User's guide for StocDeltaN

Yuichiro Tada[1,*]

[1] *Department of Physics, Nagoya University, Nagoya 464-8602, Japan*

(Dated: August 9, 2019)

I. CODE OVERVIEW

StocDeltaN is a powerful C++ package to analyze inflationary dynamics and calculate the power spectrum of curvature perturbations with use of the techniques of the stochastic-δN approach [1, 2]. This automatically includes resummation effects of super-horizon perturbations, enabling a non-perturbative approach to the curvature perturbation. This package is applicable to general multi-field models in non-trivial field spaces expressed by the general Lagrangian (1). StocDeltaN supports the full phase-space analysis by solving both the inflaton fields ϕ^I and their conjugate momenta π_I, while it automatically switch to the slow-roll field-space analysis if users do not specify the calculation box for the momenta. Though the slow-roll approximation may fail in some cases, it is computationally much more economic and often enough for leading order calculations.

StocDeltaN consists of two parts as the *source* part containing all required numerical solvers which we provide and the *main* part where users specify the inflationary model and the usage of several options like a plotting option with use of Python. Users can write the *main* code using various sample codes as references.

The *source* part in itself is divided into three parts as `JacobiPDE`, `SRK32`, and `StocDeltaN`. `JacobiPDE` and `SRK32` implement the solver classes for partial differential equations (PDE) and stochastic differential equations (SDE) respectively. Overriding these classes, users can employ the numerical solver for general problems beyond the inflationary system if want. `StocDeltaN` overrides their classes and defines the specific class as the stochastic-δN solver. The *main* code specifies the inflationary model, overriding its functions defining e.g. potential, field-space metric, and so on, and then user can use its member functions to analyze the stochastic inflation.

* tada.yuichiro@e.mbox.nagoya-u.ac.jp

II. TUTORIALS

A. Prerequisite

- **C++ compiler** : We have checked the operation in GNU, Clang, and Intel compiler on Mac system. GNU compiler will be used by default. On Mac system, generally Clang compiler is preinstalled and automatically called instead of GNU compiler, so users need not to install other compilers by themselves. By modifying Makefile, one can change the used compiler if wants.

- **Make** : For an easy compilation, StocDeltaN employs Make which is a compilation manager. On Mac system, it is easy to introduce GNU Make by installing Command Line Tools of Xcode.

- **OpenMP (optional)** : The modern concept of processor design places importance on parallelization over multicore rather than processing power on each single core. Multicore processors are implemented even on personal computers and parallelization gets much closer to end users. StocDeltaN employs automatic parallelization with use of OpenMP. Many representative compilers like GNU or Intel's one precontain OpenMP, and therefore users can make use of its parallelization simply by validating OpenMP option (e.g. -fopenmp for GNU compiler) in Makefile. Clang compiler preinstalled on Mac may not support OpenMP, then OpenMP option is commented out in a sample Makefile. But we strongly recommend the usage of OpenMP to bring out the full performance of StocDeltaN particularly in calculation on cluster computers.

- **Python & Matplotlib (optional)** : Though StocDeltaN exports all relevant data as DAT files, one can also use the automatic plotting system with use of Matplotlib which is Python's plotting library. Piping data to Python, StocDeltaN can plot the obtained results if Python and Matplotlib have been installed on a user's machine. Mac system has usually preinstalled Python, and therefore one can easily install Matplotlib as well by using Pip Installs Packages, e.g.

```
$ pip install matplotlib
```

B. First Exercise

Practice makes perfect. Let us start by downloading the source codes from our GitHub page https://github.com/NekomammaT/StocDeltaN_dist and play with attached samples. Using Make in the *sample* directory completes the compilation of the sample codes:

```
$ cd sample
$ make
```

By default the field-space double chaotic model (*double_chaotic_SR.cpp*) is compiled. Then running the executable file, one obtains some results like as follows.

```
$ ./double_chaotic_SR
[xi, xf, xmin, xmax]:
[13, 3.54334e-08, -5.06876e-07, 13]
[13, 1.40077, 1.40077, 13]

N = 84.01
Vi = 8.5543209876543225e-09,  Vf = 1.2111996327249413e-12
0.190044 sec.
```

If you could do this without any error, the machine environment is properly set up and you are ready to execute a main calculation.

One edits the following parts of *double_chaotic_SR.cpp*. Simply by commenting out `sdn.sample()` and validating `sdn.solve()` instead as indicated in List. 1, one can fully solve the system. Re-Making and running *double_chaotic_SR*, one obtains the analyzation results in *Mn_double_chaotic_SR.dat*, *traj_double_chaotic_SR.dat*, and *calP_double_chaotic_SR.dat*, which contain the data as shown in Lists. 2–4.

