Model Research

Fengguang Wang

2017-12-15

Contents

1 Stochastic Calculus **3**
1.1 Stochastic process . . . 3
1.2 Stochastic Differential Equation . . . 4
1.2.1 Ito Integral . . . 4
1.2.2 Ito Lemma . . . 5
1.3 SDE Numerical Solution . . . 6
1.3.1 Numerical Schemes for SDEs . . . 6
1.3.2 Error and Convergence . . . 6
1.4 Models research . . . 9
1.4.1 General SDE . . . 9
1.4.2 Stochastic Volatility Model . . . 9

2 An Algebraic Formalisation **11**
2.1 Formalisation of Portfolio . . . 11
2.2 Simplifying via Discretisation . . . 11
2.3 Describing the Future . . . 11
2.4 Treat Prediction as Coalgebra . . . 12

Appendices **15**

A Implementation of Numeric Simulation **17**

Chapter 1

Stochastic Calculus

1.1 Stochastic process

Definition 1.1.1. For probability space $(\Omega, \mathcal{F}, P)$, index set I with total order, and measurable space (S, Σ), a function $X : I \times \Omega \to S$ is called a stochastic process [2]

Definition 1.1.2. A probability space $(\Omega, \mathcal{F}, P)$ equipped with filtration $(\mathcal{F}_i)_{i \in I}$ of σ-algebra $\mathcal{F}$, is a filtered probability space.

Definition 1.1.3. For a stochastic process $X : I \times \Omega \to S$ under a filtered probability space with filtration $(\mathcal{F}_i)_{i \in I}$, X is called an adapted process if $\forall i \in I, X_i : \Omega \to S$ is a $(\mathcal{F}_i, \Sigma)$-measurable function

Remark 1.1.1. Intuitively, an adapted process is a process that dose not look into the future. For example, a process which defined as the maximum of a stock price in the next 1 year is a non-adapted process. Because future information about stock price is required. On the other side, process only concerns about historic value is adapted process.

Example 1.1.1. Brownian bridge is defined as $B_t = (W_t | W_T = 0), \forall t \in [0, T]$, which is a conditioned Wiener process. And it can be decomposed to adapted part and future part: $B_t = \frac{T-t}{T} W_t - \frac{t}{T} W_{T-t}$, which means it is a non-adapted process.

Definition 1.1.4. For a stochastic process $X : I \times \Omega \to S$ under a filtered probability space with filtration $(\mathcal{F}_i)_{i \in I}$, X is called a predictable process if $\forall i \in I, \forall j \in I, j < i, X_i : \Omega \to S$ is a $(\mathcal{F}_j, \Sigma)$-measurable function

Remark 1.1.2. Predictable processes form a closed compact class which covers all left continuous adapted process.

Example 1.1.2. A deterministic process $X(t) = t$ is a predictable process.

Definition 1.1.5. A $\mathcal{F}_*$-adapted process X is a $\mathcal{F}_*$-martingale under probability measure P if $\forall t, E_P(|X_t|) < +\infty$ and $\forall s < t, X_s = E_P(X_t | \mathcal{F}_s)$

Definition 1.1.6. In a filtered probability space $(\Omega, \mathcal{F}, (\mathcal{F}_i)_{i\in I}, P)$, a random variable $\tau : \Omega \to I$ is called a stopping time if $\forall i \in I, \{\omega \in \Omega : \tau(\omega) \leq t\} \in \mathcal{F}_t$

Definition 1.1.7. For an adapted process $X : I \times \Omega \to S$ and a stopping time $\tau : \Omega \to I$, there is a stopped process X^τ s.t. $X^\tau_t(\omega) = X_{min(t,\tau(\omega))}(\omega)$

Definition 1.1.8. For a $(\mathcal{F}_i)_{i\in I}$-adapted process X, if there is a sequence of stopping times τ_k s.t.

