# FDPS C Interface Tutorial

Daisuke Namekata, Masaki Iwasawa, Keigo Nitadori, Ataru Tanikawa, Takayuki Muranushi, Long Wang, Natsuki Hosono, and Jun-ichiro Makino

Paricle Simulator Research Team, RIKEN Center for Computational Science, RIKEN

## 0 Contents

# 1 Change Log

- 2018/11/7
  - The initial version is created.
- 2019/7/19
  - Description of $N$-body/SPH sample code is updated (Sec. 7.1).

# 2 Overview

In this section, we present the overview of Framework for Developing Particle Simulator (FDPS) and FDPS C language interface . FDPS is an application-development framework which helps the application programmers and researchers to develop simulation codes for particle systems. What FDPS does are calculation of the particle-particle interactions and all of the necessary works to parallelize that part on distributed-memory parallel computers wit near-ideal load balancing, using hybrid parallel programming model (uses both MPI and OpenMP). Low-cost part of the simulation program, such as the integration of the orbits of particles using the calculated interaction, is taken care by the user-written part of the code.

FDPS support two- and three-dimensional Cartesian coordinates. Supported boundary conditions are open and periodic. For each coordinate, the user can select open or periodic boundary.

The user should specify the functional form of the particle-particle interaction. FDPS divides the interactions into two categories: long-range and short-range. The difference between two categories is that if the grouping of distant particles is used to speedup calculation (long-range) or not (short range).

The long-range force is further divided into two subcategories: with and without a cutoff scale. The long range force without cutoff is what is used for gravitational $N$-body simulations with open boundary. For periodic boundary, one would usually use TreePM, $P^3M$, PME or other variant, for which the long-range force with cutoff can be used.

The short-range force is divided to four subcategories. By definition, the short-range force has some cutoff length. If the cutoff length is a constant which does not depend on the identity of particles, the force belongs to "constant" class. If the cutoff depends on the source or receiver of the force, it is of "scatter" or "gather" classes. Finally, if the cutoff depends on both the source and receiver in the symmetric way, its class is "symmetric". Example of a "constant" interaction is the Lennard-Jones potential. Other interactions appear, for example, SPH calculation with adaptive kernel size.

The user writes the code for particle-particle interaction kernel and orbital integration using C language .

# 3 Getting Started

In this section, we describe the first steps you need to do to start using FDPS and FDPS C language interface . We explain the environment (the supported operating systems), the necessary software (compilers etc), and how to compile and run the sample codes.

## 3.1 Environment

FDPS works on Linux, Mac OS X, Windows (with Cygwin).

## 3.2 Necessary software

In this section, we describe software necessary to use FDPS, first for standard functions, and then for extensions.

### 3.2.1 Standard functions

we describe software necessary to use standard functions of FDPS. First for the case of single-thread execution, then for multithread, then for multi-nodes.

#### 3.2.1.1 Single thread

- make
- A C++ compiler (We have tested with gcc version 4.8.3 and K compiler version 1.2.0)
- A C compiler that are interoperable with the above C++ compiler (We have tested with gcc version 4.8.3).
- Python 2.7.5 or later, or, Python 3.4 or later (correct operation is not guaranteed for older Python versions)

#### 3.2.1.2 Parallel processing

*3.2.1.2.1 OpenMP*

- make
- A C++ compiler with OpenMP support (We have tested with gcc version 4.8.3 and K compiler version 1.2.0)
- A C compiler with OpenMP support (it must be interoperable with the above C++ compiler. We have tested with gcc version 4.8.3).
- Python 2.7.5 or later, or, Python 3.4 or later (correct operation is not guaranteed for older Python versions)

*3.2.1.2.2 MPI*

- make
- A C++ compiler which supports MPI version 1.3 or later. (We have tested with Open MPI 1.6.4 and K compiler version 1.2.0)
- A C compiler which supports MPI version 1.3 or later (it also must be interoperable with the above C++ compiler. We have tested with OpenMPI 1.6.4).
- Python 2.7.5 or later, or, Python 3.4 or later (correct operation is not guaranteed for older Python versions)

*3.2.1.2.3 MPI+OpenMP*

- make
- A C++ compiler which supports OpenMP and MPI version 1.3 or later. (We have tested with Open MPI 1.6.4 and K compiler version 1.2.0)
- A C compiler which supports OpenMP and MPI version 1.3 or later (it also must be interoperable with the above C++ compiler. We have tested with OpenMPI 1.6.4).
- Python 2.7.5 or later, or, Python 3.4 or later (correct operation is not guaranteed for older Python versions)

### 3.2.2 Extensions

Current extension for FDPS is the "Particle Mesh" module. We describe the necessary software for it below.

#### 3.2.2.1 Particle Mesh

- make
- A C++ compiler which supports OpenMP and MPI version 1.3 or later. (We have tested with Open MPI 1.6.4)
- FFTW 3.3 or later

## 3.3 Install

In this section we describe how to get the FDPS software and how to build it.

### 3.3.1 How to get the software

We first describe how to get the latest version, and then previous versions. We recommend to use the latest version.

#### 3.3.1.1 The latest version

You can use one of the following ways.

- Using browsers
  1. Click "Download ZIP" in https://github.com/FDPS/FDPS to download `FDPS-master.zip`
  2. Move the zip file to the directory under which you want to install FDPS and unzip the file (or place the files using some GUI).
- Using CLI (Command line interface)
  - Using Subversion:

```
$ svn co --depth empty https://github.com/FDPS/FDPS
$ cd FDPS
$ svn up trunk
```

  - Using Git

```
$ git clone git://github.com/FDPS/FDPS.git
```

#### 3.3.1.2 Previous versions

You can get previous versions using browsers.

- Previous versions are listed in https://github.com/FDPS/FDPS/releases. Click the version you want to download it.
- Extract the files under the directory you want.

### 3.3.2 How to install

Because FDPS is a header library[1)], you do not have to execute the `configure` command. All you need to do is to expand the archive of FDPS in some directory and to setup the include PATH when you compile your codes. An actual procedures can be found in Makefiles of the sample codes explained in § 3.4.

When using FDPS from C language, you first must create interface programs to FDPS based on user's codes. Its procedure is described in Chap. 6 of the specification document `doc_spec_ftn_en.pdf`. Makefiles of the sample codes are written so that the interface programs are automatically generated when `make` are running. We recommend that users use Makefiles of the sample codes as a reference when making your own Makefile.

[1)] A library that consists of header files only.

## 3.4 How to compile and run the sample codes

We provide two samples: one for gravitational $N$-body simulation and the other for SPH. We first describe gravitational $N$-body simulation and then SPH. Sample codes do not use extensions.

### 3.4.1 Gravitational $N$-body simulation

#### 3.4.1.1 Summary

Through the following steps one can use this sample.

- Move to the directory `$(FDPS)/sample/c/nbody`. Here, `$(FDPS)` denotes the highest-level directory for FDPS (Note that `FDPS` is not an environmental variable). The actual value of `$(FDPS)` depends on the way you acquire the software. If you used the browser, the last part is "FDPS-master". If you used Subversion or Git, it is "trunk" or "FDPS", respectively.
- Edit `Makefile` in the current directory (`$(FDPS)/sample/c/nbody`).
- Run the `make` command to create the executable `nbody.out`.
- Run `nbody.out`
- Check the output.

In addition, we describe the way to use Phantom-GRAPE for x86.

#### 3.4.1.2 Move to the directory with the sample code

Move to `$(FDPS)/sample/c/nbody`.

#### 3.4.1.3 Edit Makefile

In the directory, there is a Makefile for GCC, `Makefile`. In this section, we mainly describe this Makefile in detail.

First, we describe the default setting of `Makefile`. There are four Makefile variables that need to be set when compiling the sample code. They are the following. `CC` that stores the command to run a C compiler, `CXX` that stores the command to run a C++ compiler, and `CFLAGS` and `CXXFLAGS`, in which compiler options for both compilers are stored. The initial values of these variables are as follows:

```
CC=gcc
CXX=g++
CFLAGS = -O3 -ffast-math -funroll-loops -finline-functions
CXXFLAGS = -O3 -ffast-math -funroll-loops $(FDPS_INC)
```

where `$(FDPS_INC)` is the variable storing the include PATH for FDPS. It is already set in this Makefile and you do not need to modify it here.

An executable file can be obtained by executing the `make` command after setting the above four Makefile variables appropriately. Edit `Makefile` according the following descriptions. The changes depend on if you use OpenMP and/or MPI.

- Without both OpenMP and MPI
  - Set the variable `CC` the command to run your C compiler
  - Set the variable `CXX` the command to run your C++ compiler
- With OpenMP but not with MPI
  - Set the variable `CC` the command to run your C compiler with OpenMP support
  - Set the variable `CXX` the command to run your C++ compiler with OpenMP support
  - Uncomment the line `CFLAGS += -DPARTICLE_SIMULATOR_THREAD_PARALLEL -fopenmp`
  - Uncomment the line `CXXFLAGS += -DPARTICLE_SIMULATOR_THREAD_PARALLEL -fopenmp`
- With MPI but not with OpenMP
  - Set the variable `CC` the command to run your C compiler that supports MPI
  - Set the variable `CXX` the command to run your C++ compiler that supports MPI
  - Uncomment the line `CFLAGS += -DPARTICLE_SIMULATOR_MPI_PARALLEL`
  - Uncomment the line `CXXFLAGS += -DPARTICLE_SIMULATOR_MPI_PARALLEL`
- With both OpenMP and MPI
  - Set the variable `CC` the command to run your C compiler that supports both OpenMP and MPI
  - Set the variable `CXX` the command to run your C++ compiler that supports both OpenMP and MPI
  - Uncomment the line `CFLAGS += -DPARTICLE_SIMULATOR_THREAD_PARALLEL -fopenmp`
  - Uncomment the line `CFLAGS += -DPARTICLE_SIMULATOR_MPI_PARALLEL`
  - Uncomment the line `CXXFLAGS += -DPARTICLE_SIMULATOR_THREAD_PARALLEL -fopenmp`
  - Un comment the line `CXXFLAGS += -DPARTICLE_SIMULATOR_MPI_PARALLEL`

Next, we describe useful information when users use this Makefile to compile users' codes. Most important variables when using this Makefile are `FDPS_LOC`, `HDR_USER_DEFINED_TYPE`, and `SRC_USER`. The variable `FDPS_LOC` is used to store the PATH of the top directory of FDPS. Based on the value of `FDPS_LOC`, this Makefile automatically sets a lot of variables related to FDPS, such as the PATH of the directory storing FDPS source files and the PATH of the Python script to generate C interface. Thus, users should set appropriately. The variable `HDR_USER_DEFINED_TYPE` is used to store a list of names of C header files in which user-defined types are implemented, while the variable `SRC_USER` is used to store a

list of names of C files in which all the rest are implemented. The reason why we divide users' source files as above is to avoid needless recompilation of FDPS (as a result, we can reduce time required to compile and link users' codes): Because FDPS C interface programs are generated based on user-defined types, we need to recompile of FDPS only when files specified by `HDR_USER_DEFINED_TYPE` are modified. However, there is one thing users should be careful of. When there are dependencies between files specified by `HDR_USER_DEFINED_TYPE` or `SRC_USER`, users must describe these dependencies in Makefile. As for the way of describing dependencies in Makefile, please see the manual of GNU make, for example.

#### 3.4.1.4 Run make

Type "make" to run `make`. In the process of `make`, C language interface programs are first generated and then they are compiled together with the sample codes.

#### 3.4.1.5 Run the sample code

- If you are not using MPI, run the following in CLI (terminal)

```
$ ./nbody.out
```

- If you are using MPI, run the following in CLI (terminal)

```
$ MPIRUN -np NPROC ./nbody.out
```

  Here, `MPIRUN` should be `mpirun` or `mpiexec` depending on your MPI configuration, and `NPROC` is the number of processes you will use.

  Upon normal completion, the following output log should appear in stderr. The exact value of the energy error may depend on the system, but it is okay if its absolute value is of the order of $1 \times 10^{-3}$.

```
time:      7.000, energy error:  -3.8957312e-03
time:      8.000, energy error:  -3.7788873e-03
time:      9.000, energy error:  -3.7627744e-03
time:     10.000, energy error:  -3.7071071e-03
MemoryPool::finalize() is completed!
******** FDPS has successfully finished. ********
```

#### 3.4.1.6 Analysis of the result

In the directory `result`, files "snap0000x-proc0000y.dat" have been created. These files store the distribution of particles. Here, x is an integer indicating time and y is an integer indicating MPI process number (y is always 0 if the program is executed without MPI). The output file format is that in each line, index of particle, mass, position (x, y, z) and velocity (vx, vy, vz) are listed.

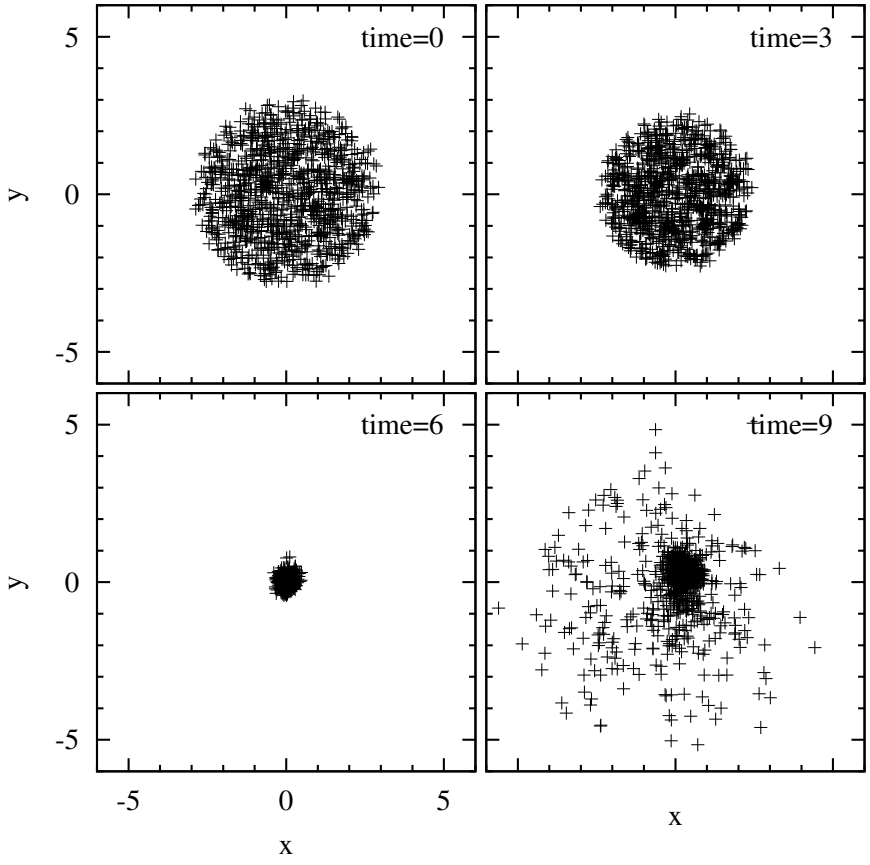


Figure 1:

What is simulated with the default sample is the cold collapse of an uniform sphere with radius three expressed using 1024 particles. Using gnuplot, you can see the particle distribution in the xy plane at time=9:

```
$ cd result
$ cat snap00009-proc* > snap00009.dat
$ gnuplot
> plot "snap00009.dat" using 3:4
```

By plotting the particle distributions at other times, you can see how the initially uniform sphere contracts and then expands again. (Figure 1).

To increate the number of particles to 10000, set the value of the parameter variable `ntot` (defined in the `void` function `c_main()` in the file `c_main.c`) to 10000, then recompile the sample codes, and run the executable file again.

#### 3.4.1.7 To use Phantom-GRAPE for x86

If you are using a computer with Intel or AMD x86 CPU, you can use Phantom-GRAPE for x86.

Move to the directory `$(FDPS)/src/phantom_grape_x86/G5/newton/libpg5`, edit the Makefile there (if necessary), and run make to build the Phantom-GRAPE library `libpg5.a`.

Then go back to directory `$(FDPS)/sample/c/nbody`, edit Makefile and remove "#" at the top of the line
"#use_phantom_grape_x86 = yes", and (after removing the existing executable) run make again. (Same for with and without OpenMP or MPI). You can run the executable in the same way as that for the executable without Phantom GRAPE.

The performance test on a machine with Intel Core i5-3210M CPU @2.50GHz (2 cores, 4 threads) indicates that, for N=8192, the code with Phantom GRAPE is faster than that without Phantom GRAPE by a factor a bit less than five.

#### 3.4.1.8 To use PIKG

PIKG (https://github.com/FDPS/PIKG) is a tool to generate a highly-optimized, two-body inter-particle interaction calculation kernel for particle simulations from a simple description of the interaction using a DSL (Domain Specific Language).

In order to use kernels generated by PIKG, open Makefile in directory `$(FDPS)/sample/c/nbody` and remove `#` at the top of the line `#use_pikg_x86 = yes`. Then, (after removing the existing executable) run `make`. (Same for with and without OpenMP or MPI). You can run the executable in the same way as that for the executable without PIKG.

In the default, PIKG generates kernels in `reference` mode. In this mode, unoptimized kernels are generated. To generate kernels optimized for specific architectures such as AVX2 and AVX-512, change `CONVERSION_TYPE` in Makefile and remove `#` at the top of the line containing `*FLAGS` (where `*` is the usual regular expression symbol).

### 3.4.2 SPH simulation code

#### 3.4.2.1 Summary

Through the following steps one can use this sample.

- Move to the directory `$(FDPS)/sample/c/sph`.
- Edit Makefile in the current directory (`$(FDPS)/sample/c/sph`).
- Run make command to create the executable `sph.out`.
- Run `sph.out`.
- Check the output.

#### 3.4.2.2 Move to the directory with the sample code

Move to `$(FDPS)/sample/c/sph`.

#### 3.4.2.3 Edit Makefile

Edit Makefile following the same description described in § 3.4.1.3.

#### 3.4.2.4 Run make

Type "make" to run `make`. As in $N$-body sample code, in the process of `make`, C language interface programs are first generated. Then, they are compiled together with SPH sample codes.

#### 3.4.2.5 Run the sample code

- If you are not using MPI, run the following in CLI (terminal)

```
$ ./sph.out
```

- If you are using MPI, run the following in CLI (terminal)

```
$ MPIRUN -np NPROC ./sph.out
```

Here, `MPIRUN` should be `mpirun` or `mpiexec` depending on your MPI configuration, and `NPROC` is the number of processes you will use.

Upon normal completion, the following output log should appear in stderr.

```
******** FDPS has successfully finished. ********
```

#### 3.4.2.6 Analysis of the result

In the directory `result`, files "snap0000x-proc0000y.dat" have been created. These files store the distribution of particles. Here, x and y are integers that indicate time and MPI process number, respectively. When executing the program without MPI, y is always 0. The output file format is that in each line, index of particle, mass, position (x, y, z), velocity (vx, vy, vz), density, internal energy and pressure are listed.

What is simulated is the three-dimensional shock-tube problem. Using gnuplot, you can see the plot of the x-coordinate and density of particles at time=40:

```
$ cd result
$ cat snap00040-proc* > snap00040.dat
$ gnuplot
> plot "snap00040.dat" using 3:9
```

When the sample worked correctly, a figure similar to Figure 2 should appear.

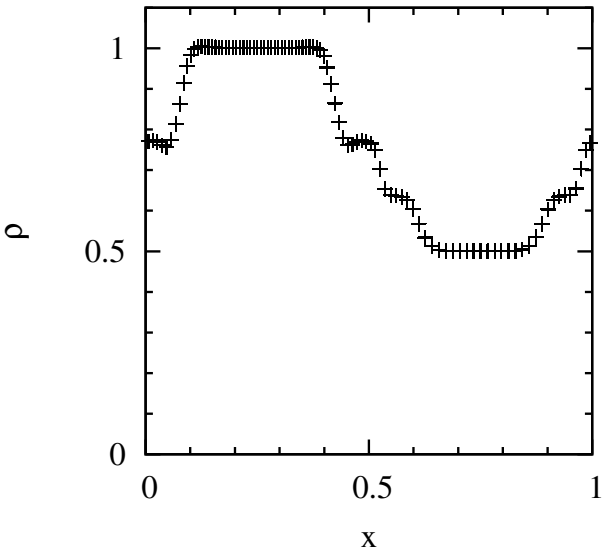


Figure 2:

# 4 How to use

In this section, we describe the sample codes used in previous section (§ 3) in more detail. Especially, the explanation will focus mainly on derived data types that users must define (hereafter, **user-defined types**) and how to use APIs of C language interface to FDPS. In order to avoid duplication of explanation, some matters are explained in § 4.1 only, where we explain the $N$-body sample code. Therefore, we recommend users who are interested in SPH simulation only to read § 4.1.

## 4.1 $N$-body simulation code

### 4.1.1 Location of source files and file structure

The source files of the sample code are in the directory `$(FDPS)/sample/c/nbody`. The sample code consists of `user_defined.h` where user-defined types are described, `user_defined.c` where interaction functions are described, and `c_main.c` where the other parts of $N$-body simulation code are implemented. In addition to these, there is a Makefile for GCC, `Makefile`.

### 4.1.2 User-defined types and user-defined functions

In this section, we describe the details of structures and `void` functions that users must define when performing an $N$-body simulation with FDPS.

#### 4.1.2.1 FullParticle type

You must define a FullParticle type. FullParticle type should contain all physical quantities necessary for an $N$-body simulation. Listing 1 shows the implementation of FullParticle type in our sample code (see `user_defined.h` ).

Listing 1: FullParticle type

```
1  typedef struct full_particle { //$fdps FP,EPI,EPJ,Force
2      //$fdps copyFromForce full_particle (pot,pot) (acc,acc)
3      //$fdps copyFromFP full_particle (id,id) (mass,mass) (eps,eps) (pos,
           pos)
4      //$fdps clear id=keep, mass=keep, eps=keep, pos=keep, vel=keep
5      long long id; // $fdps id
6      double  mass; //$fdps charge
7      double  eps;
8      fdps_f64vec pos; //$fdps position
9      fdps_f64vec vel; //$fdps velocity
10     double  pot;
11     fdps_f64vec acc;
12 } Full_particle;
```

When developing a simulation code with FDPS C language interface, users must specify which user-defined type (FullParticle, EssentialParticleI, EssentialParticleJ, and Force types) a structure corresponds to. In FDPS C language interface, this is done by adding a **FDPS directive**, which is a C language's comment text with a special format, to a structure.

Because FullParticle type is used as EssentialParticleI type, EssentialParticleJ type, and Force type in this sample code, a FDPS directive specifying that the structure acts as any types of user-defined types is described:

```
typedef struct full_particle { //$fdps FP,EPI,EPJ,Force
```

FDPS must know which member variable of FullParticle type corresponds to which necessary quantity, where **necessary quantities** are defined as the quantities that are necessary in any types of particle simulations (e.g. mass (or charge) and position of a particle), or that are necessary in particular types of particle simulations (e.g. size of a particle). This designation is also done by adding a comment text with a special format to each member variable. In this sample code, in order to specify that member variables `mass`, `pos`, `vel` correspond to mass, position, velocity of a particle, the following directives are described:

```
double mass; //$fdps charge
fdps_f64vec pos; //$fdps position
fdps_f64vec vel; //$fdps velocity
```

Note that `velocity` in the directive `//$fdps velocity` is a just reserved keyword and it does not alter the operation of FDPS at the present moment (hence, the designation is arbitrary).

FDPS copies data from FullParticle type toEssentialParticleI type and EssentialParticleJ type, or from Force type to FullParticle type. Users must describe FDPS directives that specify how to copy data. In this sample code, the following directives are described:

```
//$fdps copyFromForce full_particle (pot,pot) (acc,acc)
//$fdps copyFromFP full_particle (id,id) (mass,mass) (pos,pos)
```

where the FDPS directive with the keyword `copyFromForce` specifies which member variable of Force type is copied to which member variable of FullParticle type. Users **always have to** describe this directive in FullParticle type. The other directive with the keyword `copyFromFP` specifies how to copy data from FullParticle type to EssentialParticleI type and EssentialParticleJ type. This directive **must always** be described in EssentialParticleI type and EssentialParticleJ type. It is described here because FullParticle type in this sample code acts as EssentialParticleI type and EssentialParticleJ type.

