Issues in the Software Implementation of Stochastic Numerical Runge–Kutta

Migran N. Gevorkyan[1], Anastasiya V. Demidova[1], Anna V. Korolkova[1], and Dmitry S. Kulyabov[1,2](✉)

[1] Department of Applied Probability and Informatics, Peoples' Friendship University of Russia (RUDN University), 6 Miklukho-Maklaya str., Moscow 117198, Russia
{gevorkyan_mn,demidova_av,korolkova_av,kulyabov_ds}@rudn.university
[2] Laboratory of Information Technologies, Joint Institute for Nuclear Research, 6 Joliot-Curie, Dubna, Moscow region 141980, Russia

Abstract. This paper discusses the application of stochastic Runge-Kutta-like numerical methods with weak and strong convergences for systems of stochastic differential equations in Itô form. At the beginning a brief overview of available publications about stochastic numerical methods and information from the theory of stochastic differential equations are given. Then the difficulties that arise when trying to implement stochastic numerical methods and motivate to use source code generation are described. We discuss some implementation details, such as program languages (Python, Julia) and libraries (Jinja2, Numpy). Also the link to the repository with source code is provided in the article.

Keywords: Stochastic differential equations
Stochastic numerical methods · Automatic code generation
Python language · Julia language · Template engine

1 Introduction

The article [16] describes the Python [30] implementation of stochastic numerical Runge-Kutta type methods. This implementations heavily relies on NumPy and SciPy [18] libraries. We chose Pyhon language because of it's simplicity and development speed. NumPy's capability to work with multidimensional arrays as tensors (functions `tensor_dot` and `einsum`) was also very helpful. However, the performance was low, and not so much because of Python slowness, as because we used the large number of nested loops (up to seven). In this paper, we consider an alternative approach of stochastic numerical methods implementation, based on automatic code generation.

This article is divided into three sections. The first section provides the overview of the main sources and presents information from the theory of stochastic differential equations (SDE) and methods for their numerical solution. The second section presents stochastic numerical schemes for scalar SDE with strong convergence and for SDE systems with strong and weak convergence. In addition to the general schemes, several coefficient tables are provided. This allows to implement a specific numerical method. Finally, the third section explains the use of code generation for stochastic numerical methods and describes some details of the generator we have implemented (we use Jinja2 [1] template engine).

2 Background Overview

In this section, we give a brief overview of the available publications on stochastic Runge-Kutta methods. We study multistage numerical schemes without partial derivatives from the drift vector $\mathbf{f}(t, \mathbf{X})$ and the diffusion matrix $\mathbf{G}(t, \mathbf{X})$, so we don't consider Milstein methods [25–27]).

First, who used a stochastic Brownian process for mathematical modeling was a French mathematician, a student of Henri Poincare—Bachelier (1870–1946) in 1900 in the work [3].

The book by Kloeden and Platen [19] is classical work on numerical methods for SDE. The book provides a brief introduction to the theory of stochastic Ito and Stratonovich differential equations and their applications. The last two thirds of the book are devoted to the presentation of numerical methods in the sense of strict and weak approximations, including a number of Runge-Kutta methods.

The dissertation by Rößler [32] is a consistent report of stochastic numerical Runge-Kutta-like methods. The author considers the approximation of Ito and Stratonovich SDE systems in a weak sense for the scalar and multidimensional Wiener process. After a brief review of the previous works, the author develops the stochastic equivalent of labelled trees theory (labelled trees are used to derive the order conditions in the case of deterministic Runge-Kutta methods, see, for example, [13, 17]).

Rossler considers weakly convergent stochastic Runge-Kutta-like methods for Ito and Stratonovich SDE systems for both the scalar and the multidimensional Wiener process. In the third and the fifth part of the dissertation the specific implementation of the explicit stochastic numerical methods for weak convergence was described.

Further results of Rosler studies were described in articles [14, 15] in collaboration with Debrabant. In the preprint [15] authors continued classification of stochastic methods, Runge-Kutta method with a weak convergence. Several concrete realizations and results of numerical experiments were obtained. In the another preprint [31], they gave tables for fourth stage and strong order convergence methods $p = 3.0$.

Euler–Maruyama method described by Maruyama in the paper [23] may be considered as the first stochastic Runge-Kutta-like method. The first systematic study of stochastic numerical Runge–Kutta-like methods of strong order of convergence $p = 1.0$ is given by Rümelin [33] and Platen in his thesis [29].

Great contribution was made by Burrage and Burrage in a series of articles [7–10,12]. In these papers, they not only studied methods of strong order $p = 1.5$, but also extended the theory of labeled trees to the stochastic case.

The article of Soheili and Namjoo [34] presented the three methods with strong convergence $p = 1.0$ and the numerical comparison with the method from the book [19].

Some of the first methods with weak convergence are given in the book [19]. Further development they received in article by Komori and Mitsui [20] and in [22]. In the article [35] two three-stage methods, the weak convergence of the $p = 2.0$, as well as numerical experiments were introduced.