- $P[\tau_k < \tau_{k+1}] = 1$
- $P[\tau_k \to +\infty \text{ as } k \to +\infty] = 1$,
- $\forall k$, its stopped processes X^{τ_k} is an $(\mathcal{F}_i)_{i\in I}$-martingale

X is called $(\mathcal{F}_i)_{i\in I}$-local martingale

Definition 1.1.9. A stochastic process X is a semimartingale if it can be decomposed to:

$$X = M + A$$

where M is a local martingale and A is a cadlag adapted process of locally bounded variation.

Remark 1.1.3. For continuous stochastic process this decomposition is unique.

Definition 1.1.10. In a filtered probability space $(\Omega, \mathcal{F}, (\mathcal{F}_i)_{i\in I}, P)$, An adapted process $X : I \times \Omega \to S$ s.t.

$$\forall s < t \in I, P(X_t \in A|\mathcal{F}_s) = P(X_t \in A|X_s)$$

is called Markov process.

Remark 1.1.4. Markov property can be considered as "memory-less", which means it does not concern about historical record and works in a stateless way.

1.2 Stochastic Differential Equation

1.2.1 Ito Integral

For a stochastic process $H : I \times \Omega \to S$ and integrator W, generally, there are two calculi to formalise integral of stochastic process: 1. Ito calculus 2. Stratonovich calculus

Definition 1.2.1. In Ito calculus, integral of stochastic process is defined as

$$\int_0^t H dW = \lim_{n\to+\infty} \sum_{[t_{i-1},t_i]\in\pi_n} H_{t_{i-1}}(W_{t_i} - W_{t_{i-1}})$$

where W is a semimartingale, and H is a cadlag adapted process of locally bounded variation

Definition 1.2.2. In Stratonovich calculus, integral of stochastic process is defined as

$$\int_0^t H \circ dW = \lim_{n\to+\infty} \sum_{[t_{i-1},t_i]\in\pi_n} \frac{H_{t_i} + H_{t_i-1}}{2}(W_{t_i} - W_{t_{i-1}})$$

where W is a semimartingale, and H is a cadlag adapted process of locally bounded variation

Remark 1.2.1. Stratonovich integral and Ito integral are inter-changeable.

$$\begin{aligned}\int_0^t H \circ dW &= \lim_{n\to+\infty} \sum_{[t_{i-1},t_i]\in\pi_n} \frac{H_{t_i} + H_{t_i-1}}{2}(W_{t_i} - W_{t_{i-1}}) \\ &= \lim_{n\to+\infty} \sum_{[t_{i-1},t_i]\in\pi_n} H_{t_i-1}(W_{t_i} - W_{t_{i-1}}) \\ &+ \lim_{n\to+\infty} \sum_{[t_{i-1},t_i]\in\pi_n} \frac{H_{t_i} - H_{t_i-1}}{2}(W_{t_i} - W_{t_{i-1}}) \\ &= \int_0^t H dW + \frac{1}{2}\langle H, W\rangle\end{aligned}$$

While Stratonovich calculus is mostly used in physics, in financial mathematics, Ito calculus is used by default.

Remark 1.2.2. Eventhough W could be any semimartingale, it is thought to be Winer process in general.

Definition 1.2.3. For integral equation of the form

$$X_{t+s} - X_t = \sum_{i<n} \int_t^{t+s} H_i dW_i$$

its differential form

$$dX = \sum_{i<n} H_i dW_i$$

is called SDE.

Remark 1.2.3. SDE can also be seemed as a continuous form of Fokker-Planck equation.

1.2.2 Ito Lemma

Lemma 1.2.1. Ito drift-diffustion process X_t s.t.

$$dX_t = \mu_t dt + \sigma_t dW_t$$

, where W_t is a Wiener process. For a function $f(t, X_t)$,

$$df(t, X_t) = (\frac{\partial f}{\partial t} + \mu_t \frac{\partial f}{\partial x} + \frac{\sigma^2}{2}\frac{\partial^2 f}{\partial^2 x})dt + \sigma_t \frac{\partial f}{\partial x} dB_t$$

Remark 1.2.4. Since it is easy to be deducted from Taylor series, proof is omitted.

1.3 SDE Numerical Solution

Since mosts of SDEs do not have a closed form, that is there is no analytical solution, numeric solution would the most realistic way to investigate them.