Listing 1. *sample/double_chaotic_SR.cpp*

```
82      //sdn.sample(); // obtain 1 sample path
83      //sdn.sample_plot(); // plot obtained sample path
84
85      sdn.solve(); // solve PDE & SDE to obtain power spectrum
86      //sdn.f_plot(0); // show plot of <N>
87      //sdn.f_plot(1); // show plot of <delta N^2>
88      //sdn.calP_plot(); // show plot of power spectrum of zeta
```

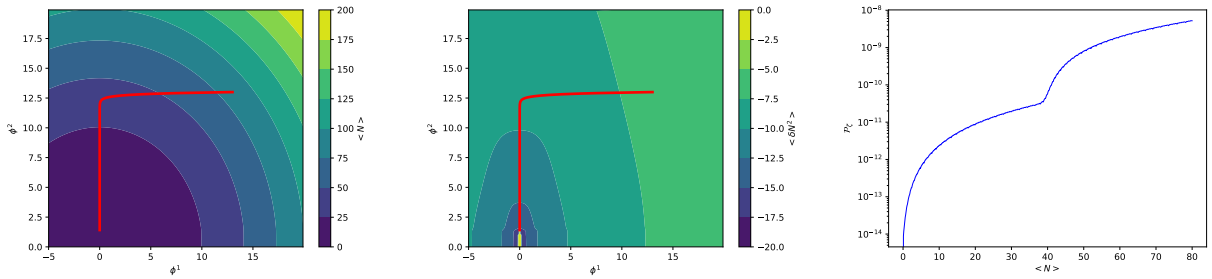

FIG. 1. *N_double_chaotic_SR.pdf*, *dN2_double_chaotic_SR.pdf*, and *calP_double_chaotic_SR.pdf* from left to right. Red lines in the left and middle panels show one realization of sample paths.

Listing 2. *Mn_*⟨model name⟩*.dat* : contour data of $\mathcal{M}_n = \langle \mathcal{N}^n \rangle$ and $\mathcal{C}_2 = \langle \delta\mathcal{N}^2 \rangle$

```
ϕ¹  ϕ²  ⋯  π₁  π₂  ⋯  M₁  C₂
```

Listing 3. *traj_*⟨model name⟩*.dat* : trajectory data of one sample path

```
N  ϕ¹  ϕ²  ⋯  π₁  π₂  ⋯
```

Listing 4. *calP_*⟨model name⟩*.dat* : data related to curvature perturbation

```
⟨N⟩  ⟨δN²⟩  P_ζ
```

If you have installed Python and Matplotlib already, the plotting option is quite useful. Validating `sdn.f_plot(0)`, `sdn.f_plot(1)`, and `sdn.calP_plot()` shown in List. 1, StocDeltaN pipes the data to Python and plots contours of $\mathcal{M}_1 = \langle \mathcal{N} \rangle$ and $\mathcal{C}_2 = \langle \delta\mathcal{N}^2 \rangle$ and the power spectrum of curvature perturbations $\mathcal{P}_\zeta$ as shown in Fig. 1. The plots are also saved as *N_double_chaotic_SR.pdf*, *dN2_double_chaotic_SR.pdf*, and *calP_double_chaotic_SR.pdf*. See Sec. III C for more details about the plotting functions.

To switch the analyzed model, one modifies *Makefile* in the *sample* directory. The `MODEL` parameter in *Makefile* as can be seen in List. 5 controls the analyzed model and one can calculate other models by changing this parameter. We preprovided four other models, *chaotic_SR*, *chaotic*, *hilltop_SR*, and *hybrid_SR*, as samples. One can use preferred C++ compiler by changing `CXX` parameter. The default setting is GNU compiler `g++` (On Mac system, it basically calls Clang compiler instead which may not support OpenMP. That is why we comment out OpenMP option `-fopenmp` by default). If you use a compiler supporting OpenMP, you may use it by validating the OpenMP option `-fopenmp` (or `-qopenmp` for Intel compiler).

Listing 5. *sample/Makefile*

```
MODEL = double_chaotic_SR
SOURCE = ../source

CXX := g++
CXXFLAGS := -std=c++11 -O3 #-fopenmp
```

C. Beyond samples

Of course the final goal is not playing with sample codes but solving your models. Though ultimately the understanding of the whole code structure is required to get a full command of StocDeltaN, we recommend to take advantage of sample codes as much as possible. That is, if you want to solve single-field models in the slow-roll limit, *chaotic_SR.cpp* or *hilltop_SR.cpp* would be helpful, while *double_chaotic_SR.cpp* or *hybrid_SR.cpp* can be used for two-field models. *chaotic.cpp* would be a reference for the phase-space approach. To adopt your model to StocDeltaN, you change the code in mainly two senses: one is the inflaton's system described by the Lagrangian, and the other is the numerical parameters used for practical calculations. We guide the readers in this way below.

