CARF-F-274

SDE WEAK APPROXIMATION LIBRARY (SDE WA) (VERSION 1.0)

Mariko Ninomiya
The University of Tokyo

December 2011

SDE WEAK APPROXIMATION LIBRARY (SDE_WA) (VERSION 1.0)

MARIKO NINOMIYA

Abstract. In application of mathematical finance to practical problems, weak approximation of stochastic differential equations (SDEs) is one of the most important themes. In probabilistic approach to this problem, the Euler–Maruyama scheme which is a first-order weak approximation scheme has been used.

Kusuoka recently proposed a weak approximation schceme for diffusion processes. Lyons and Victoir extensively developed the idea of this scheme to establish the cubature formula on the Weiner space. These results and the spread of quasi Monte Carlo method showed the efficiency of higher-order weak approximation which is often called Kusuoka approximation or KLV scheme. Ninomiya–Victoir and Ninomiya–Ninomiya successfully constructed algorithms of this scheme. These algorithms have been improved in a number of research. (Fujiwara, Ooshima–Teichman-Veluscek, etc.)

The author constructed a universal numerical library written in C for calculation of weak approximation of any fiven SDEs following the Kusuoka scheme. Two types of algorithms mentioned above (NV and NN) of the Kusuoka scheme are included in this library. The Euler–Maruyama scheme is also available in this library.

The source code for this library can be obtained by downloading it from

`https://sites.google.com/site/marikoninomiya/`

0. Introduction

We consider weak approximation of SDEs, that is, calculation of $E\left[f\left(X(T,x)\right)\right]$ where $X(t,x) = \left(X^1(t,x), \dots X^N(t,x)\right)$ is a diffusion process denoted by

$$X^j(t,x) = x_j + \int_0^t V_0^j(X(s,x))\,\mathrm{d}s + \sum_{i=1}^{d}\int_0^t V_i^j(X(s,x)) \circ \mathrm{d}B(s), \tag{0.1}$$

for $j = 1, \dots, N$. Here $V_i \in C_b^\infty(\mathbb{R}^N;\mathbb{R}^N)$, $B^0(t) = t$, $(B^1(t), \dots, B^d(t))$ is a d-dimensional standard Brownian motion, and $\circ$ denotes Stratonovich integral.

This library called SDE_WA in this paper deals with the Euler–Maruyama scheme and the Kusuoka scheme.

1. Algorithms

In this section, we introduce the three types of algorithms included in SDE_WA. One is the Euler–Maruyama scheme. The othre two called NV or NN in this manual are based on the Kusuoka approximation.

As seen in [1], if the process X is of the form as (0.1), then $X^j(t,x)$ can be rewritten with the Ito integral as follows:

$$X^j(t,x) = x_j + \int_0^t \tilde{V}_0^j(X(s,x))\,\mathrm{d}s + \sum_{i=1}^d \int_0^t V_i^j(X(s,x))\,\mathrm{d}B(s),$$

where

$$\tilde{V}_0^j(X(s,x)) = V_0^j(X(s,x)) + \frac{1}{2}\sum_{k=1}^N \sum_{i=1}^d V_i^k(X(s,x))\frac{\partial V_i^j}{\partial x_k}(X(s,x)). \tag{1.1}$$

This relation held in the drifts of an Ito SDE and a Stratonovich SDE is to be important in defining SDE as to be mentioned later.

In the rest part of this manual, we let $[0,T]$ be partitioned into n intervals by $0 = t_0 < t_1 < \cdots < t_{n-1} < t_n = T$ with $\sum_{i=0}^{n-1}(t_{i+1} - t_i) = T$. For this partitioning, Δt_k denotes $t_k - t_{k-1}$.

1.1. Order of weak approximation.

If there exists a family $\left\{X^\delta(\cdot,\cdot)\right\}_{\delta>0}$ of random variables which has constants K and δ_0 such that for an arbitrary $\delta \in (0,\delta_0)$

$$\left|E[(f(X(T,x)))] - E[f(X^\delta(T,x))]\right| \le K\delta^p,$$

then the family $\left\{X^\delta(\cdot,\cdot)\right\}_{\delta>0}$ is said to be p-th order weak approximation of X.

1.2. Euler–Maruyama scheme.

The Euler–Maruyama scheme (EM) is well-known as a first-order weak approximation ([1]) scheme implemented by a very simple algorithm represented by the following random variables: for $k = 1,\dots,n$

$$\begin{aligned} X_0^{(\mathrm{EM}),n} &= x, \\ X_{t_k}^{(\mathrm{EM}),n} &= X_{t_{k-1}}^{(\mathrm{EM}),n} + \Delta t_k \tilde{V}_0\left(X_{t_{k-1}}^{(\mathrm{EM}),n}\right) + \sqrt{\Delta t_k}\sum_{i=1}^d V_i\left(X_{t_{k-1}}^{(\mathrm{EM}),n}\right)Z_k^i \end{aligned} \tag{1.2}$$

where $\tilde{V}_0$ denotes a drift in an Ito SDE. Here Z_k's are n independent d-dimensional random variables distributed as $N(0,1)$.

1.3. Kusuoka scheme.

The Kusuoka scheme is one of the higher-order weak approximation schemes. Two kinds of algorithms for implementation of this scheme are considered in SDE_WA. They are developped by Ninomiya–Victoir (NV) [4] and Ninomiya–Ninomiya (NN) [3].

Notation 1.1. $\exp(V)x$ *denotes the solution at time 1 of the ODE*

$$\frac{\mathrm{d}z_t}{\mathrm{d}t} = V(z_t), \quad z_0 = x.$$

Now we introduce the algorithms of the Kusuoka scheme.

(1) The NV alrorithm is defined by a family of random variables defined by

(1.3)
$$X_0^{(\mathrm{NV}),n} := x$$
$$X_{t_k}^{(\mathrm{NV}),n} :=$$
$$\begin{cases} \exp\left(\frac{\Delta t_k V_0}{2}\right)\exp\left(\sqrt{\Delta t_k}Z_k^1 V_1\right)\exp\left(\sqrt{\Delta t_k}Z_k^2 V_2\right)\cdots\exp\left(\sqrt{\Delta t_k}Z_k^d V_d\right)\exp\left(\frac{\Delta t_k V_0}{2}\right)X_{t_{k-1}}^{(\mathrm{NV}),n}, \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{if } \Lambda_k = +1 \\ \exp\left(\frac{\Delta t_k V_0}{2}\right)\exp\left(\sqrt{\Delta t_k}Z_k^d V_d\right)\exp\left(\sqrt{\Delta t_k}Z_k^{d-1} V_{d-1}\right)\cdots\exp\left(\sqrt{\Delta t_k}Z_k^1 V_1\right)\exp\left(\frac{\Delta t_k V_0}{2}\right)X_{t_{k-1}}^{(\mathrm{NV}),n}, \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{if } \Lambda_k = -1, \end{cases}$$

for $k = 1, \dots, n$ where $(\Lambda_k, Z_k)'$s are n-independent random variables such that Λ_k is a Bernoulli random variable independent of $Z_k \sim N_d(0, 1)$

(2) The NN algorithm is represented by the following random variables: for $k = 1, \dots, n$

(1.4)
$$X_0^{(\mathrm{NN}),n} := x,$$
$$X_{t_k}^{(\mathrm{NN}),n} := \exp\left(\frac{\Delta t_k}{2}V_0 + \sum_{i=1}^{d}\sqrt{\Delta t_k}S_{1,k}^i V_i\right)\exp\left(\frac{\Delta t_k}{2}V_0 + \sum_{i=1}^{d}\sqrt{\Delta t_k}S_{2,k}^i V_i\right)X_{t_{k-1}}^{(\mathrm{NN}),n}$$

where $\left(S_{j,k}^i\right)_{\substack{i\in\{1,\dots,d\},j\in\{1,2\}\\ k\in\{1,\dots,n\}}}$ are constructed by

$$\begin{pmatrix} S_{1,k}^i \\ S_{2,k}^i \end{pmatrix} = \begin{pmatrix} 1/2 & 1/\sqrt{2} \\ 1/2 & -1/\sqrt{2} \end{pmatrix}\begin{pmatrix} \eta_{1,k}^i \\ \eta_{2,k}^i \end{pmatrix}, \quad \text{where } \eta_{j,k}^i \overset{i.i.d.}{\sim} N(0,1). \tag{1.5}$$

Remark 1.1. *In the NV and the NN algorithms, Stratonovich SDEs are considered. Therefore, in use of these algorithms, an Ito SDE has to be converted to the corresponding Stratonovich SDE.*

This procedure can be done automatically through SDE_WA by giving some functions corresponding to $\frac{\partial V_i(y)}{\partial y}$'s as you will see later.

Remark 1.2. *For approximation of* $\exp(W)\cdot$ *in the NN algorithm where* $W = \frac{\Delta t_k}{2}V_0 + \sum_{i=1}^{d}\sqrt{\Delta t_k}S_{j,k}^i V_i$, $j = 1, 2$, *the Runge–Kutta method is applied. Also, in the case that there does not exist an explicit form of* $\exp(sV_i)\,y$ *in the NV algorithm, it should be approximated by the Runge–Kutta method, too. For more explanation, refer to the next section.*

These three types of implementation algorithms (EM, NV, and NN) of weak approximation of SDEs are included in SDE_WA.