Basic idea of simulating numerical solution of a SDE is discretisation. For SDE $dX_t = a(t, X_t)dt + b(t, X_t)dW_t$, chop the interval $[0, T]$ into N grid which has $\Delta t = \frac{T}{N}$ interval. And via discretisation, continuous stochastic process becomes a discrete-time Markov chain.

Stochastic process is generated step by step with random number generate a float number between $[0, 1]$ and using their quantile function computing their value.

1.3.1 Numerical Schemes for SDEs

Here three types of schemes is examined and compared in aspect of convergence.

Euler-Maruyama

Definition 1.3.1. Euler-Maruyama approximation:

$$X_{n+1} = X_n + a(X_n)\Delta t + b(X_n)\Delta W_n$$

Millstein

Definition 1.3.2. Milstein approximation:

$$X_{n+1} = X_n + a(X_n)\Delta t + b(X_n)\Delta W_n + \frac{1}{2}b(X_n)b'(X_n)((\Delta W_n)^2 - \Delta t)$$

Runge-Kutta

Definition 1.3.3. Runge-Kutta approximation:

$$X_{n+1} = X_n + a(X_n)\Delta t + b(X_n)\Delta W_n + \frac{1}{2}(b(\hat{X}_n) - b(X_n))((\Delta W_n)^2 - \Delta t)(\Delta t)^{-1/2}$$

where

$$\hat{X}_n = X_n + a(X_n)\Delta t + b(X_n)\sqrt{\Delta t}$$

1.3.2 Error and Convergence

Errors of simulating is mainly caused by finiteness of state space, finiteness of simulation step, rounding-off. [3]

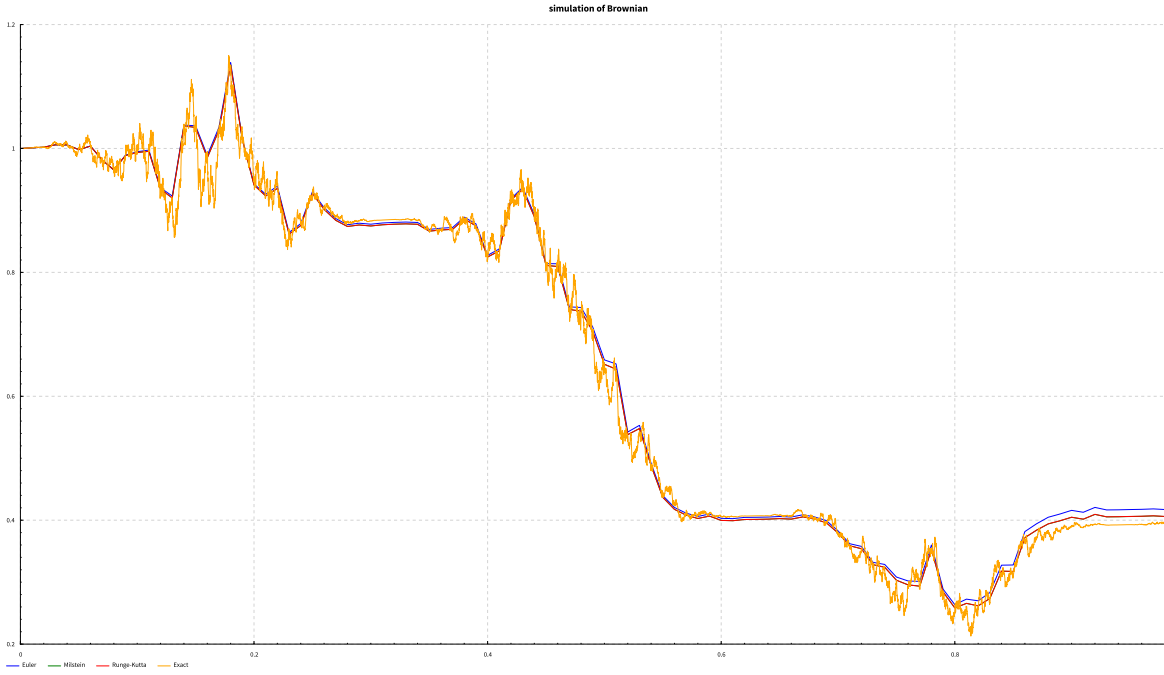
simulating result

for SDEs in form of $dX = \mu(X,t)dt + \sigma(X,t)dW$ take simulation using Euler-Maruyama scheme in a finer grid as the exact solution

