noticed that the compound Poisson process should have an expected drift which will depend on the jump distribution and the intensity. As seen in the previous section, through observing that since, for a Wiener process \( W \),

\[
W_t = W_t - W_0 \sim N(0, t),
\]

the Wiener process does not have an expected drift, i.e. that the expected value \( \mathbb{E}[W_t] = 0 \) for all \( t \), and it might be of interest to produce something similar based on the Poisson processes. The following theorem illustrates how the drift evolves over time for a CPoPr.

**Theorem 6.7** ([15] p. 459). For a CPoPr(\( \lambda, \mathcal{N} \)) \( \hat{N} \) with \( X_n \sim \mathcal{N}, n \in \mathbb{N} \), the following equality holds:

\[
\mathbb{E}[\hat{N}_t] = \lambda t \mathbb{E}[X_1].
\]

**Proof.** Let \( N \) be the underlying PoPr controlled by the same time sequence as \( \hat{N} \). By conditioning the expected value of \( N_t \) at time \( t \) on the number of jumps up until time \( t \), which is equivalent to \( N_t \), through
Definition 6.3 we get
\[
E[\hat{N}_t] = E \left[ \sum_{n \geq 1} X_n \mathbb{1}_{\{t \geq T_n\}}(t) \right] = \sum_{n=0}^{\infty} \frac{(\lambda t)^n e^{-\lambda t}}{n!} E \left[ \sum_{i=1}^{n} X_i \right] = E \left[ \sum_{i=1}^{n} X_i \right] = \lambda t E[X_1]. \quad \square
\]

Here it would be natural to include the variance as well.

Theorem 6.8 ([18] p. 460). For a CPoPr(\(\lambda, \mathcal{N}\)) \(\hat{N}\) with \(X_n \sim \mathcal{N}, n \in \mathbb{N}\), the following equality holds:
\[
\text{Var}[\hat{N}_t] = \lambda t E[X_1^2].
\]

Proof. If we let \(N\) be the underlying PoPr for \(\hat{N}\), we get through the law of total variance that
\[
\text{Var}[\hat{N}_t] = E[\text{Var}[\hat{N}_t|N_t]] + \text{Var}[E[\hat{N}_t|N_t]] = E \left[ \text{Var} \left[ \sum_{i=1}^{N_t} X_i \right] \right] + \text{Var} \left[ E \left[ \sum_{i=1}^{N_t} X_i \right] \right] = E[N_t \text{Var}[X_1]] + \text{Var}[N_t E[X_1]] = E[N_t \text{Var}[X_1] + \text{Var}[N_t E[X_1]]^2].
\]

Now, since \(N_t \sim \text{Po}(\lambda t)\) we have that
\[
\text{Var}[N_t] = \lambda t (\text{Var}[X_1] + E[X_1]^2) = \lambda t E[X_1^2]. \quad \square
\]

Knowing the expected value of a CPoPr, we can construct a process which takes the expected drift of the jumps into account and compensates for it with a deterministic counter drift.

Definition 6.9. Take a CPoPr(\(\lambda, \mathcal{N}\)) \(\hat{N}\). Then the 1-dimensional stochastic process \(\tilde{N}\) where
\[\tilde{N}_t = \hat{N}_t - \lambda t E[X_1]\]
is called the compensated compound Poisson process with intensity \(\lambda\) and distribution \(\mathcal{N}\).

As before we will repeat the notation.

Notation 6.10. A compensated compound Poisson process with intensity \(\lambda\) and distribution \(\mathcal{N}\) will be referred to as a CCPoPr(\(\lambda, \mathcal{N}\)). Or, omitting the intensity and distribution, as a CCPoPr. Note that this introduces the implicit use of \(X_n\) as the jumps.

The algorithm for this is very much like the case for the CPoPr, but the countering deterministic drift plays a role.

\begin{algorithm}
\caption{Compensated compound Poisson process simulation}
1: Get \(\lambda, T, n\)
2: Get the distribution \(\mathcal{N}\)
3: Get the expected value of \(X_1 \sim \mathcal{N}, E[X_1]\)
4: Set \(\tilde{N}_0 = 0\)
5: \(\Delta t = T/n\)
6: for \(k \in \{0, \ldots, n - 1\}\) do
7: \hspace{1em} Simulate \(m_k \sim \text{Po}(\lambda \Delta t)\)
8: \hspace{1em} Simulate \(X_1, X_2, \ldots, X_{m_k} \sim \mathcal{N}\)
9: \hspace{1em} \(\tilde{N}_{k+1} = \tilde{N}_k + \sum_{i=1}^{m_k} X_i - \lambda E[X_1] \Delta t\)
10: end for
11: Return \(\{\tilde{N}_k\}_{k=0}^n\)
\end{algorithm}
A MATLAB implementation, as stated at Algorithm 2 and Algorithm 3, can be seen in Code 2, PoissonProc.m. And yet again, as seen with Figure 5.1, Figure 6.1 and Figure 6.2, the two scripts dtdprocplot.m and ThreeProcs.m, Code 20 and Code 21 are used, following the same presentation. We can see three instances of the compensated compound Poisson process in Figure 6.3A and further, in Figure 6.3B we can see an illustration of a possible increment in case of a larger $\Delta t$.

![A sample of a CCPoPr process with a coarse increment illustrated.](image)

**Figure 6.3.** Compensated compound Poisson process samples, $\tilde{N}$, for $t \in [0,1]$, with intensity $\lambda = 12$ and distribution $\mathcal{N} = N(0,5,1)$. One with a coarse increment illustrated.

Processes simulated by Code 2, which is a MATLAB implementation of Algorithm 4.

With these three different types of the PoPr we have a few theoretical results which follows. First we can note that a CPoPr with a distribution which have mean zero is a CCPoPr.

**Lemma 6.11.** A CPoPr with $E[X_1] = 0$ is a CCPoPr.

**Proof.** The CCPoPr, $\tilde{N}$, based upon the given CPoPr, $\hat{N}$, at time $t$ is

$$\tilde{N}_t = \hat{N}_t - \lambda t E[X_1] = \hat{N}_t.$$

Continuing with the smaller results, something which ties in closely with the previous lemma and with the goal of constructing the CPoPr to begin with.

**Lemma 6.12.** All CCPoPr have expected value zero at any given time $t \geq 0$.

**Proof.** Take a CCPoPr $\tilde{N}$, with underlying CPoPr $\hat{N}$. Then Definition 6.9 and Theorem 6.7 gives that

$$E[\tilde{N}_t] = E[\hat{N}_t - \lambda t E[X_1]] = E[\hat{N}_t] - \lambda t E[X_1] = 0.$$

As we will run into the expected value of the square of increments of CCPoPr it will be useful to state the following theorem.

**Theorem 6.13.** For a CCPoPr $\tilde{N}$ the following equivalence holds

$$E[(\tilde{N}_t)^2] = \lambda t E[X_1^2].$$

**Proof.** Assuming that $\tilde{N}$ is the underlying CPoPr we have, with the help of Theorem 6.7 and Theorem 6.8 that

$$E[(\tilde{N}_t)^2] = E[(\hat{N}_t - \lambda t E[X_1])^2] = \text{Var}[\tilde{N}_t] = \lambda t E[X_1^2].$$

Now that we have properly defined the different stochastic processes this essay will utilize, we present the multiplication table Table 1, which is given in Table 15.1. We will not go into the motivation for Table 1 as it would be outside the scope of this essay.
The Itō integral and the Itō formula for Poisson processes. Defining the stochastic integrals for the different Poisson processes properly requires the use of random measures. For brevity, we refer the reader to e.g. [4] Chapter 2.6 or [17] Chapter 1.4. In addition to not detailing the background behind the definitions of these stochastic integrals, we will also omit the indication of the left limits, i.e. the notation $X^-$. Any reader considering learning more of these particular stochastic integrals are encouraged to read more.

Readers who are more comfortable with calculus might benefit from remembering the Dirac delta function, sometimes defined as the derivative of the Heaviside step function. The former is related to the above mentioned proper approach but it is not a function strictly speaking.

Definition 6.14. Given a CPoPr$(\lambda,\mathcal{N})$ $\hat{N}$ with the corresponding jump times $\{T_i\}_{i=1}^n$ within the finite interval $[a,b] \subset \mathbb{R}$ and a non-anticipating stochastic process $Y$, the compound Poisson Itō integral of $f : \mathbb{R}^+ \times \mathbb{R} \to \mathbb{R}$ over $[a,b]$ is defined as

$$
\int_a^b f(s,Y_s) d\hat{N}_s = \sum_{i=1}^n X_i f(T_i,Y_{T_i}).
$$

The generalization to cover $d$-dimensional processes follows as easily as for the Wiener process seen in Section 5. We can now move onto the corresponding definition for the CCPoPr.

Definition 6.15. Take a CCPoPr$(\lambda,\mathcal{N})$ $\hat{N}$ with underlying CPoPr$(\lambda,\mathcal{N})$ $\hat{N}$ and corresponding jump times $\{T_i\}_{i=1}^n$ within the finite interval $[a,b] \subset \mathbb{R}$. Then, given a non-anticipating stochastic process $Y$, the compensated compound Poisson Itō integral of $f : \mathbb{R}^+ \times \mathbb{R} \to \mathbb{R}$ over $[a,b]$ is defined as

$$
\int_a^b f(s,Y_s) d\hat{N}_s = \int_a^b f(s,Y_s)(d\hat{N}_s - \lambda E[X_1]ds) = \sum_{i=1}^n X_i f(T_i,Y_{T_i}) - \lambda E[X_1] \int_0^t f(s,Y_s)ds.
$$

Following the previous Itō process, driven by only a Wiener process, Definition 5.4 we expand upon it with the addition of a process from the Poisson family.

Definition 6.16. Assume that $W$ is an $m$-dimensional Wiener process, that $N^*$ is either an $n$-dimensional CPoPr or CCPoPr and that $\mu : \mathbb{R}^+ \times \mathbb{R}^d \to \mathbb{R}^d$, $\sigma : \mathbb{R}^+ \times \mathbb{R}^d \to \mathbb{R}^{d \times n}$, $\eta : \mathbb{R}^+ \times \mathbb{R}^d \to \mathbb{R}^{d \times n}$ and $X_0 \in \mathbb{R}^d$. Then the non-anticipating $d$-dimensional stochastic process $X$ where

$$
X_t = X_0 + \int_0^t \mu(s,X_s)ds + \int_0^t \sigma(s,X_s)dW_s + \int_0^t \eta(s,X_s)dN^*_s,
$$

is called an Itō process driven by Wiener and Poisson processes. Or equivalently, written in differential form,

$$
dX_t = \mu(t,X_t)dt + \sigma(t,X_t)dW_t + \eta(t,X_t)dN^*_t.
$$

Leaving aside the properties of the Poisson processes we are now ready to state the Itō formula for Itō processes driven by Wiener and Poisson processes. This can be seen for more details in [4]. Some requirements on the functions $\mu$, $\sigma$ and $f$ are omitted here, for simplicity of exposition.

Theorem 6.17. Assume that $W$ is a Wiener process, $\hat{N}$ is a CPoPr and that $\mu, \sigma : \mathbb{R}^+ \times \mathbb{R} \to \mathbb{R}$. Then, for a nice enough function $f : \mathbb{R}^+ \times \mathbb{R} \to \mathbb{R}$ and for the Itō process $Z$ which satisfies

$$
dZ_t = \mu(t,Z_t)dt + \sigma(t,Z_t)dW_t + d\hat{N}_t,
$$

\begin{table}[h]
\centering
\begin{tabular}{|c|c|c|c|}
\hline
 & $dt$ & $dW_t$ & $d\hat{N}_t$ \\
\hline
$dt$ & 0 & 0 & 0 \\
$dW_t$ & 0 & $dt$ & 0 \\
$d\hat{N}_t$ & 0 & 0 & $d\hat{N}_t$ \\
\hline
\end{tabular}
\caption{Multiplication table of differential terms for independent Wiener and CPoPr processes. The processes are named $W$ and $\hat{N}$ respectively.}
\end{table}
the stochastic process $Y_t = f(t, Z_t)$, can be represented in differential notation as

$$dY_t = \left( \frac{\partial f}{\partial t} + \mu(t, Z_t) \frac{\partial f}{\partial z} + \frac{\sigma(t, Z_t)^2}{2} \frac{\partial^2 f}{\partial z^2} \right) dt + \frac{\partial f}{\partial z} \sigma(t, Z_t) dW_t + \left[ f \left( t, \lim_{s \uparrow t} Z_s \right) - f \left( t, \lim_{s \downarrow t} Z_s \right) \right].$$


This theorem will not be used, but it serves a purpose to introduce a comparison between the case where an Itô process is driven by both a Poisson and a Wiener process or only the latter (i.e. Theorem 5.5).

6.2. Additional theorems concerning Poisson processes. The concept of a fundamental matrix is an important property which we want to use later on, but first we will have to define what it is. Following [15, Chapter 3.2], leaving a few requirements posed on the functions, we get the following definition.

Definition 6.18. Assume that $W$ is an 1-dimensional Wiener process, that $N^*$ is either an 1-dimensional CPoPr or CCPoPr and that $F, G, H : \mathbb{R}^+ \to \mathbb{R}^{d \times d}$. Let $\mathbf{e}_j$ be the unit vector at index $j$, $j = 1, \ldots, d$. Take the SDE

$$dX_t = F(t)X_t dt + G(t)X_t dW_t + H(t)X_t dN^*_t$$

and set $\Phi_j(t)$ to be the solution to (6.1) with initial value $X_{t_0} = \mathbf{e}_j$, $j = 1, \ldots, d$. Then

$$\Phi_t = [\Phi_1(t) \quad \Phi_2(t) \quad \ldots \quad \Phi_d(t)]$$

is called the fundamental matrix of (6.1).

A version of the theorem called the variation-of-constants formula is given in [15, Chapter 3.3]. This theorem is given, assuming $d, m \in \mathbb{N}$, for a $d$-dimensional stochastic process driven by $m$ Wiener processes.

Theorem 6.19. Assume that

$$F, G, H : \mathbb{R}^+ \to \mathbb{R}^{d \times d},$$

$$f, g, h : \mathbb{R}^+ \to \mathbb{R}^d,$$

and that $H(t)h(t) = \mathbf{0}$. Further, take a Wiener process $W$, either a CPoPr or a CCPoPr $N^*$ and consider the $d$-dimensional linear SDE

$$dX_t = (F(t)X_t + f(t)) dt + (G(t)X_t + g(t)) dW_t + (H(t)X_t + h(t)) dN^*_t.$$ (6.2)

Then for the initial value $X_{t_0}$ and the fundamental matrix $\Phi_t$ of the corresponding homogeneous equation,

$$dX_t = F(t)X_t dt + G(t)X_t dW_t + H(t)X_t dN^*_t,$$

the stochastic process

$$X_t = \Phi_t \left( X_{t_0} + \int_{t_0}^t \Phi_s^{-1} [f(s) - G(s)g(s)] ds + \int_{t_0}^t \Phi_s^{-1} g(s) dW_s + \int_{t_0}^t \Phi_s^{-1} h(s) dN^*_s \right)$$

satisfies (6.2).

Proof. First, we note that the fundamental matrix $\Phi_t$ to the homogeneous equation of (6.2) is the solution to

$$d\Phi_t = F(t)\Phi_t dt + G(t)\Phi_t dW_t + H(t)\Phi_t dN^*_t.$$

Set

$$\xi_t = X_{t_0} + \int_{t_0}^t \Phi_s^{-1} [f(s) - G(s)g(s)] ds + \int_{t_0}^t \Phi_s^{-1} g(s) dW_s + \int_{t_0}^t \Phi_s^{-1} h(s) dN^*_s.$$ Then $\xi_t$ has the differential

$$d\xi_t = \Phi_t^{-1} [f(t) - G(t)g(t)] dt + \Phi_t^{-1} g(t) dW_t + \Phi_t^{-1} h(t) dN^*_t.$$ By defining

$$Y_t = \Phi_t \xi_t$$

we see that $Y_{t_0} = X_{t_0}$ and that

$$dY_t = d\Phi_t \xi_t + \Phi_t d\xi_t + d\Phi_t d\xi_t.$$
Expanding the products yield

\[
dY_t = (F(t)\Phi_t dt + G(t)\Phi_t dW_t + H(t)\Phi_t dN^*_t)\xi_t
\]

\[
+ \Phi_t(\Phi_t^{-1} [f(t) - G(t)g(t)] dt + \Phi_t^{-1}g(t)dW_t + \Phi_t^{-1}h(t)dN^*_t)
\]

\[
+ (F(t)\Phi_t dt + G(t)\Phi_t dW_t + H(t)\Phi_t dN^*_t)(\Phi_t^{-1} [f(t) - G(t)g(t)] dt + \Phi_t^{-1}g(t)dW_t + \Phi_t^{-1}h(t)dN^*_t)
\]

\[
= F(t)Y_t dt + G(t)Y_t dW_t + H(t)Y_t dN^*_t + (f(t) - G(t)g(t))dt + g(t)dt + h(t)dt
\]

\[
+ G(t)g(t)(dW_t)^2 + H(t)h(t)(dN^*_t)^2.
\]

By assumption, we have that \(H(t)h(t) = 0\). Using said assumption and Table 1 gives us that

\[
dY_t = F(t)Y_t dt + G(t)Y_t dW_t + H(t)Y_t dN^*_t + (f(t) + g(t))dt + h(t)dt
\]

\[
= (f(t)Y_t + f(t))dt + (G(t)Y_t + g(t))dW_t + (H(t)Y_t + h(t))dN^*_t.
\]

This means that \(X_t = Y_t = \Phi_t\xi_t\) is a solution to (6.2).

We will now present the following theorem which can be seen in e.g. [5] as Exercise 2.1.5. This theorem will allow us to see the jump times as either the results of cumulative sums of exponentially distributed random variables or as uniformly distributed random variables over a given time interval.

**Theorem 6.20.** Assume that a PoPr \(N\) has \(n\) jumps in the interval \([0, t]\), with the jump times \(T = \{T_i\}_{i=1}^n\). Further, using an ordering \(k_i\) for \(i = 1, \ldots, n\), assume that \(U = \{u_{k_i}\}_{i=1}^n\) is an ordered set of iid random variables where \(u_i \sim U(0, t), i = 1, \ldots, n\). Then the joint pdf of \(T\) is equivalent to the joint pdf of \(U\).

**Proof.** Take a set of points \(0 = t_0 < t_1 < t_2 < \cdots < t_n < t_{n+1} = t\) and a constant \(\Delta t\) such that \(t_i + \Delta t < t_{i+1}, i = 1, \ldots, n\). Then what we are interested in is

\[
P(t_1 < T_1 \leq t_1 + \Delta t, \ldots, t_n < T_n \leq t_n + \Delta t|N_n = n)
\]

We have now split up the time interval into \(2n + 1\) pieces, \(n\) intervals with one jump and \(n + 1\) intervals with zero jumps. Therefore, using the increment properties of Definition 6.3 and defining the set \(\{Y_i\}_{i=1}^{2n+1}\), where \(Y_i(\lambda) \sim PPF(\lambda)\), gives via conditional probability that

\[
P(t_1 < T_1 \leq t_1 + \Delta t, \ldots, t_n < T_n \leq t_n + \Delta t|N_n = n)
\]

\[
= \frac{\left(\lambda \Delta t\right)^n e^{-\lambda \Delta t} e^{-\lambda \left(t_1 - t_0\right)} e^{-\lambda \left(t_2 - (t_1 + \Delta t)\right)} \cdots e^{-\lambda \left(t_{n+1} - (t_n + \Delta t)\right)}}{n!} = \frac{\left(\lambda \Delta t\right)^n e^{-\lambda \left(t_n - \Delta t\right)}}{n!} = \frac{n!(\Delta t)^n}{t^n}.
\]

Dividing by \((\Delta t)^n\) and taking the limit \(\Delta t \to 0\) yields us

\[
f_{T_1, \ldots, T_n}(t_1, \ldots, t_n) = \lim_{\Delta t \to 0} P(t_1 < T_1 \leq t_1 + \Delta t, \ldots, t_n < T_n \leq t_n + \Delta t|N_n = n) = \frac{n!}{t^n}.
\]

Take the set \(\{u_i\}_{i=1}^n\) where \(u_i \sim U(0, t)\) and define the ordering \(k_i, i = 1, \ldots, n\) such that \(u_{k_1} \leq u_{k_2} \leq \ldots \leq u_{k_n}\). Define \(U_i = u_{k_i}\) for \(i = 1, \ldots, n\), and name the ordered set \(U = \{U_i\}_{i=1}^n = \{u_{k_i}\}_{i=1}^n\). Then for \(U\) and the joint pdf \(g_{U_1, \ldots, U_n}\), we have from Theorem 4.17 that

\[
g_{U_1, \ldots, U_n}(t_1, \ldots, t_n) = \frac{n!}{t^n} = f_{T_1, \ldots, T_n}(t_1, \ldots, t_n).
\]

Now, continuing as in Section 3, we present the expected value of the stochastic integral. After that we will look at the Itô isometry which will present itself in two different ways, depending on whether the underlying variant of the Poisson process is a CPoPr or a CCPoPr.
Theorem 6.21 ([13], p. 462). Assume that we have a function \( f : \mathbb{R}^+ \times \mathbb{R} \to \mathbb{R} \), a CPoPr(\( \lambda, N \)) \( \hat{N} \), and a non-anticipating stochastic process \( Y \). Then, for \( T \in \mathbb{R}^+ \), the following equality holds.

\[
E \left[ \int_0^T f(t, Y_t) d\hat{N}_t \right] = \lambda E[X_1] E \left[ \int_0^T f(t, Y_t) dt \right].
\]

Proof. Let \( N \) be the underlying Poisson process for \( \hat{N} \) and let \( \{ T_i \}_{i=1}^\infty \) be the jump times. By conditioning on the number of jumps within \([0, T]\) we get, excluding the \( n = 0 \) case in the last step due to the inner sum evaluating to zero,

\[
E \left[ \int_0^T f(t, Y_t) d\hat{N}_t \right] = \sum_{n=0}^\infty E \left[ \sum_{i=1}^{N_T} f(T_i, Y_{T_i}) X_i \right] P(N_T = n)
\]

\[
= \sum_{n=1}^\infty \sum_{i=1}^n E \left[ f(T_i, Y_{T_i}) X_i \right] P(N_T = n) \frac{(\lambda T)^n e^{-(\lambda T)}}{n!}. \tag{6.3}
\]

Now we can represent this differently using Theorem 6.20. We can replace \( T_i \) with \( u_i \sim U(0, T) \) and this, combined using the independence and identical distribution of the various random variables, gives when we continue from the end of (6.3) that

\[
E \left[ \int_0^T f(t, Y_t) d\hat{N}_t \right] = \sum_{n=1}^\infty \sum_{i=1}^n E[X_1] E \left[ f(u_i, Y_{u_i}) \right] \frac{(\lambda T)^n e^{-(\lambda T)}}{n!}
\]

\[
= E[X_1] \sum_{n=1}^\infty n E[f(u_1, Y_{u_1})] \frac{(\lambda T)^n e^{-(\lambda T)}}{n!}. \tag{6.4}
\]

Rewriting the expected value yields

\[
E[f(u_1, Y_{u_1})] = \int_{-\infty}^\infty E[f(u_1, Y_{u_1})|Y = y] dy = \int_{-\infty}^\infty \int_0^T \frac{f(t, Y_t)}{T} dt \, dy = \frac{1}{T} E \left[ \int_0^T f(t, Y_t) dt \right]
\]

and inserting this into (6.4) gives that

\[
E \left[ \int_0^T f(t, Y_t) d\hat{N}_t \right] = \lambda E[X_1] E \left[ \int_0^T f(t, Y_t) dt \right] \sum_{n=1}^{\infty} \frac{(\lambda T)^{n-1} e^{-(\lambda T)}}{(n-1)!} = \lambda E[X_1] E \left[ \int_0^T f(t, Y_t) dt \right]. \tag{6.4}
\]

Translating this result into the compensated case is done quickly and can be seen below.

Lemma 6.22. Given a function \( f : \mathbb{R}^+ \times \mathbb{R} \to \mathbb{R} \), a non-anticipating stochastic process \( Y \), a CCPoPr \( \hat{N} \) and \( T \in \mathbb{R}^+ \), the following equality holds,

\[
E \left[ \int_0^T f(t, Y_t) \right] = 0.
\]

Proof. Assume that \( \hat{N} \) is the underlying CPoPr. Then it follows immediately from Theorem 6.21 that

\[
E \left[ \int_0^T f(t, Y_t) d\hat{N}_t \right] = E \left[ \int_0^T f(t, Y_t) (d\hat{N}_t - \lambda E[X_1] dt) \right]
\]

\[
= E \left[ \int_0^T f(t, Y_t) d\hat{N}_t - \lambda E[X_1] \int_0^T f(t, Y_t) dt \right] = 0. \tag{6.4}
\]

Now that we know the expected value of the stochastic integrals, we can move into the two Itô isometries. We will look at the compensated case first.

Theorem 6.23. Given a function \( f : \mathbb{R}^+ \times \mathbb{R} \to \mathbb{R} \), a non-anticipating stochastic process \( Y \), a CCPoPr \( \hat{N} \) and \( T \in \mathbb{R}^+ \), the following equality holds,

\[
E \left[ \left( \int_0^T f(t, Y_t) d\hat{N}_t \right)^2 \right] = \lambda E[X_1^2] E \left[ \int_0^T f(t, Y_t)^2 dt \right].
\]
Lemma 6.24. Given a function \( f : \mathbb{R}^+ \times \mathbb{R} \to \mathbb{R} \), a non-anticipating stochastic process \( Y \), a CPoPr \( \hat{N} \) and \( T \in \mathbb{R}^+ \), the following equality holds,

\[
E \left[ \left( \int_0^T f(t, Y_t) d\hat{N}_t \right)^2 \right] = \lambda E[X_1^2] E \left[ \int_0^T f(t, Y_t)^2 dt \right] + \lambda^2 E[X_1]^2 E \left[ \left( \int_0^T f(t, Y_t) dt \right)^2 \right].
\]

Proof. We have that

\[
\left( \int_0^T f(t, Y_t) d\hat{N}_t - \lambda E[X_1] dt \right)^2 = \left( \int_0^T f(t, Y_t) d\hat{N}_t - \int_0^T f(t, Y_t) \lambda E[X_1] dt \right)^2
\]

\[
= \left( \int_0^T f(t, Y_t) d\hat{N}_t \right)^2 - 2 \int_0^T f(t, Y_t) d\hat{N}_t \int_0^T f(t, Y_t) \lambda E[X_1] dt + \left( \int_0^T f(t, Y_t) \lambda E[X_1] dt \right)^2.
\]

Shuffling, defining the CCPoPr \( \tilde{\hat{N}}_t \) which is based upon the CPoPr \( \hat{N}_t \) and taking the expected value gives

\[
E \left[ \left( \int_0^T f(t, Y_t) d\hat{N}_t \right)^2 \right] = E \left[ \left( \int_0^T f(t, Y_t) d\tilde{\hat{N}}_t \right)^2 \right]
\]

\[
+ 2 \int_0^T f(t, Y_t) d\tilde{\hat{N}}_t \int_0^T f(t, Y_t) \lambda E[X_1] dt - \left( \int_0^T f(t, Y_t) \lambda E[X_1] dt \right)^2. \tag{6.5}
\]

One specific part requires a rewrite before we can use it. Defining a new function \( g \) and using Theorem 6.21 yields us

\[
E \left[ \int_0^T f(t, Y_t) d\tilde{\hat{N}}_t \int_0^T f(t, Y_t) dt \right] = E \left[ \int_0^T \int_0^T f(s, Y_s) ds \ f(t, Y_t) d\tilde{\hat{N}}_t \right] = E \left[ \int_0^T g(t, Y_t) d\tilde{\hat{N}}_t \right]
\]

\[
= \lambda E[X_1] E \left[ \int_0^T g(t, Y_t) dt \right] = \lambda E[X_1] E \left[ \int_0^T \int_0^T f(s, Y_s) ds \ f(t, Y_t) dt \right] = \lambda E[X_1] E \left[ \left( \int_0^T f(t, Y_t) dt \right)^2 \right].
\]

Linearity allows us to insert this into (6.5), which combined with Theorem 6.23 gives us

\[
E \left[ \left( \int_0^T f(t, Y_t) d\hat{N}_t \right)^2 \right] = E \left[ \left( \int_0^T f(t, Y_t) d\tilde{\hat{N}}_t \right)^2 \right]
\]

\[
+ 2 \lambda^2 E[X_1]^2 \left( \int_0^T f(t, Y_t) dt \right)^2 - \left( \int_0^T f(t, Y_t) \lambda E[X_1] dt \right)^2
\]

\[
= \lambda E[X_1^2] E \left[ \int_0^T f(t, Y_t)^2 dt \right] + \lambda^2 E[X_1]^2 E \left[ \left( \int_0^T f(t, Y_t) dt \right)^2 \right]. \tag*{\Box}
\]
General numerical schemes

Numerical approximations have become increasingly viable as the capabilities of computing and numerical theory have evolved. This becomes useful if a given problem have difficult properties which makes it harder, if not impossible, to find an analytical solution. In this section we will present a number of numerical schemes which we later will adapt to allow explicit computations for the geometric stochastic motion and the stochastic linear oscillators, see Section 8 and Section 9.

Presumably the reader is well acquainted with the explicit and implicit Euler schemes for ordinary differential equations, which are among the earliest schemes introduced within numerical analysis. They will serve as good comparison to the precision of the other schemes.

The numerical schemes will be presented for a $d$-dimensional Itô process $X_t$, with initial value $X_0$, driven by 1-dimensional Wiener and Poisson processes. I.e. we have the following SDE:

$$\text{d}X_t = \mu(t, X_t)\text{d}t + \sigma(t, X_t)\text{d}W_t + \eta(t, X_t)\text{d}N^*_t(\lambda, N),$$

(7.1)

where $W_t$ is a standard Wiener process, see Defintion 5.1, and $N^*_t(\lambda, N) = N^*$ is either a CPoPr, see Definition 6.5, or a CCPoPr, see Definition 6.9.

Assuming a fixed time step size, $\Delta t \in \mathbb{R}^+$, we set $t_n = n\Delta t$ and the schemes are defined as follows. The increments for the stochastic processes can be seen in Algorithm 1, Algorithm 2, Algorithm 3 and Algorithm 4. A MATLAB implementation of the proposed numerical schemes can be found in Appendix G.

**Forward, or explicit, Euler-Maruyama scheme.** This scheme is presented in e.g. [13, Page 305]. A slight generalization, including $dN^*$, gives

$$x_{n+1} = x_n + \mu(t_n, x_n)\Delta t + \sigma(t_n, x_n)\Delta W_n + \eta(t_n, x_n)\Delta N^*_n(\lambda, N).$$

**Backward, or implicit, Euler-Maruyama scheme.** This scheme is presented in e.g. [13, Page 396]. A slight generalization, including $dN^*$, gives

$$x_{n+1} = x_n + \mu(t_{n+1}, x_{n+1})\Delta t + \sigma(t_n, x_n)\Delta W_n + \eta(t_n, x_n)\Delta N^*_n(\lambda, N).$$

**θ-Euler scheme.** This scheme is presented in e.g. [13, Page 396] or [6, Page 54] and it assumes $\theta \in [0, 1]$. A slight generalization, including $N^*$, gives

$$x_{n+1} = x_n + (\theta\mu(t_n, x_n) + (1-\theta)\mu(t_{n+1}, x_{n+1}))\Delta t + \sigma(t_n, x_n)\Delta W_n + \eta(t_n, x_n)\Delta N^*_n(\lambda, N).$$

**SSB-Euler scheme.** This scheme is presented in e.g. [9] and is only defined for the SDE’s driven by Wiener and CPoPr processes $W$ and $N$ respectively, i.e. SDE’s of the form

$$dX_t = \mu(t, X_t)\text{d}t + \sigma(t, X_t)\text{d}W_t + \eta(t, X_t)\text{d}N_t(\lambda, N),$$

which we see is equivalent to (7.1) excluding CPoPr’s. We get

$$x^*_n = x_n + \mu(t_n, x^*_n)\Delta t,$$

$$x^*_{n+1} = x^*_n + \sigma(t_n, x^*_n)\Delta W_n + \eta(t_n, x^*_n)\Delta N^*_n(\lambda, N).$$

**CSSB-Euler scheme.** This scheme is presented in e.g. [9] and is only defined for the SDE’s driven by Wiener and CPoPr processes $W$ and $N$ respectively, i.e. SDE’s of the form

$$dX_t = \mu(t, X_t)\text{d}t + \sigma(t, X_t)\text{d}W_t + \eta(t, X_t)\text{d}N_t(\lambda, N),$$

which we see is equivalent to (7.1) excluding CPoPr’s. Taking into account that the expected value of the jumps is not necessarily 1, as [9] only looks at pure PoPr’s, yields

$$x^*_n = x_n + (\mu(t_n, x^*_n) + \lambda E[X_1]\eta(t_n, x^*_n))\Delta t,$$

$$x^*_{n+1} = x^*_n + \sigma(t_n, x^*_n)\Delta W_n + \eta(t_n, x^*_n)(\Delta N^*_n(\lambda, N) - \lambda E[X_1]\Delta t).$$
Milstein scheme. This scheme is presented in e.g. [17], though it is only presented for 1-dimensional SDE’s, driven by Wiener processes $W$. I.e. SDE’s of the form
\[
dX_t = \mu(t, X_t) dt + \sigma(t, X_t) dW_t,
\]
which we see is equivalent to (7.1) when $d = 1$ and $\eta = 0$. It reads as follows:
\[
x_{n+1} = x_n + \mu(t_n, x_n) \Delta t + \sigma(t_n, x_n) \Delta W_n + \frac{1}{2} \sigma(t_n, x_n) \left( \frac{d}{dx} \sigma(t_n, x_n) \right) ((\Delta W_n)^2 - \Delta t).
\]

7.1. Concepts of convergence. [8] presents two standard tools to analyze the accuracy of numerical methods, the strong and weak error.

**Definition 7.1.** A scheme is said to have strong order of convergence equal to $\gamma$ if there exists a constant $C$ such that
\[
E[|x_n - X_{t_n}|] \leq C \Delta t^\gamma,
\]
where $x_n$ is the numerical approximation of the exact solution, $X_{t_n}$, at time $t_n = n \Delta t$.

Here there will be some requirements again which will fall outside the scope of this essay. More specifically, the requirements on the function $f$ in the following definition. As a quick note, [8] states that typically $f$ has to satisfy smoothness and polynomial growth.

**Definition 7.2.** If there exists a constant $C$ such that for all functions $f$ in some class, a scheme is said to have weak order of convergence equal to $\beta$ if
\[
|E[f(x_n)] - E[f(X_{t_n})]| \leq C \Delta t^\beta,
\]
where $x_n$ is the numerical approximation of the exact solution, $X_{t_n}$, at time $t_n = n \Delta t$.

When we use the weak error of the energy later on, in Section 9, there is a need to take larger sample sizes to approximate the expectation than when calculating the strong order of convergence. The MATLAB implementation is done such that it should be possible to apply other norms with little effort.
8. The geometric Brownian motion

8.1. Geometric Brownian motion - Background and definitions. The geometric motions are among the first example students of differential equations come across, mainly due to it having easier derived solutions than many other models. The problem in a deterministic setting, i.e. the equation, for some \( c \in \mathbb{R} \),

\[
\frac{dX_t}{dt} + cX_t = 0
\]

has a well known solution, assuming an initial value \( X_0 \in \mathbb{R}^+ \), and can often be seen described in more detail in standard calculus literature. We will modify this problem by adding two differentiated stochastic processes to the right hand side, a standard Wiener process \( W_t \), see Definition 5.1, and either a CPoPr or a CCPoPr \( N^* \), see Definition 6.5 or Definition 6.9 respectively. I.e. we will analyze, rewritten to coincide with our earlier notation, (7.1), and with the constants \( \mu, \sigma, \eta \in \mathbb{R} \), the SDE

\[
dx_t = \mu X_t dt + \sigma X_t dW_t + \eta X_t dN^*_t.
\]

Readers who are interested in theory related to this specific stochastic process is referred to e.g. [16] or [18]. We give the following definitions.

Definition 8.1. Given that \( W_t \) is a standard Wiener process, and the constants \( \mu, \sigma \in \mathbb{R} \), we call the stochastic process \( S_t \), which fulfills the SDE

\[
dS_t = \mu S_t dt + \sigma S_t dW_t
\]

with initial value \( S_0 \in \mathbb{R}^+ \), the geometric Brownian motion or, in short, the GBM.

When we go into the extended definition we will exclude the case where it is driven by a CCPoPr as the countering deterministic drift will simply be included in the already existing deterministic part. As seen by the following lemma.