2. Use of explicit forms or application of numerical integrator

SDE_WA has the following capabilities:

- numerical calculator of Ito–Stratonovich conversion
- 5-th or 7-th order "numerical integrator" for $\exp(W)y$ where $W \in C_b^\infty(\mathbb{R}^N;\mathbb{R}^N)$.

The Runge–Kutta method plays a role of "numerical integrator" in this lirary.

2.1. **Ito–Stratonovich conversion.** As we can see in (1.2), the EM algorithm is based on an Ito SDE. Hence, if the SDE is written in Stratonovich form, the drift term V_0 has to be converted to $\tilde{V}_0$ by (1.1). **This procedure is automatically done by SDE_WA as long as the the following information of the SDE is given by users:**

- the type of the SDE (Stratonovich)
- definitions of functions for $\frac{\partial V_1(y)}{\partial y}, \frac{\partial V_2(y)}{\partial y}, \ldots \frac{\partial V_d(y)}{\partial y}$.

If the considered SDE is written in Ito form, then the second information above ($\left(\frac{\partial V_1(y)}{\partial y}, \frac{\partial V_2(y)}{\partial y}, \ldots, \frac{\partial V_d(y)}{\partial y}\right)$) is not needed and would not be used even if it is given.

For the NN algorithm, simply contrary argument done for the EM algirthm holds, which means that for an Ito SDE, Ito–Stratonovich conversion should be automatically done by SDE_WA if the type of the SDE (Ito) and the definitions of $\frac{\partial V_1(y)}{\partial y}, \frac{\partial V_2(y)}{\partial y}, \ldots \frac{\partial V_d(y)}{\partial y}$ are given. Also, if the SDE is defined in Stratonovich form, then the functions for $\left(\frac{\partial V_1(y)}{\partial y}, \frac{\partial V_2(y)}{\partial y}, \ldots, \frac{\partial V_d(y)}{\partial y}\right)$ are not needed and would not be used even if given.

There are more possibilities for the NV algorithm. Calculation of $\exp(sV_i)\,y$ is itterated for $i = 0, 1, \ldots, d$ in the NV algorithm. Each $\exp(sV_i)\,y$ denotes the solution at time s of the correspoinding ODE. In SDE_WA, users can choose the way of calculation of $\exp(sV_i)\,y$ from use of explicit forms or application of numerical integration. We give more explanation below about possible situations and choices.

From the simplicity of the form, it could be easy to find an explicit form of $\exp(sV_i)\,y$. The advantages of using explicit forms of $\exp(sV_i)\,y$ are:

- problems related to singularities caused by numerical methods (e.g. violation of the domain) can be avoided
- calculation can be speeded up.

If explicit forms do not exist or are not to be used for some reason, then numerical integration is to be applied to approximation of such $\exp(sV_i)\,y$'s as in the NN algorithm.

In SDE_WA, there are the following rules for approximation of $\exp(sV_i)$:

(1) If the definition of the explicit form of $\exp(sV_i)\,y$ is given, then it is certainly used (i.e. numerical integration is not applied.)
(2) Whichever the SDE is given in Ito or Stratonovich form, in giving the definition a function for $\exp(sV_i)\,y$, users have to obtain the explicit form of it by themselves. Hence, $\tilde{V}_0$ (Ito drift) has to be manually converted to V_0 (Stratonovich drift) by users in the process if the considered SDE is written in Ito form.
(3) For $\exp(sV_i)\,y$ whose explicit form is not given, numerical integration is applied to approximation of it. In particular, if $i = 0$ and the SDE is given in Ito form, then Ito–Stratonovich conversion would be automatically done. In this case, users must give the functions corresponding to $\frac{\partial V_1(y)}{\partial y}, \ldots, \frac{\partial V_d(y)}{\partial y}$.

2.2. **Romberg extrapolation.** Suppose that we have a p-th order scheme such that there exists a constant $K_f > 0$ satisfying that for a smooth function f

$$\left| E\left[f\left(X_T^{(\mathrm{ord}p),n}\right)\right] - E\left[f\left(X(T,x)\right)\right] - K_f\frac{1}{n^p}\right| \leq C_f\frac{1}{n^{p+1}}.$$

Then,

$$\frac{2^p}{2^p-1}E\left[f\left(X_T^{(\mathrm{ord}p),2n}\right)\right]-\frac{1}{2^p-1}E\left[f\left(X_T^{(\mathrm{ord}p),n}\right)\right]$$

attains $p+1$-st weak approximation.

Remark 2.1. *Relation between the order m of the Runge–Kutta method used in the NV and the NN algorithms and the order p of the weak approximation of SDEs is $m = 2p + 1$ because the order of approximation of ODEs becomes half in approximation of SDEs (*[2]*).*

Therefore $m = 5$ for the second-order approximation such as the naive NV and NN algorithms and $m = 7$ when the Romberg extrapolation is applied to these from $2(2+1)+7 = 7$.

Let p be the order of approximation of SDEs and m the order of the Runge–Kutta method applied in the process of the NV or the NN algorithms.

$m = 2p + 1$ always has to hold if the NN algorithm is taken (Table 2).

For the EM algorithm, m does not even appear, that is, m can take any value (Table 1). (You will see later why we mention the case of EM here.)

In application of the NV algorithm, there are more possibilities than in the other two.

(i) If explicit forms of all $\exp(sV_i)\,y$'s and $\exp(sV_0)\,y$ are given, we do not need to care about m (Table 3) because of the rule we have seen in the previous section.
(ii) For such $\exp\left(\sqrt{\Delta t}Z^iV_i\right)y$ or $\exp\left(\frac{\Delta t V_0}{2}\right)y$ that its explicit form is not given, then it has to be approximated by the Runge–Kutta method of order $m = 2p + 1$ (Table 4).

	p	m
no Rom.	1	-
with Rom.	2	-

Table 1. EM

	p	m
no Rom.	2	5
with Rom.	3	7

Table 2. NN

	p	m
no Rom.	2	-
with Rom.	3	-

Table 3. NV:(i)

	p	m
no Rom.	2	5
with Rom.	3	7

Table 4. NV:(ii)

In SDE_WA, both the 5-th and 7-th order Runge–Kutta methods are equipped.

3. Definitions of data types and functions

All data types and functions introduced here are declared in the header file `<sde_wa.h>`.

3.1. **Defining SDE system and one-step calculator.**

The considered SDE and related information which we call an SDE system are defined using `SDE_WA_SYSTEM`

–Data Type : `SDE_WA_SYSTEM`

This data type defines an SDE with arbitrary parameters by having the following members:

- `enum SDE_type sde_type;`
  `enum SDE_type` is defined by

  `enum SDE_type {ITO=0, STR=1};`

  to denote the form of the considered SDE.
- `int (**V)(const double y[], double dy[], void *params);`
  This is a pointer to an array of functions that store the vector-valued $V_i(y)$ in the vector `dy`, for arguments `y`(initial vector) and parameters `params`.

-
  ```
  int (**drift_corrector)(const double y[], double *dVdy[],
                          void *params);
  ```
  This is a pointer to an array of functions corresponding to $\frac{\partial V_0(y)}{\partial y}$, $\frac{\partial V_1(y)}{\partial y}$, $\ldots$, $\frac{\partial V_d(y)}{\partial y}$. Each `drift_correctorp[`$i$`]` stores $\frac{\partial V_i^{j+1}(y)}{\partial y_{k+1}}$ in `dVdy[`j`][`k`]` for y =`y` with parameters `params`.

  It should be remarked that this array has $(1+d)$ function pointers whose first element (function)`drift_corrector[0]` is allocated for $\frac{\partial V_0(y)}{\partial y}$ which is not to be used. Hence, we can let `drift_corrector[0]=NULL`.

-
  ```
  int (**exp_sV)(double s, const double y[], double exp_sVy[],
                 void *params);
  ```
  This is a pointer to an array of functions that store the vector of explicit forms of $\exp(sV_i)y$'s for $i = 0, 1, ..., d$ of the ODE

  $$\frac{dz_t}{dt} = V_i(z_t), \quad z_0 = y$$

  in the vector `exp_sVy`, for arguments `s`, `y`, and parameters `params`. Remark that $\exp(sV_i)\, y$ should be obtained for a Stratonovich SDE.

  Any functions appearing above could be `NULL` when they do not exist or are not needed.
- `int dim_y;`
  This is the spacial dimension of the system of equations. `dim_y` corresponds to N of (0.1)

- `int dim_BM;`
  This is the dimension of the Browninan motion. `dim_BM` corresponds to d in (0.1)

- `void *params;`
  This is a pointer to arbitrary parameters of the system.

