UDC 004.4

Automatic Code Generation for Stochastic Runge–Kutta Methods

Migran N. Gevorkyan[*], **Anastasiya V. Demidova**[*], **Anna V. Korolkova**[*], **Dmitry S. Kulyabov**[*†]

[*] *Department of Applied Probability and Informatics,*
Peoples' Friendship University of Russia (RUDN University),
6 Miklukho-Maklaya str., Moscow, 117198, Russia
[†] *Laboratory of Information Technologies*
Joint Institute for Nuclear Research
6 Joliot-Curie, Dubna, Moscow region, 141980, Russia

Email: `gevorkyan_mn@rudn.university`, `demidova_av@rudn.university`, `korolkova_av@rudn.university`, `kulyabov_ds@rudn.university`

In this paper we consider in detail the realization of Runge–Kutta stochastic numerical methods with weak and strong convergence for systems of stochastic differential equations in Ito form. The algorithm for generating the Wiener stochastic process, the algorithm for approximation of Ito stochastic integrals, and the code generation algorithms for numerical schemes are described. Python and Julia languages are used. The Jinja2 template engine is used for the code generation .

Key words and phrases: stochastic differential equations; stochastic numeric methods; automatic code generation; Python; Julia; the template engine.

1. Introduction

This article is divided into three sections. In the first section we define the Wiener stochastic process and describe its implementation in Python and Julia . In the second section the approximation algorithm of Ito integrals is given. Finally, in the third part we introduce the details of code generator implementation for stochastic numerical methods [1, 2] (in Python language with use of Jinja2 template engine)

2. Stochastic Wiener process

2.1. The Definition and Properties

The stochastic process $W(t)$, $t \geqslant 0$ is called scalar *Wiener process* if the following conditions are true [3, 4]:

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

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

The Wiener process is the 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:

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

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

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

2.2. 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

$$\begin{aligned}
&\varepsilon_1,\\
&\varepsilon_1 + \varepsilon_2,\\
&\varepsilon_1 + \varepsilon_2 + \varepsilon_3,\\
&\vdots\\
&\varepsilon_1 + \varepsilon_2 + \varepsilon_3 + \ldots \varepsilon_{N-1},\\
&\varepsilon_1 + \varepsilon_2 + \varepsilon_3 + \ldots \varepsilon_{N-1} + \varepsilon_N.
\end{aligned}$$

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` [5] and `Julia` [6]. 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` [7] 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 [8] (Mersenne Twister), and generators of pseudorandom normally distributed numbers use the Box–Mueller transformation [?, 9].

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. An example of Wiener trajectories is shown on fig. 1.

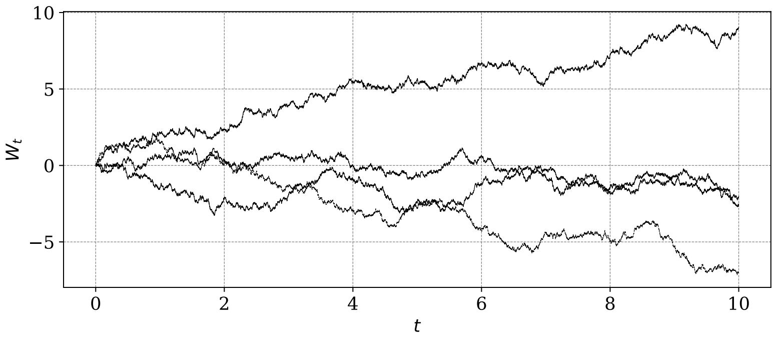

Figure 1. Multiple Wiener trajectories

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. 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 [4] for additional information. Here we will 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:

$$I^0(h_n) = \Delta t_n = h_n, \;\; I^{\alpha}(h_n) = \Delta W^{\alpha}_n, \; \alpha = 1, \ldots, m.$$

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

$$I^{00}(h_n) = \frac{1}{2}\Delta t_n = \frac{1}{2}h_n^2, \;\; I^{\alpha\alpha}(h_n) = \frac{1}{2}\left((\Delta W^{\alpha}_n)^2 - \Delta t_n\right), \; \alpha = 1, \ldots, m,$$

in other cases, when $\alpha \neq \beta$ it is not possible in the final form to express $I^{\alpha\beta}(h_n)$ in terms of ΔW^{α}_n and Δt_n , so we can only use numerical approximation.

