
Reference Documentation

Release latest

The AMUSE Team

May 04, 2012

CONTENTS

1	Interface Specifications	1
1.1	Introduction	1
1.2	Stellar Dynamics Interface Definition	4
1.3	Stellar Evolution Interface Definition	16
1.4	Hydrodynamics Interface Definition	17
1.5	Radiative Transfer Interface Definition	23
2	Message protocol between python and community codes	25
2.1	Introduction	25
2.2	Overall Operation	25
2.3	Message format	25
2.4	Examples	27
3	Support code for AMUSE framework	29
4	Support code for the community code interfaces	33
5	Currently supported Community Codes	35
5.1	Introduction	35
5.2	Stellar Dynamics	35
5.3	Stellar Evolution	47
5.4	Hydrodynamics	51
5.5	Radiative Transfer	60
6	Supported File Formats	63
6.1	Introduction	63
6.2	Use	63
6.3	Exceptions	64
6.4	Text files	65
6.5	Starlab	66
6.6	NEMO	67
6.7	Gadget	68
7	Reporting during a run	69
8	Quantities and Units	71
8.1	Introduction	71
8.2	Quantities	71
8.3	Units	77
8.4	Unit systems and converters	81
9	Datamodel	85
9.1	Introduction	85
9.2	Identity	85

9.3	Sets of particles	86
10	Particle Attributes	97
11	From Codes to Data	103
11.1	Introduction	103
11.2	The first level	103
12	Sets and Grids in Codes	105
13	Stopping Conditions	111
13.1	Introduction	111
13.2	Using stopping conditions	113
13.3	Implementing stopping conditions	114
13.4	Using the stopping conditions library	120
14	Input / Output Framework	125
14.1	Introduction	125
14.2	Extending	125
15	Options	127
15.1	Introduction	127
15.2	Configuration file location	127
15.3	Configuration file format	127
15.4	Option values	127
15.5	Sections lookup	128
15.6	Use	128
16	Directory Structure	131
16.1	The <code>src</code> directories	131
16.2	The <code>test</code> directories	131
16.3	The <code>support</code> directories	132
17	Setting up GPGPU with CUDA	133
17.1	Introduction	133
17.2	Self-help script	133
17.3	Step-by-step	133
18	Distributed AMUSE	135
18.1	Installation	135
18.2	Specifying resources using <code>Jungle</code> files	135
18.3	Starting <code>Ibis Deploy</code>	135
18.4	Running workers remotely	136
19	AMUSE Style Guide	137
19.1	Python Code	137
19.2	C / C++ code	138
19.3	Fortran code	138
20	Source Code Management	139
20.1	Committing Code	139
20.2	SVN repository location	139
21	Glossary	141
	Python Module Index	143
	Index	145

INTERFACE SPECIFICATIONS

1.1 Introduction

In this document we will specify the low-level interfaces of the community codes. These interfaces should follow a pattern. This pattern is described below. This pattern can be used to keep the interfaces of the codes consistent between codes of different physical domains and to help in learning about the functionality of the interfaces. In later chapters we define the specific interfaces for the modules of different physical domains. The actual interface of a code should derive from these interfaces.

These interfaces only specify the low-level interfaces. The low-level interfaces do not support unit handling, data conversion and attributes. This functionality is handled by the next-level interfaces described in [nextlevel_](#).

1.1.1 Data Types

The exact size (in bytes) and type of the values sent between the python layer and the community codes is very important. In this document we will specify the type of each argument and return value. Currently AMUSE supports 4 types, these are described in the following table.

Type name	Size bytes	Fortran type	C type
int32	4	integer	long
float64	8	double precision	double
float32	4	real	float
string	n		char *

1.1.2 Function template

All functions in the interface should follow the same template. Each function returns an error or status code. Results are returned through the function arguments. In C these arguments need to be pointers to valid memory locations.

```
class amuse.community.interface.example.ExampleInterface
```

example_function

Example template for the other functions defined in this specification. All functions should follow this example..

```
int32 example_function(int32 input, float64 * output, float64 * inout);
```

```
FUNCTION example_function(input, output, inout)
  INTEGER :: input
  DOUBLE PRECISION :: output, inout
  INTEGER :: example_function
END FUNCTION
```

Parameters

- **input** (*int32, IN*) – Typical input parameter, the argument is passed by value to the function.
- **output** (*float64, OUT*) – Typical output parameter, the argument is passed by reference. The argument should point to a valid memory location.
- **inout** (*float64, INOUT*) – Some arguments can be both input as well as output. The function will update the value of the passed argument.

Returns Function will return an error code.

The error codes all have the same general form. Zero stands for no error, a negative value indicates some error happened, a positive value is returned when the function ends in a special state.

0 - OK Function encountered no error or special state

<0 - ERROR Something went wrong in the execution of the function

>0 - STATE Function has encountered an expected special state. For example the code has detected a collision between two stars.

1.1.3 Function categories

Parameters

Codes can have a number of parameters. Some parameters are domain specific, these parameters are found for all codes in a specific domain (for example the smoothing length in gravitational dynamics). Other parameters are only defined and used by a specific code. The domain specific parameters are defined on the domain specific interfaces, when a code supports the parameter it should implement the specified functions. Other parameters have to be accessed with functions following the template of the `get_example_parameter()` and `set_example_parameter()` functions.

class `amuse.community.interface.example.ExampleInterface`

get_example_parameter

Retrieve the current value of the parameter. Note, values can be any of the supported types.

```
int32 get_example_parameter(float64 * value);
```

```
FUNCTION get_example_parameter(value)
  DOUBLE PRECISION :: value
  INTEGER :: get_example_parameter
END FUNCTION
```

Parameters *value* (*float64, OUT*) – The current value of the parameter.

Returns

0 - OK Current value was retrieved

-1 - ERROR The code does not have support for this parameter, use this when a code does not support a parameter pre-defined in a physical domain

set_example_parameter

Update the value of the parameter. The type of the new value argument must be the same as the `get_example_parameter()` function.

```
int32 set_example_parameter(float64 value);
```

```

FUNCTION set_example_parameter(value)
  DOUBLE PRECISION :: value
  INTEGER :: set_example_parameter
END FUNCTION

```

Parameters `value` (*float64*, *IN*) – The new value of the parameter.

Returns

- 0 - OK** New value of the parameter was set
- 1 - ERROR** The code does not have support for this parameter

A function used to access (set or get) a parameter may only retrieve or update the value of a single parameter. Functions setting two or more parameters in one go are not supported by the next-level interfaces. After all parameters have been set, the `initialize_code()` function should be called, this gives the code the opportunity to prepare the model.

class `amuse.community.interface.example.ExampleInterface`

`initialize_code`

Let the code perform initialization actions after all parameters have been set. Should be called once per running code instance.

```
int32 initialize_code();
```

```

FUNCTION initialize_code()
  INTEGER :: initialize_code
END FUNCTION

```

Returns

- 0 - OK** Code is initialized
- 1 - ERROR** Error happened during initialization, this error needs to be further specified by every code implementation

Object Management

Codes can work on particles or grids (stars, black holes or gas). The methods in the *Object Management* category define the functionality to create, remove and query the particles or gridpoints in the codes.