Lemma 8.2. Assume that \( W_t \) is a standard Wiener process, that \( \tilde{N} \) is a CCPoPr, with the underlying CPoPr \( \hat{N} \), and the constants \( \mu, \sigma, \eta \in \mathbb{R} \). Then for the SDE

\[
dS_t = \mu S_t dt + \sigma S_t dW_t + \eta d\tilde{N}_t
\]

there exists a constant \( c \in \mathbb{R} \) such that

\[
dS_t = c S_t dt + \sigma S_t dW_t + \eta d\hat{N}_t.
\]

Proof. Using the definition of the CCPoPr, Definition 6.9, we get

\[
dS_t = \mu S_t dt + \sigma S_t dW_t + \eta S_t d\tilde{N}_t = \mu S_t dt + \sigma S_t dW_t + \eta S_t (d\tilde{N}_t - \lambda E[X_1] dt)
\]

\[
= (\mu - \lambda E[X_1]) S_t dt + \sigma S_t dW_t + \eta S_t d\hat{N}_t.
\]

Defining \( c = \mu - \lambda E[X_1] \) concludes the proof. \( \square \)

This means that we can give the extended definition without having to include any CCPoPr.

Definition 8.3. Given that \( W_t \) is a standard Wiener process, that \( \tilde{N} \) is a CPoPr, and the constants \( \mu, \sigma, \eta \in \mathbb{R} \), we call the stochastic process \( S_t \) which fulfills the SDE

\[
dS_t = \mu S_t dt + \sigma S_t dW_t + \eta S_t d\tilde{N}_t
\]

with initial value \( S_0 \in \mathbb{R}^+ \), the geometric Brownian motion driven by a CPoPr or, in short, the GBPM.

8.2. GBPM - Model specific numerical schemes. Now we apply the general schemes given in Section 7 to the GBPM, or the GBM where the reduced model is required. The MATLAB implementations of these numerical schemes applied to these models can be found in Appendix H.

Explicit Euler scheme. We have

\[
x_{n+1} = x_n + \mu x_n \Delta t + \sigma x_n \Delta W_n + \eta x_n \Delta \tilde{N}_n.
\]
**Implicit Euler scheme.** We have
\[ x_{n+1} = x_n + \mu x_{n+1} \Delta t + \sigma x_n \Delta W_n + \eta x_n \Delta \hat{N}_n \]
which simplifies into
\[ x_{n+1} = \frac{x_n + \sigma x_n \Delta W_n + \eta x_n \Delta \hat{N}_n}{1 - \mu \Delta t}. \]

**θ-Euler scheme.** We have
\[ x_{n+1} = x_n + (\theta \mu x_n + (1 - \theta) \mu x_{n+1}) \Delta t + \sigma x_n \Delta W_n + \eta x_n \Delta \hat{N}_n \]
which simplifies into
\[ x_{n+1} = \frac{x_n + \theta \mu x_n \Delta t + \sigma x_n \Delta W_n + \eta x_n \Delta \hat{N}_n}{1 - (1 - \theta) \mu \Delta t}. \]

**SSB-Euler scheme.** We have
\[ x_n^* = x_n + \mu x_n \Delta t \]
\[ x_{n+1} = x_n^* + \sigma x_n^* \Delta W_n + \eta x_n^* \Delta \hat{N}_n \]
and a simplification gives
\[ x_{n+1} = x_n^* + \sigma x_n^* \Delta W_n + \eta x_n^* \Delta \hat{N}_n. \]

**CSSB-Euler scheme.** We have
\[ x_n^* = x_n + (\mu x_n^* + \lambda \mathbb{E}[X_1] \eta x_n^*) \Delta t \]
\[ x_{n+1} = x_n^* + \sigma x_n^* \Delta W_n + \eta x_n^* (\Delta \hat{N}_n - \lambda \mathbb{E}[X_1] \Delta t) \]
and, again, a simplification gives
\[ x_{n+1} = x_n^* + \sigma x_n^* \Delta W_n + \eta x_n^* (\Delta \hat{N}_n - \lambda \mathbb{E}[X_1] \Delta t). \]

**Milstein scheme.** We have
\[ x_{n+1} = x_n + \mu x_n \Delta t + \sigma x_n \Delta W_n + \frac{\sigma^2 x_n^2}{2} ((\Delta W_n)^2 - \Delta t). \]

8.3. **GBPM - Exact solutions.** So starting with the solution to the special case where we do not have an influence from any of the Poisson processes, we have a result which most who has read about stochastic differential equations should be acquainted with. The following theorem can be seen in e.g. [15, Page 105], [17, Page 47], or [18, Page 99].

**Theorem 8.4.** The GBM, Definition 8.1, has the exact solution
\[ S_t = S_0 e^{(\mu - \frac{\sigma^2}{2})t + \sigma W_t}. \]

**Proof.** Define
\[ X_t = \left( \mu - \frac{\sigma^2}{2} \right) t + \sigma W_t. \]
Then for the stochastic process \( S \) where
\[ S_t = S_0 e^{X_t}, \]
**Theorem 5.5** gives that
\[ dS_t = \left( 0 + \left( \mu - \frac{\sigma^2}{2} \right) S_0 e^{X_t} + \frac{\sigma^2}{2} S_0 e^{X_t} \right) dt + \sigma S_0 e^{X_t} dW_t = \mu S_t dt + \sigma S_t dW_t. \]
Here it could be of interest to observe how the strong error behaves for some of the schemes we have at our disposal. In the script `Code 22 (GBMStrongTime.m)` we picked four schemes; the explicit, implicit, theta Euler (with \( \theta = 0.5 \)) schemes and Milstein scheme, with the parameters \( \mu = 0.8, \sigma = 0.2 \) and \( S_0 = 1 \). The resulting plots are Figure 8.1 and Figure 8.2 which display how the strong error evolves over the time interval \([0, 1]\), and the strong error at the end point.

We see, in Figure 8.2, how the theta Euler scheme has a better precision than both the explicit and implicit schemes for larger \( \Delta t \), though it seems to have the same strong order of convergence, \( \gamma = 1/2 \). The Milstein scheme displays strong order \( \gamma = 1 \) and quickly yields a much better approximation.

![Figure 8.1](image)

**Figure 8.1.** Given the explicit Euler-Maruyama, implicit Euler-Maruyama, Milstein and theta Euler schemes, the estimated strong error of the GBM, over \( t \in [0, 1] \). 95% CI shown in black.

If we look at the solution to the GBPM we get the following theorem.

**Theorem 8.5.** Assume that \( \hat{N} \) has the jumps \( \{X_n\}_{n=1}^{\infty} \) and underlying PoPr \( N \). Then the GBPM, Definition 8.3, has the solution

\[
S_t = S_0e^{(\mu-\frac{\sigma^2}{2})t+\sigma W_t}\prod_{n=1}^{N_t}(1+\eta X_n).
\]

**Proof.** See e.g. [19] for a proof based upon Lévy processes or [18] for a lighter version. \( \Box \)

As with the GBM, we will now look at the strong error. Though we will limit this to only observing the strong error at time \( T = 1 \). In Figure 8.3 to Figure 8.7 we see the strong errors obtained from the five available schemes. We have chosen the parameters \( \mu = 0.8, \lambda = 3, S_0 = 1 \) and \( \sigma, \eta \in \{0, 0.2\} \). The implementation can be seen in `Code 23 (GBMStrongTime.m),` GBMP4StrongTime.m).

Inspecting Figure 8.3 to Figure 8.7 we see that the schemes may have different orders of convergence depending on the values of \( \sigma \) and \( \eta \). For \( \sigma \neq 0 \) they all display a strong order of convergence \( \gamma = 1/2 \). For the deterministic setting, \( \sigma = \eta = 0 \), all but one give \( \gamma = 1 \). The exception being the theta Euler scheme, \( \theta = 0.5 \), which displays \( \gamma = 2 \), see Figure 8.5.

Something which is demanding further investigation is the case where \( \sigma = 0 \) and \( \eta = 0.2 \), as it seems as if \( \gamma = 1 \) for all schemes. This is the same order of convergence as the deterministic case, despite the introduction of random elements through the CPoPr.
Figure 8.2. Given the explicit Euler-Maruyama, implicit Euler-Maruyama, Milstein and theta Euler schemes, a log-log plot of the estimated strong error of the GBM, at \( T = 1 \). 95% CI shown as bar plots. \( C_i \in \mathbb{R}, i = 1, 2 \) are constants used to translate the support lines.

Figure 8.3. The strong error of the GBPM at \( T = 1 \) obtained using the explicit Euler scheme. \( C_i \in \mathbb{R}, i = 1, 2, 3 \) are constants used to translate the support lines.
Figure 8.4. The strong error of the GBPM at $T = 1$ obtained using the implicit Euler scheme. $C_i \in \mathbb{R}, i = 1, 2, 3$ are constants used to translate the support lines.

Parameters: $\mu=0.8, \lambda=3$. Batch size: 256. Number of batches: 32.

Figure 8.5. The strong error of the GBPM at $T = 1$ obtained using the theta Euler scheme, $\theta = 0.5$. $C_i \in \mathbb{R}, i = 1, 2, 3$ are constants used to translate the support lines.
Figure 8.6. The strong error of the GBPM at $T = 1$ obtained using the SSB Euler scheme. $C_i \in \mathbb{R}$, $i = 1, 2, 3$ are constants used to translate the support lines.

Parameters: $\mu = 0.8$, $\lambda = 3$. Batch size: 256. Number of batches: 32.

Figure 8.7. The strong error of the GBPM at $T = 1$ obtained using the CSSB Euler scheme. $C_i \in \mathbb{R}$, $i = 1, 2, 3$ are constants used to translate the support lines.
8.4. **Additional comments.** In order to investigate why the GBPM driven by a CPoPr seems to display the same order of convergence as the GBPM not driven by any stochastic process we have included a script, Code 32 (GBPMOrderTest.m). In this script we have set \( \mu = 0.2, \eta = 0.8, \sigma \in \{0, 0.05, 0.2, 0.8\}, T = 1/8, \lambda = 48 \) and \( \mathcal{N} = N(0.5, 1) \). This gives us Figure 8.8 which unfortunately does not give any good indication of whether \( \gamma = 1 \) truly holds.

The expected number of jumps \( n \) within the interval \([0, T]\) would be \( n = T\lambda = 48/8 = 6 \). In order to increase the influence of the randomness introduced by the CPoPr we would need to increase the number of jumps or the size of the jumps, either through the distribution or through a larger \( \eta \). However, increasing the size of the jumps require larger sample sizes in order to overcome the concentrated variance at each point. Likewise, increasing the intensity \( \lambda \) in order to increase \( n \) will require a larger sample size.

Should the readers be interested in whether the GBPM truly is of strong order \( \gamma = 1 \), when it is driven by only a CPoPr, we encourage them to test this themselves.

![Figure 8.8](image-url)  

**Figure 8.8.** The strong error of the GBPM at \( T = 1/8 \) obtained using the implicit Euler scheme. \( C_i \in \mathbb{R}, i = 1, 2, 3 \) are constants used to translate the support lines.
9. The linear stochastic oscillator

9.1. LSO - Background and definitions. The linear oscillator in the deterministic setting, i.e. the equation for some \( \omega \in \mathbb{R}^+ \),
\[
\ddot{X}_t + \omega^2 X_t = 0,
\]
with initial values \( X_0, \dot{X}_0 \in \mathbb{R}^+ \), is another standard problem for students also known as the harmonic motion. It provides a smooth introduction to the concept of substitution and systems of ordinary differential equations. We will assume that the reader is at least familiar with systems of ordinary differential equations. In order to make it a system of differential equations of the first order we introduce a substitution,
\[
\begin{bmatrix} x_1(t) \\ x_2(t) \end{bmatrix} = \begin{bmatrix} X(t) \\ \dot{X}(t) \end{bmatrix}.
\]
(9.1)
This yields
\[
dx_t = \begin{bmatrix} 0 & 1 \\ -\omega^2 & 0 \end{bmatrix} \begin{bmatrix} x_1(t) \\ x_2(t) \end{bmatrix} dt.
\]
(9.2)
Like the GBM and the GBPM we will modify the linear oscillator with Wiener and Poisson processes, or variants thereof. I.e. we get the SDE
\[
dx_t = \begin{bmatrix} 0 & 1 \\ -\omega^2 & 0 \end{bmatrix} \begin{bmatrix} x_1(t) \\ x_2(t) \end{bmatrix} dt + \begin{bmatrix} 0 \\ \alpha \end{bmatrix} dW_t + \begin{bmatrix} 0 \\ \eta \end{bmatrix} dN^*_t.
\]
(9.3)
With this we define, much like for the GBM and the GBPM, the following two stochastic processes.

Definition 9.1. Given that \( W \) is a standard Wiener process and the constants \( \omega \in \mathbb{R}^+, \alpha \in \mathbb{R} \) we call the SDE
\[
\ddot{X}_t + \omega^2 X_t = \alpha \dot{W}_t,
\]
or equivalently, following (9.3),
\[
dx_t = \begin{bmatrix} 0 & 1 \\ -\omega^2 & 0 \end{bmatrix} \begin{bmatrix} x_1(t) \\ x_2(t) \end{bmatrix} dt + \begin{bmatrix} 0 \\ \alpha \end{bmatrix} dW_t,
\]
the linear stochastic oscillator driven by a Wiener process or, in short, the LBO.

Definition 9.2. Given that \( W \) is a standard Wiener process, that \( N^* \) is either a CPoPr or a CCPoPr, and the constants \( \omega \in \mathbb{R}^+, \alpha, \eta \in \mathbb{R} \) we call the SDE
\[
\ddot{X}_t + \omega^2 X_t = \alpha \dot{W}_t + \eta \dot{N}^*_t,
\]
or equivalently, following (9.3),
\[
dx_t = \begin{bmatrix} 0 & 1 \\ -\omega^2 & 0 \end{bmatrix} \begin{bmatrix} x_1(t) \\ x_2(t) \end{bmatrix} dt + \begin{bmatrix} 0 \\ \alpha \end{bmatrix} dW_t + \begin{bmatrix} 0 \\ \eta \end{bmatrix} dN^*_t,
\]
(9.5)
the linear stochastic oscillator driven by a Wiener process and a Poisson process or, in short, the LBPO.

A property which both [2] and [22] investigate is the energy for the LBO, which is defined as follows.

Definition 9.3. For a stochastic process \( X \) with the substitution given in (9.1), the function \( g : \mathbb{R}^+ \times \mathbb{R} \times \mathbb{R} \rightarrow \mathbb{R}^+ \) such that
\[
g(\omega, x^1, x^2) = \frac{1}{2} \omega^2 (x^1)^2 + (x^2)^2,
\]
is called the energy.

Note that both references have investigated the special case where \( \eta = 0 \) and the latter have further restricted with \( \omega = 1 \).

9.2. LSO - Model specific numerical schemes. As in Section 8 we will apply the general schemes given in Section 7, though this time to the LBPO. The MATLAB implementations of these numerical schemes applied to a slightly more general setting can be found in Appendix I.

Within this section we have not used the Milstein scheme. However, instead we have two additional schemes, presented within [2] and [22], the trigonometric scheme and the partitioned Euler scheme respectively.

35
Explicit Euler scheme. We get
\[
\begin{bmatrix}
x_n^{1+1} \\ x_n^{2+1}
\end{bmatrix} = \begin{bmatrix}
x_n^1 \\ x_n^2
\end{bmatrix} + 0 \begin{bmatrix}
-\omega^2 & 1 \\ -\omega^2 & 1
\end{bmatrix} \Delta t + \begin{bmatrix}
0 \\ 0
\end{bmatrix} \Delta W_n + \begin{bmatrix}
0 \\ 0
\end{bmatrix} \Delta N_n^*.
\] (9.7)

Implicit Euler scheme. We get
\[
\begin{bmatrix}
x_n^{1+1} \\ x_n^{2+1}
\end{bmatrix} = \begin{bmatrix}
x_n^1 \\ x_n^2
\end{bmatrix} + 0 \begin{bmatrix}
-\omega^2 & 1 \\ -\omega^2 & 1
\end{bmatrix} \Delta t + \begin{bmatrix}
0 \\ 0
\end{bmatrix} \Delta W_n + \begin{bmatrix}
0 \\ 0
\end{bmatrix} \Delta N_n^*.
\]
Which simplifies into
\[
\begin{bmatrix}
x_n^{1+1} \\ x_n^{2+1}
\end{bmatrix} = \left( I - \begin{bmatrix}
0 & 0 \\ -\omega^2 & 0
\end{bmatrix} \Delta t \right)^{-1} \left( \begin{bmatrix}
x_n^1 \\ x_n^2
\end{bmatrix} + 0 \begin{bmatrix}
-\omega^2 & 1 \\ -\omega^2 & 1
\end{bmatrix} \Delta t \right) + \begin{bmatrix}
0 \\ 0
\end{bmatrix} \Delta W_n + \begin{bmatrix}
0 \\ 0
\end{bmatrix} \Delta N_n^*.
\] (9.8)

Partitioned Euler scheme. Rewritten from how it is presented in [22], we get
\[
\begin{bmatrix}
x_n^{1+1} \\ x_n^{2+1}
\end{bmatrix} = \begin{bmatrix}
x_n^1 \\ x_n^2
\end{bmatrix} + 0 \begin{bmatrix}
-\omega^2 & 1 \\ -\omega^2 & 1
\end{bmatrix} \Delta t + \begin{bmatrix}
0 \\ 0
\end{bmatrix} \Delta W_n + \begin{bmatrix}
0 \\ 0
\end{bmatrix} \Delta N_n^*.
\]
Which simplifies into
\[
\begin{bmatrix}
x_n^{1+1} \\ x_n^{2+1}
\end{bmatrix} = \left( I - \begin{bmatrix}
0 & 0 \\ -\omega^2 & 0
\end{bmatrix} \Delta t \right)^{-1} \left( \begin{bmatrix}
x_n^1 \\ x_n^2
\end{bmatrix} + \begin{bmatrix}
-\omega^2 & 1 \\ -\omega^2 & 1
\end{bmatrix} \Delta t \right) + \begin{bmatrix}
0 \\ 0
\end{bmatrix} \Delta W_n + \begin{bmatrix}
0 \\ 0
\end{bmatrix} \Delta N_n^*.
\] (9.9)

$\theta$-Euler scheme. For $\theta \in [0, 1]$ we get
\[
\begin{bmatrix}
x_n^{1+1} \\ x_n^{2+1}
\end{bmatrix} = \begin{bmatrix}
x_n^1 \\ x_n^2
\end{bmatrix} + \theta \begin{bmatrix}
0 & 1 \\ -\omega^2 & 0
\end{bmatrix} \Delta t + (1-\theta) \begin{bmatrix}
0 & 1 \\ -\omega^2 & 0
\end{bmatrix} \Delta t + \begin{bmatrix}
0 \\ 0
\end{bmatrix} \Delta W_n + \begin{bmatrix}
0 \\ 0
\end{bmatrix} \Delta N_n^*.
\]
Which simplifies into
\[
\begin{bmatrix}
x_n^{1+1} \\ x_n^{2+1}
\end{bmatrix} = \left( I - (1-\theta) \begin{bmatrix}
0 & 1 \\ -\omega^2 & 0
\end{bmatrix} \Delta t \right)^{-1} \left( \begin{bmatrix}
x_n^1 \\ x_n^2
\end{bmatrix} + \begin{bmatrix}
0 & 1 \\ -\omega^2 & 0
\end{bmatrix} \Delta t \right) + \begin{bmatrix}
0 \\ 0
\end{bmatrix} \Delta W_n + \begin{bmatrix}
0 \\ 0
\end{bmatrix} \Delta N_n^*.
\]

SSB-Euler scheme. We get
\[
\begin{bmatrix}
y_n^1 \\ y_n^2
\end{bmatrix} = \begin{bmatrix}
x_n^1 \\ x_n^2
\end{bmatrix} + \begin{bmatrix}
0 & 1 \\ -\omega^2 & 0
\end{bmatrix} \Delta t + \begin{bmatrix}
y_n^1 \\ y_n^2
\end{bmatrix} \Delta W_n + \begin{bmatrix}
0 \\ 0
\end{bmatrix} \Delta \tilde{N}_n.
\]
Which simplifies into
\[
\begin{bmatrix}
y_n^1 \\ y_n^2
\end{bmatrix} = \left( I - \begin{bmatrix}
0 & 1 \\ -\omega^2 & 0
\end{bmatrix} \Delta t \right)^{-1} \begin{bmatrix}
x_n^1 \\ x_n^2
\end{bmatrix} + \begin{bmatrix}
y_n^1 \\ y_n^2
\end{bmatrix} \Delta W_n + \begin{bmatrix}
0 \\ 0
\end{bmatrix} \Delta \tilde{N}_n.
\]
CSSB-Euler scheme. We get
\[
\begin{bmatrix}
  y_1^n \\
  y_2^n \\
\end{bmatrix}
= \begin{bmatrix}
  x_1^n \\
  x_2^n \\
\end{bmatrix}
+ \begin{bmatrix}
  0 & -\omega^2 \\
  -\omega^2 & 0 \\
\end{bmatrix}
\begin{bmatrix}
  y_1^n \\
  y_2^n \\
\end{bmatrix}
+ \lambda E[X_1] \begin{bmatrix}
  0 \\
  \eta \\
\end{bmatrix} \Delta t
\]
\[
\begin{bmatrix}
  x_1^{n+1} \\
  x_2^{n+1} \\
\end{bmatrix}
= \begin{bmatrix}
  y_1^n \\
  y_2^n \\
\end{bmatrix}
+ \begin{bmatrix}
  0 & \alpha \\
  \alpha & 0 \\
\end{bmatrix} \Delta W_n + \begin{bmatrix}
  0 \\
  \eta \\
\end{bmatrix} (\Delta \tilde{N}_n - \lambda E[X_1] \Delta t).
\]
Which simplifies into
\[
\begin{bmatrix}
  y_1^n \\
  y_2^n \\
\end{bmatrix}
= \begin{bmatrix}
  x_1^n \\
  x_2^n \\
\end{bmatrix}
+ \begin{bmatrix}
  0 & 1 \\
  -1 & 0 \\
\end{bmatrix}
\begin{bmatrix}
  y_1^n \\
  y_2^n \\
\end{bmatrix}
+ \lambda E[X_1] \begin{bmatrix}
  0 \\
  \eta \\
\end{bmatrix} \Delta t
\]
\[
\begin{bmatrix}
  x_1^{n+1} \\
  x_2^{n+1} \\
\end{bmatrix}
= \begin{bmatrix}
  y_1^n \\
  y_2^n \\
\end{bmatrix}
+ \begin{bmatrix}
  0 & \alpha \\
  \alpha & 0 \\
\end{bmatrix} \Delta W_n + \begin{bmatrix}
  0 \\
  \eta \\
\end{bmatrix} (\Delta \tilde{N}_n - \lambda E[X_1] \Delta t).
\]

Trigonometric scheme. In [2] we are given a scheme, for the case \( \eta = 0 \), with excellent long time properties, based upon the variations-of-constants formula, Theorem 6.19. Generalizing this (to fit (9.3)) yields us the numerical scheme
\[
\begin{bmatrix}
  x_1^{n+1} \\
  x_2^{n+1} \\
\end{bmatrix}
= \begin{bmatrix}
  \cos(\omega \Delta t) & -\omega \sin(\omega \Delta t) \\
  \omega \sin(\omega \Delta t) & \cos(\omega \Delta t) \\
\end{bmatrix}
\begin{bmatrix}
  x_1^n \\
  x_2^n \\
\end{bmatrix}
+ \alpha \begin{bmatrix}
  \omega^{-1} \sin(\omega \Delta t) \\
  \cos(\omega \Delta t) \\
\end{bmatrix} \Delta W_n + \eta \begin{bmatrix}
  0 \\
  \omega \sin(\omega \Delta t) \\
\end{bmatrix} \Delta \tilde{N}_n^*.
\]

9.3. Linear stochastic oscillator - Preparatory lemmas. We start with a few results which will reduce the length and repetitiveness of the theorems regarding the solutions to the LBO and the LBPO.

**Lemma 9.4.** The homogeneous, deterministic, case of the LSO, (9.2), has the fundamental matrix
\[
\Phi_t = \begin{bmatrix}
  \cos(\omega t) & \sin(\omega t) \\
  -\omega \sin(\omega t) & \omega \cos(\omega t) \\
\end{bmatrix}.
\]
Further, the fundamental matrix has the inverse
\[
\Phi_t^{-1} = \begin{bmatrix}
  \cos(\omega t) & -\sin(\omega t) \\
  \omega \sin(\omega t) & \omega \cos(\omega t) \\
\end{bmatrix}.
\]

**Proof.** The matrix seen in (9.2) has eigenvalues and corresponding eigenvectors
\[
\lambda_1 = i\omega, \quad \vec{e}_1 = \begin{bmatrix}
  1 \\
  i\omega \\
\end{bmatrix},
\]
\[
\lambda_2 = -i\omega, \quad \vec{e}_2 = \begin{bmatrix}
  1 \\
  -i\omega \\
\end{bmatrix}.
\]
Which in turn gives us the fundamental matrix
\[
\Phi_t = \begin{bmatrix}
  \cos(\omega t) & \sin(\omega t) \\
  -\omega \sin(\omega t) & \omega \cos(\omega t) \\
\end{bmatrix} \begin{bmatrix}
  c_1 \\
  0 \\
\end{bmatrix}.
\]
Since we, at \( t = 0 \), want the fundamental matrix to be equivalent to the identity matrix, \( I \), we have
\[
\Phi_0 = \begin{bmatrix}
  \cos(0) & \sin(0) \\
  -\omega \sin(0) & \omega \cos(0) \\
\end{bmatrix} \begin{bmatrix}
  c_1 \\
  0 \\
\end{bmatrix} = \begin{bmatrix}
  1 & 0 \\
  0 & \omega \\
\end{bmatrix} \begin{bmatrix}
  c_1 \\
  0 \\
\end{bmatrix} = I.
\]
This means that
\[
c_1 = 1
\]
\[
c_2 = \frac{1}{\omega}
\]
and that (9.11) is the fundamental matrix to (9.2). Trivially, we see that the determinant of the fundamental matrix is 1, which means that there exists an inverse. Due to uniqueness of the inverse it suffices to show that, using the matrices presented in (9.11) and (9.12),

\[
\Phi_t \Phi_t^{-1} = \begin{bmatrix}
\cos(\omega t) & \sin(\omega t) \\
-\omega \sin(\omega t) & \cos(\omega t)
\end{bmatrix}
\begin{bmatrix}
\cos(\omega t) & -\sin(\omega t) \\
\omega \sin(\omega t) & \cos(\omega t)
\end{bmatrix} =
\begin{bmatrix}
\cos^2(\omega t) + \sin^2(\omega t) & 0 \\
0 & \cos^2(\omega t) + \sin^2(\omega t)
\end{bmatrix} = I.
\]

Continuing with a lemma which concerns these matrices.

**Lemma 9.5.** Assume that \(\Phi_t\) is the fundamental matrix (9.11) and \(\Phi_t^{-1}\) is the corresponding inverse matrix (9.12). Then, given a constant \(c \in \mathbb{R}\) and the vector \(\vec{c} = \begin{bmatrix} 0 \\ c \end{bmatrix}\), the following equality holds.

\[
\Phi_t \Phi_t^{-1} \vec{c} = c \begin{bmatrix}
\sin(\omega(t - s)) \\
\cos(\omega(t - s))
\end{bmatrix}.
\]

**Proof.** We compute the product immediately.

\[
\Phi_t \Phi_t^{-1} \vec{c} = \begin{bmatrix}
\cos(\omega t) & \sin(\omega t) \\
-\omega \sin(\omega t) & \cos(\omega t)
\end{bmatrix}
\begin{bmatrix}
\cos(\omega s) & -\sin(\omega s) \\
\omega \sin(\omega s) & \cos(\omega s)
\end{bmatrix} \begin{bmatrix} 0 \\ c \end{bmatrix} =
\begin{bmatrix}
\cos(\omega t) \cos(\omega s) + \sin(\omega t) \sin(\omega s) & -\cos(\omega t) \sin(\omega s) + \sin(\omega t) \cos(\omega s) \\
-\omega \sin(\omega t) \cos(\omega s) + \omega \cos(\omega t) \sin(\omega s) & \sin(\omega t) \sin(\omega s) + \cos(\omega t) \cos(\omega s)
\end{bmatrix} \begin{bmatrix} 0 \\ c \end{bmatrix} =
\begin{bmatrix}
\frac{-\cos(\omega t) \sin(\omega s)}{\omega} + \frac{\sin(\omega t) \cos(\omega s)}{\omega} \\
\frac{\omega}{\omega} \sin(\omega t) \sin(\omega s) + \cos(\omega t) \cos(\omega s)
\end{bmatrix}.
\]

The trigonometric identities

\[
\sin(x - y) = \sin(x) \cos(y) - \cos(x) \sin(y)
\]

\[
\cos(x - y) = \cos(x) \cos(y) + \sin(x) \sin(y)
\]

then gives (9.13). □

This perhaps gives a hint in how we are to use [Lemma 9.4](#Lemma9.4) and [Lemma 9.5](#Lemma9.5). We can now present a few theorems and numerical results for the linear stochastic oscillator where \(\eta = 0\), i.e. where it is only driven by a Wiener process.

### 9.4. The linear stochastic oscillator driven by a Wiener process

The exact solution of the LBO is given in e.g. [2] and can be seen in the following theorem.

**Theorem 9.6 ([2], Theorem 3).** The LBO (9.4), with initial values \([x_0^1, x_0^2]_0\), has the exact solution

\[
x_t = \begin{bmatrix}
x_0^1 \cos(\omega t) + \frac{x_0^2 \sin(\omega t)}{\omega} \\
-\omega x_0^1 \sin(\omega t) + x_0^2 \cos(\omega t)
\end{bmatrix} + \alpha \int_0^t \begin{bmatrix}
\frac{\sin(\omega(t - s))}{\omega} \\
\cos(\omega(t - s))
\end{bmatrix} dW_s,
\]
or, equivalently,

\[
x_1^t = x_0^1 \cos(\omega t) + x_0^2 \omega^{-1} \sin(\omega t) + \alpha \int_0^t \omega^{-1} \sin(\omega(t-s))dW_s,
\]

\[
x_2^t = -\omega x_0^1 \sin(\omega t) + x_0^2 \cos(\omega t) + \alpha \int_0^t \cos(\omega(t-s))dW_s.
\]

(9.14)

**Proof.** Striving to apply Theorem 6.19 to (9.4) we see that

\[
F(t) = \begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix}, \quad f(t) = \begin{bmatrix} 0 \\ 0 \end{bmatrix},
\]

\[
G(t) = \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix}, \quad g(t) = \begin{bmatrix} 0 \\ \alpha \end{bmatrix},
\]

\[
H(t) = \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix}, \quad h(t) = \begin{bmatrix} 0 \\ 0 \end{bmatrix}.
\]

Obviously the requirement that \(H(t)h(t) = \vec{0}\) is fulfilled. The homogeneous equation for the problem (9.4) is

\[
\begin{bmatrix} dx_1^t \\ dx_2^t \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ -\omega^2 & 0 \end{bmatrix} \begin{bmatrix} x_1^t \\ x_2^t \end{bmatrix} dt,
\]

which Lemma 9.4 states has the fundamental matrix, and corresponding inverse matrix,

\[
\Phi_t = \begin{bmatrix} \cos(\omega t) & \frac{\sin(\omega t)}{\omega} \\ -\omega \sin(\omega t) & \cos(\omega t) \end{bmatrix}
\]

and

\[
\Phi_t^{-1} = \begin{bmatrix} \cos(\omega t) & -\frac{\sin(\omega t)}{\omega} \\ \omega \sin(\omega t) & \cos(\omega t) \end{bmatrix}.
\]

Now, reducing the solution to (9.4) given by Theorem 6.19 since \(h(t) = 0\), yields

\[
x_t = \Phi_t \left( x_0 + \int_0^t \Phi_s^{-1} [f(s) - G(s)g(s)] ds + \int_0^t \Phi_s^{-1} g(s) dW_s \right).
\]

(9.15)

We can evaluate the first term as

\[
\Phi_t x_0 = \begin{bmatrix} x_0^1 \cos(\omega t) + \frac{x_0^2 \sin(\omega t)}{\omega} \\ -\omega x_0^1 \sin(\omega t) + \frac{x_0^2 \cos(\omega t)}{\omega} \end{bmatrix}.
\]

(9.16)

A brief inspection reveals that the second term of (9.15) is 0. Through Lemma 9.5, the third term evaluates to

\[
\Phi_t \int_0^t \Phi_s^{-1} g(s) dW_s = \alpha \int_0^t \begin{bmatrix} \frac{\sin(\omega(t-s))}{\omega} \\ \frac{\cos(\omega(t-s))}{\omega} \end{bmatrix} dW_s.
\]

(9.17)

Inserting (9.16) and (9.17) into (9.15) concludes the proof. □

As in Section 8, we will numerically study the strong error. In order to do that we will need to circumvent the stochastic integral by using a reference solution. This approximation will need to be done with a smaller step size than what the schemes are using, in order for it to be of a high enough precision. We have calculated this reference solution using the trigonometric scheme with step size \(\Delta t = 2^{-14}\).

In the script Code 24 (LBOConvPlots.m) we have chosen the following five schemes; the explicit, partitioned, theta and SSB Euler schemes and the trigonometric scheme. The reference solution is calculated alongside the approximations in order to prevent eventual memory issues. The parameters used in this case
are $\alpha = 1$ and $\omega = 3$ with initial value $x_0 = [1, 0]$. We can see how the strong error develops over the time interval $[0, 1]$ in Figure 9.1 and the strong error at the end point in Figure 9.2. The other numerical schemes are still available in Code 24 as comments.

All schemes for the LBO shown in Figure 9.2 display the strong order of convergence $\gamma = 1$.

![Figure 9.1](image1.png)

Figure 9.1. Given the explicit Euler, implicit Euler, Milstein and theta Euler schemes, the estimated strong error of the LBO, over $t \in [0, 1]$. 95% CI shown in black.

Leaving the strong error behind we now look at the expected value of the energy, which is given in both [2] and [22], though the latter gives the special case where $\omega = 1$. After the theorem we will evaluate the weak error obtained by a few schemes.

![Figure 9.2](image2.png)

Figure 9.2. Given the explicit Euler, implicit Euler, Milstein and theta Euler, $\theta = 0.5$, schemes, the estimated strong error of the LBO, at $T = 1$. 95% CI shown as bar plots. $C_i \in \mathbb{R}, i = 1, 2$ are constants used to translate the support lines.
Theorem 9.7 ([2], eq. (8)). The expected value of the energy (9.6) of the exact solution of the LBO (9.14) satisfies

\[ E[g(\omega, x^n_t, x^n_2)] = \frac{1}{2}(\omega^2(x^n_1)^2 + (x^n_2)^2 + \alpha^2 t). \]

**Proof.** Define

\[
\begin{align*}
a_1^n &= x^n_1 \cos(\omega t) + \frac{1}{\omega} x^n_2 \sin(\omega t), \\
a_2^n &= -\omega x^n_1 \sin(\omega t) + x^n_2 \cos(\omega t), \\
b_1^n &= \frac{\alpha}{\omega} \int_0^t \sin(\omega(t-s))dW_s, \\
b_2^n &= \alpha \int_0^t \cos(\omega(t-s))dW_s.
\end{align*}
\]

Then from (9.14) it follows that \( x^n_1 = a_1^n + b_1^n \) and \( x^n_2 = a_2^n + b_2^n \) we have

\[ E[g(\omega, x^n_t, x^n_2)] = \frac{1}{2} E[\omega^2(a_1^n)^2 + (a_1^n)^2 + (a_2^n)^2 + (a_2^n)^2 + 2\omega^2(a_1^n b_1^n) + 2(a_2^n b_2^n)]. \] (9.18)

Now we can look at it one piece at a time. First, the deterministic part:

\[ E[\omega^2(a_1^n)^2 + (a_2^n)^2] = \omega^2 \left(x^n_1 \cos(\omega t) + \frac{1}{\omega} x^n_2 \sin(\omega t)\right)^2 + (-\omega x^n_1 \sin(\omega t) + x^n_2 \cos(\omega t))^2 \]

\[ = \omega^2(x^n_1)^2 + \omega^2(x^n_2)^2 + \sin^2(\omega t) + 2\omega^2(a_1^n b_1^n) + 2(a_2^n b_2^n). \] (9.19)

Continuing, we get using the Itô isometry, Theorem 5.7, that

\[ E[\omega^2(b_1^n)^2 + (b_2^n)^2] = E \left[ \left( \alpha \int_0^t \sin(\omega(t-s))dW_s \right)^2 + \left( \alpha \int_0^t \cos(\omega(t-s))dW_s \right)^2 \right] \]

\[ = \alpha^2 E \left[ \int_0^t \sin^2(\omega(t-s)) + \cos^2(\omega(t-s))dW_s \right] = \alpha^2 t. \] (9.20)

Further, using Theorem 5.6 and that \( a_1^n, a_2^n \) are deterministic yields

\[ E[a_1 b_1] = E[a_2 b_2] = 0. \] (9.21)

Linearity of the expected value gives, when we insert (9.19), (9.20) and (9.21) into (9.18), that

\[ E[g(\omega, x^n_t, x^n_2)] = \frac{1}{2}(\omega^2(x^n_1)^2 + (x^n_2)^2 + \alpha^2 t). \]

\[ \square \]

Given an approximation \( x_n = [x^n_1, x^n_2] \), obtained by any given scheme, of the exact solution \( x_t = [x^n_1, x^n_2] \) for \( t_n = n\Delta t \), the weak error can be written as

\[ |E[g(\omega, x^n_t, x^n_2)] - E[g(\omega, x^n_t, x^n_2)]|. \]

We have implemented this in [Code 25](LBOWeakEnergy.m) and again we have excluded a few schemes which the reader can easily modify the script to investigate themselves. Here we used only the explicit, partitioned and SSB Euler schemes in order to best illustrate their convergence in weak error. Here we have chosen the parameters \( \alpha = 0.2 \) and \( \omega = 3 \), with the same initial value as before, \( x_0 = [1, 0] \). The resulting plots can be seen in Figure 9.3 and Figure 9.4. All schemes for the LBO shown in Figure 9.4 display the weak order of convergence \( \beta = 1 \) for the energy.