1. *Lagrangian*

StocDeltaN supports general multi-scalar models in general relativity. Such models are parametrized the following scalar part Lagrangian.

$$\mathcal{L} = -\frac{1}{2}G_{IJ}(\phi)\partial_\mu\phi^I\partial^\mu\phi^J - V(\phi). \tag{1}$$

Here I and J label the scalar fields and G_{IJ} represents the inflaton's field space metric which can be curved in general. The conjugate momenta for the phase-space approach are defined by [YT : $G_{IJ}\frac{\mathrm{d}\phi^J}{\mathrm{d}N}$ is better?]

$$\pi_I = G_{IJ}\dot{\phi}^J. \tag{2}$$

This Langrangian is set in StocDeltaN by the corresponding functions in a main code represented by e.g. *double_chaotic_SR.cpp* as shown in List. 6. Each function represent:

- `V(X)` : the inflaton potential $V(\phi)$.

- `VI(X,I)` : its derivative $\partial_{\phi^I} V(\phi)$.
- `metric(X,I,J)` : the field space metric $G_{IJ}(\phi)$.
- `inversemetric(X,I,J)`: its inverse $G^{IJ}(\phi)$.
- `affine(X,I,J,K)` : the Christoffel symbol corresponding to the field space metric, $\Gamma^I_{JK}(\phi) = \frac{1}{2} G^{IL}(G_{JL,K} + G_{KL,J} - G_{JK,L})$.
- `derGamma(X,I,J,K,L)` : its derivative $\partial_{\phi^L} \Gamma^I_{JK}$.

double_chaotic_SR.cpp studies the double mass-term inflation with a flat field space as

$$V = \frac{1}{2} M^2 \phi^2 + \frac{1}{2} m^2 \psi^2, \qquad G_{IJ} = \delta_{IJ}. \tag{3}$$

In the code the list `X` represents the scalar fields as `X[0]` $= \phi$ and `X[1]` $= \psi$, then the functions are defined as shown in List. 6. The potential parameters M and m are defined at the top of the code as `MPHI` and `MPSI` to be easily modified. Therefore one can adopt the model interested simply by rewriting this part.

Listing 6. *sample/double_chaotic_SR.cpp*

```
20 // ---------- potential parameter ----------
21 #define MPHI (1e-5)
22 #define MPSI (MPHI/9.)
23 // -----------------------------------------
```

```
99  // ---------- Lagrangian params. X[0]=phi, X[1]=psi ----------
100
101 double StocDeltaN::V(vector<double> &X)
102 {
103   return 1./2*MPHI*MPHI*X[0]*X[0] + 1./2*MPSI*MPSI*X[1]*X[1];
104 }
105
106 double StocDeltaN::VI(vector<double> &X, int I) // \partial_I V
107 {
108   if (I == 0) {
109     return MPHI*MPHI*X[0];
110   } else {
111     return MPSI*MPSI*X[1];
112   }
113 }
114
```

```
double StocDeltaN::metric(vector<double> &X, int I, int J) // G_IJ
{
  if (I == J) {
    return 1;
  } else {
    return 0;
  }
}

double StocDeltaN::inversemetric(vector<double> &X, int I, int J) // G^IJ
{
  return metric(X,I,J);
}

double StocDeltaN::affine(vector<double> &X, int I, int J, int K) // \Gamma^I_JK
{
  return 0;
}

double StocDeltaN::derGamma(vector<double> &X, int I, int J, int K, int L) //  \partial_L \
    Gamma^I_JK
{
  return 0;
}
```

2. *lattice size*

The other important parameter for StocDeltaN is the lattice size defining the inflationary region. StocDeltaN adopts the orthogonal box with arbitrary variable lattice steps as shown in Fig. 2. The box can include the non-inflationary region where the corresponding energy density is lower than the threshold ρ_c shown by the blue shade in Fig. 2, other than the yellow inflationary region Ω. Note that the boundaries of Ω other than the end of inflation ρ_c are unphysical. StocDeltaN sets the reflecting boundary condition for them by default but users should choose the boundary position so that the relevant region is sufficiently far from those unphysical boundaries.

The box is parametrized by 3-dimensional list containing the field values of lattice sites in each scalar direction, i.e.,

```
{{{φ0, φ1, φ2, ···, φmax}, {ψ0, ψ1, ψ2, ···, ψmax}}}
```