When a code supports objects, the code is responsible for managing these objects. The code needs to assign a unique index to a particle so that the particle can be referred to in other function calls. This is a *major* difference with the MUSE code. Where the user of the code was responsible for assigning unique ids to the particles. This change makes the implementation of the code simpler and allows the code to support creation of new objects during simulation. For example a hydrocode can add or delete gridpoints during the evolution of the model.

Object state

Particles in the same physical domain can have a well known, *minimal* state. For example, in the gravitational dynamics domain the state of a particle can be defined by a location vector, velocity vector, a mass and a radius. The methods in the *Object State* category provide a way to access this state in one function.

Object State, Extension Mechanism

Not all information of a particle can be transferred with the `get_state` and `set_state` functions. Some codes may support other properties of a particle, the code can define `get_` and `set_` functions for these properties. These

functions follow the pattern defined in the *Parameters* category. The functions must either get or set a scalar property (1 argument) or a vector property (3 arguments).

Model evolution

The main function of a code is often evolving a model in time or solving a steady state solution. The methods that control model evolution or start and stop the model calculations all belong to the *Model evolution* category. At this time, no pattern is defined for the functions in this category.

Diagnostics

The state of the code can be queried, before, during and after the model calculations. All the functions in this category follow the 'get_name' pattern. The state of code should not change during a function call to a function in this category. The functions must either get a scalar property (1 argument) or a vector property (3 arguments).

Services

Some codes can provide services for other codes in the same or other physical domains. For example, gravitational dynamics code might provide a function to calculate the gravity force at a point. The methods that provide these services all belong to this category.

1.2 Stellar Dynamics Interface Definition

Date	Author(s)	Version	State
13-10-2009	AvE	0.1	Initial
20-10-2009	AvE	0.2	Initial, after first review with IP
02-11-2009	AvE	0.3	Initial, after second review with MR

1.2.1 Introduction

In this chapter we describe the common interface for all stellar dynamics codes.

1.2.2 Parameters

Gravity dynamics codes have at least one specified parameter. Other parameters need to be specified on a per code basis. All parameters have to be accessed with functions following the template of the `get_eps` and `set_eps` functions. A parameter access function may only retrieve or update the value of a single parameter. After all parameters have been set, the `initialize_code` function should be called, this gives the code the opportunity to prepare the model.

```
class amuse.community.interface.gd.GravitationalDynamicsInterface
```

```
    get_eps2
```

```
        Retrieve the current value of the squared smoothing parameter.
```

```
        int32 get_eps2(float64 * epsilon_squared);
```

```
    FUNCTION get_eps2(epsilon_squared)
        DOUBLE PRECISION :: epsilon_squared
        INTEGER :: get_eps2
    END FUNCTION
```

Parameters `epsilon_squared` (*float64, OUT*) – The current value of the smoothing parameter, squared.

Returns

- 0 - OK** Current value of the smoothing parameter was set
- 1 - ERROR** The code does not have support for a smoothing parameter

set_eps2

Update the value of the squared smoothing parameter.

```
int32 set_eps2(float64 epsilon_squared);
```

```
FUNCTION set_eps2(epsilon_squared)
  DOUBLE PRECISION :: epsilon_squared
  INTEGER :: set_eps2
END FUNCTION
```

Parameters `epsilon_squared` (*float64, IN*) – The new value of the smoothing parameter, squared.

Returns

- 0 - OK** Current value of the smoothing parameter was set
- 1 - ERROR** The code does not have support for a smoothing parameter

initialize_code

Run the initialization for the code, called before any other call on the code (so before any parameters are set or particles are defined in the code).

```
int32 initialize_code();
```

```
FUNCTION initialize_code()
  INTEGER :: initialize_code
END FUNCTION
```

Returns

- 0 - OK** Code is initialized
- 1 - ERROR** Error happened during initialization, this error needs to be further specified by every code implementation
- 2 - ERROR** not yet implemented

1.2.3 Object Management

Most gravitational dynamics codes work on particles (stars, black holes or gas). The following methods define the functionality to create, remove and query the particles in the code. *Currently the interface does not handle different types of particles*

```
class amuse.community.interface.gd.GravitationalDynamicsInterface
```

new_particle

Define a new particle in the stellar dynamics code. The particle is initialized with the provided mass, radius, position and velocity. This function returns an index that can be used to refer to this particle.

```
int32 new_particle(int32 * index_of_the_particle, float64 mass,
  float64 x, float64 y, float64 z, float64 vx, float64 vy, float64 vz,
  float64 radius);
```

```
FUNCTION new_particle(index_of_the_particle, mass, x, y, z, vx, vy, vz, &
    radius)
    INTEGER :: index_of_the_particle
    DOUBLE PRECISION :: mass, x, y, z, vx, vy, vz, radius
    INTEGER :: new_particle
END FUNCTION
```

Parameters

- **index_of_the_particle** (*int32, OUT*) – An index assigned to the newly created particle. This index is supposed to be a local index for the code (and not valid in other instances of the code or in other codes)
- **mass** (*float64, IN*) – The mass of the particle
- **x** (*float64, IN*) – The initial position vector of the particle
- **y** (*float64, IN*) – The initial position vector of the particle
- **z** (*float64, IN*) – The initial position vector of the particle
- **vx** (*float64, IN*) – The initial velocity vector of the particle
- **vy** (*float64, IN*) – The initial velocity vector of the particle
- **vz** (*float64, IN*) – The initial velocity vector of the particle
- **radius** (*float64, IN*) – The radius of the particle

Returns

- **0 - OK** particle was created and added to the model
- **-1 - ERROR** particle could not be created

delete_particle

Remove the definition of particle from the code. After calling this function the particle is no longer part of the model evolution. It is up to the code if the index will be reused. This function is optional.

```
int32 delete_particle(int32 index_of_the_particle);
```

```
FUNCTION delete_particle(index_of_the_particle)
    INTEGER :: index_of_the_particle
    INTEGER :: delete_particle
END FUNCTION
```

Parameters **index_of_the_particle** (*int32, IN*) – Index of the particle to be removed. This index must have been returned by an earlier call to `new_particle()`

Returns

- **0 - OK** particle was removed from the model
- **-1 - ERROR** particle could not be removed
- **-2 - ERROR** not yet implemented

get_number_of_particles

Retrieve the total number of particles defined in the code

```
int32 get_number_of_particles(int32 * number_of_particles);
```

```
FUNCTION get_number_of_particles(number_of_particles)
    INTEGER :: number_of_particles
    INTEGER :: get_number_of_particles
END FUNCTION
```

Parameters `number_of_particles` (*int32*, *OUT*) – Count of the particles in the code

Returns

- 0 - OK** Count could be determined
- 1 - ERROR** Unable to determine the count

get_index_of_first_particle

Retrieve the index of first particle. When this index is used as the starting index for the `get_index_of_next_particle` method, all particles can be iterated over:

```
error, first_index = instance.get_index_of_first_particle()
current_index = first_index
while error == 0:
    status, mass = instance.get_mass(current_index)
    print mass
    error, current_index = instance.get_index_of_next_particle(current_index)
```

```
int32 get_index_of_first_particle(int32 * index_of_the_particle);
```

```
FUNCTION get_index_of_first_particle(index_of_the_particle)
    INTEGER :: index_of_the_particle
    INTEGER :: get_index_of_first_particle
END FUNCTION
```