Leaving behind the weak errors, we can see in both [2] and [22] some theoretical results regarding the expected energy of the approximations yielded by a selection of schemes. The theorems covering the explicit and implicit Euler, provided by [22], will be slightly generalized to include some of the cases where \( \omega \neq 1 \) and later we will tackle the cases where \( \eta \neq 0 \). First up is a theorem, provided by [2], concerning the trigonometric scheme which displays excellent precision for the long time approximation of the energy.
Figure 9.3. Given the explicit, partitioned and SSB Euler schemes, the estimated weak error of the energy of the LBO over $t \in [0, 0.8]$. 95% CI shown in black.

Figure 9.4. The weak error of the energy of the LBO at $T = 0.8$ for the explicit, partitioned and SSB Euler schemes. 95% CI shown as bar plots, $C_i \in \mathbb{R}$, $i = 1, 2$ constants used to translate. $C_i \in \mathbb{R}$, $i = 1, 2$ are constants used to translate the support lines.
Theorem 9.8 ([2], Theorem 2.2). The numerical solution \([x_n^1, x_n^2]^{T}\) of the LBO (9.4) obtained using the trigonometric scheme (9.10) with initial values \(x_0 = [x_0^1, x_0^2]\), satisfies
\[
E[g(x_n^1, x_n^2)] = \frac{1}{2}(\omega^2(x_n^1)^2 + (x_n^2)^2) + \frac{\alpha^2}{2}t_n
\]
for all times \(t_n = n\Delta t\).

Proof. Given \(x_n = [x_n^1, x_n^2]\) we can calculate \(x_{n+1} = [x_{n+1}^1, x_{n+1}^2]\). We expand the approximation matrix multiplication immediately, as obtained by (9.10), and insert it into the expected value of \(g\).

\[
E[g(x_{n+1}^1, x_{n+1}^2)] = \frac{1}{2}E\left[\omega^2(x_{n+1}^1)^2 + (x_{n+1}^2)^2\right] = \frac{1}{2}E\left[\omega^2(\cos(\omega\Delta t)x_n^1 + \omega^{-1}\sin(\omega\Delta t)x_n^2 + \alpha\omega^{-1}\sin(\omega\Delta t)\Delta W_n)^2\right] + \frac{\omega^2}{2}E\left[\alpha\omega^{-1}\sin(\omega\Delta t)x_n^1 + \cos(\omega\Delta t)x_n^2\right]^2
\]

Now, since \(E[\Delta W_n] = 0\) and \(E[(\Delta W_n)^2] = \Delta t\), from Definition 5.1 we have that
\[
E[g(x_0^1, x_0^2)] = E[g(x_1^1, x_1^2)] + \frac{\alpha^2\Delta t}{2}.
\]
Through induction we are done. □

Note how the result obtained from Theorem 9.8 coincides with Theorem 9.7. Proceeding with the explicit Euler scheme we get the following theorem. The proof will use a theoretical result which does not fit with the rest of this essay, which is why that specific piece has been moved to the appendix.

Theorem 9.9 ([22], Theorem 3). The numerical solution \([x_n^1, x_n^2]^{T}\) of the LBO (9.4) obtained using the explicit Euler scheme (9.7) with the assumption \(t_n\omega^2\Delta t \leq 2.5\) and initial values \(x_0^1 = 1, x_0^2 = 0\), satisfies
\[
E[g(x_n^1, x_n^2)] \geq \frac{\omega^2}{2}e^{\frac{\omega^2\Delta t}{2}}
\]
for all times \(t_n = n\Delta t\).

Proof. First we unpack the multiplication for \(n+1\). We get
\[
E[g(x_{n+1}^1, x_{n+1}^2)] = E\left[\frac{1}{2}(\omega^2(x_{n+1}^1)^2 + (x_{n+1}^2)^2)\right] = \frac{1}{2}E\left[\omega^2(x_n^1 + \Delta tx_n^2)^2 + (x_n^2 - \omega^2\Delta tx_n^1 + \alpha\Delta W_n)^2\right].
\]
Expanding the squares and rearranging yields
\[
E[g(x_{n+1}^1, x_{n+1}^2)] = \frac{1}{2}E[\omega^2(x_n^1)^2 + (x_n^2)^2] + \frac{1}{2}E[\omega^2\Delta t^2(\omega^2(x_n^1)^2 + (x_n^2)^2)] + \frac{1}{2}E[\alpha^2(\Delta W_n)^2] + E[\alpha\Delta W_n x_n^2 - \omega^2\alpha\Delta t\Delta W_n x_n^1].
\]
Now we use that \(\Delta W_n\) is independent of \(x_n^1\) and \(x_n^2\), that \(E[\Delta W_n] = 0\) and that \(E[(\Delta W_n)^2] = \Delta t\), which follows from Definition 5.1. Further, identifying the energy of the previous step gives us through induction that
\[
E[g(x_{n+1}^1, x_{n+1}^2)] \geq (1 + \omega^2\Delta t^2)E[g(x_n^1, x_n^2)] + \frac{\alpha^2\Delta t}{2}
\]
\[
\geq (1 + \omega^2\Delta t^2)E[g(x_n^1, x_n^2)] \geq (1 + \omega^2\Delta t^2)^{n+1}E[g(x_0^1, x_0^2)] = (1 + \omega^2\Delta t^2)^{n+1}\frac{\omega^2}{2}.
\]
I.e. we have for step \( n \) that

\[
E[g(\omega, x_n^1, x_n^2)] \geq \frac{\omega^2}{2} \left(1 + \frac{(\omega^2 \Delta t)t_n}{n}\right)^n.
\]

Since the right hand side is strictly increasing, any lower bound for the case where \( n = 1 \) holds for every \( n \). I.e. that

\[
E[g(\omega, x_n^1, x_n^2)] \geq \frac{\omega^2}{2} \left(1 + \frac{(\omega^2 \Delta t)t_n}{n}\right)^n \geq \frac{\omega^2}{2} \left(1 + (\omega^2 \Delta t)t_n\right).
\]

We now apply Lemma A.1 where

\[
c = \omega^2 \Delta t
\]

and we get the sought inequality, applying the restriction that \( t_n c = t_n \omega^2 \Delta t \leq 2.5,
\]

\[
E[g(\omega, x_n^1, x_n^2)] \geq \frac{\omega^2}{2} e^{\frac{\omega^2 \Delta t t_n}{2}}. \quad \square
\]

As with the explicit Euler scheme, we can show a bound for the implicit Euler scheme. Though not a lower bound, but an upper.

**Theorem 9.10** ([22], Theorem 4). The numerical solution \([x_n^1, x_n^2]^T\) of the LBO ([9.4]) obtained using the implicit Euler scheme ([9.8]) with initial values \( x_0^1 = 1 \) and \( x_0^2 = 0 \), satisfies

\[
E[g(\omega, x_n^1, x_n^2)] \leq \frac{\omega^2}{2} + \frac{\alpha^2 t_n}{2 \Delta t^2 \omega^2}
\]

for all times \( t_n = n \Delta t \).

**Proof.** First step, the inverse found in the scheme, see ([9.8]). We get

\[
\left(I - \begin{bmatrix} 0 & 1 \\ -\omega^2 & 0 \end{bmatrix} \Delta t\right)^{-1} = \frac{1}{\omega^2 \Delta t^2 + 1} \begin{bmatrix} 1 & \Delta t \\ -\omega^2 \Delta t & 1 \end{bmatrix}.
\]

Performing the matrix multiplication yields

\[
E[g(\omega, x_{n+1}^1, x_{n+1}^2)] = \frac{1}{2} E\left[\omega^2 (x_{n+1}^1)^2 + (x_{n+1}^2)^2\right] = \frac{1}{2(\omega^2 \Delta t^2 + 1)^2} E\left[\omega^2 (x_n^1 + \Delta t x_n^2 + \alpha \Delta t \Delta W_n)^2 + (-\omega^2 \Delta t x_n^1 + x_n^2 + \alpha \Delta W_n)^2\right].
\]

Expanding and rearranging while using that \( \Delta W_n \) is independent of \( x_n^1 \) and \( x_n^2 \), \( E[\Delta W_n] = 0 \), and \( E[\Delta W_n] = \Delta t \), which follows from Definition 5.1 yields us

\[
E[g(\omega, x_{n+1}^1, x_{n+1}^2)] = \frac{1}{2(\omega^2 \Delta t^2 + 1)^2} \left(E[\omega^2 (x_n^1)^2 + (x_n^2)^2] + \omega^2 \Delta t^2 E[\omega^2 (x_n^1)^2 + (x_n^2)^2] + \alpha^2 \omega^2 \Delta t^3 + \alpha^2 \Delta t\right).
\]

Simplifying this and relying on induction yields the sought inequality, though at time \( t_{n+1} \).

\[
E[g(\omega, x_{n+1}^1, x_{n+1}^2)] = \frac{1}{(\omega^2 \Delta t^2 + 1)^2} \left(E[\omega^2 (x_n^1)^2 + (x_n^2)^2] + \omega^2 \Delta t^2 + \frac{\alpha^2 \Delta t}{2} \leq E[g(\omega, x_{n+1}^1, x_{n+1}^2)] + \frac{\alpha^2 \Delta t}{2(\omega^2 \Delta t^2 + 1)}\right.
\]

\[
\leq E[g(\omega, x_0^1, x_0^2)] + (n+1) \frac{\alpha^2 \Delta t}{2 \Delta t^2 \omega^2} = \frac{\omega^2}{2} + \frac{\alpha^2 t_{n+1}}{2 \Delta t^2 \omega^2}. \quad \square
\]

Finally we have a result concerning the partitioned Euler scheme.

**Theorem 9.11.** Assume that \([x_n^1, x_n^2]^T\) is the numerical solution of the LBO ([9.4]) obtained using the partitioned Euler scheme ([9.9]) with \( \omega = 1 \) and initial values \( x_0^1 = 1, x_0^2 = 0 \). Then there exists functions

\[
C_{\text{lower}}, C_{\text{upper}} : (0, 2) \to \mathbb{R}^+
\]
such that, for $\Delta t < 2$,
\[
C_{\text{lower}}(\Delta t) \left( \frac{1}{2} + \frac{\alpha^2 t_n}{2} \right) \leq E[g(1, x_n^1, x_n^2)] \leq C_{\text{upper}}(\Delta t) \left( \frac{1}{2} + \frac{\alpha^2 t_n}{2} \right)
\]
for all times $t_n = n\Delta t$, where $C_{\text{lower}}$ and $C_{\text{upper}}$ are independent of $n$ with
\[
C_{\text{lower}}(\Delta t) = 1 + O(\Delta t), \quad C_{\text{upper}}(\Delta t) = 1 + O(\Delta t) \quad \text{as} \ \Delta t \to 0.
\]

**Proof.** See e.g. [22].

We see [Theorem 9.11](#) demonstrated in the subplots of Figure 9.19 and Figure 9.26 where $\eta = 0$. But this behavior seems, as indicated by the rest of the subplots, to hold for $\eta \neq 0$ as well.

We are now ready to move on to the more general linear stochastic oscillator, where we also use a CPr or a CCPoPr.

### 9.5 The linear stochastic oscillator driven by a Wiener process and either a CPr or a CCPoPr.

Beginning with the exact solution of the LBPO (9.5) we get, much like in Theorem 9.6, the following theorem.

**Theorem 9.12.** The LBPO (9.5), with initial values $[x_0^1, x_0^2]$, has the exact solution
\[
x_t = \begin{bmatrix} x_0^1 \cos(\omega t) + x_0^2 \sin(\omega t) \\ -\omega x_0^1 \sin(\omega t) + x_0^2 \cos(\omega t) \end{bmatrix} + \alpha \int_0^t \begin{bmatrix} \sin(\omega(t-s)) \\ \omega \cos(\omega(t-s)) \end{bmatrix} dW_s + \eta \int_0^t \begin{bmatrix} \sin(\omega(t-s)) \\ \omega \cos(\omega(t-s)) \end{bmatrix} dN^*_s,
\]
or, equivalently,
\[
x_t^1 = x_0^1 \cos(\omega t) + x_0^2 \omega^{-1} \sin(\omega t) + \alpha \int_0^t \omega^{-1} \sin(\omega(t-s))dW_s + \eta \int_0^t \omega^{-1} \sin(\omega(t-s))dN^*_s,
\]
\[
x_t^2 = -\omega x_0^1 \sin(\omega t) + x_0^2 \cos(\omega t) + \alpha \int_0^t \cos(\omega(t-s))dW_s + \eta \int_0^t \cos(\omega(t-s))dN^*_s.
\]

**Proof.** Striving to apply Theorem 6.19 to (9.5) we see that
\[
F(t) = \begin{bmatrix} 0 & 1 \\ -\omega^2 & 0 \end{bmatrix}, \quad f(t) = \begin{bmatrix} 0 \\ 0 \end{bmatrix},
\]
\[
G(t) = \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix}, \quad g(t) = \begin{bmatrix} 0 \\ \alpha \end{bmatrix},
\]
\[
H(t) = \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix}, \quad h(t) = \begin{bmatrix} 0 \\ \eta \end{bmatrix}.
\]

Obviously the requirement that $H(t)h(t) = \tilde{0}$ is fulfilled. The homogeneous equation for the problem (9.5) is
\[
\begin{bmatrix} dx_t^1 \\ dx_t^2 \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ -\omega^2 & 0 \end{bmatrix} \begin{bmatrix} x_t^1 \\ x_t^2 \end{bmatrix} dt,
\]
which Lemma 9.4 states has the fundamental matrix, and corresponding inverse matrix,
\[
\Phi_t = \begin{bmatrix} \cos(\omega t) & \sin(\omega t) \\ -\omega \sin(\omega t) & \cos(\omega t) \end{bmatrix}
\]
and
\[
\Phi_t^{-1} = \begin{bmatrix} \cos(\omega t) & -\sin(\omega t) \\ \omega \sin(\omega t) & \cos(\omega t) \end{bmatrix}.
\]
Now, the solution to (9.5) given by Theorem 6.19 is
\[ x_t = \Phi_t \left( x_0 + \int_0^t \Phi_s^{-1} \left[ f(s) - G(s)g(s) \right] ds + \int_0^t \Phi_s^{-1} g(s) dW_s + \int_0^t \Phi_s^{-1} h(s) dN^*_s \right) \]  \tag{9.23}
and, as in the proof of Theorem 9.6, we evaluate the first term as
\[ \Phi_t x_0 = \begin{bmatrix} x_0^1 \cos(\omega t) + x_0^2 \sin(\omega t) \\ -\omega x_0^1 \sin(\omega t) + x_0^2 \cos(\omega t) \end{bmatrix}. \]  \tag{9.24}
A brief inspection reveals that, again, the second term of (9.23) is 0. Through Lemma 9.5 we get that the third term evaluates to
\[ \Phi_t \int_0^t \Phi_s^{-1} g(s) dW_s = \alpha \hat{\Phi}_t \int_0^t \begin{bmatrix} \sin(\omega(t-s)) \\ \omega \cos(\omega(t-s)) \end{bmatrix} dW_s. \]  \tag{9.25}
The fourth term evaluates identically, with the exception of \( \alpha \) changing for \( \eta \). We get
\[ \Phi_t \int_0^t \Phi_s^{-1} h(s) dN^*_s = \eta \hat{\Phi}_t \int_0^t \begin{bmatrix} \sin(\omega(t-s)) \\ \omega \cos(\omega(t-s)) \end{bmatrix} dN^*_s. \]  \tag{9.26}
Inserting (9.24), (9.25) and (9.26) into (9.23) concludes the proof. \( \square \)

As in subsection 9.4, we will require a reference solution in order to approximate the strong error. Again, we have calculated this reference solution using the trigonometric scheme with step size \( \Delta t = 2^{-14} \).

The implementation to calculate the strong error can be seen in Code 26 (LBPOCompFourStrongTime.m) and Code 27 (LBPOCompCompFourStrongTime.m), where the former simulate the LBPO driven by a CPoPr and the latter simulate the LBPO driven by a CCPoPr. We used the parameters \( \omega = 3, \lambda = 3, \alpha, \eta \in \{0, 0.2\} \).

Finally, we used the standard PoPr as the CPoPr, as well as for the underlying CPoPr for the CCPoPr. The plots of the strong errors over the interval \( t \in [0,1] \) can be seen in Appendix B and Appendix C while we in Figure 9.5 through Figure 9.16 display the strong errors only at \( T = 1 \).

Throughout Figure 9.5 to Figure 9.16 all schemes for the LBPO display the strong order of convergence \( \gamma = 1 \), with the exception of the theta Euler scheme, \( \theta = 0.5 \), and the trigonometric scheme. In the deterministic setting, \( \alpha = \eta = 0 \), the former display \( \gamma = 2 \) and the latter does not reveal any order of convergence due to being at the MATLAB working precision.
Figure 9.5. The strong error of the LBPO driven by a CPoPr, at $T = 1$ obtained using the explicit Euler scheme. $C_i \in \mathbb{R}$, $i = 1, 2, 3$ are constants used to translate the support lines.

Figure 9.6. The strong error of the LBPO driven by a CPoPr, at $T = 1$ obtained using the implicit Euler scheme. $C_i \in \mathbb{R}$, $i = 1, 2, 3$ are constants used to translate the support lines.
Figure 9.7. The strong error of the LBPO driven by a CPoPr, at $T = 1$ obtained using the partitioned Euler scheme. $C_i \in \mathbb{R}, i = 1, 2, 3$ are constants used to translate the support lines.

Parameters: $\alpha = 3, \lambda = 3$. Batch size: 64. Number of batches: 8.

Figure 9.8. The strong error of the LBPO driven by a CPoPr, at $T = 1$ obtained using the theta Euler scheme, $\theta = 0.5$. $C_i \in \mathbb{R}, i = 1, 2, 3$ are constants used to translate the support lines.
Parameters: $\alpha=3, \ \lambda=3$. Batch size: 64. Number of batches: 8.

Figure 9.9. The strong error of the LBPO driven by a CPoPr, at $T=1$ obtained using the SSB Euler scheme. $C_i \in \mathbb{R}, \ i = 1, 2, 3$ are constants used to translate the support lines.

Figure 9.10. The strong error of the LBPO driven by a CPoPr, at $T=1$ obtained using the CSSB Euler scheme. $C_i \in \mathbb{R}, \ i = 1, 2, 3$ are constants used to translate the support lines.
Figure 9.11. The strong error of the LBPO driven by a CPoPr, at $T = 1$ obtained using the trigonometric scheme. $C_i \in \mathbb{R}, i = 1, 2, 3$ are constants used to translate the support lines.

Figure 9.12. The strong error of the LBPO driven by a CCPoPr, at $T = 1$ obtained using the explicit Euler scheme. $C_i \in \mathbb{R}, i = 1, 2, 3$ are constants used to translate the support lines.
Figure 9.13. The strong error of the LBPO driven by a CCPoPr, at $T = 1$ obtained using the implicit Euler scheme. $C_i \in \mathbb{R}$, $i = 1, 2, 3$ are constants used to translate the support lines.

Figure 9.14. The strong error of the LBPO driven by a CCPoPr, at $T = 1$ obtained using the partitioned Euler scheme. $C_i \in \mathbb{R}$, $i = 1, 2, 3$ are constants used to translate the support lines.
Parameters: $\omega=3$, $\lambda=3$. Batch size: 64. Number of batches: 8.

Figure 9.15. The strong error of the LBPO driven by a CCPoPr, at $T = 1$ obtained using the theta Euler scheme, $\theta = 0.5$. $C_i \in \mathbb{R}$, $i = 1, 2, 3$ are constants used to translate the support lines.

Figure 9.16. The strong error of the LBPO driven by a CCPoPr, at $T = 1$ obtained using the trigonometric scheme. $C_i \in \mathbb{R}$, $i = 1, 2, 3$ are constants used to translate the support lines.
Now we observe something which may be unexpected. The behavior of the expected value of the energy for the LBPO will depend on whether the process $N^*$ is either a CPoPr or a CCPoPr. If we take the intuitive route and investigate the case for a CPoPr first, we get the following theorem.

**Theorem 9.13.** The expected value of the energy of the exact solution of the LBPO driven by a CPoPr $N^* = \tilde{N}$, satisfies

\[
E[g(\omega, x_t^1, x_t^2)] = \frac{1}{2} (\omega^2 (x_0^1)^2 + (x_0^2)^2 + \alpha^2 t + \eta^2 \lambda \mathbb{E}[X^2_t]) (\cos(\omega t) - 1) + \eta \lambda \mathbb{E}[X_1] \frac{\eta^2 \lambda^2}{\omega^2} \mathbb{E}[X_1]^2 \sin(\omega t).
\]

**Proof.** Define

\[
a_t^1 = x_t^1 \cos(\omega t) + \frac{1}{\omega} x_t^2 \sin(\omega t),
a_t^2 = -\omega x_t^1 \sin(\omega t) + x_t^2 \cos(\omega t),
b_t^1 = \frac{\alpha}{\omega} \int_0^t \sin(\omega(t - s))dW_s,
b_t^2 = \alpha \int_0^t \cos(\omega(t - s))dW_s,
c_t^1 = \frac{\eta}{\omega} \int_0^t \sin(\omega(t - s))d\tilde{N}_s,
c_t^2 = \frac{\eta}{\omega} \int_0^t \cos(\omega(t - s))d\tilde{N}_s.
\]

Then for $x_t^1 = a_t^1 + b_t^1 + c_t^1$ and $x_t^2 = a_t^2 + b_t^2 + c_t^2$ we have

\[
E[g(\omega, x_t, y_t)] = \frac{1}{2} E \left[ \omega^2 (a_t^1)^2 + (a_t^2)^2 + (a_t^1)^2 + (b_t^1)^2 + (b_t^2)^2 + (c_t^1)^2 \right]
\]

\[
= \frac{1}{2} E \left[ \omega^2 (a_t^1)^2 + (a_t^2)^2 + \omega^2 (b_t^1)^2 + (b_t^2)^2 + \omega^2 (c_t^1)^2 + (c_t^2)^2 \right]
\]

\[
+ E \left[ \omega^2 (a_t^1 b_t^1 + a_t^1 c_t^1 + b_t^1 c_t^1) + (a_t^2 b_t^2 + a_t^2 c_t^2 + b_t^2 c_t^2) \right].
\]  

(9.27)

Now we can look at it one piece at a time. First, the deterministic part:

\[
E \left[ \omega^2 (a_t^1)^2 + (a_t^2)^2 \right] = \omega^2 \left( x_0^1 \cos(\omega t) + \frac{1}{\omega} x_0^2 \sin(\omega t) \right)^2 + (-\omega x_0^1 \sin(\omega t) + x_0^2 \cos(\omega t))^2
\]

\[
= \omega^2 (x_0^1)^2 \cos^2(\omega t) + \sin^2(\omega t) + (x_0^2)^2 \cos^2(\omega t) + \sin^2(\omega t) = \omega^2 (x_0^1)^2 + (x_0^2)^2.
\]  

(9.28)

Continuing, we get using the Itô isometry, Theorem 5.7, that

\[
E[\omega^2 (b_t^1)^2 + (b_t^2)^2] = E \left[ \left( \alpha \int_0^t \sin(\omega(t - s))dW_s \right)^2 \right] + \left( \alpha \int_0^t \cos(\omega(t - s))dW_s \right)^2
\]

\[
= \alpha^2 \mathbb{E} \left[ \int_0^t \sin^2(\omega(t - s)) + \cos^2(\omega(t - s))ds \right] = \alpha^2 t.
\]  

(9.29)

Further, using that $a_t^1, a_t^2$ are deterministic, that $\tilde{N}_t$ and $W_t$ are independent and Theorem 5.6 yields us

\[
E[a_t^1 b_t^1] = E[a_t^1 c_t^1] = E[a_t^2 b_t^2] = E[a_t^2 c_t^2] = 0.
\]  

(9.30)

Summarizing this, as an intermediate step, linearity of the expected value gives that

\[
E[g(\omega, x_t^1, x_t^2)] = \frac{1}{2} (\omega^2 (x_0^1)^2 + (x_0^2)^2 + \alpha^2 t) + \frac{1}{2} E \left[ \omega^2 (c_t^1)^2 + (c_t^2)^2 \right] + E \left[ \omega^2 (a_t^1 b_t^1 + a_t^2 c_t^2) \right].
\]

Due to $a_t^1$ and $a_t^2$ being deterministic and Theorem 6.21, which gives that

\[
E \left[ \int_0^t f(s)d\tilde{N}_s \right] = \lambda \mathbb{E}[X_1] \int_0^t f(s)ds,
\]

53
we see that
\[
E[\omega^2 a_1^2 c_1^2 + a_1^2 c_1^2] = \omega^2 a_1^2 E[c_1]^2 + a_1^2 E[c_1^2] = \omega^2 a_1^2 E \left[ \frac{\eta}{\omega} \int_0^t \sin(\omega(t-s))d\hat{N}_s \right] + a_1^2 E \left[ \eta \int_0^t \cos(\omega(t-s))d\hat{N}_s \right] \\
= \omega \eta \lambda a_1^2 E[X_1] \int_0^t \sin(\omega(t-s))ds + \eta \lambda a_1^2 E[X_1] \int_0^t \cos(\omega(t-s))ds \\
= \eta \lambda E[X_1] \left( a_1^2 \cos(\omega(t-s)) \right)_0^t - \frac{\alpha^2}{\omega} \left[ \sin(\omega(t-s)) \right]_0^t = \eta \lambda E[X_1] \left( a_1^2 (1 - \cos(\omega t)) - \frac{\alpha^2}{\omega} (0 - \sin(\omega t)) \right) \\
= \eta \lambda E[X_1] \left( x_0^2 \cos(\omega t) + \frac{1}{\omega} x_0^2 \sin(\omega t) \right) (1 - \cos(\omega t)) + \left( -x_0^2 \sin(\omega t) + \frac{1}{\omega} x_0^2 \cos(\omega t) \right) \sin(\omega t) \\
= \eta \lambda E[X_1] \left( x_0^2 (\cos(\omega t) - 1) + \frac{x_0^2}{\omega} \sin(\omega t) \right). \quad (9.31)
\]

Now approaching the last part,
\[
E \left[ \int_0^t \sin(\omega(t-s))d\hat{N}_s \right] = \eta^2 E \left[ \left( \int_0^t \sin(\omega(t-s))d\hat{N}_s \right)^2 \right] + \left( \int_0^t \cos(\omega(t-s))d\hat{N}_s \right)^2.
\]

Lemma 6.24 yields
\[
E \left[ \left( \int_0^t \sin(\omega(t-s))d\hat{N}_s \right)^2 \right] + \left( \int_0^t \cos(\omega(t-s))d\hat{N}_s \right)^2 \\
= \lambda E[X_1]^2 E \left[ \int_0^t \sin(\omega(t-s))^2ds \right] + \lambda^2 E[X_1]^2 E \left[ \int_0^t \sin(\omega(t-s))^2ds \right] \\
+ \lambda E[X_1]^2 E \left[ \int_0^t \cos(\omega(t-s))^2ds \right] + \lambda^2 E[X_1]^2 E \left[ \int_0^t \cos(\omega(t-s))^2ds \right] \\
= \lambda E[X_1]^2 t + \lambda^2 E[X_1]^2 \left( \frac{1}{\omega} \cos(\omega(t-s)) \right)_0^t + \lambda^2 E[X_1]^2 \left( \frac{1}{\omega} \sin(\omega(t-s)) \right)_0^t \\
= \lambda E[X_1]^2 t + \frac{\lambda^2}{\omega^2} E[X_1]^2 (1 - 2 \cos(\omega t) + \cos^2(\omega t) + \sin^2(\omega t)) \\
= \lambda E[X_1]^2 t + \frac{2\lambda^2}{\omega^2} E[X_1]^2 (1 - \cos(\omega t)) .
\]

Which means that
\[
E \left[ \omega^2 (c_1^2)^2 + (c_1^2)^2 \right] = \eta^2 \left( \lambda E[X_1]^2 t + \frac{2\lambda^2}{\omega^2} E[X_1]^2 (1 - \cos(\omega t)) \right). \quad (9.32)
\]

Inserting (9.28) to (9.32) into (9.27) gives us
\[
E[g(\omega, x_1^1, x_1^2) = \frac{1}{2} (\omega^2 (x_0^1)^2 + (x_0^2)^2 + \alpha^2 t) \\
+ \eta \lambda E[X_1] \left( x_0^1 (\cos(\omega t) - 1) + \frac{x_0^2}{\omega} \sin(\omega t) \right) \\
+ \eta^2 \left( \frac{\lambda}{2} E[X_1]^2 t + \frac{\lambda^2}{\omega^2} E[X_1]^2 (1 - \cos(\omega t)) \right) \\
= \frac{1}{2} \omega^2 (x_0^1)^2 + (x_0^2)^2 + \alpha^2 t + \eta \lambda E[X_1]^2 t + \eta^2 \left( \frac{\lambda}{2} E[X_1]^2 t + \frac{\lambda^2}{\omega^2} E[X_1]^2 (1 - \cos(\omega t)) \right) (\cos(\omega t) - 1) + \eta \lambda E[X_1] \frac{x_0^2}{\omega} \sin(\omega t).
\]
\[
\square
\]
We now illustrate the above result numerically. For the following plots and scripts we have used the parameters \( \omega = 3, \lambda = 3, \alpha \in \{0, 0.6\}, \eta \in \{0, 0.6\} \) and the standard PoPr as the CPoPr. I.e. we have chosen the trivial distribution with \( P(X_1 = 1) = 1 \).

The expected energies over the interval \( t \in [0, 10] \) can be seen in Figure 9.17 to Figure 9.23. These plots were obtained by Code 28 (LBPOFourEnergy.m).

The weak errors have been simulated over a smaller interval, \( t \in [0, 0.25] \), and they can be seen in Appendix D. These plots have been produced by Code 29 (LBPOFourWeakEnergy.m) and Code 31 (LBPOFourWeakEnergyTrig.m). The reason for the separate scripts is due to the higher precision of the theta Euler and trigonometric scheme, which requires a much larger number of simulations in order to overcome the Monte Carlo error.

Throughout Figure 9.17 to Figure 9.23 some properties which are noteworthy are:

- The expected value of the energy by the explicit Euler scheme diverges rapidly for larger \( \Delta t \).
- The expected value of the energy by the implicit, SSB and CSSB Euler schemes are dampened.
- The expected value of the energy by the partitioned Euler scheme oscillates around the theoretical expected value of the energy, given a small enough \( \Delta t \).
- The expected value of the energy of the theta Euler scheme, \( \theta = 0.5 \), and the trigonometric scheme follows the theoretical expected value of the energy well, even for large \( \Delta t \).

![Parameters: \( \omega = 3, \lambda = 3 \). Batch size: 1024. Number of batches: 16.](image)

**Figure 9.17.** The expected energy of the LBPO driven by a CPoPr, over \( t \in [0, 10] \), obtained using the explicit Euler scheme.
Figure 9.18. The expected energy of the LBPO driven by a CPoPr, over \( t \in [0, 10] \), obtained using the implicit Euler scheme.

Figure 9.19. The expected energy of the LBPO driven by a CPoPr, over \( t \in [0, 10] \), obtained using the partitioned Euler scheme.
The expected energy of the LBPO driven by a CPoPr, over $t \in [0, 10]$, obtained using the theta Euler scheme, $\theta = 0.5$.

**Figure 9.20**

The expected energy of the LBPO driven by a CPoPr, over $t \in [0, 10]$, obtained using the SSB Euler scheme.

**Figure 9.21**
Figure 9.22. The expected energy of the LBPO driven by a CPoPr, over \( t \in [0, 10] \), obtained using the CSSB Euler scheme.

Figure 9.23. The expected energy of the LBPO driven by a CPoPr, over \( t \in [0, 10] \), obtained using the trigonometric scheme.
An observant reader will perhaps have noticed the following lemma.

**Lemma 9.14.** The expected value of the energy \([9.6]\) of the exact solution of the LBPO \([9.22]\) driven by a CPoPr \(N^* = \tilde{N}\) with \(E[X_1] = 0\), satisfies
\[
E[g(\omega, x_1^t, x_2^t)] = \frac{1}{2} (\omega^2(x_0^1)^2 + (x_0^2)^2 + \alpha^2 t + \eta^2 \lambda E[X_1^t]) t.
\]

**Proof.** It follows immediately from Theorem 9.13 and the fact that \(E[X_1] = 0\). \(\square\)

This lemma hints of something interesting. As seen in Lemma 6.11 we have that CPoPr’s with zero mean are also CCPoPr’s. We see that, should we use a CCPoPr for the LBPO, the linear drift of the expected energy, as seen in Theorem 9.7, is preserved. We get the following theorem:

**Theorem 9.15.** The expected value of the energy \([9.6]\) of the exact solution of the LBPO \([9.22]\) driven by a CPoPr \(N^* = \tilde{N}\), satisfies
\[
E[g(\omega, x_1^t, x_2^t)] = \frac{1}{2} (\omega^2(x_0^1)^2 + (x_0^2)^2 + \alpha^2 t + \eta^2 \lambda E[X_1^t]) t.
\]

**Proof.** Define
\[
\begin{align*}
a_1^t &= x_1^t \cos(\omega t) + \frac{1}{\omega} x_2^t \sin(\omega t), \\
a_2^t &= -\omega x_1^t \sin(\omega t) + x_2^t \cos(\omega t), \\
b_1^t &= \frac{\alpha}{\omega} \int_0^t \sin(\omega(t-s))dW_s, \\
b_2^t &= \alpha \int_0^t \cos(\omega(t-s))dW_s, \\
c_1^t &= \frac{\eta}{\omega} \int_0^t \sin(\omega(t-s))d\tilde{N}_s, \\
c_2^t &= \eta \int_0^t \cos(\omega(t-s))d\tilde{N}_s.
\end{align*}
\]
Then for \(x_1^t = a_1^t + b_1^t + c_1^t\) and \(x_2^t = a_2^t + b_2^t + c_2^t\) we have
\[
E[g(\omega, x_t, y_t)] = \frac{1}{2} E \left[ \omega^2(a_1^t)^2 + (a_2^t)^2 + (a_1^t c_1^t + b_1^t c_1^t) + (a_2^t b_2^t + a_1^t c_1^t + b_1^t c_1^t) + (a_2^t b_2^t + a_1^t c_1^t + b_1^t c_1^t) \right].
\]
(9.33)

Now we can look at it one piece at a time. First, the deterministic part:
\[
E \left[ \omega^2(a_1^t)^2 + (a_2^t)^2 \right] = \omega^2 \left( x_0^1 \cos(\omega t) + \frac{1}{\omega} x_0^2 \sin(\omega t) \right)^2 + \left( -\omega x_0^1 \sin(\omega t) + x_0^2 \cos(\omega t) \right)^2
\]
\[= \omega^2(x_0^1)^2(\cos^2(\omega t) + \sin^2(\omega t)) + (x_0^2)^2(\cos^2(\omega t) + \sin^2(\omega t)) = \omega^2(x_0^1)^2 + (x_0^2)^2.\]
(9.34)

Continuing, we get using the Itô isometry, Theorem 5.7 that
\[
E[\omega^2(b_1^t)^2 + (b_2^t)^2] = E \left[ \left( \alpha \int_0^t \sin(\omega(t-s))dW_s \right)^2 + \left( \alpha \int_0^t \cos(\omega(t-s))dW_s \right)^2 \right]
\]
\[= \alpha^2 E \left[ \int_0^t \sin^2(\omega(t-s)) + \cos^2(\omega(t-s))ds \right] = \alpha^2 t.\]
(9.35)

Further, using that \(a_1^t, a_2^t\) are deterministic, that \(d\tilde{N}_t\) and \(dW_t\) are independent and Theorem 5.6 yields us
\[
E[a_1^t b_1^t] = E[b_1^t c_1^t] = E[a_2^t b_2^t] = E[b_2^t c_2^t] = 0.
\]
(9.36)

Summarizing this, as an intermediate step, linearity of the expected value gives that
\[
E[g(\omega, x_1^t, x_2^t)] = \frac{1}{2} (\omega^2(x_0^1)^2 + (x_0^2)^2 + \alpha^2 t) + \frac{1}{2} E \left[ \omega^2(c_1^t)^2 + (c_2^t)^2 \right] + E \left[ \omega^2 a_1^t c_1^t + a_2^t c_2^t \right].
\]
Due to $a_1^4$ and $a_2^4$ being deterministic and Lemma 6.22, which gives that
\[ E \left[ \int_0^t f(s)d\tilde{N}_s \right] = 0, \]
we see that
\[ E[\omega^2 a_1^4 c_1^4 + a_2^4 c_2^4] = \omega^2 a_1^2 E[c_1^2] + a_2^2 E[c_2^2] = 0. \]  
(9.37)

Now approaching the last part, Theorem 6.23 yields
\[ E \left[ \omega^2 (c_1^2)^2 + (c_1^2)^2 \right] = E \left[ \left( \int_0^t \sin(\omega(t-s))d\tilde{N}_s \right)^2 + \left( \int_0^t \cos(\omega(t-s))d\tilde{N}_s \right)^2 \right] \]
\[ = \eta^2 \lambda E[X_1^2] E \left[ \int_0^t \sin^2(\omega(t-s)) + \cos^2(\omega(t-s))ds \right] = \eta^2 \lambda E[X_1^2] t. \]  
(9.38)

Which means that Inserting (9.34) to (9.38) into (9.33) gives us
\[ E[g(\omega, x_1^t, x_1^t)] = \frac{1}{2} (\omega^2 (x_1^t)^2 + (x_1^t)^2 + \alpha^2 t + \eta^2 \lambda E[X_1^2] t). \]

As before, we will now illustrate these properties numerically. We will exclude the SSB and CSSB Euler schemes due to them not being defined for stochastic processes driven by a CCPoPr. We have the same parameters, $\omega = 3$, $\lambda = 3$, $\alpha \in \{0, 0.6\}$, $\eta \in \{0, 0.6\}$ and the standard PoPr as the underlying CPoPr.

