PSDF: Particle Stream Data Format for N-Body Simulations

Will Farr[a,*], Jeff Ames[1], Piet Hut[c], Junichiro Makino[b], Steve McMillan[1], Takayuki Muranushi[1], Koichi Nakamura[1], Keigo Nitadori[1], Simon Portegies Zwart[1]

[a] *Northwestern University Center for Interdisciplinary Research in Astrophysics, 2145 Sheridan Rd., Evanston IL 60208*
[b] *Interactive Research Center of Science, Graduate School of Science and Engineering Tokyo Institute of Technology, 2–12–1 Ookayama, Meguro, Tokyo 152-8551, Japan*
[c] *Institute for Advanced Study, Princeton, NJ 08540, USA*

Abstract

We present a data format for the output of general N-body simulations, allowing the presence of individual time steps. By specifying a standard, different N-body integrators and different visualization and analysis programs can all share the simulation data, independent of the type of programs used to produce the data. Our Particle Stream Data Format, PSDF, is specified in YAML, based on the same approach as XML but with a simpler syntax. Together with a specification of PSDF, we provide background and motivation, as well as specific examples in a variety of computer languages. We also offer a web site from which these examples can be retrieved, in order to make it easy to augment existing codes in order to give them the option to produce PSDF output.

Keywords: Stellar dynamics, Method: N-body simulation

1. Introduction

The simplest N-body calculations use a shared time step length for all particles. This implies a rather simple structure of the output. With N particles and k time steps, the output takes on the form of an $N * k$ matrix of particle data, where the latter typically contain the mass, position and velocity of a single particle at a specific time, with possible additional information such as higher derivatives of the position (such as acceleration, jerk, etc.), the value of the potential at the position of that particle, and so on. The output of this matrix can be done by ordering in time or by ordering by the identity of particles, in which case each world line is output separately.

Some complications may occur when particles are removed, for example because they are escaping from the system, or because they represent a star that undergoes a destructive supernova leaving no remnant. However, the basic I/O structure is simple enough that it is easy to present these kinds of data in one of the standard data formats, such as FITS or HDF, with a brief description of what is what.

The situation gets vastly more complicated, though, when we allow for individual time steps. Simulations of dense stellar systems, such as open and globular star clusters, as well as galactic nuclei, have relied on the use of individual time steps very early on, already in the 1960s. The reason is that the presence of close binaries and triples in such systems would increase the computer power needed by orders of magnitude in case of shared time steps, compared to individual time steps. In addition, cosmological codes, too, are moving toward the use of individual timesteps, given the increasingly large discrepancies of intrinsic time scales that come with increasingly high spatial resolution.

The simplest way to output data from individual time step codes would be to stick to shared time steps. Indeed, typical legacy codes, such as NBODY6, do just that by default. If all one wants to do is to make a fixed movie of a simulation run, that approach suffices. However, when we interactively inspect the results of a simulation run, we want to be able to zoom in and out, and speed up and slow down the rate at which we run the graphics presentation of the run. With a fixed initial output rate, it may not be possible to interpolate the motion of the particles that move at high speeds. Phrased differently, an output rate high enough to faithfully present the motion of all particles may be prohibitively expensive in terms of memory. It would be much better to let the graphics program itself decide how and where to extrapolate, given the original data it has received from a simulations code.

For example, when we display the dense center of a star cluster, the graphics program can then use the full information for the rapidly moving particles, while interpolating the data for the slower halo particles. Such an approach can easily save orders of magnitude of memory storage requirement. An implementation of this approach was made by Steve McMillan (199? – **Steve, please provide reference and a few-line summary**). However, this implementation was handcrafted for a specific code,

*Corresponding author
Email address: w-farr@northwestern.edu (Will Farr)

reflecting the data structure used in that code. Clearly, it would be desirable to have a more universal data format that would allow different codes to share data in a more transparent way.