FullParticle type also acts as Force type in this code. There is a FDPS directive that users must describe in Force type. It is the directive that specifies how to reset or initialize member variables of Force type before the calculation of interactions. In this code, the following directive is described to direct FDPS to zero-clear member variables corresponding to acceleration and potential only.

```
//$fdps clear id=keep, mass=keep, pos=keep, vel=keep
```

where the syntax *`mbr`*`=keep` to the right of the keyword `clear` is the syntax to direct FDPS not to change the value of member variable *`mbr`*.

Further details about the format of FDPS directive can be found in the specification document of FDPS Fortran/C interface, `doc_specs_ftn_en.pdf`.

#### 4.1.2.2 calcForceEpEp

You must define an interaction function calcForceEpEp as `void` function in C language. It should contain actual code for the calculation of interaction between particles. Listing 2 shows the implementation of calcForceEpEp (see `user_defined.h` ).

Listing 2: Function calcForceEpEp

```
1  void calc_gravity_ep_ep(Full_particle *ep_i,
2                          int n_ip,
3                          Full_particle *ep_j,
4                          int n_jp,
5                          Full_particle* f)
6  {
7      int i, j;
8      for (i=0; i<n_ip ;i++) {
9          Full_particle *pi = ep_i + i;
10         double eps2 = pi->eps * pi->eps;
11         double xi = pi->pos.x;
12         double yi = pi->pos.y;
13         double zi = pi->pos.z;
14         double ax, ay, az, pot;
15         ax = ay = az = pot = 0;
16         for (j=0; j<n_jp; j++) {
17             Full_particle *pj = ep_j + j;
18             double dx = xi - pj->pos.x;
19             double dy = yi - pj->pos.y;
20             double dz = zi - pj->pos.z;
21             double r2 = dx*dx+dy*dy+dz*dz+eps2;
22             double rinv = 1.0/sqrt(r2);
23             double mrinv = pj->mass* rinv;
24             double mr3inv = mrinv*rinv*rinv;
25             ax -= dx*mr3inv;
26             ay -= dy*mr3inv;
27             az -= dz*mr3inv;
28             pot = pot - mrinv;
29         }
30         Full_particle *pfi = f+i;
31         pfi->pot += pot;
32         pfi->acc.x += ax;
33         pfi->acc.y += ay;
34         pfi->acc.z += az;
35     }
36 }
```

In this sample code, it is implemented as the `void` function `calc_gravity_ep_ep`. Its dummy arguments are an array of EssentialParticleI type, the number of EssentialParticleI type variables, an array of EssentialParticleJ type, the number of EssentialParticleJ type variables, an array of Force type. Note that all the data types of the dummy arguments corresponding to user-defined types are `full_particle` type because FullParticle type acts as the other types of user-defined types in this sample code.

#### 4.1.2.3 calcForceEpSp

You must defined an interaction function calcForceEpSp as `void` function in C language. It should contain actual code for the calculation of interaction between a particle and a superparticle. Listing 3 shows the implementation of calcForceEpSp (see `user_defined.F90`).

Listing 3: calcForceEpSp

```
1  void calc_gravity_ep_sp(Full_particle *ep_i,
2                          int n_ip,
3                          fdps_spj_monopole *ep_j,
4                          int n_jp,
5                          Full_particle *f)
6  {
7      int i, j;
8      for (i=0; i<n_ip; i++) {
9          Full_particle *pi = ep_i + i;
10         double eps2 = pi->eps*pi->eps;
11         double xi = pi->pos.x;
12         double yi = pi->pos.y;
13         double zi = pi->pos.z;
14         double ax, ay, az, pot;
15         ax = ay = az = pot = 0;
16         for (j=0; j<n_jp; j++) {
17             fdps_spj_monopole *pj = ep_j + j;
18             double dx = xi - pj->pos.x;
19             double dy = yi - pj->pos.y;
20             double dz = zi - pj->pos.z;
21             double r2 = dx*dx+dy*dy+dz*dz+eps2;
22             double rinv = 1.0/sqrt(r2);
23             double mrinv = pj->mass* rinv;
24             double mr3inv = mrinv*rinv*rinv;
25             ax -= dx*mr3inv;
26             ay -= dy*mr3inv;
27             az -= dz*mr3inv;
28             pot = pot - mrinv;
29         }
30         Full_particle *pfi = f+i;
31         pfi->pot += pot;
32         pfi->acc.x += ax;
33         pfi->acc.y += ay;
34         pfi->acc.z += az;
35     }
36 }
```

In this sample code, it is implemented as the `void` function `calc_gravity_ep_sp`. Its dummy arguments are an array of EssentialParticleI type, the number of EssentialParticleI type variables, an array of superparticle type, the number of superparticle type variables, an array of Force type. Note that the data types of EssentialParticleI type and Force type are `full_particle` type because FullParticle type acts as these user-defined types in this sample code. Also note that the data type of superparticle type must be consistent with the type of a Tree object used in the calculation of interactions.

### 4.1.3 The main body of the user program

In this section, we describe the functions a user should write in a kind of main function, `c_main()`, to implement gravitational $N$-body calculation using the FDPS C language interface. The reason why we do not use the term main function clearly is as follows: If users use FDPS C language interface, the user code must be written in the `void` function `c_main()`. Thus the user code dose not include the main function . However, in practice, the `c_main()` plays the same role as a main function. Thus here we use the term a kind of main function. The term main function is suitable for indicating the top level function of the user code. Hereafter, we call `c_main()` the main function. The main function of this sample is written in `c_main.c`.

#### 4.1.3.1 Including the header file of FDPS C interface

To make the standard features of FDPS available, we must include header file `FDPS_c_if.h`.

Listing 4: Including header file `FDPS_c_if.h`

```
1 #include "FDPS_c_if.h"
```

#### 4.1.3.2 Initialization and Termination of FDPS

First, users must initialize FDPS by the following code.

Listing 5: Initialization of FDPS

```
1 fdps_initialize();
```

Once started, FDPS should be terminated explicitly. In the sample code, FDPS should be terminated just before the termination of the program. To achieve this, user should write the following code at the end of the main function.

Listing 6: Termination of FDPS

```
1 fdps_finalize();
```

#### 4.1.3.3 Creation and initialization of FDPS objects

Once succeed the initialization, the user needs to create objects used to talk to FDPS. In this section, we describe how to create and initialize these objects.

##### *4.1.3.3.1 Creation of FDPS objects*

In an $N$-body simulation, one needs to create objects of ParticleSystem type, DomainInfo type, and Tree type. In the C language interface, these objects can be handled by using identification number contained in integral type variables. Thus, at the beginning, you should prepare integral type variables to contain the identification numbers. We will show an example bellow. These are written in the main function `c_main.c` in the sample code.

Listing 7: Creation of FDPS objects

```
1  void c_main() {
2
3      // Create and initialize dinfo object
4      int dinfo_num;
5      fdps_create_dinfo(&dinfo_num);
6      // Create and initialize psys object
7      int psys_num;
8      fdps_create_psys(&psys_num,"full_particle");
9      // Create and initialize tree object
10     int tree_num;
11     fdps_create_tree(&tree_num,
12                      "Long,full_particle,full_particle,full_particle,
                               Monopole");
13
14 }
```

Here, the code shown is just a corresponding part of the sample code. As we can see above, to create the object of type ParticleSystem, you must give the string of the name of the derived data type corresponding to the type FullParticle. As in the case of type ParticleSystem, to create the object of type Tree, you must give the string which indicates the type of tree as an argument of the API. Note that, in both APIs, the name of the derived data type must be written in lower case.

#### *4.1.3.3.2 Initialization of DomainInfo object*

Once create the objects, user must initialize these objects. In this sample code, since the boundary condition is not periodic, users have only to call the API `fdps_init_dinfo` to initialize the objects.

Listing 8: Initialization of DomainInfo object

```
1  fdps_init_dinfo(dinfo_num,coef_ema);
```

Note that the second argument of API `fdps_init_dinfo` is a smoothing factor of an exponential moving average operation that is performed in the domain decomposition procedure. The definition of this factor is described in the specification of FDPS (see § 9.1.2 in `doc_spec_cpp_en.pdf`).

#### *4.1.3.3.3 Initialization of ParticleSystem object*

Next, you must initialize a ParticleSystem object. This is done by calling the API `fdps_init_psys` .

Listing 9: Initialization of ParticleSystem object

```
1  fdps_init_psys(psys_num);
```

#### *4.1.3.3.4 Initialization of Tree object*

Next, we must initialize a Tree object. The initialization of a Tree object is done by calling the API `fdps_init_tree` . This API should be given a rough number of particles.

In this sample, we set the total number of particles `ntot`:

Listing 10: Initialization of Tree object

```
fdps_init_tree(tree_num,ntot,theta,
               n_leaf_limit,n_group_limit);
```

The definitions of the 3rd or later arguments are as follows.

- `theta` — the so-called opening angle criterion for the tree method.
- `n_leaf_limit` — the upper limit for the number of particles in the leaf nodes.
- `n_group_limit` — the upper limit for the number of particles with which the particles use the same interaction list for the force calculation.

#### 4.1.3.4 Initialization of particle data

To initialize particle data, users must give the particle data to the ParticleSystem object. This can be done by using APIs `fdps_set_nptcl_loc` and `get_psys_cptr` as follows:

Listing 11: Initialization of particle data

```
void foo(psys_num) {
   // Set # of local particles
   nptcl_loc = 1024;
   fdps_set_nptcl_loc(psys_num,nptcl_loc);

   // Get the pointer to full particle data
   Full_particle *ptcl = (Full_particle *)  fdps_get_psys_cptr(psys_num);

   // Initialize particle data
   int i;
   for (i = 0; i < nptcl_loc; i++) {
      ptcl[i].pos.x = /* Do something */ ;
   }

}
```

First, you must allocate the memory to store the particle data. To do so, you have only to call API `fdps_set_nptcl_loc` . This API sets the number of local particles (the number of particles assigned to the local process) and allocates enough memory to store the particles. To initialize particle data, the beginning address of the allocated memory is needed. Users can obtain the beginning address by using the API `get_psys_cptr` . Note that we need to cast the returned value to `Full_particle *` type because API `fdps_get_psys_cptr` returns the address of `void *` type. After setting the pointer `ptcl`, we can use it as array.

#### 4.1.3.5 Time integration loop

In this section we describe the structure of the time integration loop.

##### *4.1.3.5.1 Domain Decomposition*

First, the computational domain is decomposed, using the current distribution of particles. In the sample, this is done by API `fdps_decompose_domain_all` of the DomainInfo object:

Listing 12: Domain Decomposition

```
1 if (num_loop % 4 == 0) {
2     fdps_decompose_domain_all(dinfo_num,psys_num);
3 }
```

In this sample code, we perform domain decomposition once in 4 main loops in order to reduce the computational cost.

#### *4.1.3.5.2 Particle Exchange*

Then, particles are exchanged between processes so that they belong to the process for the domain of their coordinates. To do so, users can use API `fdps_exchange_particle` of ParticleSystem object.

Listing 13: Particle Exchange

```
1 fdps_exchange_particle(psys_num,dinfo_num);
```

#### *4.1.3.5.3 Interaction Calculation*

After the domain decomposition and the particle exchange, an interaction calculation is done. To do so, users can use API `calc_force_all_and_write_back` of Tree object.

Listing 14: Interaction Calculation

```
1  void c_main() {
2
3      // Do somehting
4
5      fdps_calc_force_all_and_write_back(tree_num,
6                                         calc_gravity_ep_ep,
7                                         calc_gravity_ep_sp,
8                                         psys_num,
9                                         dinfo_num,
10                                        true,
11                                        FDPS_MAKE_LIST);
12
13     // Do something
14
15 }
```

Here, the second and third arguments are the function pointers of calcForceEpEp and calcForceEpSp. The sixth argument is a flag to specify whether or not to clear the result of previous interaction calculation. The seventh argument is a flag to specify the re-using feature of interaction lists is used or not. Passing `FDPS_MAKE_LIST` makes FDPS create a new interaction lists and perform interaction calculation using these lists.

#### *4.1.3.5.4 Time integration*

In this sample code, we use the Leapfrog method to integrate the particle system in time. In this method, the time evolution operator can be expressed as $K(\frac{\Delta t}{2})D(\Delta t)K(\frac{\Delta t}{2})$, where

$\Delta t$ is the timestep, $K(\Delta t)$ is the 'kick' operator that integrates the velocities of particles from $t$ to $t + \Delta t$, $D(\Delta t)$ is the 'drift' operator that integrates the positions of particles from $t$ to $t + \Delta t$ (e.g. see Springel [2005,MNRAS,364,1105]). In the sample code, these operators are implemented as the `void` functions `kick` and `drift`.

At the beginning of the main loop, the positions and the velocities of the particles are updated by the operator $D(\Delta t)K(\frac{\Delta t}{2})$:

After the force calculation, the velocities of the particles are updated by the operator $K(\frac{\Delta t}{2})$:

Listing 15: Calculation of $K(\frac{\Delta t}{2})$ operator

```
1 // Leapfrog: Kick
2 kick(psys_num,0.5d0*dt);
```

#### 4.1.3.6 Update of particle data

To update the data of particles in the subroutines such as `kick` or `drift`, you need to access the data of particles contained in the object of type ParticleSystem. To do so, the user can follow the same way described in section 4.1.3.4.

Listing 16: Update of particle data

```
1  void foo(psys_num) {
2     // Get # of local particles
3     int nptcl_loc = fdps_get_nptcl_loc(psys_num);
4
5     // Get the pointer to full particle data
6     Full_particle *ptcl = (Full_particle *) fdps_get_psys_cptr(psys_num);
7
8     // Initialize or update particle data
9     int i;
10    for (i = 0; i < nptcl_loc; i++) {
11       ptcl[i].pos.x = /* Do something */ ;
12    }
13 }
```

Using API `fdps_get_psys_cptr`, you can obtain the address of particle data contained in the object of ParticleSystem as a pointer. The pointer obtained here can be regarded as an array of size `nptcl_loc`. Thus, user can update the particle data as array.

### 4.1.4 Log file

Once the calculation starts successfully, the time and the energy error are printed in the standard output. The first step is shown in the bellow example.

Listing 17: standard output

```
1 time:    0.0000000000E+000, energy error:   -0.0000000000E+000
```

## 4.2 SPH simulation code with fixed smoothing length

In this section, we describe the sample code used in the previous section (§ 3), a standard SPH code with fixed smoothing length, in detail.

### 4.2.1 Location of source files and file structure

The source files of the sample code are in the directory `$(FDPS)/sample/c/sph`. The sample code consists of `user_defined.h` where user-defined types are described, `user_defined.c` where interaction functions are described, and `c_main.c` where the other parts of the SPH simulation code are described. In addition, there is a Makefile for GCC, `Makefile`.

### 4.2.2 User-defined types and user-defined functions

In this section, we describe the derived data types and subroutines that users must define when performing SPH simulations by using of FDPS.

#### 4.2.2.1 FullParticle type

Users must define a FullParticle type as a user-defined type. The FullParticle type must contain all physical quantities of an SPH particle necessary for the simulation. Listing 18 shows an example implementation of the `FullParticle` type in our sample code (see `user_defined.h` ).

Listing 18: FullParticle type

```
1  typedef struct full_particle { //$fdps FP
2      //$fdps copyFromForce force_dens (dens,dens)
3      //$fdps copyFromForce force_hydro (acc,acc) (eng_dot,eng_dot) (dt,dt)
4      double mass; //$fdps charge
5      fdps_f64vec pos; //$fdps position
6      fdps_f64vec vel;
7      fdps_f64vec acc;
8      double dens;
9      double eng;
10     double pres;
11     double smth; //$fdps rsearch
12     double snds;
13     double eng_dot;
14     double dt;
15     long long int id;
16     fdps_f64vec vel_half;
17     double eng_half;
18 } Full_particle;
```

Unlike the case of the $N$-body simulation sample code, the FullParticle type of the SPH simulation sample code does not double as other user-defined types. Thus, to specify that this structure is a FullParticle type, we append the following directive.

```
typedef struct full_particle { //$fdps FP
```

In the SPH simulations, the interaction force is short-range force. Therefore, a search radius is also necessary physical quantity in addition to the position and mass (charge). We can tell FDPS which member variables represent these necessary quantities in the following way:

```
double mass; //$fdps charge
fdps_f64vec pos; //$fdps position
double smth; //$fdps rsearch
```

As described in the section of the $N$-boy simulation code, the keyword `velocity` to specify that a member corresponds to the velocity of a particle is mere a reserved word and not always necessary, we do not specify that in this sample code.

The FullParticle type copies data from a Force type. Users must specify how the data is copied by using of directives. As we will describe later, there are 2 Force types in this SPH sample code. Thus, for each Force type, users must write the directives. In this sample code, these are:

```
//$fdps copyFromForce force_dens (dens,dens)
//$fdps copyFromForce force_hydro (acc,acc) (eng_dot,eng_dot) (dt,dt)
```

#### 4.2.2.2 EssentialParticleI(J) type

Users must define an EssentialParticleI type. An EssentialParticleI type must contain all necessary physical quantities to compute the Force as an $i$-particle in its member variables. Moreover in this sample code, it also doubles as an EssentialParticleJ type and all necessary physical quantities as a $j$-particle as well need to be included in the member variables. Hereinafter, we simply call this EssentialParticle type. Listing 19 shows an example of EssentialParticle type of this sample code (see `user_defined.h` ):

Listing 19: EssentialParticle type

```
1  typedef struct essential_particle { //$fdps EPI,EPJ
2      //$fdps copyFromFP full_particle (id,id) (pos,pos) (vel,vel) (mass,
           mass) (smth,smth) (dens,dens) (pres,pres) (snds,snds)
3      long long int id;
4      fdps_f64vec pos; //$fdps position
5      fdps_f64vec vel;
6      double mass; //$fdps charge
7      double smth; //$fdps rsearch
8      double dens;
9      double pres;
10     double snds;
11 } Essential_particle;
```

First, users must indicate to FDPS that this structure corresponds to both the EssentialParticleI type and EssentialParticleJ type by using the directives. This sample code describes that as follows:

```
typedef struct essential_particle { //$fdps EPI,EPJ
```

Next, users must indicate the correspondence between the each of member variable in this structure and necessary physical quantity. For this SPH simulation, a search radius needs to be indicated as well. This sample code describes them as follows:

```
fdps_f64vec pos; //$fdps position
double mass; //$fdps charge
double smth; //$fdps rsearch
```

The EssentialParticleI and EssentialParticleJ types receive data from the FullParticle type. Users must specify the source member variables in the FullParticle type and the destination member variable in the EssentialParticle? type (?=I,J) that will be copied through the directives. This sample code describes them as follows:

```
//$fdps copyFromFP full_particle (id,id) (pos,pos) (vel,vel) (mass,mass)
(smth,smth) (dens,dens) (pres,pres) (snds,snds)
```

#### 4.2.2.3 Force type

Users must define a Force type. A Force type must contain all the resultant physical quantities after performing the Force computations. In this sample code, we have 2 force computations; one for the density and the other for the fluid interactions. Thus, we have to define 2 different Force types. In Listing 20, we show an example of the Force types in this sample code.

Listing 20: Force type

```
1  typedef struct force_dens { //$fdps Force
2      //$fdps clear smth=keep
3      double dens;
4      double smth;
5  } Force_dens;
6
7  typedef struct force_hydro { //$fdps Force
8      //$fdps clear
9      fdps_f64vec acc;
10     double eng_dot;
11     double dt;
12 } Force_hydro;
```

First, users must indicate with directives that these structures correspond to the Force types. In this example, these writes:

```
typedef struct force_dens { //$fdps Force
typedef struct force_hydro { //$fdps Force
```

For these structures to be Force types, users **must** indicate the initialization methods for the member variables that are accumulated during the interaction calculations. In this sample code, we indicate that only the accumulator variables — density, acceleration (due to pressure gradient), time-derivative of energy, and time step to be zero-cleared.

```
//$fdps clear smth=keep
//$fdps clear
```

In this example the Force type `force_dens` includes a member variable `smth` that indicates the smoothing length. For a fixed length SPH, a member variable for the smoothing length in the Force type has nothing to do. We prepare this member variable for the future extension to the variable length SPH for some users. In one of the formulations of the variable length SPH in Springel [2005,MNRAS,364,1105], we need to calculate the smoothing length at the same time we calculate the density. To implement a formulation like that, a Force type need to contain a variable for the smoothing length as in this example. In this sample code for fixed length SPH, the member function `clear` will not zero-clear the variable `smth`, so as not to crush the next computation of the density.

#### 4.2.2.4 calcForceEpEp

Users must define a `void` function calcForceEpEp in C language which specifies the interaction between particles. It should contain actual code for the calculation of interaction between particles. Listing 21 shows the implementation of calcForceEpEp (see ) .

Listing 21: Function calcForceEpEp

```
1  void calc_density(Essential_particle *ep_i,
2                    int n_ip,
3                    Essential_particle *ep_j,
4                    int n_jp,
5                    Force_dens *f) {
6      int i,j;
7      fdps_f64vec dr;
8      for (i = 0; i< n_ip; i++) {
9          for (j = 0; j < n_jp; j++) {
10             dr.x = ep_j[j].pos.x - ep_i[i].pos.x;
11             dr.y = ep_j[j].pos.y - ep_i[i].pos.y;
12             dr.z = ep_j[j].pos.z - ep_i[i].pos.z;
13             f[i].dens += ep_j[j].mass * W(dr,ep_i[i].smth);
14         }
15     }
16
17 }
18
19 void calc_hydro_force(Essential_particle *ep_i,
20                       int n_ip,
21                       Essential_particle *ep_j,
22                       int n_jp,
23                       Force_hydro *f) {
24     // Local parameters
25     const double C_CFL=0.3;
26     // Local variables
27     int i,j;
28     double mass_i,mass_j,smth_i,smth_j,
29            dens_i,dens_j,pres_i,pres_j,
30            snds_i,snds_j;
31     double povrho2_i,povrho2_j,
32            v_sig_max,dr_dv,w_ij,v_sig,AV;
33     fdps_f64vec pos_i,pos_j,vel_i,vel_j,
34                 dr,dv,gradW_i,gradW_j,gradW_ij;
35
```

```
36      for (i = 0; i < n_ip; i++) {
37          // Zero-clear
38          v_sig_max = 0.0;
39          // Extract i-particle info.
40          pos_i.x = ep_i[i].pos.x;
41          pos_i.y = ep_i[i].pos.y;
42          pos_i.z = ep_i[i].pos.z;
43          vel_i.x = ep_i[i].vel.x;
44          vel_i.y = ep_i[i].vel.y;
45          vel_i.z = ep_i[i].vel.z;
46          mass_i  = ep_i[i].mass;
47          smth_i  = ep_i[i].smth;
48          dens_i  = ep_i[i].dens;
49          pres_i  = ep_i[i].pres;
50          snds_i  = ep_i[i].snds;
51          povrho2_i = pres_i/(dens_i*dens_i);
52          for (j = 0; j < n_jp; j++) {
53              // Extract j-particle info.
54              pos_j.x = ep_j[j].pos.x;
55              pos_j.y = ep_j[j].pos.y;
56              pos_j.z = ep_j[j].pos.z;
57              vel_j.x = ep_j[j].vel.x;
58              vel_j.y = ep_j[j].vel.y;
59              vel_j.z = ep_j[j].vel.z;
60              mass_j  = ep_j[j].mass;
61              smth_j  = ep_j[j].smth;
62              dens_j  = ep_j[j].dens;
63              pres_j  = ep_j[j].pres;
64              snds_j  = ep_j[j].snds;
65              povrho2_j = pres_j/(dens_j*dens_j);
66              // Compute dr & dv
67              dr.x = pos_i.x - pos_j.x;
68              dr.y = pos_i.y - pos_j.y;
69              dr.z = pos_i.z - pos_j.z;
70              dv.x = vel_i.x - vel_j.x;
71              dv.y = vel_i.y - vel_j.y;
72              dv.z = vel_i.z - vel_j.z;
73              // Compute the signal velocity
74              dr_dv = dr.x * dv.x + dr.y * dv.y + dr.z * dv.z;
75              if (dr_dv < 0.0) {
76                 w_ij = dr_dv / sqrt(dr.x * dr.x + dr.y * dr.y + dr.z * dr.z
                          );
77              } else {
78                 w_ij = 0.0;
79              }
80              v_sig = snds_i + snds_j - 3.0 * w_ij;
81              if (v_sig > v_sig_max) v_sig_max=v_sig;
82              // Compute the artificial viscosity
83              AV = - 0.5*v_sig*w_ij / (0.5*(dens_i+dens_j));
84              // Compute the average of the gradients of kernel
85              gradW_i  = gradW(dr,smth_i);
86              gradW_j  = gradW(dr,smth_j);
87              gradW_ij.x = 0.5 * (gradW_i.x + gradW_j.x);
88              gradW_ij.y = 0.5 * (gradW_i.y + gradW_j.y);
89              gradW_ij.z = 0.5 * (gradW_i.z + gradW_j.z);
```

```
90              // Compute the acceleration and the heating rate
91              f[i].acc.x -= mass_j*(povrho2_i+povrho2_j+AV)*gradW_ij.x;
92              f[i].acc.y -= mass_j*(povrho2_i+povrho2_j+AV)*gradW_ij.y;
93              f[i].acc.z -= mass_j*(povrho2_i+povrho2_j+AV)*gradW_ij.z;
94              f[i].eng_dot += mass_j * (povrho2_i + 0.5*AV)
95                              *(dv.x * gradW_ij.x
96                               +dv.y * gradW_ij.y
97                               +dv.z * gradW_ij.z);
98          }
99          f[i].dt = C_CFL*2.0*smth_i/(v_sig_max*kernel_support_radius);
100     }
101 }
```

This SPH simulation code includes two different forms of interactions, and hence, two different implementations of calcForceEpEp are needed. In either case, the dummy arguments of `void` function are, an array of EssentialParticleI, the number of EssentialParticleI, an array of EssentialParticleJ, the number of EssentialParticleJ, and an array of Force.

