pNbody

Release 3.0

Yves Revaz

February 2007

Observatoire de Paris
77 av. Denfert-Rochereau
F-75014 Paris
Email: yves.revaz@obspm.fr

Abstract

pNbody is a parallelized python module toolbox desinged to manipulate and display interactively very lage N-body systems.

Its oriented object aproche allows the user to perform complicate manipulation with only very few commands.

As python is an interpreted language, the user can load an N-body system and explore it interactively using the python interpreter. pNbody may also being used in python scripts.

The module also contains graphical facilities desinged to create maps of physical values of the system, like density maps, temperture maps, velocites maps, etc. Stereo capabilities are also implemented.

pNbody is not limited by file format. Each user may redefine in a parameter file how to read its specific format.

Its new parallel (mpi) facilities make it works on computer cluster without being limitted by memery consumption. It has already been tested with 268 millions of particles.

with **gmov**.

CONTENTS

CHAPTER

ONE

Overview

`pNbody` is a parallelized python module toolbox desinged to manipulate and display interactively very lage N-body systems.

Its oriented object aproche allows the user to perform complicate manipulation with only very few commands.

As python is an interpreted language, the user can load an N-body system and explore it interactively using the python interpreter. pNbody may also being used in python scripts.

The module also contains graphical facilities desinged to create maps of physical values of the system, like density maps, temperture maps, velocites maps, etc. Stereo capabilities are also implemented.

pNbody is not limited by file format. Each user may redefine in a parameter file how to read its specific format.

Its new parallel (mpi) facilities make it works on computer cluster without being limitted by memery consumption. It has already been tested with 268 millions of particles.

CHAPTER

TWO

Installation (linux only)

2.1 Prerequiste

The basic module of pNbody needs python and additional packages :

1. Python 2.3 or higher
   `http://www.python.org`
2. numarray 1.4.1 or higher
   `http://www.stsci.edu/resources/software_hardware/numarray`
3. Imaging 1.1.5 or higher
   `http://www.pythonware.com/products/pil/`
4. Python megawidgets 1.2 or higer (only usefull for some graphics capabilities)
   `http://pmw.sourceforge.net/`

For the parallel capabilities, an mpi distribution is needed (lam,mpich) as well as the additional python mpi wrapping:

1. mpi4py
   `http://cheeseshop.python.org/pypi/mpi4py`

In order to convert movies in standard format (gif or mpeg), the two following applications are needed :

1. **convert**
2. **ppmtompeg**

2.2 Installing from tarball

First, uncrompress the distribution file :

```
tar -xzf pNbody-3.0.tgz
cd pNbody-3.0
```

Now, depending on your python installation you have to be root. The module is compiled and installed with the following command :

```
python setup.py install
```

Be sure that `python` is the executable you want to use with `pNbody`.

2.3 Default configuration

De default configuration files are stored in the directory `PYTHON_DIRECTORY/site-packages/pNbody/config`, where `PYTHON_DIRECTORY` is the directory of your python installation and usually looks like `/usr/lib/pythonx.y/`, where the `x.y` corresponds to your python version.

It is recommanded that the user copy in `/.Nbody` the content of this default directory, in order to have its own configurations files.

The content of the `config` directory is :

parameter	**Type**	**Meaning**
`formats`	directory	this directory contains specific class definition files used to read different formats
`rgb_tables`	directory	this directory contains all colors tables accessibles by `pNbody`
`defaultparameters`	file	this file contains the default graphical parameters used by `pNbody`
`unitsparameters`	file	this file contains the default units parameters used by `pNbody`

2.4 Documentation and examples

Documentation of the package in postcript form is available in the directory `doc` of the root instalation directory.

In order to test examples proposed in the tutorial, you are advised to copy the whole `examples` directory somewere in you own directories. In this document, each time an example (script or commands) is proposed, we assume that the user is in the `examples` directory.

CHAPTER

THREE

Tutorial

pNbody may be used either interactively (in this case, the parallel capabilities are yet not available) in the python console, either in scripts. Interactive parallel capabilities are available using the **gdisp** application described later in this document.

3.1 Using pNbody with the python interpreter

In this section, we assume that pNbody in installed in /usr/lib/python2.4/site-packages/pNbody/. First, copy to the examples directory somewere in you own directories and start python :

```
[lunix@lunix ~]$ cd examples
[lunix@lunix examples]$ python
Python 2.4.2 (#2, Jul 13 2006, 15:26:48)
[GCC 4.0.1 (4.0.1-5mdk for Mandriva Linux release 2006.0)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Now, you can load the pNbody module :

```
>>> from pNbody import *
```

We can first start by creating a default pNbody objet and get info about it :

```
>>> nb = Nbody()
>>> nb.info()
----------------------------------
particle file    : ['file.dat']
ftype            : 'Nbody_default'
nbody            : 0
nbody_tot        : 0
npart            : array([0])
npart_tot        : array([0])
mass_tot         : 0.0
byteorder        : 'little'
pio              : 'no'
```

All variables linked to the object *nb* are accesible by typing *nb.* followed by the associated variables :

```
>>> nb.nbody
0
>>> nb.mass_tot
0.0
>>> nb.pio
'no'
```

Now, you can create an object by giving the positions of particles :

```
>>> pos = ones((10,3),Float32)
>>> nb = Nbody(pos=pos)
>>> nb.info()
----------------------------------
particle file    : ['file.dat']
ftype            : 'Nbody_default'
nbody            : 10
nbody_tot        : 10
npart            : array([10])
npart_tot        : array([10])
mass_tot         : 1.0
byteorder        : 'little'
pio              : 'no'

len pos             : 10
pos[0]              : array([ 1.,  1.,  1.], type=Float32)
pos[-1]             : array([ 1.,  1.,  1.], type=Float32)
len vel             : 10
vel[0]              : array([ 0.,  0.,  0.], type=Float32)
vel[-1]             : array([ 0.,  0.,  0.], type=Float32)
len mass            : 10
mass[0]             : 0.10000000149011612
mass[-1]            : 0.10000000149011612
len num             : 10
num[0]              : 1
num[-1]             : 10
rsp               : None
```

In this case, you can see that the class automatically intitialize other arrays variables (*vel*, *mass*, *num* and *rsp*) with default values. Only the first and the last element of each defined vector are displyed by the methode *info*. All defined arrays and array elements may be easily accessible using the `numarray` convensions. For exemple, to display and change the positions of the tree first particles, type :

```
>>> nb.pos[:3]
array([[ 1.,  1.,  1.],
       [ 1.,  1.,  1.],
       [ 1.,  1.,  1.]], type=Float32)
>>> nb.pos[:3]=2*ones((3,3),Float32)
>>> nb.pos[:3]
array([[ 2.,  2.,  2.],
       [ 2.,  2.,  2.],
       [ 2.,  2.,  2.]], type=Float32)
```

Now, lets try to open the gadget snapshot `gadget_z00.dat`. This is achieved by typing :

```
>>> nb = Nbody('gadget_z00.dat',ftype='gadget')
```

Again, informatins on this snapshot may be obtained using the instance *info()* :

```
>>> nb.info()
-----------------------------------
particle file    : ['gadget_z00.dat']
ftype            : 'Nbody_gadget'
nbody            : 20560
nbody_tot        : 20560
npart            : array([ 9160, 10280,     0,     0,  1120,     0])
npart_tot        : array([ 9160, 10280,     0,     0,  1120,     0])
mass_tot         : 79.717887878417969
byteorder        : 'little'
pio              : 'no'

len pos             : 20560
pos[0]              : array([-1294.48828125, -2217.09765625, -9655.49609375], type=Float32)
pos[-1]             : array([ -986.0625    , -2183.83203125,  4017.04296875], type=Float32)
len vel             : 20560
vel[0]              : array([ -69.80491638,   60.56475067, -166.32981873], type=Float32)
vel[-1]             : array([-140.59715271,  -66.44669342,  -37.01613235], type=Float32)
len mass            : 20560
mass[0]             : 0.0010856521548703313
mass[-1]            : 0.0010856521548703313
len num             : 20560
num[0]              : 21488
num[-1]             : 1005192

atime               : 1.0
redshift            : 2.22044604925e-16
flag_sfr            : 1
flag_feedback       : 1
nall                : [ 9160 10280     0     0  1120     0]
flag_cooling        : 1
num_files           : 1
boxsize             : 100000.0
omega0              : 0.3
omegalambda         : 0.7
hubbleparam         : 0.7
flag_age            : 0
flag_metals         : 0
nallhw              : [0 0 0 0 0 0]
flag_entr_ic        : 0
critical_energy_spec: 0.0

len u               : 20560
u[0]                : 6606.63037109
u[-1]               : 0.0
len rho             : 20560
rho[0]              : 7.05811936674e-11
rho[-1]             : 0.0
len rsp             : 20560
rsp[0]              : 909.027587891
rsp[-1]             : 0.0
len erd             : 20560
erd[0]              : 446292.5625
erd[-1]             : 0.0
```

You can obtain informations on physical values, like the center of mass or the total angular momentum vector by typing :

```
>>> nb.cm()
array([-1649.92650646,   609.98258649, -1689.04021837])
>>> nb.Ltot()
array([-1112071.75 ,  -755970.625, -1536666.25 ], type=Float32)
```

In order to visualise the model in position space, it is possible to generate a surface density map of it using the *display* instance :

```
>>> nb.display(size=(10000,10000),shape=(256,256),palette='light')
```

You can now performe some operations on the model in order to explore a specific region. First, translate the model in position space :

```
>>> nb.translate([3125,-4690,1720])
>>> nb.display(size=(10000,10000),shape=(256,256),palette='light')
>>> nb.display(size=(1000,1000),shape=(256,256),palette='light')
```

Ou can now rotate around

```
>>> nb.rotate(angle=pi)
>>> nb.display(size=(1000,1000),shape=(256,256),palette='light')
```

You can now display a temperature map of the model. First, create a new object with only the gas particles.

```
>>> nb_gas = nb.select('gas')
>>> nb_gas.display(size=(1000,1000),shape=(256,256),palette='light')
```

now, display the temperture mass-weighted map :

```
>>> nb_gas.display(size=(1000,1000),shape=(256,256),palette='rainbow4',mode='T',filter_name='co
```

You can select only particles within a radius smaller tha 500 (in user units) with respect to the center

```
>>> nb_sub = nb.selectc((nb.rxyz()<500))
>>> nb_sub.display(size=(1000,1000),shape=(256,256),palette='light')
```

Now, rename the new model and save it.

```
>>> nb_sub.rename('gadget_z00_sub.dat')
>>> nb_sub.write()
```

A new gadget file has been created and saved in the current directory. We can now select particles as a function of the temperature. First, display the maximum temperature among all gas particles, then selectc particles and finally save in '`T11.num`' the identifier (variable `num`) of these particles.

```
>>> log10(max(nb_gas.T()))
12.870792505729391
>>> nb_sub = nb_gas.selectc( (nb_gas.T()>1e11) )
>>> nb_sub.write_num('T11.num')
```

Now open a new snapshot, from the same simulation, but at different redshift and find the particles in previous snapshot with temperature higher than 10^{11}:

```
>>> nb = Nbody('gadget_z40.dat',ftype='gadget')
>>> nb.display(size=(10000,10000),shape=(256,256),palette='light')
>>> nb_sub = nb.selectp(file='T11.num')
>>> nb_sub.display(size=(10000,10000),shape=(256,256),palette='light')
```

Now, instead of saving it in a gadget file, save it in a binary file type. You simply need to call the *set_ftype* instance before saving it :

```
>>> nb = nb.set_ftype('binary')
>>> nb.rename('binary.dat')
>>> nb.write()
```

As a last example, we show how two `pNbody` models can be easyly merged with only 11 lines.