In view of the extreme complexity of further improving the order of accuracy of stochastic numerical schemes, the modern studies are devoted to obtaining numerical schemes for special SDE cases. It is possible to point out some of such studies about stochastic symplectic Runge–Kutta-like methods [11,21,37] and stochastic analogues of the Rosenbrock method [2].

3 Stochastic Wiener Process and Software Generation of Its Trajectories

The stochastic process $W(t)$, $t \geqslant 0$ is called scalar *Wiener process* if the following conditions are true [19,28]:

- $\mathrm{P}\{W(0) = 0\} = 1$, or in other words, $W(0) = 0$ is almost certain;
- $W(t)$ is process with independent increments, i.e. $\{\varDelta W_i\}_0^{N-1}$ are independent random variables: $\varDelta W_I = W(t_{I+1}) - W(t_I)$ and $0 \leqslant t_0 < t_1 < t_2 < \ldots < t_N \leqslant T$;
- $\varDelta W_i = W(t_{I+1}) - W(t_I) \sim \mathcal{N}(0, t_{I+1} - t_I)$ where $0 \leqslant t_{I+1} < t_I < t$, $I = 0, 1, \ldots, N-1$.

The symbol $\varDelta W_i \sim \mathcal{N}(0, \varDelta t_i)$ denotes that $\varDelta W_i$ is normally distributed random variable with expected value $\mathbb{E}[\varDelta W_i] = \mu = 0$ and variance $\mathbb{D}[\varDelta W_i] = \sigma^2 = \varDelta t_i$.

The Wiener process is a model of *Brownian motion* (random walk). If we consider the process $W(t)$ in time points $0 = t_0 < t_1 < t_2 < \ldots < t_{N-1} < t_N$ when it experiences random additive changes, then directly from the definition of Wiener process follows:

$$W(t_1) = W(t_0) + \varDelta W_0, W(t_2) = W(t_1) + \varDelta W_1, \ldots, W(t_N) = W(t_{N-1}) + \varDelta W_{N-1},$$

where $\varDelta W_i \sim \mathcal{N}(0, \varDelta t_i)$, $\forall i = 0, \ldots, N-1$.

The multidimensional Wiener process $\mathbf{W}(t) \colon \varOmega \times [t_0, T] \to \mathbb{R}^m$ is defined as a random process composed of jointly independent one-dimensional Wiener processes $W^1(t), \ldots, W^m(t)$. Increments of $\varDelta W_I^\alpha$, $\forall \alpha = 1, \ldots, m$ are jointly independent normally distributed random variables. On the other hand, the vector $\varDelta W_I^\alpha$ can be represented as a multidimensional normally distributed random variable with the expectation vector $\mu = \mathbf{0}$ and the diagonal covariance matrix.

In the case of a multidimensional stochastic process one has to generate m sequences of n normally distributed random variables should be generated.

3.1 Program Generation of Wiener Process

To simulate a one-dimensional Wiener process, it is necessary to generate N normally distributed random numbers $\varepsilon_1, \ldots, \varepsilon_N$ and to construct their cumulative sums. The result will be the simulated *sample path* of the Wiener process $W(t)$ or—using a different terminology—concrete implementation of the Wiener process.

In the case of a multidimensional random process, n sequences of m normally distributed random variables should be generated (that is, n arrays, each of m elements).

We implemented Wiener process generator in `Python` [30] and `Julia` [5]. To generate an array of numbers distributed according to the standard normal distribution in the case of `Python`, we used the function `random.normal` from the `NumPy` [18] library and, in the case of Julia, the built-in `randn` function. Both functions give qualitative pseudorandom sequences, since their work uses generators of uniformly distributed pseudorandom numbers based on an algorithm called Mersenne's vortex [24] (Mersenne Twister), and generators of pseudorandom normally distributed numbers use the Box–Mueller transformation [4,6].

To generate the Wiener process in `Python` one should use the `WienerProcess` class. The following code gives an example of this class usage.

```
import stochastic

N = 100
T = (0.0, 10.0)
W = stochastic.WienerProcess(N=N, time_interval=T)

print("Step size: ", W.dt)
print("Time points: ", W.t)
print("Process iterations: ", W.dx)
print("Wiener Process trajectory: ", W.x)
print("Intervals numbers: ", len(W.dx))
print("Points number: ", len(W.x))
```

The class constructor does not have any required arguments. By default, a process is generated on a time interval [0, 1], which is divided into 1000 parts (`N=1000`). Thus, by default, a path consisting of 1001 points with step `dt` equal to 0.001.

In the case of `Julia`, the Wiener process generator is implemented as the composite data type `struct`

```
"""Stochastic Wiener process"""
struct WienerProcess <: AbstractStochasticProcess
    "Number of process steps"
    N::Int64
    "Time interval starting point"
    t_0::Float64
    "Time interval end point"
    t_N::Float64
    "Step size"
    dt::Float64
    "Time points"
    T::Vector{Float64}
    "Winer process values"
    X::Vector{Float64}
    "Winer process increments dX ~ N(0, dt)"
    dX::Vector{Float64}
end
```

With following contractors

```
WienerProcess(N::Int64, t_0::Float64, t_N::Float64)
WienerProcess(N::Int64, dt::Float64)
WienerProcess(N::Int64)
WienerProcess()
```

3.2 Calculation and Approximation of Multiple Ito Integrals of Special Form

Here we will not go into the general theory of multiple stochastic Ito integrals, a reader can refer to the book [19] for additional information. Here we consider multiple special integrals, which are included in the stochastic numerical schemes.