Parameters `index_of_the_particle` (*int32*, *OUT*) – Index of the first particle

Returns

- 0 - OK** Index was set
- 1 - ERROR** Code has no particles, or cannot set the index

get_index_of_next_particle

Retrieve the index of the particle following the provided index. The iteration direction is determined by the code.

```
int32 get_index_of_next_particle(int32 index_of_the_particle,
    int32 * index_of_the_next_particle);
```

```
FUNCTION get_index_of_next_particle(index_of_the_particle, &
    index_of_the_next_particle)
    INTEGER :: index_of_the_particle, index_of_the_next_particle
    INTEGER :: get_index_of_next_particle
END FUNCTION
```

Parameters

- `index_of_the_particle` (*int32*, *IN*) – Index of the particle
- `index_of_the_next_particle` (*int32*, *OUT*) – Index of the particle following the particle with the provided index

Returns

- 0 - OK** Index was set
- 1 - STATE** Last index was reached
- 1 - ERROR** Particle could not be found

1.2.4 Object state

Particles in gravitational dynamics have a well known, *minimal* state. This state is defined by a location, velocity and mass and radius. The state can be retrieved and updated with the following functions.

class `amuse.community.interface.gd.GravitationalDynamicsInterface`

`get_state`

Retrieve the current state of a particle. The *minimal* information of a stellar dynamics particle (mass, radius, position and velocity) is returned.

```
int32 get_state(int32 index_of_the_particle, float64 * mass, float64 * x,
               float64 * y, float64 * z, float64 * vx, float64 * vy, float64 * vz,
               float64 * radius);
```

```
FUNCTION get_state(index_of_the_particle, mass, x, y, z, vx, vy, vz, &
                   radius)
INTEGER :: index_of_the_particle
DOUBLE PRECISION :: mass, x, y, z, vx, vy, vz, radius
INTEGER :: get_state
END FUNCTION
```

Parameters

- **index_of_the_particle** (*int32, IN*) – Index of the particle to get the state from. This index must have been returned by an earlier call to `new_particle()`
- **mass** (*float64, OUT*) – The current mass of the particle
- **x** (*float64, OUT*) – The current position vector of the particle
- **y** (*float64, OUT*) – The current position vector of the particle
- **z** (*float64, OUT*) – The current position vector of the particle
- **vx** (*float64, OUT*) – The current velocity vector of the particle
- **vy** (*float64, OUT*) – The current velocity vector of the particle
- **vz** (*float64, OUT*) – The current velocity vector of the particle
- **radius** (*float64, OUT*) – The current radius of the particle

Returns

- **0 - OK** particle was removed from the model
- **-1 - ERROR** particle could not be found

`set_state`

Update the current state of a particle. The *minimal* information of a stellar dynamics particle (mass, radius, position and velocity) is updated.

```
int32 set_state(int32 index_of_the_particle, float64 mass, float64 x,
               float64 y, float64 z, float64 vx, float64 vy, float64 vz,
               float64 radius);
```

```
FUNCTION set_state(index_of_the_particle, mass, x, y, z, vx, vy, vz, &
                   radius)
INTEGER :: index_of_the_particle
DOUBLE PRECISION :: mass, x, y, z, vx, vy, vz, radius
INTEGER :: set_state
END FUNCTION
```

Parameters

- **index_of_the_particle** (*int32, IN*) – Index of the particle for which the state is to be updated. This index must have been returned by an earlier call to `new_particle()`
- **mass** (*float64, IN*) – The new mass of the particle
- **x** (*float64, IN*) – The new position vector of the particle
- **y** (*float64, IN*) – The new position vector of the particle
- **z** (*float64, IN*) – The new position vector of the particle
- **vx** (*float64, IN*) – The new velocity vector of the particle
- **vy** (*float64, IN*) – The new velocity vector of the particle
- **vz** (*float64, IN*) – The new velocity vector of the particle
- **radius** (*float64, IN*) – The new radius of the particle

Returns

- 0 - OK** particle was found in the model and the information was set
- 1 - ERROR** particle could not be found
- 2 - ERROR** code does not support updating of a particle
- 3 - ERROR** not yet implemented

1.2.5 Object State, Extension Mechanism

Not all information of a particle can be transferred with the `get_state` and `set_state` functions. To support other properties (like acceleration), the code can define `get_` and `set_` functions. These functions must get or set one scalar property (1 argument) or a vector property (3 arguments)

```
class amuse.community.interface.gd.GravitationalDynamicsInterface
```

`get_mass`

Retrieve the mass of a particle. Mass is a scalar property of a particle, this function has one OUT argument.

```
int32 get_mass(int32 index_of_the_particle, float64 * mass);
```

```
FUNCTION get_mass(index_of_the_particle, mass)
  INTEGER :: index_of_the_particle
  DOUBLE PRECISION :: mass
  INTEGER :: get_mass
END FUNCTION
```

Parameters

- **index_of_the_particle** (*int32, IN*) – Index of the particle to get the state from. This index must have been returned by an earlier call to `new_particle()`
- **mass** (*float64, OUT*) – The current mass of the particle

Returns

- 0 - OK** particle was removed from the model
- 1 - ERROR** particle could not be found

`set_mass`

Update the mass of a particle. Mass is a scalar property of a particle.

```
int32 set_mass(int32 index_of_the_particle, float64 mass);
```

```
FUNCTION set_mass(index_of_the_particle, mass)
  INTEGER :: index_of_the_particle
  DOUBLE PRECISION :: mass
  INTEGER :: set_mass
END FUNCTION
```

Parameters

- **index_of_the_particle** (*int32, IN*) – Index of the particle for which the state is to be updated. This index must have been returned by an earlier call to `new_particle()`
- **mass** (*float64, IN*) – The new mass of the particle

Returns

- **0 - OK** particle was found in the model and the information was set
- **-1 - ERROR** particle could not be found
- **-2 - ERROR** code does not support updating of a particle

get_position

Retrieve the position vector of a particle. Position is a vector property, this function has 3 OUT arguments.

```
int32 get_position(int32 index_of_the_particle, float64 * x, float64 * y,
  float64 * z);
```

```
FUNCTION get_position(index_of_the_particle, x, y, z)
  INTEGER :: index_of_the_particle
  DOUBLE PRECISION :: x, y, z
  INTEGER :: get_position
END FUNCTION
```

Parameters

- **index_of_the_particle** (*int32, IN*) – Index of the particle to get the state from. This index must have been returned by an earlier call to `new_particle()`
- **x** (*float64, OUT*) – The current position vector of the particle
- **y** (*float64, OUT*) – The current position vector of the particle
- **z** (*float64, OUT*) – The current position vector of the particle

Returns

- **0 - OK** current value was retrieved
- **-1 - ERROR** particle could not be found
- **-2 - ERROR** not yet implemented

set_position

Update the position of a particle.

```
int32 set_position(int32 index_of_the_particle, float64 x, float64 y,
  float64 z);
```

```
FUNCTION set_position(index_of_the_particle, x, y, z)
  INTEGER :: index_of_the_particle
  DOUBLE PRECISION :: x, y, z
  INTEGER :: set_position
END FUNCTION
```