Other concerns are to make a data format standard machine independent, to make allowance for parallel processing, and to avoid serious overhead penalties with respect to performance (**Jun, do you want to add a few lines here?**)

2. Basic idea

We need to store the data sufficient to reconstruct the orbits of individual particles. Conceptually, what we need is a stream of phase-space information of particles, such as:

```
particle_id, time, mass, x, y, z, vx, vy, vz, ...
particle_id, time, mass, x, y, z, vx, vy, vz, ...
particle_id, time, mass, x, y, z, vx, vy, vz, ...
```

However, the data format must be flexible enough to be able to include more information such as

- radius, and other info related to stellar evolution
- merger history
- whatever else one can think of

One way to construct such a flexible data format is to use self-describing data format, such as XML or YAML. For simplicity, we adopt YAML here.

2.1. Some basics of YAML

The following is a simple example of a data in YAML format.

```
--- !!Particle
id: 0
r:
  - 0.1
  - 0.2
  - 0.3
v:
  - -1
  - -2
  - -3
m: 1.0
```

In the above example, the line

```
--- !!Particle
```

Is the header, which indicates that it describes the data of an object of type `Particle`.

```
id: 0
```

defines a field with name "id", and value 0.

```
r:
  - 0.1
  - 0.2
  - 0.3
```

means the field "r" is an array with three elements. The first "-" means this line is a data for an array. By default, numbers without "." are regarded as integers, and with "." floating point. Note that indentation has meaning here and "-" must be indented the same level or deeper than "r" and should be aligned. Here, "v" and "m" are similar.

3. Particle Stream Data Format

With the minimal description of YAML in the previous section, now we can define the generic data format (not yet for binaries, though): The data format is the stream of YAML representation of particle object, like

```
--- !!Particle
id: 0
t: 0
r:
  - 0.1
  - 0.2
  - 0.3
v:
  - -1
  - -2
  - -3
m: 1.0
--- !!Particle
id: 1
t: 0
r:
  - 0.2
  - 0.3
  - 0.4
v:
  - 0
  - 0
  - 0
m: 1.0
....
```

3.1. Formalities

For a data format to be understandable by a computer program, there need to be some convension. We start with the list of reserved words for the names:

name	description
id	index (can be arbitrary text)
m	mass
t	time
t_max	max time to which this record is valid
r	position, array with three elements
v	velocity, array with three elements
pot	potential
acc	acceleration, array with three elements
jerk	jerk, array with three elements
snap	snap, array with three elements
crackle	crackle, array with three elements
pop	pop, array with three elements

We require that time, position, velocity and higher derivatives are consistent (that if position is given in parsec and time in year, velocity must be in parsec/year, not km/s, for example). Also, we expect the users of this format to follow the convension of using these names to descrive these physical quantities. Here, `t_max` is rather special, in that it gives the maximum possible time that this record is used to predict the orbit of this particle.

Any of these fields are not required. One can make a record without velocity, mass, time or whatsoever. If a program needs the positions of particles but the data file does not give them, the program should raise error message. We will give example APIs in several languages in the appendix.

In the a data file, if the record of one particle is not produced after certain time, the analysis/visualization program regard that it somehow vanished after time `t_max`.

We considered the possibility of defining a special record for creation and destruction of a particle, but decided against that to keep the parser as simple as possible to write.

We call this format with basic naming convention PSDF (Particle Stream Data Format). If one wants to add his/her own data, one possibility is to use a name like "NBODY6_predicted_position". We do not enforce the use of keyword "X_", but are not against it.

We call a YAML document which corresponds to a single particle object as "worldpoint", since it is a point on the worldline associated with a particle.

4. Some remarks

4.1. File format

PSDF does not define the on-memory data description. It does not define how the individual YAML documents are stored.

There are variety of ways to store worldpoints in file(s). In one extreme, all worldpoints are stored in a single big file, in the order defined by the time sequence. In the opposite extreme, one worldpoint is stored in one file. We could think of anything in between, such as single file for one particle or single file for some period in time.

In order to do some parallel I/O, using multiple files may be more natural. We do not yet specify any particular form of parallel access, though. We might provide some examples, though.

5. Repository

`git@github.com:jmakino/Particle-Stream-Data-Format.g`

you can download it by:

```
mkdir foo
cd foo
git clone git://github.com/jmakino/Particle-Stream-Da
```

To update:

```
git pull
```

6. Sample codes

Also in the github.

