DOCUMENTATION

Zhixuan Li

Date: \today

INTRODUCTION

The Lagrangian code for planet formation code (Lag-PFS) is an open-source simulation code for simulating the disk systems, such as the evolution of disk and planets or satellites formation process in the protoplanetary disk (PPD) or circumplanetary disk (CPD) with Lagrangian method. This code is open-source on GitHub.

This code is developed generally for every kind of disk system, which allows users to implement the mechanisms they want, like the initial distribution of gas in disk, how the particles grow and drift, how the embryos migrate and accrete materials in disk, and so on. And **NOTE** that because we firstly apply this code to simulate the satellites formation in CPDs, the description in this documentation will use the 'planet' as the central body, while the satellites as orbital bodies in the system.

The basic elements of disk system include the gas, solid particles, the central body, the objects in disk and also maybe icelines, gaps, and so on. The frame of this code is under the **main** folder, which contains how to combine the elements of disk and run the system. And under **Test** folder, there are some sub-folders named by the users, which allow users to describe the physical processes in their own ways. Under **Test/simple**, there's a simplest example. Users can enter the test folder created by themselves:

```
cd ~/CpdPhysics/Test/zxl_test
```

and then run:

```
python3 ../../main/runcpd.py
```

If you see:

```
[runcpd]: Congrats!! You finished a successful run which consumed 1.17 seconds
[runcpd]: finished
```

in your terminal, you have successfully finished running your test.

We will present the content and structure in detail (sec. Content and Structure), the acceleration of code: jump (sec. Acceleration).

Structure and Content

We developed this code by Python programming language and by object-oriented programming. We put all the relevant properties of one part in disk in a corresponding class, so in the code you will see several classes below:

1. Main classes:
    - `core.Superparticles`: the aggregates of lots of solid particles.

- `core.PLANET`: includes the properties of single orbital object in disk.
- `core.ICELINE`: includes the properties of single iceline.
- `gas.GAS`: includes the properties distribution of gas.
- `core.System`: integrates every part of disk, and describes the interaction between them.

2. **Functional classes:** some other classes like `Single-SP`, `minTimes`, and so on are functional classes used in some medium steps, which have detailed comments below them.

Then, we will present you the structure of this code in the order of running, which is shown in **main/runcpd.py**, during which we will introduce you where and how to insert your own mechanisms. The overview of running process is shown in Figure 1.

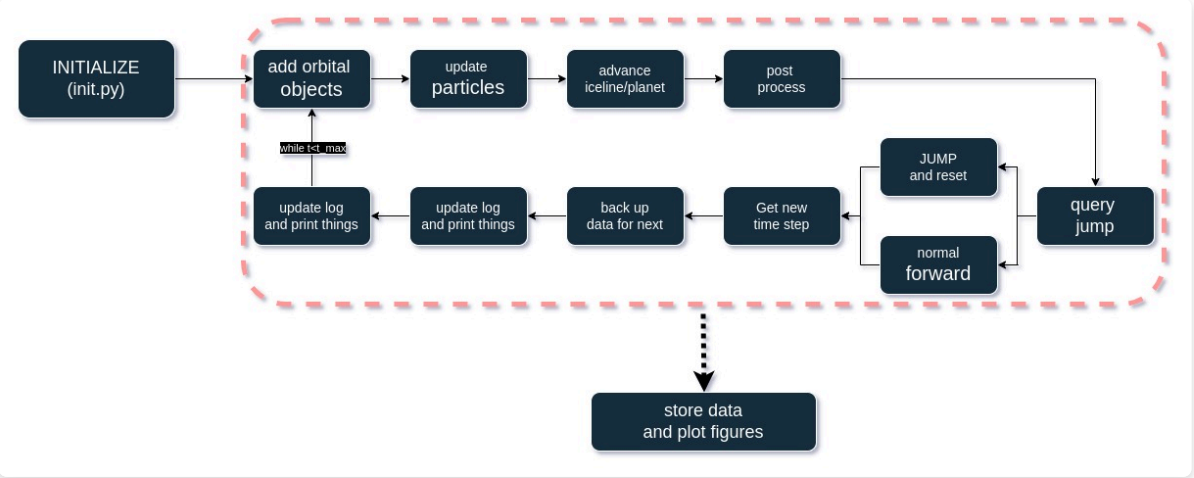

Figure 1: The process of running

INITIALIZE

Firstly, we initialize the system. The initialization is mainly done by the function **sim_init** in **main/init.py**.

The init has two options. One is running from scratch, the other is from the time point that last run ends, which need two pickle files under **/Test/user/pickles/**. Running from pickle files can be done by adding an arguement:

```
python3 ../../main/runcpd.py fromfile
```

Let's now discuss the running from scratch. In the \textbf{sim_init}, we firstly initialize an empty frame (system object), then we get the composition from .txt files under \textbf{config/(comps).txt}, from which we can initialize the gas. And then the orbital objects and icelines in order. (In the code, the orbital objects are called 'planet'). Then, we initialize the particles. The particles from \textbf{Superparticles} objects is initially logrithmically distributed with the same physical sizes, and the total mass of particles is equal to the total solid mass get from initial gas surface density and dust-to-gas ratio. The object for particles are called **Superparticles** because the particles in code are aggregates of many real particles, respectively, which help us dramatically save the computational power, also that the difference between Lagrangian code and N-body code. After getting all the parts of system, we calculate the time step from **core.new_timestep**, which we will introduce in detail later.