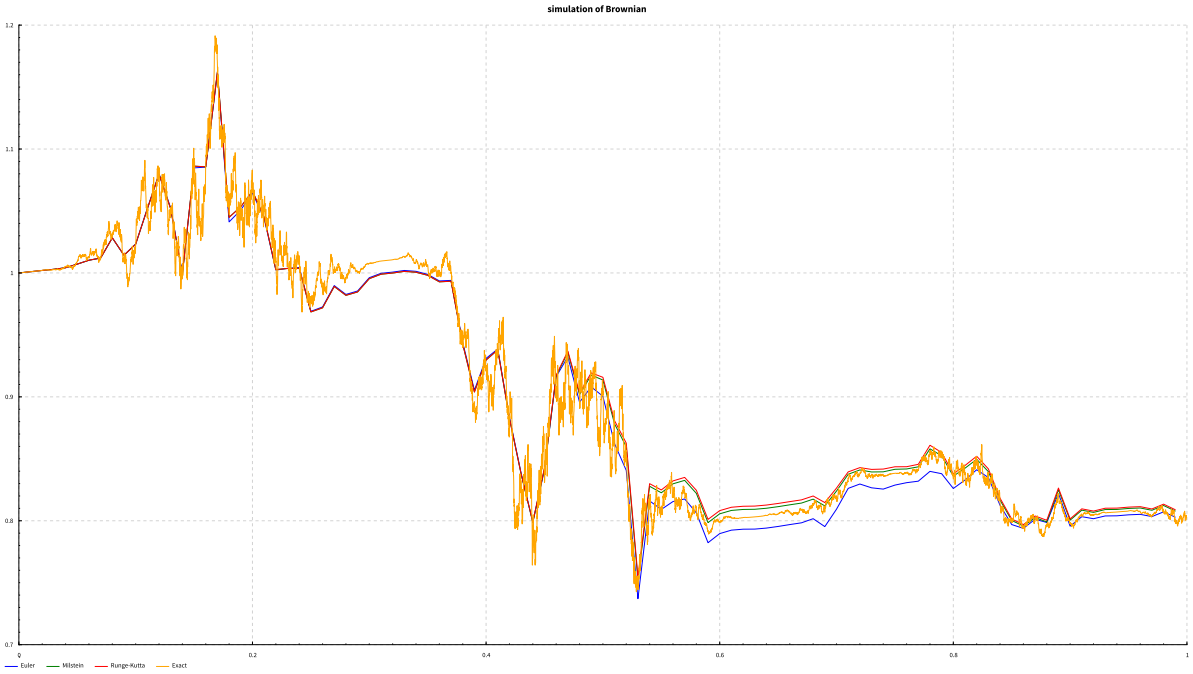
$\mu = 0.1 * X$, $\sigma = 0.8 * X$, $X_0 = 1$

$\mu = 0.1 * X$, $\sigma = 0.8 * sin(10 * t) * X$, $X_0 = 1$

$\mu = 0.1 * X$, $\sigma = 0.8 * sin(10 * t) * \sqrt{X}$, $X_0 = 1$



$\mu = 0.1 * X$, $\sigma = 0.8 * sin(10 * t) * sin(5 * t * X)$, $X_0 = 1$



1.4 Models research

1.4.1 General SDE

$$dX = \mu(X,t)dt + \sigma(X,t)dW$$

W is a vector of stochastic processes with correlation matrix M σ is a matrix

1.4.2 Stochastic Volatility Model

$$dS = \mu(S,t)Sdt + \nu^{\frac{1}{2}}SdW_1$$
$$d\nu = \alpha(\nu,t) + \beta(\nu,t)dW_2$$