```
>>> nb1 = Nbody('disk.dat',ftype='gadget')
>>> nb2 = Nbody('disk.dat',ftype='gadget')
>>> nb1.rotate2(angle=pi/4,axis=[0,1,0])
>>> nb1.translate([-150,0,0])
>>> nb1.vel = nb1.vel + [50,0,0]
>>> nb2.rotate2(angle=pi/4,axis=[1,0,0])
>>> nb2.translate([+150,0,50])
>>> nb2.vel = nb2.vel - [50,0,0]
>>> nb3 = nb1 + nb2
>>> nb3.rename('merge.dat')
>>> nb3.write()
```

Now display the result from different point of view :

```
>>> nb3.display(size=(300,300),shape=(256,256),palette='lut2')
>>> nb3 = nb3.select('disk')
>>> nb3.display(size=(300,300),shape=(256,256),palette='lut2',view='xz')
>>> nb3.display(size=(300,300),shape=(256,256),palette='lut2',view='xy')
>>> nb3.display(size=(300,300),shape=(256,256),palette='lut2',view='yz')
>>> nb3.display(size=(300,300),shape=(256,256),palette='lut2',xp=[-100,0,0])
```

or save it into a gif file :

```
>>> nb3.display(size=(300,300),shape=(256,256),palette='lut2',xp=[-100,0,0],save='image.gif')
```

3.2 Using pNbody with scripts

Instead of using `pNbody` in the python interpreter, it is very usefull to use `pNbody` in scripts. Usually a python script begin by the line `#!/usr/bin/env python` and must be executable (`chmod a+x file.py`). As an example (`slice.py`), we show how to write a script that opens a gadget file, select gas and cut a thin slice ($-1000 < y < 1000$). The new files are saved using the extention `.slice`.

```
#!/usr/bin/env python

import sys
from pNbody import *

files = sys.argv[1:]

for file in files:
  print "slicing",file
  nb = Nbody(file,ftype='gadget')
  nb = nb.select('gas')
  nb = nb.selectc((fabs(nb.pos[:,1])<1000))
  nb.rename(file+'.slice')
  nb.write()
```

You can run this script with the command

```
[lunix@lunix ~]$ ./scripts/slice.py gadget_z*0.dat
```

or

```
[lunix@lunix ~]$ python ./scripts/slice.py gadget_z*0.dat
```

In the current version of `pNbody`, scripts may also be used in parallel, if `mpi` and `mpi4py` is installed. In this case, run the script with a command like (depending of your mpi implementation):

```
[lunix@lunix ~]$ mpirun -np 2 slice.py gadget_z*0.dat
```

In this script only the processus of rank 0 open the file. It then broadcast the particles among all the other processors. The selection of gas and the slice are preformed by all processors. Finally, the `nb.write()` command will gather all particles and write the output file.

Instead of opening one file and writing another, one can ask that every processus open one file and write one. First, modify the previous script by adding the command `nb.set_pio('yes')` The script `split.py` demonstrate this capabilites.

```
#!/usr/bin/env python

import sys
from pNbody import *

files = sys.argv[1:]

for file in files:
  nb = Nbody(file,ftype='gadget')
  nb.set_pio('yes') # enable parallel input/output
  nb.write()
```

and run

```
[lunix@lunix ~]$ mpirun -np 2  ./scripts/split.py gadget_z*0.dat
```

Every file is opened by the processus of rank 0, but now, during the `nb.write` command every processus writes its own file. The files have the same name than the name given in `Nbody()` with an extention `.i` where `i` corresponds to the processus rank.

Now that you have created two output, try to read them. In the script `slice.py`, add a `pio='yes'` in the object argument.

```
#!/usr/bin/env python

import sys
from pNbody import *

files = sys.argv[1:]

for file in files:
  print "slicing",file
  nb = Nbody(file,ftype='gadget',pio='yes')
  nb = nb.select('gas')
  nb = nb.selectc((fabs(nb.pos[:,1])<1000))
  nb.rename(file+'.slice')
  nb.write()
```

and run :

```
[lunix@lunix ~]$ mpirun -np 2  ./scripts/slice2.py gadget_z*0.dat
```

Now, the script works fully in parallel. Every processus reads, writes and works on its own file, correspondig to a subset of the total number of particles.

Lets try two other scripts. The first one try to find the radial maximum distance among all particles and the center. It illustrate the difference between using `max()` wich gives the local maximum (maximum among particles of the node) and `mpi.mpi_max()` which gives the global maximum among all particles.

```
#!/usr/bin/env python

import sys
from pNbody import *

file = sys.argv[1]

nb = Nbody(file,ftype='gadget',pio='yes')
local_max  = max(nb.rxyz())
global_max = mpi.mpi_max(nb.rxyz())

print "proc %d local_max = %f global_max = %f"%(mpi.ThisTask,local_max,global_max)
```

When running it, you should get :

```
[lunix@lunix examples]$ mpirun -np 2 ./scripts/findmax.py gadget_z00.dat
proc 0 local_max = 12070.458008 global_max = 12757.492188
proc 1 local_max = 12757.492188 global_max = 12757.492188
```

which illustrate clearly the point. Finally, the latter script shows that even graphical functions support parallelisme. The script `showmap.py` illustrate this point by computing a map of the model :

```
#!/usr/bin/env python

import sys
from pNbody import *

file = sys.argv[1]

nb = Nbody(file,ftype='gadget',pio='yes')
nb.display(size=(10000,10000),shape=(256,256),palette='light')
```

When running :

```
[lunix@lunix examples]$ mpirun -np 2 ./scripts/showmap.py gadget_z00.dat
```

you get an image of the model.

3.3 Using pNbody interactively in parallel

`mpi4py` does not currently works with the standard python interpreter. This prevents us to works whith models interactively in parallel. However, using **`gwin`**, while displaying your model in a window, you can interact with it in parallel. Try to run (here, we assume you are using mpich-2 as mpi implementation):

```
[lunix@lunix examples]$ mpirun -np 2 gwin -t gadget gadget_z00.dat
```

A window opens, displaying a map of your model. With your mouse, you can interact with the model. While pressing the left button of your mouse, move it on the map. The red axis at the center moves and determines the observer orientation. A right click redisplays the model. You will find a more complete documentation of **`gwin`** futher in this document. Here, weonly focus on some parallel features. Try `node info` on the `Operations` menu. You

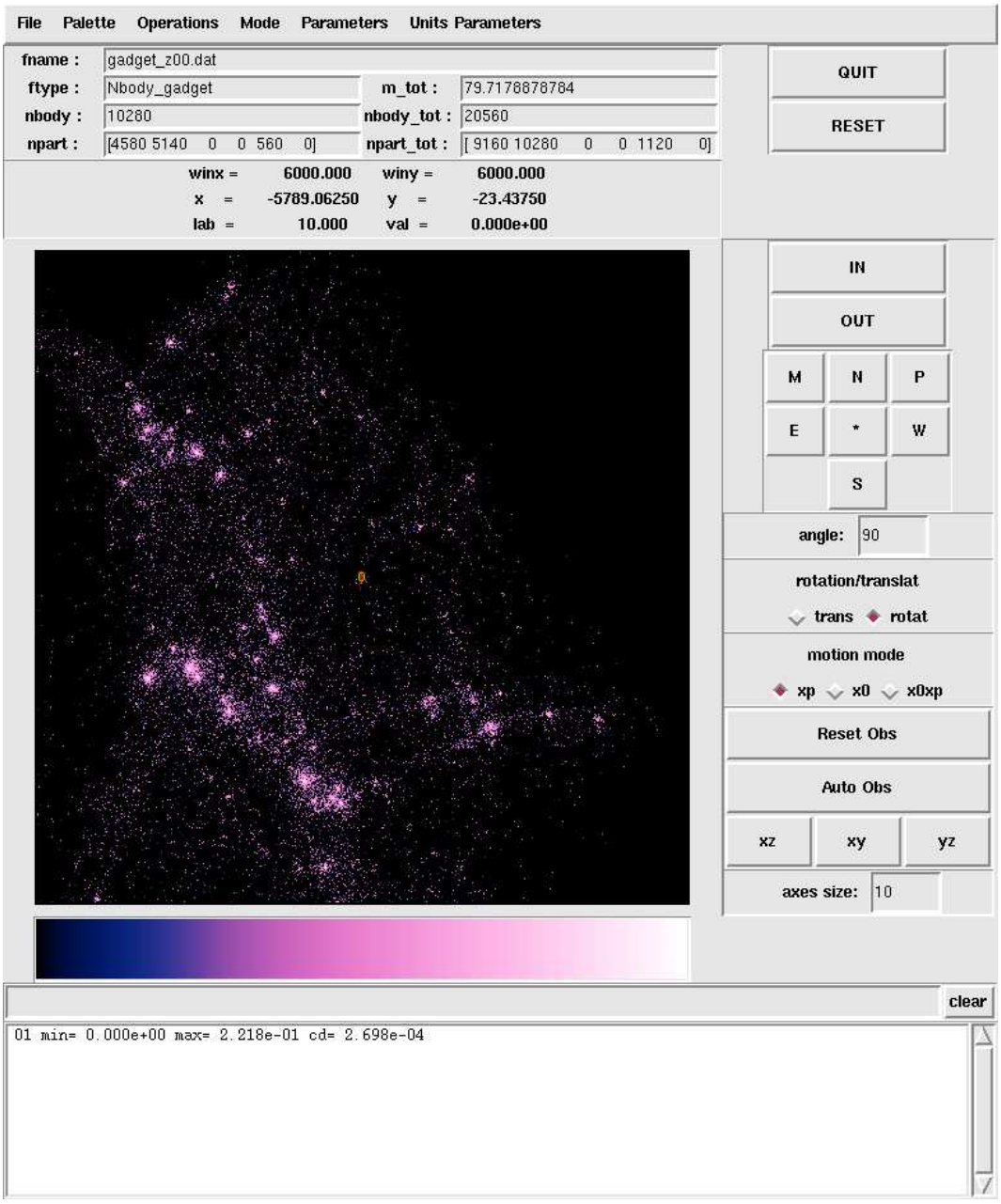

Figure 3.1: Snapshot of gwin.

get the number of particles contained by each processus as well as the number particles of each type. Now, we can interact with the model using the bottom entry, at the left of the `clear` button. The commands you can entrer here are similar to the one used in the python console or in python scripts. The only (but major difference) is that the variable representing the model object is called `self.nb` instead of `nb`. Try for example :

```
self.nb.translate([3125,-4690,1720])
```

and then press the right button of the mouse. The model is now center on $(-3125, 4690, -1720)$. Now select a small region around this point :

```
self.nb = self.nb.selectc((self.nb.rxyz()<500))
```

Now, if you display the number of particles per processus, using `node info` on the `Operations` menu, you probably see processus with much more particles. You can redistribute the number of particles in order to increase the preformaces, with the command `redistribute particles` on the `Operations` menu.

CHAPTER

FOUR

Minimal pNbody object

A minimal `pNbody` object consists of a set of 4 numarray vectors :

- *pos*: positions of particles ($n \times 3$ Float32 Numeric array)
- *vel*: velocities of particles ($n \times 3$ Float32 Numeric array)
- *mass*: mass of particles (n Float Numeric array)(default=None)
- *num*: identifier of particles (n Int Numeric array)(default=None)

and a set of default associated variables :

- *npart*: number of particles of each type (List), for the current node.
- *nbody*: number of particles for the current node (sum of *npart*)
- *npart_tot*: number of particles of each type (List), summing over all nodes.
- *nbody_tot*: total number of particles (sum of *npart_tot*)
- *mass_tot*: total mass of the system

The variable *npart* gives the list of particles of each type. For example

```
>>> nb.npart
array([10, 20,  0, 10])
```

means that there are 4 particle types, 10 particles belongs to the first type, 20 to the second, 0 to the third and 10 to the fourth.

The value of the following variables are automatically set during intitialisation (using the values of *npart*,*nbody* and *nb.mass*), even if they are defined before (during the reading process) :

- *nbody* :
- *npart_tot* : **`get_npart_tot()`**
- *nbody_tot* : **`get_nbody_tot()`**
- *mass_tot* : **`get_mass_tot()`**

The user is free, when reading a specific file format, to add specific variables (for example, the time of the snapshot) and specific numarray vectors (density, specific energy, etc.). See the formats section for more details.

CHAPTER

FIVE

Formats

Thanks to the interpreted facilities of python, `pNbody` may read any file format. You simply have to decribe it in a python file placed in your `/.Nbody/formats` directory. This python file declares a new class that inherit all atributes of the default `pNbody` class.

As an example, we will describe here after, how to write a file that reads the format written by the following Fortan instructions :

```
INTEGER   NBODY,NGAS,NSTARS
REAL*4   TNOW
REAL*4   POS(3,MXBODY),VEL(3,MXBODY),MAS(MXBOFY), MET(MXBOFY)
INTEGER   K,P

OPEN (UNIT=1,FILE='simpleformat.dat',FORM='UNFORMATTED', STATUS='UNKNOWN')
WRITE (1) TNOW,NGAS,NSTAR
WRITE (1) ((POS(K,P), K=1,3),P=1,NBODY)
WRITE (1) ((VEL(K,P), K=1,3),P=1,NBODY)
WRITE (1) (MAS(P),       P=1,NBODY)
WRITE (1) (MET(P),       P=1,NGAS)
CLOSE(UNIT=1)
```

The file consists of a header block (`TNOW`,`NGAS`,`NSTAR`), a position block of size 3 `NBODY`, a velocity block of size 3 `NBODY`, a mass block of size `NBODY` and a metallicity block of size `NGAS`.

5.1 Some init

In order to read this file, we have to create a format file : `/.Nbody/formats/simpleformat.py`. First add the following lines that tells `pNbody` which functions to use when reading and writing a file. Here, the functions will be `self.read_particles` and `self.write_particles`.