In General, for the construction of numerical schemes with order of convergence greater than $p = \frac{1}{2}$, it is necessary to calculate single, double and triple Ito integrals of the following form:

$$I^{\alpha}(t_n, t_{n+1}) = I^{\alpha}(h_n) = \int\limits_{t_n}^{t_{n+1}} \mathrm{d}W^{\alpha}(\tau),$$

$$I^{\alpha\beta}(t_n, t_{n+1}) = I^{\alpha\beta}(h_n) = \int\limits_{t_n}^{t_{n+1}} \int\limits_{t_n}^{\tau_1} \mathrm{d}W^{\alpha}(\tau_2)\mathrm{d}W^{\beta}(\tau_1),$$

$$I^{\alpha\beta\gamma}(t_n, t_{n+1}) = I^{\alpha\beta\gamma}(h_n) = \int\limits_{t_n}^{t_{n+1}} \int\limits_{t_n}^{\tau_1} \int\limits_{t_n}^{\tau_2} \mathrm{d}W^{\alpha}(\tau_3)\mathrm{d}W^{\beta}(\tau_2)\mathrm{d}W^{\gamma}(\tau_1),$$

where $\alpha, \beta, \gamma = 0 \ldots, m$ and $W^{\alpha}, \alpha = 1, \ldots, m$ are components of multidimensional Wiener process. In the case of $\alpha, \beta, \gamma = 0$, the increment of $\mathrm{d}W^0(\tau)$ is assumed to be $\mathrm{d}\tau$.

The problem is to get analytical formulas for these integrals with $\Delta W^I_n = W^I(t_{n+1}) - W^I(t_n)$ in them. Despite its apparent simplicity, this is not achievable for all possible combinations of indices. Let us consider in the beginning those cases when it is possible to obtain an analytical expression, and then turn to those cases when it is necessary to use an approximating formulas.

In the case of a single integral, the problem is trivial and the analytic expression can be obtained for any index α:

$$I^0(h_n) = \Delta t_n = h_n,\ \ I^\alpha(h_n) = \Delta W_n^\alpha,\ \alpha = 1,\dots,m.$$

In the case of a double integral $I^{\alpha\beta}(h_n)$, the exact formula takes place only at $\alpha = \beta$:

$$I^{00}(h_n) = \frac{1}{2}\Delta t_n = \frac{1}{2}h_n^2,\ \ I^{\alpha\alpha}(h_n) = \frac{1}{2}\left((\Delta W_n^\alpha)^2 - \Delta t_n\right),\ \alpha = 1,\dots,m,$$

in other cases, when $\alpha \neq \beta$ Express $I^{\alpha\beta}(h_n)$ by increments of ΔW_n^α and Δt_n in the final form is not possible, so we can only use numerical approximation.

For the mixed case $I^{0\alpha}$ and $I^{\alpha 0}$ in [32], simple formulas of the following form are given:

$$I^{0\alpha}(h_n) = \frac{1}{2}h_n\left(I^\alpha(h_n) - \frac{1}{\sqrt{3}}\zeta^\alpha(h_n)\right),$$
$$I^{\alpha 0}(h_n) = \frac{1}{2}h_n\left(I^\alpha(h_n) + \frac{1}{\sqrt{3}}\zeta^\alpha(h_n)\right),$$

where $\zeta_n^\alpha \sim \mathcal{N}(0, h_n)$ are multidimensional normal distributed random variables.

For the General case $\alpha, \beta = 1,\dots,m$, the book [19] provides the following formulas for approximating the double Ito integral $I^{\alpha\beta}$:

$$I^{\alpha\beta}(h_n) = \frac{\Delta W_n^\alpha \Delta W_n^\beta - h_n\delta^{\alpha\beta}}{2} + A^{\alpha\beta}(h_n),$$
$$A^{\alpha\beta}(h_n) = \frac{h}{2\pi}\sum_{k=1}^{\infty}\frac{1}{k}\left[V_k^\alpha\left(U_k^\beta + \sqrt{\tfrac{2}{h_n}}\Delta W_n^\beta\right) - V_k^\beta\left(U_k^\alpha + \sqrt{\tfrac{2}{h_n}}\Delta W_n^\alpha\right)\right],$$

where $V_k^\alpha \sim \mathcal{N}(0,1)$, $U_k^\alpha \sim \mathcal{N}(0,1)$, $\alpha = 1,\dots,m;\ k = 1,\dots,\infty;\ n = 1,\dots,N$ is numerical schema number. From the formulas it is seen that in the case $\alpha = \beta$, we get the final expression for the $I^{\alpha\beta}$, which we mentioned above. In the case of $\alpha \neq \beta$, one has to sum the infinite series $a^{\alpha\beta}$. This algorithm gives an approximation error of order $O(h^2/n)$, where n is number of left terms of an infinite series a^{ij}.