Heston Model

$$dS = \mu(S,t)Sdt + \nu^{\frac{1}{2}}SdW_1$$
$$d\nu = \theta(\omega - \nu) + \xi\nu^{\frac{1}{2}}dW_2$$

Constant Elasticity of Variance Model

local volatility

$$dS = \mu Sdt + \sigma S^{\frac{\beta}{2}-1}$$

where $\beta \in [0,2]$

Chapter 2

An Algebraic Formalisation

2.1 Formalisation of Portfolio

Intuitively, ownership of Assets could be separated infinitely and hold as much as you can theoretically. And a portfolio can be constructed by finitely many assets. Hence, they could be considered as a vector space over R, which is an Abelian group equipped with scalar in real number field.

Definition 2.1.1. Suppose $(A_i)_{i\in I} is finite many types of assets, r_i \in R$, a portfolio $P = \Sigma_i r_i \bullet A_i$, and all Os form an vector space over R which satisfied responded properties.

2.2 Simplifying via Discretisation

An analytical solution of an SDE, which is a random process, is considered in continuous time in oder to get local linearity and scalability. However, in real world application we can always find a minimal time step, by seconds in stock market or by Plunk time in physical world. Hence, via discretisation, it do simplified our model with an acceptable trade-off.

An analytical solution of an SDE could be simplified to its numerical solution by take constant time step Δt. Then it becomes a discrete time Markov chain.

As for general stochastic process, if it is an adaptive process, after discretisation we could transformed it to an discretisastion of Markov chain by decomposition of its corelated "historical" random variables.

And what is exciting here is that, after discretisation we get a dynamic system which could be formalised by temporal logic with coalgebraic structure.

2.3 Describing the Future

Even thought we have discretised the time span, we have to deal with stochastic nature of real world.

Basic steps is by defining the future 1 step later, we can using a free structure to construct every future we could have.

Here comes Giry monad.

Giry monad is defined over the category of all measure space $Meas$ whose homophisms are measurable functions.

Giry monad is similar to Dirac function and general distribution. First we define the "integration":

$$T : (\Omega \to R) \to (\Omega \to R) \to R$$

$$T_f = \phi \mapsto \int_w f(w)\phi(w)dw$$

Hence, let $P(A) = (\Omega \to R) \to R \; \forall A, B \in Meas, F : (\Omega \to R)$,

$$P(F) : P(A) -> P(B)$$

Obviously, P(F) is an endofunctor over Meas.

Then (P, μ, η) is the Giry monad we want, [1]where

$$\mu : P \to P \to P$$

2.4 Treat Prediction as Coalgebra

Under discretisation, we make observation or prediction step by step, that is, provide a distribution to it. we can introduce functor $N = (\Delta t \times \Delta \phi)$ [4]

Bibliography

[1] Avi Pfeffer Norman Ramsey. Stochastic lambda calculus and monads of probability distribution. *POPL'02*, 2002.

[2] Steven E. Shreve. *Stochastic Calculus for Finance II Continuous-Time Models*. Springer, New York, USA, 2010.

[3] Sergey S. Stepanov. *Stochastic World*. Springer, Switzerland, 2013.

[4] Baltasar Tranc on y Widemann. Systematic construction of temperal logics for dynamical system via coalgebra. *arXiv*, 2017.