Remark 3.1. *It should be noticed from Table5 and Table6 that there are some cases in which it is possible to avoid defining functions corresponding to* $\frac{\partial V_i(y)}{\partial y}$. $\frac{\partial V_i(y)}{\partial y}$ *in these tables indicates that the corresponding function has to be defined while* `NULL` *does that it is not needed. Of course, it would not be a problem to let* `drift_corrector[`i`]=` $\frac{\partial V_i(y)}{\partial y}$ *by giving definition of* $\frac{\partial V_i(y)}{\partial y}$*, even if* `drift_corrector[`$i$`]=NULL` *in Table5 and Table6.*

alg.	sde_type=ITO	sde_type=STR

EM	drift_corrector[0]=NULL
	drift_corrector[i]=NULL
		drift_corrector[0]=NULL
		drift_corrector[i]=$\frac{\partial V_i(y)}{\partial y}$

NN	drif_corrector[0]=NULL
	drift_corrector[i]=$\frac{\partial V_i(y)}{\partial y}$
		drift_corrector[0]=NULL
		drift_corrector[i]=NULL

Table 5. drift_corrector and combinations of algorithms(EM, NN) and sde_type($i = 1, \dots, d$)

explicit form	sde_type=ITO	sde_type=STR

exp_sV[0]=NULL	drift_corrector[0]=NULL
	drift_corrector[i]=$\frac{\partial V_i(y)}{\partial y}$
		drift_corrector[0]=NULL
		drift_corrector[i]=NULL

exp_sV[0]=$\frac{\partial V_0(y)}{\partial y}$	drift_corrector[0]=NULL
	drift_corrector[i]=NULL	

Table 6. drift_corrector for NV ($i = 1, \dots, d$)

Memory for a one-step calculator should be allocated once and reused for iterative calculation in simulation. SDE_WA_SLTN data type describes a one-step calculator.

–Data Type:SDE_WA_SLTN
This data type describes a one-step calculator object determined by an algorithm, an SDE, parameters, and the required order of numerical integration (even if not used).

Though this data type includes various information as seen below, users only consider alg, mth_is, and sde in programming. However, users have to know data types of a sample point for EM, NV, and NN because these will be directly defined by users in programs.

- enum ALG alg;
  enum ALG is defined as follows :
  ```
  enum ALG {E_M=0, N_V=1, N_N=2};
  ```
- int mth_is;
  This integer denotes the order of numerical integration (the Runge–Kutta method) used in the NV or the NN algorithm. It can be 5 or 7 when the order of the scheme is 2 or 3 respectively. In the EM scheme, mth_is has to be set to any of 5 and 7 though it is never used.
- SDE_WA_SYSTEM *sde;
  This is a pointer to the considered SDE system.
- int (*one_step)(struct sde_wa_sltn *sl, double s);
  This is a function determined for alg. This function should store the result of one-step approximation with a time interval s.

- `double *initv;`
  This is a pointer to the array storing the initial vector for one-step approximation.
- `double *destv;`
  This is a pointer to the array to store the result of one-step approximation.
- `double *drift_step_interv;`
  This is a pointer to the array used in the process of converting the drift term or keeping intermidiate values in one-step approximation.
- `double **drift_corretor_matrix;`
  This is a pointer to the array used in the process of conversion between a Stratonovich-form SDE and an Ito-form SDE.

-
  ```
   union{
      double *em;
      RV_NV *nv;
      double *nn;
   }sample_pt;
  ```
  This is a pointer to a sample point determined by the considered `alg`. Here `RV_NV` is a data type defined for the NV algorithm as follows:
  ```
   enum Bernoulli_rv {T=0, H=1};
   typedef struct{
      enum Bernoulli_rv rv_nv_b;
      double *rv_nv_n;
   } RV_NV;
  ```
  For the EM shceme, `em[0]`, `em[1]`,..., `em[d]` get values of $Z_k^1, \ldots, Z_k^d$, respectively. For the NV algorithm, `nv->rv_nv_b` gets `H` or `T` which correspond to 1 or −1, respectively. Also, `nv->rv_nv_n[0]`, `nv->rv_nv_n[1]`,..., `nv->rv_nv_n[d-1]` get values of $Z_k^1, \ldots, Z_k^d$, respectively. For the NN algorithm, `nn[0],..., nn[d-1], nn[d],..., nn[2d-1]` get values of $\eta_{1,k}^1, \eta_{1,k}^2, \ldots, \eta_{1,k}^d, \eta_{2,k}^1, \eta_{2,k}^2, \ldots, \eta_{2,k}^d$, respectively.(Correspondence can be found in Table 7.)
- `double *rk_step_interv;`
  This is a pointer to the array used in the process of the Runge–Kutta method in the NV or the NN algorithgm.

- `double *nn_sample_pt_interv;`
  This is a pointer to the array used in the process of conversion of a normally distributed sample point to a sample point satisfying (1.5). This process is conducted only in the NN algorithm.

3.2. **Allocating/instantiating and freeing objects.** A great number of calculations of (1.2), (1.3), or (1.4) is normally itterated in simulation. We give functions to allocate memory of `SDE_WA_SYSTEM` and `SDE_WA_SLTN` data types to be reused. The functions to free the allocated memory after itterative calculation are also given below.

–Function: `SDE_WA_SYSTEM *alloc_SDE_WA_SYSTEM(int N, int d,`
`void *params);`
This function returns a pointer to a newly allocated memory of `SDE_WA_SYSTEM` data type for an `N`-dimensional SDE with a `d`-dimensional Brownian motion and

parameters `params`.

–Function: `void free_SDE_WA_SYSTEM(SDE_WA_SYSTEM *sys);`
This function frees all the memory associated with an `SDE_WA_SYSTEM` data type object.

–Function:
```
SDE_WA_SLTN *alloc_SDE_WA_SLTN (enum ALG alg, int mth_is,
                                          SDE_WA_SYSTEM *sde);
```
This function returns a pointer to a newly allocated memory of one-step calculator for the algorithm `alg` with `sde` and the (`mth_is`)-th-order Runge–Kutta method if the NV or the NN algorithm. The object is instantiated at the same time.
–Function: `void free_SDE_WA_SLTN(SDE_WA_SLTN *sltn);`
This function frees all the memory associated with an `SDE_WA_SLTN` data type object.

Remark 3.2. *Since* `free_SDE_WA_SLTN` *uses* `sltn->sde->dim_BM` *in freeing memory, we must* **NOT** *free* `SDE_WA_SYSTEM *sde` *until free* `SDE_WA_SLTN *sltn`*.*

3.3. **One-step calculation.** Here "one-step" means getting $X^{(*)}_{t_{(k+1)/n}}$ from $X^{(*)}_{t_{k/n}}$.

–Function:
```
int next_SDE_WA(SDE_WA_SLTN *X, double s, double y[],
                          double dy[], void *rv);
```
One-step calculation with initial vector `y`, a time interval `s`, and a sample point `rv` should be done through this function. The result of the calculation is to be stored in `dy[0]`, `dy[1]`,... `dy[`$N-1$`]`.

`rv` has to be appropriately defined depending on each algorithm as explained in the previous chapter (also see Table 7).

`alg`	data type of `rv`	correspondence($i = 0, \dots, d-1$)
`E_M`	`double *`	`rv[`i`]`= Z^{i+1}_k
`N_V`	`RV_NV *`	`rv.rv_nv_b`= Λ_k `rv.rv_nv_n[`i`]`= Z^{i+1}_k
`N_N`	`double *`	`rv[`i`]`= $\eta^{i+1}_{1,k}$ `rv[`$d+i$`]`= $\eta^{i+1}_{2,k}$

TABLE 7. data types of `rv`

4. EXAMPLE

4.1. **Asian option under Heston SV model.** We consider pricing an Asian option under the Heston model which is a well-known stochastic volatility model. Then, the considered SDE can be written in Ito form as

$$
\begin{aligned}
X^1(t,x) =& x^1 + \int_0^t \mu X^1(s,x)\,\mathrm{d}s + \int_0^t X^1(s,x)\sqrt{X^2(s,x)}\,\mathrm{d}B^1(s),\\
X^2(t,x) =& x^2 + \int_0^t \alpha\left(\theta - X^2(s,x)\right)\mathrm{d}s + \int_0^t \beta\sqrt{X^2(s,x)}\rho\,\mathrm{d}B^1(s)\\
&+ \int_0^t \beta\sqrt{X^2(s,x)(1-\rho^2)}\,\mathrm{d}B^2(s)\\
X^3(t,x) =& 0 + \int_0^t X_1(s,x)\,\mathrm{d}s + \int_0^t 0\,\mathrm{d}B^1(s) + \int_0^t 0\,\mathrm{d}B^2(s),
\end{aligned}
\tag{4.1}
$$

where $x = (x_1, x_2, 0) \in (\mathbb{R}_{>0})^3$, $(B^1(t), B^2(t))$ is a two-dimensional standard Brownian motion (i.e. $d = 2$), $-1 \le \rho \le 1$, and α, θ, μ are some positive coefficients such that $2\alpha\theta - \beta^2 > 0$ to ensure the existence and uniqueness of a solution to the first two SDEs.

This example is featured by the following two facts:

- the SDE's drift term differs depending on the form (Ito or Stratonovich).
- $\exp(sV_0)\,y$ does not have an explicit form

4.1.1. *Allocation and instantion of SDE system.*

- **Definition of a set of parameters and allocation of memory for an SDE system**
  As we have already seen, `alloc_SDE_WA_SYSTEM` has three arguments including `void *params`.

  Five parameters, α, β, μ, ρ, and θ, are included in (4.1). In order to allocate memory of an SDE system, we first define a set of these parameters as `struct AH_params` as follows:

  cc1 (`sde_wa_manual_program_ah_system.h`)

```
struct AH_params{
double alpha;
double beta;
double mu;
double rho;
double theta;
};
```

  We let `ah_params` be a `struct AH_params` data type variable and initiate it as follows:

cc2 (sde_wa_manual_program_ah_nn.c)

```
struct AH_params ah_params;
double alpha = 2.0;
double beta = 0.1;
double rho = 0.0;
double theta = 0.09;
double mu=0.05;

ah_params.alpha=alpha;
ah_params.beta=beta;
ah_params.mu=mu;
ah_params.rho=rho;
ah_params.theta=theta;
```

It is of course possible to directly set values to `ah_params.alpha`, `ah_params.beta`, etc. without through `alpha`, `beta`, etc.