Parameters

- **index_of_the_particle** (*int32, IN*) – Index of the particle for which the state is to be updated. This index must have been returned by an earlier call to `new_particle()`
- **x** (*float64, IN*) – The new position vector of the particle
- **y** (*float64, IN*) – The new position vector of the particle
- **z** (*float64, IN*) – The new position vector of the particle

Returns

- 0 - OK** particle was found in the model and the information was set
- 1 - ERROR** particle could not be found
- 2 - ERROR** code does not support updating of a particle

set_acceleration

Update the acceleration of a particle. *Defined for symetry with the get_acceleration function. Should be removed if physaccily unsound Maybe moved to snapshot support functionality*

```
int32 set_acceleration(int32 index_of_the_particle, float64 ax,
    float64 ay, float64 az);
```

```
FUNCTION set_acceleration(index_of_the_particle, ax, ay, az)
    INTEGER :: index_of_the_particle
    DOUBLE PRECISION :: ax, ay, az
    INTEGER :: set_acceleration
END FUNCTION
```

Parameters

- **index_of_the_particle** (*int32, IN*) – Index of the particle for which the state is to be updated. This index must have been returned by an earlier call to `new_particle()`
- **ax** (*float64, IN*) – The new acceleration vector of the particle
- **ay** (*float64, IN*) – The new acceleration vector of the particle
- **az** (*float64, IN*) – The new acceleration vector of the particle

Returns

- 0 - OK** particle was found in the model and the information was set
- 1 - ERROR** particle could not be found
- 2 - ERROR** code does not support updating of a particle
- 3 - ERROR** not yet implemented

get_acceleration

Retrieve the acceleration vector of a particle. Second time derivative of the position.

```
int32 get_acceleration(int32 index_of_the_particle, float64 * ax,
    float64 * ay, float64 * az);
```

```
FUNCTION get_acceleration(index_of_the_particle, ax, ay, az)
    INTEGER :: index_of_the_particle
    DOUBLE PRECISION :: ax, ay, az
    INTEGER :: get_acceleration
END FUNCTION
```

Parameters

- **index_of_the_particle** (*int32, IN*) – Index of the particle to get the state from. This index must have been returned by an earlier call to `new_particle()`
- **ax** (*float64, OUT*) – The current position vector of the particle

- **ay** (*float64, OUT*) – The current position vector of the particle
- **az** (*float64, OUT*) – The current position vector of the particle

Returns

- 0 - OK** current value was retrieved
- 1 - ERROR** particle could not be found
- 2 - ERROR** not yet implemented

get_potential

Retrieve the potential at a particle position, for retrieving the potential anywhere in the field use `get_potential_at_point`.

```
int32 get_potential(int32 index_of_the_particle, float64 * potential);
```

```
FUNCTION get_potential(index_of_the_particle, potential)
  INTEGER :: index_of_the_particle
  DOUBLE PRECISION :: potential
  INTEGER :: get_potential
END FUNCTION
```

Parameters

- **index_of_the_particle** (*int32, IN*) – Index of the particle for which the state is to be updated. This index must have been returned by an earlier call to `new_particle()`
- **potential** (*float64, OUT*) – The current scalar potential...

Returns

- 0 - OK** current value was retrieved
- 1 - ERROR** particle could not be found
- 2 - ERROR** not yet implemented

1.2.6 Model evolution

The gravitational dynamics codes evolve the properties of the particles in time. The following functions are needed to control the evolution in the code.

```
class amuse.community.interface.gd.GravitationalDynamicsInterface
```

commit_particles

Let the code perform initialization actions after all particles have been loaded in the model. Should be called before the first `evolve` call and after the last `new_particle` call.

```
int32 commit_particles();
```

```
FUNCTION commit_particles()
  INTEGER :: commit_particles
END FUNCTION
```

Returns

- 0 - OK** Model is initialized and evolution can start
- 1 - ERROR** Error happened during initialization, this error needs to be further specified by every code implementation

1.2.7 Diagnostics

The state of the code can be queried, before, during and after the model calculations. The following function can be used to query the exact model time, the total energies and colliding particles.

class `amuse.community.interface.gd.GravitationalDynamicsInterface`

`get_time`

Retrieve the model time. This time should be close to the end time specified in the evolve code. Or, when a collision was detected, it will be the model time of the collision.

```
int32 get_time(float64 * time);
```

```
FUNCTION get_time(time)
  DOUBLE PRECISION :: time
  INTEGER :: get_time
END FUNCTION
```

Parameters `time` (*float64*, *OUT*) – The current model time

Returns

- 0 - OK** Current value of the time was retrieved
- 1 - ERROR** The code does not have support for querying the time

`get_kinetic_energy`

Retrieve the current kinetic energy of the model

```
int32 get_kinetic_energy(float64 * kinetic_energy);
```

```
FUNCTION get_kinetic_energy(kinetic_energy)
  DOUBLE PRECISION :: kinetic_energy
  INTEGER :: get_kinetic_energy
END FUNCTION
```

Parameters `kinetic_energy` (*float64*, *OUT*) – The kinetic energy of the model

Returns

- 0 - OK** Current value of the kinetic energy was set
- 1 - ERROR** Kinetic energy could not be provided

`get_potential_energy`

Retrieve the current potential energy of the model

```
int32 get_potential_energy(float64 * potential_energy);
```

```
FUNCTION get_potential_energy(potential_energy)
  DOUBLE PRECISION :: potential_energy
  INTEGER :: get_potential_energy
END FUNCTION
```

Parameters `potential_energy` (*float64*, *OUT*) – The potential energy of the model

Returns

- 0 - OK** Current value of the potential energy was set
- 1 - ERROR** Kinetic potential could not be provided

get_center_of_mass_velocity

Retrieve the velocity of the center of mass of all particles. This velocity is mass weighted mean of the velocity of all particles.

```
int32 get_center_of_mass_velocity(float64 * vx, float64 * vy,  
    float64 * vz);
```

```
FUNCTION get_center_of_mass_velocity(vx, vy, vz)  
    DOUBLE PRECISION :: vx, vy, vz  
    INTEGER :: get_center_of_mass_velocity  
END FUNCTION
```

Parameters

- **vx** (*float64, OUT*) – The mean velocity of the model
- **vy** (*float64, OUT*) – The mean velocity of the model
- **vz** (*float64, OUT*) – The mean velocity of the model

Returns

- 0 - OK** Current value of the center of mass velocity was retrieved
- 1 - ERROR** The value could not be provided

get_center_of_mass_position

Retrieve the center of mass (a point in space) of all particles.

```
int32 get_center_of_mass_position(float64 * x, float64 * y, float64 * z);
```

```
FUNCTION get_center_of_mass_position(x, y, z)  
    DOUBLE PRECISION :: x, y, z  
    INTEGER :: get_center_of_mass_position  
END FUNCTION
```

Parameters

- **x** (*float64, OUT*) – The center of mass of the model
- **y** (*float64, OUT*) – The center of mass of the model
- **z** (*float64, OUT*) – The center of mass of the model

Returns

- 0 - OK** Current value of the center was retrieved
- 1 - ERROR** The mass could not be provided
- 2 - ERROR** not yet implemented

get_total_mass

Retrieve the sum of the masses of all particles.

```
int32 get_total_mass(float64 * mass);
```

```
FUNCTION get_total_mass(mass)  
    DOUBLE PRECISION :: mass  
    INTEGER :: get_total_mass  
END FUNCTION
```