### 4.2.3 The main body of the user program

In this section, we describe `void` functions and functions to be called from the main function of the user program when a user want to do an SPH simulation using FDPS (for the meaning of "main function" see section 4.1.3) .

#### 4.2.3.1 Including the header file of FDPS C interface

To make the standard features of FDPS available, we must include header file `FDPS_c_if.h`.

Listing 22: Including header file `FDPS_c_if.h`

```
1 #include "FDPS_c_if.h"
```

#### 4.2.3.2 Initialization and termination of FDPS

You should first initialize FDPS by the following code.

Listing 23: Initialization of FDPS

```
1 fdps_initialize();
```

Once started, FDPS should be explicitly terminated. In this sample, FDPS is terminated just before the termination of the program. To achieve this, you write the following code at the end of the main function.

Listing 24: Termination of FDPS

```
1 fdps_finalize();
```

#### 4.2.3.3 Creation and initialization of FDPS objects

After the initialization of FDPS, a user need to create the objects used to talk to FDPS. In this section we describe how to create and initialize these objects.

#### *4.2.3.3.1 Creation of necessary FDPS objects*

In an SPH simulation code, one needs to create objects for particles, for domain information, for interaction calculation of Gather type (for density calculation using gather type interaction), and for interaction calculation of Symmetry type (for hydrodynamic interaction calculation using symmetric type interaction). The following is the code to create to them.

Listing 25: Creation of necessary FDPS objects

```
1  void c_main() {
2     // Create ParticleSystem object
3     int psys_num;
4     fdps_create_psys(&psys_num,"full_particle");
5
6     // Create DomainInfo object
7     int dinfo_num;
8     fdps_create_dinfo(&dinfo_num);
9
10    // Create Tree objects
11    int tree_num_dens;
12    fdps_create_tree(&tree_num_dens,
13                     "Short,dens_force,essential_particle,
                               essential_particle,Gather");
14    int tree_num_hydro;
15    fdps_create_tree(&tree_num_hydro,
16                     "Short,hydro_force,essential_particle,
                               essential_particle,Symmetry");
17
18 }
```

Note that here again this code snippet only shows the necessary part of the code from the actual sample code.

API `fdps_create_psys` and `fdps_create_tree` should receive strings indicating particle type and tree type, respectively. **All of names of structures in these strings should be in lowercases.**

#### *4.2.3.3.2 Initialization of the domain information object*

FDPS objects created by a user code should be initialized. Here, we describe the necessary procedures required to initialize a DomainInfo object. First, we need to call API `fdps_init_dinfo` of DomainInfo object. After the initialization of the object, the type of the boundary and the size of the simulation box should be set by calling APIs `fdps_set_boundary_condition` and `fdps_set_pos_root_domain` of DomainInfo object. In this code, we use the periodic boundary for all of $x$, $y$ and $z$ directions.

Listing 26: Initialization of DomainInfo object

```
1 fdps_init_dinfo(dinfo_num,coef_ema);
2 fdps_set_boundary_condition(dinfo_num,FDPS_BC_PERIODIC_XYZ);
3 fdps_set_pos_root_domain(dinfo_num,&pos_ll,&pos_ul);
```

*4.2.3.3.3 Initialization of ParticleSystem object*

Next, we need to initialize the ParticleSystem object. This is done by the following single line of code:

Listing 27: Initialization of ParticleSystem object

```
1 fdps_init_psys(psys_num);
```

*4.2.3.3.4 Initialization of Tree objects*

Finally, Tree objects should be initialized. This is done by calling API `fdps_init_tree` of Tree object. This API should be given the rough number of particles. In this sample, we set three times the total number of particles:

Listing 28: Initialization of tree objects

```
1 fdps_init_tree(dens_tree_num,3*ntot,theta,
2                n_leaf_limit,n_group_limit);
3 fdps_init_tree(hydro_tree_num,3*ntot,theta,
4                n_leaf_limit,n_group_limit);
```

#### 4.2.3.4 Time integration loop

In this section we describe the structure of the time integration loop.

*4.2.3.4.1 Domain Decomposition*

First, the computational domain is decomposed, using the current distribution of particles. To do so, the API `fdps_decompose_domain_all` of DomainInfo object is called.

Listing 29: Domain Decomposition

```
1 fdps_decompose_domain_all(dinfo_num,psys_num);
```

*4.2.3.4.2 Particle Exchange*

Then particles are exchanged between processes so that they belong to the process for the domain of their coordinates. To do so, the following API `fdps_exchange_particle` of `ParticleSystem` object is used.

Listing 30: Particle Exchange

```
1 fdps_exchange_particle(psys_num,dinfo_num);
```

*4.2.3.4.3 Interaction Calculation*

After the domain decomposition and particle exchange, interaction calculation is done. To do so, the following API `fdps_calc_force_all_and_write_back` of Tree object is used.

Listing 31: Interaction Calculation

```
1  void c_main() {
2
3     // Do something
4
5     fdps_calc_force_all_and_write_back(tree_num_dens,
6                                        calc_density,
7                                        NULL,
8                                        psys_num,
9                                        dinfo_num,
10                                       true,
11                                       FDPS_MAKE_LIST);
12    set_pressure(psys_num);
13    fdps_calc_force_all_and_write_back(tree_num_hydro,
14                                       calc_hydro_force,
15                                       NULL,
16                                       psys_num,
17                                       dinfo_num,
18                                       true,
19                                       FDPS_MAKE_LIST);
20
21    // Do something
22
23 }
```

For the second argument of API, the function pointer calcForceEpEp should be given. The sixth argument determines whether the result of the previous interaction calculation is cleared before performing interaction calculation. The seventh argument determines whether the interaction lists are re-used or not. Passing `FDPS_MAKE_LIST` to this argument makes FDPS construct new interaction lists and perform interaction calculation using these lists, and tells FDPS that the lists used this time are not used in the next interaction calculation.

### 4.2.4 Compilation of the program

Run `make` at the working directory. You can use the Makefile attached to the sample code.

```
$ make
```

### 4.2.5 Execution

To run the code without MPI, you should execute the following command in the command shell.

```
$ ./sph.out
```

To run the code using MPI, you should execute the following command in the command shell, or follow the document of your system.

```
$ MPIRUN -np NPROC ./sph.out
```

Here, `MPIRUN` represents the command to run your program using MPI such as `mpirun` or

`mpiexec`, and `NPROC` is the number of MPI processes.

### 4.2.6 Log and output files

Log and output files are created under `result` directory.

### 4.2.7 Visualization

In this section, we describe how to visualize the calculation result using `gnuplot`. To enter the interactive mode of `gnuplot`, execute the following command.

```
$ gnuplot
```

In the interactive mode, you can visualize the result. In the following example, using the 50th snapshot file, we create the plot in which the abscissa is the x coordinate of particles and the ordinate is the density of particles.

```
gnuplot> plot "result/snap00050-proc00000.dat" u 3:9
```

where the integral number after the string of characters `proc` represents the rank number of a MPI process.

# 5 Sample Codes

## 5.1 $N$-body simulation

In this section, we show a sample code for the $N$-body simulation. This code is the same as what we described in section 4. One can create a working code by cut and paste this code and compile and link the resulted source program.

Listing 32: Sample code of $N$-body simulation (`user_defined.h`)

```
#pragma once
/* Standard headers */
#include <math.h>
/* FDPS headers */
#include "FDPS_c_if.h"

typedef struct full_particle { //$fdps FP,EPI,EPJ,Force
    //$fdps copyFromForce full_particle (pot,pot) (acc,acc)
    //$fdps copyFromFP full_particle (id,id) (mass,mass) (eps,eps) (pos,
        pos)
    //$fdps clear id=keep, mass=keep, eps=keep, pos=keep, vel=keep
    long long id; // $fdps id
    double  mass; //$fdps charge
    double  eps;
    fdps_f64vec pos; //$fdps position
    fdps_f64vec vel; //$fdps velocity
    double  pot;
    fdps_f64vec acc;
} Full_particle;

void  calc_gravity_ep_ep(Full_particle *ep_i,
                         int n_ip,
                         Full_particle *ep_j,
                         int n_jp,
                         Full_particle *f);

void  calc_gravity_ep_sp(Full_particle *ep_i,
                         int n_ip,
                         fdps_spj_monopole *ep_j,
                         int n_jp,
                         Full_particle *f);
```

Listing 33: Sample code of $N$-body simulation (`user_defined.c`)

```
#include "user_defined.h"

void calc_gravity_ep_ep(Full_particle *ep_i,
                        int n_ip,
                        Full_particle *ep_j,
                        int n_jp,
                        Full_particle* f)
{
    int i, j;
    for (i=0; i<n_ip ;i++) {
        Full_particle *pi = ep_i + i;
```

```
12          double eps2 = pi->eps * pi->eps;
13          double xi = pi->pos.x;
14          double yi = pi->pos.y;
15          double zi = pi->pos.z;
16          double ax, ay, az, pot;
17          ax = ay = az = pot = 0;
18          for (j=0; j<n_jp; j++) {
19              Full_particle *pj = ep_j + j;
20              double dx = xi - pj->pos.x;
21              double dy = yi - pj->pos.y;
22              double dz = zi - pj->pos.z;
23              double r2 = dx*dx+dy*dy+dz*dz+eps2;
24              double rinv = 1.0/sqrt(r2);
25              double mrinv = pj->mass* rinv;
26              double mr3inv = mrinv*rinv*rinv;
27              ax -= dx*mr3inv;
28              ay -= dy*mr3inv;
29              az -= dz*mr3inv;
30              pot = pot - mrinv;
31          }
32          Full_particle *pfi = f+i;
33          pfi->pot += pot;
34          pfi->acc.x += ax;
35          pfi->acc.y += ay;
36          pfi->acc.z += az;
37      }
38  }
39
40  void calc_gravity_ep_sp(Full_particle *ep_i,
41                          int n_ip,
42                          fdps_spj_monopole *ep_j,
43                          int n_jp,
44                          Full_particle *f)
45  {
46      int i, j;
47      for (i=0; i<n_ip; i++) {
48          Full_particle *pi = ep_i + i;
49          double eps2 = pi->eps*pi->eps;
50          double xi = pi->pos.x;
51          double yi = pi->pos.y;
52          double zi = pi->pos.z;
53          double ax, ay, az, pot;
54          ax = ay = az = pot = 0;
55          for (j=0; j<n_jp; j++) {
56              fdps_spj_monopole *pj = ep_j + j;
57              double dx = xi - pj->pos.x;
58              double dy = yi - pj->pos.y;
59              double dz = zi - pj->pos.z;
60              double r2 = dx*dx+dy*dy+dz*dz+eps2;
61              double rinv = 1.0/sqrt(r2);
62              double mrinv = pj->mass* rinv;
63              double mr3inv = mrinv*rinv*rinv;
64              ax -= dx*mr3inv;
65              ay -= dy*mr3inv;
66              az -= dz*mr3inv;
```

```
67              pot = pot - mrinv;
68          }
69          Full_particle *pfi = f+i;
70          pfi->pot += pot;
71          pfi->acc.x += ax;
72          pfi->acc.y += ay;
73          pfi->acc.z += az;
74      }
75  }
```

Listing 34: Sample code of $N$-body simulation (`c_main.c`)

```
1  /* Standard headers */
2  #include <stdio.h>
3  #include <stdlib.h>
4  #include <stdbool.h>
5  #include <math.h>
6  /* FDPS headers */
7  #include "user_defined.h"
8  #include "FDPS_c_if.h"
9
10 void dump_fullp(Full_particle p)
11 {
12     printf("%lld␣%15.7e␣%15.7e␣%15.7e␣%15.7e␣%15.7e␣%15.7e␣%15.7e",
13            p.id, p.mass, p.pos.x, p.pos.y, p.pos.z,
14            p.vel.x, p.vel.y, p.vel.z);
15     printf("%15.7e␣%15.7e␣%15.7e␣%15.7e\n",
16            p.acc.x, p.acc.y, p.acc.z, p.pot);
17 }
18 void dump_fullpsys(Full_particle *p, int n)
19 {
20     int i;
21     for (i=0;i<n;i++)dump_fullp(p[i]);
22 }
23
24 void dump_particles(int psys_num)
25 {
26     Full_particle *ptcl;
27     ptcl = (Full_particle *) fdps_get_psys_cptr(psys_num);
28     int n = fdps_get_nptcl_loc(psys_num);
29     dump_fullpsys(ptcl, n);
30 }
31
32 void setup_IC(int psys_num,
33               int nptcl_glb)
34 {
35
36     double m_tot=1.0;
37     double rmax=3.0;
38     double r2max=rmax*rmax;
39     //   Get # of MPI processes and rank number
40     int nprocs = fdps_get_num_procs();
41     int myrank = fdps_get_rank();
42     // Make an initial condition at RANK 0
43     if (myrank == 0 ){
44         //Set # of local particles
```

```
45          fdps_set_nptcl_loc(psys_num,nptcl_glb);
46          Full_particle *ptcl;
47          ptcl = (Full_particle *) fdps_get_psys_cptr(psys_num);
48          //** initialize Mersenne twister
49          int mtts_num;
50          fdps_create_mtts(&mtts_num);
51          fdps_mtts_init_genrand(mtts_num,0);
52          int i;
53          for (i=0; i < nptcl_glb; i++){
54              Full_particle *q = ptcl+i;
55              q->id = i;
56              q->mass = m_tot/nptcl_glb;
57              double r2 = r2max*2;
58              fdps_f64vec pos;
59              while (r2 >= r2max){
60                  pos.x= (2*fdps_mtts_genrand_res53(mtts_num)-1.0) * rmax;
61                  pos.y= (2*fdps_mtts_genrand_res53(mtts_num)-1.0) * rmax;
62                  pos.z= (2*fdps_mtts_genrand_res53(mtts_num)-1.0) * rmax;
63                  r2 = pos.x*pos.x + pos.y*pos.y + pos.z*pos.z;
64              }
65              q->pos = pos;
66              q->vel.x = 0.0;
67              q->vel.y = 0.0;
68              q->vel.z = 0.0;
69              q->eps = 1.0/32.0;
70          }
71          fdps_f64vec cm_pos;
72          fdps_f64vec cm_vel;
73          cm_pos.x = 0.0; cm_pos.y = 0.0; cm_pos.z = 0.0;
74          cm_vel.x = 0.0; cm_vel.y = 0.0; cm_vel.z = 0.0;
75          double cm_mass = 0;
76          for (i=0; i < nptcl_glb; i++){
77              Full_particle *pi = ptcl+i;
78              cm_pos.x +=  pi->pos.x* pi->mass;
79              cm_pos.y +=  pi->pos.y* pi->mass;
80              cm_pos.z +=  pi->pos.z* pi->mass;
81              cm_vel.x +=  pi->vel.x* pi->mass;
82              cm_vel.y +=  pi->vel.y* pi->mass;
83              cm_vel.z +=  pi->vel.z* pi->mass;
84              cm_mass += pi->mass;
85          }
86          cm_pos.x /= cm_mass;
87          cm_pos.y /= cm_mass;
88          cm_pos.z /= cm_mass;
89          cm_vel.x /= cm_mass;
90          cm_vel.y /= cm_mass;
91          cm_vel.z /= cm_mass;
92          for (i=0; i < nptcl_glb; i++){
93              Full_particle* q = ptcl+i;
94              q->pos.x -= cm_pos.x;
95              q->pos.y -= cm_pos.y;
96              q->pos.z -= cm_pos.z;
97              q->vel.x -= cm_vel.x;
98              q->vel.y -= cm_vel.y;
99              q->vel.z -= cm_vel.z;
```

```
100         }
101         //dump_fullpsys(ptcl, nptcl_glb);
102     } else{
103         fdps_set_nptcl_loc(psys_num,0);
104     }
105 }
106
107 void calc_energy(int psys_num,
108                  double *etot,
109                  double *ekin,
110                  double *epot)
111 {
112     *etot = *ekin = *epot = 0;
113     int nptcl_loc = fdps_get_nptcl_loc(psys_num);
114     Full_particle *ptcl;
115     ptcl = (Full_particle *) fdps_get_psys_cptr(psys_num);
116
117     double  ekin_loc = 0.0;
118     double  epot_loc = 0.0;
119     int i;
120     for (i=0;i < nptcl_loc; i++){
121         Full_particle *pi = ptcl+i;
122         fdps_f64vec v = pi->vel;
123         ekin_loc += pi->mass * (v.x*v.x+v.y*v.y+v.z*v.z);
124         epot_loc += pi->mass * (pi->pot + pi->mass/pi->eps);
125     }
126     ekin_loc *= 0.5;
127     epot_loc *= 0.5;
128     double etot_loc = ekin_loc + epot_loc;
129     *ekin = fdps_get_sum_f64(ekin_loc);
130     *epot = fdps_get_sum_f64(epot_loc);
131     *etot = fdps_get_sum_f64(etot_loc);
132 }
133
134 void  kick(int psys_num, double dt)
135 {
136     Full_particle *ptcl;
137     ptcl = (Full_particle *) fdps_get_psys_cptr(psys_num);
138     int n = fdps_get_nptcl_loc(psys_num);
139     int i;
140     for (i=0;i < n; i++){
141         Full_particle *pi = ptcl+i;
142         fdps_f64vec *pv, *pa;
143         pv = &(pi->vel);
144         pa = &(pi->acc);
145         pv->x += pa->x * dt;
146         pv->y += pa->y * dt;
147         pv->z += pa->z * dt;
148     }
149 }
150
151 void  drift(int psys_num, double dt)
152 {
153     Full_particle *ptcl;
154     ptcl = (Full_particle *) fdps_get_psys_cptr(psys_num);
```

```
155     int n = fdps_get_nptcl_loc(psys_num);
156     int i;
157     for (i=0;i < n; i++){
158         Full_particle *pi = ptcl+i;
159         fdps_f64vec *px, *pv;
160         pv = &(pi->vel);
161         px = &(pi->pos);
162         px->x += pv->x * dt;
163         px->y += pv->y * dt;
164         px->z += pv->z * dt;
165     }
166 }
167
168 void output(int psys_num) {
169    // Set file name
170    static int snap_num = 0;
171    int myrank = fdps_get_rank();
172    char fname[64];
173    sprintf(fname,"./result/snap%05d-proc%05d.dat",snap_num,myrank);
174    // Open file
175    FILE *fp;
176    if ((fp = fopen(fname,"w")) == NULL) {
177       fprintf(stderr,"failed␣to␣open␣file␣%s.\n", fname);
178       exit(EXIT_FAILURE);
179    }
180    // Output
181    Full_particle *ptcl;
182    ptcl = (Full_particle *) fdps_get_psys_cptr(psys_num);
183    int i;
184    int n = fdps_get_nptcl_loc(psys_num);
185    for (i = 0; i < n; i++) {
186       Full_particle *pi = ptcl + i;
187       fprintf(fp,"%ld␣␣%25.15e␣␣%25.15e␣␣%25.15e␣␣%25.15e␣␣%25.15e␣␣%25.15
              e␣␣%25.15e\n",
188              pi->id,pi->mass,pi->pos.x,pi->pos.y,pi->pos.z,
189              pi->vel.x,pi->vel.y,pi->vel.z);
190    }
191    fclose(fp);
192    // Update snap_num
193    snap_num++;
194
195 }
196
197 int c_main()
198 {
199     fprintf(stderr, "FDPS␣on␣C␣test␣code\n");
200     fdps_initialize();
201     // Create and initialize dinfo object
202     int dinfo_num;
203     float coef_ema=0.3;
204     fdps_create_dinfo(&dinfo_num);
205     fdps_init_dinfo(dinfo_num,coef_ema);
206     // Create and initialize psys object
207     int psys_num;
208     fdps_create_psys(&psys_num,"full_particle");
```

```
209     fdps_init_psys(psys_num);
210     // Create and initialize tree object
211     int tree_num;
212     fdps_create_tree(&tree_num,
213                      "Long,full_particle,full_particle,full_particle,
                               Monopole");
214     int ntot=1024;
215     double theta = 0.5;
216     int n_leaf_limit = 8;
217     int n_group_limit = 64;
218     fdps_init_tree(tree_num, ntot, theta, n_leaf_limit, n_group_limit);
219     // Make an initial condition
220     setup_IC(psys_num,ntot);
221     // Domain decomposition and exchange particle
222     fdps_decompose_domain_all(dinfo_num,psys_num,-1.0);
223     fdps_exchange_particle(psys_num,dinfo_num);
224     // Compute force at the initial time
225     fdps_calc_force_all_and_write_back(tree_num,
226                                        calc_gravity_ep_ep,
227                                        calc_gravity_ep_sp,
228                                        psys_num,
229                                        dinfo_num,
230                                        true,
231                                        FDPS_MAKE_LIST);
232     //dump_particles(psys_num);
233     // Compute energies at the initial time
234     double etot0,ekin0,epot0;
235     calc_energy(psys_num, &etot0, &ekin0,&epot0);
236     printf("Energies␣=␣%21.14e␣␣%21.14e␣␣%21.14e\n",etot0,ekin0,epot0);
237     // Time integration
238     double time_diag = 0;
239     double time_snap = 0;
240     double time_sys  = 0;
241     double time_end = 10.0;
242     double dt = 1.0/128.0;
243     double dt_diag = 1.0;
244     double dt_snap = 1.0;
245     int    num_loop = 0;
246     while (time_sys <= time_end){
247         if (time_sys + dt/2 >= time_snap){
248             output(psys_num);
249             time_snap += dt_snap;
250         }
251         double etot1, ekin1, epot1;
252         calc_energy(psys_num, &etot1,&ekin1,&epot1);
253         //printf( "Energies = %21.14e  %21.14e  %21.14e\n",etot1,ekin1,
                epot1);
254         //dump_particles(psys_num);
255         if (fdps_get_rank() == 0){
256             if (time_sys + dt/2 >= time_diag){
257                 printf ("time:␣%10.3f,␣energy␣error:␣%15.7e\n",
258                         time_sys, (etot1-etot0)/etot0);
259                 time_diag = time_diag + dt_diag;
260             }
261         }
```

```
262         kick(psys_num,0.5*dt);
263         time_sys += dt;
264         drift(psys_num,dt);
265         // Domain decomposition & exchange particle
266         if (num_loop %4  == 0) {
267             fdps_decompose_domain_all(dinfo_num,psys_num,-1.0);
268         }
269         fdps_exchange_particle(psys_num,dinfo_num);
270         // Force calculation
271         fdps_calc_force_all_and_write_back(tree_num,
272                                            calc_gravity_ep_ep,
273                                            calc_gravity_ep_sp,
274                                            psys_num,
275                                            dinfo_num,
276                                            true,
277                                            FDPS_MAKE_LIST);
278         kick(psys_num,0.5*dt);
279         num_loop += 1;
280     }
281     fdps_finalize();
282     return 0;
283 }
```

## 5.2 SPH simulation with fixed smoothing length

In this section, we show a sample code for the SPH simulation with fixed smoothing length. This code is the same as what we described in section 4. One can create a working code by cut and paste this code and compile and link the resulted source program.