Now we have all arguments for `alloc_SDE_WA_SYSTEM` : spacial dimension = 3, dimension of Brownian motion = 2, and a pointer to a set of parameters `ah_params`.

cc3 (sde_wa_manual_program_ah_nn.c)

```
SDE_WA_SYSTEM *sde;
sde=alloc_SDE_WA_SYSTEM(3, 2, &ah_params);
```

- **Definitions of $\tilde{V}_0$, V_1, and V_2**
  We need define functions for `sde->V[0]`, `sde->V[1]`, and `sde->V[2]`. It should be noticed here that (4.1) is an Ito SDE. This is the reason that we do not use V_0 but $\tilde{V}_0$.
  The definitions of $\tilde{V}_0$, V_1, and V_2 are given by

$$\tilde{V}_0\begin{pmatrix} y_1 \\ y_2 \\ y_3 \end{pmatrix} = \begin{pmatrix} \mu y_1 \\ \alpha(\theta - y_2) \\ y_1 \end{pmatrix}, \; V_1\begin{pmatrix} y_1 \\ y_2 \\ y_3 \end{pmatrix} = \begin{pmatrix} y_1\sqrt{y_2} \\ \beta\sqrt{y_2}\rho \\ 0 \end{pmatrix}, \tag{4.2}$$
$$V_2\begin{pmatrix} y_1 \\ y_2 \\ y_3 \end{pmatrix} = \begin{pmatrix} 0 \\ \beta\sqrt{y_2(1-\rho)} \\ 0 \end{pmatrix}.$$

We define `ah_V_0`, `ah_V_1`, and `ah_V_2` that correspond to $\tilde{V}_0$, V_1, and V_2, respectively. We recall

```
int (**V)(const double y[], double dy[], void *params);
```

Since `ah_V_0`, `ah_V_1`, and `ah_V_2` are to be set to `V[0]`, `V[1]`, and `V[2]`, respectively, the types of these functions should be

```
int ah_V_0(const double y[], double dy[], void *params);
int ah_V_1(const double y[], double dy[], void *params);
int ah_V_2(const double y[], double dy[], void *params);
```

These functions are defined as **cc4**, **cc5** and **cc6**.

cc4 (sde_wa_manual_program_ah_system.c)

```
int ah_V_0(const double y[], double dy[],
           void *params){

  struct AH_params *pparams;
  pparams=params;

  dy[0]=pparams->mu*y[0];
  dy[1]=pparams->alpha*(pparams->theta-y[1]);
  dy[2]=y[0];

  return SDE_WA_SUCCESS;
}
```

cc5 (sde_wa_manual_program_ah_system.c)

```
int ah_V_1(const double y[], double dy[],
           void *params){

  struct AH_params *pparams;
  pparams=params;

  dy[0]=y[0]*sqrt(y[1]);
  dy[1]=(pparams->rho)*(pparams->beta)
        *sqrt(y[1]);
  dy[2]=0.0;

  return SDE_WA_SUCCESS;
}
```

cc6 (sde_wa_manual_program_ah_system.c)

```
int ah_V_2(const double y[], double dy[],
           void *params){

  struct AH_params *pparams;
  pparams=params;

  dy[0]=0.0;
  dy[1]=(pparams->beta)*sqrt((1.0
        -(pparams->rho)*(pparams->rho))*y[1]);
  dy[2]=0.0;

  return SDE_WA_SUCCESS;
}
```

- **Definitions of partial derivatives of V_1 and V_2**
  The partial derivatives of V_1 and V_2 are used in the process of Ito-Stratonovich conversion (1.1) though the combination of the algorithm to be used and

the form of the SDE (Ito or Stratonovich) can avoid the use of these as seen in Table5 and Table6.

The array of partial derivatives of V_1 and V_2 become

$$\frac{\partial V_1}{\partial y}(y) = \begin{pmatrix} \sqrt{y_2} & y_1/(2\sqrt{y_2}) & 0 \\ 0 & \beta\rho/(2\sqrt{y_2}) & 0 \\ 0 & 0 & 0 \end{pmatrix}, \frac{\partial V_2}{\partial y}(y) = \begin{pmatrix} 0 & 0 & 0 \\ 0 & \beta\sqrt{1-\rho^2}/(2\sqrt{y_2}) & 0 \\ 0 & 0 & 0 \end{pmatrix}.$$

Remark that $\frac{\partial V_i}{\partial y}(y) = \left(a_{jk}\right)_{j,k\in\{1,\ldots,N\}}$ with $a_{jk} = \frac{\partial V_i^j}{\partial y_k}(y)$.
We define $\frac{\partial V_1}{\partial y}y$ and $\frac{\partial V_2}{\partial y}y$ as `diff_ah_V_1` and `diff_ah_V_2`, respectively. It should be remarked that the first element of `drift_corrector` is allocated for $\frac{\partial \tilde{V}_0}{\partial y}(y)$, though it will never be used in Ito–Stratonovich conversion.

cc7 (sde_wa_manual_program_ah_system.c)

```
int diff_ah_V_1(const double y[], double *dVdy[],
                void *params){

  struct AH_params *pparams;
  pparams=params;

  dVdy[0][0]=sqrt(y[1]);
  dVdy[0][1]=(y[0]/sqrt(y[1]))/2.0;
  dVdy[0][2]=0.0;

  dVdy[1][0]=0.0;
  dVdy[1][1]=pparams->rho*pparams->beta
            /(sqrt(y[1])*2.0);
  dVdy[1][2]=0.0;

  dVdy[2][0]=0.0;
  dVdy[2][1]=0.0;
  dVdy[2][2]=0.0;

  return SDE_WA_SUCCESS;
}
```

cc8 (sde_wa_manual_program_ah_system.c)

```
int diff_ah_V_2(const double y[], double *dVdy[],
                void *params){

  struct AH_params *pparams;
  pparams=params;

  dVdy[0][0]=0.0;
  dVdy[0][1]=0.0;
  dVdy[0][2]=0.0;

  dVdy[1][0]=0.0;
  dVdy[1][1]=pparams->beta*
             sqrt(1-pparams->rho*pparams->rho)
             /(sqrt(y[1])*2.0);
  dVdy[1][2]=0.0;

  dVdy[2][0]=0.0;
  dVdy[2][0]=0.0;
  dVdy[2][0]=0.0;

  return SDE_WA_SUCCESS;

}
```

- **Definitions of explicit forms of** $\exp(sV_i)$**'s**
  The explicit forms of $\exp(sV_i)\,y$'s can be used in the process of the NV algorithm while these are never needed for the EM scheme or the NN algorithm.

  For $\exp(sV_i))\,y$ which does not have an explicit form, numerical integration (the Runge–Kutta method) of order 5 or 7 is applied in the NV algorithm.

  If the explicit form of $\exp(sV_i)\,y$ is obtained to be used, the corresponding function has to be defined and provided to the SDE system object, as you will see below. On the other hand, if an explicit form of $\exp(sV_i)\,y$ is not to be used, then the corresponding element of `sde->exp_sV` should be `NULL`.

  Equation (4.1) is an Ito SDE. Since explicit forms must be expressed through an Stratonovich SDE, we convert the drift term of (4.1) to V_0 by (1.1) to obtain

$$V_0\begin{pmatrix} y_1 \\ y_2 \\ y_3 \end{pmatrix} = \begin{pmatrix} y_1\,(\mu - y_2/2 - \rho\beta/4) \\ \alpha\,(\theta - y_2) - \beta^2/4 \\ y_1 \end{pmatrix}. \tag{4.3}$$

Then $\exp(sV_0)\, y$ does not have an explicit form while $\exp(sV_1)y$ and $\exp(sV_2)y$ dose as

$$
\exp(sV_1)\begin{pmatrix} y_1 \\ y_2 \\ y_3 \end{pmatrix} = \begin{pmatrix} y_1 \exp\left(s\sqrt{y_2} + \rho\beta s^2/4\right) \\ \left(\rho\beta s/2 + \sqrt{y_2}\right)^2 \\ y_3 \end{pmatrix},
$$

$$
\exp(sV_2)\begin{pmatrix} y_1 \\ y_2 \\ y_3 \end{pmatrix} = \begin{pmatrix} y_1 \\ \left(\frac{s\beta\sqrt{1-\rho^2}}{2} + \sqrt{y_2}\right)^2 \\ y_3 \end{pmatrix}.
$$