In the article [36] a matrix form of approximating formulas is introduced. Let $\mathbf{1}_{m\times m}$, $\mathbf{0}_{m\times m}$ be the unit and zero matrices $m \times m$, then

$$\mathbf{I}(h_n) = \frac{\Delta\mathbf{W}_n\Delta\mathbf{W}_n^T - h_n\mathbf{1}_{m\times m}}{2} + \mathbf{A}(h_n),$$
$$\mathbf{A}(h_n) = \frac{h}{2\pi}\sum_{k=1}^{\infty}\frac{1}{k}\left(\mathbf{V}_k(\mathbf{U}_k + \sqrt{2/h_n}\Delta\mathbf{W}_n)^T - (\mathbf{U}_k + \sqrt{2/h_n}\Delta\mathbf{W}_n)\mathbf{V}_k^T\right),$$

where $\Delta\mathbf{W}_n, \mathbf{V}_k, \mathbf{U}_k$ are independent normally distributed multidimensional random variables:

$$\Delta\mathbf{W}_n = (\Delta W_n^1, \Delta W_n^2, \dots, \Delta W_n^m)^T \sim \mathcal{N}(\mathbf{0}_{m\times m}, h_n\mathbf{1}_{m\times m}),$$
$$\mathbf{V}_k = (V_k^1, V_k^2, \dots, V_k^m)^T \sim \mathcal{N}(\mathbf{0}_{m\times m}, \mathbf{1}_{m\times m}),$$
$$\mathbf{U}_k = (U_k^1, U_k^2, \dots, U_k^m)^T \sim \mathcal{N}(\mathbf{0}_{m\times m}, \mathbf{1}_{m\times m}).$$

If the programming language supports vectored operations with multidimensional arrays, these formulas can provide a benefit to the performance of the program.

Finally, consider a triple integral. In the only numerical scheme in which it occurs, it is necessary to be able to calculate only the case of identical indexes $\alpha = \beta = \gamma$. For this case, [32] gives the following formula:

$$I^{\alpha\alpha\alpha}(h_n) = \frac{1}{6}\left((I^{\alpha}(h_n))^3 - 3I^0(h_n)I^{\alpha}(h_n)\right) = \frac{1}{6}\left((\varDelta W_n^{\alpha})^3 - 3h_n \varDelta W_n^{\alpha}\right).$$

3.3 Strong and Weak Convergence of the Approximating Function

Before proceeding to the formulation of numerical schemes, it is necessary to determine the criterion of accuracy of approximation of the simulated process $\mathbf{x}(t)$ by the grid function $\mathbf{x}_n$. Two criteria are used: *weak* and *strong* convergence.

The sequence of approximating functions $\{\mathbf{x}_n\}_1^N$ converges with order p to the exact solution $\mathbf{x}(t)$ of SDE in moment T in *strong sense* if constant $C > 0$ exists and $\delta_0 > 0$ such as $\forall h \in (0, \delta_0$ and following condition is fulfilled:

$$(\|\mathbf{x}(T) - \mathbf{x}_N\|) \leqslant Ch^p.$$

The sequence of approximating functions $\{\mathbf{x}_n\}_1^N$ converges with order p to the exact solution $\mathbf{x}(t)$ of SDE in moment T in *weak sense* if constant $C_F > 0$ exists and $\delta_0 > 0$ such as $\forall h \in (0, \delta_0]$ and the following condition is fulfilled:

$$|\mathbb{E}\left[F(\mathbf{x}(T))\right] - \mathbb{E}\left[F(\mathbf{x}_N)\right]| \leqslant C_F h^p.$$

Here $F \in C_{\mathrm{P}}^{2(p+1)}(\mathbb{R}, \mathbb{R}^d)$ is a continuous differentiable functional with polynomial growth.

If the $\mathbf{G}$ matrix is zero, then the strong convergence condition is equivalent to the deterministic case, but the order of strong convergence is not necessarily a natural number and can take fractional-rational values.

It is important to note that the choice of the convergence type depends on the problem one has to solve. Increasing the order of strict convergence leads to more accurate approximation of the trajectories of $\mathbf{x}(t)$. If one wants to calculate, for example, the moment of a random process $\mathbf{x}(t)$ or a generalized functional of the form $\mathbb{E}[F(\mathbf{x}(t))]$, one should increase the order of weak convergence.

4 Stochastic Runge–Kutta-like Numerical Methods

4.1 Euler–Maruyama Numerical Method

The simplest numerical method for solving scalar equations and systems of SDEs is the Euler–Maruyama method, named in honor of Gisiro Maruyama, who extended the classical Euler method for ODEs to the case of equation [23]. The method is easily The each step requires only corresponding to this step increment $\varDelta W_n^{\beta}$. The method has a strong order $(p_d, p_s) = (1.0, 0.5)$. The value p_d denotes the deterministic accuracy order, when the method is used for the equation with $G(t, x^{\alpha}(t)) \equiv 0$. The value p_s denotes the order of the stochastic part approximation.