Listing 35: Sample code of SPH simulation (`user_defined.h`)

```
1  #pragma once
2  /* Standard headers */
3  #include <math.h>
4  /* FDPS headers */
5  #include "FDPS_c_if.h"
6  /* User-defined headers */
7  #include "mathematical_constants.h"
8
9  /* Force types */
10 typedef struct force_dens { //$fdps Force
11     //$fdps clear smth=keep
12     double dens;
13     double smth;
14 } Force_dens;
15
16 typedef struct force_hydro { //$fdps Force
17     //$fdps clear
18     fdps_f64vec acc;
19     double eng_dot;
20     double dt;
21 } Force_hydro;
22
23 /* Full particle type */
```

```
24 typedef struct full_particle { //$fdps FP
25     //$fdps copyFromForce force_dens (dens,dens)
26     //$fdps copyFromForce force_hydro (acc,acc) (eng_dot,eng_dot) (dt,dt)
27     double mass; //$fdps charge
28     fdps_f64vec pos; //$fdps position
29     fdps_f64vec vel;
30     fdps_f64vec acc;
31     double dens;
32     double eng;
33     double pres;
34     double smth; //$fdps rsearch
35     double snds;
36     double eng_dot;
37     double dt;
38     long long int id;
39     fdps_f64vec vel_half;
40     double eng_half;
41 } Full_particle;
42
43 /* Essential particle type */
44 typedef struct essential_particle { //$fdps EPI,EPJ
45     //$fdps copyFromFP full_particle (id,id) (pos,pos) (vel,vel) (mass,
            mass) (smth,smth) (dens,dens) (pres,pres) (snds,snds)
46     long long int id;
47     fdps_f64vec pos; //$fdps position
48     fdps_f64vec vel;
49     double mass; //$fdps charge
50     double smth; //$fdps rsearch
51     double dens;
52     double pres;
53     double snds;
54 } Essential_particle;
55
56 /* Prototype declarations */
57 double W(fdps_f64vec dr, double h);
58 fdps_f64vec gradW(fdps_f64vec dr, double h);
59
60 void calc_density(Essential_particle *ep_i,
61                   int n_ip,
62                   Essential_particle *ep_j,
63                   int n_jp,
64                   Force_dens *f);
65 void calc_hydro_force(Essential_particle *ep_i,
66                       int n_ip,
67                       Essential_particle *ep_j,
68                       int n_jp,
69                       Force_hydro *f);
70
71 /* Gloabl variable */
72 extern const double kernel_support_radius;
```

Listing 36: Sample code of SPH simulation (`user_defined.c`)

```
1 #include "user_defined.h"
2
3 /* Global variable */
```

```
 4 const double kernel_support_radius=2.5;
 5
 6 /* Kernel functions */
 7 double W(fdps_f64vec dr, double h) {
 8     double s,s1,s2,ret;
 9     s = sqrt(dr.x * dr.x
10              +dr.y * dr.y
11              +dr.z * dr.z)/h;
12     s1 = 1.0 - s;
13     if (s1 < 0.0) s1 = 0.0;
14     s2 = 0.5 - s;
15     if (s2 < 0.0) s2 = 0.0;
16     ret = (s1*s1*s1) - 4.0*(s2*s2*s2);
17     ret = ret * 16.0e0/(pi*h*h*h);
18     return ret;
19 }
20
21 fdps_f64vec gradW(fdps_f64vec dr, double h) {
22     double dr_abs,s,s1,s2,coef;
23     fdps_f64vec ret;
24     dr_abs = sqrt(dr.x * dr.x
25                   +dr.y * dr.y
26                   +dr.z * dr.z);
27     s = dr_abs/h;
28     s1 = 1.0 - s;
29     if (s1 < 0.0) s1 = 0.0;
30     s2 = 0.5 - s;
31     if (s2 < 0.0) s2 = 0.0;
32     coef = - 3.0*(s1*s1) + 12.0*(s2*s2);
33     coef = coef * 16.0/(pi*h*h*h);
34     coef = coef / (dr_abs*h + 1.0e-6*h);
35     ret.x = dr.x * coef;
36     ret.y = dr.y * coef;
37     ret.z = dr.z * coef;
38     return ret;
39 }
40
41 /* Interaction functions */
42 void calc_density(Essential_particle *ep_i,
43                   int n_ip,
44                   Essential_particle *ep_j,
45                   int n_jp,
46                   Force_dens *f) {
47     int i,j;
48     fdps_f64vec dr;
49     for (i = 0; i< n_ip; i++) {
50         for (j = 0; j < n_jp; j++) {
51             dr.x = ep_j[j].pos.x - ep_i[i].pos.x;
52             dr.y = ep_j[j].pos.y - ep_i[i].pos.y;
53             dr.z = ep_j[j].pos.z - ep_i[i].pos.z;
54             f[i].dens += ep_j[j].mass * W(dr,ep_i[i].smth);
55         }
56     }
57
58 }
```

```
59
60 void calc_hydro_force(Essential_particle *ep_i,
61                        int n_ip,
62                        Essential_particle *ep_j,
63                        int n_jp,
64                        Force_hydro *f) {
65     // Local parameters
66     const double C_CFL=0.3;
67     // Local variables
68     int i,j;
69     double mass_i,mass_j,smth_i,smth_j,
70            dens_i,dens_j,pres_i,pres_j,
71            snds_i,snds_j;
72     double povrho2_i,povrho2_j,
73            v_sig_max,dr_dv,w_ij,v_sig,AV;
74     fdps_f64vec pos_i,pos_j,vel_i,vel_j,
75                 dr,dv,gradW_i,gradW_j,gradW_ij;
76
77     for (i = 0; i < n_ip; i++) {
78         // Zero-clear
79         v_sig_max = 0.0;
80         // Extract i-particle info.
81         pos_i.x = ep_i[i].pos.x;
82         pos_i.y = ep_i[i].pos.y;
83         pos_i.z = ep_i[i].pos.z;
84         vel_i.x = ep_i[i].vel.x;
85         vel_i.y = ep_i[i].vel.y;
86         vel_i.z = ep_i[i].vel.z;
87         mass_i  = ep_i[i].mass;
88         smth_i  = ep_i[i].smth;
89         dens_i  = ep_i[i].dens;
90         pres_i  = ep_i[i].pres;
91         snds_i  = ep_i[i].snds;
92         povrho2_i = pres_i/(dens_i*dens_i);
93         for (j = 0; j < n_jp; j++) {
94             // Extract j-particle info.
95             pos_j.x = ep_j[j].pos.x;
96             pos_j.y = ep_j[j].pos.y;
97             pos_j.z = ep_j[j].pos.z;
98             vel_j.x = ep_j[j].vel.x;
99             vel_j.y = ep_j[j].vel.y;
100            vel_j.z = ep_j[j].vel.z;
101            mass_j  = ep_j[j].mass;
102            smth_j  = ep_j[j].smth;
103            dens_j  = ep_j[j].dens;
104            pres_j  = ep_j[j].pres;
105            snds_j  = ep_j[j].snds;
106            povrho2_j = pres_j/(dens_j*dens_j);
107            // Compute dr & dv
108            dr.x = pos_i.x - pos_j.x;
109            dr.y = pos_i.y - pos_j.y;
110            dr.z = pos_i.z - pos_j.z;
111            dv.x = vel_i.x - vel_j.x;
112            dv.y = vel_i.y - vel_j.y;
113            dv.z = vel_i.z - vel_j.z;
```

```
114             // Compute the signal velocity
115             dr_dv = dr.x * dv.x + dr.y * dv.y + dr.z * dv.z;
116             if (dr_dv < 0.0) {
117                w_ij = dr_dv / sqrt(dr.x * dr.x + dr.y * dr.y + dr.z * dr.z
                        );
118             } else {
119                w_ij = 0.0;
120             }
121             v_sig = snds_i + snds_j - 3.0 * w_ij;
122             if (v_sig > v_sig_max) v_sig_max=v_sig;
123             // Compute the artificial viscosity
124             AV = - 0.5*v_sig*w_ij / (0.5*(dens_i+dens_j));
125             // Compute the average of the gradients of kernel
126             gradW_i  = gradW(dr,smth_i);
127             gradW_j  = gradW(dr,smth_j);
128             gradW_ij.x = 0.5 * (gradW_i.x + gradW_j.x);
129             gradW_ij.y = 0.5 * (gradW_i.y + gradW_j.y);
130             gradW_ij.z = 0.5 * (gradW_i.z + gradW_j.z);
131             // Compute the acceleration and the heating rate
132             f[i].acc.x -= mass_j*(povrho2_i+povrho2_j+AV)*gradW_ij.x;
133             f[i].acc.y -= mass_j*(povrho2_i+povrho2_j+AV)*gradW_ij.y;
134             f[i].acc.z -= mass_j*(povrho2_i+povrho2_j+AV)*gradW_ij.z;
135             f[i].eng_dot += mass_j * (povrho2_i + 0.5*AV)
136                             *(dv.x * gradW_ij.x
137                              +dv.y * gradW_ij.y
138                              +dv.z * gradW_ij.z);
139          }
140          f[i].dt = C_CFL*2.0*smth_i/(v_sig_max*kernel_support_radius);
141       }
142    }
```

Listing 37: Sample code of SPH simulation (`c_main.c`)

```
1  /* Standard headers */
2  #include <stdio.h>
3  #include <stdlib.h>
4  #include <stdbool.h>
5  #include <math.h>
6  /* FDPS headers */
7  #include "FDPS_c_if.h"
8  /* user-defined headers */
9  #include "mathematical_constants.h"
10 #include "user_defined.h"
11
12 void setup_IC(int psys_num,
13               double *end_time,
14               fdps_f32vec *pos_ll,
15               fdps_f32vec *pos_ul) {
16    // Get # of MPI processes and rank number
17    int nprocs = fdps_get_num_procs();
18    int myrank = fdps_get_rank();
19
20    // Set the box size
21    pos_ll->x = 0.0;
22    pos_ll->y = 0.0;
23    pos_ll->z = 0.0;
```

```
24      pos_ul->x = 1.0;
25      pos_ul->y = pos_ul->x / 8.0;
26      pos_ul->z = pos_ul->x / 8.0;
27
28      // Make an initial condition at RANK 0
29      if (myrank == 0) {
30          // Set the left and right states
31          const double dens_L = 1.0;
32          const double eng_L  = 2.5;
33          const double dens_R = 0.5;
34          const double eng_R  = 2.5;
35          // Set the separation of particle of the left state
36          const double dx = 1.0 / 128.0;
37          const double dy = dx;
38          const double dz = dx;
39          // Set the number of local particles
40          int nptcl_glb = 0;
41          // (1) Left-half
42          const int nx_L = 0.5*pos_ul->x/dx;
43          const int ny_L = pos_ul->y/dy;
44          const int nz_L = pos_ul->z/dz;
45          nptcl_glb += nx_L * ny_L * nz_L;
46          printf("nptcl_glb(L)␣␣␣=␣%d\n",nptcl_glb);
47          // (2) Right-half
48          const int nx_R = 0.5*pos_ul->x/((dens_L/dens_R)*dx);
49          const int ny_R = ny_L;
50          const int nz_R = nz_L;
51          nptcl_glb += nx_R * ny_R * nz_R;
52          printf("nptcl_glb(L+R)␣=␣%d\n",nptcl_glb);
53          // Place SPH particles
54          fdps_set_nptcl_loc(psys_num,nptcl_glb);
55          Full_particle *ptcl = (Full_particle *) fdps_get_psys_cptr(
                psys_num);
56          int id = -1;
57          // (1) Left-half
58          int i,j,k;
59          for (i = 0; i < nx_L; i++) {
60              for (j = 0; j < ny_L; j++) {
61                  for (k = 0; k < nz_L; k++) {
62                      id++;
63                      ptcl[id].id    = id;
64                      ptcl[id].pos.x = dx * i;
65                      ptcl[id].pos.y = dy * j;
66                      ptcl[id].pos.z = dz * k;
67                      ptcl[id].dens  = dens_L;
68                      ptcl[id].eng   = eng_L;
69                  }
70              }
71          }
72          // (2) Right-half
73          for (i = 0; i < nx_R; i++) {
74              for (j = 0; j < ny_R; j++) {
75                  for (k = 0; k < nz_R; k++) {
76                      id++;
77                      ptcl[id].id    = id;
```

```
78                      ptcl[id].pos.x = 0.5*pos_ul->x + ((dens_L/dens_R)*dx)*
                              i;
79                      ptcl[id].pos.y = dy * j;
80                      ptcl[id].pos.z = dz * k;
81                      ptcl[id].dens  = dens_R;
82                      ptcl[id].eng   = eng_R;
83                  }
84              }
85          }
86          printf("nptcl(L+R) = %d\n",id+1);
87          // Set particle mass and smoothing length
88          for (i = 0; i < nptcl_glb; i++) {
89              ptcl[i].mass = 0.5*(dens_L+dens_R)
90                           * (pos_ul->x*pos_ul->y*pos_ul->z)
91                           / nptcl_glb;
92              ptcl[i].smth = kernel_support_radius * 0.012;
93          }
94      } else {
95          fdps_set_nptcl_loc(psys_num,0);
96      }
97
98      // Set the end time
99      *end_time = 0.12;
100
101     // Inform to STDOUT
102     if (fdps_get_rank() == 0) printf("setup...completed!\n");
103     //fdps_finalize();
104     //exit(0);
105
106 }
107
108 double get_timestep(int psys_num) {
109     int nptcl_loc = fdps_get_nptcl_loc(psys_num);
110     Full_particle *ptcl = (Full_particle *) fdps_get_psys_cptr(psys_num);
111     double dt_loc = 1.0e30;
112     int i;
113     for (i = 0; i < nptcl_loc; i++)
114         if (ptcl[i].dt < dt_loc)
115             dt_loc = ptcl[i].dt;
116     return fdps_get_min_value_f64(dt_loc);
117 }
118
119 void initial_kick(int psys_num, double dt) {
120     int nptcl_loc = fdps_get_nptcl_loc(psys_num);
121     Full_particle *ptcl = (Full_particle *) fdps_get_psys_cptr(psys_num);
122     int i;
123     for (i = 0; i < nptcl_loc; i++) {
124         ptcl[i].vel_half.x = ptcl[i].vel.x + 0.5 * dt * ptcl[i].acc.x;
125         ptcl[i].vel_half.y = ptcl[i].vel.y + 0.5 * dt * ptcl[i].acc.y;
126         ptcl[i].vel_half.z = ptcl[i].vel.z + 0.5 * dt * ptcl[i].acc.z;
127         ptcl[i].eng_half = ptcl[i].eng + 0.5 * dt * ptcl[i].eng_dot;
128     }
129 }
130
131 void full_drift(int psys_num, double dt) {
```

```
132     int nptcl_loc = fdps_get_nptcl_loc(psys_num);
133     Full_particle *ptcl = (Full_particle *) fdps_get_psys_cptr(psys_num);
134     int i;
135     for (i = 0; i < nptcl_loc; i++) {
136         ptcl[i].pos.x += dt * ptcl[i].vel_half.x;
137         ptcl[i].pos.y += dt * ptcl[i].vel_half.y;
138         ptcl[i].pos.z += dt * ptcl[i].vel_half.z;
139     }
140 }
141
142 void predict(int psys_num, double dt) {
143     int nptcl_loc = fdps_get_nptcl_loc(psys_num);
144     Full_particle *ptcl = (Full_particle *) fdps_get_psys_cptr(psys_num);
145     int i;
146     for (i = 0; i < nptcl_loc; i++) {
147         ptcl[i].vel.x += dt * ptcl[i].acc.x;
148         ptcl[i].vel.y += dt * ptcl[i].acc.y;
149         ptcl[i].vel.z += dt * ptcl[i].acc.z;
150         ptcl[i].eng += dt * ptcl[i].eng_dot;
151     }
152 }
153
154 void final_kick(int psys_num, double dt) {
155     int nptcl_loc = fdps_get_nptcl_loc(psys_num);
156     Full_particle *ptcl = (Full_particle *) fdps_get_psys_cptr(psys_num);
157     int i;
158     for (i = 0; i < nptcl_loc; i++) {
159         ptcl[i].vel.x = ptcl[i].vel_half.x + 0.5 * dt * ptcl[i].acc.x;
160         ptcl[i].vel.y = ptcl[i].vel_half.y + 0.5 * dt * ptcl[i].acc.y;
161         ptcl[i].vel.z = ptcl[i].vel_half.z + 0.5 * dt * ptcl[i].acc.z;
162         ptcl[i].eng = ptcl[i].eng_half + 0.5 * dt * ptcl[i].eng_dot;
163     }
164 }
165
166 void set_pressure(int psys_num) {
167     const double hcr=1.4;
168     int nptcl_loc = fdps_get_nptcl_loc(psys_num);
169     Full_particle *ptcl = (Full_particle *) fdps_get_psys_cptr(psys_num);
170     int i;
171     for (i = 0; i < nptcl_loc; i++) {
172         ptcl[i].pres = (hcr - 1.0) * ptcl[i].dens * ptcl[i].eng;
173         ptcl[i].snds = sqrt(hcr * ptcl[i].pres / ptcl[i].dens);
174     }
175 }
176
177 void output(int psys_num, int nstep) {
178     int myrank = fdps_get_rank();
179     int nptcl_loc = fdps_get_nptcl_loc(psys_num);
180     Full_particle *ptcl = (Full_particle *) fdps_get_psys_cptr(psys_num);
181     char filename[64] = {'\0'};
182     sprintf(filename,"./result/snap%05d-proc%05d.txt",nstep,myrank);
183     FILE *fp;
184     if ((fp = fopen(filename,"w")) == NULL) {
185         fprintf(stderr,"Cannot␣open␣file␣%s\n",filename);
186         exit(EXIT_FAILURE);
```

```
187     }
188     int i;
189     for (i = 0; i < nptcl_loc; i++) {
190         fprintf(fp,"%ld␣%15.7e␣%15.7e␣%15.7e␣%15.7e",
191                 ptcl[i].id,ptcl[i].mass,
192                 ptcl[i].pos.x,ptcl[i].pos.y,ptcl[i].pos.z);
193         fprintf(fp,"%15.7e␣%15.7e␣%15.7e␣%15.7e␣%15.7e␣%15.7e\n",
194                 ptcl[i].vel.x,ptcl[i].vel.y,ptcl[i].vel.z,
195                 ptcl[i].dens,ptcl[i].eng,ptcl[i].pres);
196     }
197     fclose(fp);
198 }
199
200 void check_cnsrvd_vars(int psys_num){
201     int nptcl_loc = fdps_get_nptcl_loc(psys_num);
202     Full_particle *ptcl = fdps_get_psys_cptr(psys_num);
203     fdps_f64vec mom_loc;
204     mom_loc.x = 0.0; mom_loc.y = 0.0; mom_loc.z = 0.0;
205     double eng_loc = 0.0;
206     int i;
207     for (i = 0; i < nptcl_loc; i++) {
208         mom_loc.x += ptcl[i].vel.x * ptcl[i].mass;
209         mom_loc.y += ptcl[i].vel.y * ptcl[i].mass;
210         mom_loc.z += ptcl[i].vel.z * ptcl[i].mass;
211         eng_loc += ptcl[i].mass *(ptcl[i].eng
212                                   + 0.5*(ptcl[i].vel.x * ptcl[i].vel.x
213                                         +ptcl[i].vel.y * ptcl[i].vel.y
214                                         +ptcl[i].vel.z * ptcl[i].vel.z));
215     }
216     double eng = fdps_get_sum_f64(eng_loc);
217     fdps_f64vec mom;
218     mom.x = fdps_get_sum_f64(mom_loc.x);
219     mom.y = fdps_get_sum_f64(mom_loc.y);
220     mom.z = fdps_get_sum_f64(mom_loc.z);
221     if (fdps_get_rank() == 0) {
222         printf("eng␣␣␣=␣%15.7e\n",eng);
223         printf("mom.x␣=␣%15.7e\n",mom.x);
224         printf("mom.y␣=␣%15.7e\n",mom.y);
225         printf("mom.z␣=␣%15.7e\n",mom.z);
226     }
227 }
228
229 void c_main() {
230     // Initialize some global variables
231     setup_math_const();
232
233     // Initialize FDPS
234     fdps_initialize();
235
236     // Make an instance of ParticleSystem and initialize it
237     int psys_num;
238     fdps_create_psys(&psys_num,"full_particle");
239     fdps_init_psys(psys_num);
240
241     // Make an initial condition and initialize the particle system
```

```
242     double end_time;
243     fdps_f32vec pos_ll, pos_ul;
244     setup_IC(psys_num,&end_time,&pos_ll,&pos_ul);
245
246     // Make an instance of DomainInfo and initialize it
247     int dinfo_num;
248     fdps_create_dinfo(&dinfo_num);
249     float coef_ema = 0.3;
250     fdps_init_dinfo(dinfo_num,coef_ema);
251     fdps_set_boundary_condition(dinfo_num,FDPS_BC_PERIODIC_XYZ);
252     fdps_set_pos_root_domain(dinfo_num,&pos_ll,&pos_ul);
253
254     // Perform domain decomposition and exchange particles
255     fdps_decompose_domain_all(dinfo_num,psys_num,-1.0);
256     fdps_exchange_particle(psys_num,dinfo_num);
257
258     // Make two tree structures
259     int ntot = fdps_get_nptcl_glb(psys_num);
260     // tree_dens (used for the density calculation)
261     int tree_num_dens;
262     fdps_create_tree(&tree_num_dens,
263                      "Short,force_dens,essential_particle,
                               essential_particle,Gather");
264     float theta = 0.5;
265     int n_leaf_limit = 8;
266     int n_group_limit = 64;
267     fdps_init_tree(tree_num_dens,3*ntot,theta,n_leaf_limit,n_group_limit);
268
269     // tree_hydro (used for the force calculation)
270     int tree_num_hydro;
271     fdps_create_tree(&tree_num_hydro,
272                      "Short,force_hydro,essential_particle,
                               essential_particle,Symmetry");
273     fdps_init_tree(tree_num_hydro,3*ntot,theta,n_leaf_limit,n_group_limit)
            ;
274
275     // Compute density, pressure, acceleration due to pressure gradient
276     fdps_calc_force_all_and_write_back(tree_num_dens,
277                                        calc_density,
278                                        NULL,
279                                        psys_num,
280                                        dinfo_num,
281                                        true,
282                                        FDPS_MAKE_LIST);
283     set_pressure(psys_num);
284     fdps_calc_force_all_and_write_back(tree_num_hydro,
285                                        calc_hydro_force,
286                                        NULL,
287                                        psys_num,
288                                        dinfo_num,
289                                        true,
290                                        FDPS_MAKE_LIST);
291     // Get timestep
292     double dt = get_timestep(psys_num);
293
```

```
294     // Main loop for time integration
295     int nstep = 0; double time = 0.0;
296     for (;;) {
297         // Leap frog: Initial Kick & Full Drift
298         initial_kick(psys_num,dt);
299         full_drift(psys_num,dt);
300 
301         // Adjust the positions of the SPH particles that run over
302         // the computational boundaries.
303         fdps_adjust_pos_into_root_domain(psys_num,dinfo_num);
304 
305         // Leap frog: Predict
306         predict(psys_num,dt);
307 
308         // Perform domain decomposition and exchange particles again
309         fdps_decompose_domain_all(dinfo_num,psys_num,-1.0);
310         fdps_exchange_particle(psys_num,dinfo_num);
311 
312         // Compute density, pressure, acceleration due to pressure
                gradient
313         fdps_calc_force_all_and_write_back(tree_num_dens,
314                                            calc_density,
315                                            NULL,
316                                            psys_num,
317                                            dinfo_num,
318                                            true,
319                                            FDPS_MAKE_LIST);
320         set_pressure(psys_num);
321         fdps_calc_force_all_and_write_back(tree_num_hydro,
322                                            calc_hydro_force,
323                                            NULL,
324                                            psys_num,
325                                            dinfo_num,
326                                            true,
327                                            FDPS_MAKE_LIST);
328 
329         // Get a new timestep
330         dt = get_timestep(psys_num);
331 
332         // Leap frog: Final Kick
333         final_kick(psys_num,dt);
334 
335         // Output result files
336         int output_interval = 10;
337         if (nstep % output_interval == 0) {
338            output(psys_num,nstep);
339            check_cnsrvd_vars(psys_num);
340         }
341 
342         // Output information to STDOUT
343         if (fdps_get_rank() == 0) {
344            printf("================================\n");
345            printf("time␣␣=␣%15.7e\n",time);
346            printf("nstep␣=␣%d\n",nstep);
347            printf("================================\n");
```

```
348         }
349
350         // Termination condition
351         if (time >= end_time) break;
352
353         // Update time & step
354         time  += dt;
355         nstep++;
356     }
357     fdps_finalize();
358 }
```

# 6 Extensions

## 6.1 $P^3M$ code

In this section, we explain the usage of a FDPS extension "Particle Mesh" (hereafter PM) using a sample program for $P^3M$(Particle-Particle-Particle-Mesh) method. The sample code calculates the crystal energy of sodium chloride (NaCl) crystal using the $P^3M$ method and compares the result with the analytical solution. In the $P^3M$ method, the calculation of force and potential energy is performed by splitting into Particle-Particle(PP) part and Particle-Mesh(PM) part. In this sample code, the PP part is calculated by using FDPS standard features and the PM part is computed by using a FDPS extension "PM". Note that the detail of the extension "PM" is described in § 9.2 of the specification of FDPS and please see it for detail.