These explicit forms are defined as follows:

cc9 (sde_wa_manual_program_ah_system.c)

```
int exp_ah_sVy_1(double s, const double y[],
                 double dy[], void *params){

  struct AH_params *pparams;

  pparams=params;
  dy[0]=y[0]*exp(s*sqrt(y[1])
        +pparams->rho*pparams->beta*s*s/4.0);
  dy[1]=(pparams->rho*psarams->beta*s/2.0
         +sqrt(y[1]))*
        (parameters.rho*parameters.beta*s/2.0
         +sqrt(y[1]));
  dy[2]=y[2];

  return SDE_WA_SUCCESS;
}
```

cc10 (sde_wa_manual_program_ah_system.c)

```
int exp_ah_sVy_2(double s, const double y[],
                 double dy[], void *params){

  struct AH_params *pparams;
  pparams=params;

  dy[0]=y[0];
  dy[1]=(s*pparams->beta
          *sqrt(1-pparams->rho*pparams->rho)/2.0
         +sqrt(y[1]))
         *(s*pparams->beta
          *sqrt(1-params->rho*pparams->rho)/2.0
         +sqrt(y[1]));
  dy[2]=y[2];

  return SDE_WA_SUCCESS;
}
```

Remark 4.1. *There are two possibilities of improvements in terms of speed of calculation as follows:*

(i) Since we have obtained V_0 (4.3) *from $\tilde{V}_0$, we can easily define V_0 as follows:*

cc4-STR

```
int ah_str_V_0(const double y[], double dy[],
               void *params){

  struct AH_params *pparams;
  pparams=params;

  dy[0]=y[0]*(pparams->mu-y[1]/2
               -pparams->rho*pparams->beta/4);
  dy[1]=pparams->alpha*(pparams->theta-y[1])
        -pparams->beta*pparams->beta/4;
  dy[2]=y[0];

  return SDE_WA_SUCCESS;
}
```

If we define the SDE system in Stratonovich form, then Ito–Stratonovich conversion would not be needed for the NV or the NN algorithm. This fact results in higher speed of calculation as long as the NV or the NN algorithm is considered.

(ii) Though $\exp(sV_0)\,y$ *does not an explicit form, it can be approximated as follows:*

$$\exp(sV_0)\begin{pmatrix} y_1 \\ y_2 \\ y_3 \end{pmatrix} = \begin{pmatrix} y_1 \exp\left(\left(\mu - \frac{\rho\beta^2}{4} - \frac{J}{2}\right)s + \frac{y_2 - J}{2\alpha}\left(e^{-\alpha s} - 1\right)\right) \\ J + (y_2 - J)\,e^{-\alpha s} \\ y_3 + \frac{y_1\left(e^{As} - 1\right)}{A} \end{pmatrix}, \tag{4.4}$$

where $J = \theta - \frac{\beta^2}{4\alpha}$ *and* $A = \mu - \frac{\rho\beta^2}{4} - \frac{y_2}{2}$*. The corresponding function can be defined as follows:*

cc-*exp*$(sV_0)y$**-approx.**

```
int exp_ah_sVy_0(double s, const double y[],
                double dy[], void *params){

  double J;
  double A;
  struct AH_params *pparams;

  pparams=params;
  J=pparams->theta-pparams->beta*pparams->beta
     /(4.0*pparams->alpha);
  A=pparams->mu-pparams->rho*pparams->beta
     *pparams->beta/4.0-y[1]/2.0;

  dy[0]=y[0]*exp((pparams->mu-pparams->rho
                 *pparams->beta*pparams->beta/4.0
                 -J/2.0)*s
                +(y[1]-J)
                 *(exp(-pparams->alpha*s)-1.0)
                          /(2.0*pparams->alpha));
  dy[1]=J+(y[1]-J)*exp(-pparams->alpha*s);
  dy[2]=y[2]+y[0]*(exp(A*s)-1.0)/A;

  return SDE_WA_SUCCESS;

}
```

Since use of closed-form approximation effectively speeds up the calculation for the NV algorithm, it is strongly recommended to pursue explicit forms or this type of approximation as long as the NV algorithm is considered.

- **Instantiation of SDE system**
  The functions which are related to the considered SDE and are defined above are to be used for instantiation of `sde`.

  As is often the case, some algorithms can be applied to one common SDE system for the purpose of comparison or some reasons. Such an SDE system should have all the definitions of functions which could be used in any algorithm.

cc11-EM, NV, NN (sde_wa_manual_program_ah_nn.c)

```
      sde->sde_type=STR;
      sde->V[0]=ah_str_V_0;
      sde->V[1]=ah_V_1;
      sde->V[2]=ah_V_2;
      sde->drift_corrector[0]=NULL;
      sde->drift_corrector[1]=diff_ah_V_1;
      sde->drift_corrector[2]=diff_ah_V_2;
      sde->exp_sV[0]=exp_ah_sVy_0;
      sde->exp_sV[1]=exp_ah_sVy_1;
      sde->exp_sV[2]=exp_ah_sVy_2;
```

cc11-NV, NN

```
      sde->sde_type=STR;
      sde->V[0]=ah_str_V_0;
      sde->V[1]=ah_V_1;
      sde->V[2]=ah_V_2;
      sde->drift_corrector[0]=NULL;
      sde->drift_corrector[1]=NULL;
      sde->drift_corrector[2]=NULL;
      sde->exp_sV[0]=exp_ah_sVy_0;
      sde->exp_sV[1]=exp_ah_sVy_1;
      sde->exp_sV[2]=exp_ah_sVy_2;
```

If the SDE system is defined by **cc11-EM**, neither the NV nor the NN is applicable.

cc11-EM

```
      sde->sde_type=ITO;
      sde->V[0]=ah_V_0;
      sde->V[1]=ah_V_1;
      sde->V[2]=ah_V_2;
      sde->drift_corrector[0]=NULL;
      sde->drift_corrector[1]=NULL;
      sde->drift_corrector[2]=NULL;
      sde->exp_sV[0]=NULL;
      sde->exp_sV[1]=NULL;
      sde->exp_sV[2]=NULL;
```

4.1.2. *Instantiation of one-step calculator.* We instantiate a one-step calculator for each algorithm by **cc12-EM**, **cc12-NV**, or **cc12-NN**, with the Romberg extrapolation which indicates that the seventh-order Runge–Kutta method is applied in the NV or the NN algorithm.

cc12-EM

```
enum ALG alg = E_M;
int m_moment = 7; /* whichever 5 or 7 */
SDE_WA_SLTN *sl;
sl=alloc_SDE_WA_SLTN(alg, m_moment, sde);
```

cc12-NV

```
enum ALG alg = N_V;
int m_moment = 7;
SDE_WA_SLTN *sl;
sl=alloc_SDE_WA_SLTN(alg, m_moment, sde);
```

cc12-NN (sde_wa_manual_program_ah_nn.c)

```
enum ALG alg = N_N;
int m_moment = 7;
SDE_WA_SLTN *sl;
sl=alloc_SDE_WA_SLTN(alg, m_moment, sde);
```

4.1.3. *Arguments to be used in approximation.* `next_SDE_WA` is a function to do one-step calculation. We recall

```
int next_SDE_WA(SDE_WA_SLTN *X, double s, double y[], double dy[], void *rv);
```

Since we have already constructed a one-step calculator, we should define the rest four arguments here.

- **Time interval**
  In this example, we use equidistant partition of $[0, T]$ with $T = 1.0$. Since the Romberg extrapolation is to be applied in this example, we have two kinds of interval depending on the numebr of partitions. Here `dt` denotes the time interval.

  cc13 (sde_wa_manual_program_ah_nn.c)

  ```
  double dt=1.0/(double)n;
  ```

  cc14 (sde_wa_manual_program_ah_nn.c)

  ```
  dt=2.0/(double)n;
  ```

- **Memory for an initial vector and result of one-step calculation**
  We allocate two-dimensional arrays `x` for `n`-partition calculation and `xR` for `n/2`-partition calculation (for the Romberg extrapolation).

cc15 (sde_wa_manual_program_ah_nn.c)

```
  /* x[0]:initial x[1]:destination(n partitions) */
    double **x;

  /* xR[0]:initial xR[1]:destination(n/2 partitions)*/
    double **xR;

    x=(double **)malloc(sizeof(double *)*2);
    for (i=0; i<2; i++)
    x[i]=(double *)malloc(sizeof(double)*sde->dim_y);

    xR=(double **)malloc(sizeof(double *)*2);
    for (i=0; i<2; i++)
    xR[i]=(double *)malloc(sizeof(double)*sde->dim_y);
```

- **Sample point** sp
  Let $D(d)$ be the number of random variables included in one-step calculation of each algorithm. Then

$$D(d) = \begin{cases} d & \text{if} \quad \text{EM} \\ d+1 & \text{if} \quad \text{NV} \\ 2d & \text{if} \quad \text{NN} \end{cases} \tag{4.5}$$

A $D(d)$-dimensional sample point vector is used for one step. Hence, a $(D(d) \times n)$-dimensional low-discrepancy sequence (or $D(d) \times n$ pseudorandom numbers) is needed for n steps. When the Romberg extrapolation is applied, a $(D(d) \times (n + n/2))$-dimensional low-discrepancy sequence (or $D(d) \times (n + n/2)$ pseudorandom numbers) should be given for all calculation for one sample.