For the mixed case $I^{0\alpha}$ and $I^{\alpha 0}$ in [10], 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 $\alpha, \beta = 1, \ldots, m$, the book [4] provides the following formulas for approximating the double Ito integral $I^{\alpha\beta}$:

$$I^{\alpha\beta}(h_n) = \frac{\Delta W_n^{\alpha} \Delta W_n^{\beta} - h_n \delta^{\alpha\beta}}{2} + A^{\alpha\beta}(h_n),$$

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

where $V_k^{\alpha} \sim \mathcal{N}(0, 1)$, $U_k^{\alpha} \sim \mathcal{N}(0, 1)$, $\alpha = 1, \ldots, m$; $k = 1, \ldots, \infty$; $n = 1, \ldots, N$ is the numerical schema number. From the formulas it is seen that in the case of $\alpha = \beta$, we may 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 the approximation error of order $O(h^2/n)$, where n is number of left terms of an infinite series a^{ij}.

In the article [11] the 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:

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

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, [10] gives the following formula:

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

4. Stochastic Runge–Kutta methods

Consider the random process $\mathbf{x}(t) = (x^1(t), \ldots, x^d(t))^T$, where $\mathbf{x}(t)$ belongs to the functional space $\mathrm{L}^2(\Omega)$ with the norm $\|\cdot\|$. We assume that the random process $\mathbf{x}(t)$ is a solution for the Ito SDE [3, 4] if:

$$\mathbf{x}(t) = \mathbf{f}(t, \mathbf{x}(t)) \mathrm{d}t + \mathbf{G}(t, \mathbf{x}(t)) \mathrm{d}\mathbf{W},$$

where $\mathbf{W} = (W^1, \ldots, W^m)^T$ is the multidimensional Wiener process, known as the *driving* process for SDE. The function $\mathbf{f}\colon [t_0, T] \times \mathbb{R}^d \to \mathbb{R}^d$ is called as the *drift vector*, and the matrix-valued function $\mathbf{G}\colon [t_0, T] \times \mathbb{R}^d \times \mathbb{R}^m \to \mathbb{R}^d \times \mathbb{R}^m$ is called the *diffusion matrix.* The same equation can be rewritten in the indexed form

$$x^\alpha(t) = f^\alpha(t, x^\gamma(t))\mathrm{d}t + \sum_{\beta=1}^{m} g^\alpha_\beta(t, x^\gamma(t))\mathrm{d}W^\beta,$$

where $\alpha, \gamma = 1, \dots, d$, $\beta = 1, \dots, m$, and $f^\alpha(t, x^\gamma(t)) = f^\alpha(t, x^1(t), \dots, x^d(t))$.

On the interval $[t_0, T]$, we introduce the grid $t_0 < t_1 < \ldots < t_N = T$ with step $h_n = t_{n+1} - t_n$, where $n = 0, \dots, N-1$ and the maximum grid step $h = \max\{h_{n-1}\}_1^N$. Next, we assume that the grid is uniform, then $h_n = h = \text{const}$. $\mathbf{x}_n$ is grid function, which approximate the stochastic process $\mathbf{x}(t)$, so $\mathbf{x}_0 = \mathbf{x}(t_0)$, $\mathbf{x}_n \approx \mathbf{x}(t_n)$ $\forall n = 1, \dots, N$.

4.1. Euler–Maruyama numerical method

The simplest numerical method for solving scalar equations and systems of SDEs is the Euler–Maruyama method.

$$x^\alpha_0 = x^\alpha(t_0),\ x^\alpha_{n+1} = x^\alpha_n + f^\alpha(t_n, x^\alpha_n)h_n + \sum_{\gamma=1}^{d} G^\alpha_\beta(t_n, x^\gamma_n)\Delta W^\beta_n.$$

The method has a strong order $(p_d, p_s) = (1.0, 0.5)$. The value p_d denotes the deterministic accuracy order, and value p_s denotes the stochastic part approximation order.

4.2. Weak stochastic Runge–Kutta-like method with order 1.5 for a scalar Wiener process

In the case of a scalar SDE it is possible to construct a numerical scheme with strong convergence $p = 1.5$ [12–16]:

$$X^i_0 = x_n + \sum_{j=1}^{s} A^i_{0j} f(t_n + c^j_0 h_n, X^j_0)h_n + \sum_{j=1}^{s} B^i_{0j} g(t_n + c^j_1 h_n, X^j_1)\frac{I^{10}(h_n)}{\sqrt{h_n}},$$

$$X^i_1 = x_n + \sum_{j=1}^{s} A^i_{1j} f(t_n + c^j_0 h_n, X^j_0)h_n + \sum_{j=1}^{s} B^i_{1j} g(t_n + c^j_1 h_n, X^j_1)\sqrt{h_n},$$