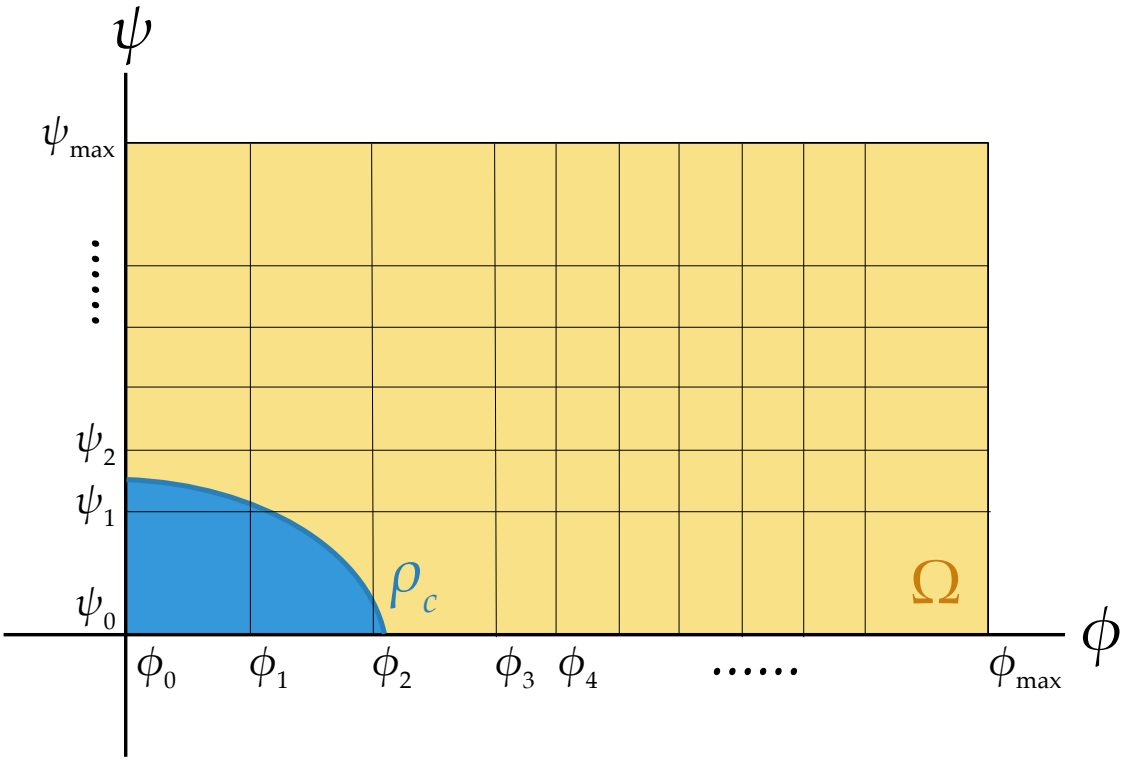

FIG. 2. Schematic image of the calculable lattice in StocDeltaN. Users declare each field value ϕ_0, ϕ_1, ϕ_2, $\cdots$, $\phi_{\rm max}$, and ψ_0, ψ_1, ψ_2, $\cdots$, $\psi_{\rm max}$.

for the SR field-space approach, while it also includes the lattice sites for the momenta as

```
{{{ϕ0, ϕ1, ϕ2, ···, ϕmax}, {ψ0, ψ1, ψ2, ···, ψmax}},
 {{π_ϕ0, π_ϕ1, π_ϕ2, ···, π_ϕmax}, {π_ψ0, π_ψ1, π_ψ2, ···, π_ψmax}}}
```

for the full phase-space approach in the case of Fig. 2. StocDeltaN automatically switches the approach, according to the presence of the lattice for momenta.

In *double_chaotic_SR.cpp*, the box boundary is defined by `PHIMIN`, `PHIMAX`, `PSIMIN`, and `PSIMAX` with the constant lattice steps `HPHI` and `HPSI`, declared at the top of the code as shown in List. 7. The concrete 3-dimensional list is created in the latter part as `sitepack`.

Listing 7. *sample/double_chaotic_SR.cpp*

```
6  // ---------- box size & step h ------------
7  #define PHIMIN -5
8  #define PHIMAX 20
9  #define PSIMIN 0
10 #define PSIMAX 20
11 #define HPHI 0.1
```

```
#define HPSI 0.1
// ----------------------------------------
```

```
    // ---------- set box ---------------
    double h = HPHI, sitev = PHIMIN;
    vector<double> site;
    vector< vector<double> > xsite;
    vector< vector< vector<double> > > sitepack;
    while (sitev <= PHIMAX) {
        site.push_back(sitev);
        sitev += h;
    }
    xsite.push_back(site);
    site.clear();

    h = HPSI, sitev = PSIMIN;
    while (sitev <= PSIMAX) {
        site.push_back(sitev);
        sitev += h;
    }
    xsite.push_back(site);
    site.clear();

    sitepack.push_back(xsite);
    xsite.clear();
    // ---------------------------------
```

One might prefer the logarithmic lattice to the constant step size. In such a case, it is useful to determine the step size by the field value itself. *hybrid_SR.cpp* employs this procedure in ψ's direction. In line 70 shown in List. 8, the step size in ψ's direction is determined by its ψ-value itself with a fraction `HPSIOPSI` and the lower bound `HPSIMIN`. Therefore the lattice step size becomes smaller for the smaller value of $|\psi|$, resulting in the logarithmic scale.