The expected energies over the interval $t \in [0, 10]$ can be seen in Figure 9.24 to Figure 9.28. These plots were obtained by Code 28 (LBPOFourEnergy.m).

The weak errors have been simulated over a smaller interval, $t \in [0, 0.25]$, and they can be seen in Appendix E. These plots have been produced by Code 29 (LBPOFourWeakEnergy.m), Code 30 (LBPOFourWeakEnergyTheta.m) and Code 31 (LBPOFourWeakEnergyTrig.m). The reason for the separate scripts is due to the higher precision of the theta Euler and trigonometric scheme, which requires a much larger number of simulations in order to overcome the Monte Carlo error. Note here that we have not given any weak error plot for the LBPO driven by a CCPoPr for the trigonometric scheme. This is due to the excellent energy preserving properties of this specific scheme, see below for a theoretical justification.

The schemes for Figure 9.24 to Figure 9.28 show much the same behavior as in Figure 9.17 to Figure 9.23.

**Figure 9.24.** The expected energy of the LBPO driven by a CCPoPr, over $t \in [0, 10]$, obtained using the explicit Euler scheme.
Figure 9.25. The expected energy of the LBPO driven by a CCPoPr, over $t \in [0,10]$, obtained using the implicit Euler scheme.

Figure 9.26. The expected energy of the LBPO driven by a CCPoPr, over $t \in [0,10]$, obtained using the partitioned Euler scheme.
Figure 9.27. The expected energy of the LBPO driven by a CCPoPr, over \( t \in [0,10] \), obtained using the theta Euler scheme, \( \theta = 0.5 \).

Figure 9.28. The expected energy of the LBPO driven by a CCPoPr, over \( t \in [0,10] \), obtained using the trigonometric scheme.
As in subsection 9.4, where we looked at bounds for the expected energy for some numerical schemes applied to the LBO (9.4), we present the corresponding bounds for the LBPO (9.5). Though we will not include one for the partitioned scheme.

**Theorem 9.16.** The numerical solution $[x_n^1, x_n^2]^T$ of the LBPO (9.5) driven by a CCPr $N^* = \tilde{N}$, obtained using the trigonometric scheme (9.10) with initial values $x_0^1 = 1, x_0^2 = 0$, satisfies

$$E[g(\omega, x_n^1, x_n^2)] = \frac{1}{2}(\omega^2(x_0^1)^2 + (x_0^2)^2) + \frac{\alpha^2 + \eta^2 \lambda E[X_0^2]}{2} t_n$$

for all times $t_n = n\Delta t$.

This means that the trigonometric scheme have the same linear drift in the expected energy as the exact solution, see Theorem 9.15.

**Proof.** Given $x_n = [x_n^1, x_n^2]$ we can calculate $x_{n+1} = [x_{n+1}^1, x_{n+1}^2]$. We expand the matrix multiplication immediately, as obtained by (9.10), and insert it into the expected value of $g$.

$$E[g(\omega, x_{n+1}^1, x_{n+1}^2)] = \frac{1}{2} E[\omega^2(x_{n+1}^1)^2 + (x_{n+1}^2)^2]$$

$$= \frac{1}{2} E \left[ \omega^2(\cos(\omega \Delta t)x_n^1 + \omega^{-1} \sin(\omega \Delta t)x_n^2 + \alpha \omega^{-1} \sin(\omega \Delta t)\Delta W_n + \eta \omega^{-1} \sin(\omega \Delta t)\Delta \tilde{N}_n)^2 
+ (-\omega \sin(\omega \Delta t)x_n^1 + \cos(\omega \Delta t)x_n^2 + \alpha \cos(\omega \Delta t)\Delta W_n + \eta \cos(\omega \Delta t)\Delta \tilde{N}_n)^2 \right]. \quad (9.39)$$

If we expand those squares one at a time we get that

$$(x_{n+1}^1)^2 = \cos^2(\omega \Delta t)x_n^1 + 2\omega^{-1} \sin(\omega \Delta t)x_n^1 \Delta W_n + \eta \omega^{-1} \sin(\omega \Delta t)\Delta \tilde{N}_n$$

$$= \cos^2(\omega \Delta t)(x_n^1)^2 + \omega^{-2} \sin^2(\omega \Delta t)(x_n^2)^2 + \alpha^2 \omega^{-2} \sin^2(\omega \Delta t)(\Delta W_n)^2 + \eta^2 \omega^{-2} \sin^2(\omega \Delta t)(\Delta \tilde{N}_n)^2$$

$$+ 2\omega^{-1} \cos(\omega \Delta t) \sin(\omega \Delta t)x_n^1 x_n^2 + 2\alpha \omega^{-1} \sin(\omega \Delta t) \sin(\omega \Delta t)x_n^1 \Delta W_n + 2\eta \omega^{-1} \cos(\omega \Delta t) \sin(\omega \Delta t)x_n^1 \Delta \tilde{N}_n$$

$$+ 2\alpha \omega^{-1} \sin^2(\omega \Delta t)x_n^2 \Delta W_n + 2\eta \omega^{-1} \cos^2(\omega \Delta t)x_n^2 \Delta \tilde{N}_n$$

and that

$$(x_{n+1}^2)^2 = (-\omega \sin(\omega \Delta t)x_n^1 + \cos(\omega \Delta t)x_n^2 + \alpha \cos(\omega \Delta t)\Delta W_n + \eta \cos(\omega \Delta t)\Delta \tilde{N}_n)^2$$

$$= \omega^2 \sin^2(\omega \Delta t)(x_n^1)^2 + 2\alpha \omega \sin(\omega \Delta t) \cos(\omega \Delta t)x_n^1 \Delta W_n - 2\eta \omega \sin(\omega \Delta t) \cos(\omega \Delta t)x_n^1 \Delta \tilde{N}_n$$

$$+ 2\alpha \cos^2(\omega \Delta t)x_n^1 \Delta W_n + 2\eta \cos^2(\omega \Delta t)x_n^2 \Delta W_n + 2\alpha \eta \cos^2(\omega \Delta t)\Delta \tilde{N}_n.$$ 

Inserting these two expanded squares into (9.39), canceling out a few terms and using a trigonometric identity gives

$$\frac{1}{2} E \left[ \omega^2(x_{n+1}^1)^2 + (x_{n+1}^2)^2 \right]$$

$$= \frac{1}{2} E \left[ \omega^2(x_n^1)^2 + (x_n^2)^2 + 2\alpha x_n^1 \Delta W_n + 2\eta x_n^2 \Delta \tilde{N}_n + 2\alpha \Delta W_n \Delta \tilde{N}_n + \alpha^2 (\Delta W_n)^2 + \eta^2 (\Delta \tilde{N}_n)^2 \right]. \quad (9.40)$$

Assume that $\tilde{N}$ is the underlying CPC for $\tilde{N}$. Recall the independence of increments in Definition 5.1 and Definition 6.3. Then we have that $E[\Delta W_n] = 0$ and $E[(\Delta W_n)^2] = \Delta t$. Also, with the help of Lemma 6.12 and Theorem 6.8 we have $E[\Delta \tilde{N}_n] = 0$ and $E[(\Delta \tilde{N}_n)^2] = \lambda E[X_0^2] \Delta t$. This combined with (9.40) gives that

$$E[g(\omega, x_{n+1}^1, x_{n+1}^2)] = E[g(\omega, x_n^1, x_n^2)] + \frac{(\alpha^2 + \eta^2 \lambda E[X_0^2]) \Delta t}{2}.$$

Through induction we are done. □

As before, the bound of the expected value of the energy of the explicit Euler scheme is much less exact.
We now apply Lemma A.1, where we have for step \( n \)
guaranteeing very large errors for large \( n \), for all times \( t_n \).

First we unpack the multiplication for \( n+1 \). We get

\[
E[g(\omega, x^1_n, x^2_n)] = \frac{1}{2}\left(\omega^2(x^1_{n+1})^2 + (x^2_{n+1})^2\right)
\]

Expanding the squares and rearranging yields

\[
E[g(\omega, x^1_{n+1}, x^2_{n+1})] = \frac{1}{2}E\left[\omega^2(x^1_{n+1})^2 + (x^2_{n+1})^2\right] + \frac{1}{2}E[\omega^2\Delta t(x^2_{n})^2 + (x^1_{n})^2] + \frac{1}{2}E[\alpha^2(\Delta W_n)^2] + \eta^2(\Delta \tilde{N}_n)^2]
\]

Now we use that \( \Delta W_n \) is independent of \( x^1_n \) and \( x^2_n \), that \( E[\Delta W_n] = 0 \) and that \( E[(\Delta W_n)^2] = \Delta t \), which follows from Definition 5.1. As before we identify the energy at \( x_n \) and get

\[
E[g(\omega, x^1_{n+1}, x^2_{n+1})] = (1 + \omega^2\Delta t^2)E[g(\omega, x^1_n, x^2_n)] + \alpha^2\Delta t + \eta^2\frac{\Delta t}{2}E[(\Delta \tilde{N}_n)^2] + \eta\Delta \tilde{N}_n x^2_n - \omega^2\Delta t \Delta \tilde{N}_n x^2_n.
\]

We get from Definition 6.3, Lemma 6.12 and Theorem 6.8 that \( E[\Delta \tilde{N}_n] = 0 \) and that \( E[(\Delta \tilde{N}_n)^2] = \lambda E[X^2_t] \Delta t \). We therefore have that

\[
E[g(\omega, x^1_{n+1}, x^2_{n+1})] = (1 + \omega^2\Delta t^2)E[g(\omega, x^1_n, x^2_n)] + \frac{\alpha^2\Delta t}{2} + \frac{\eta^2\lambda E[X^2_t]\Delta t}{2}.
\]

This yields us via induction that

\[
E[g(\omega, x^1_n, x^2_n)] \geq (1 + \omega^2\Delta t^2)E[g(\omega, x^1_0, x^2_0)] = (1 + \omega^2\Delta t^2)^n t_n \omega^2.
\]

I.e. we have for step \( n \) that

\[
E[g(\omega, x^1_n, x^2_n)] \geq \frac{\omega^2}{2}\left(1 + \frac{(\omega^2\Delta t)t_n}{n}\right)^n.
\]

Since the right hand side is strictly increasing, any lower bound for the case where \( n = 1 \) holds for every \( n \). I.e. that

\[
E[g(\omega, x^1_n, x^2_n)] \geq \frac{\omega^2}{2}\left(1 + \frac{(\omega^2\Delta t)t_n}{n}\right)^n \geq \frac{\omega^2}{2}\left(1 + (\omega^2\Delta t)t_n\right).
\]

We now apply Lemma A.1 where

\[
c = \omega^2\Delta t
\]

and we get the sought inequality, applying the restriction that \( t_n c = t_n \omega^2\Delta t \leq 2.5 \),

\[
E[g(\omega, x^1_n, x^2_n)] \geq \frac{\omega^2}{2}e^{\omega^2\Delta t t_n}. \quad \Box
\]

The corresponding upper bound for the expected energy of the implicit Euler scheme is as follows.

**Theorem 9.18.** The numerical solution \( [x^1_n, x^2_n]^T \) of the LBPO \((9.5)\) driven by a CCPoPr \( N^* = \tilde{N} \), obtained using the implicit Euler scheme \((9.8)\) with initial values \( x^1_0 = 1, x^2_0 = 0 \), satisfies

\[
E[g(\omega, x^1_n, x^2_n)] \leq \frac{\omega^2}{2} + \frac{(\alpha^2 + \eta^2\lambda E[X^2_t])t_n}{2\Delta t^2 \omega^2}
\]

for all times \( t_n = n\Delta t \).
This means that unless $\Delta t < \omega$, we have that the expected energy obtained by the implicit Euler scheme is always less than the theoretical expected energy, see Theorem 9.15.

**Proof.** First step, the inverse. We get

$$
\left( I - \begin{bmatrix} 0 & \Delta t \\ -\omega^2 & 1 \end{bmatrix} \right)^{-1} = \frac{1}{\omega^2 \Delta t^2 + 1} \begin{bmatrix} 1 & -\omega^2 \Delta t \\ -\Delta t & 1 \end{bmatrix}.
$$

Performing the matrix multiplication yields

$$
E[g(\omega, x_{n+1}^1, x_{n+1}^2)] = \frac{1}{2} E \left[ \omega^2 (x_{n+1}^1)^2 + (x_{n+1}^2)^2 \right]
= \frac{1}{2(\omega^2 \Delta t^2 + 1)^2} E \left[ \omega^2 (x_n^1 + \Delta t x_n^2 + \alpha \Delta t \Delta W_n + \eta \Delta t \Delta \tilde{N}_n)^2 + (-\omega^2 \Delta t x_n^1 + x_n^2 + \alpha \Delta W_n + \eta \Delta \tilde{N}_n)^2 \right].
$$

From Definition 5.1 we have that $E[\Delta W_n] = 0$ and $E[(\Delta W_n)^2] = \Delta t$, so expanding and rearranging while using independences yields us

$$
E[g(\omega, x_{n+1}^1, x_{n+1}^2)] = \frac{1}{2} \left( E[\omega^2 (x_n^1)^2 + (x_n^2)^2] + \omega^2 \Delta t^2 E[\omega^2 (x_n^1)^2 + (x_n^2)^2] + \alpha^2 \omega^2 \Delta t^3 + \alpha^2 \Delta t + E[\eta^2 \omega^2 \Delta t^2 (\Delta \tilde{N}_n)^2 + 2 \eta \omega^2 \Delta t \Delta \tilde{N}_n x_n^2 + \eta^2 (\Delta \tilde{N}_n)^2 + 2 \eta \Delta \tilde{N}_n x_n^2] \right).
$$

Now, Definition 6.3 with Lemma 6.12 and Theorem 6.13 gives that $E[\Delta \tilde{N}_n] = 0$ and $E[(\Delta \tilde{N}_n)^2] = \lambda E[X_1^2] \Delta t$. Then, assuming that $E[X_1^2] \geq 0$, we have

$$
E[g(\omega, x_{n+1}^1, x_{n+1}^2)] = \frac{1}{(\omega^2 \Delta t^2 + 1)^2} \left( (\omega^2 \Delta t^2 + 1) E[g(\omega, x_n^1, x_n^2)] + \frac{\omega^2 \Delta t^2 + 1}{2} (\alpha^2 \Delta t + \eta^2 E[\Delta \tilde{N}_n^2]) \right)
= \frac{1}{\omega^2 \Delta t^2 + 1} \left( E[g(\omega, x_n^1, x_n^2)] + \frac{\alpha^2 \Delta t}{2} + \frac{\eta^2}{2} \lambda E[X_1^2] \Delta t \right)
\leq E[g(\omega, x_n^1, x_n^2)] + \frac{\alpha^2 \Delta t + \eta^2 \lambda E[X_1^2] \Delta t}{2(\omega^2 \Delta t^2 + 1)}
\leq E[g(\omega, x_0^1, x_0^2)] + (n + 1) \frac{\alpha^2 \Delta t + \eta^2 \lambda E[X_1^2] \Delta t}{2(\omega^2 \Delta t^2 + 1)}
= \frac{\omega^2}{2} + \frac{(\alpha^2 + \eta^2 \lambda E[X_1^2]) \Delta t}{2(\omega^2 \Delta t^2 + 1)},
$$

which concludes the proof. \qed
10. Final words

10.1. Summary of the numerical results. Summarizing the strong and weak order convergences we have shown throughout this essay we get Table 2 to Table 6.

From those we can discern that the theta Euler scheme, with $\theta = 0.5$ have a much greater precision than the other schemes. In the deterministic settings it gives a strong convergence of order 2 for both the GBPM and the LBPO and MATLAB working precision when it comes to the weak error of the energy of the LBPO. When it comes to other settings, non-zero $\alpha$ or $\eta$, it does not display any significant difference (other than initial precision) from the other Euler schemes for the strong error. However, for the weak error for the energy of the LBPO, it gives order 2 instead of order 1, which is an improvement worthy of mentioning.

The trigonometric scheme, which is limited to the LBPO, does not display any higher order of convergence compared to the other schemes, but it does display a much higher initial precision. Also, at the deterministic setting it displays MATLAB working precision for both the strong and the weak error of the energy.

<table>
<thead>
<tr>
<th>Scheme</th>
<th>$\sigma = \eta = 0$</th>
<th>$\sigma = 0.2, \eta = 0$</th>
<th>$\sigma = 0, \eta = 0.2$</th>
<th>$\sigma = \eta = 0.2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Explicit Euler</td>
<td>1</td>
<td>1/2</td>
<td>1</td>
<td>1/2</td>
</tr>
<tr>
<td>Implicit Euler</td>
<td>1</td>
<td>1/2</td>
<td>1</td>
<td>1/2</td>
</tr>
<tr>
<td>Theta Euler, $\theta = 0.5$</td>
<td>2</td>
<td>1/2</td>
<td>1</td>
<td>1/2</td>
</tr>
<tr>
<td>SSB Euler</td>
<td>1</td>
<td>1/2</td>
<td>1</td>
<td>1/2</td>
</tr>
<tr>
<td>CSSB Euler</td>
<td>1</td>
<td>1/2</td>
<td>1</td>
<td>1/2</td>
</tr>
</tbody>
</table>

Table 2. Estimated strong order of convergence for the GBPM, Definition 8.3. Visual inspection of Figure 8.3 to Figure 8.7.

<table>
<thead>
<tr>
<th>Scheme</th>
<th>$\alpha = \eta = 0$</th>
<th>$\alpha = 0.2, \eta = 0$</th>
<th>$\alpha = 0, \eta = 0.2$</th>
<th>$\alpha = \eta = 0.2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Explicit Euler</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Implicit Euler</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Partitioned Euler</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Theta Euler, $\theta = 0.5$</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>SSB Euler</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>CSSB Euler</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Trigonometric</td>
<td>$\sim$</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 3. Estimated strong order of convergence for the LBPO, Definition 9.2, driven by a CPoPr. Visual inspection of Figure 9.5 to Figure 9.11.

10.2. Theoretical summary. Perhaps unexpectedly, the work of gathering and rephrasing the required results was insufficient to investigate the properties of the LBPO. Most of them required extensions or alternative variants (see e.g. Theorem 6.19 and Lemma 6.24).

Due to Theorem 6.19 we see that the solution of the LBPO, Theorem 9.12 is very much similar to the solution of the LBO, Theorem 9.6 ([2], Theorem 3). Note that this holds for whether the LBPO is driven by a CPoPr or a CCPoPr.

The expected energy of the LBPO displays a linear drift when it is driven by a CCPoPr, thanks to Theorem 6.23 being so similar to Theorem 5.7. However, this does not hold for the LBPO driven by a
CPOPr, as we see that Lemma 6.24 and Theorem 6.21 preserves some of the oscillating properties the trigonometric functions have.

The bounds of the expected energy of the numerical solutions for the LBO, Theorem 9.8 ([2], Theorem 2.2), Theorem 9.9 ([22], Theorem 3) and Theorem 9.10 ([22], Theorem 4) have equivalent results for the LBPO driven by a CPOPr, Theorem 9.16, Theorem 9.17 and Theorem 9.18. The precision shown in Theorem 9.8 is preserved as we look at the case for the LBPO driven by a CPOPr, shown in Theorem 9.16.

<table>
<thead>
<tr>
<th>Scheme</th>
<th>$\alpha = \eta = 0$</th>
<th>$\alpha = 0.2, \eta = 0$</th>
<th>$\alpha = 0, \eta = 0.2$</th>
<th>$\alpha = \eta = 0.2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Explicit Euler</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Implicit Euler</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Partitioned Euler</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Theta Euler, $\theta = 0.5$</td>
<td>$\sim$</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>Trigonometric</td>
<td>$\sim$</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 4. Estimated strong order of convergence for the LBPO, Definition 9.2, driven by a CPOPr. Visual inspection of Figure 9.12 to Figure 9.16.

<table>
<thead>
<tr>
<th>Scheme</th>
<th>$\alpha = \eta = 0$</th>
<th>$\alpha = 0.2, \eta = 0$</th>
<th>$\alpha = 0, \eta = 0.2$</th>
<th>$\alpha = \eta = 0.2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Explicit Euler</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Implicit Euler</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Partitioned Euler</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Theta Euler, $\theta = 0.5$</td>
<td>$\sim$</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>SSB Euler</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>CSSB Euler</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Trigonometric</td>
<td>$\sim$</td>
<td>$\sim$</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 5. Estimated weak order of convergence for the energy, (9.6), of the LBPO, Definition 9.2, driven by a CPOPr. Visual inspection of Figure D.8 to Figure D.14.

<table>
<thead>
<tr>
<th>Scheme</th>
<th>$\alpha = \eta = 0$</th>
<th>$\alpha = 0.6, \eta = 0$</th>
<th>$\alpha = 0, \eta = 0.6$</th>
<th>$\alpha = \eta = 0.6$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Explicit Euler</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Implicit Euler</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Partitioned Euler</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Theta Euler, $\theta = 0.5$</td>
<td>$\sim$</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>

Table 6. Estimated weak order of convergence for the energy, (9.6), of the LBPO, Definition 9.2, driven by a CPOPr. Visual inspection of Figure E.5 to Figure E.8.
10.3. **Further research.** As mentioned in subsection 8.4, a question which has arisen is whether the GBPM, driven only by a CPoPr, truly is of strong order $\gamma = 1$.

One of the most adjacent paths of research, related to the LSO, would be to extend the linear stochastic oscillator into more dimensions. The reader would then possibly benefit from knowing of the quantum harmonic oscillator.

Alternatively, one could add more driving Wiener or Poisson processes or allow the ones already present to affect more coordinates. The former proposition could be done by following [15] and extending Theorem 6.19. The latter is already supported by Theorem 6.19.

Another path, which could be more complicated, would be to see whether any of the theorems given in Section 9 could be extended to the linear stochastic oscillator driven by Lévy processes, as seen described in e.g. [4].

Following the increasing curve of difficulty, another path could be to see how using a Poisson process, or variant thereof, would affect stochastic partial differential equations, or SPDE’s. We have e.g. the stochastic wave equation, see e.g. [3], which is a very analogous model to the linear stochastic oscillator.
References


Appendices

A. The Lambert W Function

This section will quickly cover the most relevant material concerning the Lambert W function, here denoted as \( W \). The Lambert W function, \( W \), is a set of functions which represents the inverse of the function
\[
f(z) = ze^z,
\]
or, equivalently, a set of functions such that
\[
z = W(ze^z).
\]
An older publication, [14], states that the equation, for some constants \( a, b \in \mathbb{R} \),
\[
a^x = x + b
\]
has the solution
\[
x = -b - \frac{W(-a^{-b} \ln(a))}{\ln(a)}.
\]
A slight generalization, which can be found in e.g. [21] (though without proof), gives that the equation, for some constants \( d, a, r \in \mathbb{R} \),
\[
e^{-dx} = a(x - r)
\]
is satisfied by
\[
x = r + \frac{1}{d}W\left(\frac{de^{-dx}}{a}\right).
\]
Given this we construct the following lemma, which is used for the proof of the lower bounds for the expected energy we obtain by the explicit Euler scheme.

Lemma A.1. For \( c, x \in \mathbb{R}^+ \), such that \( cx \leq 2.5 \), the following inequality holds:
\[
1 + cx \geq e^{\frac{1}{2}cx}.
\]

Proof. Given the constant \( c \in \mathbb{R}^+ \) we begin by defining the function
\[
f(x) = 1 + cx - e^{\frac{1}{2}cx}.
\]
We note that
\[
\lim_{x \to -\infty} f(x) = -\infty
\]
and
\[
\lim_{x \to \infty} f(x) = -\infty.
\]
Deriving \( f \) yields us
\[
f'(x) = c - \frac{1}{2}ce^{\frac{1}{2}cx}
\]
setting this to zero gives
\[
c \left(1 - \frac{1}{2}e^{\frac{1}{2}cx}\right) = 0
\]
or, equivalently,
\[
2 = e^{\frac{1}{2}cx}.
\]
This means that we have one local maximum, located at
\[
x = \frac{2\ln(2)}{c}.
\]
This means that \( f \) will intersect with the \( x \)-axis twice and we trivially see that the left intersection occurs at \( x_1 = 0 \). The other is gained via
\[
f(x) = 1 + cx - e^{\frac{1}{2}cx} = 0,
\]
or
\[
e^{\frac{1}{2}cx} = 1 + cx.
\]
When we match with equation (A.1) we see that this corresponds to
\[ d = -\frac{c}{2}, \]
\[ -2d = a_0 \]
and
\[ r = \frac{1}{2d}. \]
This gives the solution
\[ x_2 = r + \frac{1}{d} \text{W} \left( \frac{de^{-dr}}{a} \right) = \frac{1}{2d} + \frac{1}{d} \text{W} \left( -\frac{1}{2\sqrt{e}} \right) = \frac{-2\text{W} \left( -\frac{1}{2\sqrt{e}} \right) - 1}{c} \approx \frac{2.513}{c}. \]
We therefore have that \( f(x) > 0 \) for \( 0 = x_1 < x < x_2 \). This means that, taking a numerical value well below the exact value,
\[ 1 + x \geq e^{\frac{2.5}{c}}x, \quad x < \frac{2.5}{c} < x_2, \]
from which the sought inequality follows. \( \square \)
B. Strong error estimations of the linear stochastic oscillator - Driven by a CPoPr

This appendix section contains plots illustrating estimations of the strong error of the LBPO driven by a CPoPr over the time interval $[0, 1]$. These plots have been produced by Code 26, which use the schemes given in Section 9 (seen implemented in Appendix I).

The reader is encouraged to note that the error observed for the trigonometric scheme in the deterministic setting, Figure B.7 and $\alpha = \eta = 0$, is so small that it is not visible on the same axis as the rest. This is supported by the error seen in Figure 9.11, which is at the MATLAB working precision.

Figure B.1. The strong error of the LBPO over $t \in [0, 1]$ obtained using the explicit Euler scheme.
Figure B.2. The strong error of the LBPO over $t \in [0, 1]$ obtained using the implicit Euler scheme.

Figure B.3. The strong error of the LBPO over $t \in [0, 1]$ obtained using the partitioned Euler scheme.
Figure B.4. The strong error of the LBPO over $t \in [0, 1]$ obtained using the theta Euler scheme, $\theta = 0.5$.

Figure B.5. The strong error of the LBPO over $t \in [0, 1]$ obtained using the SSB Euler scheme.
Figure B.6. The strong error of the LBPO over $t \in [0, 1]$ obtained using the CSSB Euler scheme.

Parameters: $\omega=3$, $\lambda=3$. Batch size: 64. Number of batches: 8.

Figure B.7. The strong error of the LBPO over $t \in [0, 1]$ obtained using the trigonometric scheme.
C. Strong error estimations of the linear stochastic oscillator - Driven by a CCPoPr

This appendix section contains plots illustrating estimations of the strong error, of the LBPO driven by a CCPoPr over the time interval \([0, 1]\). These plots have been produced by Code 27 which use the schemes given in Section 9 (seen implemented in Appendix 1).

The reader is again encouraged to note that the error observed for the trigonometric scheme in the deterministic setting, Figure C.3 and \(\alpha = \eta = 0\), is so small that it is not visible on the same axis as the rest. This is supported by the error seen in Figure 9.16 which is at the MATLAB working precision.

**Figure C.1.** The strong error of the LBPO over \(t \in [0, 1]\) obtained using the explicit Euler scheme.
Figure C.2. The strong error of the LBPO over $t \in [0, 1]$ obtained using the implicit Euler scheme.

Figure C.3. The strong error of the LBPO over $t \in [0, 1]$ obtained using the partitioned Euler scheme.
\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure_c_4}
\caption{The strong error of the LBPO over $t \in [0, 1]$ obtained using the theta Euler scheme, $\theta = 0.5$.}
\end{figure}

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure_c_5}
\caption{The strong error of the LBPO over $t \in [0, 1]$ obtained using the trigonometric scheme.}
\end{figure}
D. Energy weak error estimations of the linear stochastic oscillator - Driven by a CPoPr

This appendix section contains plots illustrating estimations of the weak error of the energy, of the LBPO driven by a CPoPr over the time interval $t \in [0, 0.25]$ and at the end time, $T = 0.25$. These plots have been produced by Code 29, Code 30, and Code 31 which use the schemes given in Section 9 (seen implemented in Appendix I).

The reader is encouraged to note that the error observed for the theta Euler scheme, with $\theta = 0.5$, and the trigonometric scheme in the deterministic setting, Figure D.4 and Figure D.7 with $\alpha = \eta = 0$, is so small that it is not visible on the same axis as the rest. This is supported by the error seen in Figure D.11 and Figure D.14 which is at the MATLAB working precision.

Another error which is noteworthy is the error observed for the trigonometric scheme where $\alpha = 0.6$ and $\eta = 0$. This error is also not visible in Figure D.7 though the error is not due to the MATLAB working precision but due to the Monte Carlo error. We see that the number of trials $n$ is close to $4 \cdot 10^9$ and this yields that the standard error of the estimation will land within the range of $1/\sqrt{n} \approx 10^{-5}$, which seems to be larger than the observed weak error. This is supported by the error seen in Figure D.14.

Figure D.1. The weak error of the energy of the LBPO, driven by a CPoPr, over $t \in [0, 0.25]$, obtained using the explicit Euler scheme.
Figure D.2. The weak error of the energy of the LBPO, driven by a CPoPr, over $t \in [0, 0.25]$, obtained using the implicit Euler scheme.

Figure D.3. The weak error of the energy of the LBPO, driven by a CPoPr, over $t \in [0, 0.25]$, obtained using the partitioned Euler scheme.
Figure D.4. The weak error of the energy of the LBPO, driven by a CPoPr, over $t \in [0, 0.25]$, obtained using the theta Euler scheme, $\theta = 0.5$.

Figure D.5. The weak error of the energy of the LBPO, driven by a CPoPr, over $t \in [0, 0.25]$, obtained using the SSB Euler scheme.
The weak error of the energy of the LBPO, driven by a CPoPr, over $t \in [0, 0.25]$, obtained using the CSSB Euler scheme.

**Figure D.6.**

The weak error of the energy of the LBPO, driven by a CPoPr, over $t \in [0, 0.25]$, obtained using the trigonometric scheme.

**Figure D.7.**
Figure D.8. The weak error of the energy of the LBPO, driven by a CPoPr, at $T = 0.25$, obtained using the explicit Euler scheme. $C_i \in \mathbb{R}$, $i = 1, 2, 3$ are constants used to translate the support lines.

Figure D.9. The weak error of the energy of the LBPO, driven by a CPoPr, at $T = 0.25$, obtained using the implicit Euler scheme. $C_i \in \mathbb{R}$, $i = 1, 2, 3$ are constants used to translate the support lines.
Figure D.10. The weak error of the energy of the LBPO, driven by a CPoPr, at $T = 0.25$, obtained using the partitioned Euler scheme. $C_i \in \mathbb{R}, i = 1, 2, 3$ are constants used to translate the support lines.

Figure D.11. The weak error of the energy of the LBPO, driven by a CPoPr, at $T = 0.25$, obtained using the theta Euler scheme, $\theta = 0.5$. $C_i \in \mathbb{R}, i = 1, 2, 3$ are constants used to translate the support lines.
Figures D.12. The weak error of the energy of the LBPO, driven by a CPoPr, at $T = 0.25$, obtained using the SSB Euler scheme. $C_i \in \mathbb{R}$, $i = 1, 2, 3$ are constants used to translate the support lines.

Figures D.13. The weak error of the energy of the LBPO, driven by a CPoPr, at $T = 0.25$, obtained using the CSSB Euler scheme. $C_i \in \mathbb{R}$, $i = 1, 2, 3$ are constants used to translate the support lines.
Figure D.14. The weak error of the energy of the LBPO, driven by a CPoPr, at $T = 0.25$, obtained using the trigonometric scheme. $C_i \in \mathbb{R}, i = 1, 2, 3$ are constants used to translate the support lines.
E. Energy weak error estimations of the linear stochastic oscillator - Driven by a CCPoPr

This appendix section contains plots illustrating estimations of the weak error of the energy, of the LBPO driven by a CCPoPr over the time interval $t \in [0, 0.25]$ and at the end time, $T = 0.25$. These plots have been produced by Code 29, Code 30, and Code 31 which use the schemes given in Section 9 (seen implemented in Appendix I).

**Figure E.1.** The weak error of the energy of the LBPO, driven by a CCPoPr, over $t \in [0, 0.25]$, obtained using the explicit Euler scheme.
The weak error of the energy of the LBPO, driven by a CCPoPr, over $t \in [0, 0.25]$, obtained using the implicit Euler scheme.


The weak error of the energy of the LBPO, driven by a CCPoPr, over $t \in [0, 0.25]$, obtained using the partitioned Euler scheme.

The weak error of the energy of the LBPO, driven by a CCPoPr, over $t \in [0, 0.25]$, obtained using the theta Euler scheme, $\theta = 0.5$.

\begin{figure}
\centering
\includegraphics[width=\textwidth]{figure_e4.pdf}
\caption{The weak error of the energy of the LBPO, driven by a CCPoPr, over $t \in [0, 0.25]$, obtained using the theta Euler scheme, $\theta = 0.5$.}
\end{figure}

The weak error of the energy of the LBPO, driven by a CCPoPr, at $T = 0.25$, obtained using the explicit Euler scheme. $C_i \in \mathbb{R}$, $i = 1, 2, 3$ are constants used to translate the support lines.

\begin{figure}
\centering
\includegraphics[width=\textwidth]{figure_e5.pdf}
\caption{The weak error of the energy of the LBPO, driven by a CCPoPr, at $T = 0.25$, obtained using the explicit Euler scheme. $C_i \in \mathbb{R}$, $i = 1, 2, 3$ are constants used to translate the support lines.}
\end{figure}
Figure E.6. The weak error of the energy of the LBPO, driven by a CCPoPr, at $T = 0.25$, obtained using the implicit Euler scheme. $C_i \in \mathbb{R}$, $i = 1, 2, 3$ are constants used to translate the support lines.