```
class Nbody_simpleformat(NbodyDefault):

  def get_read_fcts(self):
    return [self.read_particles]

  def get_write_fcts(self):
    return [self.write_particles]
```

Now, if you need it, specify specific variables and their default values, with the function `get_spec_vars`

```
def get_spec_vars(self):
  '''
  return specific variables default values for the class
  '''
  return {'tnow'   :0.,
          'ngas'   :0,
          'nstars' :0
  }
```

5.2 The read function

Now, its time to write the `read_particles` functions.

```
def read_particles(self,f):

  tpl = (Float32,Int32,Int32)
  tnow, ngas, nstar = io.ReadBlock(f,tpl,byteorder=self.byteorder,pio=self.pio)
```

The variable *tpl* describe the type of the data contained in the header, then, use the function **`io.ReadBlock()`** to read the header and send it to other nodes, if necessary (if *self.pio*='no').

Now, before reading the vectors blocks, we have to define how particles, depending on their type, will be distributed among nodes. First, create the *npart_read* particle, that corresponds to *npart* read from the header. Then simply use **`get_npart_and_npart_all()`** to create *npart* and *npart_all*. *npart* gives you the number of particles of each type present in the local node. *npart_all* is a list of *npart* corresponding to all nodes. Thanks to this latter variable, the **`io.ReadArray()`** function will be able to read data blocks and send the right particles to the right nodes, taking into account the type of the particles.

```
  npart_read = array([ngas,nstar])
  npart,npart_all       = self.get_npart_and_npart_all(npart_read)
```

Now, as the metallicity block contains informations only for gas particles, we have to compute an equivalent of *npart_all* but only for the first type.

```
  npartgas,npartgas_all = self.get_npart_and_npart_all(npart_read[0])
```

Now, it is usefull to compute the specific local variables *ngas*,*nstar* and *nbody*.

```
  # now, get the corect values
  ngas  = npart[0]
  nstar = npart[1]
  nbody = sum(npart)
```

It is time now to read the position, velocities, mass and metallicity blocks, by giving to the function **`io.ReadArray()`** the right particle distribution (*npart_all* or *npartgas_all*). The `self.` in front of the variables is important to make these vectors global. In the last line, we complete the vector *self.meta* by adding zeros, in

order to have a value for each particles (even for stars).

```
self.pos = io.ReadArray(f,Float32,shape=(nbody,3),byteorder=self.byteorder,pio=self.pio,nlo
self.vel = io.ReadArray(f,Float32,shape=(nbody,3),byteorder=self.byteorder,pio=self.pio,nlo
self.mass= io.ReadArray(f,Float32,               byteorder=self.byteorder,pio=self.pio,nlo
self.meta= io.ReadArray(f,Float32,               byteorder=self.byteorder,pio=self.pio,nlo
self.meta  = concatenate((self.meta,zeros(nbody-npart[0]).astype(Float32)))
```

Finally, invoce the two following commands to ensure that all variables will be accessible in the main class.

```
self.make_default_variables_global(vars())
self.make_specific_variables_global(vars())
```

5.3 The write function

The `write_particles` functions begins to gather all npart. Then depending if we will write in parallel (*self.pio* = 'yes') or not, we have to reduce or not the values of *ngas* and *nstar*. This is important to write a consistent header.

```
def write_particles(self,f):

  npart_all = array(mpi.mpi_Allgather(self.npart))

  if self.pio == 'yes':
    ngas  = self.ngas
    nstar = self.nstar
  else:
    ngas  = mpi.mpi_reduce(self.ngas)
    nstar = mpi.mpi_reduce(self.nstar)
```

The header is written with the function **`io.WriteBlock()`** using the variables *tpl* telling what to write with wich format.

```
tpl = ((self.tnow,Float32),(ngas,Int32),(nstar,Int32))
io.WriteBlock(f,tpl,byteorder=self.byteorder)
```

And finally, write the vectors, using the function **`io.WriteArray()`**

```
io.WriteArray(f,self.pos.astype(Float32),byteorder=self.byteorder,pio=self.pio,nlocal=npart
io.WriteArray(f,self.vel.astype(Float32),byteorder=self.byteorder,pio=self.pio,nlocal=npart
io.WriteArray(f,self.mass.astype(Float32),byteorder=self.byteorder,pio=self.pio,nlocal=npar
io.WriteArray(f,self.meta[:self.npart[0]].astype(Float32),byteorder=self.byteorder,pio=self
```

5.4 Adding specific functions to the class

Here we describe how we can add specific functions to the new class defined in `simpleformat.py`. This may be usefull if you want for example to get specific informations on the class. You can simplay redefine the **`spec_info()`**

functions that is called automatically by the **`info()`**.

```
def spec_info(self):
  """
  Write spec info
  """
  infolist = []
  infolist.append("")
  infolist.append("tnow                : %s"%self.tnow)
  infolist.append("ngas                : %s"%self.ngas)
  infolist.append("nstar               : %s"%self.nstar)
  return infolist
```

Other functions may be defined, for example, a function that simplay return the metallicity of each particles.

```
def Metallicity(self):
  """
  Return the metallicity
  """
  return self.meta
```

With this function, you can easilly display the metallicity value of each particles in a plot, witout redefining anything else (See further).

5.5 Example

Try :

```
[lunix@lunix ~/examples]$ gwin -t simpleformat simpleformat.dat
```

CHAPTER

SIX

Parameters

The module `pNbody` uses two sets of parameters. Default values for these parameters are stored in two files : `/.Nbody/defaultparameters` (display parameters) and `/.Nbody/unitsparameters` (units parameters).

6.1 Display parameters

The list and the meaning of display parameters is given in the following table :

parameter	Type	Meaning
obs	ArrayObs	Observer matrix
xp	List	Observing position
x0	List	Position of observer
alpha	Float	Angle of head
view	String	Projection (”xy”,”yz”,”xz”)
r_obs	Float	Distance to the observer
eye	String	Name of the eye (”left” or ”right”) (stereo mode)
dist_eye	Float	Distance between the eyes (stereo mode)
foc	Float	Focal distance (stereo mode)
persp	String	”on”=frustum projection ”off”=orthogonal projection
clip	Tuple	Clip plane (near,far)
cut	String	Remove particles outisde clip plane (”yes” or ”no”)
shape	Tuple	Shape of the image in pixels (nw,nh)
size	Tuple	Physical half size of the image (dx,dy)
space	String	Working space, ”pos”=position space ”vel”=velocity space
mode	String	Physical value to plot
frsp	Float	Factor for gas smoothing
filter_name	String	Name of the image filter
filter_opts	List	Options of the filter
scale	String	”lin”=linear scaling ”log”=logarithmic scaling
cd	Float	Logarithmic scaling parameter (use for logarithmic scaling)
mn	Float	Minimum value of the map (use for the scaling)
mx	Float	Maximum value of the map (use for the scaling)
l_n	Int	Number of contour levels
l_min	Float	Minimum value of contours
l_max	Float	Maximum value of contours
l_kx	Int	Contour precision in x
l_ky	Int	Contour precision in y
l_color	Int	Contour color (0-255)
l_cruch	String	Remove background map when plotting contours (”yes” or ”no”)
b_weight	Int	Box line width (0 or 1)
b_xopts	Tuple	Ticks box parameters in x
b_yopts	Tuple	Ticks box parameters in y
b_color	Int	Box color (0-255)

All these parameters may be displayed or set using the methods :

```
>>> nb.parameters.get('mode')
>>> nb.parameters.set('mode','T')
```

6.1.1 How to define the observer position ?

When creating an image from a model, one has to choose the observer position, the look at point and the orientation of the head. The user has tree possibilities to define these parameters :

1. Define manually the observer matrix *obs.obs* is a 4×3 array matrix. Meaning of the four vectors composing this matrix is given in the following table :

obs	**Meaning**
obs[0]	position of the observer
obs[1]	position of the look at point (with respect to the position of the observer)
obs[2]	position of the head (with respect to the position of the observer)
obs[3]	position of the arm (with respect to the position of the observer)

2. Using the parameters *xp*, *x0* and *alpha*, where *x0* is the observer position, *xp* is the look at point and *alpha* the angle between the head and the z axis.

3. Using the parameters *view* and *r_obs*. *view* can be equal to 'xy', 'xz' or 'yz', the projection being parallel to one of the main axis. *r_obs* gives the distance between the observer and the look at point.

6.1.2 Stereo mode

When creating stereo view, you can specify the eye you are looking with ('right','left'). When exposition the model with the method **expose**, the observer will be rotate around an axis parallel to its head, with a center rotation in the direction of the look at point, at a distance *foc* of the eye. The angle of the rotation is such as to move the observer of a distance (*dist_eye*)/2.

6.1.3 Perspective

You have two possibilties for the perspective view. If *persp*='on', the model is projectec using a frustrum projection matrix. In the other case, it uses an ortho matrix (orthogonal projection). The near and far clipping planes are given by the parameter *clip*. The left, right, bottom and top clipping planes are given by the parameter *size*. If *cut* is set to 'yes', particles outside the box defined by the 6 planes are not displayed.

6.2 Units parameters

The new version of `pNbody` also uses a default parameters used for units conversion. The list and the meaning of these parameters is given in the following table :

parameter	**Type**	**Meaning**
xi	Float	Hydrogen mass fraction
ionisation	Int	Ionisation flag (ionized gas = 1, neutral gaz = 0)
metalicity	Int	Metalicity index (between 0 and 5), used for cooling function
coolingfile	String	Path to the file containing the tabulated cooling function
Nsph	Int	Number of sph neighbors (unused)
Cosmo	String	use cosmological units ('yes' or 'no')
OmegaLambda	Float	Omega Lambda
Omega0	Float	Omega matter
UnitLength_in_cm	Float	Model units lenght in cm
UnitMass_in_g	Float	Model units mass in g
UnitVelocity_in_cm_per_s	Float	Model units velocity in cm/s

All these parameters may be displayed or set using the methods :

```
>>> nb.unitsparameters.get('HubbleParam')
>>> nb.unitsparameters.set('HubbleParam',0.73)
```

CHAPTER

SEVEN

Main module functions

These part describe all instances contained in an `pNbody` object.

7.1 Io functions

`Nbody`([*p_name=None*], [*pos=None*], [*vel=None*], [*mass=None*], [*num=None*], [*npart=None*], [*ftype=None*], [*status='old'*], [*byteorder=sys.byteorder*], [*pio='no'*], [*log=None*])

This instance is the constructor for the `pNbody` object. Optional arguments are:

- *p_name*: name of the file or list of files containing the data (default=None)
- *pos*: positions of particles ($n \times 3$ Float32 Numeric array)(default=None)
- *vel*: velocities of particles ($n \times 3$ Float32 Numeric array)(default=None)
- *mass*: mass of particles (n Float Numeric array)(default=None)
- *num*: identifier of particles (n Int Numeric array)(default=None)
- *npart*: Number of particles of each type (List) (default=None)
- *ftype*: type of the *p_file* (default=None)

Value	Meaning
`"binary"`	standard format of the Geneva Observatory Nbody files
`"gadget"`	gadget file

- *status*: status of the file, *"new"* or *"old"* (default=*"old"*)
- *byteorder*: byte order "littelendian" or "bigendian" (default=sys.byteorder)
- *pio*: parallel input/output (default="no")
- *log*: pointer to log ouput object (default=None)

If *status* = 'old', the constructor automatically call the function **`read`** wich will read the content of the file(s) in the *p_name*. If *status* = 'new', defaut values are given to the variables. One the constructor has been called, all optional varables are are initialized. The following variables are also initialized :

- *nbody*: number of particles for the current node
- *nbody_tot*: total number of particles
- *npart_tot*: total number of particles of each type
- *mass_tot*: total mass of the system

Other variables may be initialized, depending on the type of the file read.