### 6.1.1 Location of sample code and working directory

The sample code is placed at `$(FDPS)/sample/c/p3m`. Change the current directory to there.

```
$ cd (FDPS)/sample/c/p3m
```

The sample code consists of `user_defined.h` where user-defined types are implemented, `user_defined.c` where interaction functions are implemented, `c_main.c` where the other parts of the user code are implemented, and a Makefile for GCC, `Makefile`.

### 6.1.2 User-defined types

In this section, we describe structures that you need to define in order to perform $P^3M$ calculation using FDPS.

#### 6.1.2.1 FullParticle type

You must define a FullParticle type. Listing 38 shows the implementation of FullParticle type in the sample code. FullParticle type must have all physical quantities required to perform a calculation with $P^3M$ method.

Listing 38: FullParticle type

```
1  typedef struct fp_nbody { //$fdps FP
2     //$fdps copyFromForce force_pp (pot,pot) (acc,acc)
3     //$fdps copyFromForcePM acc_pm
4     long long int id;
5     double mass; //$fdps charge
6     double rcut; //$fdps rsearch
7     fdps_f64vec pos; //$fdps position
8     fdps_f64vec acc;
9     double pot;
10    fdps_f32vec acc_pm;
11    float pot_pm;
12 } FP_nbody;
```

At first, users must specify which user-defined type this structure corresponds to. The following directive specify that this structure is a FullParticle type:

```
typedef struct fp_nbody { //$fdps FP
```

In this P$^3$M code, the interaction force is long-range force with cutoff. Therefore, a cutoff radius is also necessary physical quantity in addition to the position and mass (charge). In the current version of FDPS, designation of cutoff radius is done by the same directive used for search radius (see § 4.2). We can tell FDPS which member variables represent these necessary quantities in the following way:

```
double mass; //$fdps charge
double rcut; //$fdps rsearch
fdps_f64vec pos; //$fdps position
```

FullParticle type copies data from a Force type. Users must specify how the data is copied by using of directives. Also, when using the FDPS extension "PM" to calculate interaction, users must specify how a FullParticle type receives the result of interaction calculation from a "PM" module. In this sample code, there directives are written as follows.

```
//$fdps copyFromForce force_pp (pot,pot) (acc,acc)
//$fdps copyFromForcePM acc_pm
```

#### 6.1.2.2 EssentialParticleI type

You must define a EssentialParticleI type. EssentialParticleI type must have member variables that store all physical quantities necessary for an $i$ particle to perform the PP part of the Force calculation. In the sample code, it is also used as EssentialParticleJ type. Therefore, it should have member variables that store all physical quantities necessary for a $j$ particle to perform the PP part of the Force calculation. Listing 39 shows the implementation of EssentialParticleI type in the sample code.

Listing 39: EssentialParticleI 型

```
1 typedef struct ep_nbody { //$fdps EPI,EPJ
2    //$fdps copyFromFP fp_nbody (id,id) (mass,mass) (rcut,rcut) (pos,pos)
3    long long int id;
4    double mass; //$fdps charge
5    double rcut; //$fdps rsearch
6    fdps_f64vec pos; //$fdps position
7 } EP_nbody;
```

At first, users must tell FDPS this structure corresponds to EssentialParticleI and EssentialParticleJ types using a directive. This is done as follows.

```
typedef struct ep_nbody { //$fdps EPI,EPJ
```

Next, users must specify which member variable corresponds to which necessary quantity using a directive. As described in the explanation of FullParticle type, cutoff radius is also

necessary quantity. Therefore, the following directives are written in this sample code.

```
double mass; //$fdps charge
double rcut; //$fdps rsearch
fdps_f64vec pos; //$fdps position
```

Both EssentialParticleI and EssentialParticleJ types copy data from a FullParticle type. Users must specify how data copy is performed by using of directives. In this sample code, the directives are written as follows.

```
//$fdps copyFromFP fp_nbody (id,id) (mass,mass) (rcut,rcut) (pos,pos)
```

#### 6.1.2.3 Force type

You must define a Force type. Force type must have member variables that store the results of the PP part of the Force calculation. Listing 40 shows the implementation of Force type in this sample code. Because we consider Coulomb interaction only, one Force type is defined.

Listing 40: Force 型

```
1 typedef struct force_pp { //$fdps Force
2    //$fdps clear
3    double pot;
4    fdps_f64vec acc;
5 } Force_pp;
```

At first, users must specify this structure is a Force type using a directive. In this sample code, it is written as.

```
typedef struct force_pp { //$fdps Force
```

Because this structure is a Force type, users **must** specify how member variables are initialized before interaction calculation via directives. In this sample code, we adopt the default initialization for all of the member variables. This is realized by writing a FDPS directive with `clear` keyword only:

```
//$fdps clear
```

#### 6.1.2.4 calcForceEpEp

You must define an interaction function calcForceEpEp. calcForceEpEp must contain actual code for the PP part of the Force calculation and must be implemented as `void` function . Its arguments is an array of EssentialParticleI objects, the number of EssentialParticleI objects, an array of EssentialParticleJ objects, the number of EssentialParticleJ objects, and an array of Force objects. Listing 41 shows the implementation of calcForceEpEp in this sample code.

Listing 41: Interaction function calcForceEpEp

```
1  void calc_force_ep_ep(EP_nbody *ep_i,
2                        int n_ip,
3                        EP_nbody *ep_j,
4                        int n_jp,
5                        Force_pp *f) {
6      int i,j;
7      for (i = 0; i < n_ip; i++) {
8          for (j = 0; j < n_jp; j++) {
9              fdps_f64vec dr;
10             dr.x = ep_i[i].pos.x - ep_j[j].pos.x;
11             dr.y = ep_i[i].pos.y - ep_j[j].pos.y;
12             dr.z = ep_i[i].pos.z - ep_j[j].pos.z;
13             double rij  = sqrt(dr.x * dr.x
14                                +dr.y * dr.y
15                                +dr.z * dr.z);
16             if ((ep_i[i].id == ep_j[j].id) && (rij == 0.0)) continue;
17             double rinv = 1.0/rij;
18             double rinv3 = rinv*rinv*rinv;
19             double xi = 2.0*rij/ep_i[i].rcut;
20             f[i].pot   += ep_j[j].mass * S2_pcut(xi) * rinv;
21             f[i].acc.x += ep_j[j].mass * S2_fcut(xi) * rinv3 * dr.x;
22             f[i].acc.y += ep_j[j].mass * S2_fcut(xi) * rinv3 * dr.y;
23             f[i].acc.z += ep_j[j].mass * S2_fcut(xi) * rinv3 * dr.z;
24         }
25         // Self-interaction term
26         f[i].pot -= ep_i[i].mass * (208.0/(70.0*ep_i[i].rcut));
27     }
28 }
```

The PP part in the P$^3$M method is a two-body interaction with cutoff (i.e. the interaction is truncated if the distance between the particles is larger than the cutoff distance). Hence, cutoff functions (`S2_pcut()`, `S2_fcut()`) appears in the calculations of potential and acceleration. These cutoff functions must be the ones that are constructed assuming that the particle shape function is $S2(r)$, which is introduced by Hockney & Eastwood (1988)(Eq.(8.3)) and takes the form of

$$S2(r) = \begin{cases} \dfrac{48}{\pi a^4}\left(\dfrac{a}{2} - r\right) & r < a/2, \\ 0 & \text{otherwise.} \end{cases} \tag{1}$$

where $r$ is the distance from the center of a particle, $a$ is the scale length of the shape function. When assuming this shape function, the charge density distribution due to a particle, $\rho(r)$, is expressed as $\rho(r) = q\,S2(r)$, where $q$ is the charge of the particle. Thus, $S2(r)$ shape function gives linear density distribution. The reason why we have to use the cutoff functions that correspond to $S2(r)$ shape function is that the cutoff functions used in the PM part also assumes the $S2(r)$ shape function (the cutoff functions in the PM and PP parts should be consistent with each other).

The cutoff functions must be defined by a user. Possible implementations for `S2_pcut()` and `S2_fcut()` are given at the beginning of the sample code (see the lines 22-72 in `main.cpp`). In these examples, we used Eqs.(8-72) and (8-75) in Hockney & Eastwood

(1988) and we define them such that the PP interaction takes of the form:

$$\Phi_{\rm PP}(\boldsymbol{r}) = \frac{m}{|\boldsymbol{r}-\boldsymbol{r}'|}\texttt{S2\_pcut}(\xi) \tag{2}$$

$$\boldsymbol{f}_{\rm PP}(\boldsymbol{r}) = \frac{m(\boldsymbol{r}-\boldsymbol{r}')}{|\boldsymbol{r}-\boldsymbol{r}'|^3}\texttt{S2\_fcut}(\xi) \tag{3}$$

where $\xi = 2|\boldsymbol{r}-\boldsymbol{r}'|/a$. In this sample code, $a$ is expressed as a variable `rcut` .

As is clear from Eq.(8-75) in Hockney & Eastwood (1988), the mesh potential $\phi^m$ has a finite value at $r = 0$ (we omit a factor $1/4\pi\varepsilon_0$ here):

$$\phi^m(0) = \frac{208}{70a} \tag{4}$$

This term is taken into account the last line in the $i$-particle loop:

```
1 f[i].pot -= ep_i[i].mass * (208.0/(70.0*ep_i[i].rcut));
```

Note that this term is necessary to match the numerical result with the analytical solution.

#### 6.1.2.5 calcForceEpSp

You must define an interaction function calcForceEpSp[2]. calcForceEpSp must contain actual code for particle-superparticle interaction and must be implemented as `void` function . Its arguments is an array of EssentialParticleI objects, the number of EssentialParticleI objects, an array of SuperParticleJ objects, the number of SuperParticleJ objects, and an array of Force objects. Listing 42 shows the implementation of calcForceEpSp in the sample code.

Listing 42: Interaction function calcForceEpSp

```
1  void calc_force_ep_sp(EP_nbody *ep_i,
2                        int n_ip,
3                        fdps_spj_monopole_cutoff *ep_j,
4                        int n_jp,
5                        Force_pp *f) {
6     int i,j;
7     for (i = 0; i < n_ip; i++) {
8        for (j = 0; j < n_jp; j++) {
9           fdps_f64vec dr;
10          dr.x = ep_i[i].pos.x - ep_j[j].pos.x;
11          dr.y = ep_i[i].pos.y - ep_j[j].pos.y;
12          dr.z = ep_i[i].pos.z - ep_j[j].pos.z;
13          double rij = sqrt(dr.x * dr.x
14                            +dr.y * dr.y
15                            +dr.z * dr.z);
16          double rinv = 1.0/rij;
17          double rinv3 = rinv*rinv*rinv;
```

[2] As describe at the beginning of this section, the sample code uses P$^3$M for the calculation of interaction. In order to realize it using FDPS, we perform the calculation of interaction with the opening angle criterion $\theta$ of 0. Hence, particle-superparticle interaction should not occur. However, API `fdps_calc_force_all_and_write_back` requires a function pointer of a `void` function that calculates particle-superparticle interaction. Therefore, we defined calcForceEpSp here.

```
18            double xi = 2.0*rij/ep_i[i].rcut;
19            f[i].pot   += ep_j[j].mass * S2_pcut(xi) * rinv;
20            f[i].acc.x += ep_j[j].mass * S2_fcut(xi) * rinv3 * dr.x;
21            f[i].acc.y += ep_j[j].mass * S2_fcut(xi) * rinv3 * dr.y;
22            f[i].acc.z += ep_j[j].mass * S2_fcut(xi) * rinv3 * dr.z;
23        }
24    }
25 }
```

### 6.1.3 Main body of the sample code

In this section, we explain the main body of the sample code. Before going into details, we first give a simple explanation about the content and the structure of the sample code. As described in § 6.1, this code computes the crystal energy of NaCl crystal using the $P^3M$ method and compares the result with the analytical solution. The NaCl crystal is expressed as an uniform grid of particles in this sample code. Na and Cl are placed in the staggered layout. Particles corresponding to Na has a positive charge, while those corresponding to Cl has a negative charge. We place a crystal expressed as an grid of charged particles into a periodic computational box of the sizes $[0, 1)^3$ and calculates the crystal energy. The computational accuracy of the crystal energy should depend on the number of particles and the configuration of particles (to the grid used in the PM calculation). Hence, in the sample code, we measure the relative energy errors for a different set of these parameters and output the result of the comparisons into a file.

The structure of the sample code is as follows:

(1) Create and initialize FDPS objects
(2) Create a NaCl crystal for given number of particles and configuration (in `void` function `setup_NaCl_crystal()` )
(3) Compute the potential energy of each particle by the $P^3M$ method ( In `c_main()` )
(4) Compute the total energy of the crystal and compare it with the analytical solution (`void` function `calc_energy_error()`)
(5) Repeat (2)-(4)

In the following, we explain in detail each steps described above.

#### 6.1.3.1 Including the header file of FDPS C interface

To make the standard features of FDPS available, we must include header file `FDPS_c_if.h`.

Listing 43: Including header file `FDPS_c_if.h`

```
1 #include "FDPS_c_if.h"
```

#### 6.1.3.2 Initialization and Termination of FDPS

First, you must initialize FDPS by the following code.

Listing 44: Initialization of FDPS

```
1 fdps_initialize();
```

Once started, FDPS should be terminated explicitly. In this sample, FDPS is terminated just before the termination of the program. Hence, you need to write the following code at the end of the main function.

Listing 45: Termination of FDPS

```
1 fdps_finalize();
```

#### 6.1.3.3 Creation and initialization of FDPS objects

After the initialization of FDPS, a user need to create the objects used to talk to FDPS. In this section, we describe how to create and initialize these objects.

##### *6.1.3.3.1 Creation of necessary FDPS objects*

In the calculation using the $P^3M$ method, we must create ParticleSystem and DomainInfo objects. In addition, Tree and ParticleMesh objects are also needed to calculate the PP and PM parts of the force calculation.

Listing 46: Creation of FDPS objects

```
1 fdps_create_psys(&psys_num,"fp_nbody");
2 fdps_create_dinfo(&dinfo_num);
3 fdps_create_pm(&pm_num);
4 fdps_create_tree(&tree_num,
5                   "Long,force_pp,ep_nbody,ep_nbody,MonopoleWithCutoff");
```

Note that the code snippet shown above differs from the actual sample code.

##### *6.1.3.3.2 Initialization of FDPS objects*

After the creation of FDPS objects, you must initialize these objects before you use them in a user code. In the following, we explain how to initialize each object.

*(i) Initialization of a ParticleSystem object* A ParticleSystem object is initialized as follows:

Listing 47: Initialization of a ParticleSystem object

```
1 fdps_init_psys(psys_num);
```

This is done in the main function in the sample code.

*(ii) Initialization of a DomainInfo object* A DomainInfo object is initialized as follows:

Listing 48: Initialization of a DomainInfo object

```
1 fdps_init_dinfo(dinfo_num,coef_ema);
```

This is done in the main function in the sample code.

After the initialization, you need to specify the boundary condition and the size of the simulation box through APIs `fdps_set_boundary_condition` and `fdps_set_pos_root_domain` . In the sample code, these procedures are performed in `void` function `setup_NaCl_crystal` that sets up the distribution of particles:

```
1 fdps_set_boundary_condition(dinfo_num,FDPS_BC_PERIODIC_XYZ);
2 fdps_f32vec pos_ll, pos_ul;
3 pos_ll.x = 0.0; pos_ll.y = 0.0; pos_ll.z = 0.0;
4 pos_ul.x = 1.0; pos_ul.y = 1.0; pos_ul.z = 1.0;
5 fdps_set_pos_root_domain(dinfo_num,&pos_ll,&pos_ul);
```

*(iii) Initialization of a Tree object* A Tree object is initialized by API `fdps_init_tree` :

Listing 49: Initialization of a Tree object

```
1 fdps_init_tree(tree_num,3*nptcl_loc,theta,
2                n_leaf_limit,n_group_limit);
```

You need to give a rough number of particles to this API as the second argument. Here, we set three times the number of local particles at the time of calling. The third argument of this API represents the opening angle criterion $\theta$ for the tree method. In the sample, we do not use the tree method in the PP part of the force calculation. Therefore, we set $\theta = 0$.

*(iv) Initialization of a ParticleMesh object* No explicit initialization is needed.

#### 6.1.3.4 Generation of a distribution of particles

In this section, we explain `void` function `setup_NaCl_crystal` that generates a distribution of particles, and FDPS APIs called within it. Given the number of particles per one space dimension and the position of the particle that is nearest to the origin $(0, 0, 0)$, `void` function `setup_NaCl_crystal` makes a three-dimensional uniform grid of particles. These parameters are specified through an object of structure `crystal_parameters` , `NaCl_params`:

```
1 // In user_defined.h
2 typedef struct crystal_parameters {
3    int nptcl_per_side;
4    fdps_f64vec pos_vertex;
5 } Crystal_parameters;
6 // In c_main.c
7 Crystal_parameters NaCl_params;
8 setup_NaCl_crystal(psys_num, dinfo_num, NaCl_params);
```

In the first half of `void` function `setup_NaCl_crystal` , it makes an uniform grid of particles based on the value of `NaCl_params`. In this process, we scale the particle charge $m$ to satisfy the relation

$$\frac{2Nm^2}{R_0} = 1, \tag{5}$$

where $N$ is the total number of molecules (the total number of atomic particles is $2N$) and $R_0$ is the distance to the nearest particle. This scaling is introduced just for convenience:

The crystal energy can be written analytically as

$$E = -\frac{N\alpha m^2}{R_0}, \tag{6}$$

where $\alpha$ is the Madelung constant and $\alpha \approx 1.747565$ for the NaCl crystal (e.g. see Kittel (2004) "Introduction to Solid State Physics"). Thus, the crystal energy depends on the total number of particles. This is inconvenient when comparing the calculation result with the analytical solution. By scaling the particle charge as described above, the crystal energy becomes independent from $N$.

After generating a particle distribution, this function performs domain decomposition and particle exchange using FDPS APIs. In the following, we explain these APIs.

#### *6.1.3.4.1 Domain Decomposition*

API `fdps_decompose_domain_all` of the DomainInfo object is used to perform domain decomposition based on the current distribution of particles:

Listing 50: Domain Decomposition

```
1 fdps_decompose_domain_all(dinfo_num,psys_num);
```

#### *6.1.3.4.2 Particle Exchange*

API `fdps_exchange_particle` of the ParticleSystem object is used to exchange particles based on the current decomposed domains:

Listing 51: Particle Exchange

```
1 fdps_exchange_particle(psys_num,dinfo_num);
```

### 6.1.3.5 Interaction Calculation

After these procedures are completed, we must perform the interaction calculation. In the sample code, it is performed in the main function.

Listing 52: Interaction calculation

```
1  // [4] Compute force and potential with P^{3}M method
2  // [4-1] Get the pointer to FP and # of local particles
3  int nptcl_loc = fdps_get_nptcl_loc(psys_num);
4  FP_nbody *ptcl = (FP_nbody *) fdps_get_psys_cptr(psys_num);
5  // [4-2] PP part
6  fdps_calc_force_all_and_write_back(tree_num,
7                                     calc_force_ep_ep,
8                                     calc_force_ep_sp,
9                                     psys_num,
10                                    dinfo_num,
11                                    true,
12                                    MAKE_LIST);
13 // [4-3] PM part
14 fdps_calc_pm_force_all_and_write_back(pm_num,
```

```
15                                          psys_num,
16                                          dinfo_num);
17 int i;
18 for (i = 0; i < nptcl_loc; i++) {
19    fdps_f32vec pos32;
20    pos32.x = ptcl[i].pos.x;
21    pos32.y = ptcl[i].pos.y;
22    pos32.z = ptcl[i].pos.z;
23    fdps_get_pm_potential(pm_num,&pos32,&ptcl[i].pot_pm);
24 }
25 // [4-4] Compute the total acceleration and potential
26 for (i = 0; i < nptcl_loc; i++) {
27    ptcl[i].pot -= ptcl[i].pot_pm;
28    ptcl[i].acc.x -= ptcl[i].acc_pm.x;
29    ptcl[i].acc.y -= ptcl[i].acc_pm.y;
30    ptcl[i].acc.z -= ptcl[i].acc_pm.z;
31 }
```

We use API `fdps_calc_force_all_and_write_back` for the PP part and API `fdps_calc_pm_force_all_and_write_back` for the PM part. After calculating the PM part, the total acceleration and total potential are computed. Please note that this summation is done by **subtraction**. The reason why we use subtraction is that the FDPS extension "PM" computes the potential energy assuming gravity. In other words, the FDPS extension "PM" treats a charge with $m(>0)$ creates negative potential. Hence, ***we need to invert the signs of potential energy and acceleration*** in order to use the FDPS extension "PM" for the Coulomb interaction calculation.

#### 6.1.3.6 Calculation of relative energy error

The relative error of the crystal energy is computed in the function `calc_energy_error()`, where we assume that the analytical solution is $E_0 \equiv 2E = -1.7475645946332$, which is numerically evaluated by the PM$^3$(Particle-Mesh Multipole Method).