4.2 Weak Stochastic Runge–Kutta-like Method with Order 1.5 for a Scalar Wiener Process

In the case of a scalar SDE, the drift vector $f^\alpha(t, x^\gamma)$ and the diffusion matrix $G^\alpha_\beta(t, x^\gamma)$ become $f(t, x)$ and $g(t, x)$ scalar functions, and the driving Wiener process W^β_t is the scalar W_t. For scalar SDE it is possible to construct a numerical scheme with strong convergence $p = 1.5$. In the above numerical scheme, the Wiener stochastic process is present in implicit way. It is "hidden" inside the stochastic Ito integrals: $I^{10}(h_n)$, $I^1(h_n)$, $I^{11}(h_n)$, $I^{111}(h_n)$.

4.3 Stochastic Runge–Kutta Method with Strong Order $p = 1.0$ for Vector Wiener Process

For SDE system with a multidimensional Wiener process, one can construct a stochastic numerical Runge-Kutta scheme of strong order $p_s = 1.0$ by using single and double Ito integrals [31].

Methods `SRK1Wm` and `SRK2Wm` have strong order $(p_d, p_s) = (1.0, 1.0)$ and $(p_d, p_s) = (2.0, 1.0)$.

4.4 Stochastic Runge–Kutta Method with Weak Order $p = 2.0$ for the Vector Wiener Process

Numerical methods with weak convergence are good for approximation of the distribution characteristics of stochastic process $x^\alpha(t)$. The weak numerical method does not need information about the trajectory of driving Wiener process W^α_n and random increments for these methods can be generated on another probability space. From the paper [15] we get two Butcher tables.

5 Analysis of Implementation Difficulties of Stochastic Runge–Kutta Numerical Methods

As can be seen from the formulas, stochastic Runge-Kutta methods are much more complicated than their classical analogues. In addition to the cumbersome formulas, we can highlight the following factors that complicate the implementation of stochastic methods in software, as well as their application to the numerical solution of SDEs.

– When choosing a particular method, it is necessary to consider what type of convergence is necessary to provide for this problem, as well as which of the stochastic equations should be solved—in Ito or Stratonovich form. This increases the number of algorithms one has to implement.
– For methods with strong convergence greater than one at each step it is necessary to solve the resource-intensive problem of stochastic integrals approximation.

– In the numerical scheme, there are not only matrices and vectors, but also tensors (four-dimensional arrays), it is necessary to perform a convolution operation on several indexes. The implementation of convolution via summation by using normal cycles results in a significant performance drop.
– Weak methods requires the Monte Carlo simulation and, therefore, a large number of repeated computations for the numerical solution. Since the Monte Carlo method converges approximately as $1/\sqrt{N}$, where N—number of calculations, to achieve an accuracy of at least 10^{-3}, it is necessary to perform minimum 10^6 tests.

The most significant performance drop occurs when implementing a universal algorithm, that is, a program that can make a calculation using an arbitrary coefficient table. In this case, we have to use a large number of nested loops in order to organize the summation. The presence of double sums in the schemes as well as complex combination of indices in the multipliers under the sign of these sums complicates the implementation even more and the number of nested cycles increases to six. In addition to these specific features, we mention a few reasons for the performance drop, which also take place in case of deterministic numerical methods. The obvious way to store the coefficients of the methods is to use arrays. However, in explicit methods, that we consider, the matrix is lower-diagonal and storing it as a two-dimensional array results in more than half of the allocated memory being spent on storing zeros.

While examine the source codes of popular routines that implement classical explicit embedded Runge–Kutta methods, one may find that these programs use a set of named constants rather than arrays to store the coefficients of the method. It is also caused by the fact that the operations with scalar variables in most programming languages are faster than operations on arrays.

We wish to preserve the requirement of code flexibility and at the same time to increase the speed of calculations and reduce the memory consumption. That led us to automatic code generation from one template.

In addition to performance gains, automatic code generation allows you to add or modify all functions at once by editing only one template. This allows both to reduce the number of errors and to generate different variants of functions for different purposes.

6 Automatic Code Generation

For code generation we use Python 3 language. The program is open source and available on bitbucket repository by URL bitbucket.org/mngev/sde_num_generation. The repository contains module `stochastic`. This module implements Wiener stochastic process and the numerical methods we considered in this paper. Most part of the module's code are generated by scripts from `generator` directory.

For the code generation, we used Jinja2 [1] template engine. This library was originally developed to generate HTML pages, but it has a very flexible syntax and can be used as a universal tool for generating text files of any kind, including source codes in any programming languages. In addition to Jinja2, we also used NumPy library to work with arrays and speed-up some calculations.

In addition to the two external libraries listed above, the standard `fraction` module was used. It allows to specify the coefficients of the method as rational fractions, and then convert them to float type with the desired order of accuracy. Also we use `typing` module to annotate the types of function arguments (Python 3.5 and above feature).

Templates are files with Python source code with insertions of Jinja2 specific commands. Information about the coefficients of the methods is stored separately, in a structured form of JSON format. This makes it easy to add new methods and modify old ones by editing JSON files. Currently we use methods with coefficients presented in [14,15,19].