```
>>> from pNbody import *
>>> nb = Nbody('disk.gad',ftype='gadget')
>>> nb.pos[0:10]
array([[  4.89041433e-02,   2.52863064e+01,  -7.45196819e+00],
       [  5.82345047e+01,  -5.67267685e+01,   1.64647083e+01],
       [  1.17894821e+01,  -5.01083970e-01,   3.68080497e+00],
       [ -2.43042755e+01,  -2.24240952e+01,   5.40977783e+01],
       [ -1.03210754e+01,   8.96411300e-01,  -5.02210388e+01],
       [ -1.38344803e+01,  -2.56113482e+00,  -1.29380960e+01],
       [  1.84468384e+01,  -3.16423264e+01,   2.17839069e+01],
       [  6.11360121e+00,   1.42906122e+01,   3.05427265e+01],
       [ -7.02638292e+00,   2.44790745e+01,   9.55757523e+00],
       [  8.33313751e+00,  -5.49589968e+00,  -4.00754318e+01]], type=Float32)
>>> nb.mass_tot
2.3262434005737305
>>> nb.npart
array([   0, 2001, 1000,    0,    0,    0])
```

7.2 Init functions

init()
Initialize a model. Ensure that all variables are well defined.

set_ftype([*ftype='binary'*],[*npart=None*])
Return a new object with a type defined by *ftype*.

```
>>> from pNbody import *
>>> nb = Nbody('disk.gad',ftype='gadget')
>>> nb.ftype
'Nbody_gadget'
>>> nb = nb.set_ftype('binary')
>>> nb.ftype
'Nbody_binary'
```

get_num()
Compute the *num* variable in order to be consistent with npart.

get_spec_vars()
Return specific variables default values for the class

set_pio(*pio*)
Set parallel input/output or not io. *pio* may be "yes" or "no".

rename(*p_name*)
Rename the file(s). *p_name* may be a string or a list of string, in case of multiple files.

set_filenames(*p_name*,[*pio=None*])
Set the local and global names, in case of parallel input/output.

print_filenames()
Print files names

get_massarr_and_nzero()
Return the *massarr* and *nzero* variables, computed from *mass*.

7.3 Parameters functions

set_parameters(*params*)
Set parameters for the class.

set_unitsparameters(*params*)
Set units parameters for the class.

set_local_system_of_units()
Set local system of units

7.4 Informations functions

info()
Write informations about the object variables content.

spec_info()
Write specific info.

object_info()
Write class(object) info.

nodes_info()
Write info on nodes.

memory_info()
Write info on memory size of the current object (only counting arrays size)

get_list_of_array()
Return the list of numarray vectors of size nbody.

get_list_of_method()
Return the list of instance methods (functions).

get_list_of_vars()
Get the list of vars that are linked to the model.

has_array(*name*)
Return true if the object `pNbody` has an array called `self.name`

find_vars(*name*)
This function return a list of variables defined in the current object.

7.5 Read and write functions

read()
Read the file(s) defined by *p_name*. If the values of the files is *None*, they are simply ignored.

open_and_read(*name,readfct*)
Open and read the file *name*, using the reading function *readfct*.

write()
Write the file(s) defined by *p_name*. If the values of the files is *None*, they are simply not written.

open_and_write(*name,writefct*)
Open and write the file *name*, using the wrinting function *readfct*.

write_num(*name*)
Write the content of the *num* in a file *name*.

read_num(*name*)
Not implemented yes.

7.6 Physical values and operations

rxyz()
Return a 1xn Float array that corresponds to the distance from the center of each particle.

phi_xyz()
Return a 1xn Float array that corresponds to the azimuth in spherical coordinate of each particle.

theta_xyz()
Return a 1xn Float array that corresponds to the elevation angle in spherical coordinate of each particle.

rxy()
Return a 1xn Float array that corresponds to the projected distance from the center of each particle.

cart2sph()
Transform carthesian coodinates x,y,z into spherical coordinates r,p,t Return a 3xn Float array.

cart2sph([*pos=None*])
Transform carthesian coodinates x,y,z into spherical coordinates r,p,t Return a 3xn Float array.

sph2cart([*pos=None*])
Transform spherical coordinates r,p,t into carthesian coodinates x,y,z Return a 3xn Float array.

vrxyz()
Return a 1xn Float array that corresponds to the radial velocity in spherical system

Vr()
Return the radial velocies of particles The output is an 3xn Float array.

Vt()
Return the tangential velocies of particles The output is an 3xn Float array.

get_nt()
Return the total number of particles.

get_ns()
Return in an array the number of particles of each node.

get_mass_tot()
Return the total mass of system.

get_npart_tot()
Return the total nuber of particles of each types.

cm()
Return the mass center of the model. The output is an 3×1 Float Numeric array.

get_histocenter([*rbox=50*],[*nb=500*])
Return the position of the higher density region found by the function **histocenter**.

- *rbox*: box dimension, where to compute the histograms (default=*50*)
- *nb*: number of bins for the histograms (default=*500*)

cv()
Return the barycenter of the velocities of the model. The output is an 3×1 Float Numeric array.

minert()

Return the diagonal of the inertial momentum I. Each component of the diagonal is defined as:

$$Ij = \sqrt{\frac{1}{M} \sum mi \cdot xj^2}, \tag{7.1}$$

where $j = x, y, z$ and M is the total mass. i runs over all particles. The output is an 3×1 Float Numeric array.

x_sigma()
Return the norm of the position dispersions.

v_sigma()
Return the norm of the velocity dispersions σ :

$$\sigma = \sqrt{\frac{1}{M} \sum mi \cdot ((vxi - vcx)^2 + (vyi - vcy)^2 + (vzi - vcz)^2)}, \tag{7.2}$$

where $[vcx, vcy, vcz]$ is the barycenter of velocities. i runs over all particles.

dx_mean()
Return the average distance between particles.

dv_mean()
Return the average relative speed between particles.

Ekin()
Return the total kinetic energy.

ekin()
Return the total specific kinetic energy.

Epot(*eps***)**
Return the total potential energy using the softening lenght *eps* (may take a very long time to compute). WARNING : THIS FUNCTION DO NOT WORK IN MPI MODE

epot(*eps***)**
Return the total specific potential energy using the softening lenght *eps* (may take a very long time to compute) WARNING : THIS FUNCTION DO NOT WORK IN MPI MODE

L()
Return the angular momentum in x,y,z of all particles. The output is an 3xn Float array.

l()
Return the specific angular momentum in x,y,z of all particles. The output is an 3xn Float array.

Ltot()
Return the total angular momentum. The output is an 3×1 Float Numeric array.

ltot()
Return the specific total angular momentum. The output is an 3×1 Float Numeric array.

Pot(*x,eps***)**
Return the potential at a given position, using the softening lenght *eps*. *x* is a 3×1 Float Numeric array.

tork(*acc***)**
Return the total tork on the system due to the force acting on each particle (acc). The output is an 3xn Float array. *acc* is an 3xn Float array.

dens([*r=None***],[***nb=25***],[***rm=50***])**
Return the number density at radius *r* (supposing a spherical density distribution). If *r* is not specified, it is computed with *nb* and *rm*. The output is an n x 1 Float array. This function do not use *mass*.

- *r*: radius where to compute the surface density ($n \times 1$ Numeric array) (default=*None*)
- *nb*: number of bins (default=*25*)

- *rm*: maximal radius (default=*50*)

The output is an $n \times 1$ Float Numeric array.

mdens([*r=None*],[*nb=25*],[*rm=50*])

Return the surface density at radius *r* (supposing a spherical density distribution). If *r* is not specified, it is computed with *nb* and *rm*.

- *r*: radius where to compute the surface density ($n \times 1$ Numeric array) (default=*None*)
- *nb*: number of bins (default=*25*)
- *rm*: maximal radius (default=*50*)

The output is an $n \times 1$ Float Numeric array.

mr([*r=None*],[*nb=25*],[*rm=50*])

Return tthe mass inside radius *r* (supposing a spherical density distribution). If *r* is not specified, it is computed with *nb* and *rm*.

- *r*: radius where to compute the surface density ($n \times 1$ Numeric array) (default=*None*)
- *nb*: number of bins (default=*25*)
- *rm*: maximal radius (default=*50*)

The output is an $n \times 1$ Float Numeric array.

sdens([*r=None*],[*nb=25*],[*rm=50*])

Return the number surface density at radius *r*. If *r* is not specified, it is computed with *nb* and *rm*. This function do not use *mass*.

- *r*: radius where to compute the surface density ($n \times 1$ Numeric array) (default=*None*)
- *nb*: number of bins (default=*25*)
- *rm*: maximal radius (default=*50*)

The output is an $n \times 1$ Float Numeric array.

msdens([*r=None*],[*nb=25*],[*rm=50*])

Return the mass surface density at radius *r*. If *r* is not specified, it is computed with *nb* and *rm*.

- *r*: radius where to compute the surface density ($n \times 1$ Numeric array) (default=*None*)
- *nb*: number of bins (default=*25*)
- *rm*: maximal radius (default=*50*)

The output is an $n \times 1$ Float Numeric array.

zprof([*z=None*],[*r=2.5*],[*dr=0.5*],[*nb=25*],[*zm=5*])

Return the z-profile in an $n \times 1$ Float Numeric array, for a given radius *r*.

- z: bins in z, $n \times 1$ Numeric array (default=*None*)
- *r*: radius of the cut (default=*2.5*)
- *dr*: width in r of the cut (default=*0.5*)
- *nb*: number of bins (default=*25*)
- *zm*: maximal height (default=*5*)

!!! This routine works only if particles have equal masses !!!

sigma([*r=None*],[*nb=25*],[*rm=50*])

Return the 3 velocity dispersions (in cylindrical coordinates) and the mean azimuthal velocity curve. If *r* is not specified, it is computed with *nb* and *rm*.

- *r*: radius where to compute the values ($n \times 1$ Numeric array) (default=*None*)
- *nb*: number of bins (default=*25*)
- *rm*: maximal radius (default=*50*)

The output is a list (*r,sr,st,sz,mt*) of 5 $n \times 1$ Float Numeric arrays, where *r* is the radius, *sr* the radial velocity dispersion, *st*, the azimuthal velocity dispersion, *sz*, the vertical velocity dispersion and *mt*, the mean azimuthal velocity curve.

!!! This routine works only if particles have equal masses !!!

histovel([*nb*],[*vmin*],[*vmax*],[*plot*],[*mode*])
Return and plot the histogram of the norm of velocities or of the radial velocities.

- *nb*: number of bins (default=*100*)
- *vmin*: maximum velocity (default=*None*)
- *vmax*: minimum velocity (default=*None*)
- *plot*: 'yes'=trace a plot, 'no'=don't trace a plot (default='*no*')
- *mode*: 'n'=norm of the velocities, 'r'=radial velocities (default='*n*')

The output is a list (*r,h*) of 2 $n \times 1$ Float Numeric arrays, where *r* is the radius and *h* the values of the histogram.

zmodes([*nr=32*],[*nm=16*],[*rm=32*])
Compute the vertical modes of a model

zmodes([*nr=32*],[*nm=16*],[*rm=32*])
Compute the density modes of a model

7.7 Thermodynamic functions

These functions works if *rho* (density of particles) and *u* (specific energy of particles) are correctely defined.

Rho()
Return the gas density of the model. The output is an nx1 Float array.

T()
Return the gas temperature of the model. The output is an nx1 Float array.

A()
Return the gas entropy of the mode The output is an nx1 Float array.

P()
Return the gas pressure of the model. The output is an nx1 Float array.

Tcool()
Return the cooling time of the model. The output is an nx1 Float array.

Ne()
Return the electron density of the model. The output is an nx1 Float array.

S()
Return the 'entropy' of the model, defined as S = T * Ne**(1-gamma) The output is an nx1 Float array.

Lum()
Return the luminosty of the model, defined as Lum = m*u/Tcool = m*Lambda/rho The output is an nx1 Float array.

7.8 Adding particles

append(*new*)
Append to the current Nbody object, particles form the Nbody object *new*.

7.9 Selection of particles

selectc(*c*)
Return an Nbody object that contains only particles where the corresponding value in *c* is not zero. *c* is a $n \times 1$ Numeric array.

selectp([*lst*],[*file*])
Return an Nbody object that contains only particles with indexes.

The list of indexes is given either by *lst* ($n \times 1$ Int Numeric array) or by the name (*file*) of a file containing the list of id's.

sub([*n1*],[*n2*])
Return an Nbody object that have particles with indexes in the range [*n1*,*n2*].

- *n1*: number of the first particle (default=*1*)
- *n2*: number of the last particle (default=*None*)

reduc(*n*)
Return an Nbody object that contains a fraction 1/*n* of particles. Only particles that have an index equal to a multiple of *n* are conserved.

select(*type*)
Return an Nbody object that contain only particles of type *type*. This function depends on the object format.

getindex(*num*)
Return an array of index of a particle from its specific number id. The array is empty if no particle corresponds to the specific number id. num : Id of the particle

7.10 Redistribution of particles

redistribute()
This function redistribute particles amoung all nodes in order to have a similar number of particles per nodes

7.11 Geometrical operations

cmcenter()
Move the Nbody object in order to center the mass center at the origin.

cvcenter()
Center the center of velocities at the origin.

histocenter([*rbox*],[*nb*])
Move the Nbody object in order to center the higher density region found near the mass center. The higher density region is determined with density histograms.

- *rbox*: box dimension, where to compute the histograms (default=*50*)
- *nb*: number of bins for the histograms (default=*500*)

hdcenter()

Move the Nbody object in order to center the higher density region found. The higher density region is the maximum of variable *rho*.

translate(*dx*,[*mode*])

Translate the positions or the velocities of the object.

- *dx*: translation vector (3×1 Float0 Numeric array)
- *mode*: 'p'= translate the positions, 'v'= translate the velocities (default=*'p'*)

rotate([*angle*],[*mode*],[*axis*])

Rotate the positions and/or the velocities of the object around a specific axis.

- *angle*: rotation angle in radian (default=*0*)
- *mode*: 'p'= rotate only the positions, 'v'=rotate only the velocities, 'a'=rotate the positions and velocities (default=*'a'*)
- *axis*: axis of rotation : 'x', 'y', 'z' or $[x, y, z]$ (default=*'x'*)

rotate2([*angle*],[*axis*],[*point*])

Rotate the positions and/or the velocities of the object around a specific axis. This function rotate positions and velocities

- *angle*: rotation angle in radian (default=*0*)
- *axis*: rotation axis
- *point*: rotation center

align(*axis*,[*mode*],[*sgn*],[*fact*])

Rotate the object in order to align the axis *axis* with the z axis.

- *axis*: axis to align : $[x, y, z]$
- *mode*: 'p'= rotate only the positions, 'v'=rotate only the velocities, 'a'=rotate the positions and velocities (default=*'a'*)
- *sgn*: '+': normal rotation, '-': reverse sense of rotation (default='+')
- *fact*: multiply the rotation angle by a factor *fact* (default=*None*)

align2([*axis1=[1,0,0]*],[*axis2=[0,0,1]*],*point=[0,0,0]*)

Rotate the object in order to align the axis1 *axis1* with the axis2 *axis2*.

- *axis1*: first axis
- *axis2*: second axis
- *point*: center of rotation

spin([*omega=None*],[*L=None*],[*j=None*],[*E=None*])

Spin the object with angular velocity "omega" (rigid rotation). Omega is a 1 x 3 array object. If L (total angular momentum) is explicitely given, compute Omega from L (1 x 3 array object).

- *omega*: first axis
- *L*: desired angular momentum
- *j*: desired energy fraction in rotation
- *E*: Total energy (without rotation)

7.12 Specific parallel functions

gather_pos()
Gather in a unique array all positions of all nodes.

gather_vel()
Gather in a unique array all velocites of all nodes.

gather_mass()
Gather in a unique array all mass of all nodes.

gather_num()
Gather in a unique array all num of all nodes.

gather_vec()
Gather in a unique array all vectors vec of all nodes.

7.13 Graphical functions

display(*parameters*,[*palette=None*],[*save=None*])
Display the model using parameters *parameters*. This function allows to display or save an image from an `Nbody` object. It calls the method **Map**, and display or save it.

- *parameters*: display parameters (see 6.1)
- *palette*: name of the palette (file in `rgb_tables/` directory)
- *save*: output file name ('.gif' or '.fits')