### 6.1.4 Compile

Before compiling your program, you need to install the FFTW(Fast Fourier Transform in the West) library. Then, edit the file `Makefile` in the working directory to set the PATHs of the locations of FFTW and FDPS to the variables `FFTW_LOC` and `FDPS_LOC`. After that, run `make`.

```
$ make
```

The execution file `p3m.x` will be created in the directory `work` if the compilation is succeeded.

### 6.1.5 Run

You must run your program using MPI with the number of MPI processes is equal to or greater than 2, because of the specification of FDPS extensions. Therefore, you should run the following command:

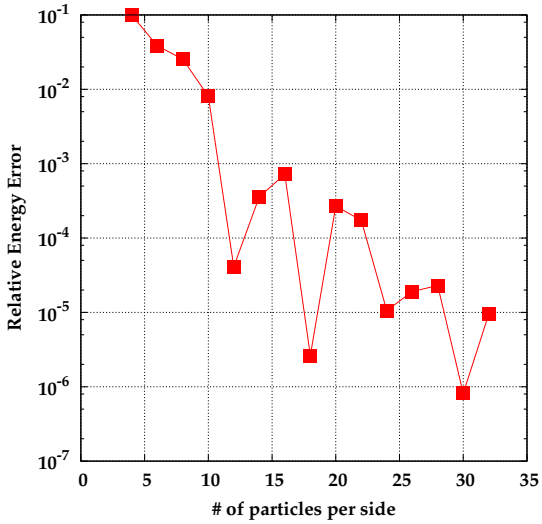


Figure 3: The relative error of the crystal energy as a function of the number of particles per side, where we assume that the number of the PM grids is $16^3$ and the cutoff radius is 3/16.

```
$ MPIRUN -np NPROC ./p3m.x
```

where "MPIRUN" represents the command to run your program using MPI such as `mpirun` or `mpiexec`, and "NPROC" is the number of MPI processes.

### 6.1.6 Check the result

After the program ended, a file that records the relative error of the crystal energy is output in the directory `work`. Figure 3 shows the dependency of the relative error on the number of particles used.

# 7 Practical Applications

In previous sections, we have explained fundamental features of FDPS using relatively simple application codes. However, we need to develop a more complex application in actual research, in which for example we need to treat different types of particles. In this section, we will explain advanced features of FDPS using practical applications. To keep the explanations short and simple, we require the readers understand the contents of the previous sections in this document.

## 7.1 $N$-body/SPH code

In this section, we explain the accompanying sample code for $N$-body/SPH simulation of a disk galaxy. In this code, dark matter and stars, which perform gravitational interaction only, are represented by $N$-body particles, while interstellar gas, which performs both gravitational and hydrodynamic interactions, is represented by SPH particles. The tree method is used for the gravity calculation. The SPH scheme adopted in this code is the one proposed by Springel & Hernquist [2002, MNRAS, 333, 649] and Springel [2005, MNRAS, 364, 1105] (hereafter, we call it Springel's SPH scheme). The readers can understand how to treat different types of particles using FDPS by reading this section.

Below, we first explain the usage of the code. Next, we give a brief explanation of the Springel's SPH scheme. Then, we explain the contents of the sample source codes in detail.

### 7.1.1 How to run the sample code

As we described, this code simulates the dynamical evolution of a disk galaxy. This code sets the initial distributions of dark matter and stars by reading a file created by MAGI (Miki & Umemura [2018, MNRAS, 475, 2269]), which is a software to make an initial condition of a galaxy simulation. On the other hand, the initial gas distribution is set inside the code. Therefore, the following procedures are required to use the code.

- Move to directory `$(FDPS)/sample/c/nbody+sph`
- Edit `Makefile` in the current directory
- Create particle data using MAGI and place it under directory `./magi_data/dat`
- Run the `make` command to create the executable `nbodysph.out`
- Run `nbodysph.out`
- Check the output

Below, we explain each procedure.

#### 7.1.1.1 Move to the directory the sample code is placed

Move to `$(FDPS)/sample/c/nbody+sph`.

#### 7.1.1.2 File structure of the sample code

The following is the file structure of the sample code.

```
$ ls | awk '{print $0}'
Makefile
Makefile.ofp
c_main.c
ic.c
ic.h
job.ofp.sh
leapfrog.c
leapfrog.h
macro_defs.h
magi_data/
mathematical_constants.c
mathematical_constants.h
physical_constants.c
physical_constants.h
tipsy_file_reader.cpp
tipsy_file_reader.h
user_defined.c
user_defined.h
```

We explain briefly the content of each source file. In `ic.*`, functions to create initial conditions are implemented. Users can choose an initial condition other than that for a disk galaxy (described later). In `leapfrog.*`, we implement functions necessary to integrate the orbits of particles based on the Leapfrog method. In `macro_defs.h`, we define macros that are used to control numerical simulation. In `c_main.c`, the main function is implemented. In `mathematical_constants.*`, we define some mathematical constants. In `physical_constants.*`, we define some physical constants. In `tipsy_file_reader.+`, we define functions to read particle data created by MAGI. In `user_defined.*`, we define user-defined types and interaction functions.

Directory `magi_data` stores a parameter file input to the software MAGI (`magi_data/cfg/*`) and a script file used to run MAGI (`magi_data/sh/run.sh`).

#### 7.1.1.3 Edit Makefile

Edit `Makefile` following the description below.

- Set the variable `CXX` the command to run your C++ compiler.
- Set the variable `CC` the command to run your C compiler.
- Set the variable `CXXFLAGS` compile options of the C++ compiler.
- Set the variable `CFLAGS` compile options of the C compiler.

- In this code, several macros are used to control numerical simulations. Table 1 lists the names of the macros and their definitions. In addition, there are macros whose states (i.e. value or defined/undefined states) are automatically set according to the value of macro `INITIAL_CONDITION`. Generally, users do not have to change them. Please see `macro_defs.h` directly for detail.
- Phantom-GRAPE library for x86 can be used for the gravity calculation. To use it, set the variable `use_phantom_grape_x86 yes`.

As for the way to specify the use/non-use of OpenMP and MPI, see § 3.

| Macro name | Defintion |
|---|---|
| INITIAL_CONDITION | It specifies the type of initial condition or the operation mode of the code. It must take a value from 0 to 3. According to its value, the code operates as follows. 0: an initial condition for a disk galaxy is used, 1: an initial condition for cold collapse test problem is used, 2: an initial condition for Evrard test is used, 3: the code operates in the mode to make a glass-like distribution of SPH particles. |
| ENABLE_VARIABLE_SMOOTHING_LENGTH | It specifies that smoothing length of SPH particles is variable or not. If it is defined, variable smoothing length is used and the SPH calculation is performed according to the Springel's SPH scheme. If it is not defined, the fixed smoothing length is used and the SPH calculation is done in almost the same way as the sample code described in § 3-4. |
| USE_ENTROPY | It specifies whether to use entropy or specific internal energy as an independent variable to describe the thermodynamic state of SPH particle. If defined, entropy is used. But, if macro ISOTHERMAL_EOS described below is defined, specific internal energy is forcibly used (specific internal energy is used to calculate pressure). |
| USE_BALSARA_SWITCH | It specifies whether Balsara switch (Balsara [1995, JCP, 121, 357]) is used or not. If defined, the Balsara switch is used. |
| USE_PRESCR_OF_THOMAS_COUCHMAN_1992 | It specifies whether a simple prescription proposed by Thomas & Couchman [1992, MNRAS,257, 11] to prevent the tensile instability is used or not. If defined, this prescription is used. |
| ISOTHERMAL_EOS | It specifies whether isothermal process is assumed or not. If defined, isothermal process is assumed (specific internal energy is assumed to be constant). If not defined, the code solve the entropy equation or the internal energy equation. |
| READ_DATA_WITH_BYTESWAP | It specifies whether the program reads particle data with performing byte swap (byte swap is applied for each variable of basic data type). If defined, byte swap is performed. |



Table 1: Compile-time macros and their definitions

#### 7.1.1.4 Create particle data using MAGI

As described earlier, users need to create particle data using the software MAGI before simulation according to the procedures described below. For users who cannot use `MAGI` for some reasons, we prepared sample particle data in web sites described below. In the following, we explain each case in detail.

**Create particle data using MAGI** Create particle data as follows.

1. Download the source file of MAGI from the web side https://bitbucket.org/ymiki/magi and install it in appropriate PATH according to the descriptions in Section "How to compile MAGI" in the above web side. But, our $N$-body/SPH sample code supports TIPSY file format only. Therefore, please build MAGI with `USE_TIPSY_FORMAT=ON`.
2. Edit `./magi_data/sh/run.sh` and set the variable `MAGI_INSTALL_DIR` the PATH of the directory where the `magi` command is stored. Also, set the variable `NTOT` the number of $N$-body particles (MAGI automatically assigns the numbers of dark matter particles and star particles).
3. Edit `./magi_data/cfg/*` to specify a galaxy model. For detail of the format of input file for MAGI, please see the web side above or Section 2.4 in the original paper Miki & Umemura [2018, MNRAS, 475, 2269]. In the default, galaxy model consists of the following four components (hereafter, we call this **default galaxy model**):
   (i) Dark matter halo (NFW profile, $M = 10^{12}$ $M_\odot$, $r_s = 21.5$ kpc, $r_c = 200$ kpc, $\Delta_c = 10$ kpc)
   (ii) Stellar bulge (King model, $M = 5 \times 10^{10}$ $M_\odot$, $r_s = 0.7$ kpc, $W_0 = 5$)
   (iii) Thick stellar disk (Sérsic profile, $M = 2.5 \times 10^{10}$ $M_\odot$, $r_s = 3.5$ kpc, $n = 1.5$, $z_d = 1$ kpc, $Q_{T,\mathrm{min}} = 1.0$)
   (iv) Thin stellar disk (exponential disk, $M = 2.5 \times 10^{10}$ $M_\odot$, $r_s = 3.5$ kpc, $z_d = 0.5$ kpc, $Q_{T,\mathrm{min}} = 1.0$)

   In the default galaxy model, two stellar disks are marginally unstable to a bar-mode in view of the Ostriker-Peebles criterion. Therefore, a simulated galaxy is expected to evolve into a spiral galaxy having a weak bar. In the latest release of MAGI (version 1.1.1 [as of July 19th, 2019]), its default operation mode is changed from previous releases. With this demand, we have replaced parameter $f$ in thick and thin disks by $Q_{T,\mathrm{min}}$, where $f$ is a parameter controlling the velocity dispersion of disk and is used in the previous releases of MAGI to specify the stability of a disk component. $Q_{T,\mathrm{min}}$ is the minimum of Toomre Q value in the disk. (In the sample code in FDPS 5.0d or earlier, we used $f = 0.125$).
4. Move to directory `magi_data` and run the following command:

```
$ ./sh/run.sh
```

5. If MAGI stops successfully, particle data whose extension is `tipsy` will be created in directory `magi_data/dat`.

**Download sample particle data form our web sites** Download a particle data file from one of the following URLs and place it under directory `./magi_data/dat/`. All of particle data is made with the default galaxy model. Only the number of particles is different for each data.

- $N = 2^{21}$: http://particle.riken.jp/~fdps/magi_data/Galaxy/21/Galaxy.tipsy
- $N = 2^{22}$: http://particle.riken.jp/~fdps/magi_data/Galaxy/22/Galaxy.tipsy
- $N = 2^{23}$: http://particle.riken.jp/~fdps/magi_data/Galaxy/23/Galaxy.tipsy
- $N = 2^{24}$: http://particle.riken.jp/~fdps/magi_data/Galaxy/24/Galaxy.tipsy

#### 7.1.1.5 Run make

Type "make" to run the `make` command.

#### 7.1.1.6 Run the sample code

- If you are not using MPI, run the following in CLI (terminal)

```
$ ./nbodysph.out
```

- If you are using MPI, run the following in CLI (terminal)

```
$ MPIRUN -np NPROC ./nbodysph.out
```

where `MPIRUN` should be `mpirun` or `mpiexec` depending on your MPI configuration, and `NPROC` is the number of processes you will use.

#### 7.1.1.7 Analysis of the result

In the directory `result`, data of $N$-body and SPH particles are output as files "nbody0000x-proc0000y.dat" and "sph0000x-proc0000y.dat", where x is an integer representing time and y is an integer representing a process number (MPI rank number). The output file format of $N$-body particle data is that in each line, index of particle, mass, position (x, y, z), velocity (vx, vy, vz) are listed. The output file format of SPH particle data is that in each line, index of particle, mass, positon (x, y, z), velocity (vx, vy, vz), density, specific internal energy, entropy, pressure are listed.

Figure 4 shows the distribution of star and SPH particles at $T = 0.46$ for a disk galaxy simulation with the number of $N$-body particles is $2^{21}$ and the number of SPH particles is $2^{18}$.

Below, we briefly explain the Springel's SPH scheme and then explain the implementation of the sample code.

### 7.1.2 Springel's SPH scheme

Springel & Hernquist [2002, MNRAS, 333, 649] proposed a formulation of SPH (actually,

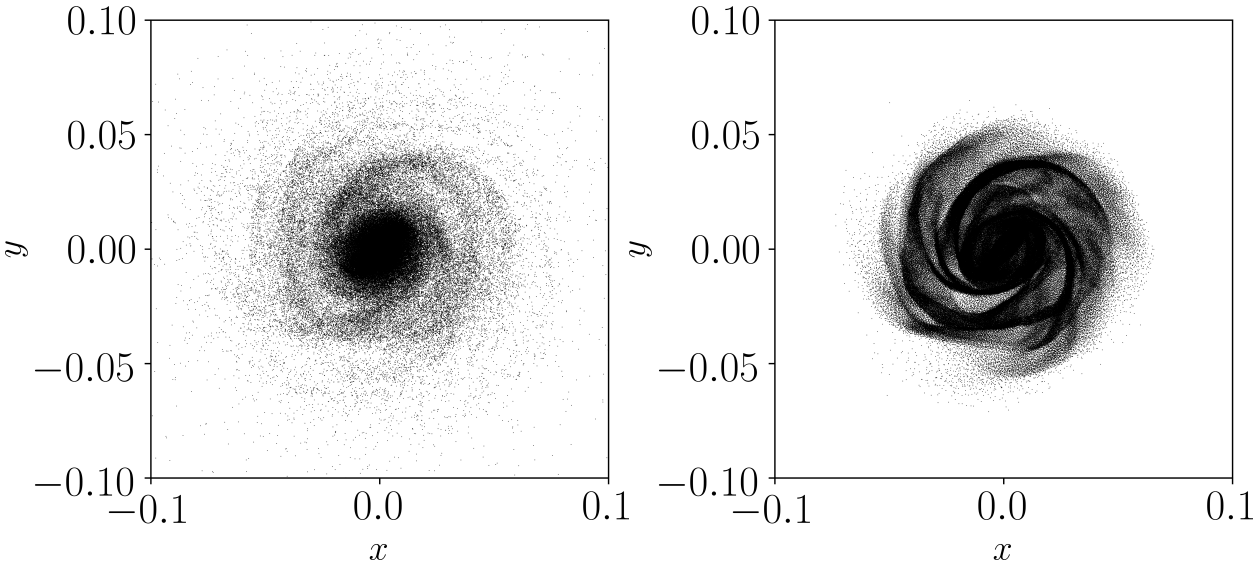


Figure 4: Face-on view of distributions of stars (left) and gas (right) (simulation configuration: the simulation is performed the number of $N$-body particles is $2^{21}$, the number of SPH particles is $2^{18}$, isothermal, gas temperature is $10^4$ K, mean molecular weight to the mass of hydrogen $\mu = 0.5$)

equation of motion[EoM]) where the total energy and entropy of a system are conserved even if smoothing length changes with time. In this section, we briefly explain their formulation. The outline of the derivation is as follows. Construct a Lagrangian of the system assuming that smoothing length is also independent variable, then solve the Euler-Lagrange equations under $N$ constraints, where $N$ is the number of particles.

More specifically, they consider the Lagrangian

$$L(\boldsymbol{q}, \dot{\boldsymbol{q}}) = \frac{1}{2}\sum_{i=1}^{N} m_i \dot{\boldsymbol{r}}_i^2 - \frac{1}{\gamma - 1}\sum_{i=1}^{N} m_i A_i \rho_i^{\gamma-1} \tag{7}$$

where $\boldsymbol{q} = (\boldsymbol{r}_1, ..., \boldsymbol{r}_N, h_1, ...h_N)$ is the generalized coordinates (the subscripts represent the indice of particles), $\boldsymbol{r}_i$ is the position, $h_i$ is smoothing length, $m_i$ is mass, $\gamma$ is the ratio of specific heats, $\rho_i$ is density, $A_i$ is called entropy function and it is related with specific internal energy $u_i$ and $\rho_i$ through the equation

$$u_i = \frac{A_i}{\gamma - 1}\rho_i^{\gamma-1} \tag{8}$$

The first and second terms of Eq.(7) represents the kinetic energy and the internal energy of the system, respectively. Because solving the Euler-Lagrangian equation directly using this Lagrangian results in $4N$ equations, which is not undesirable, they introduce the following $N$ constraints.

$$\phi_i = \frac{4\pi}{3} h_i^3 \rho_i - \overline{m} N_{\text{neigh}} = 0 \tag{9}$$

where $\overline{m}$ is the average mass of SPH particles[3)], $N_{\text{neigh}}$ is the number of neighbor particles (constant). Under these constraints, using the method of Lagrange multiplier, they solve

[3)] This must be treated as constant.

the Euler-Lagrange equations to obtain the following equations of motion:

$$\frac{\mathrm{d}\boldsymbol{v}_i}{\mathrm{d}t} = -\sum_{j=1}^{N} m_j \left[ f_i \frac{P_i}{\rho_i^2} \nabla_i W(r_{ij}, h_i) + f_j \frac{P_j}{\rho_j^2} \nabla_i W(r_{ij}, h_j) \right] \tag{10}$$

where $P_i$ is pressure, $r_{ij} = |\boldsymbol{r}_i - \boldsymbol{r}_j|$, $W$ is the kernel function, $f_i$ is the so-called $\nabla h$ term, defined by

$$f_i = \left( 1 + \frac{h_i}{3\rho_i} \frac{\partial \rho_i}{\partial h_i} \right)^{-1} \tag{11}$$

The thermodynamic state of the system is described by the independent variable $A_i$, the entropy. If the flow is adiabatic, the entropy is constant along the flow except for locations of shock waves where the entropy is increased. Springel [2005, MNRAS, 364, 1105] modeled the increase of the entropy by passing shock waves using the method of artificial viscosity:

$$\frac{\mathrm{d}A_i}{\mathrm{d}t} = \frac{1}{2} \frac{\gamma - 1}{\rho_i^{\gamma - 1}} \sum_{j=1}^{N} m_j \Pi_{ij} \boldsymbol{v}_{ij} \cdot \nabla_i \overline{W}_{ij} \tag{12}$$

$$\left. \frac{\mathrm{d}\boldsymbol{v}_i}{\mathrm{d}t} \right|_{\mathrm{visc}} = -\sum_{j=1}^{N} m_j \Pi_{ij} \nabla_i \overline{W}_{ij} \tag{13}$$

where $\boldsymbol{v}_{ij} = \boldsymbol{v}_i - \boldsymbol{v}_j$, $\boldsymbol{v}_i$ is velocity, $\overline{W}_{ij} = \frac{1}{2}(W(r_{ij}, h_i) + W(r_{ij}, h_j))$. For $\Pi_{ij}$, please see the original papers.

The procedures of SPH calculation is summarized as follows:

(1) Solve Eq.(9) and the following equation self-consistently to determine the density $\rho_i$ and the smoothing length $h_i$.

$$\rho_i = \sum_{j=1}^{N} m_j W(r_{ij}, h_i) \tag{14}$$

(2) Calculate $\nabla h$ term defined by Eq.(11).
(3) Calculate the right-hand side of Eqs.(10), (12)、(13).
(4) Update the positions, velocities, entropies of SPH particles.

In the remaining sections, we first explain the implementations of user-defined classes and interaction functions. Then, we explain the implementation of the main routine where we explain how to treat different types of particles in FDPS.

### 7.1.3 User-defined types

All user-defined types are defined in `user_defined.h` . Here, we explain the types of user-defined types used in this code. As described earlier, this code use two types of particles, $N$-body and SPH particles. Thus, this code defines **two** FullParticle types ( `fp_nbody` type for $N$-body particles and `fp_sph` type for SPH particles). The number of types of *physical* interactions are two, the gravitational and hydrodynamic interactions. But, as explained

in § 4, we need to perform (at least) two interaction calculations (for density and acceleration) in SPH calculations. Therefore, the code defines **three** Force types ( `force_grav` type for the gravity calculation, `force_dens` type for the density calculation, and `force_hydro` type for the calculation of acceleration due to pressure gradient (hereafter we call it pressure-gradient acceleration for simplicity)). For simplicity, this code uses one structure for both EssentialParticleI type and EssentialParticleJ type (hereafter, we call them together EssentialParticle type). Also this code uses the same EssentialParticle type for the calculations of density and pressure-gradient acceleration. Therefore, the number of types of EssentialParticle types is **two** ( `ep_grav` type for the gravity calculation and `ep_hydro` type for SPH calculation).

Below, we explain the implementation of each user defined type.

#### 7.1.3.1 FullParticle type

First, we explain structure `fp_nbody` , which is used to store the information of $N$-body particles. This data type contains all physical quantities that a $N$-body particle should have as member variables. Listing 53 shows the implementation of `fp_nbody` type . The definitions of the member variables are almost the same as those of $N$-body sample code introduced in § 3-4. Thus, please see the corresponding section for detail.

Listing 53: FullParticle type (`fp_nbody` type)

```
1 typedef struct fp_nbody { //$fdps FP
2    //$fdps copyFromForce force_grav (acc,acc) (pot,pot)
3    long long int id; //$fdps id
4    double mass; //$fdps charge
5    fdps_f64vec pos; //$fdps position
6    fdps_f64vec vel;
7    fdps_f64vec acc;
8    double pot;
9 } FP_nbody;
```

Next, we explain structure `fp_sph` , which is used to store the information of SPH particles. This data type contains all physical quantities that a SPH particle should have as member variables. Listing 54 shows the implementation of `fp_sph` type . The definitions of main member variables are as follows: `id` (identification number), `mass` (mass), `pos` (position$[\boldsymbol{r}_i]$), `vel` (velocity$[\boldsymbol{v}_i]$), `acc_grav` (gravitational acceleration), `pot_grav` (gravitational potential), `acc_hydro` (pressure-gradient acceleration), `dens` (density$[\rho_i]$), `eng` (specific internel energy$[u_i]$), `ent` (entropy function [hereafter, entropy]$[A_i]$), `pres` (pressure$[P_i]$), `smth` (smoothing length[4]$[h_i]$), `gradh` ($\nabla h$ term$[f_i]$), `divv` ($(\nabla \cdot \boldsymbol{v})_i$, where the subscript $i$ means that the derivative is performed at particle position), `rotv` ($(\nabla \times \boldsymbol{v})_i$), `balsw` (coefficient for Balsara switch and its definition is the same as $f(a)$ in Balsara [1995, JCP, 121, 357]), `snds` (sound speed), `eng_dot` (time rate of change of `eng`), `ent_dot` (time rate of change of `ent`), `dt` (the maximum allowable time step to integrate the orbit of this particle).

The following points should be noted.

- SPH particles are involved with three types of interaction calculations (gravity, density,

[4] It is defined as the distance from the center of a particle where the value of the SPH kernel function is 0.

pressure-gradient acceleration). Thus, **three** types of `copyFromForce` directives are written.