Python itself is used as the language for already generated functions with the active use of NumPy library, which allows to get acceptable performance. However, the generated code can be easily reformatted to match the syntax of any other programming language. We plan to modify the program to generate code in Julia language (julialang.org). This language was introduced in 2012 and initially focused on scientific computing. Currently, he is intensively developing and gaining popularity. To date, the current version is 0.6.2. Julia provides performance comparable to C++ and Fortran, but it is a dynamic language with interactive command line (REPL) capability similar to IPython and can be integrated into an interactive Jupyter environment.

The current version of the library exceeds the one described by the authors in [16]. The use of auto-generation made it possible not to use nested loops, which reduced the number of memory allocations, and greatly simplified the code.

6.1 Realisation of Automatic Code Generation

To study the calculation errors and the efficiency of different stochastic numerical methods, it is necessary to have a universal implementation of such methods. The universality means the possibility to use any stochastic method with a desired strong or weak error by setting its coefficient table. With direct transfer of mathematical formulas to the program code, one need to use about five nested cycles, which extremely reduces performance, since such code does not take into account a large number of zeros in the coefficient tables and arithmetic operations on zero components are still performed, although this is an extra waste of processor time.

One way to achieve versatility and acceptable performance is to generate code for a numerical method step. This approach minimizes the number of arithmetic operations and saves memory, since the zero coefficients of the method do not have to be stored.

We implemented a code generator for the three stochastic numerical methods mentioned above:

- scalar method with strong convergence $p_s = 1.5$,

– vector method with strong convergence $p_s = 1.0$,
– vector method with weak convergence of $p_s = 2.0$.

We use Python to implement the code generator and Jinja2 [1] template engine. This template engine was originally created to generate HTML code, but its syntax is universal and allows you to generate text of any kind without reference to any programming or markup language.

Information about the coefficients of each particular method is stored as a JSON file of the following structure:

```
{
  "name": "method’s name (the future name of the function)",
  "description": "method’s short description",
  "stage": 4,
  "det_order": "2.0",
  "stoch_order": "1.5",
  "A0": [...],
  "B0": [...],
  "A1": [...],
  "B1": [
    ["0", "0", "0", "0"],
    ["1/2", "0", "0", "0"],
    ["-1", "0", "0", "0"],
    ["-5", "3", "1/2", "0"]
  ],
  "c0": ["0", "3/4", "0", "0"],
  "c1": ["0", "1/4", "1", "1/4"],
  "a": ["1/3", "2/3", "0", "0"],
  "b1": ["-1", "4/3", "2/3", "0"],
  "b2": ["-1", "4/3", "-1/3", "0"],
  "b3": ["2", "-4/3", "-2/3", "0"],
  "b4": ["-2", "5/3", "-2/3", "1"]
}
```

The parameter `stage` is the number of method’s stages, `det_order` is the error order of the deterministic part (p_d), `stoch_order` is the error order of the stochastic part (p_s), `name` is the name of the method, which will then be used to create the name of the generated function, so it should be written in one word without spaces. All other parameters are the coefficients of the method. In this case, we give the coefficients of the scalar method with strong convergence $p_s = 1.5$, omitting the coefficients $\mathbf{a}_0$, $\mathbf{a}_1$ and $\mathbf{B}_0$ to save text space. It is necessary to note that the values of the coefficients can be specified in the form of rational fractions, for which they should be presented as JSON strings and enclosed in double quotes.

For internal representation of stochastic numerical methods we created three Python classes: `ScalarMethod`, `StrongVectorMethod` and `WeakVectorMethod`. The implementation of these classes is contained in the file `coefficients_` `table.py`. The constructors of these classes read the JSON file and, based on them, create objects, which can later be used for code generation. The Fraction class from the Python standard library is used to represent rational coefficients. Each class has a method that generates a coefficient table in LaTeX format.

The file `stoch_rk_generator.py` is a script which handles the `jinja2` templates and, based on them, generates a code of python functions. For vector stochastic methods, a code is generated for dimensions up to 6. Functions are named based on the information specified in `JSON` files, such as `strong_srk1w2`, `strong_srk2w5`, `weak_srk2w6`, and so on.

In addition to the code in `Python`, LaTeX formulas are generated. It allows one to check the correctness of the generator. For example, we give below the formula generated automatically based on the data from `JSON` file for Runge–Kutta method `strong_srk1w2` with stages $s = 3$, and 2 dimensioned Wiener process. Nonzero coefficients of the method are as follows:

$$A^2_{01} = 1,\ A^2_{11} = 1,\ A^3_{11} = 1,\ B^2_{11} = 1,\ B^3_{11} = -1,$$
$$a_1 = 1/2,\ a_2 = 1/2,\ c^2_0 = 1,\ c^2_1 = 1,\ c^3_1 = 1,\ b^1_1 = 1,\ b^2_2 = 1/2,\ b^2_3 = -1/2.$$

7 Parallel SDE Integration with Weak Numerical Methods

Stochastic numerical methods with strong convergence are well suited for computing a specific trajectory of SDE solution. If we are not interested in a specific trajectory, but in some probabilistic characteristics (distribution of a random process, mathematical expectation, variance, etc.), then we should use numerical methods with weak convergence.