Figure E.7. The weak error of the energy of the LBPO, driven by a CCPoPr, at $T = 0.25$, obtained using the partitioned Euler scheme. $C_i \in \mathbb{R}$, $i = 1, 2, 3$ are constants used to translate the support lines.
Figure E.8. The weak error of the energy of the LBPO, driven by a CCPoPr, at $T = 0.25$, obtained using the theta Euler scheme, $\theta = 0.5$. $C_i \in \mathbb{R}$, $i = 1, 2, 3$ are constants used to translate the support lines.
F. Code - Process batch simulators

Code 1. WienerProc.m

```matlab
function [W] = WienerProc(T,dt,M,d)
% WienerProc(T,dt,M,d)
% Input:
% T - End time
% dt - Time steps
% M - Number of processes
% d - The dimension number denoting the processes
% Output:
% W - A batch containing M d-dimensional Wiener processes spanning the
% time interval [0,T] with time steps dt

% Calculate number of time steps
n = T/dt;
% Add an initial matrix of zeroes to the process and construct the process
W = cat(2,zeros(M,1,d), cumsum(randn(M,n,d)*sqrt(dt),2));
end
```

Code 2. PoissonProc.m

```matlab
function [NRet] = PoissonProc(T,dt,M,lambda,dist,distExp)
% PoissonProc(T,dt,M,lambda,dist,distExp)
% Input:
% T - End time
% dt - Time steps
% M - Number of processes
% lambda - Scalar describing increment intensity
% dist - A function which gives x increments
% distExp - The expected value of the increment given by dist
% Output:
% NRet - A struct containing the Poisson process, the compound Poisson process, the compensated compound Poisson process and the increments

t = 0:dt:T;
tlength = length(t);
% Allocate memory
N = zeros(M,tlength);
CompN = zeros(M,tlength);
CompCompN = zeros(M,tlength);
dCompN = zeros(M,1);
% Poisson increment storage.
NInc = cell(M,tlength);
for i = 1:(tlength-1)
% Simulate number of Poisson increments
dN = poissrnd(lambda*dt,[M,1]);
% Allocate increment memory
dNInc = cell(M,1);
% The odds of multiple increments occurring during one time
% step decreases in likelihood as dt decreases. Deal with the
% case where dN == (0 | 1) first
oneInc = (dN == 1);
nOneInc = sum(oneInc);
dNInc(oneInc) = num2cell(dist(nOneInc));
% In case we have more than one increments in one time step,
% locate which indexes there are
indMoreInc = find(dN > 1);
NMoreInc = length(indMoreInc);
% Loop over each case where dN > 1
for m = 1:NMoreInc
```

97
% Retrieve index
currentIndex = indMoreInc(m);

% Store the simulated variables
dNInc{currentIndex} = dist(dN(currentIndex));

% Construct the compound Poisson process increment This step
could be more efficient, perhaps through storing the cell
% arrays as a sparse matrix
for m = 1:M
    % Store increments
    NInc{m} = [NInc{m}; dNInc{m}];
    % Add increments to compound Poisson process
    dCompN{m} = sum(dNInc{m});
end

% Construct the compensated compound Poisson increment
dCompCompN = dCompN - lambda*distExp*dt;

% With the steps simulated, add them to the processes
N(:,i+1) = N(:,i) + dN;
CompN(:,i+1) = CompN(:,i) + dCompN;
CompCompN(:,i+1) = CompCompN(:,i) + dCompCompN;

NRet.Poisson = N;
NRet.Compound = CompN;
NRet.Compensated = CompCompN;
NRet.increments = NInc;
G. Code - General step functions

Code 3. ExplicitEulerStepper.m

```matlab
function [Xnext] = ExplicitEulerStepper(Xn,dW,dN,dt,t,mu,sigma,eta)
% Assuming the differential equation is of the form
% dX = mu(t,X) * dx + sigma(t,X) * dW + eta(t,X) * dN
% where mu, sigma, eta: (R+)x(R^d) -> R^d are functions
% Input:
% Xn - Previous X-value
% dW - An increment for the standard Wiener process
% dN - An increment for the compound Poisson process
% dt - The time increment
% t - The current time
Xnext = Xn + mu(t,Xn)*dt + sigma(t,Xn).*dW + eta(t,Xn).*dN;
end
```

Code 4. ImplicitEulerStepper.m

```matlab
function [Xnext] = ImplicitEulerStepper(Xn,dW,dN,dt,t,mu,sigma,eta)
% Assuming the differential equation is of the form
% dX = mu(t,X) * dx + sigma(t,X) * dW + eta(t,X) * dN
% where mu, sigma, eta: (R+)x(R^d) -> R^d are functions
% Input:
% Xn - Previous X-value
% dW - An increment for the standard Wiener process
% dN - An increment for the compound Poisson process
% dt - The time increment
% t - The current time
% NOTE: Xn, dW and dN are of size (M)x(1)x(d) where d is the dimension of
% the process and M is the batch size

% Extract the batch size
M = size(Xn,1);
% Initialize the memory
Xnext = Xn;
% Construct the options to not display the outcome of the implicit solving
options = optimoptions('fsolve','Display','none');
% Loop over the batches for one multiplication per element
for i=1:M
Xnext(i,1,:) = fsolve(@(X) X - ...
    (Xn(i,1,:) + ...
        mu(t+dt,X)*dt + ...
        sigma(t,Xn(i,1,:))*dW(i,1,:) + ...
        eta(t,Xn(i,1,:))*dN(i,1,:))..., ...
    Xn(i,1,:),options);
end
end
```
CODE 5. MilsteinStepper.m

```
function [Xnext] = MilsteinStepper(Xn,dW,dt,t,mu,sigma,sigmap)
% Assuming the differential equation is of the form
% dX = mu(t,X)*dx + sigma(t,X)*dW + eta(t,X)*dN
% where mu, sigma, eta: (R+)x(Rˆd)->Rˆd are functions
% Input:
% Xn - Previous X-value
% dW - An increment for the standard Wiener process
% dN - An increment for the compound Poisson process
% dt - The time increment
% t - The current time
% sigmap - The derivative of sigma
Xnext = Xn + mu(t,Xn)*dt + ...
    sigma(t,Xn).*dW + ...
    1/2*sigma(t,Xn).*sigmap(t,Xn).*{dW.^2-dt};
end
```

CODE 6. ThetaEulerStepper.m

```
function [Xnext] = ThetaEulerStepper(Xn,dW,dN,dt,t,mu,sigma,eta,theta)
% Assuming the differential equation is of the form
% dX = mu(t,X) *dx + sigma(t,X)*dW + eta(t,X)*dN
% where mu, sigma, eta: (R+)x(Rˆd)->Rˆd are functions
% Input:
% Xn - Previous X-value
% dW - An increment for the standard Wiener process
% dN - An increment for the compound Poisson process
% dt - The time increment
% t - The current time
% theta - The scalar constant controlling the weighing
% NOTE: Xn, dW and dN are of size (M)x(1)x(d) where d is the dimension of
% the process and M is the batch size
% Extract the batch size
M = size(Xn,1);
% Initialize the memory
Xnext = Xn;
% Construct the options to not display the outcome of the implicit solving
options = optimoptions('fsolve','Display','none');
% Loop over the batches for one multiplication per element
for i=1:M
    Xnext(i,1,:) = fsolve(@(X) X - ...
        (Xn(i,1,:) + ...
            theta*mu(t,Xn(i,1,:))*dt + ...
            (1-theta)*mu(t+dt,X)*dt + ...
            sigma(t,Xn(i,1,:))*dW(i,1,:) + ...
            eta(t,Xn(i,1,:))*dN(i,1,:)));
end
end
```
function [Xnext] = SSBEulerStepper(Xn,dW,dN,dt,t,mu,sigma,eta)
% Assuming the differential equation is of the form
% dX = mu(t,X)*dx + sigma(t,X)*dW + eta(t,X)*dN
% where mu, sigma, eta: (R+)^x(R^d)--R^d are functions
% Input:
% Xn - Previous X-value
% dW - An increment for the standard Wiener process
% dN - An increment for the compound Poisson process
% dt - The time increment
% t - The current time
% NOTE: Xn, dW and dN are of size (M)x(1)x(d) where d is the dimension of
% the process and M is the batch size
% Extract the batch size
M = size(Xn,1);
% Initialize the memory
Xnext = Xn;
% Construct the options to not display the outcome of the implicit solving
options = optimoptions('fsolve','Display','none');
% Loop over the batches for one multiplication per element
for i=1:M
  Xtemp = fsolve(@(X) X - (Xn(i,1,:) + mu(t,X)*dt), options);
  Xnext(i,1,:) = Xtemp + ...
                sigma(t,Xtemp)*dW(i,1,:) + ...
                eta(t,Xtemp)*dN(i,1,:);
end
end

function [Xnext] = CSSBEulerStepper(Xn,dW,dN,dt,t,mu,sigma,eta,drift)
% Assuming the differential equation is of the form
% dX = mu(t,X)*dx + sigma(t,X)*dW + eta(t,X)*dN
% where mu, sigma, eta: (R+)^x(R^d)--R^d are functions
% Input:
% Xn - Previous X-value
% dW - An increment for the standard Wiener process
% dN - An increment for the compound Poisson process
% dt - The time increment
% t - The current time
% drift - The expected value of the drift introduced by the compound
% Poisson process
% NOTE: Xn, dW and dN are of size (M)x(1)x(d) where d is the dimension of
% the process and M is the batch size
% Extract the batch size
M = size(Xn,1);
% Initialize the memory
Xnext = Xn;
% Construct the options to not display the outcome of the implicit solving
options = optimoptions('fsolve','Display','none');
% Loop over the batches for one multiplication per element
for i=1:M
  Xtemp = fsolve(@(X) X - ...
               (Xn(i,1,:) + (mu(t,X) + drift*eta(t,X))*dt), options);
  Xnext(i,1,:) = Xtemp + ...
                 sigma(t,Xtemp)*dW(i,1,:) + ...
                 eta(t,Xtemp)*(dN(i,1,:)-drift*dt);
end
end

101
H. Code - Geometric differential equation step functions, as used in the GBPM

**Code 9. ImplicitEulerStepperGeom.m.**

```matlab
function [Xnext] = ImplicitEulerStepperGeom(Xn,dW,dN,dt,mu,sigma,eta)
% Assuming the differential equation is of the form
% dX = mu*X*dx + sigma*X*dW + eta*X*dN
% where mu, sigma and eta are constants
% Input:
% Xn - Previous X-value
% dW - An increment for the standard Wiener process
% dN - An increment for the compound Poisson process
% dt - The time increment
% NOTE: Xn, dW and dN vectors of length M, where M is the batch size
% Calculate
Xnext = (1+sigma*dW+eta*dN).*Xn/(1-mu*dt);
end
```

**Code 10. ThetaEulerStepperGeom.m.**

```matlab
function [Xnext] = ThetaEulerStepperGeom(Xn,dW,dN,dt,mu,sigma,eta,theta)
% Assuming the differential equation is of the form
% dX = mu*X*dx + sigma*X*dW + eta*X*dN
% where mu, sigma and eta are constants
% Input:
% Xn - Previous X-value
% dW - An increment for the standard Wiener process
% dN - An increment for the compound Poisson process
% dt - The time increment
% theta - The scalar constant controlling the weighing
% NOTE: Xn, dW and dN vectors of length M, where M is the batch size
% Calculate
Xnext = (1+theta*mu*dt+sigma*dW+eta*dN)/(1-(1-theta)*mu*dt).*Xn;
end
```
function \[X_{\text{next}}\] = SSBEulerStepperGeom(Xn,dW,dN,dt,mu,sigma,eta)
% Assuming the differential equation is of the form
% \[dX = \mu X dx + \sigma X dW + \eta X dN\]
% where \(\mu\), \(\sigma\) and \(\eta\) are constants
% Input:
% \(Xn\) - Previous X-value
% \(dW\) - An increment for the standard Wiener process
% \(dN\) - An increment for the compound Poisson process
% \(dt\) - The time increment
% NOTE: \(Xn\), \(dW\) and \(dN\) vectors of length \(M\), where \(M\) is the batch size

M = size(Xn,1);
A = sparse(M,M);
% Note: Excluding the middle step results in it being equivalent to the
% implicit Euler scheme
% Set the diagonal
A(1:(M+1):end) = (1+sigma*dW+eta*dN)/(1-mu*dt);
% Calculate
Xnext = A*Xn;
end

function \[X_{\text{next}}\] = CSSBEulerStepperGeom(Xn,dW,dN,dt,mu,sigma,eta,drift)
% Assuming the differential equation is of the form
% \[dX = \mu X dx + \sigma X dW + \eta X dN\]
% where \(\mu\), \(\sigma\) and \(\eta\) are constants
% Input:
% \(Xn\) - Previous X-value
% \(dW\) - An increment for the standard Wiener process
% \(dN\) - An increment for the compound Poisson process
% \(dt\) - The time increment
% drift - The expected value of the drift introduced by the compound
% Poisson process
% NOTE: \(Xn\), \(dW\) and \(dN\) vectors of length \(M\), where \(M\) is the batch size

M = size(Xn,1);
A = sparse(M,M);
% Note: Excluding the middle step results in it being equivalent to the
% implicit Euler scheme
% Set the diagonal
Xtemp = Xn/(1-(mu+drift*eta)*dt);
A(1:(M+1):end) = 1+sigma*dW+eta*(dN-drift*dt);
% Calculate
Xnext = A*Xtemp;
end
I. Code - Matrix differential equation step functions, as used in the LBPO

**Code 13. ExplicitEulerStepperMatrix.m**

```matlab
function [Xnext] = ExplicitEulerStepperMatrix(Xn,dW,dN,dt,fmat,alphaVec,etaVec)
% Assuming the differential equation is of the form
dX = fmat*dx + alphaVec*dW + etaVec*dN
% where fmat is a matrix of size (d)x(d) and alphaVec, etaVec are vectors
% of size (d)
% Input:
% Xn - Previous X-value
% dW - An increment for the standard Wiener process
% dN - An increment for the compound Poisson process
% dt - The time increment
% NOTE: Xn a vector of length M*d where d is the dimension of the process
% and M is the batch size. dW and dN are vectors of length M.

% Fetch size and dimension
d = length(alphaVec);
M = length(Xn)/d;
% Create matrices and vectors
oneMat = eye(d);
A = (oneMat+fmat*dt);
B = zeros(M*d,1);
B(1:2:(end-1)) = alphaVec(1)*dW;
B(2:2:end) = alphaVec(2)*dW;
C = zeros(M*d,1);
C(1:2:(end-1)) = etaVec(1)*dN;
C(2:2:end) = etaVec(2)*dN;
% Initialize cell memory in preparation of sparse matrix
ADiag = cell(M,1);
% Fill the cell memory
ADiag(:) = {sparse(A)};
% Create a sparse matrix with a block diagonal consisting of A
ASparse = blkdiag(ADiag{:});
Xnext = ASparse*Xn + B + C;
end
```

**Code 14. ImplicitEulerStepperMatrix.m**

```matlab
function [Xnext] = ImplicitEulerStepperMatrix(Xn,dW,dN,dt,fmat,alphaVec,etaVec)
% Assuming the differential equation is of the form
dX = fmat*dx + alphaVec*dW + etaVec*dN
% where fmat is a matrix of size (d)x(d) and alphaVec, etaVec are vectors
% of size (d)
% Input:
% Xn - Previous X-value
% dW - An increment for the standard Wiener process
% dN - An increment for the compound Poisson process
% dt - The time increment
% NOTE: Xn a vector of length M*d where d is the dimension of the process
% and M is the batch size. dW and dN are vectors of length M.

% Fetch size and dimension
d = length(alphaVec);
M = length(Xn)/d;
% Create matrices and vectors
oneMat = eye(d);
A = (oneMat-fmat*dt);
B = zeros(M*d,1);
B(1:2:(end-1)) = alphaVec(1)*dW;
20 C = zeros(M*d,1);
C(1:2:(end-1)) = etaVec(1)*dN;
C(2:2:end) = etaVec(2)*dN;
% Initialize cell memory in preparation of sparse matrix
A % Fetch size and dimension
14 d = length(alphaVec);
15 M = length(Xn)/d;
16 % Create matrices and vectors
17 oneMat = eye(d);
18 A = (oneMat-fmat*dt);
19 B = zeros(M*d,1);
20 B(1:2:(end-1)) = alphaVec(1)*dW;
```
function [Xnext] = ThetaEulerStepperMatrix(Xn,dW,dN,dt,fmat,alphaVec,etaVec,theta)

% Assuming the differential equation is of the form
% dX = fmat *dx + alphaVec*dW + etaVec*dN
% where fmat is a matrix of size (d)x(d) and alphaVec, etaVec are vectors
% of size (d)
% Input:
% Xn - Previous X-value
% dW - An increment for the standard Wiener process
% dN - An increment for the compound Poisson process
% dt - The time increment
% NOTE: Xn a vector of length M*d where d is the dimension of the process
% and M is the batch size. dW and dN are vectors of length M.
% Fetch size and dimension
d = length(alphaVec);
M = length(Xn)/d;
% Create matrices and vectors
oneMat = eye(d);
A1 = (oneMat + theta*fmat*dt);
A2 = (oneMat - (1-theta)*fmat*dt);
B = zeros(M,d,1);
B(1:2:(end-1)) = alphaVec(1)*dW;
B(2:2:end) = alphaVec(2)*dW;
C = zeros(M,d,1);
C(1:2:(end-1)) = etaVec(1)*dN;
C(2:2:end) = etaVec(2)*dN;
% Initialize cell memory in preparation of sparse matrix
A1Diag = cell(M,1);
A2Diag = cell(M,1);
% Fill the cell memory
A1Diag(:) = {sparse(A1)};
A2Diag(:) = {sparse(A2)};
% Create a sparse matrix with a block diagonal consisting of A
A1Sparse = blkdiag(A1Diag{:});
A2Sparse = blkdiag(A2Diag{:});
Xnext = A2Sparse\(A1Sparse*Xn + B + C);
end
function [Xnext] = PartitionedEulerStepper(Xn, dW, dN, dt, fmat1, fmat2, alphaVec, etaVec)
% Assuming the differential equation is of the form
% dX = fmat*dx + alphaVec*dW + etaVec*dN
% where fmat = fmat1 + fmat2 is a matrix of size (d)x(d) and alphaVec,
% etaVec are vectors of size (d)
% Input:
% Xn - Previous X-value
% dW - An increment for the standard Wiener process
% dN - An increment for the compound Poisson process
% dt - The time increment
% NOTE: Xn a vector of length M*d where d is the dimension of the process
% and M is the batch size. dW and dN are vectors of length M.
% Fetch size and dimension
d = length(alphaVec);
M = length(Xn)/d;
% Create matrices and vectors
oneMat = eye(d);
A1 = (oneMat+fmat2*dt);
A2 = (oneMat-fmat1*dt);
B = zeros(M*d,1);
B(1:2:(end-1)) = alphaVec(1)*dW;
B(2:2:end) = alphaVec(2)*dW;
C = zeros(M*d,1);
C(1:2:(end-1)) = etaVec(1)*dN;
C(2:2:end) = etaVec(2)*dN;
% Initialize cell memory in preparation of sparse matrix
A1Diag = cell(M,1);
A2Diag = cell(M,1);
% Fill the cell memory
A1Diag(:) = {sparse(A1)};
A2Diag(:) = {sparse(A2)};
% Create a sparse matrix with a block diagonal consisting of A
A1Sparse = blkdiag(A1Diag{:});
A2Sparse = blkdiag(A2Diag{:});
Xnext = A2Sparse\(A1Sparse*Xn + B + C);
end

function [Xnext] = SSBEulerStepperMatrix(Xn, dW, dN, dt, fmat, alphaVec, etaVec)
% Assuming the differential equation is of the form
% dX = fmat*dx + alphaVec*dW + etaVec*dN
% where fmat is a matrix of size (d)x(d) and alphaVec, etaVec are vectors
% of size (d)
% Input:
% Xn - Previous X-value
% dW - An increment for the standard Wiener process
% dN - An increment for the compound Poisson process
% dt - The time increment
% NOTE: Xn a vector of length M*d where d is the dimension of the process
% and M is the batch size. dW and dN are vectors of length M.
% Fetch size and dimension
d = length(alphaVec);
M = length(Xn)/d;
% Create matrices and vectors
oneMat = eye(d);
A1 = (oneMat+fmat*dt);
A2 = (oneMat-fmat*dt);
B = zeros(M*d,1);
C = zeros(M*d,1);
% Initialize cell memory in preparation of sparse matrix
A1Diag = cell(M,1);
A2Diag = cell(M,1);
% Fill the cell memory
A1Diag(:) = {sparse(A1)};
A2Diag(:) = {sparse(A2)};
% Create a sparse matrix with a block diagonal consisting of A
A1Sparse = blkdiag(A1Diag{:});
A2Sparse = blkdiag(A2Diag{:});
Xnext = A2Sparse\(A1Sparse*Xn + B + C);
end
21  \quad B(1:2:\text{(end-1)}) = \text{alphaVec}(1)\times dW;
22  \quad B(2:2:end) = \text{alphaVec}(2)\times dW;
23  \quad C = \text{zeros}(M\times d,1);
24  \quad C(1:2:\text{(end-1)}) = \text{etaVec}(1)\times dN;
25  \quad C(2:2:end) = \text{etaVec}(2)\times dN;
26  \quad \% \text{Initialize cell memory in preparation of sparse matrix}
27  \quad \text{ADiag} = \text{cell}(M,1);
28  \quad \% \text{Fill the cell memory}
29  \quad \text{ADiag}(:) = \{\text{sparse}(A)\};
30  \quad \% \text{Create a sparse matrix with a block diagonal consisting of A}
31  \quad \text{ASparse} = \text{blkdiag}(\text{ADiag}(:));
32  \quad Xnext = \text{ASparse}\backslash(Xn + B + C);
33  \text{end}

\text{Code 18. CSSBEulerStepperMatrix.m.}

1  \quad \text{function } [\text{Xnext}] = \text{CSSBEulerStepperMatrix}(Xn,dW,dN,dt,fmat,alphaVec,etaVec,drift)
2  \quad \% \text{Assuming the differential equation is of the form}
3  \quad \% \quad dX = fmat \times dx + \text{alphaVec} \times dW + \text{etaVec} \times dN
4  \quad \% \text{where fmat is a matrix of size } (d)\times(d) \text{ and alphaVec, etaVec are vectors}
5  \quad \% \text{of size } (d)
6  \quad \% \text{Input:}
7  \quad \% \quad Xn \quad - \text{Previous X-value}
8  \quad \% \quad dW \quad - \text{An increment for the standard Wiener process}
9  \quad \% \quad dN \quad - \text{An increment for the compound Poisson process}
10  \quad \% \quad dt \quad - \text{The time increment}
11  \quad \% \quad \text{drift} \quad - \text{The expected value of the drift introduced by the compound}
12  \quad \% \quad \text{Poisson process}
13  \quad \% \text{NOTE: Xn a vector of length } M\times d \text{ where } d \text{ is the dimension of the process}
14  \quad \% \text{and } M \text{ is the batch size. dW and dN are vectors of length } M.
15  \quad \% \text{Fetch size and dimension}
16  \quad d = \text{length} (alphaVec);
17  \quad M = \text{length} (Xn)/d;
19  \quad \% \text{Create matrices and vectors}
20  \quad \text{oneMat} = \text{eye}(d);
21  \quad \text{A} = (\text{oneMat} - \text{fmat} \times dt);
22  \quad \text{B} = \text{zeros}(M\times d,1);
23  \quad B(1:2:\text{(end-1)}) = \text{alphaVec}(1)\times dW;
24  \quad B(2:2:end) = \text{alphaVec}(2)\times dW;
25  \quad C = \text{zeros}(M\times d,1);
26  \quad C(1:2:\text{(end-1)}) = \text{etaVec}(1)\times (dN - \text{drift} \times dt);
27  \quad C(2:2:end) = \text{etaVec}(2)\times (dN - \text{drift} \times dt);
28  \quad \% \text{Initialize cell memory in preparation of sparse matrix}
29  \quad \text{ADiag} = \text{cell}(M,1);
30  \quad \% \text{Fill the cell memory}
31  \quad \text{ADiag}(:) = \{\text{sparse}(A)\};
32  \quad \% \text{Create a sparse matrix with a block diagonal consisting of A}
33  \quad \text{ASparse} = \text{blkdiag}(\text{ADiag}(:));
34  \quad Xnext = \text{ASparse}\backslash(Xn + \text{drift} \times \text{repmat} (\text{etaVec},M,1)\times dt) + B + C;
36  \text{end}
function [Xnext] = TrigStepper(Xn,dW,dN,dt,alpha,eta,omega)
% Assuming the differential equation is of the form
% X'' + omegaˆ2*X = alpha*W' + eta*N'
% where alpha, eta and omega are scalar constants
% Input:
% Xn - Previous X-value
% dW - An increment for the standard Wiener process
% dN - An increment for the compound Poisson process
% dt - The time increment
% NOTE: Xn a vector of length M*d where d is the dimension of the process
% and M is the batch size. dW and dN are vectors of length M.

d = 2;
M = length(Xn)/d;

% Fetch size and dimension (dimension fixed for this scheme)
A = [cos(omega*dt),omegaˆ-1*sin(omega*dt);-omega*sin(omega*dt),cos(omega*dt)];
trigVec = [omegaˆ-1*sin(omega*dt) ; cos(omega*dt)];

B = zeros(M*d,1);
B(1:2:((end-1))) = alpha*trigVec(1)*dW;
B(2:2:end) = alpha*trigVec(2)*dW;
C = zeros(M*d,1);
C(1:2:((end-1))) = eta*trigVec(1)*dN;
C(2:2:end) = eta*trigVec(2)*dN;

% Initialize cell memory in preparation of sparse matrix
ADiag = cell(M,1);
ADiag(:) = {sparse(A)};

ADiag(:,1) = {sparse(A)};

% Create a sparse matrix with a block diagonal consisting of A
ASparse = blkdiag(ADiag{1});

Xnext = ASparse*Xn + B + C;
end
rng(101)

% Variable controlling font size
largeFont = 20;
normalFont = 16;

% Set constants
T = 1;
dt = 2^-10;
M = 1;
d = 1;
t = 0:dt:T;

f = figure('Position', [100, 100, 1049, 895]);
procs = WienerProc(T, dt, M, d);

plot(t,procs(1,:,1))
axis([0 T 0 1])

set(gca,'XTick',[0 1],'fontsize',largeFont)
xlabel('t','fontsize',largeFont)
ylabel('$W_t$','rot',0,'fontsize',largeFont,'interpreter','latex')
% Translate the position of the ylabel slightly
ylabh = get(gca,'YLabel');
set(ylabh,'Position',get(ylabh,'Position') - [0.02 0 0]);

% Now that we've constructed the processes, we want to mark out a step for illustration
hold on
lineType = 'ro-';
distance = 80;
nInitPos = 450;
earlierPos = nInitPos-2*distance;
endPos = nInitPos+distance;

% Vertical lines
plot([t(earlierPos) t(earlierPos)],[procs(1,earlierPos,1) 0],lineType)
plot([t(endPos) t(endPos)],[procs(1,endPos,1) 0],lineType)

% Horizontal lines
plot([t(1) t(earlierPos)],[procs(1,earlierPos,1) procs(1,earlierPos,1)],lineType)
plot([t(1) t(endPos)],[procs(1,endPos,1) procs(1,endPos,1)],lineType)

% Mark out the distance
% Vertical line
plot([t(earlierPos) t(earlierPos)],[procs(1,earlierPos,1) procs(1,endPos,1)],'k')

% Horizontal line
plot([t(earlierPos) t(earlierPos)],[procs(1,earlierPos,1) 0.5 procs(1,endPos,1)-0.5],'k')

% Give it a label

% pause(0.2)
printToPDF(f,'Figs/WienerDW')

rng(101)
60  \begin{align*}  
& T = 1; \\
& dt = 2^{-10}; \\
& M = 1; \\
& t = 0:dt:T; \\
& dist = @(x) 1; \\
& \text{expectedValue} = 1; 
\end{align*}

68  f = figure('Position', [100, 100, 1049, 895]);  
69  procs = PoissonProc(T,dt,M,12,dist,expectedValue);  
70  simpleProc = procs.Poisson(1,:,1);  
71  plot(t,simpleProc)  
72  set(gca,'XTick',[0 T],'fontsize',largeFont)  
73  xlab('t','fontsize',largeFont)  
74  ylab('N_{\,t}','rot',0,'fontsize',largeFont,'interpreter','latex')  
75  \% Translate the position of the ylabel slightly  
76  ylabh = get(gca,'YLabel');  
77  set(ylabh,'Position',get(ylabh,'Position') - [0.02 0 0]);  
78  \% Now that we've constructed the processes, we want to mark out a step for  
79  \% illustration  
80  hold on  
81  lineType = 'ro-';  
82  lineCol = 'r';  
83  distance = 90;  
84  initPos = 450;  
85  earlierPos = initPos-2*distance;  
86  endPos = initPos+distance;  
87  \% Vertical lines  
88  plot([t(initPos) t(initPos)], [simpleProc(1,initPos,1) 0],lineType)  
89  text(t(initPos),0,'t_k','VerticalAlignment','top','fontsize',largeFont)  
90  plot([t(endPos) t(endPos)], [simpleProc(1,endPos,1) 0],lineType)  
91  text(t(endPos),0,'t_{k+1}','VerticalAlignment','top','fontsize',largeFont)  
92  \% Horizontal lines  
93  plot([t(1) t(initPos)], [simpleProc(1,initPos,1) simpleProc(1,initPos,1)],lineType)  
94  plot([t(1) t(endPos)], [simpleProc(1,endPos,1) simpleProc(1,endPos,1)],lineType)  
95  \% Mark out the distance  
96  \% Vertical line  
97  plot([t(earlierPos) t(earlierPos)], [simpleProc(1,initPos,1) ...  
98  simpleProc(1,endPos,1)],'k')  
99  \% Horizontal line  
100  plot([t(endPos) t(endPos)], [simpleProc(1,initPos,1)-0.5 ...  
101  simpleProc(1,endPos,1)-0.5],'k')  
102  \% Give it a label  
103  text([t(endPos),mean([simpleProc(1,initPos,1),simpleProc(1,endPos,1)])/2, ...  
104  \text{\footnotesize\textcolor{red}{\Delta t}}],'HorizontalAlignment','right', ...  
105  'fontsize',largeFont,'interpreter','latex')  
106  text(mean([t(1),t(endPos)])-0.04,simpleProc(1,initPos,1)-0.5,...  
107  [\text{num2str(distance)}, 'text{\Delta N_{\,t}}'],'VerticalAlignment','top','fontsize',normalFont)  
108  hold off  
111  printToPDF(f,'Figs/PoissonDJ')

lambda = 5;
dist = @(x) randn([x,1])+0.5;
expectedValue = 0.5;
f = figure('Position', [100, 100, 1049, 895]);
procs = PoissonProc(T,dt,M,lambda,dist,expectedValue);
simpleProc = procs.Compound(1,:,1);
plot(t,simpleProc(1,:,1))
set(gca,'XTick', [0 T], 'fontsize', largeFont)
xlabel('t', 'fontsize', largeFont)
ylabel('$\hat{\mathcal{N}}_{t_k}$', 'rot', 0, 'fontsize', largeFont, 'interpreter', 'latex')
set(ylab, 'Position', get(ylab, 'Position') - [0.02 0 0]);

Now that we've constructed the processes, we want to mark out a step for illustration hold on
distance = 140;
initPos = 400;
earlierPos = initPos-2*distance;
endPos = initPos+distance;
% Vertical lines
plot([t(initPos) t(initPos)],[simpleProc(1,initPos,1) 0],lineType)
text(t(initPos),0,'$t_k$', 'VerticalAlignment','top','fontsize',largeFont)
plot([t(endPos) t(endPos)],[simpleProc(1,endPos,1) 0],lineType)
text(t(endPos),0,$t_{k+1}$','VerticalAlignment','top','fontsize',largeFont)
% Horizontal lines plot([t(1) t(initPos)],[simpleProc(1,initPos,1) simpleProc(1,initPos,1)],lineType)
plot([t(1) t(endPos)],[simpleProc(1,endPos,1) simpleProc(1,endPos,1)],lineType)
% Mark out the distance % Vertical line
plot([t(earlierPos) t(earlierPos)],[simpleProc(1,initPos,1) ... simpleProc(1,endPos,1)],'k')
% Horizontal line
plot([t(initPos) t(endPos)],[simpleProc(1,initPos,1)-1.7 ... simpleProc(1,initPos,1)-1.7],'k')
% Give it a label
text(t(earlierPos),mean([simpleProc(1,initPos,1),simpleProc(1,endPos,1)]),... '\\Delta \mathcal{N}_{t_k}\$','HorizontalAlignment','right',... 'fontsize',largeFont,'interpreter','latex')
text(mean([t(initPos),t(endPos)])-0.05,simpleProc(1,initPos,1)-1.7,... [num2str(distance) '\Delta t','VerticalAlignment','top','fontsize',largeFont])
hold off
pause(0.2)
printToPDF(f,'Figs/CompPoissonDJ')

t  = 1;
dt = 2^-10;
M = 2;
t  = 0:dt:T;
lambda = 5;
dist = @(x) randn([x,1])+0.5;
expectedValue = 0.5;
f = figure('Position', [100, 100, 1049, 895]);
procs = PoissonProc(T,dt,M,lambda,dist,expectedValue);
simpleProc = procs.Compensated(1,:,1);
plot(t,simpleProc(1,:,1))
set(gca,'XTick',[0 T],'fontsize',largeFont)
xlabel('$\tilde{N}_t$','rot',0,'fontsize',largeFont,'interpreter','latex')
ylabel('ylabel','rot',0,'fontsize',largeFont,'interpreter','latex')

% Translate the position of the ylabel slightly
ylabh = get(gca,'YLabel');
set(ylabh,'Position',get(ylabh,'Position') - [0.02 0 0]);

% Now that we've constructed the processes, we want to mark out a step for
% illustration
hold on

% Vertical lines
plot([t(initPos) t(initPos)],[simpleProc(1,initPos,1) -1],lineType)
text(t(initPos),-2,'t_k','VerticalAlignment','top','fontsize',largeFont)
plot([t(endPos) t(endPos)],[simpleProc(1,endPos,1) -1],lineType)
text(t(endPos),-2,'t_{k+1}','VerticalAlignment','top','fontsize',largeFont)

% Horizontal lines
plot([t(1) t(initPos)],[simpleProc(1,initPos,1) simpleProc(1,initPos,1)],lineType)
plot([t(1) t(endPos)],[simpleProc(1,endPos,1) simpleProc(1,endPos,1)],lineType)

% Mark out the distance

% Vertical line
plot([t(earlierPos) t(earlierPos)],[simpleProc(1,initPos,1) ...
  simpleProc(1,endPos,1)],'k')

% Horizontal line
plot([t(initPos) t(endPos)],[simpleProc(1,initPos,1) -0.6 ...
  simpleProc(1,initPos,1)-0.6],'k')

% Give it a label

pause(0.2)
printToPDF(f,'Figs/CompCompPoissonDJ')
rng(101)

% Variable controlling font size
largeFont = 20;

% Initialize constants
T = 1;
dt = 2^(-14);
M = 3;
lambda = 5;
dist = @(x) randn([x,1])+0.5;
expectedValue = 0.5;

% Simulate the processes
W = WienerProc(T,dt,M,1);
% While it would be possible to simulate N1, N2 and N3 from the same
% call, they would use the same underlying Poisson processes
N1 = PoissonProc(T,dt,M,lambda,dist,expectedValue);
N2 = PoissonProc(T,dt,M,lambda,dist,expectedValue);
N3 = PoissonProc(T,dt,M,lambda,dist,expectedValue);

% Plot all three processes
fig = figure('Position', [100, 100, 1049, 895]);
plot(0:dt:T,squeeze(W))
set(gca,'XTick', [0 T], 'FontSize', largeFont)
xlabel('$t$', 'FontSize', largeFont)
ylabel('$W_t$', 'rot', 0, 'FontSize', largeFont, 'Interpreter', 'latex')
set(ylab, 'Position', get(ylab, 'Position') - [0.02 0 0]);
pause(0.2); printToPDF(fig, 'Figs/WienerProc.pdf')

% Plot all three processes
fig = figure('Position', [100, 100, 1049, 895]);
plot(0:dt:T,squeeze(N1.Poisson))
set(gca,'XTick', [0 T], 'FontSize', largeFont)
xlabel('$t$', 'FontSize', largeFont)
ylabel('$N_t$', 'rot', 0, 'FontSize', largeFont, 'Interpreter', 'latex')
set(ylab, 'Position', get(ylab, 'Position') - [0.02 0 0]);
pause(0.2); printToPDF(fig, 'Figs/PoiProc.pdf')

% Plot all three processes
fig = figure('Position', [100, 100, 1049, 895]);
plot(0:dt:T,squeeze(N2.Compound))
set(gca,'XTick', [0 T], 'FontSize', largeFont)
xlabel('$\hat{N}_t$', 'FontSize', largeFont, 'Interpreter', 'latex')
set(ylab, 'Position', get(ylab, 'Position') - [0.02 0 0]);
pause(0.2)
printToPDF(fig, 'Figs/CompPoiProc.pdf')

% Plot all three processes
fig = figure('Position', [100, 100, 1049, 895]);
plot(0:dt:T,squeeze(N3.Compensated))
set(gca,'XTick', [0 T], 'FontSize', largeFont)
xlabel('$\tilde{N}_t$', 'FontSize', largeFont, 'Interpreter', 'latex')
set(ylab, 'Position', get(ylab, 'Position') - [0.02 0 0]);
pause(0.2); printToPDF(fig, 'Figs/CompCompPoiProc.pdf')
% Time information
range = 2:2:14;
dt = 2.^-range;
dtLength = length(dt);
T = 1;

% Poisson process information
lambda = 0;
% A distribution which gives a vector of random variables
dist = @(x) ones(x,1);
% Expected value of the distribution
distExp = 1;

% Model information
% Stochastic geometric motion
x0 = 1;
mu = 0.8;
muFun = @(t,x) mu*x;
sigma = 0.2;
sigmaFun = @(t,x) sigma*x;
sigmaFunDeriv = @(t,x) sigma;
eta = 0;
etaFun = @(t,x) eta*x;

% Exact solution at time t
effectSol = @(t,W,N,PoInc) GBPMSolution(W,N,t,mu,sigma,eta,PoInc,x0);

% Stepper information
theta = 0.5;
ExplStep = @(Xn,t,dW,dCN,dt) ExplicitEulerStepper(Xn,dW,dCN,dt,t,muFun,sigmaFun,etaFun);
ImplStep = @(Xn,t,dW,dCN,dt) ImplicitEulerStepperGeom(Xn,dW,dCN,dt,mu,sigma,eta);
MileStep = @(Xn,t,dW,dCN,dt) MilsteinStepperGeom(Xn,dW,dt,muFun,sigmaFun,sigmaFunDeriv);
ThetaStep = @(Xn,t,dW,dCN,dt) ThetaEulerStepperGeom(Xn,dW,dt,mu,sigma,eta,theta);

% Combine into a cell array
noSteppers = length(steppers);
noSteppers = length(steppers);