Listing 8. *sample/hybrid_SR.cpp*

```
// ---------- box size & step h ------------
#define PHIMIN 0.1409
#define PHIMAX 0.142
#define PSIMIN -(1e-3)
#define PSIMAX (1e-3)
#define HPHI (1e-5)
#define HPSIOPSI (1e-2) // hpsi/|psi|
```

```
#define HPSIMIN (1e-10)
// ----------------------------------------
```

```
    // ---------- set box ---------------
    double h = HPHI, sitev = PHIMIN;
    vector<double> site;
    vector< vector<double> > xsite;
    vector< vector< vector<double> > > sitepack;
    while (sitev <= PHIMAX) {
        site.push_back(sitev);
        sitev += h;
    }
    xsite.push_back(site);
    site.clear();

    sitev = PSIMIN;
    while (sitev <= PSIMAX) {
        h = max(fabs(sitev)*HPSIOPSI,HPSIMIN);

        site.push_back(sitev);
        sitev += h;
    }
    xsite.push_back(site);
    site.clear();

    sitepack.push_back(xsite);
    xsite.clear();
    // ----------------------------------
```

chaotic.cpp shown in List. 9 will be helpful for the box declaration in the phase-space approach. The logarithmic scale is adopted for the momentum. In any case, `site` corresponds with the most inner list, `xsite`/`xpsite` assembles the fields box or the momenta box, and `sitepack` is the full box list in our codes, but any other implementation also works as long as the final list is consistently configured as indicated in the beginning of this subsection.

Listing 9. *sample/chaotic.cpp*

```
// ---------- box size & step h ------------
#define XMIN 0
#define XMAX 15
#define PMIN (-1e-4)
#define PMAX 0
#define HX 0.01
#define HPMIN (1e-6)
```

```
#define HPOV (1./20)
// ----------------------------------------
```

```
    // ---------- set box ---------------
    double h = HX, sitev = XMIN;
    vector<double> site;
    vector< vector<double> > xpsite;
    vector< vector< vector<double> > > sitepack;
    while(sitev <= XMAX) {
        site.push_back(sitev);
        sitev += h;
    }
    xpsite.push_back(site);
    sitepack.push_back(xpsite);
    site.clear();
    xpsite.clear();

    sitev = PMIN;
    while (sitev <= PMAX) {
        h = max(fabs(sitev)*HPOV,HPMIN);

        site.push_back(sitev);
        sitev += h;
    }
    xpsite.push_back(site);
    sitepack.push_back(xpsite);
    site.clear();
    xpsite.clear();
    // ------------------------------------
```

3. other parameters

Other parameters are set at the top of the code and gathered into the parameter list `params` which is called for the declaration of the `StocDeltaN` class. The initial condition for the inflationary trajectories is declared by `xpi` in the form of

```
{{φ_i , ψ_i}}
```

or

```
{{φ_i , ψ_i}, {π_φi , π_ψi}}
```

List. 10 shows the sample code. The parameter list `params` consists of the maximal step number and tolerance for the PDE solver, the order of e-folds moment to be calculated, the energy surface of the end of inflation, the number of the independent noise degrees of freedom, e-folds time step for the trajectory calculation, the maximum and step of e-folds for the power spectrum calculation, and the recursion number of sample trajectories in the stochastic-δN approach. The detailed explanation of these parameters can be found in the next section.

Listing 10. *sample/double_chaotic_SR.cpp*

```
4  #define MODEL "double_chaotic_SR" // model name
```

```
15 // ---------- for PDE ----------------------
16 #define MAXSTEP 100000 // max recursion
17 #define TOL 1e-10 // tolerance
18 // -----------------------------------------
```

```
25 #define RHOC (MPSI*MPSI) // end of inflation
26
27 // ---------- for SDE ----------------------
28 #define RECURSION 100 // recursion for power spectrum
29 #define PHIIN 13 // i.c. for phi
30 #define PSIIN 13 // i.c. for psi
31 #define TIMESTEP (1e-2) // time step : delta N
32 // -----------------------------------------
33
34 // ---------- for power spectrum -----------
35 #define DELTAN 0.1 // calc. PS every DELTAN e-folds
36 #define NMAX 80 // calc. PS for 0--NMAX e-folds
37 // -----------------------------------------
```

```
76     vector<double> params = {MAXSTEP,TOL,2,RHOC,(double)sitepack[0].size(),TIMESTEP,NMAX,
           DELTAN,RECURSION}; // set paramters
77
78     vector< vector<double> > xpi = {{PHIIN,PSIIN}}; // set i.c. for inflationary
           trajectories
79
80     StocDeltaN sdn(MODEL,sitepack,xpi,0,params); // declare the system
```