In the case of numerical methods with weak convergence, we have to use Monte Carlo method. It means that we should solve our SDE system multiple times and each time with different trajectory. The error of the Monte Carlo method depends on the number of trials N as $\sqrt{N}$, so to achieve the accuracy of 10^{-3} we need 10^6 trials. However, since the trajectories of the Wiener process are independent, the SDE for each specific trajectory can be solved independently in parallel mode.

We have implemented a script in Python, which allows to find solutions of SDE for N different trajectories in parallel mode by spawning a given number of processes. For processes spawning we use `multiprocessing` module. The following features of the Cpython interpreter should be noted.

- Because of the global interpreter lock (GIL), it is not possible to use threads for the Monte Carlo method. The standard `threading` module is only suitable for asynchronous tasks.
- When using processes, you should reinitialize the random number generator with new seed for each process separately, because otherwise all generated processes will generate the same sequence of random numbers.

The source code of the implemented script is located in the tests directory. It is based on two functions.

– Function `calculation` performs the necessary calculations for a given number of trajectories. As arguments, the function takes the drift vector, the diffusion matrix, the required number of simulations, the initializing value for the random generator, the initial value of the SDU solution, the number of steps of the Wiener process, the time interval at which it is necessary to carry out integration, the dimension of the Wiener process and optionally the function for testing the obtained solution for adequacy.
– Function `run_parallel` distributes the Monte Carlo tests equally between processes, creates a pool of processes, and runs them. Each process performs the function `calculation`.

When carrying out a large number of tests, the storage of all the resulting trajectories requires a significant amount of RAM. Therefore, it is more reasonable to immediately decide what probabilistic characteristics we need and calculate them using on-line algorithms. For example, to calculate the average trajectory, we use the following formula

$$\bar{\mathbf{x}}_n = \bar{\mathbf{x}}_{n-1} + \frac{\mathbf{x}_n - \bar{\mathbf{x}}_{n-1}}{n}.$$

This formula allows you to update the mean values of all path steps $\bar{\mathbf{x}}_n$ based on the previous mean values $\bar{\mathbf{x}}_{n-1}$ and the current value $\mathbf{x}_n$. As a result, each process must store only one array of constant length, which saves memory.

8 Conclusion

Stochastic numerical schemes with convergence order higher than 0.5 are considered. It is shown that such methods are much more complicated than equivalent numerical methods for systems of ordinary differential equations. Their specifics makes efficient software implementation of such methods not a trivial task. We discuss an approach based on automatic generation of code, which allows to obtain an efficient implementation of the methods and gives the possibility to use any table of coefficients. We also give a short description of our program and provide the url link to the repository with the source code.

Acknowledgments. The work is partially supported by Russian Foundation for Basic Research (RFBR) grants No 16-07-00556. Also the publication was prepared with the support of the "RUDN University Program 5-100".

References

1. Jinja2 official site. http://jinja.pocoo.org
2. Amiri, S., Hosseini, S.M.: Stochastic Runge-Kutta rosenbrock type methods for SDE systems. Appl. Numer. Math. **115**, 1–15 (2017). https://doi.org/10.1016/j.apnum.2016.11.010
3. Bachelier, L.: Théorie de la spéculation. Ann. Sci. l'École Norm. Supér. **3**(17), 21–86 (1900)