RUNNING

After the initialization, it comes normal running. The running steps are shown in the dotted frame of figure 1. The order is critical for running, let's see them in detail.

1. **Add planet (orbital objects)**. This step is done by **add_planet** function under **System** object. You can choose adding objects at a certain time and certain location or add them from certain planetesimal forming mechanism (TBD).

2. **Update particles**. The particles is from **Superparticles** object, which includes all the 'superparticles' in the disk. The main properties of particles include location, physical mass and total mass. The 'total mass' is the sum of the masses of all particles the **Superparticle** represents.

   The updating is done by **update_particles** under Syetem object, for which the users need to supply how the particles grows in the **dm_dt** function under **userfun.py**. The drift of particles is done by function **radial_v** in \text{main/physics.py} as:

   $v_r = -\frac{St}{1+St^2}\eta v_K$

   in which the St is stokes number, representing the areodynamical size of particles, the η is the ratio between the gas pressure gradient and gravity in disk, and v_K represents Kepler velocity. The composition of particles are also considered, but in our code, the change of composition happens only when cross the corresponding iceline.

3. **Advance iceline and planet**. This step discribe the interaction between particles and icelines/planets (orbital objects) when particles drift across the location of icelines/planets.

   The planets accretes materials from superparticles, decreasing the total mass of superparticles, and increasing the mass of planets, also this mass will change the planets' composition. Users can supply the analitical accretion efficiency algorithm to function **epsilon_PA** in **userfun.py**. And also planets may migrates in disk, which is discribed by function **planet_migration**. For iceline, the iceline will also move, because the change of midplane temperature, which is discribed by function **get_iceline_location** under ICELINE object. The \textbf{advance_iceline} also has the task to change the composition of particles because of evaporation.

4. **Post process.** After update the properties of every part, the **post_process** will do some making up things. There's brief introduction aboout what is done under this function (also
   a comment in the code):
   (a) add and remove particles which cross the inner edge and enter from outer edge.
   (b) resample particle distribution to keep the number of superparticles stable, which has
   two algorithms can be chosen in **Test/users/parameters.py** by change the value of 'resampleMode'. The two algorithms are 'Nplevel' and 'splitmerge', the former one stablize the number by adjust the total mass of superparticles, while the later by spilt and merge superparticles.
   (c) update the particle state vector
   (d) remove planets which are engulted by central body.
   (e) update mass of central body.

5. **Get new time step**. Whether we do the jump or not, we need to get the time step for next
   step. The **new_timestep** is a complex function, inwhich we compare several timescales and let the shortest one to be next time step. And here we also get some 'milestones' for restricting the jump step.
6. **Back up data**. Back up the properties of current state for calculating the timescales.
7. **Print and update log**. Print some things to the terminal and update the log file under **/Test/user/log/** This step is done by the **do_stuff** in **userfun.py**, which needs to be supplied by users.
8. Finally, all of these steps are in a while loop, when the time of system is smaller then tmax
   parameter in **parameters.py.**

FINAL

When the **system.time** larger equal than the tmax in **parameters.py**, the 'final' parameter will be 'True', then the while loop will be stopped, and the system class will be stored as .pickle file.

Under the **/Test/zxl_test/userfun**, there's a 'data' class, which can store and process the data to plot some figures.

JUMP

The jump is a typical programing method aiming at accelerating simulation. Under the principle that the evolution of system will enter into a quasi-static state, during which we can regard the rate of evolution of system properties as fixed values. For example, when the distribution of solid particles in the circumplanetary disk has a very long timescale, the accretion rate of satellites in this disk will not change significantly, then we don't need to calculate the accretion rate every step, which can be in other words 'jumped'. Of course implementing the 'jump' thing need to be cautious and well thought out. So Let's present you how we do the jump in detail.

To get how long we can jump over, we consider several characteristic timescales, including those related to satellite mass increase, satellite migration, iceline position changes, and mass inflow (specifically for CPDs). These timescales can be expressed as follows:
$\tau_P = \frac{P_{new}}{(|P_{old} - P_{new}|)} \Delta t,$
in which P represents the kinds of properties of system, which in our CPD case are the location of icelines and the mass inflow rate.