Parameters **mass** (*float64, OUT*) – The total mass of the model

Returns

- 0 - OK** Current value of the kinetic mass was retrieved

-1 - **ERROR** Total mass could not be provided

-2 - **ERROR** not yet implemented

get_total_radius

Return the radius of the sphere, centered on the center of mass that contains all the particles. *get_size?*

```
int32 get_total_radius(float64 * radius);
```

```
FUNCTION get_total_radius(radius)
  DOUBLE PRECISION :: radius
  INTEGER :: get_total_radius
END FUNCTION
```

Parameters *radius* (*float64*, *OUT*) – The maximum distance from a star to the center of mass of the model

Returns

0 - **OK** Current value of the radius was retrieved

-1 - **ERROR** The value could not be provided

1.2.8 Services

Some Gravitational Dynamics codes can provide services for other codes. Currently calculating the gravity at a given point is the only specified function.

class `amuse.community.interface.gd.GravitationalDynamicsInterface`

get_gravity_at_point

Determine the gravitational force on a given point

```
int32 get_gravity_at_point(float64 eps, float64 x, float64 y,
  float64 z, float64 * forcex, float64 * forcey, float64 * forcez);
```

```
FUNCTION get_gravity_at_point(eps, x, y, z, forcex, forcey, forcez)
  DOUBLE PRECISION :: eps, x, y, z, forcex, forcey, forcez
  INTEGER :: get_gravity_at_point
END FUNCTION
```

Parameters

- **eps** (*float64*, *IN*) – The smoothing parameter
- **x** (*float64*, *IN*) – The position vector of the point
- **y** (*float64*, *IN*) – The position vector of the point
- **z** (*float64*, *IN*) – The position vector of the point
- **forcex** (*float64*, *OUT*) – Force created by the particles in the code at the given position
- **forcey** (*float64*, *OUT*) – Force created by the particles in the code at the given position
- **forcez** (*float64*, *OUT*) – Force created by the particles in the code at the given position

Returns

0 - **OK** Force could be calculated

-1 - **ERROR** No force calculation supported

1.3 Stellar Evolution Interface Definition

Date	Author(s)	Version	State
13-10-2009	AvE	0.1	Initial
20-10-2009	AvE	0.2	Initial, after first review with IP
02-11-2009	AvE	0.3	Initial, after second review with MR

1.3.1 Introduction

In this chapter we describe the common interface for stellar evolution codes. Currently the interface for stellar evolutions codes that store state of a star is specified. The Stellar Evolution codes that get all the needed state from the function interface (are stateless), are not yet described.

1.3.2 Parameters

Stellar Evolution codes have at least one specified parameter. Other parameters need to be specified on a per code basis. All parameters have to be accessed with functions following the template of the `get_metallicity` and `set_metallicity` functions. A parameter access function may only retrieve or update the value of a single parameter. After all parameters have been set, the `initialize_code` function should be called, this gives the code the opportunity prepare the model.

```
class amuse.community.interface.se.StellarEvolution (legacy_interface, **options)
```

1.3.3 Object Management

A number of stellar evolution codes work on star objects. The following methods define the functionality to create, remove and query the particles in the code. *Currently the interface does not specify query function for stellar evolution, see stellar evolution for possible direction*

```
class amuse.community.interface.se.StellarEvolution (legacy_interface, **options)
```

1.3.4 Object State

To support properties (like acceleration), the code must define `get_` and `set_` functions. These functions must get or set one scalar property (1 argument) or a vector property (3 arguments) *Currently only get functions are specified*

```
class amuse.community.interface.se.StellarEvolution (legacy_interface, **options)
```

1.3.5 Model evolution

The stellar evolution codes evolve the properties of the star in time. The following functions are needed to control the evolution in the code.

```
class amuse.community.interface.se.StellarEvolution (legacy_interface, **options)
```

1.3.6 Diagnostics

The state of the code can be queried, before, during and after the model calculations. *Currently no specific stellar evolution diagnostics functions have been defined*

1.3.7 Services

Some stellar evolution codes can provide services for other codes. *Currently no specific stellar evolution service functions have been defined*

1.4 Hydrodynamics Interface Definition

Date	Author(s)	Version	State
02-11-2009	AvE	0.0	TBD
11-10-2010	AvE	0.1	Initial

1.4.1 Introduction

In this chapter we describe the common interface for all hydrodynamics, grid based codes. For particle based, SPH codes see the gravitational dynamics specifications.

1.4.2 Parameters

All parameters have to be accessed with functions following the template of the `get_timestep` and `set_timestep` functions. A parameter access function may only retrieve or update the value of a single parameter. After all parameters have been set, the `commit_parameters` function should be called, this gives the code the opportunity to prepare the model.

```
class amuse.community.interface.hydro.HydrodynamicsInterface
```

`commit_parameters`

Perform initialization in the code dependent on the values of the parameters. Called after the parameters have been set or updated.

```
int32 commit_parameters();
```

```
FUNCTION commit_parameters()
  INTEGER :: commit_parameters
END FUNCTION
```

Returns

- 0 - OK** Code is initialized
- 1 - ERROR** Error happened during initialization, this error needs to be further specified by every code implementation
- 2 - ERROR** not yet implemented

1.4.3 Grid Management

Most hydrodynamical codes work on grids or a hierarchy of grids. The following methods define the functionality to setup and query the grids.

```
class amuse.community.interface.hydro.HydrodynamicsInterface
```

`get_index_range_inclusive()`

Returns the min and max values of indices in each direction. The range is inclusive, the min index and max index both exist and can be queried. The total number of cells in one direction is $\text{max} - \text{min} + 1$.

The return type is a list of 3 tuples, each tuple contains the minimum and maximum value in the index range.

For C/C++ codes the returned values will usually be: ((0, nmeshx-1), (0, nmeshy-1), (0, nmeshz-1))

For Fortran codes the returned values will usually be: ((1, nmeshx), (1, nmeshy), (1, nmeshz))

set_boundary (*xbound1, xbound2, ybound1, ybound2, zbound1, zbound2*)

Sets the boundary conditions on the grid. Boundaries can be: “reflective”, “periodic”.

Parameters

- **xbound1** – inner or left boundary in the x direction
- **xbound2** – outer or right boundary in the x direction
- **ybound1** – inner or front boundary in the y direction
- **ybound2** – outer or back boundary in the y direction
- **zbound1** – inner or bottom boundary in the z direction
- **zbound2** – outer or top boundary in the z direction

setup_mesh (*nmeshx, nmeshy, nmeshz, xlength, ylength, zlength*)

Sets the number of mesh cells in each direction, and the length of the grid in each direction.