% Monte Carlo simulation information and memory
batchSize = 2^5;
batchAmount = 2^5;
ResultStrong.mean = cell(dtLength,noSteppers);
ResultStrong.std = cell(dtLength,noSteppers);

% Loop over the different time steps
for i=1:dtLength
% Set up time information
currDt = dt(i);
t = 0:currDt:T;
tLength = length(t);

% Initialize result storage
strong = zeros(tLength,noSteppers);
strongErrorBatchMeans = zeros(batchAmount,tLength,noSteppers);

% Loop over the number of batches
for k=1:batchAmount
MonteCarloMemoryInitialization;
currT = t(1);

% Loop over time
for j = 2:tLength
% Progress info
disp(['Dt: ' num2str((i-1)/dtLength*100,'%6.1f') ... 
', Batch: ' num2str((k-1)/batchAmount*100,'%6.1f') ... 
', Time: ' num2str((j-1)/tLength*100,'%6.1f')]})

% Simulate the process steps
MonteCarloProcessSteps;

% Perform the approximate steps
for m = 1:noSteppers
currVal{m} = steppers{m}(currVal{m},currT,dW,dCompN,c currDt);
% Calculate the next process values
currT = t(j);
W = W + dW;
N = N + dN;
CompN = CompN + dCompN;
CompCompN = CompCompN + dCompCompN;
% Calculate the exact solution
ExactSol = exactSol(currT,W,N,NInc);
% Compute errors at each time step
for m = 1:noSteppers
    strong{j,m} = mean(abs(currVal{m}-ExactSol));
end
% Calculate means and standard deviation over batches
for m = 1:noSteppers
    ResultStrong.mean{i,m} = mean(strongErrorBatchMeans{i,:,m});
    ResultStrong.std{i,m} = std(strongErrorBatchMeans{i,:,m});
end
% Plot strong convergence dependent on time
confLevel = 0.95;
plotInArg.monteRes = ResultStrong;
plotInArg.dt = dt;
plotInArg.T = T;
plotInArg.oneDim = false;
plotInArg.confLevel = confLevel;
plotInArg.batchAmount = batchAmount;
plotInArg.batchSize = batchSize;
plotInArg.ylab = 'Strong error';
plotInArg.functionName = {'Explicit', 'Implicit', 'Milstein', 'Theta, \theta=0.5'};
plotInArg.legendInput = {'Location', 'NorthWest'};
plotInArg.figureInput = {'pos', [400 400 1500 500]};
plotInArg.figName = 'Figs\TimeDrift.pdf';

% Plot end time comparison convergence
plotMonte(plotInArg)
128 plotInArg.translation = [-1 -0.2];
129 plotInArg.ylab = 'Strong error';
130 plotInArg.legendInput = { 'Explicit', 'Implicit', 'Milstein', 'Theta',  
131   '$C_1\sqrt{\Delta t}$', '$C_2\Delta t$', 'Location', 'SouthEast' }; 
132 plotInArg.figureInput = { 'pos', [400 400 800 600] }; 
133 plotInArg.figName = 'Figs\StrongEndTime.pdf';
134 % Plot end time comparison convergence
135 plotMonte(plotInArg)

Code 23. GBPMFourStrongTime.m.
exactSolCell((3*noSchemes+1):4*noSchemes) = {exactSol(sigmaAlt,etaAlt)};

% Monte Carlo simulation information and memory
batchSize = 2^8;
batchAmount = 2^5;
ResultStrong.mean = cell(dtLength,noSteppers);
ResultStrong.std = cell(dtLength,noSteppers);

% Loop over the different time steps
for i=1:dtLength
  % Set up time information
  currDt = dt(i);
t = 0:currDt:T;
tlength = length(t);
  % Initialize result storage
  strong = zeros(tlength,noSteppers);
  strongErrorBatchMeans = zeros(batchAmount,tlength,noSteppers);
  % Loop over the number of batches
  for k=1:batchAmount
    MonteCarloMemoryInitialization;
    % Set current time
    currT = t(1);
    % Loop over time
    for j = 2:tlength
      % Progress info
      disp(['Dt: ' num2str((i-1)/dtLength*100,'%6.1f') ...
        ', Batch: ' num2str((k-1)/batchAmount*100,'%6.1f') ...
        ', Time: ' num2str((j-1)/tlength*100,'%6.1f')])
      MonteCarloProcessSteps;
      % Perform the approximate steps
      for m = 1:noSteppers
        currVal{m} = steppers{m}(currVal{m},currT,dW,dCompN,currDt);
      end
      % Calculate the next process values
      currT = t(j);
      W = W + dW;
      N = N + dN;
      CompN = CompN + dCompN;
      CompCompN = CompCompN + dCompCompN;
      % Compute the error at each time step
      for m = 1:noSteppers
        strong(j,m) = mean(abs(currVal{m}-exactSolCell{m}(currT,W,N,NInc)));
      end
    end
    % Store the means
    strongErrorBatchMeans(k,:,:) = strong;
  end
  % Calculate means and standard deviation over batches
  for m = 1:noSteppers
    ResultStrong.mean{i,m} = mean(strongErrorBatchMeans(:,;,:));
    ResultStrong.std{i,m} = std(strongErrorBatchMeans(:,;,:));
  end
end

%% Plot strong convergence dependent on time
% Parameters: \mu=' num2str(mu)...
% , \lambda=' num2str(lambda) '. Batch size: ' ...
% ,num2str (batchSize) '. Number of batches: ' num2str (batchAmount) '.
subTitleStrings = {'\sigma=0, \eta=0.2', ...
  '\sigma=0.2, \eta=0.2', ...
  '\sigma=0.2, \eta=0', ...
  '\sigma=0', '\eta=0.2'};
se = 'SouthEast';
nw = 'NorthWest';
ee = 'NorthEast';
legendPos = {nw nw nw nw};
nn nw nw nw ;

nn nw nw nw ;

nn nw nw nw ;

nn nw nw nw ;

ymax = [0.07 0.04 0.04 0.04 0.1];
ylab = 'Strong error';

figNames = {'Figs/GBPMExplicitEulerTime',...'
    Figs/GBPMImplicitEulerTime',...'
    Figs/GBPMThetaEulerTime',...'
    Figs/GBPMSSBEulerTime',...'Figs/GBPMCSSBEulerTime'};

hplot.val = false;
CIInfo.val = true;
CIInfo.confLevel = 0.95;
CIInfo.batchSize = batchSize;

% Plot it
FourPlotsInOne(ResultStrong,dt,T,titleString,subTitleStrings,legendPos,...
    ymax,ylab,figNames,hplot,CIInfo);

%% Plot strong convergence at time T
 tempSize = size(ResultStrong.mean);
 endTimeResult.mean = cell(tempSize);
 endTimeResult.std = cell(tempSize);
 for i = 1:tempSize(1)
    for j = 1:tempSize(2)
        endTimeResult.mean{i,j} = ResultStrong.mean{i,j}(end);
        endTimeResult.std{i,j} = ResultStrong.std{i,j}(end);
    end
 end

titleString = ['Parameters: \mu=' num2str(mu)...'
    \lambda=' num2str(lambda) '. Batch size: '...
    'num2str(batchSize) '. Number of batches: ' num2str(batchAmount) '];

subTitleStrings = {'\sigma=0, \eta=0.2',...'
    \sigma=0, \eta=0',...'
    \sigma=0.2, \eta=0','...'
    \sigma=0.2, \eta=0.2'};

sw = 'SouthWest';
se = 'SouthEast';
w = 'NorthWest';
ne = 'NorthEast';
legendInfo.legendNames = {'Explicit Euler','Implicit Euler',...'
    Theta Euler, $\theta$= ' num2str(theta)','$SSB','$CSSB'};

legendInfo.legendPos = {se se se se ;
    se se se ;
    se se nw ;
    se se se ;
    se se se ;
    se se se }; ylab = 'Strong error';

figNames = {'Figs/GBPMExplicitEuler',...
    'Figs/GBPMImplicitEuler',...'
    Figs/GBPMThetaEuler',...'
    Figs/GBPMSSBEuler',...
    'Figs/GBPMCSSBEuler'};

hplot.val = true;
hplot.exponent = [2, 1, 0.5];
hplot.translation = [0, 0, 0];
hplot.legendInput = {'$C_1 \Delta t^2$', '$C_2 \Delta t$', '$C_3 \sqrt{\Delta t}$'};
176  CIInfo.val = true;  
177  CIInfo.confLevel = 0.95;  
178  CIInfo.batchSize = batchSize;  
179  
180  % Plot it  
181  FourPlotsInOne(endTimeResult,dt,T,titleString,subTitleStrings,legendInfo,...  
182  ymax,ylab,figNames,hplot,CIInfo);  
183  

% Time information  
range = 1:2:9;  
dt = 2.^-range;  
dtLength = length(dt);  
T = 1;  
dtReferenceSolPower = 14;  
dtReferenceSol = 2.^-dtReferenceSolPower;  
fineTVector = 0:dtReferenceSol:T;  
fineTimeLength = length(fineTVector);  

% Poisson process information  
lambda = 0;  

% A distribution which gives a vector of random variables  
dist = @(x) ones(x,1);  

% Distribution information, expected value of X and expected value of X^2  
distExp = 1;  
distSqExp = 1;  

% Model information  
% Linear stochastic oscillator  
x0 = [1;0];  
omega = 3;  
alpha = 1;  
wieVec = [0;alpha];  
eta = 0;  
poiVar = [0;eta];  
detMat = [0,1;-omega^2,0];  
partMat = [0,0;-omega^2,0];  
partMat2 = [0,1;0,0];  

% Stepper information  
% Theta-Euler stepper info  
theta = 0.5;  

% CCSV-Euler stepper info  
drift = lambda*distExp;  

% Steppers  
Expl = @(Xn,t,dW,dN,dt) ExplicitEulerStepperMatrix(Xn,dW,dN,dt,...  
  detMat,wieVec,poiVec);  
Impl = @(Xn,t,dW,dN,dt) ImplicitEulerStepperMatrix(Xn,dW,dN,dt,...  
  detMat,wieVec,poiVec);  
Part = @(Xn,t,dW,dN,dt) PartitionedEulerStepper(Xn,dW,dN,dt,...  
  partMat,partMat2,wieVec,poiVec);  
Thet = @(Xn,t,dW,dN,dt) ThetaEulerStepperMatrix(Xn,dW,dN,dt,...  
  detMat,wieVec,poiVec,theta);  
SSB = @(Xn,t,dW,dN,dt) SSBStepperMatrix(Xn,dW,dN,dt,...  
  detMat,wieVec,poiVec);  
CSSB = @(Xn,t,dW,dN,dt) CSSBStepperMatrix(Xn,dW,dN,dt,...  
  detMat,wieVec,poiVec,drift);  
trigScheme = @(Xn,t,dW,dN,dt) TrigStepper (Xn,dW,dN,dt,alpha,eta,omega);  

% Combine into a cell array  
steppers = {Expl,Impl,Part,Thet,SSB,CSSB,trigScheme};  
noSteppers = length(steppers);  

% Monte Carlo simulation information and memory  

Code 24. LBOConvPlots.m.
batchSize = 2^6;
batchAmount = 2^3;
ResultStrong.mean = cell(dtLength,noSteppers);
ResultStrong.std = cell(dtLength,noSteppers);

% Loop over the different time steps
for i=1:dtLength
  % Set up time information
  currDt = dt(i);
t = 0:currDt:T;
tlength = length(t);
exactSolSteps = 2^(dtReferenceSolPower-range(i));
% Initialize result storage
strong = zeros(tlength,noSteppers);
strongErrorBatchMeans = zeros(batchAmount,tlength,noSteppers);
% Loop over the number of batches
for k=1:batchAmount
  % Initialize process storage
  MonteCarloMemoryInitialization;
  % Initialize reference solution memory
  refSol = repmat(x0,batchSize,1);
  % Set current time
  currT = t(1);
currFineT = currT;
  % Loop over time
  for j = 2:tlength
    % Progress info
    disp(['Dt: ' num2str((i-1)/dtLength*100,'%6.1f') ... 
          ', Batch: ' num2str((k-1)/batchAmount*100,'%6.1f') ... 
          ', Time: ' num2str((j-1)/tlength*100,'%6.1f'))]
    % Simulate the process steps
    MonteCarloProcessSteps;
    % Ignore Wiener simulation and simulate the finer steps
    dWFiner = sqrt(dtReferenceSol)*randn(batchSize,exactSolSteps);
dW = sum(dWFiner,2);
    % Calculate the integral up to the next time
    for m = 1:exactSolSteps
      % We have that the trigonometric scheme with the proper
      % settings are available at index 4
      % It does not matter which dCompN we use due to eta=0
      refSol = steppers{4}(refSol,currFineT,...
                  dWFiner(:,m),dCompN,dtReferenceSol);
      % Time progression
      currFineT = currT + dtReferenceSol;
    end
    % Perform the approximate steps
    for m = 1:noSteppers
      currVal{m} = steppers{m}(currVal{m},currT,dW,dCompN,currDt);
    end
    % Calculate the next process values
    currT = t(j);
    W = W + dW;
    N = N + dN;
    CompN = CompN + dCompN;
    CompCompN = CompCompN + dCompCompN;
    % Compute errors at each time step
    for m = 1:noSteppers
      % Only look at first coordinate error
      strong{j,m} = mean(abs(currVal{m}(1:2:(end-1))-refSol(1:2:(end-1))));
    end
    strongErrorBatchMeans(k,:,j) = strong;
  end
end
Calculate means and standard deviation over batches

for m = 1:noSteppers
  ResultStrong.mean{i,m} = mean(strongErrorBatchMeans{i,:,m});
  ResultStrong.std{i,m} = std(strongErrorBatchMeans{i,:,m});
end
disp('Done')

%% Plot strong convergence dependent on time
confLevel = 0.95;

% Plot specifications
plotInArg.monteRes = ResultStrong;
plotInArg.dt = dt;
plotInArg.T = T;
plotInArg.oneDim = false;
plotInArg.confLevel = confLevel;
plotInArg.batchAmount = batchAmount;
plotInArg.batchSize = batchSize;
plotInArg.ylab = 'Strong error';
plotInArg.functionName = {'Explicit','Implicit','Partitioned','Theta', ...  \'\theta=0.5\','SSB','CSSB','Trigonometric'};
plotInArg.legendInput = {'Location','NorthWest'};
plotInArg.figureInput = {'pos',[400 400 1500 500]};
plotInArg.figName = 'Figs\TimeDriftLSO.pdf';

% Plot end time comparison convergence
plotMonte(plotInArg)

%% Plot strong convergence at time T
% Extract the last times
endTimeResult.mean = cell(tempSize);
endTimeResult.std = cell(tempSize);
for i = 1:tempSize(1)
  for j = 1:tempSize(2)
    endTimeResult.mean{i,j} = ResultStrong.mean{i,j}(end);
    endTimeResult.std{i,j} = ResultStrong.std{i,j}(end);
  end
end
confLevel = 0.95;

% Plot specifications
plotInArg.monteRes = endTimeResult;
plotInArg.dt = dt;
plotInArg.T = T;
plotInArg.oneDim = true;
plotInArg.confLevel = confLevel;
plotInArg.batchAmount = batchSize;
plotInArg.batchSize = batchSize;
plotInArg.ylab = 'Strong error';
plotInArg.exponent = [0.5 1];
plotInArg.translation = [-2 -0.2];
plotInArg.legendInput = {'Explicit','Implicit','Partitioned','Theta','SSB',...  \'\CSB\','Trigonometric','C*sqrt(dt)', \'C*dt\','Location','SouthEast'};
plotInArg.figureInput = {'pos',[400 400 800 600]};
plotInArg.figName = 'Figs\StrongEndTimeLSO.pdf';

% Plot end time comparison convergence
plotMonte(plotInArg)
% Time information
range = 6:2:10;
dt = 2.^-range;
dtLength = length(dt);
T = 0.8;

% Poisson process information
lambda = 0;
% A distribution which gives a vector of random variables
dist = @(x) ones(x,1);
% Distribution information, expected value of X and expected value of X^2
distExp = 1;
distSqExp = 1;

% Model information
% Linear stochastic oscillator
x0 = [1;0];
omega = 3;
alpha = 0.2;
wieVec = [0;alpha];
eta = 0;
poiVec = [0;eta];
detMat = [0,1;-omega^2,0];
partMat = [0,0;-omega^2,0];
partMat2 = [0,1;0,0];

% Batch energy function
energy = @(X) 1/2*(omega^2*X(1:2:(end-1)).^2 + X(2:2:end).^2);

% Exact mean at time t
exactEnergyMean = @(t) 1/2*(omega^2*x0(1)^2 + x0(2)^2 ...
+ alpha^2*t + eta^2*lambda*distSqExp*t)...
+ (eta*lambda*x0(1) - eta^2*lambda^2/omega^2)*cos(omega*t) ...
+ eta*lambda*x0(2)/omega*sin(omega*t);
t = 0:currDt:T;
tlength = length(t);

% Initialize result storage
weakEnergy = zeros(tlength,noSteppers);
weakEnergyBatchMeans = zeros(batchAmount,tlength,noSteppers);

% Loop over the number of batches
for k=1:batchAmount
  % Initialize process storage
  MonteCarloMemoryInitialization;
  % Set current time
  currT = t(1);
  % Set initial energy
  weakEnergy(1,:) = energy(x0);
  % Loop over time
  for j = 2:tlength
    % Progress info
    disp(['Dt: ' num2str((i-1)/dtLength*100,'%6.1f') ... 
      ', Batch: ' num2str((k-1)/batchAmount*100,'%6.1f') ... 
      ', Time: ' num2str((j-1)/tlength*100,'%6.1f'))]
    % Simulate the process steps
    MonteCarloProcessSteps;
    % Perform the approximate steps
    for m = 1:noSteppers
      currVal{m} = steppers{m}(currVal{m},currT,dW,dCompN,currDt);
    end
    % Calculate the next process values
    currT = t(j);
    W = W + dW;
    N = N + dN;
    CompN = CompN + dCompN;
    CompCompN = CompCompN + dCompCompN;
    % Compute errors at each time step
    for m = 1:noSteppers
      % Look at the energy
      weakEnergy(j,m) = mean(energy(currVal{m}));
    end
  end
  weakEnergyBatchMeans(k,:,:) = weakEnergy;
end

% Calculate mean energy
meanEn = exactEnergyMean(t);
% Calculate means and standard deviation over batches
for m = 1:noSteppers
  ResultEnergyWeak.mean{i,m} = abs(mean(weakEnergyBatchMeans(:,:,m))-meanEn);
  ResultEnergyWeak.std{i,m} = std(weakEnergyBatchMeans(:,:,m));
end

disp('Done')

%% Plot weak convergence dependent on time
confLevel = 0.95;
plotInArg.monteRes = ResultEnergyWeak;
plotInArg.dt = dt;
plotInArg.T = T;
plotInArg.oneDim = false;
plotInArg.confLevel = confLevel;
plotInArg.batchAmount = batchAmount;
plotInArg.batchSize = batchSize;
plotInArg.ylab = 'Weak error of energy';
plotInArg.functionName = {'Explicit','Implicit','Partitioned','Theta, ... 
\theta=0.5','SSB','CSSB','Trigonometric'};
plotInArg.functionName = {'Explicit','Partitioned','SSB'};
plotInArg.legendInput = {'Location','NorthWest'};
% Time information
range = 1:2:11;
dt = 2.^-range;
dtLength = length(dt);
T = 1;
dtReferenceSolPower = 14;
dtReferenceSol = 2.^-dtReferenceSolPower;
fineTVector = 0:dtReferenceSol:T;
fineTimeLength = length(fineTVector);

% Poisson process information
lambda = 3;
% A distribution which gives a vector of random variables
dist = @(x) ones(x,1);
% Distribution information, expected value of X and expected value of X^2
distExp = 1;
distExp2 = 1;

%% Plot weak convergence at time T
% Extract the last times
tempSize = size(ResultEnergyWeak.mean);
endTimeResult.mean = cell(tempSize);
endTimeResult.std = cell(tempSize);
for i = 1:tempSize(1)
    for j = 1:tempSize(2)
        endTimeResult.mean{i,j} = ResultEnergyWeak.mean{i,j} ;(end);
        endTimeResult.mean{i,j} = endTimeResult.mean{i,j};
        endTimeResult.std{i,j} = ResultEnergyWeak.std{i,j}(end);
    end
end

% Plot specifications
plotInArg.monteRes = endTimeResult;
plotInArg.dt = dt;
plotInArg.T = T;
plotInArg.oneDim = true;
plotInArg.confLevel = confLevel;
plotInArg.batchAmount = batchSize;
plotInArg.batchSize = batchSize;
plotInArg.exponent = [0.5 1];
plotInArg.translation = [-1 1.3];

% Plot end time comparison convergence
plotMonte(plotInArg)
% Linear stochastic oscillator
x0 = [1; 0];
omega = 3;
alpha = 0.2;
alphaAlt = 0;
wieVec = @(alpha) [0; alpha];
eta = 0.2;
ettaAlt = 0;
poiVec = @(eta) [0; eta];
detMat = [0,1; -omega^2, 0];
partMat = [0,0; -omega^2, 0];
partMat2 = [0,1; 0, 0];

% Stepper information
% Theta-Euler stepper info
theta = 0.5;

% CCSB-Euler stepper info
drift = lambda*distExp;

% Steppers
Expl = @(alpha, eta)@(Xn,t,dW,dN,dt) ExplicitEulerStepperMatrix(Xn,dW,dN,dt,...
detMat,wieVec(alpha),poiVec(eta));
Impl = @(alpha, eta)@(Xn,t,dW,dN,dt) ImplicitEulerStepperMatrix(Xn,dW,dN,dt,...
detMat,wieVec(alpha),poiVec(eta));
Part = @(alpha, eta)@(Xn,t,dW,dN,dt) PartitionedEulerStepper(Xn,dW,dN,dt,...
partMat,partMat2,wieVec(alpha),poiVec(eta));
Thet = @(alpha, eta)@(Xn,t,dW,dN,dt) ThetaEulerStepperMatrix(Xn,dW,dN,dt,...
detMat,wieVec(alpha),poiVec(eta),theta);
SSB = @(alpha, eta)@(Xn,t,dW,dN,dt) SSBEulerStepperMatrix(Xn,dW,dN,dt,...
detMat,wieVec(alpha),poiVec(eta));
CSSB = @(alpha, eta)@(Xn,t,dW,dN,dt) CSSBEulerStepperMatrix(Xn,dW,dN,dt,...
detMat,wieVec(alpha),poiVec(eta),drift);
trigScheme = @(alpha, eta)@(Xn,t,dW,dN,dt) TrigStepper(Xn,dW,dN,dt, alpha, eta, omega);

% Combine into a cell array
steppers = {Expl(alpha,eta), Impl(alpha,eta), Part(alpha,eta),...ثير
Theta(alpha, eta), SSB(alpha, eta), CSSB(alpha, eta), trigScheme(alpha, eta),...
Expl(alphaAlt, eta), Impl(alphaAlt, eta), Part(alphaAlt, eta),...ثير
Theta(alphaAlt, eta), SSB(alphaAlt, eta), CSSB(alphaAlt, eta), trigScheme(alphaAlt, eta),...
Expl(alphaAlt, etaAlt), Impl(alphaAlt, etaAlt), Part(alphaAlt, etaAlt),...ثير
Theta(alphaAlt, etaAlt), SSB(alphaAlt, etaAlt), CSSB(alphaAlt, etaAlt), trigScheme(alphaAlt, etaAlt),...}
noSteppers = length(steppers);

% Function information
refSolCell = cell(noSteppers,1);
noSchemes = noSteppers/4;
refSolCell(1:noSchemes) = {trigScheme(alpha, eta)};
refSolCell((noSchemes+1):2*noSchemes) = {trigScheme(alphaAlt, eta)};
refSolCell((2*noSchemes+1):3*noSchemes) = {trigScheme(alpha, etaAlt)};
refSolCell((3*noSchemes+1):4*noSchemes) = {trigScheme(alphaAlt, etaAlt)};

% Monte Carlo simulation information and memory
batchSize = 2^6;
batchAmount = 2^3;
ResultStrong.mean = cell(dtLength,noSteppers);
ResultStrong.std = cell(dtLength,noSteppers);

% Loop over the different time steps
for i=1:dtLength
% Set up time information
currDt = dt(i);
t = 0:currDt:T;
length = length(t);
refSolSteps = 2^(`dtReferenceSolPower-range(i)`);
% Initialize result storage
strong = zeros(tlength,noSteppers);
strongErrorBatchMeans = zeros(batchAmount,tlength,noSteppers);

% Loop over the number of batches
for k=1:batchAmount
    % Initialize process storage
    MonteCarloMemoryInitialization;
    % Initialize reference solution memory
    refSol = cell(noSteppers,1);
    for m = 1:noSteppers
        refSol{m} = repmat(x0,batchSize,1);
    end
    % Set current time
    currT = t{1};
    currFineT = currT;
    % Loop over time
    for j = 2:tlength
        % Progress info
        disp(['Dt: ' num2str((i-1)/dtLength*100,'%6.1f') ... 
             ', Batch: ' num2str((k-1)/batchAmount*100,'%6.1f') ... 
             ', Time: ' num2str((j-1)/tlength*100,'%6.1f')]i)
        % Simulate the process steps
        MonteCarloProcessSteps;
        % Ignore Wiener simulation and simulate the finer steps
        dWFiner = sqrt(dtReferenceSol)*randn(batchSize,refSolSteps);
        dW = sum(dWFiner,2);
        % As for the Poisson process we will opt for a less memory and
        % batch size efficient approach
        % Allocate memory
        dNFiner = zeros(batchSize,refSolSteps);
        dCompNFiner = zeros(batchSize,refSolSteps);
        % Construct a finer time interval
        timeIntervalFiner = currT:dtReferenceSol:(currT+dt(i));
        % We will use theorem 5.19
        % Loop over each index where we have one increment on the
        % interval
        % Simulate a time for when the increment occurs. Since it’s
        % uniformly over the interval it is equivalent to one of
        % the indexes
        indAtLeastOneInc = find(dN > 0);
        nAtLeastOneInc = length(indAtLeastOneInc);
        indexCol = 1:refSolSteps;
        % Loop over the batches which have an increment
        for m = 1:nAtLeastOneInc
            currBatchIndex = indAtLeastOneInc(m);
            % Loop over all increments and position them randomly
            % within the finer time scale
            for l = 1:dN(currBatchIndex)
                timeIndex = randsample(indexCol,1);
                dCompNFiner(currBatchIndex,timeIndex) = + ...
                dCompNFiner(currBatchIndex,timeIndex) + dNInc{currBatchIndex}(l);
            end
        end
        % Now that we have the fine compound Poisson process done,
        % calculate the compensated compound Poisson process
        dCompCompNFiner = dCompNFiner - lambda*distExp*dtReferenceSol;
        % Calculate the reference solution up to the next time
        for m = 1:4
            refSol{((m-1)*noSchemes+1):m*noSchemes} = ... 
            {refSolCell{noSchemes*m}{refSol{noSchemes+m}},... 
            currFineT,dWFiner(:,l),dCompNFiner(:,l),dtReferenceSol});
        end
    end
end
% Perform the approximate steps
for m = 1:noSteppers
currVal{m} = steppers{m}(currVal{m},currT,dW,dCompN,currDt);
end

% Calculate the next process values
currT = t(j);
W = W + dW;
N = N + dN;
CompN = CompN + dCompN;
CompCompN = CompCompN + dCompCompN;

% Compute errors at each time step
for m = 1:noSteppers
% Only look at first coordinate error, map m to {1,2,3,4}
% for reference solution
strong(j,m) = mean(abs(currVal{m}(1:2:(end-1))-...
refSol{m}(1:2:(end-1))));
end
end

% Calculate means and standard deviation over batches
for m = 1:noSteppers
ResultStrong.mean{i,m} = mean(strongErrorBatchMeans(:,:,m));
ResultStrong.std{i,m} = std(strongErrorBatchMeans(:,:,m));
end

% Plot strong convergence dependent on time
titleString = ['Parameters: \omega=' num2str(omega) ...'
\lambda=' num2str(lambda) '. Batch size: '....
',num2str(batchSize) '. Number of batches: ' num2str(batchAmount) '.'];
subTitleStrings = {'\alpha=0, \eta=0.2' ,...
'\alpha=0.2, \eta=0.2' ,...
'\alpha=0.2, \eta=0' ,...
'\alpha=0\eta=0'};
se = 'SouthEast';
wv = 'NorthWest';
ne = 'NorthEast';
legendPos = {nw nw nw nw ;
 northwest nw nw ;
nw nw nw nw ;
 nw nw nw nw ;
 nw nw nw nw ;
 nw nw nw nw ;
 nw nw nw nw ;
 nw nw nw nw ;

ymax = [0.5 0.5 0.1 0.6 0.6 0.01];
ylab = 'Strong error';
figNames = {'Figs/LBPOExplicitEulerTime',...
'Figs/LBPOImplicitEulerTime',...
'Figs/LBPOPartitionedEulerTime',...
'Figs/LBPOThetaEulerTime',...
'Figs/LBPOStabilityEulerTime',...
'Figs/LBPOTrigTime'};
hplot.val = false;
CIInfo.val = true;
CIInfo.confLevel = 0.95;
CIInfo.batchSize = batchSize;
% Plot it
FourPlotsInOne(ResultStrong, dt, T, titleString, subTitleStrings, legendInfo,...
ymax, ylab, figNames, hplot, CIInfo);

%% Plot strong convergence at time T

tempSize = size(ResultStrong.mean);
endTimeResult.mean = cell(tempSize);
endTimeResult.std = cell(tempSize);
for i = 1:tempSize(1)
  for j = 1:tempSize(2)
    endTimeResult.mean{i,j} = ResultStrong.mean{i,j}(end);
    endTimeResult.std{i,j} = ResultStrong.std{i,j}(end);
  end
end
titleString = ['Parameters: \omega=' num2str(omega) ...'
', '\lambda=' num2str(lambda) '. Batch size: '
', num2str(batchSize) '. Number of batches: ' num2str(batchAmount) ' .'];

subTitleStrings = {'\alpha=0, \eta=0.2', ...'
\alpha=0.2, \eta=0', ...'
\alpha=0.2, \eta=0.2' , ...'
\alpha=0', ...'
\alpha=0'};

sw = 'SouthWest';
se = 'SouthEast';
nw = 'NorthWest';
ne = 'NorthEast';
legendInfo.legendNames = {'Explicit Euler', 'Implicit Euler', 'Partitioned', 'Theta Euler', 'SSB', 'CSSB', 'Trigonometric'};

legendInfo.legendPos = {se se se se ;
  se se se se ;
  se se se se ;
  se se se se ;
  se se nw };

ylab = 'Strong error';

figNames = {'Figs/LBPOExplicitEuler',...'
  'Figs/LBPOImplicitEuler',...'
  'Figs/LBPOPartitionedEuler',...'
  'Figs/LBPOThetaEuler',...'
  'Figs/LBPOSSBEuler',...'
  'Figs/LBPOCSSBEuler',...'
  'Figs/LBPOTrig'};

hplot.val = true;
hplot.exponent = [2, 1, 0.5];
hplot.translation = [0, 0, 0];
hplot.legendInput = {'$C_1 \Delta t^2$', '$C_2 \Delta t$', '$C_3 \sqrt{\Delta t}$'};
CIInfo.val = true;
CIInfo.confLevel = 0.95;
CIInfo.batchSize = batchSize;

% Plot it
FourPlotsInOne(endTimeResult, dt, T, titleString, subTitleStrings, legendInfo,...
ymax, ylab, figNames, hplot, CIInfo);
% Time information
range = 1:2:11;
dt = 2.^-range;
dtLength = length(dt);
T = 1;
dtReferenceSolPower = 14;
dtReferenceSol = 2.^-dtReferenceSolPower;
fineTVector = 0:dtReferenceSol:T;
fineTimeLength = length(fineTVector);

% Poisson process information
lambda = 3;
dist = @(x) ones(x,1);
distExp = 1;
distSqExp = 1;

% Model information
x0 = [1;0];
omega = 3;
alpha = 0.2;
alphaAlt = 0;
wieVec = @(alpha) [0;alpha];
eta = 0.2;
etAlt = 0;
poiVec = @(eta) [0;eta];
detMat = [0,1;-omega^2,0];
partMat = [0,0;-omega^2,0];
partMat2 = [0,1;0,0];

% Exact solution at time t
% This uses the fundamental matrix representation of the solution
fundMat = @(t) [cos(omega*t) , sin(omega*t)/omega ;
               -omega*sin(omega*t) , cos(omega*t)];
fundMatInv = @(t) [cos(omega*t) , -sin(omega*t)/omega ;
                   omega*sin(omega*t) , cos(omega*t)];
g = @(t) [0;1];
extactSol = @(alpha,eta) @(t,WInt,NInt) x0(1)*cos(omega*t) + x0(2)*sin(omega*t)/omega ...
            + alpha*WInt + eta*NInt;

% Stepper information
% Theta-Euler stepper info
theta = 0.5;

% SSB-Euler stepper info
drift = lambda*distExp;

% Steppers
Expl = @(alpha,eta) @(Xn,t,dW,dN,dt) ExplicitEulerStepperMatrix(Xn,dW,dN,dt,...
detMat,wieVec(alpha),poiVec(eta));
Impl = @(alpha,eta) @(Xn,t,dW,dN,dt) ImplicitEulerStepperMatrix(Xn,dW,dN,dt,...
detMat,wieVec(alpha),poiVec(eta));
Part = @(alpha,eta) @(Xn,t,dW,dN,dt) PartitionedEulerStepper(Xn,dW,dN,dt,...
partMat,partMat2,wieVec(alpha),poiVec(eta));
Thet = @(alpha,eta) @(Xn,t,dW,dN,dt) ThetaEulerStepperMatrix(Xn,dW,dN,dt,...
detMat,wieVec(alpha),poiVec(eta),theta);
SSB = @(alpha,eta) @(Xn,t,dW,dN,dt) SSBEulerStepperMatrix(Xn,dW,dN,dt,...
detMat,wieVec(alpha),poiVec(eta),drift);
CSSB = @(alpha,eta) @(Xn,t,dW,dN,dt) CSSBEulerStepperMatrix(Xn,dW,dN,dt,...
detMat,wieVec(alpha),poiVec(eta),drift);
trigScheme = @(alpha,eta) @(Xn,t,dW,dN,dt) TrigStepper(Xn,dW,dN,dt,alpha,eta,omega);

% Combine into a cell array
steppers = {Expl(alpha,eta),Impl(alpha,eta),Part(alpha,eta),...
Thet(alpha,eta),SSB(alpha,eta),CSSB(alpha,eta),trigScheme(alpha,eta),...
Expl(alphaAlt,eta),Impl(alphaAlt,eta),Part(alphaAlt,eta),...
Thet(alphaAlt,eta),SSB(alphaAlt,eta),CSSB(alphaAlt,eta),trigScheme(alphaAlt,eta),...}
Expl(alpha, etaAlt), Impl(alpha, etaAlt), Part(alpha, etaAlt), ...
Thet(alpha, etaAlt), SSB(alpha, etaAlt), CSSB(alpha, etaAlt), trigScheme(alpha, etaAlt), ...
Expl(alpha, etaAlt), Impl(alphaAlt, etaAlt), Part(alpha, etaAlt), ...
Thet(alphaAlt, etaAlt), SSB(alphaAlt, etaAlt), CSSB(alphaAlt, etaAlt), trigScheme(alpha, etaAlt), ...

noSteppers = length(steppers);

% Function information
refSolCell = cell(noSteppers,1);
noSchemes = noSteppers/4;
refSolCell(1:noSchemes) = {trigScheme(alpha, eta)};
refSolCell((noSchemes+1):2*noSchemes) = {trigScheme(alphaAlt, eta)};
refSolCell((2*noSchemes+1):3*noSchemes) = {trigScheme(alpha, etaAlt)};
refSolCell((3*noSchemes+1):4*noSchemes) = {trigScheme(alphaAlt, etaAlt)};

% Monte Carlo simulation information and memory
batchSize = 2^6;
batchAmount = 2^3;
ResultStrong.mean = cell(dtLength, noSteppers);
ResultStrong.std = cell(dtLength, noSteppers);

% Want to perform following multiplication for each batch
ProcInt = ProcInt + fundMatInv*g.*dProc
% Initialize cell memory in preparation of sparse matrices
fundInvDiag = cell(batchSize,1);
fundDiag = cell(batchSize,1);

% Temporary memory for tall vector
dWLarge = zeros(2*batchSize,1);
dCompCompNLarge = zeros(2*batchSize,1);