$$x_{n+1} = x_n + \sum_{i=1}^{s} a_i f(t_n + c^i_0 h_n, X^i_0)h_n +$$
$$+ \sum_{i=1}^{s}\left(b^1_i I^1(h_n) + b^2_i\frac{I^{11}(h_n)}{\sqrt{h_n}} + b^3_i\frac{I^{10}(h_n)}{h_n} + b^4_i\frac{I^{111}(h_n)}{h_n}\right) g(t_n + c^i_1 h_n, X^i_1),$$

where $i, j = 1, \dots, s$ (s is the number of method's stages). In the above numerical scheme, the Wiener stochastic process may be 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 $p_s = 1.0$ [4, 17] using single and double Ito integrals [18].

$$X^{0i\alpha} = x_n^\alpha + \sum_{j=1}^{s} A_{0j}^i f^\alpha(t_n + c_0^j h_n, X^{0j\beta}) h_n + \sum_{l=1}^{m}\sum_{j=1}^{s} B_{0j}^i G_l^\alpha(t_n + c_1^j h_n, X^{lj\beta}) I^l(h_n),$$

$$X^{ki\alpha} = x_n^\alpha + \sum_{j=1}^{s} A_{1j}^i f^\alpha(t_n + c_0^j h_n, X^{0j\beta}) h_n + \sum_{l=1}^{m}\sum_{j=1}^{s} B_{1j}^i G_l^\alpha(t_n + c_1^j h_n, X^{lj\beta}) \frac{I^{lk}(h_n)}{\sqrt{h_n}},$$

$$x_{n+1}^\alpha = x_n^\alpha + \sum_{i=1}^{s} a_i f^\alpha(t_n + c_0^i h_n, X^{0i\beta}) h_n + \sum_{k=1}^{m}\sum_{i=1}^{s} (b_i^1 I^k(h_n) + b_i^2 \sqrt{h_n}) G_k^\alpha(t_n + c_1^i h_n, X^{ki\beta}),$$

$n = 0, 1, \ldots, N-1$; $i = 1, \ldots, s$; $\beta, k = 1, \ldots, m$; $\alpha = 1, \ldots, d$.

4.4. Stochastic Runge–Kutta method with weak order $p = 2.0$ for vector Wiener process

Numerical methods with weak convergence are good for approximation 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_n^α and random increments for these methods can be generated on another probability space [4, 19–21].

$$X^{0i\alpha} = x_n^\alpha + \sum_{j=1}^{s} A_{0j}^i f^\alpha(t_n + c_0^j h_n, X^{0j\beta}) h_n + \sum_{j=1}^{s}\sum_{l=1}^{m} B_{0j}^i G_l^\alpha(t_n + c_1^j h_n, X^{lj\beta}) \hat{I}^l,$$

$$X^{ki\alpha} = x_n^\alpha + \sum_{j=1}^{s} A_{1j}^i f^\alpha(t_n + c_0^j h_n, X^{0j\beta}) h_n + \sum_{j=1}^{s} B_{1j}^i G_k^\alpha(t_n + c_1^j h_n, X^{kj\beta}) \sqrt{h_n},$$

$$\widehat{X}^{ki\alpha} = x_n^\alpha + \sum_{j=1}^{s} A_{2j}^i f^\alpha(t_n + c_0^j h_n, X^{0j\beta}) h_n + \sum_{j=1}^{s}\sum_{l=1, l\neq k}^{m} B_{2j}^i G_l^\alpha(t_n + c_1^j h_n, X^{lj\beta}) \frac{\hat{I}^{kl}}{\sqrt{h_n}},$$

$$x_{n+1}^\alpha = x_n^\alpha + \sum_{i=1}^{s} a_i f^\alpha(t_n + c_0^i, X^{0i\beta}) h_n + \sum_{i=1}^{s}\sum_{k=1}^{m} \left(b_i^1 \hat{I}^k + b_i^2 \frac{\hat{I}^{kk}}{\sqrt{h_n}} \right) G_k^\alpha(t_n + c_1^i h_n, X^{ki\beta}) +$$
$$+ \sum_{i=1}^{s}\sum_{k=1}^{m} \left(b_i^3 \hat{I}^k + b_i^4 \sqrt{h_n} \right) G_k^\alpha(t_n + c_2^i h_n, \widehat{X}^{ki\beta})$$