Parameters

- **nmeshx** – number of mesh cells in the x direction
- **nmeshy** – number of mesh cells in the y direction
- **nmeshz** – number of mesh cells in the z direction
- **xlength** – total length of the grid in the x direction
- **ylength** – total length of the grid in the y direction
- **zlength** – total length of the grid in the z direction

get_position_of_index

Retrieves the x, y and z position of the center of the cell with coordinates i, j, k in the grid specified by the `index_of_grid`

```
int32 get_position_of_index(int32 i, int32 j, int32 k,  
    int32 index_of_grid, float64 * x, float64 * y, float64 * z);
```

```
FUNCTION get_position_of_index(i, j, k, index_of_grid, x, y, z)  
    INTEGER :: i, j, k, index_of_grid  
    DOUBLE PRECISION :: x, y, z  
    INTEGER :: get_position_of_index  
END FUNCTION
```

Parameters

- **i** (*int32, IN*) –
- **j** (*int32, IN*) –
- **k** (*int32, IN*) –
- **index_of_grid** (*int32, IN*) –
- **x** (*float64, OUT*) –
- **y** (*float64, OUT*) –
- **z** (*float64, OUT*) –

Returns

get_index_of_position

Retrieves the i, j and k index of the grid cell containing the given x, y and z position. The cell is looked up in the grid specified by `index_of_grid`.

```
int32 get_index_of_position(float64 x, float64 y, float64 z,
    int32 index_of_grid, float64 * i, float64 * j, float64 * k);
```

```
FUNCTION get_index_of_position(x, y, z, index_of_grid, i, j, k)
    INTEGER :: index_of_grid
    DOUBLE PRECISION :: x, y, z, i, j, k
    INTEGER :: get_index_of_position
END FUNCTION
```

Parameters

- **x** (*float64, IN*) –
- **y** (*float64, IN*) –
- **z** (*float64, IN*) –
- **index_of_grid** (*int32, IN*) –
- **i** (*float64, OUT*) –
- **j** (*float64, OUT*) –
- **k** (*float64, OUT*) –

Returns

1.4.4 Grid state

Grid points in a hydrodynamics code have a well known, *minimal* state. This state is defined by a density, momentum density and energy density. The state can be retrieved and updated with the following functions.

class `amuse.community.interface.hydro.HydrodynamicsInterface`

set_grid_state

```
int32 set_grid_state(int32 i, int32 j, int32 k, float64 rho,
    float64 rhovx, float64 rhovy, float64 rhovz, float64 en,
    int32 index_of_grid, int32 number_of_points);
```

```
FUNCTION set_grid_state(i, j, k, rho, rhovx, rhovy, rhovz, en, &
    index_of_grid, number_of_points)
    INTEGER :: i, j, k, index_of_grid, number_of_points
    DOUBLE PRECISION :: rho, rhovx, rhovy, rhovz, en
    INTEGER :: set_grid_state
END FUNCTION
```

Parameters

- **i** (*int32, IN*) –
- **j** (*int32, IN*) –
- **k** (*int32, IN*) –
- **rho** (*float64, IN*) –
- **rhovx** (*float64, IN*) –
- **rhovy** (*float64, IN*) –

- **rhovz** (*float64, IN*) –
- **en** (*float64, IN*) –
- **index_of_grid** (*int32, IN*) –
- **number_of_points** (*int32,*) –

Returns

get_grid_state

```
int32 get_grid_state(int32 i, int32 j, int32 k, int32 index_of_grid,  
float64 * rho, float64 * rhovx, float64 * rhovy, float64 * rhovz,  
float64 * en, int32 number_of_points);
```

```
FUNCTION get_grid_state(i, j, k, index_of_grid, rho, rhovx, rhovy, rhovz, &  
en, number_of_points)  
INTEGER :: i, j, k, index_of_grid, number_of_points  
DOUBLE PRECISION :: rho, rhovx, rhovy, rhovz, en  
INTEGER :: get_grid_state  
END FUNCTION
```

Parameters

- **i** (*int32, IN*) –
- **j** (*int32, IN*) –
- **k** (*int32, IN*) –
- **index_of_grid** (*int32, IN*) –
- **rho** (*float64, OUT*) –
- **rhovx** (*float64, OUT*) –
- **rhovy** (*float64, OUT*) –
- **rhovz** (*float64, OUT*) –
- **en** (*float64, OUT*) –
- **number_of_points** (*int32,*) –

Returns

1.4.5 Grid State, Extension Mechanism

Not all information of a grid point can be transferred with the `fill_grid_state` and `get_grid_state` functions. To support other properties (like pressure or MHD properties), the code can define `get_` and `set_` functions. These functions must get or set one scalar property (1 argument) or a vector property (3 arguments)

```
class amuse.community.interface.hydro.HydrodynamicsInterface
```

set_grid_density

```
int32 set_grid_density(int32 i, int32 j, int32 k, float64 rho,  
int32 index_of_grid, int32 number_of_points);
```

```
FUNCTION set_grid_density(i, j, k, rho, index_of_grid, number_of_points)  
INTEGER :: i, j, k, index_of_grid, number_of_points  
DOUBLE PRECISION :: rho  
INTEGER :: set_grid_density  
END FUNCTION
```

Parameters

- **i** (*int32, IN*) –
- **j** (*int32, IN*) –
- **k** (*int32, IN*) –
- **rho** (*float64, IN*) –
- **index_of_grid** (*int32, IN*) –
- **number_of_points** (*int32,*) –

Returns**set_grid_momentum_density**

```
int32 set_grid_momentum_density(int32 i, int32 j, int32 k,
    float64 rhovx, float64 rhovy, float64 rhovz, int32 index_of_grid,
    int32 number_of_points);
```

```
FUNCTION set_grid_momentum_density(i, j, k, rhovx, rhovy, rhovz, &
    index_of_grid, number_of_points)
INTEGER :: i, j, k, index_of_grid, number_of_points
DOUBLE PRECISION :: rhovx, rhovy, rhovz
INTEGER :: set_grid_momentum_density
END FUNCTION
```

Parameters

- **i** (*int32, IN*) –
- **j** (*int32, IN*) –
- **k** (*int32, IN*) –
- **rhovx** (*float64, IN*) –
- **rhovy** (*float64, IN*) –
- **rhovz** (*float64, IN*) –
- **index_of_grid** (*int32, IN*) –
- **number_of_points** (*int32,*) –

Returns**set_grid_energy_density**

```
int32 set_grid_energy_density(int32 i, int32 j, int32 k, float64 en,
    int32 index_of_grid, int32 number_of_points);
```

```
FUNCTION set_grid_energy_density(i, j, k, en, index_of_grid, &
    number_of_points)
INTEGER :: i, j, k, index_of_grid, number_of_points
DOUBLE PRECISION :: en
INTEGER :: set_grid_energy_density
END FUNCTION
```

Parameters

- **i** (*int32, IN*) –
- **j** (*int32, IN*) –

- **k** (*int32, IN*) –
- **en** (*float64, IN*) –
- **index_of_grid** (*int32, IN*) –
- **number_of_points** (*int32,*) –

Returns

1.4.6 Model evolution

The hydrodynamics codes evolve the properties all grid cells in time. The following functions are needed to control the evolution in the code.

class `amuse.community.interface.hydro.HydrodynamicsInterface`

initialize_grid

Perform accounting before evolving the model. This method will be called after setting the parameters and filling the grid points but just before evolving the system

```
int32 initialize_grid();
```

```
FUNCTION initialize_grid()  
  INTEGER :: initialize_grid  
END FUNCTION
```

Returns

1.4.7 Diagnostics

The state of the code can be queried, before, during and after the model calculations. The following function can be used to query the exact model time.

class `amuse.community.interface.hydro.HydrodynamicsInterface`

get_time

Returns the current model time.

```
int32 get_time(float64 * value);
```

```
FUNCTION get_time(value)  
  DOUBLE PRECISION :: value  
  INTEGER :: get_time  
END FUNCTION
```

Parameters `value` (*float64, OUT*) –

Returns

1.4.8 External fields

Some Hydrodynamics codes support external acceleration or potential fields. The following functions can be used to enter a gravitational potential field.

class `amuse.community.interface.hydro.HydrodynamicsInterface`

1.5 Radiative Transfer Interface Definition

Date	Author(s)	Version	State
02-11-2009	AvE	0.0	TBD

1.5.1 Introduction

TBD

MESSAGE PROTOCOL BETWEEN PYTHON AND COMMUNITY CODES

2.1 Introduction

The implementation of the interfaces of the community codes is based on sending messages. These messages are encoded as MPI primitive datatypes and send to each code using MPI. In this section we will describe the overall operation of the interface implementation and specify the message format.