Listing 54: FullParticle type (`fp_sph` type)

```
1  typedef struct fp_sph { //$fdps FP
2     //$fdps copyFromForce force_grav (acc,acc_grav) (pot,pot_grav)
3     //$fdps copyFromForce force_dens (flag,flag) (dens,dens) (smth,smth) (
            gradh,gradh) (divv,divv) (rotv,rotv)
4     //$fdps copyFromForce force_hydro (acc,acc_hydro) (eng_dot,eng_dot) (
            ent_dot,ent_dot) (dt,dt)
5     long long id; //$fdps id
6     double mass; //$fdps charge
7     fdps_f64vec pos; //$fdps position
8     fdps_f64vec vel;
9     fdps_f64vec acc_grav;
10    double pot_grav;
11    fdps_f64vec acc_hydro;
12    int flag;
13    double dens;
14    double eng;
15    double ent;
16    double pres;
17    double smth;
18    double gradh;
19    double divv;
20    fdps_f64vec rotv;
21    double balsw;
22    double snds;
23    double eng_dot;
24    double ent_dot;
25    double dt;
26    fdps_f64vec vel_half;
27    double eng_half;
28    double ent_half;
29 } FP_sph;
```

#### 7.1.3.2 EssentialParticle type

First, we explain structure `ep_grav` , which is used for the gravity calculation. This data type has all physical quantities that $i$- and $j$-particles should have in order to perform gravity calculation as member variables. Listing 55 shows the implementation of `ep_grav` type . EssentialParticle type should have `copyFromFP` directive(s) to specify the way of copy data from FullParticle type(s). In this code, there are two FullParticle types and hence **two** `copyFromFP` directives are written.

Listing 55: EssentialParticle type (`ep_grav` type)

```
1  typedef struct ep_grav { //$fdps EPI,EPJ
2     //$fdps copyFromFP fp_nbody (id,id) (mass,mass) (pos,pos)
3     //$fdps copyFromFP fp_sph (id,id) (mass,mass) (pos,pos)
4     long long id; //$fdps id
5     double mass; //$fdps charge
```

```
6     fdps_f64vec pos; //$fdps position
7 } EP_grav;
```

Next, we explain structure `ep_hydro` , which is used for the calculations of density and pressure-gradient acceleration. This data type has all physical quantities that $i$- and $j$-partiles should have in order to perform the calculations of density and pressure-gradient acceleration. Listing 56 shows the implementation of `ep_hydro` type .

Listing 56: EssentialParticle type (`ep_hydro` type)

```
1  typedef struct ep_hydro { //$fdps EPI,EPJ
2     //$fdps copyFromFP fp_sph (id,id) (pos,pos) (vel,vel) (mass,mass) (smth
           ,smth) (dens,dens) (pres,pres) (gradh,gradh) (snds,snds) (balsw,
           balsw)
3     long long id; //$fdps id
4     fdps_f64vec pos; //$fdps position
5     fdps_f64vec vel;
6     double mass; //$fdps charge
7     double smth; //$fdps rsearch
8     double dens;
9     double pres;
10    double gradh;
11    double snds;
12    double balsw;
13 } EP_hydro;
```

#### 7.1.3.3 Force type

First, we explain structure `force_grav` , which is a Force type used for the gravity calculation. This data type must have all physical quantities that are obtained as the result of the gravity calculation. Listing 57 shows the implementation of `force_grav` type .

Listing 57: Force type (`force_grav` type)

```
1 typedef struct force_grav { //$fdps Force
2     //$fdps clear
3     fdps_f64vec acc;
4     double pot;
5 } Force_grav;
```

Next, we explain structure `force_dens` , which is a Force type used for the density calculation. This data type must have all physical quantities that are obtained as the result of the density calculation. Listing 58 shows the implementation of `force_dens` type . In the Springel's SPH scheme, the smoothing length $h_i$ changes depending on the density at the position of a particle, $\rho_i$. In other words, $h_i$ is also updated with $\rho_i$. Therefore, there is member variable `smth` to store updated smoothing length. In this code, we calculate $\nabla h$ term, $(\nabla \cdot \boldsymbol{v})_i$ $(\nabla \times \boldsymbol{v})_i$ at the same time (if `USE_BALSARA_SWITCH` is defined). Thus, there are member variables `gradh`, `divv`, `rotv` to store them. Member variable `flag` is used to store the result of iteration calculation of $\rho_i$ and $h_i$ (for detail, see § 7.1.4.2)。

Listing 58: Force type (`force_dens` type)

```
1 typedef struct force_dens { //$fdps Force
```

```
2     //$fdps clear smth=keep
3     int flag;
4     double dens;
5     double smth;
6     double gradh;
7     double divv;
8     fdps_f64vec rotv;
9  } Force_dens;
```

Finally, we explain structure `force_hydro` , which is a Force type used for the calculation of pressure-gradient acceleration. This data type must have all physical quantities that are obtained as the result of the calculation of pressure-gradient acceleration. Listing 59 shows the implementation of `force_hydro` type .

Listing 59: Force type (`force_hydro` type)

```
1  typedef struct force_hydro { //$fdps Force
2     //$fdps clear
3     fdps_f64vec acc;
4     double eng_dot;
5     double ent_dot;
6     double dt;
7  } Force_hydro;
```

### 7.1.4 Interaction functions

All interaction functions are implemented in `user_defined.c` . There are **three** types of interaction functions. Below, we explain them.

#### 7.1.4.1 Interaction function for the gravity calculation

Interaction functions for the gravity calculation are implemented as `void` functions `calc_gravity_ep_ep` and `calc_gravity_ep_sp` . Listing 60 shows the implementation. The implementation is almost the same as that of the $N$-body sample code introduced in § 3-4. For detail, please the corresponding section.

Listing 60: Interaction function for the gravity calculation

```
1  #if defined(ENABLE_PHANTOM_GRAPE_X86)
2  void calc_gravity_ep_ep(struct ep_grav *ep_i,
3                          int n_ip,
4                          struct ep_grav *ep_j,
5                          int n_jp,
6                          struct force_grav *f) {
7      int i,j;
8      int nipipe = n_ip;
9      int njpipe = n_jp;
10     double (*xi)[3] = (double (*)[3])malloc(sizeof(double) * nipipe * 3);
11     double (*ai)[3] = (double (*)[3])malloc(sizeof(double) * nipipe * 3);
12     double  *pi     = (double  *    )malloc(sizeof(double) * nipipe);
13     double (*xj)[3] = (double (*)[3])malloc(sizeof(double) * njpipe * 3);
14     double  *mj     = (double  *    )malloc(sizeof(double) * njpipe);
15     for (i = 0; i < n_ip; i++) {
```

```
16          xi[i][0] = ep_i[i].pos.x;
17          xi[i][1] = ep_i[i].pos.y;
18          xi[i][2] = ep_i[i].pos.z;
19          ai[i][0] = 0.0;
20          ai[i][1] = 0.0;
21          ai[i][2] = 0.0;
22          pi[i]    = 0.0;
23      }
24      for (j = 0; j < n_jp; j++) {
25          xj[j][0] = ep_j[j].pos.x;
26          xj[j][1] = ep_j[j].pos.y;
27          xj[j][2] = ep_j[j].pos.z;
28          mj[j]    = ep_j[j].mass;
29      }
30  #if defined(PARTICLE_SIMULATOR_THREAD_PARALLEL) && defined(_OPENMP)
31      int devid = omp_get_thread_num();
32      // [IMPORTANT NOTE]
33      //   The function calc_gravity_ep_ep is called by a OpenMP thread
34      //   in the FDPS. This means that here is already in the parallel
              region.
35      //   So, you can use omp_get_thread_num() without !$OMP parallel
              directives.
36      //   If you use them, a nested parallel resions is made and the
              gravity
37      //   calculation will not be performed correctly.
38  #else
39      int devid = 0;
40  #endif
41      g5_set_xmjMC(devid, 0, n_jp, xj, mj);
42      g5_set_nMC(devid, n_jp);
43      g5_calculate_force_on_xMC(devid, xi, ai, pi, n_ip);
44      for (i = 0; i < n_ip; i++) {
45         f[i].acc.x += ai[i][0];
46         f[i].acc.y += ai[i][1];
47         f[i].acc.z += ai[i][2];
48         f[i].pot   -= pi[i];
49      }
50      free(xi);
51      free(ai);
52      free(pi);
53      free(xj);
54      free(mj);
55  }
56
57  void calc_gravity_ep_sp(struct ep_grav *ep_i,
58                          int n_ip,
59                          fdps_spj_monopole *ep_j,
60                          int n_jp,
61                          struct force_grav *f) {
62      int i,j;
63      int nipipe = n_ip;
64      int njpipe = n_jp;
65      double (*xi)[3] = (double (*)[3])malloc(sizeof(double) * nipipe * 3);
66      double (*ai)[3] = (double (*)[3])malloc(sizeof(double) * nipipe * 3);
67      double  *pi     = (double  *    )malloc(sizeof(double) * nipipe);
```

```
68      double (*xj)[3] = (double (*)[3])malloc(sizeof(double) * njpipe * 3);
69      double  *mj     = (double  *    )malloc(sizeof(double) * njpipe);
70      for (i = 0; i < n_ip; i++) {
71          xi[i][0] = ep_i[i].pos.x;
72          xi[i][1] = ep_i[i].pos.y;
73          xi[i][2] = ep_i[i].pos.z;
74          ai[i][0] = 0.0;
75          ai[i][1] = 0.0;
76          ai[i][2] = 0.0;
77          pi[i]    = 0.0;
78      }
79      for (j = 0; j < n_jp; j++) {
80          xj[j][0] = ep_j[j].pos.x;
81          xj[j][1] = ep_j[j].pos.y;
82          xj[j][2] = ep_j[j].pos.z;
83          mj[j]    = ep_j[j].mass;
84      }
85  #if defined(PARTICLE_SIMULATOR_THREAD_PARALLEL) && defined(_OPENMP)
86      int devid = omp_get_thread_num();
87      // [IMPORTANT NOTE]
88      //   The function calc_gravity_ep_ep is called by a OpenMP thread
89      //   in the FDPS. This means that here is already in the parallel
              region.
90      //   So, you can use omp_get_thread_num() without !$OMP parallel
              directives.
91      //   If you use them, a nested parallel resions is made and the
              gravity
92      //   calculation will not be performed correctly.
93  #else
94      int devid = 0;
95  #endif
96      g5_set_xmjMC(devid, 0, n_jp, xj, mj);
97      g5_set_nMC(devid, n_jp);
98      g5_calculate_force_on_xMC(devid, xi, ai, pi, n_ip);
99      for (i = 0; i < n_ip; i++) {
100        f[i].acc.x += ai[i][0];
101        f[i].acc.y += ai[i][1];
102        f[i].acc.z += ai[i][2];
103        f[i].pot   -= pi[i];
104     }
105     free(xi);
106     free(ai);
107     free(pi);
108     free(xj);
109     free(mj);
110 }
111 #else
112 void calc_gravity_ep_ep(struct ep_grav *ep_i,
113                         int n_ip,
114                         struct ep_grav *ep_j,
115                         int n_jp,
116                         struct force_grav *f) {
117     int i,j;
118     double eps2 = eps_grav * eps_grav;
119     for (i = 0; i < n_ip; i++) {
```

```
120         fdps_f64vec xi,ai;
121         double poti;
122         xi.x = ep_i[i].pos.x;
123         xi.y = ep_i[i].pos.y;
124         xi.z = ep_i[i].pos.z;
125         ai.x = 0.0;
126         ai.y = 0.0;
127         ai.z = 0.0;
128         poti = 0.0;
129         for (j = 0; j < n_jp; j++) {
130             fdps_f64vec rij;
131             rij.x  = xi.x - ep_j[j].pos.x;
132             rij.y  = xi.y - ep_j[j].pos.y;
133             rij.z  = xi.z - ep_j[j].pos.z;
134             double r3_inv = rij.x * rij.x
135                           + rij.y * rij.y
136                           + rij.z * rij.z
137                           + eps2;
138             double r_inv  = 1.0/sqrt(r3_inv);
139             r3_inv = r_inv * r_inv;
140             r_inv  = r_inv * ep_j[j].mass;
141             r3_inv = r3_inv * r_inv;
142             ai.x -= r3_inv * rij.x;
143             ai.y -= r3_inv * rij.y;
144             ai.z -= r3_inv * rij.z;
145             poti -= r_inv;
146         }
147         f[i].acc.x += ai.x;
148         f[i].acc.y += ai.y;
149         f[i].acc.z += ai.z;
150         f[i].pot   += poti;
151     }
152 }
153
154 void calc_gravity_ep_sp(struct ep_grav *ep_i,
155                         int n_ip,
156                         fdps_spj_monopole *ep_j,
157                         int n_jp,
158                         struct force_grav *f) {
159     int i,j;
160     double eps2 = eps_grav * eps_grav;
161     for (i = 0; i < n_ip; i++) {
162         fdps_f64vec xi,ai;
163         double poti;
164         xi.x = ep_i[i].pos.x;
165         xi.y = ep_i[i].pos.y;
166         xi.z = ep_i[i].pos.z;
167         ai.x = 0.0;
168         ai.y = 0.0;
169         ai.z = 0.0;
170         poti = 0.0;
171         for (j = 0; j < n_jp; j++) {
172             fdps_f64vec rij;
173             rij.x  = xi.x - ep_j[j].pos.x;
174             rij.y  = xi.y - ep_j[j].pos.y;
```

```
175             rij.z  = xi.z - ep_j[j].pos.z;
176             double r3_inv = rij.x * rij.x
177                           + rij.y * rij.y
178                           + rij.z * rij.z
179                           + eps2;
180             double r_inv  = 1.0/sqrt(r3_inv);
181             r3_inv = r_inv * r_inv;
182             r_inv  = r_inv * ep_j[j].mass;
183             r3_inv = r3_inv * r_inv;
184             ai.x -= r3_inv * rij.x;
185             ai.y -= r3_inv * rij.y;
186             ai.z -= r3_inv * rij.z;
187             poti -= r_inv;
188         }
189         f[i].acc.x += ai.x;
190         f[i].acc.y += ai.y;
191         f[i].acc.z += ai.z;
192         f[i].pot   += poti;
193     }
194 }
195 #endif
```

#### 7.1.4.2 Interaction function for the density calculation

Interaction function for the density calculation is implemented as `void` function `calc_density` . Listing 61 shows its implementation. The implementation actually used differs depending on the state of macro `ENABLE_VARIABLE_SMOOTHING_LENGTH`. If this macro is not defined, an implementation for fixed smoothing length is used. Its source code is almost the same as the interaction function for the density calculation of the SPH sample code described in § 3-4. Thus, we omit explanation for this case. Below, we explain an implementation used for the case that the above macro is defined.

As described in § 7.1.2, we need to determine the density $\rho_i$ and smoothing length $h_i$ at the same time by solving Eqs.(14) and (9) self-consistently. For this, we need to perform an iterative calculation. This calculation is performed in the infinite `for` loop in the code. As you'll see by reading the source code of `void` function `calc_density_wrapper` in `c_main.c` , this sample code performs the density calculation after multiplying the smoothing lengths of all particles by a constant `SCF_smth` in order to make the density calculation efficiently. By this, we can change $h_i$ between 0 and $h_{\mathrm{max,alw}} \equiv$ `SCF_smth` $\times h_{i,0}$, during the iteration, where $h_{i,0}$ is the value of the smoothing length of particle $i$ before we multiply by `SCF_smth`. This is because all of particles that is eligible to be $j$-particles are contained in the current $j$-particle list (`ep_j`). If the iteration does not converge for some particle $i$, we cannot determine $\rho_i$ and $h_i$ for this particle by using the current $j$ particle list because the value of the smoothing length we want to obtain will be larger than $h_{\mathrm{max,alw}}$. In this case, we need to perform the density calculation again after increasing $h_{i,0}$. This "outer" iteration is performed in `void` function `calc_density_wrapper` in `c_main.c` . We will describe this `void` function in § 7.1.5.

After the infinite `for` loop, this `void` function performs the calculations of $\nabla h$, $(\nabla \cdot \boldsymbol{v})_i$, and $(\nabla \times \boldsymbol{v})_i$.

Listing 61: Interaction function for the density calculation

```
1  void calc_density(struct ep_hydro *ep_i,
2                    int n_ip,
3                    struct ep_hydro *ep_j,
4                    int n_jp,
5                    struct force_dens *f) {
6  #if defined(ENABLE_VARIABLE_SMOOTHING_LENGTH)
7      // Local parameters
8      const double eps=1.0e-6;
9      // Local variables
10     int i,j;
11     int n_unchanged;
12     double M,M_trgt;
13     double dens,drho_dh;
14     double h,h_max_alw,h_L,h_U,dh,dh_prev;
15     fdps_f64vec dr,dv,gradW_i;
16     double *mj  = (double *)malloc(sizeof(double) * n_jp);
17     double *rij = (double *)malloc(sizeof(double) * n_jp);
18     M_trgt = mass_avg * N_neighbor;
19     for (i = 0; i < n_ip; i++) {
20         dens = 0.0;
21         h_max_alw = ep_i[i].smth; // maximum allowance
22         h = h_max_alw / SCF_smth;
23         // Note that we increase smth by a factor of scf_smth
24         // before calling calc_density().
25         h_L = 0.0;
26         h_U = h_max_alw;
27         dh_prev = 0.0;
28         n_unchanged = 0;
29         // Software cache
30         for (j = 0; j < n_jp; j++) {
31             mj[j] = ep_j[j].mass;
32             dr.x = ep_i[i].pos.x - ep_j[j].pos.x;
33             dr.y = ep_i[i].pos.y - ep_j[j].pos.y;
34             dr.z = ep_i[i].pos.z - ep_j[j].pos.z;
35             rij[j] = sqrt(dr.x * dr.x
36                           +dr.y * dr.y
37                           +dr.z * dr.z);
38         }
39         for(;;) {
40             // Calculate density
41             dens = 0.0;
42             for (j = 0; j < n_jp; j++)
43                dens += mj[j] * W(rij[j], h);
44             // Check if the current value of the smoohting length
                       satisfies
45             // Eq.(5) in Springel (2005).
46             M = 4.0 * pi * h * h * h * dens / 3.0;
47             if ((h < h_max_alw) && (fabs(M/M_trgt - 1.0) < eps)) {
48                 // In this case, Eq.(5) holds within a specified accuracy.
49                 f[i].flag = 1;
50                 f[i].dens = dens;
51                 f[i].smth = h;
52                 break;
53             }
54             if (((h == h_max_alw) && (M < M_trgt)) || (n_unchanged == 4))
```

```
                {
55                  // In this case, we skip this particle forcibly.
56                  // In order to determine consistently the density
57                  // and the smoohting length for this particle,
58                  // we must re-perform calcForceAllAndWriteBack().
59                  f[i].flag = 0;
60                  f[i].dens = dens;
61                  f[i].smth = h_max_alw;
62                  break;
63              }
64              // Update h_L & h_U
65              if (M < M_trgt) {
66                  if (h_L < h) h_L = h;
67              } else if (M_trgt < M) {
68                  if (h < h_U) h_U = h;
69              }
70              dh = h_U - h_L;
71              if (dh == dh_prev) {
72                 n_unchanged++;
73              } else {
74                 dh_prev = dh;
75                 n_unchanged = 0;
76              }
77              // Update smoothing length
78              h = pow((3.0 * M_trgt)/(4.0 * pi * dens),1.0/3.0);
79              if ((h <= h_L) || (h == h_U)) {
80                 // In this case, we switch to the bisection search.
81                 // The inclusion of '=' in the if statement is very
82                 // important to escape a limit cycle.
83                 h = 0.5 * (h_L + h_U);
84              } else if (h_U < h) {
85                 h = h_U;
86              }
87          }
88          // Calculate grad-h term
89          if (f[i].flag == 1) {
90              drho_dh = 0.0;
91              for (j = 0; j < n_jp; j++)
92                 drho_dh += mj[j] * dWdh(rij[j], h);
93              f[i].gradh = 1.0 / (1.0 + (h * drho_dh) / (3.0 * dens));
94          } else {
95              f[i].gradh = 1.0; // dummy value
96          }
97          // Compute \div v & \rot v for Balsara switch
98  #if defined(USE_BALSARA_SWITCH)
99          for (j = 0; j < n_jp; j++) {
100            dr.x = ep_i[i].pos.x - ep_j[j].pos.x;
101            dr.y = ep_i[i].pos.y - ep_j[j].pos.y;
102            dr.z = ep_i[i].pos.z - ep_j[j].pos.z;
103            dv.x = ep_i[i].vel.x - ep_j[j].vel.x;
104            dv.y = ep_i[i].vel.y - ep_j[j].vel.y;
105            dv.z = ep_i[i].vel.z - ep_j[j].vel.z;
106            gradW_i = gradW(dr, f[i].smth);
107            f[i].divv -= mj[j] * (dv.x * gradW_i.x
108                                  +dv.y * gradW_i.y
```

```
109                                 +dv.z * gradW_i.z);
110             f[i].rotv.x -= mj[j] * (dv.y * gradW_i.z - dv.z * gradW_i.y);
111             f[i].rotv.y -= mj[j] * (dv.z * gradW_i.x - dv.x * gradW_i.z);
112             f[i].rotv.z -= mj[j] * (dv.x * gradW_i.y - dv.y * gradW_i.x);
113         }
114         f[i].divv   /= f[i].dens;
115         f[i].rotv.x /= f[i].dens;
116         f[i].rotv.y /= f[i].dens;
117         f[i].rotv.z /= f[i].dens;
118 #endif
119     }
120     free(mj);
121     free(rij);
122 #else
123     int i,j;
124     for (i = 0; i < n_ip; i++) {
125         f[i].dens = 0.0;
126         for (j = 0; j < n_jp; j++) {
127             fdps_f64vec dr;
128             dr.x = ep_j[j].pos.x - ep_i[i].pos.x;
129             dr.y = ep_j[j].pos.y - ep_i[i].pos.y;
130             dr.z = ep_j[j].pos.z - ep_i[i].pos.z;
131             double rij = sqrt(dr.x * dr.x
132                               +dr.y * dr.y
133                               +dr.z * dr.z);
134             f[i].dens += ep_j[j].mass * W(rij,ep_i[i].smth);
135         }
136         f[i].smth = ep_i[i].smth;
137         f[i].gradh = 1.0;
138         // Compute \div v & \rot v for Balsara switch
139 #if defined(USE_BALSARA_SWITCH)
140         for (j = 0; j < n_jp; j++) {
141             double mj = ep_j[j].mass;
142             fdps_f64vec dr,dv,gradW_i;
143             dr.x = ep_i[i].pos.x - ep_j[j].pos.x;
144             dr.y = ep_i[i].pos.y - ep_j[j].pos.y;
145             dr.z = ep_i[i].pos.z - ep_j[j].pos.z;
146             dv.x = ep_i[i].vel.x - ep_j[j].vel.x;
147             dv.y = ep_i[i].vel.y - ep_j[j].vel.y;
148             dv.z = ep_i[i].vel.z - ep_j[j].vel.z;
149             gradW_i = gradW(dr, f[i].smth);
150             f[i].divv -= mj * (dv.x * gradW_i.x
151                                +dv.y * gradW_i.y
152                                +dv.z * gradW_i.z);
153             f[i].rotv.x -= mj * (dv.y * gradW_i.z - dv.z * gradW_i.y);
154             f[i].rotv.y -= mj * (dv.z * gradW_i.x - dv.x * gradW_i.z);
155             f[i].rotv.z -= mj * (dv.x * gradW_i.y - dv.y * gradW_i.x);
156         }
157         f[i].divv   /= f[i].dens;
158         f[i].rotv.x /= f[i].dens;
159         f[i].rotv.y /= f[i].dens;
160         f[i].rotv.z /= f[i].dens;
161 #endif
162     }
163 #endif
```

```
164 }
```

#### 7.1.4.3 Interaction function for the calculation of pressure-gradient acceleration

Interaction function for the calculation of pressure-gradient acceleration is implemented as `void` function `calc_hydro_force` . Listing 62 shows its implementation. This performs the calculations of the right hand sides of Eqs.(10), (12), and (13), and `dt` according to Eq.(16) in Springel [2005, MNRAS, 364, 1105] (for `dt`, see the definition of `fp_sph` type )。