```
>>> from pNbody import *
>>> nb = Nbody('gadget_z00.dat',ftype='gadget')
>>> nb.display()
>>> nb.display(size=(10000,10000),shape=(256,256),palette='lut2')
>>> params = {"size":(10000,10000),"shape":(256,256),"filter_name":'convol'}
>>> nb.display(params,palette='rainbow4')
```

show(*parameters*)
This function is an alias to the function *display*.

```
>>> from pNbody import *
>>> nb = Nbody('gadget_z00.dat',ftype='gadget')
>>> nb.show()
>>> nb.show(size=(10000,10000),shape=(256,256),palette='lut2')
>>> params = {"size":(10000,10000),"shape":(256,256),"filter_name":'convol'}
>>> nb.show(params,palette='rainbow4')
```

Map(*parameters*)
This function creates a map using parameters defined in *parameters*, where *parameters* corresponds to parameters defined in section 6.1.

It calls the methods **CombiMap** that create the matrix *mat*.

Then, it makes an integer matrix (*matint*) calling the function **set_ranges** and optionally add contours or box, using the functions **contours** and **add_box**.

It finally returns a tuple (*mat*,*matint*,*mn_opts*,*mx_opts*,*cd_opts*)

- *mat*: real matrix (Nx*Ny Float array)
- *matint*: image matrix with values between 0 and 255 (Nx*Ny Int array)

•*mn_opt*: minimum physical value used to create *matint*

•*mx_opt*: maximum physical value used to create *matint*

•*cd_opt*: *cd* parameter used to create *matint*

```
>>> from pNbody import *
>>> nb = Nbody('gadget_z00.dat',ftype='gadget')
>>> mat,matint,mn_opt,mx_opt,cd_opt = nb.Map()
>>> mplot(matint)
```

CombiMap(*parameters*)

This function returns a Float numarray matrix corresponding to the physical value defined by the parameter `mode`. The function uses either **ComputeMeanMap**, **ComputeSigmaMap**, or use directely the function **ComputeMap** to compute the desired matrix. The meaning of the parameter `mode` is given in the following table.

Value	Meaning	Formula
"sr"	dispertion in r	$\sqrt{\langle\sum m\, r^2\rangle/\langle\sum m\rangle - (\langle\sum m\, r\rangle/\langle\sum m\rangle)^2}$
"svr"	dispertion in vr	$\sqrt{\langle\sum m\, vr^2\rangle/\langle\sum m\rangle - (\langle\sum m\, vr\rangle/\langle\sum m\rangle)^2}$
"svxyr"	dispertion in vxy	$\sqrt{\langle\sum m\, vxy^2\rangle/\langle\sum m\rangle - (\langle\sum m\, vxy\rangle/\langle\sum m\rangle)^2}$
"svtr"	dispertion in vt	$\sqrt{\langle\sum m\, vt^2\rangle/\langle\sum m\rangle - (\langle\sum m\, vt\rangle/\langle\sum m\rangle)^2}$
"szr"	ratio sigma z/sigma r	

If the value of `mode` may also be set to a variable or function name. In this case, a map using the function **ComputeMeanMap**, with a physical value taken as the given variable is returned.

```
>>> from pNbody import *
>>> nb = Nbody('gadget_z00.dat',ftype='gadget')
>>> nb = nb.select('gas')
>>> mat = nb.CombiMap(mode='T',filter_name='convol')
>>> matint,mn_opt,mx_opt,cd_opt = set_ranges(mat,scale='log',cd=0,mn=0,mx=0)
>>> mplot(matint,palette='rainbow4')
>>> mat = nb.CombiMap(mode='log10(nb.T())',filter_name='convol')
>>> matint,mn_opt,mx_opt,cd_opt = set_ranges(mat,scale='lin',cd=0,mn=7,mx=12)
>>> mplot(matint,palette='rainbow4')
```

ComputeMeanMap(*parameters,mode1*)

Compute the mean map of an observable and return int in an Float32 numarray matrix.

$$M = \frac{M1}{M0} \tag{7.3}$$

$M0$ is the zero momentum map and $M1$ is the first momentum map of the variable given by `mode1`.

```
>>> from pNbody import *
>>> nb = Nbody('disk.dat',ftype='gadget')
>>> nb = nb.select('disk')
>>> mat = nb.ComputeMeanMap(filter_name='convol',size=(75,75),view='xy',mode1='z')
>>> matint,mn_opt,mx_opt,cd_opt = set_ranges(mat,scale='lin',cd=0,mn=-10,mx=10)
>>> mplot(matint,palette='rainbow4')
```

ComputeSigmaMap(*parameters,mode1,mode2*)

Compute the dispersion map an observable and return it in an Float32 numarray matrix.

$$M = \sqrt{\frac{M2}{M0} - \left(\frac{M1}{M0}\right)^2} \tag{7.4}$$

$M0$ is the zero momentum map and $M1$($M2$) is a first(second) momentum map of the variable given by `mode1`(`mode2`).

```
>>> from pNbody import *
>>> nb = Nbody('disk.dat',ftype='gadget')
>>> nb = nb.select('disk')
>>> mat = nb.ComputeSigmaMap(filter_name='convol',size=(75,75),view='xy',mode1='z',mode2='z
>>> matint,mn_opt,mx_opt,cd_opt = set_ranges(mat,scale='lin',cd=0,mn=-2,mx=2)
>>> mplot(matint,palette='rainbow4')
```

`ComputeMap`(*parameters*)

Compute the momentum map of an observable and return it in an Float32 numarray matrix. It calls the functions :

1. **`getval`** : If the physical value to plot is not dependent on the observer position, get it from *mode*.
2. **`get_obs`** : if the observer position matrix *obs* is not defined, compute it using *x0*, *xp*, *alpha*, *view*, *r_obs*.
3. **`expose`** : expose the model, using *obs*, *eye*, *dist_eye*, *foc*, *space*.
4. **`getval`** : If the physical value to plot depends on the observer position, get it now from *mode*.
5. **`frustrum`** or **`ortho`** : Project the model, using *clip* and *size*.
6. **`getval`** : If the physical value to plot depends on the observer position, get it now from *mode*.
7. None : Performe some selections
8. **`viewport`** : perform viewport transformation using *shape*.
9. **`mkmapsph`** or **`mkmap`** : compute the map using *shape*.
10. **`apply_filter`** : apply filter on the map using *filter_name* and *filter_opts*.

The following table lists corresponding physical values fo the parameter *mode* that do not dependends on the observer position :

Value	Meaning	Formula
”0”	zero momentum	$\sum m$
”m”	zero momentum	$\sum m$
”x”	first momentum in x	
”y”	first momentum in y	
”z”	first momentum in z	
”x2”	second momentum in x	
”y2”	second momentum in y	
”z2”	second momentum in z	
”vx”	first velocity momentum in x	$\sum m\,vx$
”vy”	first velocity momentum in y	$\sum\sum m\,vy$
”vz”	first velocity momentum in z	$\sum\sum m\,vz$
”vx2”	second velocity momentum in x	$\sum\sum m\,vx^2$
”vy2”	second velocity momentum in y	$\sum\sum m\,vy^2$
”vz2”	second velocity momentum in z	$\sum m\,vz^2$
”lx”	first specific kinetic momemtum in x	
”ly”	first specific kinetic momemtum in y	
”lz”	first specific kinetic momemtum in z	
”Lx”	first kinetic momemtum in x	
”Ly”	first kinetic momemtum in y	
”Lz”	first kinetic momemtum in z	
”u”	first momentum of specific energy	
”rho”	first momentum of density	
”T”	first momentum of temperature	
”A”	first momentum of entropy	
”P”	first momentum of pressure	
”Tcool”	first momentum of cooling time	
”Lum”	first momentum of luminosity	
”Ne”	first momentum of electronic density	

The following table lists corresponding physical values fo the parameter *mode* that do dependends on the observer position :

Value	Meaning	Formula
”r”	first momentum of radial distance	
”r2”	second momentum of radial distance	
”vr”	first momentum of radial velocity	
”vr2”	second momemtum of radial velocity	
”vxyr”	first momentum of radial velocity in the plane	$\sum m\,(x\,vx + y\,vy)/\sqrt{x^2+y^2}$
”vxyr2”	second momentum of radial velocity in the plane	$\sum m\,[(x\,vx + y\,vy)/\sqrt{x^2+y^2}]^2$
”vtr”	first momentum of tangential velocity in the plane	$\sum m\,(x\,vx - y\,vy)/\sqrt{x^2+y^2}$
”vtr2”	second momentum of tangential velocity in the plane	$\sum m\,[(x\,vx - y\,vy)/\sqrt{x^2+y^2}]^2$