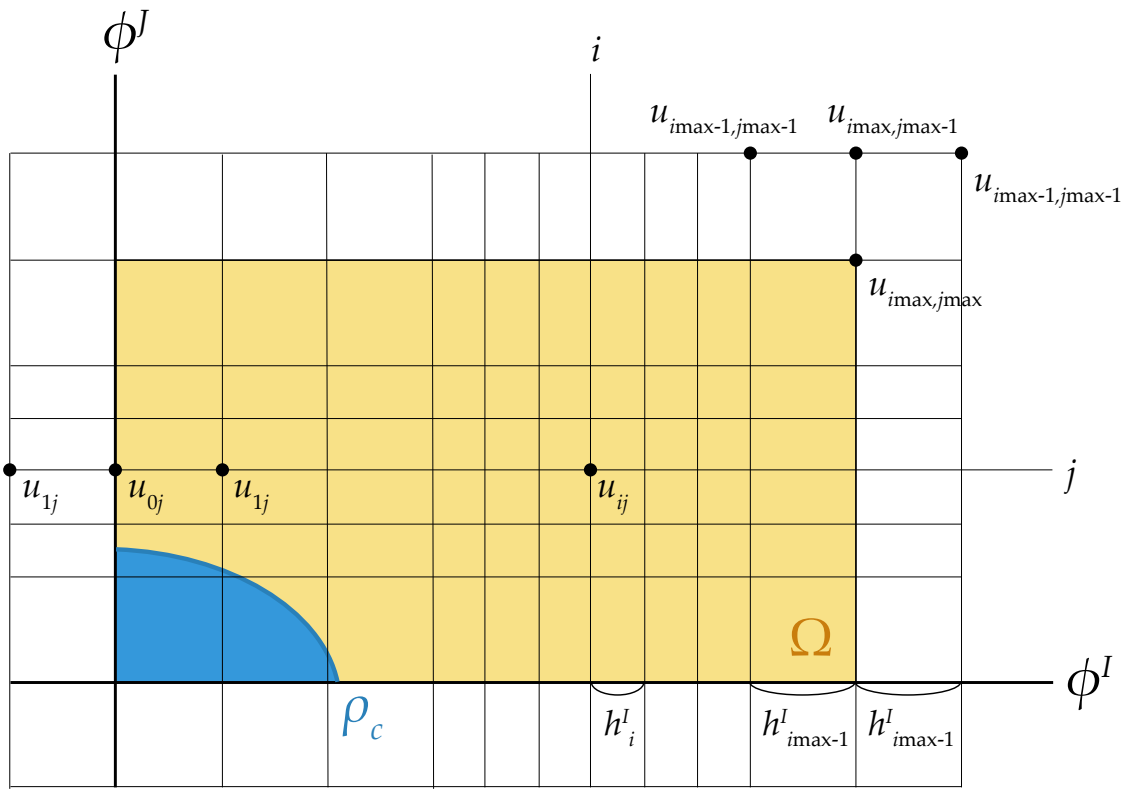

FIG. 3.

III. CODE STRUCTURES

Though the previous Tutorial section is basically enough for light users, one might desire to customize the codes for his/her own problem. In this section, the detailed structures of three parts (`JacobiPDE`, `SRK32`, and `StocDeltaN`) are described, suggesting possible arranges.

A. `JacobiPDE`

`JacobiPDE` is the partial differential equation (PDE) solver with use of the Jacobi method. The Jacobi method is a recursive procedure. The next step value of each site is determined by the nearest sites so that they locally satisfy the discretized PDE, and then all site values are updated simultaneously.

To describe this method in detail, let us reillustrate the lattice box in Fig. 3. The site point in the direction-I is parametrized by its small character i. The index I is naturally

generalized for momenta in a phase-space case. We first mention the boundary condition. The inflationary region Ω has two types of boundaries: the end of inflation ρ_c and the artificial box boundary $\partial\Omega$ due to the finiteness of the box. The end of inflation surface is determined by a certain value ρ_c of the inflatons' energy density since the last surface should be a uniform-density slice in the δN formalism [3] which we use. The calculables in the stochastic-δN is moments of the backward e-folds $\mathcal{N}$ from this end of inflation surface, and therefore they are set to zero on this boundary. Practically it is enough to fix the site values to zero if the corresponding energy density is equal to/lower than ρ_c (blue region):

$$u_{ij} = 0, \quad \text{(fixed and not updated)} \quad \text{for } \rho_{ij} \leq \rho_c. \tag{4}$$

On the other hand we choose the reflecting condition for the artificial boundary $\partial\Omega$ [4]. That is, the imaginary sites are set one-step outside of Ω and their values are equal to those of their reflective position sites to update the boundary values. These boundary conditions are controlled by the member function `BoundaryCondition()` and can be overwritten in `StocDeltaN` if users want (see also the `StocDeltaN` subsection III C).