2.2 Overall Operation

The method call interface is a request/response protocol. For every method call a message is send to the code. This message is decoded by the code into a function-id and arguments. The code will call the function with the function-id using the decoded arguments. After the function completes a result message is returned containing all the results and the function-id of the called function. This process is depicted in the following table.

Python script (client)		Code (server)
start of function call encode arguments send MPI messages	>	receive MPI messages decode messages handle (setting data, evolving the solution) encode response messages
receive MPI response messages decode response message return result to script end of function call	<	send response messages

2.3 Message format

Every message sent between the python script and a code has the same format. This format consists of a header and zero or more content arrays. The header contains the function-id, the number of calls to the same function and the number of values (arguments or results) per primitive type for one function call. Every content array contains the sent values of a primitive datatype. For example, a content array with all the integer values in the arguments of the function.

2.3.1 Message header

The header is sent with a MPI broadcast message. The header consists of an array of $n + 2$ integers. 1 integer to specify the function, 1 integer to specify the number of calls and n integers to specify the number of arguments of each type. Version 0.2 of the interface contains support for 4 types (float64, int32, float32 and string) The header is a 5 integer long array, the specification of each integer is given in the following table:

Message header

Position	Description
0	function-id
1	number of calls
2	number of arguments/results of type float64
3	number of arguments/results of type int32
4	number of arguments/results of type float32
5	number of arguments/results of type string

2.3.2 Content array

The arguments are sent with a MPI broadcast message, the results are sent with a MPI send message. The arguments or results are sent when 1 or more values are needed for a function. If no values are needed for a type, no message is sent for that type.

Encoding of the content arrays

To sent the arguments or results values of a function, the values must be encoded in arrays (each of a different primitive type).

The arguments of a function are normally not sorted by type. The first argument may be an integer, the second a double and the last an integer. The message format does specify a fixed sorting, all float64 values are sent first, then the integers, then the other types. To sent the arguments or result, the values are encoded following a fixed scheme.

The arguments are encoded by extracting all values of a primitive type in the order they occur in the function definition. This is done for every type. For example when the first argument of a function is an integer, the second a double and the last an integer, two content arrays will be sent. One for the two integers and one for the single double. The integer array has at the first position the first argument and at the second position the last argument. The double array has at the first position the second double.

For this function:

```
int example_function(int id, double x, int type)
```

Two content arrays are sent:

```
int[id, type]
    (the first argument and the last argument
     to the method are integers)

double[x]
    (the second, argument to the method is a double)
```

The arguments are encoded in order, going from left to right in C or fortran function definition. A content array is as long as the number of arguments or results of a primitive type, the specification of member in a content array is given in the following table:

Content Array

Position	Description
0	first argument of type X
1	second argument of type X
...	
n	last argument of type X

Multiple calls to the same function

The MPI messaging has a significant overhead. To reduce this overhead the arguments and results of a number of calls to the same method can be encoded in one set of messages (header and content-arrays).

Creating arrays of values:

```
x = [i for i in range(1000)] y = [i * 2 for i in range(1000)] z = [i * 3 for i in range(1000)]
```

Calling the same function multiple times with different values for the arguments:

```
for i in range(100):
    instance.add_position(x[i], y[i], z[i])
```

Can be converted to calling the function once with an array of arguments:

```
instance.add_position(x, y, z)
```

The encoding of the arguments for the call with arrays follows the same strategy as the call with single value. The values of the first argument are encoded first, the values of the second found argument of a type are encoded second etc.

Content Array format, when message is encoded for multiple calls to the same function

Position	Description
0	first value in the array of the first argument of type X
1	second value in the array of the first argument of type X
...	
m - 1	last value in the array of the first argument of type X
m	first value in the array of the second argument of type X
m + 1	second value in the array of the second argument of type X
...	
n * m	last value in the array of the last argument of type X

To get the value of the Mth value of the Nth argument of a type (starting to count at zero, n = 0 is the first argument, m = 0 is the first value of the argument):

```
value = array[ n * len + m ]
```

This encoding degrades into the case for the single call when len = 1 (m = 0, as the array contains only one value):

```
value = array[ n ]
```

2.4 Examples

TBD

SUPPORT CODE FOR AMUSE FRAMEWORK

class `amuse.support.core.late` (*initializer*)

An attribute that is set at first access.

The value of the attribute will be determined from the *initializer* method. The name of the attribute is the same as the name of the *initializer* method.

A late attribute is comparable with attributes set in the `__init__` method. Except the value of the late attribute is determined when first accessed and not when the class is instantiated.

Typical use to define a managed attribute `x`:

```
>>> class C(object):
...     @late
...     def x(self):
...         return "i'm late!"
...
>>> c = C()
>>> print c.x
i'm late!
>>> c.x = "overridden"
>>> print c.x
overridden
```

Parameters `initializer` – function to determine the initial value of the property

Returns a descriptor to determine and set the value on first access

class `amuse.support.core.print_out`

Efficient way to construct large strings.

Strings are build up out of parts. Objects of this class store these parts while building the string. Only on request the parts are concatenated into a large string.

Strings and numbers can be added to the `print_out`. For other objects `str(object)` is called before adding it to the `print_out`.

```
>>> p = print_out()
>>> p + "number of counts : " + 10
<amuse.support.core.print_out object at 0x...>
>>> print p.string
number of counts : 10
```

All methods return the `print_out` instance, so that calls can be chained.

`__add__` (*x*)

Add a new part to the `print_out`.

dedent ()

Decrease the indent. The next line will start dedented.

```
>>> p = print_out()
>>> p + "01"
<amuse.support.core.print_out object at 0x...>
>>> p.indent().lf() + "2"
<amuse.support.core.print_out object at 0x...>
>>> p.dedent().lf() + "01"
<amuse.support.core.print_out object at 0x...>
>>> print p.string
01
    2
01
```

indent ()

Increase the indent. The next and following lines will start indented.

```
>>> p = print_out()
>>> p + "01"
<amuse.support.core.print_out object at 0x...>
>>> p.indent().lf() + "2"
<amuse.support.core.print_out object at 0x...>
>>> p.lf() + "3"
<amuse.support.core.print_out object at 0x...>
>>> print p.string
01
    2
    3
```

indent_characters ()

The indent characters, by default 2 spaces.