```
>>> from pNbody import *
>>> nb = Nbody('disk.dat',ftype='gadget')
>>> nb = nb.select('disk')
>>> nb.cmcenter()
>>> nb.rotate(axis='x',angle=pi/4)
>>> mat = nb.ComputeMap(filter_name='convol',size=(50,50),mode='vr')
>>> matint,mn_opt,mx_opt,cd_opt = set_ranges(mat,scale='lin',cd=0,mn=0,mx=0)
>>> mplot(matint,palette='rainbow4')
>>> mat = nb.ComputeMap(filter_name='convol',size=(50,50),mode='nb.rxyz()')
>>> matint,mn_opt,mx_opt,cd_opt = set_ranges(mat,scale='lin',cd=0,mn=0,mx=0.0004)
>>> mplot(matint,palette='rainbow4')
>>> mat = nb.ComputeMeanMap(filter_name='convol',size=(50,50),mode1='nb.rxyz()')
>>> matint,mn_opt,mx_opt,cd_opt = set_ranges(mat,scale='lin',cd=0,mn=0,mx=50)
>>> mplot(matint,palette='rainbow4')
```

expose(*obs*,[*eye=None*],[*dist_eye=None*],[*foc=None*],[*space='pos'*])

Rotate and translate the object in order to be seen as if the observer was in *x0*, looking at a point in *xp*.

- *obs*: observer matrix
- *eye*: 'right' or 'left'
- *dist_eye*: distance between eyes (separation = angle)
- *space*: "pos" or "vel"
- *foc*: focal

CHAPTER

EIGHT

Libutil module

These functions are not methods of the `Nbody` object. However, there are useful to process some outputs of `Nbody` methods.

`cross_product`(*x,y*)
Return the cross product of two $3 \times n$ arrays.

`histogram`(*a,bins*)
Return the histogram ($n \times 1$ Float0 Numeric array) of the $n \times 1$ Float0 Numeric array *a*. *bins* ($m \times 1$ Float0 Numeric array) specify the bins of the histogram.

`getr`([*nr*],[*nt*],[*rm*])
Return a sequence of number ($n \times 1$ Float0 Numeric array), where $n = \mathrm{nr} + 1$ defined by:

$$Rj = \mathrm{rm}\frac{\exp\left[\mathrm{j}/\left(\frac{1}{2} + \frac{\mathrm{nt}}{2\pi}\right)\right] - 1}{\exp\left[(\mathrm{nr}-1)/\left(\frac{1}{2} + \frac{\mathrm{nt}}{2\pi}\right)\right] - 1} \tag{8.1}$$

```
>>> from Nbody import *
>>> getr()
array([  0.00000000e+00,   7.69677545e-02,
      1.69004032e-01,   2.79058904e-01,
      4.10659999e-01,   5.68025574e-01,
      7.56199722e-01,   9.81214059e-01,
      1.25028105e+00,   1.57202519e+00,
      1.95675946e+00,   2.41681589e+00,
      2.96694083e+00,   3.62476762e+00,
      4.41138184e+00,   5.35199710e+00,
      6.47676327e+00,   7.82173290e+00,
      9.43001676e+00,   1.13531658e+01,
      1.36528233e+01,   1.64027011e+01,
      1.96909419e+01,   2.36229450e+01,
      2.83247442e+01,   3.39470479e+01,
      4.06700700e+01,   4.87093057e+01,
      5.83224396e+01,   6.98176049e+01,
      8.35632603e+01,   1.00000000e+02])
```

`get_eyes`(*x0,xp,alpha,dr*)
Return the position of two eyes (two 3×1 Float0 Numeric array).

- *x0*: position of the observer
- *xp*: position where the observer look at
- *alpha*: rotation around the axis (x0,xp)
- *dx*: distance between eyes

```
>>> from pNbody import *
>>> get_eyes([0,-50,50],[0,0,0],0,5)
(array([  5., -50.,  50.], type=Float32), array([ -5., -50.,  50.], type=Float32))
```

apply_filter(*mat,*[*filter_name*],[*filter_opts*])

Apply a filter on an $n \times 1$ Float0 Numeric array. The only supported filter has *filter_name='convol'* and *filter_opts=(nx,ny,sx,sy)*. The filter operates a convolution of the array with a Gaussian kernel of half width *sx* and *sy* truncated on radius *nx* and *ny*.

- *mat*: input array ($n \times m$ Float Numeric array)
- *filter_name*: name of the filter (default=*None*)
- *filter_opts*: parameters of the filter (default=*None*)

```
>>> from pNbody import *
>>> from numarray import random_array as RandomArray
>>> mat = RandomArray.random([256,256])
>>> mplot(mat)
>>> mat = apply_filter(mat,name='convol',opt=[10,10,3,3])
>>> mplot(mat)
```

set_ranges(*mat,*[*scale*],[*cd*],[*mn*],[*mx*])

Transform an $n \times m$ Float Numeric array into an $n \times m$ Int0 Numeric array that will be used to create an image. The float values are rescaled and cutted in order to range between 0 and 255. The function returns the new Int0 array *mat* and the used values of *mn*, *mx* and *cd*.

- *mat*: input array ($n \times m$ Float Numeric array)
- *scale*: scale "lin" or "log" (default=*log*)
- *cd*: scaling parameter (default=*0*)
- *mn*: minimum value for the cutoff (default=*None*)
- *mx*: maximum value for the cutoff (default=*None*)

If *mn* or *mx* are not defined, they are set to the minimum and maximum value of *mat*.

- 'lin' scale:

$$mat' = 255\,\frac{mat - mn}{mx - mn} \tag{8.2}$$

  In this case, *cd* is not used.

- 'log' scale:

$$mat' = 255\,\frac{ln\left(1 + \frac{mat-mn}{cd}\right)}{ln\left(1 + \frac{mx-mn}{cd}\right)} \tag{8.3}$$

  If *cd* is not defined, it is set to the mean value of *mat*.

```
>>> from pNbody import *
>>> n = 256
>>> x,y = indices((n+1,n+1))
>>> x = 2*pi*(x-n/2)/(n/2)
>>> y = 2*pi*(y-n/2)/(n/2)
>>> R = sqrt(x**2+y**2)
>>> R1 = sqrt((x-pi)**2+(y-pi)**2)
>>> mat = cos(R)/(1+R) + 0.5*cos(R1)/(1+R1)
>>> mplot(mat)
>>> mat_int,mn,mx,cd = set_ranges(mat,scale='lin',mn=0,mx=0.5)
>>> mplot(mat_int)
```

contours(*m,matint,*[*nl*],[*mn*],[*mx*],[*kx*],[*ky*],[*color*],[*crush*])

Compute iso-contours on a $n \times m$ Float Numeric array. Contours computed from *mat* are superposed on the integer matrix *matint*.

- *mat*: input array ($n \times m$ Float Numeric array)
- *matint*: input array ($n \times m$ Int Numeric array)
- *l_n*: number of levels (default=*15*)
- *l_min*: minimum level value (default=*0*)
- *l_max*: maximum level value (default=*0*)
- *l_kx*: smoothing of contours in x (default=*10*)
- *l_ky*: smoothing of contours in y (default=*10*)
- *l_color*: color between 0 and 255 (default=*0*)
- *l_crush*: "yes"=remove background colors, "no"=keep background colors (default="*no*")

If *l_min* equal *l_max*, levels are automatically between the minimum and maximum values of the matrix *mat*. If *color*=0, no contour are displayed.

```
>>> from pNbody import *
>>> n = 256
>>> x,y = indices((n+1,n+1))
>>> x = 2*pi*(x-n/2)/(n/2)
>>> y = 2*pi*(y-n/2)/(n/2)
>>> R = sqrt(x**2+y**2)
>>> R1 = sqrt((x-pi)**2+(y-pi)**2)
>>> mat = cos(R)/(1+R) + 0.5*cos(R1)/(1+R1)
>>> mat_int,mn,mx,cd = set_ranges(mat,scale='lin')
>>> mat_int = contours(mat,mat_int,nl=30,mn=mn,mx=mx,kx=10,ky=10,color=1)
>>> mplot(mat_int)
>>> mat_int = contours(mat,mat_int,nl=30,mn=mn,mx=mx,kx=10,ky=10,color=1,crush='yes')
>>> mplot(mat_int)
```

add_box(*matint,*[*shape=(256,256)*],[*size=(30.,30.)*],[*box_opts=(0,None,None,255)*])

Superpose of box to the matrix *matint*.

- *matint*: input array ($n \times m$ Int Numeric array)
- *shape*: shape of the matrix
- *size*: physical size of the matrix
- *box_opts*: box options (lweight,xticks,yticks,color)

```
\begin{verbatim}
>>> from pNbody import *
>>> from numarray import random_array as RandomArray
>>> mat = RandomArray.random([256,256])
>>> mat = apply_filter(mat,name='convol',opt=[10,10,3,3])
>>> mat_int,mn,mx,cd = set_ranges(mat,scale='lin',mn=0,mx=0.5)
>>> mat_int = add_box(mat_int,shape=(256,256),size=(100,100),box_opts=(1,None,None,1))
>>> mplot(mat_int)
```

mplot(*mat*,[*palette='light'*],[*save='no'*])

Plot a or save the matrix array *mat*.

```
>>> from pNbody import *
>>> from numarray import random_array as RandomArray
>>> mat = RandomArray.random([256,256])
>>> mplot(mat)
>>> mplot(mat,save='image.gif')
>>> mplot(mat,save='image.fits')
```

get_image(*mat*,[*name*],[*palette_name*])

From the $n \times m$ Int Numeric array, create a PIL (Python Image Library) image. If *name* is given, the image is saved on the disk. *palette_name* is the name of the palette and is one of the following: aips0, backgr, bgyrw, blackwhite, blue, blulut, color, green, half, heat, idl11, idl12, idl14, idl15, idl2, idl4, idl5, idl6, isophot, light, lut0, lut1, lut2, lut3, lut4, lut5, lut6, lut7, lut8, lut9, manycol, pastel, pmar, rainbow, rainbow1, rainbow2, rainbow3, rainbow4, ramp, random, random1, random2, random3, random4, random5, random6, real, red, smooth, smooth1, smooth2, smooth3, staircase, stairs8, stairs9, standard, whiteblack

```
>>> from pNbody import *
>>> from numarray import random_array as RandomArray
>>> mat = RandomArray.random([256,256])
>>> mat_int,mn,mx,cd = set_ranges(mat,scale='lin')
>>> image = get_image(mat_int,palette_name='rainbow4')
```

display(*image*)

Display in a TK window the image *image* (PIL image).