In the weak numerical schema $\hat{I}^{kl}$ are

$$\hat{I}^{kl} = \begin{cases} \frac{1}{2}(\hat{I}^k \hat{I}^l - \sqrt{h}_n \tilde{I}^k), \ k < l, \\ \frac{1}{2}(\hat{I}^k \hat{I}^l + \sqrt{h}_n \tilde{I}^l), \ l < k, \\ \frac{1}{2}((\hat{I}^k)^2 - h_n). \ k = l. \end{cases}$$

Here $\hat{I}^k$ denotes three point distributed random variable. It means, that $\hat{I}^k$ may have three values $\{-\sqrt{3h_n}, 0, \sqrt{3h_n}\}$ with probabilities 1/6, 2/3 and 1/6 respectively. $\tilde{I}^k$ denotes two point distributed random variable $\{-\sqrt{h_n}, \sqrt{h_n}\}$ with probabilities 1/2 and 1/2 respectively.

4.5. Automatic code generation for stochastic numerical methods of Runge–Kutta type

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 [22] 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:

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

The numerical scheme formulas are quite cumbersome, despite the large number of zeros in the coefficient table:

$$X^{01\alpha} = x^\alpha_n,\ X^{11\alpha} = x^\alpha_n,\ X^{21\alpha} = x^\alpha_n,$$
$$X^{02\alpha} = x^\alpha_n + h_n\Big[A^2_{01} f^\alpha(t_n, X^{01\beta})\Big],$$
$$\begin{aligned}
X^{12\alpha} &= x^\alpha_n + h_n\Big[A^2_{11} f^\alpha(t_n, X^{01\beta})\Big] \\
&+ B^2_{11} G^\alpha_1(t_n, X^{11\beta})\frac{I^{11}(h_n)}{\sqrt{h_n}} + B^2_{11} G^\alpha_2(t_n, X^{21\beta})\frac{I^{21}(h_n)}{\sqrt{h_n}}, \\
X^{22\alpha} &= x^\alpha_n + h_n\Big[A^2_{11} f^\alpha(t_n, X^{01\beta})\Big] \\
&+ B^2_{11} G^\alpha_1(t_n, X^{11\beta})\frac{I^{12}(h_n)}{\sqrt{h_n}} + B^2_{11} G^\alpha_2(t_n, X^{21\beta})\frac{I^{22}(h_n)}{\sqrt{h_n}}, \\
X^{13\alpha} &= x^\alpha_n + h_n\Big[A^3_{11} f^\alpha(t_n, X^{01\beta})\Big] \\
&+ B^3_{11} G^\alpha_1(t_n, X^{11\beta})\frac{I^{11}(h_n)}{\sqrt{h_n}} + B^3_{11} G^\alpha_2(t_n, X^{21\beta})\frac{I^{21}(h_n)}{\sqrt{h_n}}, \\
X^{23\alpha} &= x^\alpha_n + h_n\Big[A^3_{11} f^\alpha(t_n, X^{01\beta})\Big] \\
&+ + B^3_{11} G^\alpha_1(t_n, X^{11\beta})\frac{I^{12}(h_n)}{\sqrt{h_n}} + B^3_{11} G^\alpha_2(t_n, X^{21\beta})\frac{I^{22}(h_n)}{\sqrt{h_n}},
\end{aligned}$$

$$\begin{aligned}
x^\alpha_{n+1} = x^\alpha_n + h_n\Big[a_1 f^\alpha(t_n, X^{01\beta}) + a_2 f^\alpha(t_n + c^2_0 h_n, X^{02\beta})\Big] + b^1_1 I^1(h_n) G^\alpha_1(t_n, X^{11\beta}) \\
+ b^2_2\sqrt{h_n} G^\alpha_1(t_n + c^2_1 h_n, X^{12\beta}) + b^2_3\sqrt{h_n} G^\alpha_1(t_n + c^3_1 h_n, X^{13\beta})
\end{aligned}$$

$$+ b_1^1 I^2(h_n) G_2^{\alpha}(t_n, X^{21\beta}) + b_2^2 \sqrt{h_n} G_2^{\alpha}(t_n + c_1^2 h_n, X^{22\beta}) \\ + b_3^2 \sqrt{h_n} G_2^{\alpha}(t_n + c_1^3 h_n, X^{23\beta}).$$

5. Conclusion

We present the details of the software implementation of the Wiener process generation in Julia and Python, the algorithm of approximation of multiple Ito integrals and the details of the software implementation of the code generator for stochastic numerical methods of Runge–Kutta type. The code generator allows one to get an effective implementation of the methods and making it possible to use any table of coefficients. The source code of the described programs is open and available at the link `https://bitbucket.org/mngev/sde_num_generation`.