Appendices

Appendix A

Implementation of Numeric Simulation

```
module Levy where

import Statistics.Distribution
import Statistics.Distribution.Normal

import System.Random

realisation :: (RandomGen g, ContDistr d) => g -> d -> [Double]
realisation gen distr = quantile distr <$> randomRs (0, 1) gen

wiener :: Double -> NormalDistribution
wiener dt = normalDistr 0 (sqrt dt)

wieners :: (RandomGen g) => g -> Double -> [Double]
wieners g dt = realisation g $ wiener dt

{-# LANGUAGE TemplateHaskell #-}

module Scheme where

-- import Data.Vector (Vector(..), iterateN)
import Control.Lens
import Numeric.LinearAlgebra

data SDE = SDE -- Ytn tn
  { _a :: Double -> Double -> Double
  , _b :: Double -> Double -> Double
  , _b' :: Double -> Double -> Double -- d/dx
```

```
  , _x0 :: Double
  }

makeLenses ''SDE

type Scheme = SDE -> Double -> Double -> Double -> Double -> Double

schemeEuler :: Scheme
schemeEuler sde dt dw xn t = sum [a * dt, b * dw]
  where
    a = (_a sde) xn t
    b = (_b sde) xn t

schemeMilstein :: Scheme
schemeMilstein sde dt dw xn t =
  sum $ zipWith (*) [a, b, 0.5 * b * b'] [dt, dw, dw ^ 2 - dt]
  where
    a = (_a sde) xn t
    b = (_b sde) xn t
    b' = (_b' sde) xn t

schemeRungeKutta :: Scheme
schemeRungeKutta sde dt dw xn t =
  sum $ zipWith (*) [a, b, 0.5 * (br - b)] [dt, dw, (dw ^ 2 - dt) / sqrt dt]
  where
    a = (_a sde) xn t
    b = (_b sde) xn t
    br = (_b sde) r t
    r = xn + a * dt + b * sqrt dt

stepper :: SDE -> Scheme -> Double -> [Double] -> [(Double, Double)]
stepper sde scheme dt dws =
  (\(x, t, _) -> (t, x)) <$>
  iterate
    (\(xn, t, (dw:dwss)) -> (incr xn t dw + xn, t + dt, dwss))
    (x0, 0, dws)
  where
    x0 = _x0 sde
    incr xn t dw = scheme sde dt dw xn t

data SDEs = SDEs
  { _mu :: Vector Double -> Double -> Vector Double
  , _sigma :: Vector Double -> Double -> Matrix Double
  , _s0 :: Vector Double
  }
```

```
makeLenses ''SDEs

type Schemes
   = SDEs -> Double -> Vector Double -> Vector Double -> Double -> Vector Double

schemeEuler' :: Schemes
schemeEuler' sdes dt dw xn t =
  sum [((sdes ^. mu) xn t) & scale dt, ((sdes ^. sigma) xn t) #> dw]

stepper' ::
     SDEs -> Schemes -> Double -> [Vector Double] -> [(Double, Vector Double)]
stepper' sde schemes dt dws =
  (\(x, t, _) -> (t, x)) <$>
  iterate
    (\(xn, t, (dw:dwss)) -> (incr xn t dw + xn, t + dt, dwss))
    (x0, 0, dws)
  where
    x0 = _s0 sde
    incr xn t dw = schemes sde dt dw xn t

module Main where

import Levy
import Scheme
import Visual

import Statistics.Sample
import System.Random

import Control.Arrow
import Data.Fixed
import Data.Foldable
import Data.List
import Data.Monoid
import qualified Data.Vector as V

main :: IO ()
main
  -- traverse_ monteCarlo [0.1,0.2..2.0]
 = do
  traverse_ monteCarloStat [0.1,0.2 .. 2.0]
  -- monteCarlo 0.1
  return ()

precisionStudy
  -- traverse_ (simu sde1) [0..9]
```

```
 = do
  traverse_ (simu sde2) [0 .. 4]
  traverse_ (simu sde3) [5 .. 9]
  traverse_ (simu sde4) [10 .. 14]
  traverse_ (simu sde5) [15 .. 19]

simu sde i = do
  g <- newStdGen
  let rdat = realisation g (wiener $ 0.01 * 0.01)
  let dat1 = take 100 $ exempli1 sde $ listAdd 100 rdat
  let dat2 = take 100 $ exempli2 sde $ listAdd 100 rdat