```
>>> from pNbody import *
>>> from numarray import random_array as RandomArray
>>> mat = RandomArray.random([256,256])
>>> mat_int,mn,mx,cd = set_ranges(mat,scale='lin')
>>> image = get_image(mat_int,palette_name='rainbow4')
>>> display(image)
```

sbox(*shape,size,*[*lweight=1*],[*xticks=None*],[*yticks=None*],[*color=255*])

Return a matrix of integer, containing a box with labels

drawxticks(*matint,m0,d0,n0,h0,shape,size,center,color*)

Return a matrix of integer, containing ticks in x.

drawyticks(*matint,m0,d0,n0,h0,shape,size,center,color*)

Return a matrix of integer, containing ticks in y.

getval(*nb,mode='m',obs=None*)

Return a specifc value extracted from the `pNbody` object *nb* according to the mode *mode*. These function is used in the method **ComputeMap**.

getvaltype(*mode='m'*)

Return 'normal' or 'in projection' depending on the mode *mode*. These function is used in the method **ComputeMap**.

CHAPTER

NINE

Mpi module

9.1 Output functions

mpi_pprint(*msg*)
Synchronized print.

```
[lunix@lunix ~/examples]$ mpirun -np 2 python scripts/mpi_pprint.py
[0] : hi !
[1] : hi !
```

mpi_rprint(*msg*)
Rooted print.

```
[lunix@lunix ~/examples]$ mpirun -np 2 python scripts/mpi_rprint.py
hi !
hi !
```

mpi_iprint(*msg*)
Synchronized print, including info on node.

```
[lunix@lunix ~/examples]$ mpirun -np 2 python scripts/mpi_rprint.py
hi, I am the master !
```

9.2 Communication functions

mpi_Bcast(*source,x*)
Broadcast from node *source* the variable *x*.

```
[lunix@lunix ~/examples]$ mpirun -np 2 python scripts/mpi_Bcast.py
[0] : hello from the master
[1] : hello from the master
```

mpi_Send(*x,dest*)
Send *x* to node *dest*.

```
[lunix@lunix ~/examples]$ mpirun -np 2 python scripts/mpi_SendRecv.py
[1] : get 123 from node 0
```

mpi_Recv(*source*)
Return a variable sent by node *source*.

```
[lunix@lunix ~/examples]$ mpirun -np 2 python scripts/mpi_SendRecv.py
[1] : get 123 from node 0
```

mpi_reduce(*source*)
Reduce *x* from nodes.

```
[lunix@lunix ~/examples]$ pirun -np 2 python scripts/mpi_reduce.py
[0] : 3
[1] : 3
```

mpi_Gather(*x,dest*)
Gather *x* from all nodes to node *dest*. Returns a list.

```
[lunix@lunix ~/examples]$ mpirun -np 2 python scripts/mpi_Gather.py
[0] : None
[1] : [1, 2]
```

mpi_AllGather(*x*)
Gather *x* from all nodes to all nodes. Returns a list.

```
[lunix@lunix ~/examples]$ mpirun -np 2 python scripts/mpi_Allgather.py
[0] : [1, 2]
[1] : [1, 2]
```

mpi_AllgatherAndConcatArray(*x*)
AllGather array *x* and concatenate it in a unique array (concatenation order is reversed).

```
[lunix@lunix ~/examples]$ mpirun -np 2 python scripts/mpi_AllgatherAndConcatArray.py
[0] : [0 1 2]
[1] : [3 4 5]
[0] : [3 4 5 0 1 2]
[1] : [3 4 5 0 1 2]
```

9.3 Array functions

mpi_sum(*x*)
Sum elements of array *x*.

```
[lunix@lunix ~/examples]$ mpirun -np 2 python scripts/mpi_sum.py
[0] : [0 1 2]
[1] : [3 4 5]
[0] : 15
[1] : 15
```

mpi_min(*x*)
Minimum element of array *x*.

```
[lunix@lunix ~/examples]$ mpirun -np 2 python scripts/mpi_minmax.py
[0] : [0 1 2]
[1] : [3 4 5]
[0] : min= 0
[1] : min= 0
[0] : max= 5
```

mpi_max(*x*)
Maximum element of array *x*.

```
[lunix@lunix ~/examples]$ mpirun -np 2 python scripts/mpi_minmax.py
[0] : [0 1 2]
[1] : [3 4 5]
[0] : min= 0
[1] : min= 0
[0] : max= 5
```

mpi_mean(*x*)
Mean of elements of array *x*.

```
[lunix@lunix ~/examples]$ mpirun -np 2 python scripts/mpi_mean.py
[0] : [ 0.  1.  2.]
[1] : [ 3.  4.  5.]
[0] : mean= 2.5
[1] : mean= 2.5
```

mpi_len(*x*)
Lenght of array *x*.

```
[lunix@lunix ~/examples]$ mpirun -np 2 python scripts/mpi_len.py
[0] : [0 1 2]
[1] : [3 4 5]
[0] : len= 6
[1] : len= 6
```

mpi_arange(*n*)
Create an integer array containing elements from 0 to *n* spreaded over all nodes.

```
[lunix@lunix ~/examples]$ mpirun -np 2 python scripts/mpi_arange.py
[0] : [10 11 12 13 14 15 16 17 18 19]
[1] : [0 1 2 3 4 5 6 7 8 9]
```

mpi_arange(*nall*)
Create an integer array containing elements from 0 to *n* spreaded over all nodes. The repartition of elements and type of elements over nodes is given by the array *nall*.

```
[lunix@lunix ~/examples]$ mpirun -np 2 python scripts/mpi_sarange.py
[0] : [ 5  6  8  9 10]
[1] : [0 1 2 3 4 7]
```

mpi_argmax(*x*)
Find arument of the maximum value in x.

```
[lunix@lunix ~/examples]$ mpirun -np 2 python scripts/mpi_argfind.py
\begin{verbatim}
[lunix@lunix ~/examples]$ mpirun -np 2 python scripts/mpi_argfind.py
[0] : [3 0 4]
[1] : [2 5 6]
[0] : (0, 1)
[1] : (0, 1)
[0] : (1, 2)
[1] : (1, 2)
[0] : 0
[1] : 0
[0] : 6
[1] : 6
```

mpi_argmin(*x*)

Find arument of the mimimum value in x.

```
[lunix@lunix ~/examples]$ mpirun -np 2 python scripts/mpi_argfind.py
\begin{verbatim}
[lunix@lunix ~/examples]$ mpirun -np 2 python scripts/mpi_argfind.py
[0] : [3 0 4]
[1] : [2 5 6]
[0] : (0, 1)
[1] : (0, 1)
[0] : (1, 2)
[1] : (1, 2)
[0] : 0
[1] : 0
[0] : 6
[1] : 6
```

mpi_getval(*x,idx*)

Return the value of array *x* corresponding to the index *idx*.

```
[lunix@lunix ~/examples]$ mpirun -np 2 python scripts/mpi_argfind.py
\begin{verbatim}
[lunix@lunix ~/examples]$ mpirun -np 2 python scripts/mpi_argfind.py
[0] : [3 0 4]
[1] : [2 5 6]
[0] : (0, 1)
[1] : (0, 1)
[0] : (1, 2)
[1] : (1, 2)
[0] : 0
[1] : 0
[0] : 6
[1] : 6
```

mpi_histogram(*x,binx*)

Return an histogram of vector *x* binned using *binx*.

```
[lunix@lunix ~/examples]$ mpirun -np 2 python scripts/mpi_histogram.py
[0] : [ 0.84406817  0.36298877  0.97603917  0.93461412  0.79251379
  0.34625384  0.68936312  0.34766984  0.00754931  0.18829931]
[1] : [ 0.84404922  0.59192866  0.99656236  0.06286281  0.48847792
  0.51723957  0.44006979  0.10298632  0.3450942   0.56593537]
[0] : [ 0.    0.25  0.5   0.75]
[1] : [ 0.    0.25  0.5   0.75]
[0] : [4 6 4 6]
[1] : [4 6 4 6]
```

9.4 io functions

mpi_ReadAndSendBlock(*f,data_type,*[*shape=None*],[*byteorder=sys.byteorder*],[*split=None*])
Read and brodcast a binary block.

mpi_ReadAndSendArray(*f,data_type,*[*shape=None*],[*skip=None*],[*byteorder=sys.byteorder*],[*nlocal=None*])
Read and brodcast a binary block assuming it contains an array. The array is splitted acroding to the variable *nlocal*.

mpi_GatherAndWriteArray(*f,data_type*[*byteorder=sys.byteorder*],[*nlocal=None*])
Gather and array and write it in a binary block.

Ic module

CHAPTER

ELEVEN

Io module

CHAPTER

TWELVE

Cosmo module

CHAPTER

THIRTEEN

Thermodyn module

CHAPTER

FOURTEEN

Geometry module

CHAPTER

FIFTEEN

Associated programs

15.1 gwin

15.1.1 Description

gwin allows to display an pNbody object with a user friendly interface. It contains all possibilities of selection/filtering/exposition/stereo offered by the module pNbody.

15.1.2 Usage

```
usage: gwin [options] file

options:
  -h, --help        show this help message and exit
  -t  TYPE          type of the file
  --stereo          enable stereo view
  -p  PALETTE NAME  palette name
  -f  FILE NAME     parameter file
  -u  FILE NAME     unitsparameter file
  --fullscreen      fullscreen mode
  --pio             parallel io mode
```

15.1.3 Example

```
[lunix@lunix ~/examples]$ gwin -t gadget gadget_z00.dat
```

15.2 gpy

15.2.1 Description

gpy allows open an `pNbody` object and enter the python interpreter, where you can interact with the object using the variable *nb*.

15.2.2 Usage

```
usage: gpy [options] file

options:
  -h, --help      show this help message and exit
  -t  TYPE        type of the file
  --exec= STRING  give command to execute before
```

15.2.3 Example

```
[lunix@lunix ~/examples]$ gpy -t gadget disk.dat
>>> len(nb.mass)
3001
```

15.3 gcmd

15.3.1 Description

Perfome a serie of operations on a list of files.

15.3.2 Usage

```
usage: gcmd [options] file

options:
  -h, --help      show this help message and exit
  -t  TYPE        type of the file
  --exec= STRING  give command to execute before making the image
```

15.3.3 Examples

```
[lunix@lunix ~/examples]$ gcmd -t gadget gadget_z00.dat gadget_z40.dat --exec=’’’print nb.p_name[0];
gadget_z00.dat
a=  1.000
gadget_z40.dat
a=  0.027
```

15.4 gimage

15.4.1 Description

Create an image from a model file.

15.4.2 Usage

```
usage: gimage [options] file

options:
  -h, --help            show this help message and exit
  -f  FILE NAME         parameter file
  -u  FILE NAME         unitsparameter file
  -t  TYPE              type of the file
  -o  STRING            oputput file name
  --mode= STRING        observable mode
  --palette= STRING     palette name
  --mn= FLOAT           min value
  --mx= FLOAT           max value
  --cd= FLOAT           cd value
  --scale= STRING       scale
  --view= STRING        view xy,xz,yz
  --space= STRING       space pos,vel
  --filter_name= STRING
                        filter name
  --filter_opts= STRING
                        filter opts
  --shape= STRING       shape (x,y)
  --size= STRING        size (x,y)
  --exec= STRING        give command to execute before making the image
```

15.4.3 Examples

```
[lunix@lunix ~/examples]$ gimage -t gadget gadget_z00.dat --palette=’rainbow4’ --mode=’T’ \
? --shape=’(256,256)’ --size=’(10000,10000)’ --exec=’’’nb = nb.select("gas")’’’ -o image.gif
min= 0.000e+00 max= 6.487e+12 cd= 1.114e+10
```

15.5 ginter

15.5.1 Description

15.5.2 Usage

15.5.3 Examples

15.6 mkgmov

15.6.1 Description

mkgmov allows to create **gmov** movie from a list of Nbody files.

15.6.2 Usage