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. M. N. Gevorkyan, A. V. Demidova, T. R. Velieva, A. V. Korol'kova, D. S. Kulyabov, L. A. Sevast'yanov, Implementing a Method for Stochastization of One-Step Processes in a Computer Algebra System, Programming and Computer Software 44 (2) (2018) 86–93. `arXiv:1805.03190`, `doi:10.1134/S0361768818020044`.
2. A. V. Demidova, M. N. Gevorkyan, D. S. Kulyabov, A. V. Korolkova, L. A. Sevastianov, The Automation of Stochastization Algorithm with Use of SymPy Computer Algebra Library, EPJ Web of Conferences 173 (2018) 05006_1–4. `doi:10.1051/epjconf/201817305006`.
3. B. Øksendal, Stochastic differential equations. An introduction with applications, 6th Edition, Springer, Berlin Heidelberg New York, 2003.
4. P. E. Kloeden, E. Platen, Numerical Solution of Stochastic Differential Equations, 2nd Edition, Springer, Berlin Heidelberg New York, 1995.
5. G. Rossum, Python reference manual, Tech. rep., Amsterdam, The Netherlands, The Netherlands (1995).
6. J. Bezanson, A. Edelman, S. Karpinski, V. B. Shah, Julia: A Fresh Approach to Numerical Computing, SIAM Review 59 (2017) 65–98.
7. E. Jones, T. Oliphant, P. Peterson, et al., SciPy: Open source scientific tools for Python, [Online; accessed 08.10.2017] (2001–). URL `http://www.scipy.org/`
8. M. Matsumoto, T. Nishimura, Mersenne twister: A 623-dimensionally Equidistributed Uniform Pseudo-random Number Generator, ACM Trans. Model. Comput. Simul. 8 (1) (1998) 3–30. `doi:10.1145/272991.272995`.
9. G. E. P. Box, M. E. Muller, A note on the generation of random normal deviates, The Annals of Mathematical Statistics 29 (2) (1958) 610–611.
10. A. Rößler, Runge-Kutta Methods for the Numerical Solution of Stochastic Differential Equations, Ph.D. thesis, Technischen Universität Darmstadt, Darmstadt (februar 2003).
11. M. Wiktorsson, Joint characteristic function and simultaneous simulation of iterated Itô integrals for multiple independent Brownian motions, The Annals of Applied Probability 11 (2) (2001) 470–487.
12. K. Burrage, P. M. Burrage, High strong order explicit Runge-Kutta methods for stochastic ordinary differential equations, Appl. Numer. Math. (22) (1996) 81–101.

13. K. Burrage, P. M. Burrage, J. A. Belward, A bound on the maximum strong order of stochastic Runge-Kutta methods for stochastic ordinary differential equations., BIT (37) (1997) 771–780.
14. K. Burrage, P. M. Burrage, General order conditions for stochastic Runge-Kutta methods for both commuting and non-commuting stochastic ordinary differential equation systems, Appl. Numer. Math. (28) (1998) 161–177.
15. P. M. Burrage, Runge-Kutta Methods for Stochastic Differential Equations, Ph.D. thesis, University of Qeensland, Australia (1999).
16. K. Burrage, P. M. Burrage, Order conditions of stochastic Runge-Kutta methods by B-series, SIAM J. Numer. Anal. (38) (2000) 1626–1646.
17. A. R. Soheili, M. Namjoo, Strong approximation of stochastic differential equations with Runge–Kutta methods, World Journal of Modelling and Simulation 4 (2) (2008) 83–93.
18. A. Rößler, Strong and Weak Approximation Methods for Stochastic Differential Equations — Some Recent Developments (2010).
19. Y. Komori, T. Mitsuri, Stable ROW-Type Weak Scheme for Stochastic Differential Equations, RIMS Kokyuroku (932) (1995) 29–45.
20. V. Mackevičius, Second-order weak approximations for stratonovich stochastic differential equations, Lithuanian Mathematical Journal 34 (2) (1994) 183–200. doi:10.1007/BF02333416.
URL http://dx.doi.org/10.1007/BF02333416
21. A. Tocino, R. Ardanuy, Runge–Kutta methods for numerical solution of stochastic differential equations, Journal of Computational and Applied Mathematics (138) (2002) 219–241.
22. Jinja2 official site.
URL http://http://jinja.pocoo.org