  let dat3 = take 100 $ exempli3 sde $ listAdd 100 rdat
  let datE = take (100 * 100) $ exempliE sde $ rdat
  renderTrace
    ("trial_" <> show i)
    [ ("Euler", dat1)
    , ("Milstein", dat2)
    , ("Runge-Kutta", dat3)
    , ("Exact", datE)
    ]

monteCarlo beta = do
  let genFunc = do
        g <- newStdGen
        let timeInterval = 0.01 * 1
            rdat = realisation g (wiener $ timeInterval)
        return $
          take (100 * 1 * 10) $
          stepper (sdeCEV beta) schemeMilstein timeInterval rdat
  ds <- sequenceA $ replicate 100 genFunc
  print $ beta
  let dataSample = (V.fromList) <$> transpose (fmap snd <$> ds)
  let timeGrid = fst <$> head ds
  let meanData = zip timeGrid $ mean <$> dataSample
  let varData = zip timeGrid $ variance <$> dataSample
  let rangeData = zip timeGrid $ range <$> dataSample
  renderTrace
    ("monte_stat" <> show beta)
    [("mean", meanData), ("var", varData), ("range", rangeData)]
  renderMultiPlot ("monte_simu" <> show beta) ds

monteCarloStat beta = do
  let genFunc = do
        g <- newStdGen
        let timeInterval = 0.01 * 1
            rdat = realisation g (wiener $ timeInterval)
```

```
        return $
          take (100 * 1 * 10) $
          stepper (sdeCEV beta) schemeMilstein timeInterval rdat
  let genStat = do
        ds <- sequenceA $ replicate 100 genFunc
        print $ beta
        let dataSample = (V.fromList) <$> transpose (fmap snd <$> ds)
        let timeGrid = fst <$> head ds
        let varData = zip timeGrid $ variance <$> dataSample
        return varData
  dvar <- sequenceA $ replicate 100 genStat
  let dataVarSample = (V.fromList) <$> transpose (fmap snd <$> dvar)
  let timeGrid = fst <$> head dvar
  let varvarData = zip timeGrid $ variance <$> dataVarSample
  renderMultiPlot ("monte_var" <> show beta) [varvarData]

sdeCEV beta =
  SDE
  { _a = \xn tn -> xn * 0.1
  , _b = \xn tn -> (abs (xn) ** (beta / 2)) * 0.8
  , _b' = \xn tn -> (beta / 2) * (abs (xn) ** (beta / 2 - 1)) * 0.8
  , _x0 = 1
  }

-- abs' x = if x < 0.01 then 0 else x
sde1 = SDE {_a = \xn tn -> 0, _b = \xn tn -> 1, _b' = \xn tn -> 0, _x0 = 0}

sde2 =
  SDE
  { _a = \xn tn -> 0.1 * xn
  , _b = \xn tn -> 0.8 * xn
  , _b' = \xn tn -> 0.8
  , _x0 = 1
  }

sde3 =
  SDE
  { _a = \xn tn -> 0.1 * xn
  , _b = \xn tn -> sin' (10 * tn) * 0.8 * xn
  , _b' = \xn tn -> sin' (10 * tn) * 0.8
  , _x0 = 1
  }

sde4 =
  SDE
  { _a = \xn tn -> 0.1 * xn
```

```
  , _b = \xn tn -> (sin' (10 * tn)) ^ 2 * 0.8 * sqrt xn
  , _b' = \xn tn -> (sin' (10 * tn)) ^ 2 * 0.8 * 0.5 / sqrt xn
  , _x0 = 1
  }

sde5 =
  SDE
  { _a = \xn tn -> 0.1 * xn
  , _b = \xn tn -> (sin' (10 * tn)) ^ 2 * 0.8 * sin (xn * 5 * tn)
  , _b' = \xn tn -> (sin' (10 * tn)) ^ 2 * 0.8 * 5 * tn * cos (xn * 5 * tn)
  , _x0 = 1
  }

sin' = sin . (flip mod' $ 2 * pi)

cos' = cos . (flip mod' $ 2 * pi)

exempli1 sde dws = stepper sde schemeEuler 0.01 dws

exempli2 sde dws = stepper sde schemeMilstein 0.01 dws

exempli3 sde dws = stepper sde schemeRungeKutta 0.01 dws

exempliE sde dws = stepper sde schemeEuler (0.01 * 0.01) dws

listAdd :: (Num a) => Int -> [a] -> [a]
listAdd n ls = sum s : listAdd n lss
  where
    (s, lss) = splitAt n ls
```