```
Options: -h        -- this help message
           -p        -- parameter file
           -f        -- g-parameter file
           -s        -- disable the softening of rsp
           -c        -- enable auto cd for each image
           -z        -- convert time in redshift
           --info    -- give the optimal factor for each files in the list
           --help    -- this help message
           --version -- displays version
```

15.6.3 Example

```
[lunix@lunix ~/examples]$ cd films/
[lunix@lunix ~/films]$ mkgmov -p filmparam.py film.gmv treo*
```

15.6.4 Parameter file

To create a movie, the best is to provide **mkgmov** a parameter file. Such an example is given below:

```
# number of horizontal and vertical subfilms
nh = 1  # horizontal
nw = 2 # vertical

# size of subfilms
width = 256
height = 256

# size of the film
numByte = width * nw
numLine = height * nh
# init parameters
param = initparams(nh,nw)


# params 1
param[1]['ftype'] = 'binary'
param[1]['view'] = 'xy'
param[1]['mode'] = 'm'
param[1]['size'] = (35.,35.)


# params 2
param[2]['ftype'] = 'binary'
param[2]['x0'] = [0,-1e-5,50]
param[2]['xp'] = [0,0,0]
param[2]['size'] = (35.,35.)
param[2]['exec'] = '''nb = nb.selectp(file='sub.txt')'''
param[2]['mode'] = 'vr'
param[2]['filter_name'] = 'convol'
param[2]['mn'] = -0.14
param[2]['mx'] = 0.14
param[2]['cd'] = 0.0002346
```

First, we have to give the number of horizontal `nh` and vertical `nv` subfilms. Then, the `width` and `height` of each subfilms. The next five lines must remain untouched :

```
# size of the film
numByte = width * nw
numLine = height * nh
# init parameters
param = initparams(nh,nw)
```

Then, for each subfilm, we have to define parameters to use. The parameter corresponds to the parameter of the function **show** described in the paragraph 7.13. Each parameter is given with a line of the type:

`param[` *subfilm number* `][`*parameter name*`]=`*value*

- *subfilm number*: number of the subfilm, from 1 to `nh`$\times$`nv`
- *parameter name*: parameter name (see function **show**)
- *value*: value of the parameter

15.6.5 Notes

- compare to command **`show`** additional parameters are :

Value	Meaning
exec	execute a command
macro	execute a macro (path of the file)
n1	first particle
n2	last particle
mdis	select particle in function of the value of *dis* (see function **`subdis`**)
ftype	type of the file

- parameters *palette* and *save* are not take into account.
- If *dark* is given as a value form the parameter *mode*, the corresponding submovie will simply be black.
- The option `--info` do not write any output. It simply gives the minimum and maximum value, as well as the optimum parameter *cd* of each frame. This is often useful to run **`mkgmov`** first with option `--info`, before choosing the values of *mn*, *mx* and *cd*.

15.7 gmov

15.7.1 Description

This program is used to show **gmov** movie files created for example with **mkgmov**.

15.7.2 Usage

```
usage: gmov [options] file

options:
  -h, --help            show this help message and exit
  -f, --fullscreen      fullscreen mode
  -a, --auto            automatically run the movie
  --nopalette           do not display the palette
  --nocontroll          do not display the controll pannel
  --notime              do not display the time
  -p  PALETTE NAME      palette name
  -z  INT               zoom factor
  -s  INT               speed
  -i                    read the whole film before displaying it (get info)
  --stepsize= INT       size of step
  --delay= FLOAT        delay before starting in auto mode
  --presentation        choose parameters for presentation
  --loop                loop on
  -b  COLOR, --background= COLOR
                        palette name
```

15.7.3 Main window

The main window (see Fig. 15.1) allows to control the movie. Main functions are:

- `RUN`: Run the movie from the frame `start.pic` to the frame `stop.pic`. Use it only when the film is stopped.
- `STOP`: Stop the movie.
- `STEP`: Move to the next frame, step by step. Use it only when the film is stopped.
- `BACK`: Move to the previous frame, step by step. Use it only when the film is stopped.
- `RESET`: Go to the `start.pic` frame.
- `QUIR`: Quit
- `ZOOM`: Zoom in.
- `UNZOOM`: Zoom out.
- `RESET ZOOM`: Disable zoom.

The `filename` entry display the name of the current film. A new film can be loaded by modifying the name and pressing *enter*.

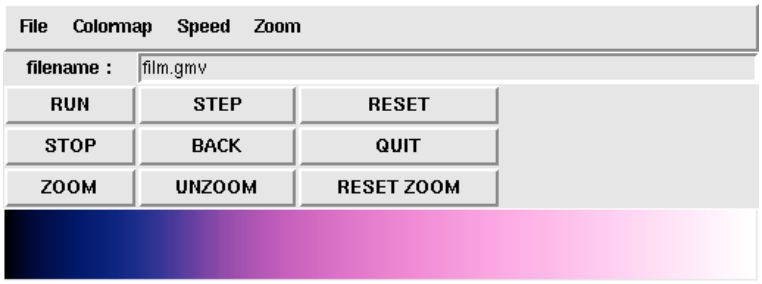

Figure 15.1: Main window of gmov.

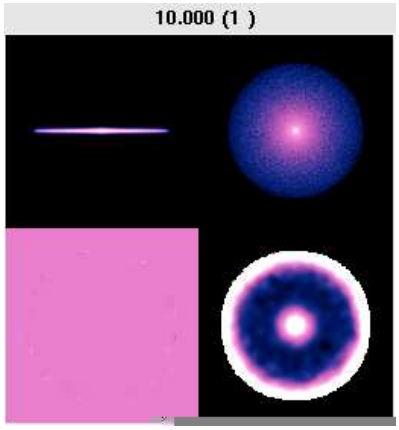

Figure 15.2: Display window of gmov.

15.7.4 Menus

File

- `open`: Open a new film.
- `reload`: Reopen the film.
- `save image`: Save an image (format must be .gif, .fits or .png).
- `quit`: quit the application

Colormap

Change the color map.

Speed

Change the speed of the movie. 1 is the fastest, 100 is the slowest.

Zoom

Zoom in or zoom out.

15.7.5 Examples

```
[lunix@lunix ~/films]$ gmov film.gmv
```

15.8 infogmov

15.8.1 Description

Give info from a given film.

15.8.2 Usage

```
infogmov -f film [options]

  This function gives info on a film

  Options: -h        -- this help message
           -f        -- name of the input
           --help    -- this help message
           --version -- displays version
```

15.8.3 Examples

```
[lunix@lunix ~/films]$  infogmov -f film.gmv

    -- film.gmv  --

    initial time           :  500.000
    final   time           : 2000.000
    dt                     :  500.000
    number of images       : 4
    number of collumns     : 256
    number of lines        : 512
    length of header       : 256
```

15.9 cutgmov

15.9.1 Description

From a given film, **`cutgmov`** creates a new film that contains only frames from *n1* to *n2*.

15.9.2 Usage

```
cutgmov -f filmin -o filmout --n1=n1 --n2=n2

  Options: -h        -- this help message
           -f        -- name of the input
           -o        -- name of the output
           --n1      -- number of the first frame
           --n2      -- number of the last frame
           --info    -- give info on the film
           --help    -- this help message
           --version -- displays version
```

15.9.3 Examples

```
[lunix@lunix ~/films]$ cutgmov -f film.gmv -o film_cutted1.gmv --n1=0 --n2=2
0
1
[lunix@lunix ~/films]$ cutgmov -f film.gmv -o film_cutted2.gmv --n1=2 --n2=4
0
1
2
3
```

15.10 addgmov

15.10.1 Description

addgmov concatenates a number of films of same geometry. The lenght of the films may be different.

15.10.2 Usage

```
addgmov -o outfilm film1 film2 ...

  Options: -h        -- this help message
           -o        -- name of the output
           --help    -- this help message
           --version -- displays version
```

15.10.3 Examples

```
[lunix@lunix ~/films]$ addgmov -o film_new.gmv  film_cutted1.gmv film_cutted2.gmv
500.0
1000.0
1500.0
2000.0
```

15.11 splitgmov

15.11.1 Description

splitgmov split each frame of a movie in *nh*×*nv* subframes. *nh* is the number of horizontal subframes while *nv* is the number of vertical subframes. In output, **splitgmov** return *nh*×*nv* new movies made with the subframes. Each new movie is named with the name of the input movie, followed by the corresponding value of *nh* and *nv*.

15.11.2 Usage

```
splitgmov -f input --nh=nh --nv=nv

  Options: -h        -- this help message
           -f        -- input movie
           --nh      -- number of horizontal images
           --nv      -- number of vertical images
           --help    -- this help message
           --version -- displays version
```

15.11.3 Examples

```
[lunix@lunix ~/films]$ splitgmov -f film.gmv --nh=1 --nv=2
500.0
1000.0
1500.0
2000.0
```

15.12 mergegmov

15.12.1 Description

mergegmov do the opposit as **splitgmov**. In input, it takes $nh \times nv$ films of identical geometry and lenght. It returns a new film composed out of it.

15.12.2 Usage

```
mergegmov -o output --nh=nh --nv=nv film1 film2 ...

  Options: -h        -- this help message
           -o        -- name of the output
           --nh      -- number of horizontal images
           --nv      -- number of vertical images
           --help    -- this help message
           --version -- displays version
```

15.12.3 Examples

```
[lunix@lunix ~/films]$ mergegmov -o film_new.gmv --nh=2 --nv=1 film_11.gmv film_12.gmv
0 / 3
1 / 3
2 / 3
```

15.13 combinegmov

15.13.1 Description

combinegmov put on top of each other images from two different films of same shape and same number of frames.

15.13.2 Usage

```
Usage : combinegmov -o output film1 film2
  Options: -h        -- this help message
           -o        -- name of the output
           --help    -- this help message
           --version -- displays version
```

15.14 supgmov

15.14.1 Description

supgmov put on top of each other images from several different films of same shape and same number of frames.

15.14.2 Usage

```
Usage : supgmov -o outfilm film1 film2 film3...
  Options: -h        -- this help message
           -o        -- name of the output
           --help    -- this help message
           --version -- displays version
```

15.15 gmov2gif

15.15.1 Description

gmov2gif create a *gif* movie from the input film.

15.15.2 Usage

```
gmov2gif -f file [options ...]
  Options: -h        -- this help message
           -f        -- input file
           -o        -- output file
           -p        -- name of a palette
           -z        -- zoom factor
           -d        -- pause between images (ms)
           -b        -- number of initial image
           -e        -- number of final image
           -s        -- number of images to leap
           -l        -- add the label (time)
           -i        -- gives information on the film
           -k        -- keep temporary images
           -t        -- add a text
           --version -- displays version
```

15.15.3 Note

This scripts needs the program **convert** to be installed.

15.15.4 Examples

```
[lunix@lunix ~/films]$ gmov2gif -f film.gmv -o film.gif -p rainbow4
tmp/0000_0500.000
tmp/0001_1000.000
tmp/0002_1500.000
tmp/0003_2000.000
convert -delay 10 tmp/*.gif film.gmv.gif
```

15.16 gmov2mpeg

15.16.1 Description

gmov2gif create an *mpeg* movie from the input film.

15.16.2 Usage

```
Usage : gmov2mpeg -f file [options ...]
  Options: -h        -- this help message
           -f        -- input file
           -o        -- output file
           -p        -- name of a palette
           -z        -- zoom factor
           -d        -- pause between images (ms)
           -b        -- number of initial image
           -e        -- number of final image
           -s        -- number of images to leap
           -l        -- add the label (time)
           -i        -- gives information on the film
           -k        -- keep temporary images
           -t        -- add a text
           --cosmo   -- cosmoligical option
           --version -- displays version
```

15.16.3 Note

This scripts needs the program **convert** and **ppmtompeg** to be installed.

```
[lunix@lunix ~/films]$ gmov2mpeg -f film.gmv -o film.gif -p rainbow4
tmp/0000_0500.000
tmp/0001_1000.000
tmp/0002_1500.000
tmp/0003_2000.000
...
```