4. Bell, J.R.: Algorithm 334: normal random deviates. Commun. ACM **11**(7), 498 (1968). https://doi.org/10.1016/j.apnum.2016.11.010
5. Bezanson, J., Edelman, A., Karpinski, S., Shah, V.B.: Julia: a fresh approach to numerical computing. SIAM Rev. **59**, 65–98 (2017)
6. Box, G.E.P., Muller, M.E.: A note on the generation of random normal deviates. Ann. Math. Stat. **29**(2), 610–611 (1958). https://doi.org/10.1214/aoms/1177706645
7. Burrage, K., Burrage, P.M.: High strong order explicit Runge-Kutta methods for stochastic ordinary differential equations. Appl. Numer. Math. **22**, 81–101 (1996)
8. Burrage, K., Burrage, P.M.: General order conditions for stochastic Runge-Kutta methods for both commuting and non-commuting stochastic ordinary differential equation systems. Appl. Numer. Math. **28**, 161–177 (1998)
9. Burrage, K., Burrage, P.M.: Order conditions of stochastic Runge-Kutta methods by B-series. SIAM J. Numer. Anal. **38**, 1626–1646 (2000)
10. Burrage, K., Burrage, P.M., Belward, J.A.: A bound on the maximum strong order of stochastic Runge-Kutta methods for stochastic ordinary differential equations. BIT **37**, 771–780 (1997)
11. Burrage, K., Burrage, P.M.: Low rank Runge-kutta methods, symplecticity and stochastic hamiltonian problems with additive noise. J. Computat. Appl. Math. **236**(16), 3920–3930 (2012). https://doi.org/10.1016/j.cam.2012.03.007
12. Burrage, P.M.: Runge-Kutta methods for stochastic differential equations. Ph.D. thesis, University of Qeensland, Australia (1999)
13. Butcher, J.: Numerical Methods for Ordinary Differential Equations, 2nd edn. Wiley, New Zealand (2003)
14. Debrabant, K., Rößler, A.: Continuous weak approximation for stochastic differential equations. J. Comput. Appl. Math. **214**, 259–273 (2008)
15. Debrabant, K., Rößler, A.: Classification of stochastic Runge-Kutta methods for the weak approximation of stochastic differential equations, March 2013. arXiv:1303.4510v1
16. Gevorkyan, M.N., Velieva, T.R., Korolkova, A.V., Kulyabov, D.S., Sevastyanov, L.A.: Stochastic Runge–Kutta software package for stochastic differential equations. In: Zamojski, W., Mazurkiewicz, J., Sugier, J., Walkowiak, T., Kacprzyk, J. (eds.) Dependability Engineering and Complex Systems. AISC, vol. 470, pp. 169–179. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-39639-2_15
17. Hairer, E., Nørsett, S.P., Wanner, G.: Solving Ordinary Differential Equations I, 2nd edn. Springer, Berlin (2008). https://doi.org/10.1007/978-3-662-12607-3
18. Jones, E., Oliphant, T., Peterson, P., et al.: SciPy: open source scientific tools for Python (2001). http://www.scipy.org/. Accessed 08 10 2017
19. Kloeden, P.E., Platen, E.: Numerical Solution of Stochastic Differential Equations, 2nd edn. Springer, Heidelberg (1995). https://doi.org/10.1007/978-3-662-12616-5
20. Komori, Y., Mitsuri, T.: Stable ROW-type weak scheme for stochastic differential equations. RIMS Kokyuroku (932), 29–45 (1995)
21. Ma, Q., Ding, X.: Stochastic symplectic partitioned Runge-Kutta methods for stochastic hamiltonian systems with multiplicative noise. Appl. Math. Comput. **252**, 520–534 (2015). https://doi.org/10.1016/j.amc.2014.12.045
22. Mackevičius, V.: Second-order weak approximations for stratonovich stochastic differential equations. Lith. Math. J. **34**(2), 183–200 (1994). https://doi.org/10.1007/BF02333416
23. Maruyama, G.: Continuous Markov processes and stochastic equations. Rend. Circ. Mat. **4**, 48–90 (1955)

24. Matsumoto, M., Nishimura, T.: Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator. ACM Trans. Model. Comput. Simul. **8**(1), 3–30 (1998). https://doi.org/10.1145/272991.272995
25. Milstein, G.N.: Approximate integration of stochastic differential equations. Theory Probab. Appl. **19**, 557–562 (1974)
26. Milstein, G.N.: A method of second-order accuracy integration of stochastic differential equations. Theory Probab. Appl. **23**, 396–401 (1979)
27. Milstein, G.N.: Weak approximation of solutions of systems of stochastic differential equations. Theory Probab. Appl. **30**, 750–766 (1986)
28. Øksendal, B.: Stochastic Differential Equations. An Introduction with Applications, 6th edn. Springer, Heidelberg (2003). https://doi.org/10.1007/978-3-642-14394-6
29. Platen, E.: Beiträge zur zeitdiskreten Approximation von Itoprozessen. Ph.D. thesis, Akad. der Wiss., Berlin (1984)
30. Rossum, G.: Python reference manual. Technical report, Amsterdam, The Netherlands (1995). https://docs.python.org/3/
31. Rößler, A.: Strong and weak approximation methods for stochastic differential equations—some recent developments. In: Devroye, L., Karasözen, B., Kohler, M., Korn, R. (eds.) Recent Developments in Applied Probability and Statistics, pp. 127–153. Physica-Verlag HD, Heidelberg (2010). https://doi.org/10.1007/978-3-7908-2598-5_6
32. Rößler, A.: Runge-Kutta methods for the numerical solution of stochastic differential equations. Ph.D. thesis, Technischen Universität Darmstadt, Darmstadt, februar 2003
33. Rümelin, W.: Numerical treatment of stochastic differential equations. SIAM J. Numer. Anal. **19**(3), 604–613 (1982)
34. Soheili, A.R., Namjoo, M.: Strong approximation of stochastic differential equations with Runge-Kutta methods. World J. Model. Simul. **4**(2), 83–93 (2008)
35. Tocino, A., Ardanuy, R.: Runge-Kutta methods for numerical solution of stochastic differential equations. J. Comput. Appl. Math. **138**, 219–241 (2002)
36. Wiktorsson, M.: Joint characteristic function and simultaneous simulation of iterated Itô integrals for multiple independent Brownian motions. Ann. Appl. Probab. **11**(2), 470–487 (2001)
37. Zhou, W., Zhang, J., Hong, J., Song, S.: Stochastic symplectic Runge-Kutta methods for the strong approximation of hamiltonian systems with additive noise. J. Comput. Appl. Math. **325**, 134–148 (2017). https://doi.org/10.1016/j.cam.2017.04.050