In this example, we take a Sobol sequence generated through GNU Scientific Library. u of u_seq stands for **u**niform distribution and n of n_seq does for **n**ormal distribution in the following source codes. Also, B in u_seqB represents Bernoulli distribution. (/******/ represents some lines for other calculation)

cc16-EM

```
gsl_qrng *q;
double *u_seq1, *u_seq2;
double *n_seq1, *n_seq2;
double *sp; /* sample point for 1 step*/
sp=(double *)malloc(sizeof(double)*sde->dim_BM);

u_seq1=(double *)malloc(sizeof(double)*(n+n/2)
                                *sde->dim_BM);
u_seq2=u_seq1+(n+n/2);
n_seq1=(double *)malloc(sizeof(double)*(n+n/2)
                               *sde->dim_BM);
n_seq2=n_seq1+(n+n/2);

q=gsl_qrng_alloc(gsl_qrng_sobol,
                 (n+n/2)*sde->dim_BM);
```

cc16-NV

```
gsl_qrng *q;
double *u_seq1, *u_seq2, *u_seqB;
double *n_seq1, *n_seq2;
RV_NV *sp; /* sample point for 1 step*/
sp.rv_nv=(double *)malloc(sizeof(double)
                                 *sde->dim_BM);

u_seq1=(double *)malloc(sizeof(double)*(n+n/2)
                                *(sde->dim_BM+1));
u_seq2=u_seq1+(n+n/2);
u_seqB=u_seq1+(n+n/2)*sde->dim_BM;
n_seq1=(double *)malloc(sizeof(double)*(n+n/2)
                               *(sde->dim_BM+1));
n_seq2=n_seq1+(n+n/2);

q=gsl_qrng_alloc(gsl_qrng_sobol,
                 (n+n/2)*(sde->dim_BM+1));
```

cc16-NN (sde_wa_manual_program_ah_nn.c)

```
    gsl_qrng *q;
    double *u_seq1, *u_seq2;
    double *n_seq1, *n_seq2;
    double *sp; /* sample point for 1 step*/
    sp=(double *)malloc(sizeof(double)*sde->dim_BM*2);

    u_seq1=(double *)malloc(sizeof(double)*(n+n/2)
                                *sde->dim_BM*2);
    u_seq2=u_seq1+(n+n/2)*sde->dim_BM;
    n_seq1=(double *)malloc(sizeof(double)*(n+n/2)
                               *sde->dim_BM*2);
    n_seq2=n_seq1+(n+n/2)*sde->dim_BM;

    q=gsl_qrng_alloc(gsl_qrng_sobol,
                     (n+n/2)*sde->dim_BM*2);
```

cc17-EM

```
      gsl_qrng_get(q, u_seq1);

      for (k=0; k<n+n/2; k++){
          n_seq1[k] = sqrt(-2.0*log(u_seq1[k]))
                      *cos(2.0*M_PIl*u_seq2[k]);
          n_seq2[k] = sqrt(-2.0*log(u_seq1[k]))
                      *sin(2.0*M_PIl*u_seq2[k]);
      }/* for k */

      /* for n partitions */
      for (k=0; k< n; k++){

       /********************************/

         sp[0]=n_seq1[k];
         sp[1]=n_seq2[k];

       /********************************/

       }/* for k */

       /********************************/

       /* for n/2 partitions */
      for (k=0; k< n/2; k++){

       /********************************/
         sp[0]=n_seq1[n+k];
         sp[1]=n_seq2[n+k];

       /********************************/

      } /* for k */
```

cc17-NV

```
     gsl_qrng_get(q, u_seq1);

     for (k=0; k<n+n/2; k++){
         n_seq1[k] = sqrt(-2.0*log(u_seq1[k]))
                     *cos(2.0*M_PIl*u_seq2[k]);
         n_seq2[k] = sqrt(-2.0*log(u_seq1[k]))
                     *sin(2.0*M_PIl*u_seq2[k]);
     }/* for k */

     /* for n partitions */
     for (k=0; k< n; k++){

      /********************************/
        if(u_seqB[k]>=0.5) sp.rv_nv_b=T;
        else sp.rv_nv_b=H;
        sp.rv_nv_n[0]=n_seq1[k];
        sp.rv_nv_n[1]=n_seq2[k];

      /********************************/

     }/* for k */

      /********************************/

       /* for n/2 partitions */
     for (k=0; k< n/2; k++){

      /********************************/

       if(u_seqB[n+k]>=0.5) sp.rv_nv_b=0;
       else sp.rv_nv_b=1;
       sp[0]=n_seq1[n+k];
       sp[1]=n_seq2[n+k];

      /********************************/

     } /* for k */
```

cc17-NN (sde_wa_manual_program_ah_nn.c)

```
      gsl_qrng_get(q, u_seq1);

      for (k=0; k<n+n/2; k++){
          n_seq1[k] = sqrt(-2.0*log(u_seq1[k]))
                      *cos(2.0*M_PIl*u_seq2[k]);
          n_seq2[k] = sqrt(-2.0*log(u_seq1[k]))
                      *sin(2.0*M_PIl*u_seq2[k]);
      }/* for k */

      /* for n partitions */
      for (k=0; k< n; k++){

       /*******************************/

         sp[0]=n_seq1[k];
         sp[1]=n_seq1[n+n/2+k];
         sp[2]=n_seq2[k];
         sp[3]=n_seq2[n+n/2+k];

       /*******************************/

      }/* for k */

       /*******************************/

        /* for n/2 partitions */
      for (k=0; k< n/2; k++){

       /*******************************/

         sp[0]=n_seq1[n+k];
         sp[1]=n_seq1[n+n/2+k];
         sp[2]=n_seq2[n+k];
         sp[3]=n_seq2[n+n/2+k];

       /*******************************/

      } /* for k */
```

4.1.4. *n-step calculation of M samples.* There are a couple of differences among **cc18-EM**, **cc18-NV**, and **cc18-NN** such as

- the coefficients used for the Romberg extrapolation in the EM scheme differ from those in the other two algorithms because of the difference of order
- a sample point used in the NV algorithm has special form.

/**********/ represents some lines for other calculation such as calculation of sp.

cc18-EM

```
     int M;
     double sum;
     double rom_weight1=2.0, rom_weight2=1.0;
     double *tmp_pt;

     for (sum=0.0, i=0; i < M; i++){

       /*********************************************/
       for (x[0][0]=x0, x[0][1]=x1, x[0][2]=0.0, k=0; k< n;
            k++){

         /******************************************/
         next_SDE_WA(sl, dt, x[0], x[1], sp);
         tmp_pt=x[0];
         x[0]=x[1];
         x[1]=tmp_pt;
       }/* for k */

       for (xR[0][0]=x0, xR[0][1]=x1, xR[0][2]=0.0, k=0;
            k< n/2; k++){

         /******************************************/
         next_SDE_WA(sl, dt, xR[0], xR[1], sp);
         tmp_pt=xR[0];
         xR[0]=xR[1];
         xR[1]=tmp_pt;
       } /* for k */

       sum+=rom_weight1*asian_heston_call_payoff(x[0][2],K)
         -rom_weight2*asian_heston_call_payoff(xR[0][2],K);
     } /* for i */
```

cc18-NV

```
      int M;
      double sum;
      double rom_weight1=4.0/3.0, rom_weight2=1.0/3.0;
      double *tmp_pt;

      for (sum=0.0, i=0; i < M; i++){

        /**********************************************/
        for (x[0][0]=x0, x[0][1]=x1, x[0][2]=0.0, k=0; k< n;
             k++){

          /********************************************/
          next_SDE_WA(sl, dt, x[0], x[1], &sp);
          tmp_pt=x[0];
          x[0]=x[1];
          x[1]=tmp_pt;
        }/* for k */

        for (xR[0][0]=x0, xR[0][1]=x1, xR[0][2]=0.0, k=0;
             k< n/2; k++){

          /********************************************/
          next_SDE_WA(sl, dt, xR[0], xR[1], &sp);
          tmp_pt=xR[0];
          xR[0]=xR[1];
          xR[1]=tmp_pt;
        } /* for k */

        sum+=rom_weight1*asian_heston_call_payoff(x[0][2],K)
          -rom_weight2*asian_heston_call_payoff(xR[0][2],K);
      } /* for i */
```

cc18-NN (sde_wa_manual_program_ah_nn.c)

```
      int M;
      double sum;
      double rom_weight1=4.0/3.0, rom_weight2=1.0/3.0;
      double *tmp_pt;

      for (sum=0.0, i=0; i < M; i++){

        /**********************************************/
        for (x[0][0]=x0, x[0][1]=x1, x[0][2]=0.0, k=0; k< n;
             k++){

          /**********************************************/
          next_SDE_WA(sl, dt, x[0], x[1], sp);
          tmp_pt=x[0];
          x[0]=x[1];
          x[1]=tmp_pt;
        }/* for k */

        for (xR[0][0]=x0, xR[0][1]=x1, xR[0][2]=0.0, k=0;
             k< n/2; k++){

          /**********************************************/
          next_SDE_WA(sl, dt, xR[0], xR[1], sp);
          tmp_pt=xR[0];
          xR[0]=xR[1];
          xR[1]=tmp_pt;
        } /* for k */

        sum+=rom_weight1*asian_heston_call_payoff(x[0][2],K)
          -rom_weight2*asian_heston_call_payoff(xR[0][2],K);
      } /* for i */
```

4.1.5. *Freeing objects.* After $M \times (n + n/2)$ iterations of next_SDE_WA, we can free all memory for sl and sde paying attention to the order of freeing objects as Remark 3.2.

cc19 (sde_wa_manual_program_ah_nn.c)

```
free_SDE_WA_SLTN(sl);
free_SDE_WA_SYSTEM(sde);
```

4.2. **Stochastic Area.** The stochastic area is defined by

$$A(t) := \frac{1}{2}\left(\int_0^t B^2(s)\,\mathrm{d}B^1(s) - \int_0^t B^1(s)\,\mathrm{d}B^2(s)\right) \tag{4.6}$$

where $\left(B^1(t), B^2(t)\right)$ is a two-dimensional standard Brownian motion.