Under these boundary conditions, each site value is updated according to the discretized PDE. PDEs in general can be expressed as (see Eqs. (12), (13) for concrete forms of D^I, D^{IJ}, and C)

$$\left(D^I(\phi)\partial_I + \frac{1}{2}D^{IJ}(\phi)\partial_I\partial_J\right)u(\phi) = C(\phi). \tag{5}$$

At the leading order, the first derivative can be discretized either by

$$\partial_I u \simeq \frac{u_{i+1} - u}{h_i} \simeq \frac{u - u_{i-1}}{h_{i-1}} \simeq \frac{u_{i+1} - u_{i-1}}{h_i + h_{i-1}}. \tag{6}$$

Here we only show the relevant indices. Second derivatives reads, depending on whether the direction is the same or not,

$$\begin{cases} \partial_I^2 u \simeq 2\dfrac{u_{i+1}h_{i-1} + u_{i-1}h_i - u(h_i + h_{i-1})}{h_i h_{i-1}(h_i + h_{i-1})}, \\ \partial_I\partial_J u \simeq \dfrac{u_{i+1,j+1} - u_{i+1,j-1} - u_{i-1,j+1} + u_{i-1,j-1}}{(h_i + h_{i-1})(h_j + h_{j-1})}, \quad (I \neq J). \end{cases} \tag{7}$$

Regarding the first derivative, one recalls that the physical boundary is set only on the end of inflation surface and therefore the PDE should be solved mainly in a backward way from this surface. Because D^I roughly corresponds with $-V^I$, u_{i+1} should be chosen at the site

where $D^I > 0$ (labeled i_+), and u_{i-1} otherwise (labeled i_-). Therefore the left-hand side of PDE (5) is discretized as

$$
\begin{aligned}
&\left(D^I\partial_I + \frac{1}{2}D^{IJ}\partial_I\partial_J\right)u \\
&\quad\simeq \sum_{i_+} D^I \frac{u_{i_++1} - u}{h_{i_+}} + \sum_{j_-} D^J \frac{u - u_{j_--1}}{h_{j_--1}} + \sum_I D^{II}\frac{u_{i+1}h_{i-1} + u_{i-1}h_i - u(h_i + h_{i-1})}{h_i h_{i-1}(h_i + h_{i-1})} \\
&\qquad + \frac{1}{2}\sum_{I\neq J} D^{IJ}\frac{u_{i+1,j+1} - u_{i+1,j-1} - u_{i-1,j+1} + u_{i-1,j-1}}{(h_i + h_{i-1})(h_j + h_{j-1})} \\
&\quad= \left(-\sum_{i_+}\frac{D^I}{h_{i_+}} + \sum_{j_-}\frac{D^J}{h_{j_--1}} - \sum_I \frac{D^{II}}{h_i h_{i-1}}\right)u \\
&\qquad - \left(-\sum_{i_+} D^I\frac{u_{i_++1}}{h_{i_+}} + \sum_{j_-} D^J\frac{u_{j_--1}}{h_{j_--1}} - \sum_I D^{II}\frac{u_{i+1}h_{i-1} + u_{i-1}h_i}{h_i h_{i-1}(h_i + h_{i-1})}\right. \\
&\qquad\qquad \left. -\frac{1}{2}\sum_{I\neq J} D^{IJ}\frac{u_{i+1,j+1} - u_{i+1,j-1} - u_{i-1,j+1} + u_{i-1,j-1}}{(h_i + h_{i-1})(h_j + h_{j-1})}\right) \\
&\quad=: Au - B.
\end{aligned}
\tag{8}
$$

Therefore the site value u is updated by

$$
u = \frac{B + C}{A}. \tag{9}
$$

Note that the right-hand side is independent of the corresponding site value u, so that this is indeed an explicit equation.

Though the updated value locally satisfies the discretized PDE, it is not the global solution. Instead of solving the local equations over whole sites simultaneously in an algebraic way, one can expect these local solutions to converge to the global one by updating recursively. In the Jacobi method, the next value is evaluated for all sites first, and then they are updated simultaneously. The convergence is assessed the error function defined by

$$
\mathrm{err} = \frac{|\Delta u|}{|u|}, \tag{10}
$$

where Δu represent the difference between the updated and previous value and their absolute values are defined by their root mean square. When this error becomes smaller than some preset tolerance, the local solution is assumed to represent the global one well.