Also for some special preperties like mass growth of objects in disk, which is not continuous, cannot be calculated like that. For these kinds of properties, we logrithmically fit the points on which mass change, and get the evolving timescale from the slope. For these kinds of properties, we can get how the errors grow when we prolong the jump time, and we artificially supply a parameter serving as the maximum error can be tolerated in jump, from which we can further restrict the jump time. The maximum jump time can be derived as:
$t_{jump,max} = \frac{\eta m}{\sigma_{dm/dt} - \sigma_{max} dm/dt}$
in which the σ_{max} is the maximum error we artificially defined in paramteters.py under every user directory. Then we can choose the minimum time between these timescales and maximum jump time, and restrict it further by just simply multiplying a pre-factor that is smaller than 1 in front of

this time, finally we get the jump time: **tjump** . Also, there are some points that cannot be jumped over, which are called 'milestones', like the time we inserting the objects. Comparing tjump and the time to milestones, we finally get the time we can jump over. After we get how long we can jump over, we also need to know whether we can jump, which is critical because if we do the jump very frequently, we can imagine the error will be large. So we consider the conditions below:

1. The jump step size must be significantly larger than the evolution step size to ensure its significance.
2. The jump step size must be greater than 100 times average steps.
3. Jumps cannot occur continuously; there must be sufficient time between jumps to update
   various physical processes' evolution rates. For instance, we currently require at least 100 steps of regular evolution between jumps.
4. Jumps happen only after the system enter into the quasi-static state. We do this by letting
   all the initially distributed particles are accreted by the central body in our CPD cases. (NOT SUITABLE FOR PPD)

Only when all these conditions are met can a jump occur. Figure 2 illustrates the jump durations in a simulation with 107 years, in which we can found:

1. Throughout the simulation, significant jumps occur, greatly reducing the required computa-
   tion time.
2. The time spent on regular equation solving and other steps between jumps is minimal.
3. Jumps halt when satellites enter the system.
4. Sharp decreases in jump duration correspond to satellite capture into resonances, as expected.
   Since we cannot precisely predict when satellites will enter resonances, we reduce jump time
   near resonance points to avoid skipping them.

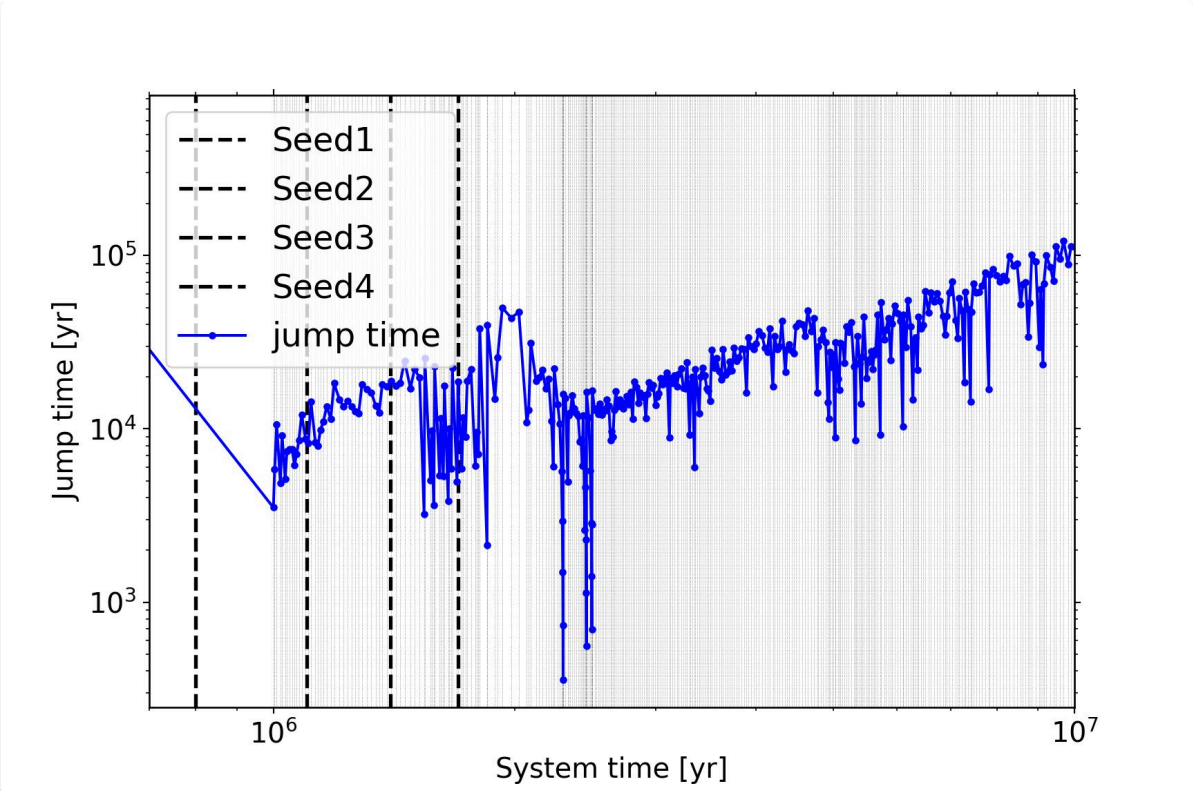

Figure 2: The jump steps through the simulation. The blue line with dots represents the length of jump steps. The black dashed line represents the time points where the satellites enter into the system. The slim dotted lines represents the ends of every times, which is almost overlap with the beginning of next jump.