In order to calculate $E\left[f(A(T))\right]$, we consider the three-dimensional SDE written in the form by

$$
\begin{aligned}
X^1(t,x) &= \int_0^t 0\,\mathrm{d}s + \int_0^t \mathrm{d}B^1(s) + \int_0^t 0\,\mathrm{d}B^2(s)\\
X^2(t,x) &= \int_0^t 0\,\mathrm{d}s + \int_0^t 0\,\mathrm{d}B^1(s) + \int_0^t \mathrm{d}B^2(s)\\
X^3(t,x) &= \int_0^t 0\,\mathrm{d}s + \frac{1}{2}\int_0^t X^2(s)\,\mathrm{d}B^1(s) - \frac{1}{2}\int_0^t X^1(s)\,\mathrm{d}B^2(s),
\end{aligned}
\tag{4.7}
$$

where $\left(B^1(t), B^2(t)\right)$ is a two-dimensional standard Brownian motion (i.e. $d = 2$). There is no parameter in this SDE.
This example is featured by the following two facts:

- there is no difference between the Ito SDE and the Stratonovich SDE.
- all $\exp(sV_i)\,y$'s have explicit forms.

4.2.1. *Allocation and instantiation of SDE system.* From (4.7), we notice that the spacial dimension is 3, the dimension of Brownian motion is 2, and there is not any parameter. From these facts, we have all arguments for alloc_SDE_WA_SYSTEM.

cc20 (sde_wa_manual_program_sa_nv.c)

```
SDE_WA_SYSTEM *sde;
sde=alloc_SDE_WA_SYSTEM(3, 2, NULL);
```

- **Definitions of V_0, V_1, and V_2**
  We give the definition of V_0, V_1, and V_2 here.

$$
V_0\begin{pmatrix} y_1\\ y_2\\ y_3\end{pmatrix} = \begin{pmatrix} 0\\ 0\\ 0\end{pmatrix},\ V_1\begin{pmatrix} y_1\\ y_2\\ y_3\end{pmatrix} = \begin{pmatrix} 1\\ 0\\ \frac{1}{2}y_2\end{pmatrix},\ V_2\begin{pmatrix} y_1\\ y_2\\ y_3\end{pmatrix} = \begin{pmatrix} 0\\ 1\\ -\frac{1}{2}y_1\end{pmatrix}.
$$

Then these correspond to sa_V_0, sa_V_1, and sa_V_2,respectively, defined as follows:

cc21 (sde_wa_manual_program_sa_system.c)

```
int sa_V_0(const double y[], double dy[],
           void *params){

  dy[0]=0.0;
  dy[1]=0.0;
  dy[2]=0.0;

  return SDE_WA_SUCCESS;

}
```

cc22 (sde_wa_manual_program_sa_system.c)

```
int sa_V_1(const double y[], double dy[],
           void *params){

  dy[0]=1.0;
  dy[1]=0.0;
  dy[2]=0.5*y[1];

  return SDE_WA_SUCCESS;
}
```

cc23 (sde_wa_manual_program_sa_system.c)

```
int sa_V_2(const double y[], double dy[],
           void *params){

  dy[0]=0.0;
  dy[1]=1.0;
  dy[2]=-0.5*y[0];

  return SDE_WA_SUCCESS;
}
```

- **Definitions of partial derivatives of V_1, and V_2**
  As we have seen in Table 5 and Table 6, there is no need of $\frac{\partial V_i(y)}{\partial y}$ as long as we focus only on the NV or the NN algorithm and (4.7) is regareded as a Stratonovich SDE. Also, we can avoid giving definitions of $\frac{\partial V_i(y)}{\partial y}$ even for the EM scheme by regarding (4.7) as the Ito SDE by letting $\tilde{V}_0 = V_0$.
  In order to construct one SDE system to be used in common by every algorithm, we give definitions of $\frac{\partial V_1(y)}{\partial y}$ and $\frac{\partial V_2(y)}{\partial y}$ here.
  Let

$$\frac{\partial V_i}{\partial y}(y) = \left(a_{jk}\right)_{j,k\in\{1,\ldots,N\}} \quad \text{with} \quad a_{jk} = \frac{\partial V_i^j}{\partial y_k}(y).$$

Then,

$$\frac{\partial V_1}{\partial y}(y) = \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 1/2 & 0 \end{pmatrix}, \ \frac{\partial V_2}{\partial y}(y) = \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ -1/2 & 0 & 0 \end{pmatrix}.$$

These are defined as follows:

cc24 (sde_wa_manual_program_sa_system.c)

```
int diff_sa_V_1(const double y[], double *dVdy[],
                void *params){
  dVdy[0][0]=0.0;
  dVdy[0][1]=0.0;
  dVdy[0][2]=0.0;

  dVdy[1][0]=0.0;
  dVdy[1][1]=0.0;
  dVdy[1][2]=0.0;

  dVdy[2][0]=0.0;
  dVdy[2][1]=0.5;
  dVdy[2][2]=0.0;

  return SDE_WA_SUCCESS;
}
```

cc25 (sde_wa_manual_program_sa_system.c)

```
int diff_sa_V_2(const double y[], double *dVdy[],
                void *params){
  dVdy[0][0]=0.0;
  dVdy[0][1]=0.0;
  dVdy[0][2]=0.0;

  dVdy[1][0]=0.0;
  dVdy[1][1]=0.0;
  dVdy[1][2]=0.0;

  dVdy[2][0]=-0.5;
  dVdy[2][1]=0.0;
  dVdy[2][2]=0.0;

  return SDE_WA_SUCCESS;
}
```

- **Definition of an explicit form of each ODE with V_i**

  All V_i's in this example have the explicit forms of $\exp(sV_i)\, y$'s:

$$\exp(tV_0)\begin{pmatrix} y_1 \\ y_2 \\ y_3 \end{pmatrix} = \begin{pmatrix} y_1 \\ y_2 \\ y_3 \end{pmatrix}, \ \exp(tV_1)\begin{pmatrix} y_1 \\ y_2 \\ y_3 \end{pmatrix} = \begin{pmatrix} y_1 + t \\ y_2 \\ y_3 + \frac{1}{2}y_2 t \end{pmatrix},$$

$$\exp(tV_2)\begin{pmatrix} y_1 \\ y_2 \\ y_3 \end{pmatrix} = \begin{pmatrix} y_1 \\ y_2 + t \\ y_3 - \frac{1}{2}y_1 t \end{pmatrix}.$$

For these solutions, we define exp_sa_sVy_0, exp_sa_sVy_1, and exp_sa_sVy_2 as follows:

cc26 (sde_wa_manual_program_sa_system.c)

```
int exp_sa_sVy_0(double s, const double y[],
                 double dy[], void *params){

  dy[0]=y[0];
  dy[1]=y[1];
  dy[2]=y[2];

  return SDE_WA_SUCCESS;
 }
```

cc27 (sde_wa_manual_program_sa_system.c)

```
int exp_sa_sVy_1(double s, const double y[],
                 double dy[], void *params){

  dy[0]=y[0]+s;
  dy[1]=y[1];
  dy[2]=y[2]+0.5*y[1]*s;

  return SDE_WA_SUCCESS;
}
```

cc28 (sde_wa_manual_program_sa_system.c)

```
 int exp_sa_sVy_2(double s, const double y[],
                  double dy[], void *params){

  dy[0]=y[0];
  dy[1]=y[1]+s;
  dy[2]=y[2]-0.5*y[0]*s;

  return SDE_WA_SUCCESS;
}
```

- **Instantiation of SDE system**
  We instantiate sde by using the functions defined above.
  We should notice again that there is no difference between the Ito form and the Stratonovich form in this example.

cc29-EM, NV, NN

```
      sde->sde_type=STR;
      sde->V[0]=sa_V_0;
      sde->V[1]=sa_V_1;
      sde->V[2]=sa_V_2;
      sde->drift_corrector[0]=NULL;
      sde->drift_corrector[1]=diff_sa_V_1;
      sde->drift_corrector[2]=diff_sa_V_2;
      sde->exp_sV[0]=exp_sa_sVy_0;
      sde->exp_sV[1]=exp_sa_sVy_1;
      sde->exp_sV[2]=exp_sa_sVy_2;
```

cc29-NV, NN (sde_wa_manual_program_sa_nv.c)

```
      sde->sde_type=STR;
      sde->V[0]=sa_V_0;
      sde->V[1]=sa_V_1;
      sde->V[2]=sa_V_2;
      sde->drift_corrector[0]=NULL;
      sde->drift_corrector[1]=NULL;
      sde->drift_corrector[2]=NULL;
      sde->exp_sV[0]=exp_sa_sVy_0;
      sde->exp_sV[1]=exp_sa_sVy_1;
      sde->exp_sV[2]=exp_sa_sVy_2;
```

If the SDE system is defined by **cc29-EM**, neither the NV nor the NN is applicable.

cc29-EM

```
      sde->sde_type=ITO;
      sde->V[0]=sa_V_0;
      sde->V[1]=sa_V_1;
      sde->V[2]=sa_V_2;
      sde->drift_corrector[0]=NULL;
      sde->drift_corrector[1]=NULL;
      sde->drift_corrector[2]=NULL;
      sde->exp_sV[0]=NULL;
      sde->exp_sV[1]=NULL;
      sde->exp_sV[2]=NULL;
```

4.2.2. *Allocation of one-step calculator.* We allocate memory of a one-step calculator object for each algorithm. In this example, the order of the Runge–Kutta method is set to be 5, which indicates that the Romberg extrapolation is not to be applied.