Listing 62: Interaction function for the calculation of pressure-gradient acceleration

```
1  void calc_hydro_force(struct ep_hydro *ep_i,
2                        int n_ip,
3                        struct ep_hydro *ep_j,
4                        int n_jp,
5                        struct force_hydro *f) {
6      // Local variables
7      int i,j;
8      double mass_i,mass_j,smth_i,smth_j,
9             dens_i,dens_j,pres_i,pres_j,
10            gradh_i,gradh_j,balsw_i,balsw_j,
11            snds_i,snds_j;
12     double povrho2_i,povrho2_j,
13            v_sig_max,dr_dv,w_ij,v_sig,AV;
14     fdps_f64vec pos_i,pos_j,vel_i,vel_j,
15                 dr,dv,gradW_i,gradW_j,gradW_ij;
16     for (i = 0; i < n_ip; i++) {
17         // Zero-clear
18         v_sig_max = 0.0;
19         // Extract i-particle info.
20         pos_i = ep_i[i].pos;
21         vel_i = ep_i[i].vel;
22         mass_i  = ep_i[i].mass;
23         smth_i  = ep_i[i].smth;
24         dens_i  = ep_i[i].dens;
25         pres_i  = ep_i[i].pres;
26         gradh_i = ep_i[i].gradh;
27         balsw_i = ep_i[i].balsw;
28         snds_i  = ep_i[i].snds;
29         povrho2_i = pres_i/(dens_i*dens_i);
30         for (j = 0; j < n_jp; j++) {
31             // Extract j-particle info.
32             pos_j.x = ep_j[j].pos.x;
33             pos_j.y = ep_j[j].pos.y;
34             pos_j.z = ep_j[j].pos.z;
35             vel_j.x = ep_j[j].vel.x;
36             vel_j.y = ep_j[j].vel.y;
37             vel_j.z = ep_j[j].vel.z;
38             mass_j  = ep_j[j].mass;
39             smth_j  = ep_j[j].smth;
40             dens_j  = ep_j[j].dens;
41             pres_j  = ep_j[j].pres;
42             gradh_j = ep_j[j].gradh;
43             balsw_j = ep_j[j].balsw;
```

```
44              snds_j  = ep_j[j].snds;
45              povrho2_j = pres_j/(dens_j*dens_j);
46              // Compute dr & dv
47              dr.x = pos_i.x - pos_j.x;
48              dr.y = pos_i.y - pos_j.y;
49              dr.z = pos_i.z - pos_j.z;
50              dv.x = vel_i.x - vel_j.x;
51              dv.y = vel_i.y - vel_j.y;
52              dv.z = vel_i.z - vel_j.z;
53              // Compute the signal velocity
54              dr_dv = dr.x * dv.x + dr.y * dv.y + dr.z * dv.z;
55              if (dr_dv < 0.0) {
56                 w_ij = dr_dv / sqrt(dr.x * dr.x + dr.y * dr.y + dr.z * dr.z
                        );
57              } else {
58                 w_ij = 0.0;
59              }
60              v_sig = snds_i + snds_j - 3.0 * w_ij;
61              if (v_sig > v_sig_max) v_sig_max = v_sig;
62              // Compute the artificial viscosity
63              AV = - 0.5*v_sig*w_ij / (0.5*(dens_i+dens_j)) * 0.5*(balsw_i+
                      balsw_j);
64              // Compute the average of the gradients of kernel
65              gradW_i  = gradW(dr,smth_i);
66              gradW_j  = gradW(dr,smth_j);
67              gradW_ij.x = 0.5 * (gradW_i.x + gradW_j.x);
68              gradW_ij.y = 0.5 * (gradW_i.y + gradW_j.y);
69              gradW_ij.z = 0.5 * (gradW_i.z + gradW_j.z);
70              // Compute the acceleration and the heating rate
71              f[i].acc.x -= mass_j*(gradh_i * povrho2_i * gradW_i.x
72                                    +gradh_j * povrho2_j * gradW_j.x
73                                    +AV * gradW_ij.x);
74              f[i].acc.y -= mass_j*(gradh_i * povrho2_i * gradW_i.y
75                                    +gradh_j * povrho2_j * gradW_j.y
76                                    +AV * gradW_ij.y);
77              f[i].acc.z -= mass_j*(gradh_i * povrho2_i * gradW_i.z
78                                    +gradh_j * povrho2_j * gradW_j.z
79                                    +AV * gradW_ij.z);
80              f[i].eng_dot += mass_j * gradh_i * povrho2_i * (dv.x * gradW_i
                      .x
81                                                             +dv.y * gradW_i
                                                                    .y
82                                                             +dv.z * gradW_i
                                                                    .z);
83                           + mass_j * 0.5 * AV * (dv.x * gradW_ij.x
84                                                 +dv.y * gradW_ij.y
85                                                 +dv.z * gradW_ij.z);
86              f[i].ent_dot += 0.5 * mass_j * AV * (dv.x * gradW_ij.x
87                                                   +dv.y * gradW_ij.y
88                                                   +dv.z * gradW_ij.z);
89          }
90          f[i].ent_dot *= ((specific_heat_ratio - 1.0)
91                           /pow(dens_i,specific_heat_ratio - 1.0));
92          f[i].dt = CFL_hydro*2.0*smth_i/v_sig_max;
93      }
```

```
94 }
```

### 7.1.5 Main body of the sample code

In this section, we describe the main body of the sample code implemented mainly in `c_main.c`. Before entering a detailed explanation, we describe here the overall structure of the code. As described in the beginning of § 7.1, this code performs a $N$-body/SPH simulation of a disk galaxy. Thus, in the default, the code sets an initial condition for a disk galaxy. But, initial conditions for simple test calculations are also prepared in the code. More specifically, the code supports the following four types of initial conditions:

(a) Initial condition for a disk galaxy simulation. It is selected when `-DINITIAL_CONDITION=0` is specified at the compile-time. The initial condition is created in `void` function `galaxy_IC` in `ic.c`. The initial distributions of dark matter and star particles are set by reading a file created by MAGI. The initial distribution of gas (SPH) particles is determined in the subroutine. In the default, an exponential disk ($M = 10^{10}$ $M_\odot$, $R_s = 7$ kpc [scale radius], $R_t = 12.5$ kpc [truncation radius], $z_d = 0.4$ kpc [scale height], $z_t = 1$ kpc [truncation height]) is created with the number of SPH particles of $2^{18}$.
(b) Initial condition for cold collapse test. It is selected when `-DINITIAL_CONDITION=1` is specified at the compile-time. The initial condition is created in `void` function `cold_collapse_test_IC` in `ic.c`.
(c) Initial condition for the Evrard test (§ 3.3 in Evrard [1988,MNRAS,235,911]). It is selected when `-DINITIAL_CONDITION=2` is specified at the compile-time. This initial condition is created in `void` function `Evrard_test_IC` in `ic.c`. There are two options for the way of creating an initial condition. We can specify the way by manually set the value of the last argument of the function 0 or 1. If 0 is given, the function creates the density profile of the Evrard gas sphere by rescaling the positions of particles which are placed in a grid. If 1 is specified, it creates the density profile by rescaling the positions of particles which are distributed glass-like. In order to use the second option, we have to create particle data by executing the code with the mode described in the next item.
(d) Operation mode to create a glass-like distribution of SPH particles in a box of $[-1, 1)^3$. This mode is selected when `-DINITIAL_CONDITION=3` is specified at the compile-time. The initial condition is created in `void` function `make_glass_IC` in `ic.c`.

The structure of the sample code is as follows:

(1) Create and initialize FDPS objects
(2) Initialize the Phantom-GRAPE library for x86 if needed
(3) Read a data file of $N$-body particles and make an initial condition
(4) Calculate the motions of particles until the end time we specify

Below, we explain each item in detail.

#### 7.1.5.1 Include the header file of FDPS C interface

In order to use the features of FDPS, `FDPS_c_if.h` is included in the beginning part of `c_main.c`.

Listing 63: Include the header file of FDPS C interface

```
1 #include "FDPS_c_if.h"
```

#### 7.1.5.2 Initialization and and termination of FDPS

We need first to initialize FDPS by calling API `fdps_initialize` :

Listing 64: Initialize FDPS

```
1 fdps_initialize();
```

Once started, FDPS should be explicitly terminated by calling API `fdps_finalize` . This sample code terminates FDPS just before the termination of the program. You can find the following code at the last part of `c_main.c`.

Listing 65: Finalize FDPS

```
1 fdps_finalize();
```

#### 7.1.5.3 Creation and initialization of FDPS objects

After the initialization of FDPS, a user need to create the objects used to talk to FDPS. In this section, we describe how to create and initialize these objects.

##### *7.1.5.3.1 Creation and initialization of ParticleSystem objects*

This sample code uses different ParticleSystem objects to manage $N$-body and SPH particles. Two integer variables `psys_num_nbody` and `psys_num_sph` are used to store the identification numbers for ParticleSystem objects for $N$-body and SPH particles, respectively. Using these variables, the creation and the initialization of the objects are done as follows.

Listing 66: Creation and initialization of ParticleSystem objects

```
1 fdps_create_psys(&psys_num_nbody,"fp_nbody");
2 fdps_init_psys(psys_num_nbody);
3 fdps_create_psys(&psys_num_sph,"fp_sph");
4 fdps_init_psys(psys_num_sph);
```

##### *7.1.5.3.2 Creation and initialization of DomainInfo object*

This sample code decomposes the computational domain so that the *total* ($N$-body + SPH) particle distribution is divided equally. In this case, we need one DomainInfo object. Thus, using one integer variable `dinfo_num`, the creation and initialization of DomainInfo object are performed as follows.

Listing 67: Creation and initialization of DomainInfo object

```
1 fdps_create_dinfo(&dinfo_num);
2 fdps_init_dinfo(dinfo_num,coef_ema);
```

#### *7.1.5.3.3 Creation and initialization of TreeForForce objects*

The code uses three types of TreeForForce objects and they are used for the gravity calculation, the density calculation, and the calculation of pressure-gradient acceleration. When initializing a TreeForForce object, we must pass a typical number of particles used in the interaction calculation as the second argument of API `fdps_init_tree` . For TreeForForce object `tree_num_grav` , the value that is three times of the number of local particles ($N$-body + SPH) is passed. On the other hand, for TreeForForce objects `tree_num_dens` and `tree_num_hydro` , the value that is three times of the number of local SPH particles is passed.

Listing 68: Creation and initialization of TreeForForce objects

```
1     // Make three tree structures
2     int nptcl_loc_sph = 1;
3     if (fdps_get_nptcl_loc(psys_num_sph) > 1)
4         nptcl_loc_sph = fdps_get_nptcl_loc(psys_num_sph);
5     int nptcl_loc_nbody = fdps_get_nptcl_loc(psys_num_nbody);
6     int nptcl_loc_all   = nptcl_loc_nbody + nptcl_loc_sph;
7     // tree for gravity calculation
8     int tree_num_grav;
9     fdps_create_tree(&tree_num_grav,
10                     "Long,force_grav,ep_grav,ep_grav,Monopole");
11    const float theta=0.5;
12    const int n_leaf_limit=8, n_group_limit=64;
13    fdps_init_tree(tree_num_grav, 3*nptcl_loc_all, theta,
14                   n_leaf_limit, n_group_limit);
15    // tree for the density calculation
16    int tree_num_dens;
17    fdps_create_tree(&tree_num_dens,
18                     "Short,force_dens,ep_hydro,ep_hydro,Gather");
19    fdps_init_tree(tree_num_dens, 3*nptcl_loc_sph, theta,
20                   n_leaf_limit, n_group_limit);
21    // tree for the hydrodynamic force calculation
22    int tree_num_hydro;
23    fdps_create_tree(&tree_num_hydro,
24                     "Short,force_hydro,ep_hydro,ep_hydro,Symmetry");
25    fdps_init_tree(tree_num_hydro, 3*nptcl_loc_sph, theta,
26                   n_leaf_limit, n_group_limit);
```

### 7.1.5.4 Setting initial condition

The initial condition is set in `void` function `setup_IC` , which internally calls a different `void` function depending on the value of macro `INITIAL_CONDITION`. The correspondence relation between the name of a internally-called `void` function and the value of the macro has been described already in the beginning part of § 7.1.5. The arguments `time_dump`, `dt_dump`, `time_end` represents the initial time of data output, the time interval of data output, and the end time of the simulation, respectively. These must be set in this `void` function. Also, the boundary condition, the gravitational softening (`eps_grav`), the maximum allowable time step of the system (`dt_max`) are set in this `void` function (a user does not necessarily set `dt_max`).

Listing 69: Setting initial condition

```
1 setup_IC(psys_num_nbody, psys_num_sph, dinfo_num,
2          &time_dump, &dt_dump, &time_end);
```

In what follows, we describe some of points to remember for `void` function `galaxy_IC` .

- MAGI outputs particle data in its code unit. The information about the MAGI's code unit is described in file `./magi_data/doc/unit.txt` (see section "Computational unit"). This file is created when executing MAGI. The variables `magi_unit_mass`, `magi_unit_leng`, `magi_unit_time` in the `void` function must be consistent with the MAGI's code unit.

- The `void` function reads particle data from file of the name of `./magi_data/dat/Galaxy.tipsy` in the default. If you make the code read a different file, please change the source code manually.

- The `void` function generates an initial gas distribution which has exponential profile along both $R$ ($\equiv \sqrt{x^2+y^2}$) and $z$ directions. The variables `Rs` and `zd` represents the scale lengths. The variables `Rt` and `zt` represents the truncation (cutoff) lengths.

- The initial thermodynamic state is specified by both the initial gas temperature `temp` and the mean molecular weight relative to the mass of hydrogen atom `mu`. Regardless of the state of the macro `USE_ENTROPY`, a user must specify the thermodynamic state of SPH particles via the specific internal energy (member variable `eng` in `fp_sph` type )[the sample code automatically does this]. If the macro `USE_ENTROPY` is defined, the initial value of the entropy is automatically set by `void` function `set_entropy` called in the `void` function `c_main()`, using the initial value of the specific internal energy and the calculated density. On the other hand, if the macro is not defined, the value of `eng` set in the `void` function `galaxy_IC` is treated as the initial value of the specific internal energy.

#### 7.1.5.5 Domain decomposition

When there are different types of ParticleSystem objects, the domain decomposition based on the combined distribution of particles can be realized by using APIs `fdps_collect_sample_particle` and `fdps_decompose_domain` . First, a user have to collect sample particles from each ParticleSystem object using API `fdps_collect_sample_particle` . **Here, we must pass `.false.` to the third argument of this API for the second or later ParticleSystem object because the previous information is cleared without this**. After collecting sample particles from all of ParticleSystem objects, call API `fdps_decompose_domain` to perform domain decomposition.

Listing 70: Domain decomposition

```
1 fdps_collect_sample_particle(dinfo_num, psys_num_nbody, clear);
2 fdps_collect_sample_particle(dinfo_num, psys_num_sph, unclear);
3 fdps_decompose_domain(dinfo_num);
```

#### 7.1.5.6 Particle exchange

In order to perform particle exchange based on the previous-calculated domain information, it is only necessary to call API `fdps_exchange_particle` .

Listing 71: Particle exchange

```
fdps_exchange_particle(psys_num_nbody,dinfo_num);
fdps_exchange_particle(psys_num_sph,dinfo_num);
```

#### 7.1.5.7 Interaction calculations

After the domain decomposition and particle exchange, interaction calculations are done. Below, we show the implementation of the interaction calculations just after setting the initial condition. At first, the code performs the gravity calculation. Then, it performs the calculations of density and pressure-gradient acceleration.

Listing 72: Interaction calculations

```
    // Gravity calculation
    double t_start = fdps_get_wtime();
#if defined(ENABLE_GRAVITY_INTERACT)
    fdps_set_particle_local_tree(tree_num_grav, psys_num_nbody, true);
    fdps_set_particle_local_tree(tree_num_grav, psys_num_sph, false);
    fdps_calc_force_making_tree(tree_num_grav,
                                calc_gravity_ep_ep,
                                calc_gravity_ep_sp,
                                dinfo_num,
                                true);
    nptcl_loc_nbody = fdps_get_nptcl_loc(psys_num_nbody);
    FP_nbody *ptcl_nbody = (FP_nbody *) fdps_get_psys_cptr(psys_num_nbody)
        ;
    for (i = 0; i < nptcl_loc_nbody; i++) {
        Force_grav f_grav;
        void *pforce = (void *) &f_grav;
        fdps_get_force(tree_num_grav, i, pforce);
        ptcl_nbody[i].acc.x = f_grav.acc.x;
        ptcl_nbody[i].acc.y = f_grav.acc.y;
        ptcl_nbody[i].acc.z = f_grav.acc.z;
        ptcl_nbody[i].pot   = f_grav.pot;
    }
    int offset = nptcl_loc_nbody;
    nptcl_loc_sph = fdps_get_nptcl_loc(psys_num_sph);
    FP_sph *ptcl_sph = (FP_sph *) fdps_get_psys_cptr(psys_num_sph);
    for (i = 0; i < nptcl_loc_sph; i++) {
        Force_grav f_grav;
        fdps_get_force(tree_num_grav, i + offset, (void *)&f_grav);
        ptcl_sph[i].acc_grav.x = f_grav.acc.x;
        ptcl_sph[i].acc_grav.y = f_grav.acc.y;
        ptcl_sph[i].acc_grav.z = f_grav.acc.z;
        ptcl_sph[i].pot_grav   = f_grav.pot;
    }
#endif
    double t_grav = fdps_get_wtime() - t_start;
    // SPH calculations
```

```
36      t_start = fdps_get_wtime();
37  #if defined(ENABLE_HYDRO_INTERACT)
38      calc_density_wrapper(psys_num_sph, dinfo_num, tree_num_dens);
39      set_entropy(psys_num_sph);
40      set_pressure(psys_num_sph);
41      fdps_calc_force_all_and_write_back(tree_num_hydro,
42                                         calc_hydro_force,
43                                         NULL,
44                                         psys_num_sph,
45                                         dinfo_num,
46                                         true,
47                                         FDPS_MAKE_LIST);
48  #endif
49      double t_hydro = fdps_get_wtime() - t_start;
```

First, we explain the part of the implementation for the gravity calculation. In the gravity calculation, both $N$-body and SPH particles are involved. In order to perform an interaction calculation between different types of particles, we must use in combination TreeForForce object's APIs `fdps_set_particle_local_tree` and `fdps_calc_force_making_tree`. We first pass the particle information stored in each ParticleSystem object to a TreeForForce object using API `fdps_set_particle_local_tree`. **Here, we must pass `false` to the third argument of this API for the second or later ParticleSystem objects because all of the previously-passed information is cleared without this**. After finishing calling this API for all of ParticleSystem objects that are involved in the gravity calculation, call API `fdps_calc_force_making_tree` to perform the interaction calculation. In order to obtain the result of the interaction calculation, we need to use API `fdps_get_force`. This API takes an integral argument $i$, and it writes the force of the $i$th particle read by API `fdps_set_particle_local_tree` in the address specified by the third argument of the API. Hence, we must use appropriate offset to obtain the results of the interaction calculation of the second or later ParticleSystem.

Next, we explain the part of the implementation for the calculations of density and pressure-gradient acceleration. These interaction calculations involves only single type of particles, SPH particles. Therefore, we can use API `fdps_calc_force_all_and_write_back`, which is frequently used in the sample code introduced in this document. For the calculation of pressure-gradient acceleration, the code performs this API in the `void` function `c_main()`. On the other hand, we need to handle the case that the iteration calculation of $\rho_i$ and $h_i$ does not converge for some particles as described in § 7.1.4. This handling is done in the `void` function `calc_density_wrapper`. The implementation of this `void` function is shown below. The implementation actually used differs depending on the state of the macro `ENABLE_VARIABLE_SMOOTHING_LENGTH`. If it is not defined, the code calls API `fdps_calc_force_all_and_write_back` only once because in this case the code performs SPH calculation as the fixed smoothing length SPH code. If the macro is defined, the code calls the API repeatedly until $\rho_i$ and $h_i$ of all the particles are self-consistently determined. The member variable `flag` stores the result of the iteration calculation and the value of 1 means that the iteration converges successfully. So, the code stops the infinite `for` loop when the number of SPH particles whose `flag` has the value of 1 agrees with the total number of SPH particles.

Listing 73: Function `calc_density_wrapper`

```
1  void calc_density_wrapper(int psys_num,
2                            int dinfo_num,
3                            int tree_num) {
4  #if defined(ENABLE_VARIABLE_SMOOTHING_LENGTH)
5     int nptcl_loc = fdps_get_nptcl_loc(psys_num);
6     int nptcl_glb = fdps_get_nptcl_glb(psys_num);
7     FP_sph *ptcl = (FP_sph *) fdps_get_psys_cptr(psys_num);
8     // Determine the density and the smoothing length
9     // so that Eq.(6) in Springel (2005) holds within a specified accuracy.
10    for (;;) {
11        // Increase smoothing length
12        int i;
13        for (i = 0; i < nptcl_loc; i++) ptcl[i].smth *= SCF_smth;
14        // Compute density, etc.
15        fdps_calc_force_all_and_write_back(tree_num,
16                                           calc_density,
17                                           NULL,
18                                           psys_num,
19                                           dinfo_num,
20                                           true,
21                                           FDPS_MAKE_LIST);
22        // Check convergence
23        int n_compl_loc = 0;
24        for (i = 0; i < nptcl_loc; i++)
25            if (ptcl[i].flag == 1) n_compl_loc++;
26        int n_compl = fdps_get_sum_s32(n_compl_loc);
27        if (n_compl == nptcl_glb) break;
28    }
29 #else
30    fdps_calc_force_all_and_write_back(tree_num,
31                                       calc_density,
32                                       NULL,
33                                       psys_num,
34                                       dinfo_num,
35                                       true,
36                                       FDPS_MAKE_LIST);
37 #endif
38 }
```

`void` function `set_entropy` is called only once just after setting an initial condition. As described earlier, this `void` function is used to set the initial value of the entropy. Because we need the initial density to set the initial value of the entropy using Eq. (8), this `void` function is placed just after `void` function `calc_density_wrapper` . After this, the entropy becomes the independent variable to describe the thermodynamic state of gas if the macro `USE_ENTROPY` is defined.

#### 7.1.5.8 Time integration

This code performs the time integration using the Leapfrog method (see § 4.1.3.5.4 for this method). In this code, $D(\cdot)$ operator is implemented as the `void` function `full_drift`, while $K(\cdot)$ operator is implemented as `void` functions `initial_kick` and `final_kick`.

# 8 User Supports

We accept questions and comments on FDPS at the following mail address:
fdps-support@mail.jmlab.jp
Please provide us with the following information.

## 8.1 Compile-time problem

- Compiler environment (version of the compiler, compile options etc)
- Error message at the compile time
- (if possible) the source code

## 8.2 Run-time problem

- Run-time environment
- Run-time error message
- (if possible) the source code

## 8.3 Other cases

For other problems, please do not hesitate to contact us. We sincerely hope that you'll find FDPS useful for your research.

# 9 License

This software is MIT licensed. Please cite Iwasawa et al. (2016, Publications of the Astronomical Society of Japan, 68, 54) and Namekata et al. (2018, Publications of the Astronomical Society of Japan, 70, 70) if you use the standard functions only.

The extended feature "Particle Mesh" is implemented by using a module of GreeM code (Developers: Tomoaki Ishiyama and Keigo Nitadori) (Ishiyama, Fukushige & Makino 2009, Publications of the Astronomical Society of Japan, 61, 1319; Ishiyama, Nitadori & Makino, 2012 SC'12 Proceedings of the International Conference on High Performance Computing, Networking Stroage and Analysis, No. 5). GreeM code is developed based on the code in Yoshikawa & Fukushige (2005, Publications of the Astronomical Society of Japan, 57, 849). Please cite these three literatures if you use the extended feature "Particle Mesh".

Please cite Tanikawa et al.(2012, New Astronomy, 17, 82) and Tanikawa et al.(2012, New Astronomy, 19, 74) if you use the extended feature "Phantom-GRAPE for x86".