6.1. Base library

psdf.rb:

```
require 'yaml'
class Particle
  def taguri
    return 'x-private:Particle'
  end
end
YAML.add_private_type('Particle') do |type, val|
  YAML.object_maker(Particle, val)
end
```

This is current minimal "library", which defines particle class and "Particle" tag.

6.2. Simple program to generate a psdf file

writetest.rb:

```
require "psdf.rb"
class Particle
  attr_accessor :id, :r
  def initialize
    @id=0
    @t=0
    @r=[0,1,2]
  end
end
(0..10).each{|id|
  obj=Particle.new
  obj.id =id;
  obj.x[0]=id*0.1
  print YAML.dump(obj)
}
```

This one generate 10 worldpoints.

6.3. Simple program to read a psdf file

readtest.rb:

```
require "psdf.rb"
a = []
while s = gets("--- ")
  print s
  print "\n\nend of one gets\n\n"
  print s, "\n"
  obj = YAML.load(s)
  a.push obj if obj
end
p a
```

This one reads in 10 particles and store everything in a single array.

6.4. An ACS Body method to write a psdf worldpoint

acs_psdf.rb:

```
require "psdf.rb"
class Particle
  attr_accessor :id, :r, :v, :a, :p, :j, :m, :t, :dt
end

class Body
  def to_psdf
    obj=Particle.new
    obj.id = @body_id
    obj.t  = @time
    obj.r = Array[*@pos]
    obj.v = Array[*@vel]
    obj.a = Array[*@acc]
    obj.j = Array[*@jerk]
    obj
  end
  def psdf_output
    print YAML.dump(self.to_psdf)
  end
end
```

In the ACS Body class, position, velocity and other vector data are of class `Vector`, not `Array`. So you need to convert them to `Array`.

6.5. An OpenGL animation program

The file testplot.rb gives a simple example. It it intended to be a skeleton code, with minimal UI and other stuff. It takes an input file with name "testin", in which a sequence of psdf world points is stored (can come from shared-timestep or individual-timestep code).

This program requires ruby-opengl library. To install this (if not already installed) try:

```
sudo gem install opengl
```

On my machine with CentOS 5.5, it failed with:

```
sudo gem install ruby-opengl
Building native extensions.  This could take a while...
ERROR:  Error installing ruby-opengl:
        ERROR: Failed to build gem native extension.

/usr/bin/ruby -rubygems /usr/lib/ruby/gems/1.8/gems/rak
RUBYARCHDIR=3D/usr/lib/ruby/gems/1.8/gems/ruby-opengl-0
RUBYLIBDIR=3D/usr/lib/ruby/gems/1.8/gems/ruby-opengl-0.
/usr/bin/ruby mkrf_conf.rb
(in /usr/lib/ruby/gems/1.8/gems/ruby-opengl-0.60.1)
rake
gcc  -fPIC -g -O2    -Wall -DRUBY_VERSION=3D187  -I/usr
-I/usr/lib/ruby/1.8/x86_64-linux -I/usr/lib/ruby/site_r
gl-enums.c
(in /usr/lib/ruby/gems/1.8/gems/ruby-opengl-0.60.1/ext/
gl-enums.c:6  include :
../common/common.h:45:21: error: GL/glut.h:=20
                                                rake abor
Command failed with status (1): [gcc  -fPIC -g -O2    -
-DRUBY_VERSION=3D...]
/usr/lib/ruby/gems/1.8/gems/ruby-opengl-0.60.1/ext/gl/R
(See full trace by running task with --trace)
rake aborted!
Command failed with status (1): [rake...]

(See full trace by running task with --trace)


Gem files will remain installed in =
/usr/lib/ruby/gems/1.8/gems/ruby-opengl-0.60.1 for insp
Results logged to =
/usr/lib/ruby/gems/1.8/gems/ruby-opengl-0.60.1/gem_make
```

In this case, you need to install GLUT library, by

```
sudo yum install freeglut   freeglut-devel
```

Acknowledgment

References