Since explicit forms of all $\exp(sV_i)\,y$ are given to the SDE system in this example, numerical integration is not to be proceeded in the NV algorithm as well as the EM scheme.

cc30-EM

```
      enum ALG alg = E_M;
      int m_moment = 5;
      SDE_WA_SLTN *sl;
      sl=alloc_SDE_WA_SLTN(alg, m_moment, sde);
```

cc30-NV (sde_wa_manual_program_sa_nv.c)

```
      enum ALG alg = N_V;
      int m_moment = 5;
      SDE_WA_SLTN *sl;
      sl=alloc_SDE_WA_SLTN(alg, m_moment, sde);
```

cc30-NN

```
      enum ALG alg = N_N;
      int m_moment = 5;
      SDE_WA_SLTN *sl;
      sl=alloc_SDE_WA_SLTN(alg, m_moment, sde);
```

4.2.3. *Arguments to be used in approximation.* As in the previous example, we define `double dt`, `double y[]`, `double dy[]`, and `void *sp` (or `RV_NV sp`) which are to be arguments of `next_SDE_WA`.

- **Time interal**
  In this example, we use equidistant parition of $[0, T]$ with $T = 1.0$.

  cc31 (sde_wa_manual_program_sa_nv.c)

  ```
        double dt=1.0/(double)n;
  ```

- **Memory for an initial vector and result of one-step calculation**

  cc32 (sde_wa_manual_program_sa_nv.c)

  ```
        /* x[0]:initial x[1]:destination*/
        double **x;

        x=(double **)malloc(sizeof(double *)*2);
        for (i=0; i<2; i++)
         x[i]=(double *)malloc(sizeof(double)*sde->dim_y);
  ```

- **Sample point**
  An $(nD(d))$-dimensional low-discrepancy sequence (or $nD(d)$ pseudorandom numbers) is used for n-step calculation. In this example, we take a low-discrepancy sequence generated through GNU Scientific Library as the previous example.
  (/**********/ represents some lines for other calculation.)

cc33-EM

```
gsl_qrng *q;
double *u_seq1, *u_seq2;
double *n_seq1, *n_seq2;
double *sp;

sp=(double *)malloc(sizeof(double)*sde->dim_BM);
u_seq1=(double *)malloc(sizeof(double)*n
                        *sde->dim_BM);
u_seq2=u_seq1+n;
n_seq1=(double *)malloc(sizeof(double)*n
                        *sde->dim_BM);
n_seq2=n_seq1+n;

q=gsl_qrng_alloc(gsl_qrng_sobol,
                 sde->dim_BM*n);
```

cc33-NV (sde_wa_manual_program_sa_nv.c)

```
gsl_qrng *q;
double *u_seq1, *u_seq2, *u_seqB;
double *n_seq1, *n_seq2;
RV_NV sp;

sp.rv_nv_n
   =(double *)malloc(sizeof(double)*sde->dim_BM);
u_seq1=(double *)malloc(sizeof(double)*n
                        *(sde->dim_BM+1));
u_seq2=u_seq1+n;
u_seqB=u_seq1+2*n;
n_seq1=(double *)malloc(sizeof(double)*n
                        *(sde->dim_BM+1));
n_seq2=n_seq1+n;

q=gsl_qrng_alloc(gsl_qrng_sobol,
                 (sde->dim_BM+1)*n);
```

cc33-NN

```
gsl_qrng *q;
double *u_seq1, *u_seq2;
double *n_seq1, *n_seq2;
double *sp;

sp=(double *)malloc(sizeof(double)*2*sde->dim_BM);
u_seq1=(double *)malloc(sizeof(double)*n
                        *2*sde->dim_BM);
u_seq2=u_seq1+n*sde->dim_BM;
n_seq1=(double *)malloc(sizeof(double)*n
                        *2*sde->dim_BM);
n_seq2=n_seq1+n*sde->dim_BM;

q=gsl_qrng_alloc(gsl_qrng_sobol,
                 2*sde->dim_BM*n);
```

cc34-EM

```
gsl_qrng_get(q, u_seq1);
for (k=0; k<n; k++){
    n_seq1[k]=sqrt(-2.0*log(u_seq1[k]))
             *cos(2.0*M_PIl*u_seq2[k]);
    n_seq2[k]=sqrt(-2.0*log(u_seq1[k]))
             *sin(2.0*M_PIl*u_seq2[k]);
}/* for k */
for (k=0;k<n; k++){

  /*****************************/

   sp[0]=n_seq1[k];
   sp[1]=n_seq2[k];

  /*****************************/
} /* for k */
```

cc34-NV (`sde_wa_manual_program_sa_nv.c`)

```
      gsl_qrng_get(q, u_seq1);
      for (k=0; k<n; k++){
          n_seq1[k]=sqrt(-2.0*log(u_seq1[k]))
                    *cos(2.0*M_PIl*u_seq2[k]);
          n_seq2[k]=sqrt(-2.0*log(u_seq1[k]))
                    *sin(2.0*M_PIl*u_seq2[k]);
        }/* for k */
      for (k=0;k<n; k++){

         /******************************/
          if(u_seqB[k]>=0.5) sp.rv_nv_b=0;
          else sp.rv_nv_b=1;
          sp.rv_nv_n[0]=n_seq1[k];
          sp.rv_nv_n[1]=n_seq2[k];

         /******************************/
        } /* for k */
```

cc34-NN

```
      gsl_qrng_get(q, u_seq1);
      for (k=0; k<n*sde->dim_BM; k++){
         n_seq1[k]=sqrt(-2.0*log(u_seq1[k]))
                    *cos(2.0*M_PIl*u_seq2[k]);
         n_seq2[k]=sqrt(-2.0*log(u_seq1[k]))
                    *sin(2.0*M_PIl*u_seq2[k]);
      }/* for k */

      for (k=0;k<n; k++){

        /******************************/

         sp[0]=n_seq1[k];
         sp[1]=n_seq2[n+k];
         sp[2]=n_seq2[k];
         sp[3]=n_seq2[n+k];

        /******************************/
      } /* for k */
```

4.2.4. *n-step calculations of M samples.* Since the Romberg extrapolation is not applied in this example, there is no difference in procedures of n-step calculation of M samples by the EM scheme and the NN algorithm. The only difference between the NV and the other two is the way of giving a sample point to `next_SDE_WA`.

cc35-EM-NN

```
int M;
double sum;
double *tmp_pt
for (sum=0.0, i=0; i< M; i++){

  /************************************************/
   for (x[0][0]=x0, x[0][1]=x1, x[0][2]=0.0, k=0; k<n;
        k++){

        /**************************************/
        next_SDE_WA(sl, dt, x[0], x[1], sp);
        tmp_pt=x[0];
        x[0]=x[1];
        x[1]=tmp_pt;

    }/* for k */
    sum +=pf_f(x[0][2], K);
 }/* for i */
 printf("\n%.12e\n", sum/(double)M);
```

cc35-NV (sde_wa_manual_program_sa_nv.c)

```
int M;
double sum;
double *tmp_pt
for (sum=0.0, i=0; i< M; i++){

  /************************************************/
   for (x[0][0]=x0, x[0][1]=x1, x[0][2]=0.0, k=0; k<n;
        k++){

        /**************************************/
        next_SDE_WA(sl, dt, x[0], x[1], &sp);
        tmp_pt=x[0];
        x[0]=x[1];
        x[1]=tmp_pt;

    }/* for k */
    sum +=pf_f(x[0][2], K);
 }/* for i */
 printf("\n%.12e\n", sum/(double)M);
```

4.2.5. *Freeing objects.* Freeing procedure is exactly the same as the previous example.

Acknowledgements

This research was partly supported by Center for Advanced Research in Finance at Graduate School of Economics, the University of Tokyo and by the Grant-in-Aid for Young Scientists(B) of Japan Society for the Promotion of Science, 23730297, 2011.

References

1. Peter E. Kloeden and Eckhard Platen, *Numerical Solution of Stochastic Differential Equations*, Springer Verlag, Berlin, 1999.
2. Shigeo Kusuoka, *Approximation of Expectation of Diffusion Process and Mathematical Finance*, Advanced Studies in Pure Mathematics, Proceedings of Final Taniguchi Symposium, Nara 1998 (T. Sunada, ed.), vol. 31, 2001, pp. 147–165.
3. Mariko Ninomiya and Syoiti Ninomiya, *A new weak approximation scheme of stochastic differential equations by using the Runge–Kutta method*, Finance and Stochastics **13** (2009), no. 3, 415–443.
4. Syoiti Ninomiya and Nicolas Victoir, *Weak Approximation of Stochastic Differential Equations and Application to Derivative Pricing*, Applied Mathematical Finance **15** (2008), 107–121.

Graduate School of Economics, The University of Tokyo, 7-3-1 Hongo, Bunkyo-ku, Tokyo 113-0033, Japan