In StocDeltaN, the following PDEs are solved by default:

$$\begin{cases} \left(D^I\partial_I + \frac{1}{2}D^{IJ}\partial_I\partial_J\right)\mathcal{M}_1 = -1, \\ \left(D^I\partial_I + \frac{1}{2}D^{IJ}\partial_I\partial_J\right)\mathcal{C}_2 = -D^{IJ}(\partial_I\mathcal{M}_1)(\partial_J\mathcal{M}_1), \end{cases} \tag{11}$$

where

$$D^I = -\frac{V^I}{V} - \frac{1}{2}\Gamma^I_{JK}D^{JK}, \qquad D^{IJ} = \frac{V}{12\pi^2}G^{IJ}, \tag{12}$$

in the field-space approach while

$$\begin{cases} D^I_\phi = \dfrac{\pi^I}{H} - \dfrac{1}{2}\Gamma^I_{JK}D^{JK}_{\phi\phi}, \\ D_{\pi I} = -3\pi_I - \dfrac{V_I}{H} + \Gamma^S_{IJ}\dfrac{\pi_S\pi^J}{H} + \dfrac{1}{2}(\Gamma^S_{IJ,K} - \Gamma^R_{JK}\Gamma^S_{IR} - \Gamma^R_{IJ}\Gamma^S_{KR})\pi_S D_{\phi\phi}{}^{JK} + \Gamma^J_{IK}D_{\phi\pi}{}^K{}_J, \\ D_{\phi\phi}{}^{IJ} = \left(\dfrac{H}{2\pi}\right)G^{IJ}, \qquad D_{\phi\pi}{}^I{}_J = \Gamma^K_{JL}\pi_K D_{\phi\phi}{}^{IL}, \qquad D_{\pi\pi IJ} = \Gamma^K_{IL}\Gamma^M_{JN}\pi_K\pi_M D_{\phi\phi}{}^{LN}. \end{cases} \tag{13}$$

We used $M_{\rm Pl} = 1$ unit. $\mathcal{M}_1$ and $\mathcal{C}_2$ correspond with $\langle\mathcal{N}\rangle$ and $\langle\delta\mathcal{N}^2\rangle$ respectively (see Ref. [5] for details). Note that the PDE for $\mathcal{C}_2$ (11) has derivatives also in the right-hand side. We adopt the last expression of Eq. (6) to discretize them.

Let us finally make descriptions of the member functions declared in *JacobiPDE.hpp*

- `JacobiPDE(Site,Params)` : Constructor setting initial conditions for the considered system. By default `JacobiPDE` requires four parameters: `Params[0]` is the maximum recursion number for the Jacobi method, `Params[1]` is the tolerance for the convergence error (10), `Params[2]` specifies how many moments of $\mathcal{N}$ are calculated (`Params[2]` $= 1$ solves only $\mathcal{M}_1$ while `Params[2]` $= 2$ solves both $\mathcal{M}_2$ and $\mathcal{C}_2$), and `Params[3]` is the energy density of the end of inflation ρ_c.

- `PDE_1step(num,func)` :

- `PDE_solve(func)` :

- `Ind2No(index)` :

- `No2Ind(num,xp,I)` :

- `No2PSV(num,xp,I)` :
- `ceilXP(xp,I,psv)` :
- `Interpolation_f(psv,func)` :
- `export_fg(filename)` :

The following functions are virtual so that users can easily overwrite them in an overriding class like `StocDeltaN`. In StocDeltaN case for example, if one wants to redefine them, validate their declarations in *StocDeltaN.hpp* and redefine them in *StocDeltaN.cpp* or one's main code (refer to `V`, `VI`, `metric`, `inversemetric`, `affine`, and `derGamma` in the sample codes).

- `H(X,P)` :
- `V(X)` :
- `VI(X,I)` :
- `metric(X,I,J)` :
- `inversemetric(X,I,J)` :
- `affine(X,I,J,K)` :
- `derGamma(X,I,J,K,L)` :
- `DI(xp,I,psv)` :
- `DIJ(xpI,I,xpJ,J,psv)` :
- `CC(num,psv,func)` :
- `BoundaryCondition()` :
- `EndSurface(psv)` :

B. `SRK32`

description of SRK32
member function

C. `StocDeltaN`

description of stochastic-δN

member function

---

[1] Tomohiro Fujita, Masahiro Kawasaki, Yuichiro Tada, and Tomohiro Takesako, "A new algorithm for calculating the curvature perturbations in stochastic inflation," JCAP **1312**, 036 (2013), arXiv:1308.4754 [astro-ph.CO].

[2] Vincent Vennin and Alexei A. Starobinsky, "Correlation Functions in Stochastic Inflation," Eur. Phys. J. **C75**, 413 (2015), arXiv:1506.04732 [hep-th].

[3] David H. Lyth, Karim A. Malik, and Misao Sasaki, "A General proof of the conservation of the curvature perturbation," JCAP **0505**, 004 (2005), arXiv:astro-ph/0411220 [astro-ph].

[4] Chris Pattison, Vincent Vennin, Hooshyar Assadullahi, and David Wands, "Quantum diffusion during inflation and primordial black holes," JCAP **1710**, 046 (2017), arXiv:1707.00537 [hep-th].

[5] Sébastien Renaux-Petel, Yuichiro Tada, and Vincent Vennin, In preparation.