% Loop over the different time steps
for i=1:dtLength
  \text{Set up time information}
  currDt = dt(i);
  t = 0:currDt:T;
tlength = length(t);
refSolSteps = 2^{\text{dtReferenceSolPower}-\text{range}(i)};
% Initialize result storage
strong = zeros(tlength, noSteppers);
strongErrorBatchMeans = zeros(batchAmount, tlength, noSteppers);
weakEnergy = zeros(tlength, noSteppers);
weakEnergyBatchMeans = zeros(batchAmount, tlength, noSteppers);
% Loop over the number of batches
for k=1:batchAmount
  \text{Monte Carlo Memory Initialization;}
  \text{Initialize reference solution memory}
  refSol = cell(noSteppers,1);
  for m = 1:noSteppers
    refSol{m} = repmat(x0, batchSize, 1);
  end
  \text{Set current time}
  currT = t(1);
currFineT = currT;
% Loop over time
  for j = 2:tlength
    \text{Progress info}
    disp(['Dt: ' num2str((i-1)/dtLength*100, '%6.1f') ...
          ', Batch: ' num2str((k-1)/batchAmount*100, '%6.1f') ...
          ', Time: ' num2str((j-1)/tlength*100, '%6.1f'))]
    \text{Simulate the process steps}
    MonteCarloProcessSteps;
    \text{Ignore Wiener simulation and simulate the finer steps}
    dWFiner = sqrt(dtReferenceSol)\times\text{randn}(batchSize, refSolSteps);
    d\bar{W} = \text{sum}(dWFiner,2);
% As for the Poisson process we will opt for a less memory and batch size efficient approach
% Allocate memory
dNFiner = zeros(batchSize,refSolSteps);
dCompNFiner = zeros(batchSize,refSolSteps);
% Construct a finer time interval
TimeIntervalFiner = currT:dtReferenceSol:(currT+dt(i));
% We will use theorem 5.19
% Loop over each index where we have one increment on the interval
% Simulate a time for when the increment occurs. Since it's uniformly over the interval it is equivalent to one of the indexes
indAtLeastOneInc = find(dN > 0);
nAtLeastOneInc = length(indAtLeastOneInc);
indexCol = 1:refSolSteps;
% Loop over the batches which have an increment
for m = 1:nAtLeastOneInc
    currBatchIndex = indAtLeastOneInc(m);
    % Loop over all increments and position them randomly within the finer time scale
    for l = 1:dN(currBatchIndex)
        timeIndex = randsample(indexCol,1);
        dCompNFiner(currBatchIndex,timeIndex) = + ...
        dCompNFiner(currBatchIndex,timeIndex) + dNInc{currBatchIndex}(l);
    end
end
% Now that we have the fine compound Poisson process done, calculate the compensated compound Poisson process
dCompCompNFiner = dCompNFiner - lambda*distExp*dtReferenceSol;
% Calculate the refernce solution up to the next time
for m = 1:4
    for l = 1:refSolSteps
        refSol(((m-1)*noSchemes+1):m*noSchemes) =...
        {refSolCell{noSchemes*m}refSol{noSchemes*m},...
        currFineT,dWFiner(:,l),dCompCompNFiner(:,l),dtReferenceSol});
    end
end
% Perform the approximate steps
for m = 1:noSteppers
    currVal{m} = steppers{m}{currVal{m},currT,dW,dCompCompN,currDt};
end
% Calculate the next process values
currT = t(j);
W = W + dW;
N = N + dN;
CompN = CompN + dCompN;
CompCompN = CompCompN + dCompCompN;
% Compute errors at each time step
for m = 1:noSteppers
    % Only look at first coordinate error, map m to {1,2,3,4}
    % for reference solution
    strong{1,m} = mean(abs(currVal{m}(1:2:(end-1)))-...
    refSol{m}(1:2:(end-1))));
    end
strongErrorBatchMeans(k,:,:) = strong;
% Calculate means and standard deviation over batches
for m = 1:noSteppers
    ResultStrong.mean{i,m} = mean(strongErrorBatchMeans(:,i,m));
191     ResultStrong.std{i,m} = std(strongErrorBatchMeans(:, :, m));
192 end
193 end
194 disp('Done')
195
196 %% Plot strong convergence dependent on time
197 titleString = ['Parameters: \omega=' num2str(omega) ...
198         ', \lambda= ' num2str(lambda) '. Batch size: '...
199         ',num2str(batchSize) '. Number of batches: ' num2str(batchAmount) '.'];
200 subTitleStrings = {{'\alpha=\eta=0.2' ,...}
201    {'\alpha=0, \eta=0.2' ,...}
202    {'\alpha=0.2, \eta=0' ,...}
203    {'\alpha=\eta=0'};
204 se = 'SouthEast';
205 nw = 'NorthWest';
206 ne = 'NorthEast';
207 legendPos = {nw nw nw nw ;
208    nw nw nw nw ;
209    nw nw nw nw ;
210    nw nw nw nw ;
211    nw nw nw nw ;
212    nw nw nw nw ;
213    nw nw nw nw ;
214    ymax = [0.5 0.5 1 0.1 0.6 0.6 0.01];
215 ylab = 'Strong error';
216
217 figNames = {'Figs/LBPOCompExplicitEulerTime',...
218    'Figs/LBPOCompImplicitEulerTime',...
219    'Figs/LBPOCompPartitionedEulerTime',...
220    'Figs/LBPOCompThetaEulerTime',...
221    'Figs/LBPOCompSSBEulerTime',...
222    'Figs/LBPOCompCSSBEulerTime',...
223    'Figs/LBPOCompTrigTime'};
224
225 hplot.val = false;
226 CIInfo.val = true;
227 CIInfo.confLevel = 0.95;
228 CIInfo.batchSize = batchSize;
229
230 % Plot it
231 figure
232 ymax,ylab,figNames,hplot,CIInfo);adelic
233
234 %% Plot strong convergence at time T
235 %% Extract the last times
236 tempSize = size(ResultStrong.mean);
237 endTimeResult.mean = cell(tempSize);
238 endTimeResult.std = cell(tempSize);
239 for i = 1:tempSize(1)
240     for j = 1:tempSize(2)
241         endTimeResult.mean{i,j} = ResultStrong.mean{i,j}(end);
242         endTimeResult.std{i,j} = ResultStrong.std{i,j}(end);
243     end
244 end
245
titleString = ['Parameters: \omega=' num2str(omega) ...
246         ', \lambda= ' num2str(lambda) '. Batch size: '...
247         ',num2str(batchSize) '. Number of batches: ' num2str(batchAmount) '.'];
248 subTitleStrings = {{'\alpha=\eta=0.2' ,...}
249    {'\alpha=0, \eta=0.2' ,...}
250    {'\alpha=0.2, \eta=0' ,...}
251    {'\alpha=\eta=0'};
252 sw = 'SouthWest';
253 se = 'SouthEast';
nw = 'NorthWest';
ne = 'NorthEast';

legendInfo.legendNames = {'Explicit Euler','Implicit Euler','Partitioned','Theta Euler','SSB','CSSB','Trigonometric'};

legendInfo.legendPos = {se se se se ; se se se se ; se se se se ; se se se se ; se se nw };
ylab = 'Strong error';

figNames = {'Figs/LBPOCompExplicitEuler','Figs/LBPOCompImplicitEuler','Figs/LBPOCompPartitionedEuler','Figs/LBPOCompThetaEuler','Figs/LBPOCompSSBEuler','Figs/LBPOCompCSSBEuler','Figs/LBPOCompTrig'};

hplot.val = true;
hplot.exponent = [2, 1, 0.5];
hplot.translation = [0, 0, 0];
hplot.legendInput = {abs(C1 \Delta t^2), abs(C2 \Delta t), abs(C3 \sqrt{\Delta t})};

CIInfo.val = true;
CIInfo.confLevel = 0.95;
CIInfo.batchSize = batchSize;

% Plot it
FourPlotsInOne(endTimeResult,dt,T,titleString,subTitleStrings,legendInfo,...
ymax,ylab,figNames,hplot,CIInfo);

% Plot legend positions
sw = 'SouthWest';
se = 'SouthEast';
w = 'NorthWest';
ne = 'NorthEast';

% Time information
range = 1:2:7;
dt = 2.^-range;
dtLength = length(dt);
T = 10;

% Poisson process information
lambda = 3;
% A distribution which gives a vector of random variables
dist = @(x) ones(x,1);
% Distribution information, expected value of X and expected value of X^2
distExp = 1;
distSqExp = 1;

% Model information
% Linear stochastic oscillator
x0 = [1;0];
omega = 3;
alpha = 0.6;
alphaAlt = 0;
wieVec = @(alpha) [0;alpha];
et = 0.6;
etAlt = 0;
poiVec = @(eta) [0;eta];
detMat = [0,1;-omega^2,0];
partMat = [0,0;-omegaˆ2,0];
partMat2 = [0,1;0,0];

% Batch energy function
energy = @(X) 1/2*(omegaˆ2*X(1:2:(end-1)).ˆ2 + X(2:2:end).ˆ2 ... + alphaˆ2*t + etaˆ2*lambda*distSqExp*t)... + (eta*lambda*distExp*x0(1) - etaˆ2*lambdaˆ2/omegaˆ2*distExpˆ2)*(cos(omega*t)-1) ... + eta*lambda*distExp*x0(2)/omega*sin(omega*t);

% Energy mean at time t for compound Poisson process
exactEnergyMeanCompound = @(alpha,eta) @(t) 1/2*(omegaˆ2*x0(1)ˆ2 + x0(2)ˆ2 ... + betaˆ2*t + etaˆ2*lambda*distExp*t)...
+ (eta*lambda*distExp*x0(1) - etaˆ2*lambdaˆ2/omegaˆ2*distExpˆ2)*(cos(omega*t)-1) ...
+ eta*lambda*distExp*x0(2)/omega*sin(omega*t);

% Energy mean at time t for compensated compound Poisson process
exactEnergyMeanCompensated = @(alpha,eta) @(t) ...
1/2*(omegaˆ2*x0(1)ˆ2 + x0(2)ˆ2 + alphaˆ2*t + etaˆ2*lambda*distSqExp*t);

% Stepper information
% Theta-Euler stepper info
theta = 0.5;
% CCSB-Euler stepper info
drift = lambda*distExp;

% Steppers
Expl = @(alpha,eta) @(Xn,t,dW,dN,dt) ExplicitEulerStepperMatrix(Xn,dW,dN,dt,...
detMat,wieVec(alpha),poiVec(eta));
Impl = @(alpha,eta) @(Xn,t,dW,dN,dt) ImplicitEulerStepperMatrix(Xn,dW,dN,dt,...
detMat,wieVec(alpha),poiVec(eta));
Part = @(alpha,eta) @(Xn,t,dW,dN,dt) PartitionedEulerStepper(Xn,dW,dN,dt,...
partMat,partMat2,wieVec(alpha),poiVec(eta));
Thet = @(alpha,eta) @(Xn,t,dW,dN,dt) ThetaEulerStepperMatrix(Xn,dW,dN,dt,...
detMat,wieVec(alpha),poiVec(eta),theta);
SSB = @(alpha,eta) @(Xn,t,dW,dN,dt) SSB Euler Stepper Matrix(Xn,dW,dN,dt,...
detMat,wieVec(alpha),poiVec(eta));
CSSB = @(alpha,eta) @(Xn,t,dW,dN,dt) CSSB Euler Stepper Matrix(Xn,dW,dN,dt,...
detMat,wieVec(alpha),poiVec(eta),drift);
trigScheme = @(alpha,eta) @(Xn,t,dW,dN,dt) TrigStepper(Xn,dW,dN,dt,alpha,eta,omega);
%
% Combine into a cell array
steppers = {Expl(alpha,eta), Impl(alpha,eta), Part(alpha,eta),... Thet(alpha,eta), SSB(alpha,eta), CSSB(alpha,eta), trigScheme(alpha,eta),... Expl(alphaAlt,eta), Impl(alphaAlt,eta), Part(alphaAlt,eta),... Thet(alpha,eta), CSSB(alpha,eta), CSSB(alpha,eta), trigScheme(alpha,eta),... Expl(alphaAlt,eta), Impl(alphaAlt,eta), Part(alphaAlt,eta),... Thet(alpha,eta), CSSB(alpha,eta), CSSB(alpha,eta), trigScheme(alpha,eta),... CSSB(alpha,eta), CSSB(alpha,eta), trigScheme(alpha,eta),etaAlt)};
noSteppers = length(steppers);

% Function information
% Compound case
exactEnCellCompound = cell(noSteppers,1);
noSchemes = noSteppers/4;
exactEnCellCompound1:noSchemes = {exactEnergyMeanCompound(alpha,eta)};
exactEnCellCompound1:noSchemes = {exactEnergyMeanCompound(alpha,eta)};

% Compensated compound case
exactEnCellCompensated1:noSchemes = {exactEnergyMeanCompensated(alpha,eta)};
exactEnCellCompensated1:noSchemes = {exactEnergyMeanCompensated(alpha,eta)};

% Monte Carlo simulation information and memory
batchSize = 2^10;
batchAmount = 2^4;
% Loop over the different time steps
for i=1:dtLength
    % Set up time information
    currDt = dt(i);
    t = 0:currDt:T;
    tlength = length(t);
    % Initialize result storage, one for a CPoPr, one for a CCPoPr
    EnergyCompound = zeros(tlength,noSteppers);
    EnergyCompound(:,:,i,:) = ones(1,noSteppers)*energy(x0);
    EnergyBatchMeansCompound = zeros(batchAmount,tlength,noSteppers);
    EnergyCompensated = zeros(tlength,noSteppers);
    EnergyCompensated(:,:,i,:) = ones(1,noSteppers)*energy(x0);
    EnergyBatchMeansCompensated = zeros(batchAmount,tlength,noSteppers);
    % Loop over the number of batches
    for k=1:batchAmount
        % Initialize process storage
        MonteCarloMemoryInitialization;
        % Initialize stepper memory
        % Standard currVal used for compound Poisson process
        currVal = cell(noSteppers,1);
        currValComp = cell(noSteppers,1);
        for m = 1:noSteppers
            currVal{m} = repmat(x0,batchSize,1);
        end
        % Set current time
        currT = t(1);
        % Loop over time
        for j = 2:tlength
            % Progress info
            disp([\'Dt: \' num2str((i-1)/dtLength*100,\'%.1f\') ...
                  \', Batch: \' num2str((k-1)/batchAmount*100,\'%.1f\') ...
                  \', Time: \' num2str((j-1)/tlength*100,\'%.1f\')]')
            % Simulate the process steps
            MonteCarloProcessSteps;
            % Perform the approximate steps
            for m = 1:noSteppers
                currVal{m} = steppers{m}(currVal{m},currT,dW,dCompN,ccurrT);
                currValComp{m} = steppers{m}(currValComp{m},currT,dW,dCompN,ccurrT);
            end
            % Calculate the next process values
            currT = t(j);
            NW = NW + dW;
            NN = NN + dN;
            CompN = CompN + dCompN;
            CompCompN = CompCompN + dCompCompN;
            % Compute errors at each time step
            for m = 1:noSteppers
                % Only look at first coordinate error
                EnergyCompound(j,m) = mean(energy(currVal{m}));
                EnergyCompensated(j,m) = mean(energy(currValComp{m}));
            end
        end
        EnergyBatchMeansCompound(k,:,i,:) = EnergyCompound;
        EnergyBatchMeansCompensated(k,:,i,:) = EnergyCompensated;
    end
    % Calculate means and standard deviation over batches
    for m = 1:noSteppers
        ResultEnergyCompound.mean{i,m} = mean(EnergyBatchMeansCompound(:,i,m));
        ResultEnergyCompound.std{i,m} = std(EnergyBatchMeansCompound(:,i,m));
    end
\begin{verbatim}
ResultEnergyCompensated.mean\{i,m\} = mean(EnergyBatchMeansCompensated(:, :, m));
ResultEnergyCompensated.std\{i,m\} = std(EnergyBatchMeansCompensated(:, :, m));
end

disp('Done')

%% Plot energy drift dependent on time
%% Compound Poisson process
titleString = ['Parameters: \omega=' num2str(omega)...
', \lambda=' num2str(lambda) '. Batch size: '...
',num2str(batchSize) '. Number of batches: ' num2str(batchAmount) ''];
subTitleStrings = {['\alpha=' num2str(alpha) ', \eta=' num2str(eta)] ,... 
['\alpha=' num2str(alphaAlt) ', \eta=' num2str(etaAlt)] ,... 
['\alpha=' num2str(alpha) ', \eta=' num2str(etaAlt)] ,... 
['\alpha=' num2str(alphaAlt) ', \eta=' num2str(etaAlt)]};
legendPos = {nw nw nw nw ; 
se se nw nw ; 
se se nw nw ; 
se se nw nw ; 
se se nw nw ; 
se se nw nw ; 
se se nw nw ; 
se se nw nw ; 
se se nw nw ; 
se se nw nw ; 
se se nw nw ; 
se se nw nw ;

ymax = [30 10 10 10 10 10 10];
ylab = 'Energy';

figNames = {'Figs/EnergyTimeLBPOExplicitEuler',...
'Figs/EnergyTimeLBPOImplicitEuler',...
'Figs/EnergyTimeLBPOPartitionedEuler',...
'Figs/EnergyTimeLBPOThetaEuler',...
'Figs/EnergyTimeLBPOSSBEuler',...
'Figs/EnergyTimeLBPOCSSBEuler',...
'Figs/EnergyTimeLBPOTrig'};

hplot.val = false;
CIInfo.val = false;
CIInfo.confLevel = 0.95;
CIInfo.batchSize = batchSize;

% Add the information of the analytical solution
anSol = {exactEnergyMeanCompound(alpha,eta),exactEnergyMeanCompound(alphaAlt,eta),...
exactEnergyMeanCompound(alpha,etaAlt),exactEnergyMeanCompound(alphaAlt,etaAlt)};

% Plot it
FourPlotsInOne(ResultEnergyCompound,dt,T,titleString,...
subTitleStrings,legendPos,ymax,ylab,figNames,hplot,CIInfo,anSol);

%% Plot energy drift dependent on time
%% Compensated compound Poisson process
titleString = ['Parameters: \omega=' num2str(omega)...
', \lambda=' num2str(lambda) '. Batch size: '...
',num2str(batchSize) '. Number of batches: ' num2str(batchAmount) ''];
subTitleStrings = {['\alpha=' num2str(alpha) ', \eta=' num2str(eta)] ,... 
['\alpha=' num2str(alphaAlt) ', \eta=' num2str(etaAlt)] ,... 
['\alpha=' num2str(alpha) ', \eta=' num2str(etaAlt)] ,... 
['\alpha=' num2str(alphaAlt) ', \eta=' num2str(etaAlt)]};
legendPos = {nw nw nw nw ; 
se se nw nw ; 
se se nw nw ; 
se se nw nw ; 
se se nw nw ; 
se se nw nw ; 
se se nw nw ; 
se se nw nw ; 
se se nw nw ; 
se se nw nw ; 
se se nw nw ; 
se se nw nw ;

x0En = energy(x0);
\end{verbatim}
spareHeight = 0.1;
niceMax = (x0En + (alpha^2 + eta^2*lambda*distSqExp)*T/2)*(1 + spareHeight);

ymax = [35 niceMax 35 niceMax niceMax niceMax niceMax];
ylab = 'Energy';

figNames = {'Figs/EnergyTimeCompLBPOExplicitEuler',... 'Figs/EnergyTimeCompLBPOImplicitEuler',... 'Figs/EnergyTimeCompLBPOPartitionedEuler',... 'Figs/EnergyTimeCompLBPOThetaEuler',... 'Figs/EnergyTimeCompLBPOSSBEuler',... 'Figs/EnergyTimeCompLBPOCSSBEuler',... 'Figs/EnergyTimeCompLBPOTrig'};

hplot.val = false;
CIInfo.val = false;
CIInfo.confLevel = 0.95;
CIInfo.batchSize = batchSize;

anSolComp = ... {exactEnergyMeanCompensated(alpha,eta),exactEnergyMeanCompensated(alphaAlt,eta),... exactEnergyMeanCompensated(alpha,etaAlt),exactEnergyMeanCompensated(alphaAlt,etaAlt)};

% Plot it
FourPlotsInOne(ResultEnergyCompensated,dt,T,titleString,... subTitleStrings,legendPos,ymax,ylab,figNames,hplot,CIInfo,anSolComp);
% Plot legend positions
sw = 'SouthWest';
se = 'SouthEast';
w = 'NorthWest';
ne = 'NorthEast';

% Time information
range = 2:5;
dt = 2.^-range;
dtLength = length(dt);
T = 0.25;

% Poisson process information
lambda = 3;
% A distribution which gives a vector of random variables
dist = @(x) ones(x,1);
% Distribution information, expected value of X and expected value of X^2
distExp = 1;
distSqExp = 1;

% Model information
% Linear stochastic oscillator
x0 = [1;0];
omega = 3;
alpha = 0.6;
alphaAlt = 0;
wieVec = @(alpha) [0;alpha];
eta = 0.6;
etaAlt = 0;
poiVec = @(eta) [0;eta];
detMat = [0,1;-omega^2,0];
partMat = [0,0;-omega^2,0];
partMat2 = [0,1;0,0];

% Batch energy function
energy = @(X) 1/2*(omega^2*X(1:2:end-1).^2 + X(2:2:end).^2);
% Energy mean at time t for compound Poisson process
exactEnergyMeanCompound = @(alpha,eta) @(t) 1/2*(omega^2*x0(1)^2 + x0(2)^2 ... 
+ alpha^2*t + eta^2*lambda*distSqExp*t) ... 
+ (eta*lambda*distExp*x0(1) - eta^2*lambda^2/omega^2*distExp^2)*(cos(omega*t)-1) ... 
+ eta*lambda*distExp*x0(2)/omega*sin(omega*t);

% Energy mean at time t for compensated compound Poisson process
exactEnergyMeanCompensated = @(alpha,eta) @(t) ... 
1/2*(omega^2*x0(1)^2 + x0(2)^2 + alpha^2*t + eta^2*lambda*distSqExp*t);

% Stepper information
theta = 0.5;

% CCSB-Euler stepper info
drift = lambda*distExp;

% Steppers
Expl1 = @(alpha,eta) @(Xn,t,dW,dN,dt) ExplicitEulerStepperMatrix(Xn,dW,dN,dt,...
detMat,wieVec(alpha),poiVec(eta));
Impl1 = @(alpha,eta) @(Xn,t,dW,dN,dt) ImplicitEulerStepperMatrix(Xn,dW,dN,dt,...
detMat,wieVec(alpha),poiVec(eta));
Part1 = @(alpha,eta) @(Xn,t,dW,dN,dt) PartitionedEulerStepper(Xn,dW,dN,dt,...
partMat,partMat2,wieVec(alpha),poiVec(eta));
SSB1 = @(alpha,eta) @(Xn,t,dW,dN,dt) SSBExplicitEulerStepperMatrix(Xn,dW,dN,dt,...
detMat,wieVec(alpha),poiVec(eta));
CSSB1 = @(alpha,eta) @(Xn,t,dW,dN,dt) CSSBExplicitEulerStepperMatrix(Xn,dW,dN,dt,...
detMat,wieVec(alpha),poiVec(eta),drift);
Expl(alphaAlt, etaAlt), Impl(alphaAlt, etaAlt), Part(alphaAlt, etaAlt),...
SSB(alphaAlt, etaAlt), CSSB(alphaAlt, etaAlt)};

% Function information
% Compound case
exactEnCellCompound = cell(noSteppers, 1);
noSchemes = noSteppers/4;
exactEnCellCompound(1:noSchemes) = {exactEnergyMeanCompound(alpha, eta)};
exactEnCellCompound((noSchemes+1):2*noSchemes) = {exactEnergyMeanCompound(alphaAlt, eta)};

% Compensated compound case
exactEnCellCompensated = cell(noSteppers, 1);
exactEnCellCompensated(1:noSchemes) = {exactEnergyMeanCompensated(alpha, eta)};
exactEnCellCompensated((noSchemes+1):2*noSchemes) = {exactEnergyMeanCompensated(alphaAlt, eta)};

% Monte Carlo simulation information and memory
batchSize = 2^8;
batchAmount = 2^4;
ResultEnergyCompound.mean = cell(dtLength, noSteppers);
ResultEnergyCompound.std = cell(dtLength, noSteppers);
ResultEnergyCompensated.mean = cell(dtLength, noSteppers);
ResultEnergyCompensated.std = cell(dtLength, noSteppers);

% Loop over the different time steps
for i=1:dtLength
% Set up time information
currDt = dt(i);
t = 0:curreDt:T;
tlength = length(t);
% Initialize result storage, one for a CPoPr, one for a CCPoPr
EnergyCompound = zeros(tlength, noSteppers);
EnergyBatchMeansCompound = zeros(batchAmount, tlength, noSteppers);
EnergyCompensated = zeros(tlength, noSteppers);
EnergyBatchMeansCompensated = zeros(batchAmount, tlength, noSteppers);
% Loop over the number of batches
for k=1:batchAmount
% Initialize process storage
MonteCarloMemoryInitialization;
% Initialize stepper memory
% Standard currVal used for compound Poisson process
currValComp = cell(noSteppers, 1);
for m = 1:noSteppers
    currValComp{m} = repmat(x0, batchSize, 1);
end
% Set current time
currT = t(1);
% Loop over time
for j = 2:tlength
% Progress info
disp(['Dt: ' num2str((i-1)/dtLength*100, '%6.1f') ...
    ', Batch: ' num2str((k-1)/batchAmount*100, '%6.1f') ...
    ', Time: ' num2str((j-1)/tlength*100, '%6.1f'))
% Simulate the process steps
MonteCarloProcessSteps;
% Perform the approximate steps
for m = 1:noSteppers
    currVal{m} = steppers{m}(currVal{m}, currT, dW, dCompN, currDt);
    currValComp{m} = steppers{m}(currValComp{m}, currT, dW, dCompCompN, currDt);
end

% Calculate the next process values
currT = t(j);
W = W + dW;
N = N + dN;
CompN = CompN + dCompN;
CompCompN = CompCompN + dCompCompN;

% Compute errors at each time step
for m = 1:noSteppers
    % Only look at first coordinate error
    EnergyCompound(j,m) = mean(energy(currVal{m}));
    EnergyCompensated(j,m) = mean(energy(currValComp{m}));
end

% Calculate the next process values
currT = t(j);
W = W + dW;
N = N + dN;
CompN = CompN + dCompN;
CompCompN = CompCompN + dCompCompN;

% Compute errors at each time step
for m = 1:noSteppers
    % Only look at first coordinate error
    EnergyCompound(j,m) = mean(energy(currVal{m}));
    EnergyCompensated(j,m) = mean(energy(currValComp{m}));
end

EnergyBatchMeansCompound(k,:,:,:) = EnergyCompound;
EnergyBatchMeansCompensated(k,:,:,:) = EnergyCompensated;

% Calculate means and standard deviation over batches
for m = 1:noSteppers
    ResultEnergyCompound.mean{i,m} = mean(EnergyBatchMeansCompound(:,:,m));
    ResultEnergyCompound.std{i,m} = std(EnergyBatchMeansCompound(:,:,m));
    ResultEnergyCompensated.mean{i,m} = mean(EnergyBatchMeansCompensated(:,:,m));
    ResultEnergyCompensated.std{i,m} = std(EnergyBatchMeansCompensated(:,:,m));
end

disp('Done')

%% Extract the weak error from the energy information
%% Compound Poisson process
ResultEnergyWeakCompound.mean = ResultEnergyCompound.mean;
ResultEnergyWeakCompound.std = ResultEnergyCompound.std;

for i = 1:dtLength
    tVec = 0:dt(i):T;
    for j = 1:noSteppers
        ResultEnergyWeakCompound.mean{i,j} = abs(ResultEnergyWeakCompound.mean{i,j} - ...
            exactEnCellCompound{j}(tVec));
    end
end

%% Plot weak error dependent on time
%% Compound Poisson process
titleString = ['Parameters: \omega=' num2str(omega) ...'
    \lambda=' num2str(lambda) '. Batch size: ...'
    num2str(batchSize) '. Number of batches: ' num2str(batchAmount) '.'];
subTitleStrings = {'[\alpha=' num2str(alpha) ', \eta=' num2str(eta)] ','[\alpha=' num2str(alphaAlt) ', \eta=' num2str(etaAlt)] ','[\alpha=' num2str(alpha) ', \eta=' num2str(etaAlt)] ','[\alpha=' num2str(alphaAlt) ', \eta=' num2str(etaAlt)]'};
legendPos = {[nw nw nw nw];
    nw nw nw nw;
    nw nw nw nw;
    nw nw nw nw};
ymax = [2 2 2 2 2];
ylab = 'Weak error of energy';
figNames = {'Figs/EnergyTimeWeakErrorLBPOExplicitEuler',...
    'Figs/EnergyTimeWeakErrorLBPOImplicitEuler',...
    'Figs/EnergyTimeWeakErrorLBPOPartitionedEuler',...
    'Figs/EnergyTimeWeakErrorLBPOSSBEuler',...
    'Figs/EnergyTimeWeakErrorLBPOSSBEuler'};
\begin{verbatim}
        'Figs/EnergyTimeWeakErrorLBPOCSSBEuler'});

        hplot.val = false;
        CIInfo.val = true;
        CIInfo.confLevel = 0.95;
        CIInfo.batchSize = batchSize;

        % Plot it
        FourPlotsInOne(ResultEnergyWeakCompound,dt,T,titleString,...
        subTitleStrings,legendPos,ymax,ylab,figNames,hplot,CIInfo);

        % Plot weak error at time T
        % Compound Poisson process
        % Extract the last times
        tempSize = size(ResultEnergyWeakCompound.mean);
        endTimeResult.mean = cell(tempSize);
        endTimeResult.std = cell(tempSize);
        for i = 1:tempSize(1)
            for j = 1:tempSize(2)
                endTimeResult.mean{i,j} = ResultEnergyWeakCompound.mean{i,j}(end);
                endTimeResult.std{i,j} = ResultEnergyWeakCompound.std{i,j}(end);
            end
        end

        titleString = ['Parameters: \omega=' num2str(omega)...
            ', \lambda=' num2str(lambda) '. Batch size: '...
            ',num2str(batchSize) '. Number of batches: ' num2str(batchAmount) '.'];
        subTitleStrings = {{'\alpha=' num2str(alpha) ', \eta=' num2str(eta) }...
            {{'\alpha=' num2str(alphaAlt) ', \eta=' num2str(etaAlt) }...
            {{'\alpha=' num2str(alpha) ', \eta=' num2str(etaAlt) }...
            {{'\alpha=' num2str(alphaAlt) ', \eta=' num2str(etaAlt) ];
        legendInfo.legendNames = {'Explicit Euler','Implicit Euler','Partitioned',...
            'SSB','CSSB'};
        legendInfo.legendPos = {se se se se ;
            se se se ;
            se se se ;
            se se se ;
            se se se ;};
        ylab = 'Weak error of energy';

        figNames = {'Figs/EnergyWeakErrorLBPOExplicitEuler',...
            'Figs/EnergyWeakErrorLBPOImplicitEuler',...
            'Figs/EnergyWeakErrorLBPOPartitionedEuler',...
            'Figs/EnergyWeakErrorLBPOSSBEuler',...
            'Figs/EnergyWeakErrorLBPOCSSBEuler'};

        hplot.val = true;
        hplot.exponent = [2, 1, 0.5];
        hplot.translation = [0, 0, 0];
        hplot.legendInput = {\'$C_1\Delta t^2$', \'$C_2\Delta t$', \'$C_3\sqrt{\Delta t}$'};
        CIInfo.val = true;
        CIInfo.confLevel = 0.95;
        CIInfo.batchSize = batchSize;

        % Plot it
        FourPlotsInOne(endTimeResult,dt,T,titleString,...
            subTitleStrings,legendInfo,...
            ymax,ylab,figNames,hplot,CIInfo);

        % Extract the weak error from the energy information
        % Compensated compound Poisson process
        ResultEnergyWeakCompensated.mean = ResultEnergyCompensated.mean;
        ResultEnergyWeakCompensated.std = ResultEnergyCompensated.std;

        for i = 1:dtLength
            ...
\end{verbatim}
tVec = 0:dt(i):T;
for j = 1:noSteppers
    ResultEnergyWeakCompensated.mean{i,j} = abs(ResultEnergyWeakCompensated.mean{i,j} ... - exactEnCellCompensated{j}(tVec));
end

%% Plot weak error dependent on time
%% Compensated compound Poisson process
titleLabel = ['Parameters: \omega= ' num2str(omega) ... ', \lambda= ' num2str(lambda) '. Batch size: ' ... ,num2str(batchSize) '. Number of batches: ' num2str(batchAmount) '.'];
subTitleStrings = { ['\alpha= ' num2str(alpha) ', \eta= ' num2str(eta) ] ,... ['\alpha= ' num2str(alphaAlt) ', \eta= ' num2str(etaAlt) ] ,... ['\alpha= ' num2str(alpha) ', \eta= ' num2str(etaAlt) ] ,... ['\alpha= ' num2str(alphaAlt) ', \eta= ' num2str(etaAlt) ]};
legendPos = {nw nw nw nw ; nw nw nw nw ; nw nw nw nw ; nw nw nw nw }; ymax = [3 2 3 2 1]; ylab = 'Weak error of energy';
figNames = { 'Figs/EnergyTimeCompWeakErrorLBPOExplicitEuler',... 'Figs/EnergyTimeCompWeakErrorLBPOImplicitEuler',... 'Figs/EnergyTimeCompWeakErrorLBPOPartitionedEuler',... 'Figs/EnergyTimeCompWeakErrorLBPOSSBEuler',... 'Figs/EnergyTimeCompWeakErrorLBPOCSSBEuler'};
hplot.val = false;
CIInfo.val = true;
CIInfo.confLevel = 0.95;
CIInfo.batchSize = batchSize;
% Plot it
FourPlotsInOne(ResultEnergyWeakCompensated,dt,T,titleLabel,... subTitleStrings,legendPos,ymax,ylab,figNames,hplot,CIInfo);

%% Plot weak error at time T
%% Compensated compound Poisson process
% Extract the last times
endTimeResultCompensated.mean = cell(tempSize); endTimeResultCompensated.std = cell(tempSize);
for i = 1:tempSize(1)
    for j = 1:tempSize(2)
        endTimeResultCompensated.mean{i,j} = ResultEnergyWeakCompensated.mean{i,j}(end);
        endTimeResultCompensated.std{i,j} = ResultEnergyWeakCompensated.std{i,j}(end);
    end
end
titleLabel = ['Parameters: \omega= ' num2str(omega) ... ', \lambda= ' num2str(lambda) '. Batch size: ' ... ,num2str(batchSize) '. Number of batches: ' num2str(batchAmount) '.'];
subTitleStrings = { ['\alpha= ' num2str(alpha) ', \eta= ' num2str(eta) ] ,... ['\alpha= ' num2str(alphaAlt) ', \eta= ' num2str(etaAlt) ] ,... ['\alpha= ' num2str(alpha) ', \eta= ' num2str(etaAlt) ] ,... ['\alpha= ' num2str(alphaAlt) ', \eta= ' num2str(etaAlt) ]};
legendInfo.legendNames = { 'Explicit Euler','Implicit Euler','Partitioned','SSB','CSSB'};
legendInfo.legendPos = {se se se se ; se se se se ; se se se se ; se se se se ; se se se se ; se se se se ; se se se se ; se se se se};
ylab = 'Weak error of energy';

figNames = {'Figs/EnergyWeakErrorCompLBPOExplicitEuler', ...
            'Figs/EnergyWeakErrorCompLBPOImplicitEuler', ...
            'Figs/EnergyWeakErrorCompLBPOPartitionedEuler', ...
            'Figs/EnergyWeakErrorCompLBPOSSBEuler', ...
            'Figs/EnergyWeakErrorCompLBPOCSSBEuler'};

hplot.val = true;
hplot.exponent = [2, 1, 0.5];
hplot.translation = [0, 0, 0];
CIInfo.val = true;
CIInfo.confLevel = 0.95;
CIInfo.batchSize = batchSize;

% Plot it
FourPlotsInOne(endTimeResultCompensated, dt, T, titleString, ...
               subTitleStrings, legendInfo, ymax, ylab, figNames, hplot, CIInfo);

% Plot legend positions
sw = 'SouthWest';
se = 'SouthEast';
nw = 'NorthWest';ne = 'NorthEast';

% Time information
range = 2:5;
dt = 2.^-range;
dtLength = length(dt);
T = 0.25;

% Poisson process information
lambda = 3;

% A distribution which gives a vector of random variables
dist = @(x) ones(x,1);

% Distribution information, expected value of X and expected value of X^2
distExp = 1;
distSqExp = 1;

% Model information
% Linear stochastic oscillator
x0 = [1;0];
omega = 3;
alpha = 0.6;
alphaAlt = 0;
wieVec = @(alpha) [0;alpha];
eta = 0.6;
etAlt = 0;

% Poisson process
poiVec = @(eta) [0;eta];
detMat = [0,1;-omega^2,0];
partMat = [0,0;-omega^2,0];
partMat2 = [0,1;0,0];

% Batch energy function
energy = @(X) 1/2*(omega^2*X(1:2:(end-1)).^2 + X(2:2:end).^2);

% Energy mean at time t for compound Poisson process
exactEnergyMeanCompound = @(alpha,eta) @(t) 1/2*(omega^2*x0(1).^2 + x0(2).^2 ... 
                        + alpha^2*t + eta^2*lambda*distSqExp*t) ... 
                        + (eta*lambda*distExp*x0(1) - eta^2*lambda^2/omega^2*distExp^2)*(cos(omega*t)-1) ... 
                        + eta*lambda*distExp*x0(2)/omega*sin(omega*t);

% Energy mean at time t for compensated compound Poisson process
exactEnergyMeanCompensated = @(alpha,eta) @(t) ...
\[ 1/2 \{ \omega^2 \cdot x(0)^2 + x(2)^2 + \alpha^2 \cdot t + \eta^2 \cdot \lambda \cdot \text{distSqExp} \cdot t \}; \]

% Stepper information
% Theta-Euler stepper info
\theta = 0.5;

% CCSB-Euler stepper info
drift = \lambda \cdot \text{distExp};

% Steppers
Thet = @(alpha,eta) @(Xn,t,dW,dN,dt) ThetaEulerStepperMatrix(Xn,dW,dN,dt,...
\text{detMat}, \text{wieVec}(alpha), \text{poiVec}(eta), \theta);

% Combine into a cell array
steppers = \{Thet(alpha,eta), Thet(alphaAlt,eta),...;\}
noSteppers = length(steppers);

% Function information
% Compound case
exactEnCellCompound = cell(noSteppers,1);
noSchemes = noSteppers/4;
exactEnCellCompound(1:noSchemes) = \{exactEnergyMeanCompound(alpha,eta)\};
exactEnCellCompound((noSchemes+1):2\cdot noSchemes) = \{exactEnergyMeanCompound(alphaAlt,eta)\};
exactEnCellCompound((2\cdot noSchemes+1):3\cdot noSchemes) = \{\}
exactEnCellCompound((3\cdot noSchemes+1):4\cdot noSchemes) = \{exactEnergyMeanCompound(alpha,etaAlt)\};

% Compensated compound case
exactEnCellCompensated = cell(noSteppers,1);
exactEnCellCompensated(1:noSchemes) = \{exactEnergyMeanCompensated(alpha,eta)\};
exactEnCellCompensated((noSchemes+1):2\cdot noSchemes) = \{\}
exactEnCellCompensated((2\cdot noSchemes+1):3\cdot noSchemes) = \{\}
exactEnCellCompensated((3\cdot noSchemes+1):4\cdot noSchemes) = \{exactEnergyMeanCompensated(alpha,etaAlt)\};

% Monte Carlo simulation information and memory
batchSize = 2^{16};
batchAmount = 2^{12};
ResultEnergyCompound.mean = cell(dtLength,noSteppers);
ResultEnergyCompound.std = cell(dtLength,noSteppers);
ResultEnergyCompensated.mean = cell(dtLength,noSteppers);
ResultEnergyCompensated.std = cell(dtLength,noSteppers);

% Loop over the different time steps
for i=1:dtLength
% Set up time information
currDt = dt(i);
t = 0:currDt:T;
tLength = length(t);
% Initialize result storage, one for a CPoPr, one for a CCPoPr
EnergyCompound = zeros(tLength,noSteppers);
EnergyCompound(1,:) = ones(1,noSteppers)\cdot energy(x0);
EnergyBatchMeansCompound = zeros(batchAmount,tLength,noSteppers);
EnergyCompensated = zeros(tLength,noSteppers);
EnergyCompensated(1,:) = ones(1,noSteppers)\cdot energy(x0);
EnergyBatchMeansCompensated = zeros(batchAmount,tLength,noSteppers);
% Loop over the number of batches
for k=1:batchAmount
% Initialize process storage
MonteCarloMemoryInitialization;
% Initialize stepper memory
% Standard currVal used for compound Poisson process
currValComp = cell(noSteppers,1);
for m = 1:noSteppers
currValComp[m] = repmat(x0,batchSize,1);
end
% Set current time
currT = t(1);
% Loop over time
for j = 2:tlength
  % Progress info
  disp(sprintf('Dt: %.1f, Batch: %.1f, Time: %.1f',
               (i-1)/dtLength*100,(k-1)/batchAmount*100,(j-1)/tlength*100))
  % Simulate the process steps
  MonteCarloProcessSteps;
  % Perform the approximate steps
  for m = 1:noSteppers
    currVal{m} = steppers{m}(currVal{m},currT,dW,dCompN,currDt);
    currValComp{m} = steppers{m}(currValComp{m},currT,dW,dCompCompN,currDt);
  end
  % Calculate the next process values
  currT = t(j);
  W = W + dW;
  N = N + dN;
  CompN = CompN + dCompN;
  CompCompN = CompCompN + dCompCompN;
  % Compute errors at each time step
  for m = 1:noSteppers
    % Only look at first coordinate error
    EnergyCompound(j,m) = mean(energy(currVal{m}));
    EnergyCompensated(j,m) = mean(energy(currValComp{m}));
  end
end
EnergyBatchMeansCompound(k,:,:,:) = EnergyCompound;
EnergyBatchMeansCompensated(k,:,:,:) = EnergyCompensated;
end
% Calculate means and standard deviation over batches
for m = 1:noSteppers
  ResultEnergyCompound.mean{i,m} = mean(EnergyBatchMeansCompound(:,i,m));
  ResultEnergyCompound.std{i,m} = std(EnergyBatchMeansCompound(:,i,m));
  ResultEnergyCompensated.mean{i,m} = mean(EnergyBatchMeansCompensated(:,i,m));
  ResultEnergyCompensated.std{i,m} = std(EnergyBatchMeansCompensated(:,i,m));
end
end
% Extract the weak error from the energy information
% Compound Poisson process
ResultEnergyWeakCompound.mean = ResultEnergyCompound.mean;
ResultEnergyWeakCompound.std = ResultEnergyCompound.std;
for i = 1:dtLength
  tVec = 0:dt(i):T;
  for j = 1:noSteppers
    ResultEnergyWeakCompound.mean{i,j} = abs(ResultEnergyWeakCompound.mean{i,j} - ...
      exactEnCellCompound{j}(tVec));
  end
end
% Plot weak error dependent on time
% Compound Poisson process
titleString = ['Parameters: \omega=' num2str(omega)'
                 \lambda=' num2str(lambda) ', \alpha=' num2str(alpha) ', \eta=' num2str(eta) ];
subTitleStrings = {'\alpha=' num2str(alphaAlt) ', \eta=' num2str(etaAlt) };
legendPos = {nw nw nw nw};
ymax = [0.01];
ylab = 'Weak error of energy';
figNames = {'Figs/EnergyTimeWeakErrorLBPOThetaEuler'};

hplot.val = false;
CIInfo.val = true;
CIInfo.confLevel = 0.95;
CIInfo.batchSize = batchSize;

% Plot it
FourPlotsInOne(ResultEnergyWeakCompound,dt,T,titleString,...
subtitleStrings,legendPos,ymax,ylab,figNames,hplot,CIInfo);

%% Plot weak error at time T
%% Compound Poisson process
% Extract the last times
tempSize = size(ResultEnergyWeakCompound.mean);
endTimeResult.mean = cell(tempSize);
endTimeResult.std = cell(tempSize);
for i = 1:tempSize(1)
    for j = 1:tempSize(2)
        endTimeResult.mean{i,j} = ResultEnergyWeakCompound.mean{i,j}(end);
        endTimeResult.std{i,j} = ResultEnergyWeakCompound.std{i,j}(end);
    end
end

titleString = ['Parameters: \omega=', num2str(omega), ' Lambda: ', num2str(lambda), ' Batch size: ', num2str(batchSize), ' Number of batches: ', num2str(batchAmount), ' '];
subTitleStrings = {
    ['\alpha=', num2str(alpha), ' \eta=', num2str(eta)] ,
    ['\alpha=', num2str(alphaAlt), ' \eta=', num2str(eta)] ,
    ['\alpha=', num2str(alpha), ' \eta=', num2str(etaAlt)] ,
    ['\alpha=', num2str(alphaAlt), ' \eta=', num2str(etaAlt)]
};
legendInfo.legendNames = {'Theta Euler, $\theta=0.5$'};
legendInfo.legendPos = {se se se nw}
ylab = 'Weak error of energy';
figNames = {'Figs/EnergyWeakErrorLBPOThetaEuler'};

hplot.val = true;
hplot.exponent = [2, 1, 0.5];
hplot.translation = [0, 0, 0];
hplot.legendInput = {'$C_{1}\Delta t^2$', '$C_{2}\Delta t$', '$C_{3}\sqrt{\Delta t}$'};
CIInfo.val = true;
CIInfo.confLevel = 0.95;
CIInfo.batchSize = batchSize;

% Plot it
FourPlotsInOne(endTimeResult,dt,T,titleString,...
subtitleStrings,legendInfo,...
ymax,ylab,figNames,hplot,CIInfo);

%% Extract the weak error from the energy information
%% Compensated compound Poisson process
ResultEnergyWeakCompensated.mean = ResultEnergyCompensated.mean;
ResultEnergyWeakCompensated.std = ResultEnergyCompensated.std;

for i = 1:dtLength
    tVec = 0:dt(i):T;
    for j = 1:noSteppers
        ResultEnergyWeakCompensated.mean{i,j} = abs(ResultEnergyWeakCompensated.mean{i,j} ... - exactEnCellCompensated{j}(tVec));
    end
end

%% Plot weak error dependent on time
%% Compensated compound Poisson process

titleString = ['Parameters: \omega=' num2str(omega)...',' lambda=' num2str(lambda) '.  Batch size: '...','num2str(batchSize) '.  Number of batches: ' num2str(batchAmount) '.'];

subTitleStrings = {
    ['\alpha=' num2str(alpha) ', \eta=' num2str(eta) ] ,
    ['\alpha=' num2str(alphaAlt) ', \eta=' num2str(etaAlt) ] ,
    ['\alpha=' num2str(alpha) ', \eta=' num2str(etaAlt) ] ,
    ['\alpha=' num2str(alphaAlt) ', \eta=' num2str(etaAlt) ]
};

legendPos = {nw nw nw nw};
ymax = [0.025];
ylab = 'Weak error of energy';

figNames = {'Figs/EnergyTimeCompWeakErrorLBPOThetaEuler'};

hplot.val = false;
CIInfo.val = true;
CIInfo.confLevel = 0.95;
CIInfo.batchSize = batchSize;

% Plot it
FourPlotsInOne(ResultEnergyWeakCompensated,dt,T,titleString,...
    subTitleStrings,legendPos,ymax,ylab,figNames,hplot,CIInfo);

%% Plot weak error at time T
%% Compensated compound Poisson process
% Extract the last times
tempSize = size(ResultEnergyWeakCompensated.mean);
endTimeResultCompensated.mean = cell(tempSize);
endTimeResultCompensated.std = cell(tempSize);
for i = 1:tempSize(1)
    for j = 1:tempSize(2)
        endTimeResultCompensated.mean{i,j} = ResultEnergyWeakCompensated.mean{i,j}{end};
        endTimeResultCompensated.std{i,j} = ResultEnergyWeakCompensated.std{i,j}{end};
    end
end

% Plot it
FourPlotsInOne(endTimeResultCompensated,dt,T,titleString,...
    subTitleStrings,legendPos,ylab,figNames,hplot,CIInfo);
% Plot legend positions
sw = 'SouthWest';
se = 'SouthEast';
nw = 'NorthWest';
ne = 'NorthEast';

% Time information
range = 2:5;
dt = 2.^-range;
dtLength = length(dt);
T = 0.25;

% Poisson process information
lambda = 3;

% A distribution which gives a vector of random variables
dist = @(x) ones(x,1);

% Distribution information, expected value of X and expected value of X^2
distExp = 1;
distExp2 = 1;

% Model information

% Linear stochastic oscillator
x0 = [1;0];
omega = 3;
alpha = 0.6;
alphaAlt = 0;
wieVec = @(alpha) [0;alpha];
et = 0.6;
etAlt = 0;

% poeVec = @(eta) [0;eta];
detMat = [0,1;-omega^2,0];
partMat = [0,0;-omega^2,0];
partMat2 = [0,1;0,0];

% Batch energy function
energy = @(X) 1/2*(omega^2*X(1:2:(end-1)).^2 + X(2:2:end).^2);

% Energy mean at time t for compound Poisson process
exactEnergyMeanCompound = @(alpha,eta) @(t) 1/2*(omega^2*x0(1).^2 + x0(2).^2 ...
 + alpha^2*t + eta^2*lambda*distExp+t)...
 + (eta*lambda*distExp*x0(1) - eta^2*lambda^2/omega^2*distExp^2)*(cos(omega*t)-1) ...
 + eta*lambda*distExp*x0(2)/omega*sin(omega*t);

% Stepper information

% Theta-Euler stepper info
theta = 0.5;

% CCSB-Euler stepper info
drift = lambda*distExp;

% Steppers
trigScheme = @(alpha,eta) @(Xn,t,dW,dN,dt) TrigStepper(Xn,dW,dN,dt,alpha,eta,omega);

% Combine into a cell array
steppers = {trigScheme(alpha,eta),trigScheme(alphaAlt,eta),...
 trigScheme(alpha,etaAlt),trigScheme(alphaAlt,etaAlt)};

% Function information (Compound only)
effectEnCellCompound = cell(noSteppers,1);

% Monte Carlo simulation information and memory
batchSize = 2^17;
bachAmount = 2^15;
ResultEnergyCompound = cell(dtLength,noSteppers);

% Loop over the different time steps
for i=1:dtLength
    % Set up time information
    currDt = dt(i);
    t = 0:currDt:T;
    tLength = length(t);
    % Initialize result storage, one for a CPoPr, one for a CCPoPr
    EnergyCompound = zeros(tLength,noSteppers);
    EnergyCompound(1,:) = ones(1,noSteppers)*energy(x0);
    EnergyBatchMeansCompound = zeros(batchAmount,tLength,noSteppers);
    % Loop over the number of batches
    for k=1:batchAmount
        % Initialize process storage
        MonteCarloMemoryInitialization;
        % Initialize stepper memory
        % Standard currVal used for compound Poisson process
        currValComp = cell(noSteppers,1);
        for m = 1:noSteppers
            currValComp{m} = repmat(x0,batchSize,1);
        end
        % Set current time
        currT = t(1);
        % Loop over time
        for j = 2:tLength
            % Progress info
            disp(['Dt: ' num2str((i-1)/dtLength*100,'%6.1f') ... '
                  Batch: ' num2str((k-1)/batchAmount*100,'%6.1f') ... '
                  Time: ' num2str((j-1)/tLength*100,'%6.1f')])
            % Simulate the process steps
            MonteCarloProcessSteps;
            % Perform the approximate steps
            for m = 1:noSteppers
                currVal{m} = steppers{m}(currVal{m},currT,dW,dCompN,currDt);
            end
            % Calculate the next process values
            currT = t(j);
            W = W + dW;
            N = N + dN;
            CompN = CompN + dCompN;
            CompCompN = CompCompN + dCompCompN;
            % Compute errors at each time step
            for m = 1:noSteppers
                % Only look at first coordinate error
                EnergyCompound(j,m) = mean(energy(currVal{m}));
            end
            % Calculate means and standard deviation over batches
            for m = 1:noSteppers
                ResultEnergyCompound.mean{i,m} = mean(EnergyBatchMeansCompound(:,:,m));
                ResultEnergyCompound.std{i,m} = std(EnergyBatchMeansCompound(:,:,m));
            end
        end
    end
    disp('Done')
    % Extract the weak error from the energy information
    % Compound Poisson process
    ResultEnergyWeakCompound.mean = ResultEnergyCompound.mean;
    ResultEnergyWeakCompound.std = ResultEnergyCompound.std;
    for i = 1:dtLength
        tVec = 0:dt(i):T;
        for j = 1:noSteppers
            exactEnCellCompound{j}(tVec) = abs(ResultEnergyWeakCompound.mean{i,j} - ...
Plot weak error dependent on time

Compound Poisson process

Parameters: \( \omega = \text{num2str}(\omega) \), \( \lambda = \text{num2str}(\lambda) \), Batch size: \( \text{num2str}(\text{batchSize}) \), Number of batches: \( \text{num2str}(\text{batchAmount}) \).

\[ [\alpha = \text{num2str}(\alpha) , \eta = \text{num2str}(\eta)] , \]
\[ [\alpha = \text{num2str}(\alphaAlt) , \eta = \text{num2str}(\eta)] , \]
\[ [\alpha = \text{num2str}(\alpha) , \eta = \text{num2str}(\etaAlt)] , \]
\[ [\alpha = \text{num2str}(\alphaAlt) , \eta = \text{num2str}(\etaAlt)] \}

legendPos = {nw nw nw nw};

\[ \text{ymax} = [0.3]; \]
ylab = 'Weak error of energy';

figNames = {'Figs/EnergyTimeWeakErrorLBPOTrig'};

hplot.val = false;
CIInfo.val = true;
CIInfo.confLevel = 0.95;
CIInfo.batchSize = batchSize;

% Plot it
FourPlotsInOne(ResultEnergyWeakCompound,dt,T,titleString,...
subTitleStrings,legendPos,ymax,ylab,figNames,hplot,CIInfo);

Plot weak error at time T

tempSize = size(ResultEnergyWeakCompound.mean);
endTimeResult.mean = cell(tempSize);
endTimeResult.std = cell(tempSize);
for i = 1:tempSize(1)
  for j = 1:tempSize(2)
    endTimeResult.mean{i,j} = ResultEnergyWeakCompound.mean{i,j}(end);
    endTimeResult.std{i,j} = ResultEnergyWeakCompound.std{i,j}(end);
  end
end

titleString = ['Parameters: \( \omega = \text{num2str}(\omega) \), \( \lambda = \text{num2str}(\lambda) \), Batch size: \( \text{num2str}(\text{batchSize}) \), Number of batches: \( \text{num2str}(\text{batchAmount}) \).

[\alpha = \text{num2str}(\alpha) , \eta = \text{num2str}(\eta)] , \]
[\alpha = \text{num2str}(\alphaAlt) , \eta = \text{num2str}(\eta)] , \]
[\alpha = \text{num2str}(\alpha) , \eta = \text{num2str}(\etaAlt)] , \]
[\alpha = \text{num2str}(\alphaAlt) , \eta = \text{num2str}(\etaAlt)] \}

legendInfo.legendNames = {'Trigonometric'};
legendInfo.legendPos = {se se ne ne};
ylab = 'Weak error of energy';

figNames = {'Figs/EnergyWeakErrorLBPOTrig'};

hplot.val = true;
hplot.exponent = [2, 1, 0.5];
hplot.translation = [0, 0, 0];
hplot.legendInput = {'$C_1\Delta t^2$', '$C_2\Delta t$', '$C_3\sqrt{\Delta t}$'};
CIInfo.val = true;
CIInfo.confLevel = 0.95;
CIInfo.batchSize = batchSize;
% Plot it
FourPlotsInOne(endTimeResult,dt,T,titleString,subTitleStrings,legendInfo,...
ymax,ylab,figNames,hplot,CIInfo);
% Time information
range = 3:4:15;
dt = 2.^-range;
dtLength = length(dt);
T = 1/8;
% Poisson process information
lambda = 48;
dist = @(x) randn(x,1)+0.5;
% Expected value of the distribution
distExp = 1;
% Model information
% Stochastic geometric motion
x0 = 1;
mu = 0.2;
muFun = @(t,x) mu*x;
sigma1 = 0;
sigma2 = 0.05;
sigma3 = 0.2;
sigma4 = 0.8;
sigmaFun = @(sigma) @(t,x) sigma*x;
sigmaFunDeriv = @(sigma) @(t,x) sigma;
etta = 0.8;
ettaFun = @(etta) @(t,x) etta*x;
% Stepper information
theta = 0.5;
ExplStep = @(sigma,etta) @(Xn,t,dW,dCN,dt) ... ExplicitEulerStepper(Xn,dW,dCN,dt,muFun,sigmaFun(sigma),ettaFun(etta));
ImplStep = @(sigma,etta) @(Xn,t,dW,dCN,dt) ... ImplicitEulerStepperGeom(Xn,dW,dCN,dt,mu,sigma,etta);
ThetaStep = @(sigma,etta) @(Xn,t,dW,dCN,dt) ... ThetaEulerStepperGeom(Xn,dW,dCN,dt,mu,sigma,etta,theta);
SSBStep = @(sigma,etta) @(Xn,t,dW,dCN,dt) ... SSBEulerStepperGeom(Xn,dW,dCN,dt,mu,sigma,etta);
CSSBStep = @(sigma,etta) @(Xn,t,dW,dCN,dt) ... CSSBEulerStepperGeom(Xn,dW,dCN,dt,mu,sigma,etta,distExp*lambda);
% Combine into a cell array
steppers = {ImplStep(sigma1,etta)
ImplStep(sigma2,etta)
ImplStep(sigma3,etta)
ImplStep(sigma4,etta)};
noSteppers = length(steppers);
% Function information
exactSol = @(sigma,etta) @(t,W,N,PoInc) GBPMSolution(W,N,t,mu,sigma,etta,PoInc,x0);
exactSolCell = {exactSol(sigma1,etta)
exactSol(sigma2,etta)
exactSol(sigma3,etta)
exactSol(sigma4,etta)};
% Monte Carlo simulation information and memory
batchSize = 2^10;
batchAmount = 2^6;
ResultStrong.mean = cell(dtLength,noSteppers);
ResultStrong.std = cell(dtLength,noSteppers);
% Loop over the different time steps
for i=1:dtLength
% Set up time information
currDt = dt(i);
t = 0:currDt:T;
tlength = length(t);
% Initialize result storage
strong = zeros(tlength,noSteppers);
strongErrorBatchMeans = zeros(batchAmount,tlength,noSteppers);
% Loop over the number of batches
for k=1:batchAmount
MonteCarloMemoryInitialization;
% Set current time
currT = t(1);
% Loop over time
for j = 2:tlength
    % Progress info
    disp(['Dt: ' num2str((i-1)/dtLength*100,'%6.1f') ... 
         ', Batch: ' num2str((k-1)/batchAmount*100,'%6.1f') ... 
         ', Time: ' num2str((j-1)/tlength*100,'%6.1f')])
    MonteCarloProcessSteps;
    % Perform the approximate steps
    for m = 1:noSteppers
        currVal{m} = steppers{m}(currVal{m},currT,dW,dCompN,currDt);
    end
    % Calculate the next process values
    currT = t(j);
    W = W + dW;
    N = N + dN;
    CompN = CompN + dCompN;
    CompCompN = CompCompN + dCompCompN;
    % Compute the error at each time step
    for m = 1:noSteppers
        strong(j,m) = mean(abs(currVal{m}-exactSolCell{m}(currT,W,N,NInc)));
    end
end
% Store the means
strongErrorBatchMeans(:,:,:) = strong;
% Calculate means and standard deviation over batches
for m = 1:noSteppers
    ResultStrong.mean{i,m} = mean(strongErrorBatchMeans(:,:,m));
    ResultStrong.std{i,m} = std(strongErrorBatchMeans(:,:,m));
end
%
% Plot strong convergence dependent on time

% Extract the last times
tempSize = size(ResultStrong.mean);

% Plot it
FourPlotsInOne(ResultStrong,dt,T,titleString,subTitleStrings,legendPos,... 
    ymax,ylab,figNames,hplot,CIInfo);

% Plot strong convergence at time T
% Extract the last times
tempSize = size(ResultStrong.mean);
endTimeResult.mean = cell(tempSize);
endTimeResult.std = cell(tempSize);
for i = 1:tempSize(1)
    for j = 1:tempSize(2)
        endTimeResult.mean{i,j} = ResultStrong.mean{i,j}(end);
        endTimeResult.std{i,j} = ResultStrong.std{i,j}(end);
    end
end

titleString = ['Parameters: \mu=' num2str(mu) ...
', \lambda=' num2str(lambda) '. Batch size: ' ...
',num2str(batchSize) '. Number of batches: ' num2str(batchAmount) '.
';
s subTitleStrings = {['\sigma=' num2str(sigma1) ', \eta=' num2str(eta)] , ...['\sigma=' num2str(sigma2) ', \eta=' num2str(eta)] , ...['\sigma=' num2str(sigma3) ', \eta=' num2str(eta)] , ...['\sigma=' num2str(sigma4) ', \eta=' num2str(eta)]};
sw = 'SouthWest';
se = 'SouthEast';
nw = 'NorthWest';
ne = 'NorthEast';
legendInfo.legendNames = {'Impl Euler'};
legendInfo.legendPos = {se se se se};
ylab = 'Strong error';
figNames = {'Figs/GBPMImplEulerNormDistOrderTest'};
hplot.val = true;
hplot.exponent = [2, 1, 0.5];
hplot.translation = [1, 1, 1];
hplot.legendInput = {'C_1 \Delta t^2$, 'C_2 \Delta t$', '$C_3 \sqrt{\Delta t}$'};
CIInfo.val = true;
CIInfo.confLevel = 0.95;
CIInfo.batchSize = batchSize;
% Plot it
FourPlotsInOne(endTimeResult,dt,T,titleString,subTitleStrings,legendInfo,...
ymax,ylab,figNames,hplot,CIInfo);
K. Code - Plot and print functions

Code 33. plotMonte.m

function plotMonte(inStruct)

% Instruct have the following members
% Input:
% The input will be a struct, inStruct, with the following fields:
% monteRes - A struct containing the means and standard deviations
% but with the assumption that dt-dependency is false
% dt - An array detailing what dt’s has been looked at
% T - Simulation end time
% confLevel - Confidence level to determine the C.I.
% batchAmount - The number of batches
% batchSize - The size of the batches
% oneDim - A boolean value which tells whether we have evaluated
% a function mapped onto a 1-dimensional space or not
% legendInput - The input to the legend of the plot in form of a cell
% exponent - An array detailing dt.^exponent in the support lines
% translation - An array detailing vertical translation of the s.l.
% figName - A string containing the save file name
% ylab - A string for the y-axis label
% Output:
% A figure containing a plot which also has been saved as Figname

% Calculate the percentile based on the confidence level
perc = abs(norminv((1-inStruct.confLevel)/2,0,1));

% Find out how many functions we're looking at
if inStruct.oneDim
    numFun = size(inStruct.monteRes.mean,2);
else
    numFun = size(inStruct.monteRes.mean,2); % Extraneous line
    l = length(inStruct.dt);
end

% Find out how many supporting lines we want
numLines = length(inStruct.exponent);

% Create figure
fig = figure(inStruct.figureInput);
hold on

% Plot our function means and the CI
for i=1:numFun
    tempMean = cell2mat(inStruct.monteRes.mean(:,i));
    tempStd = cell2mat(inStruct.monteRes.std(:,i));
    h = errorbar(inStruct.dt,tempMean,...
                 perc*tempStd/sqrt(inStruct.batchSize));
    h.Marker = 'o';
end

% Plot our support lines
for i=1:numLines
    plot(inStruct.dt,power(inStruct.dt,inStruct.exponent(i))*...
         10^inStruct.translation(i),'--')
end

ylabel(inStruct.ylab)
xlabel('$\Delta t$')
hold off
ax = get(fig,'CurrentAxes');
set(ax,'XScale','log','YScale','log')
LEG = legend(inStruct.legendInput);
set(LEG,'Interpreter','latex')
title(['Batch size: ' num2str(inStruct.batchSize) ...
       ', Number of batches: ' num2str(inStruct.batchAmount) '.'])
% Plot our functions
% Here it is assumed that the plots will be subplots. This can easily
% be changed to produce separate plots
fig = figure(inStruct.figureInput);

h = cell(numFun,1);
LEG = cell(numFun,1);
for j = 1:numFun
    legendArgs = cell(l,1);
    h(j) = subplot(1,numFun,j);
    hold on
    for i=1:l
        plot(0:inStruct.dt(i):inStruct.T,inStruct.monteRes.mean{i,j})
        legendArgs{i,1} = ['$\Delta t$: ' num2str(inStruct.dt(i),5)];
    end
    LEG(j) = legend(legendArgs{:},inStruct.legendInput{:},'interpreter','latex');
for i=1:l
    % Lower limit
    plot(0:inStruct.dt(i):inStruct.T,...
        inStruct.monteRes.mean{i,j}-...
        perc*inStruct.monteRes.std{i,j}/sqrt(inStruct.batchSize),'k')
    % Upper limit
    plot(0:inStruct.dt(i):inStruct.T,...
        inStruct.monteRes.mean{i,j}+...
        perc*inStruct.monteRes.std{i,j}/sqrt(inStruct.batchSize),'k')
end
ylabel(inStruct.ylab)
xlabel('t')
title(inStruct.functionName{j});
hold off
end

% Lower limit
plot(0:inStruct.dt(i):inStruct.T,...
    inStruct.monteRes.mean{i,j}-...
    perc*inStruct.monteRes.std{i,j}/sqrt(inStruct.batchSize),'k')
end

% Upper limit
plot(0:inStruct.dt(i):inStruct.T,...
    inStruct.monteRes.mean{i,j}+...
    perc*inStruct.monteRes.std{i,j}/sqrt(inStruct.batchSize),'k')
end

ylabel(inStruct.ylab)
xlabel('t')
title(inStruct.functionName{j});
hold off
end

% Lower limit
plot(0:inStruct.dt(i):inStruct.T,...
    inStruct.monteRes.mean{i,j}-...
    perc*inStruct.monteRes.std{i,j}/sqrt(inStruct.batchSize),'k')
end

% Upper limit
plot(0:inStruct.dt(i):inStruct.T,...
    inStruct.monteRes.mean{i,j}+...
    perc*inStruct.monteRes.std{i,j}/sqrt(inStruct.batchSize),'k')
end

end

suptitle([num2str(inStruct.confLevel*100) '% CI. Batch size: '...
    num2str(inStruct.batchSize) '. Number of batches: ' num2str(inStruct.batchAmount) ...
    '.'])
linkaxes([h{:}],'y');
end
pause(1)
printToPDF(fig,inStruct.figName)
end

function FourPlotsInOne(Results,dt,T,titleString,...
    subTitleStrings,legendInfo,ymax,ylab,figNames,hplot,CIInfo,exactSol)

% function FourPlotsInOne(Results,dt,T,titleString,...
%    subTitleStrings,legendInfo,ymax,ylab,figNames,hplot,CIInfo,exactSol)
% Plot a Monte Carlo result over [0,T] or at T. Saves these N plots to PDF,
% where N is the number of steppers used.
% Input:
% Results - A struct containing two fields, mean and std. These
% two fields are cells of the same size, size
% length(dt) times 4*N
% dt - A vector of different time step sizes
% T - The end time
% titleString - Cell array of length N containing the plot titles
% subTitleStrings - Cell array of size 4, containing the subplot titles
% legendInfo - A struct containing the legend strings and positions
% ymax - If plotting over the interval this is the y axis max
% ylab - A string containing the y-axis label
% figNames - A cell array containing the names of the files
% hplot - A struct determining whether plotting over the
% interval or at the end point. Contains support line
% information and legend
% CIInfo - Contains CI information
% exactSol - An alternative input of the

% Output:
% A number of PDF files at specified location
% If the exact solution is provided, plot it alongside the others
if nargin == 12
plotAnalytic = true;
else
plotAnalytic = false;
end

% If we want CI within our plots or if we have a end time plot
if CIInfo.val || hplot.val

% Calculate the percentile based on the confidence level
perc = abs(norminv((1-CIInfo.confLevel)/2,0,1));

% Extract the number of time steps and initialize memory related to it
noDifferentTimes = length(dt);
t = cell(noDifferentTimes,1);
legendEntry = cell(noDifferentTimes,1);
for k=1:noDifferentTimes
t{k} = 0:dt(k):T;
end

% Extract the number of figures saved
noDifferentSteppers = size(Results.mean,2)/4;
fig = cell(noDifferentSteppers,1);

% Produce the plots one at a time
for i=1:noDifferentSteppers
fig{i} = figure('Position', [100, 100, 1049, 895]);
LEG = cell(4,1);
% Loop over the subplots
for j=1:4
subplot(2,2,j)
currPlot = i + (j-1)*noDifferentSteppers;
% If we are plotting over the interval
if ~hplot.val
hold on
for k=1:noDifferentTimes
plot(t{k},Results.mean{k,c currPlot});
legendEntry{k} = ['$\Delta t$=' num2str(dt(k))];
end
if plotAnalytic
% Plot analytical solution
plot(t{k},exactSol{j}{t{k}});
LEG{j} = legend(legendEntry{:,},'Analytical solution',...
'Location',legendInfo{i,j});
else
LEG{j} = legend(legendEntry{:,},'Location',legendInfo{i,j});
end
% If we want CI, plot it
if CIInfo.val
for k=1:noDifferentTimes
plot(t{k},Results.mean{k,c currPlot} + ...
perc*Results.std{k,c currPlot}]/sqrt(CIInfo.batchSize),'k')
plot(t{k},Results.mean{k,c currPlot} - ...
perc*Results.std{k,c currPlot}]/sqrt(CIInfo.batchSize),'k')
end
end
set(LEG{j},'Interpreter','latex')
hold off
ylabel(ylab)
xlabel('t')
title(subTitleStrings{j})
if size(ymax,1)==4
    axis([0 T 0 ymax(j,i)])
else
    axis([0 T 0 ymax(i)])
end
else % If we are plotting at time T
    hold on
    h = errorbar(dt, [Results.mean(:, currPlot)], ... 
        perc* [Results.std(:, currPlot)]/sqrt(CIInfo.batchesize));
    h.Marker = 'o';
    % Plot support lines
    numLines = length(hplot.exponent);
    for m = 1:numLines
        plot(dt, power(dt, hplot.exponent(m)) * ... 
            10^hplot.translation(m), '--')
    end
    hold off
    % Set a loglog scale against the time steps
    set(gca, 'xscale', 'log');
    set(gca, 'yscale', 'log');
    ylabel(ylab)
xlabel('\Delta \ t')
    LEG{j} = legend(legendInfo.legendNames{i}, ... 
        hplot.legendInput{:, 'Location', legendInfo.legendPos{i, j}});
    set(LEG{j}, 'Interpreter', 'latex')
title(subTitleStrings{j})
end
grid on
end

% Pause to allow everything to render before saving
pause(1)
printToPDF(fig{i}, figNames{i})
end

---

**Code 35. printToPDF.m**

```matlab
function printToPDF(myHandle, name)

% Input:
% myHandle - The handle to the figure which is to be saved
% name - A string containing the file name

screen_size = get(0, 'ScreenSize');
origSize = get(myHandle, 'Position'); % grab original on screen size
set(myHandle, 'Position', [0 0 screen_size(3) screen_size(4) ]); %set to screen size
set(myHandle, 'PaperPositionMode', 'auto') %set paper pos for printing
print(myHandle, name, '-dpdf', '-r0') % save figure
set(myHandle, 'Position', origSize) %set back to original dimensions

set(myHandle, 'Units', 'Inches');
pos = get(myHandle, 'Position');
set(myHandle, 'PaperPositionMode', 'Auto', 'PaperUnits', 'Inches', 'PaperSize', [pos(3), ... 
pos(4)])
print(myHandle, name, '-dpdf', '-r0')
end
```
L. Code - Miscellaneous functions and scripts

**Code 36. GBPMSolution.m.**

```matlab
function [X] = GBPMSolution(W,N,t,mu,sigma,eta,increments,x0)
% Input:
% W - The Wiener at time t, a vector of size M
% N - The PoPr at time t, a vector of size M
% t - The time value
% mu - Deterministic parameter of GBPM
% sigma - Wiener diffusion parameter of GBPM
% eta - Poisson diffusion parameter of GBPM
% increments - Cell containing increments from the CPoPr
% x0 - A scalar containing the initial value
% Output:
% X - The resulting M solutions

batchSize = length(W);
tempProd = zeros(batchSize,1);
% Loop over the batch size
for i=1:batchSize
% Construct the product from the Poisson process
    tempProd(i) = prod(1+eta*increments{1:N(i)});
end
% Combine the information
X = x0*exp((mu-sigmaˆ2/2)*t+sigma*W).*tempProd;
end
```

**Code 37. MonteCarloMemoryInitialization.m.**

```matlab
% This scripts purpose is to save space, due to the repetition of the Monte Carlo simulations. This script assumes that the following variables are defined:
% batchSize, noSteppers, x0

% Initialize stepper memory
currVal = cell(noSteppers,1);
for m = 1:noSteppers
    currVal{m} = repmat(x0,batchSize,1);
end
```

161
% This scripts purpose is to save space, due to the repetition of the Monte Carlo simulations. This script assumes that the following variables are defined:
% currDt, batchSize, lambda, dist, distExp

% Simulate Wiener process step
dW = sqrt(currDt)*randn(batchSize,1);

% Simulate number of Poisson increments
dN = poissrnd(lambda*currDt,[batchSize,1]);

% Now simulate the compound and compensated compound Poisson proc
% Allocate increment memory
dNInc = cell(batchSize,1);
% The odds of multiple increments occurring during one time step decreases in likelihood as dt decreases. Deal with the case where dN == (0 | 1) first
oneInc = (dN == 1);
nOneInc = sum(oneInc);
dNInc(oneInc) = num2cell(dist(nOneInc));

% In case we have more than one increments in one time step,
% locate which indexes there are
indMoreInc = find(dN > 1);
nMoreInc = length(indMoreInc);
% Loop over each case where dN > 1
for m = 1:nMoreInc
    % Retrieve index
    currentIndex = indMoreInc(m);
    % Store the simulated variables
    dNInc(currentIndex) = dist(dN(currentIndex));
end
% Construct the compound Poisson process increment
% This step could be more efficient, perhaps through storing the cell arrays as a sparse matrix
for m = 1:batchSize
    % Store increments
    NInc{m} = [NInc{m} ; dNInc{m}];
    % Add increments to compound Poisson process
    dCompN(m) = sum(dNInc{m});
end
% Construct the compensated compound Poisson increment
dCompCompN = dCompN - lambda*distExp*currDt;