Override this method to change the indent characters.

lf ()

Start a new-line

lf_noindent ()

Start a new-line

n ()

Start a new-line, if the current line is not-empty.

```
>>> p = print_out()
>>> for i in range(3):
...     p.n() + i
...
<amuse.support.core.print_out object at 0x...>
<amuse.support.core.print_out object at 0x...>
<amuse.support.core.print_out object at 0x...>
>>> print p.string
0
1
2
```

string

String version of the print_out.

class `amuse.support.core.OrderedDictionary`

A dictionary that keeps the keys in the dictionary in order.

Ordered dictionaries are just like regular dictionaries but they remember the order that items were inserted. When iterating over an ordered dictionary, the values are returned in the order their keys were first added.

```
>>> d = OrderedDict()
>>> d["first"] = 0
>>> d["second"] = 1
>>> d["third"] = 2
>>> [x for x in d]
[0, 1, 2]
```


SUPPORT CODE FOR THE COMMUNITY CODE INTERFACES

class `amuse.rfi.core.CodeFunction` (*interface, owner, specification*)

Implementation of the runtime call to the remote process.

Performs the encoding of python arguments into lists of values, sends a message over an MPI channel and waits for a result message, decodes this message and returns.

class `amuse.rfi.core.CodeInterface` (*name_of_the_worker='worker_code', **options*)

Abstract base class for all interfaces to legacy codes.

When a subclass is instantiated, a number of subprocesses will be started. These subprocesses are called workers as they implement the interface and do the actual work of the instantiated object.

get_data_directory ()

Returns the root name of the directory for the application data files

get_output_directory ()

Returns the root name of the directory to use by the application to store it's output / temporary files in.

redirection

Redirect the output of the code to null, standard streams or file

class `amuse.rfi.core.LegacyFunctionSpecification`

Specification of a legacy function. Describes the name, result type and parameters of a legacy function.

The legacy functions are implemented by legacy codes. The implementation of legacy functions is in C/C++ or Fortran. To interact with these functions a specification of the legacy function is needed. This specification is used to determine how to encode and decode the parameters and results of the function. Objects of this class describe the specification of one function.

```
>>> specification = LegacyFunctionSpecification()
>>> specification.name = "test"
>>> specification.addParameter("one", dtype="int32", direction = specification.IN)
>>> specification.addParameter("two", dtype="float64", direction = specification.OUT)
>>> specification.result_type = "int32"
>>> print specification
function: int test(int one)
output: double two, int __result
```

IN

Used to specify that a parameter is used as an input parameter, passed by value

INOUT

Used to specify that a parameter is used as an input and an output parameter, passed by reference

LENGTH

Used to specify that a parameter is used as the length parameter for the other parameters

OUT

Used to specify that a parameter is used as an output parameter, passed by reference

addParameter (*name*, *dtype='i'*, *direction=<object object at 0x29112b0>*, *description=''*, *default=None*)

Extend the specification with a new parameter.

The sequence of calls to `addParameter` is important. The first call will be interpreted as the first argument, the second call as the second argument etc.

Parameters

- **name** – Name of the parameter, used in documentation and function generation
- **dtype** – Datatype specification string
- **direction** – Direction of the argument, can be IN, OUT or INOUT
- **description** – Description of the argument, for documenting purposes
- **default** – An optional default value for the parameter

class `amuse.rfi.core.PythonCodeInterface` (*implementation_factory=None*,
name_of_the_worker=None, ***options*)

Base class for codes having a python implementation

Parameters **implementation_factory** – Class of the python implementation

`amuse.rfi.core.legacy_function`

Decorator for legacy functions.

The decorated function cannot have any arguments. This means the decorated function must not have a `self` argument.

The decorated function must return a `LegacyFunctionSpecification`.

```
>>> class LegacyExample(object):
...     @legacy_function
...     def evolve():
...         specification = LegacyFunctionSpecification()
...         return specification
...
>>> x = LegacyExample()
>>> x.evolve.specification
<amuse.rfi.core.LegacyFunctionSpecification object at 0x...>
>>> LegacyExample.evolve
<amuse.rfi.core.legacy_function object at 0x...>
>>> x.evolve
<amuse.rfi.core.CodeFunction object at 0x...>
```

Parameters **specification_function** – The function to be decorated

`amuse.rfi.core.legacy_global`
deprecated!

`amuse.rfi.core.stop_interfaces()`
Stop the workers of all instantiated interfaces.

All instantiated interfaces will become unstable after this call!

CURRENTLY SUPPORTED COMMUNITY CODES

5.1 Introduction

Here we provide an overview of all currently supported community codes in AMUSE, along with a concise explanation of how each code works. This document serves as an initial guide in finding the code with the highest applicability to a specific astrophysical problem. The supported codes have been sorted according to their astrophysical domain:

- *Stellar Dynamics*
- *Stellar Evolution*
- *Hydrodynamics*
- *Radiative Transfer*

5.2 Stellar Dynamics

- BHtree
- hermite0
- phiGRAPE
- twobody
- smallN
- octgrav
- mercury
- huayno
- mmc
- HiGPUs

5.2.1 General

The general parameters and methods for the gravitational dynamics are described in *Stellar Dynamics Interface Definition*. Here we describe the exceptions for the specific codes under “Specifics”.

5.2.2 Code comparison table

Note: holds for AMUSE implementation.

name	approximation scheme	timestep scheme	CPU	GPU	GRAPE	language	stop-cond (1)	parallel (2)
bhtree	tree	shared/fixed	Y	N	N	C/C++	CST	N
phi-grape	direct	block/variable	Y	Y sap- poro	Y	FOR-TAN	CSPT	N
oct-grav	tree	shared	N	Y	N	C/C++	S	N
hermite	direct	shared/variable	Y	N	N	C/C++	CSOPT	Y
two-body	universal variables, Kepler eq.	none, exact	Y	N	N	Python		N
smallN	direct Hermite 4th order	individual	Y	N	N	C/C++		N
fi	tree	block/variable	Y	N	N	FOR-TRAN	S	N
mercury	MVS symplectic		Y	N	N	FOR-TRAN		N
huayno	Approx symplectic tree	individual	Y	Y(opencl)	N	C		N
gadget			Y	N	N	C/C++	S	Y
HiG-PUs	direct	block time steps	N	Y	N	C/C++		Y (on gpus cluster)

1. stopping conditions

code	name of stopping condition
C	Collision detection
E	Escaper detection
S	Number of steps detection
O	Out of box detection
P	Pair detection
T	Timeout detection

2. Parallel in the following sense: AMUSE uses MPI to communicate with the codes, but for some codes it can be used to parallelize the calculations. Some codes (GPU) are already parallel, however in this table we *do not* refer to that.

Codes designated *Y* for parallel can set the number of (parallel) workers, e.g. to set 10 workers for hermite do:

```
>>> instance = Hermite(number_of_workers=10)
```

5.2.3 BHtree

N-body integration module. An implementation of the Barnes & Hut tree code ¹ by Jun Makino [BHTree-code](#).

¹ Barnes, J. & Hut, P. 1986. *Nature* **324**, 446.

Specifics

Parameters

name	default value	unit	description
use_self_gravity	1	none	flag for usage of self gravity, 1 or 0 (true or false)
timestep	0.015625	time	time step
epsilon squared	0.125	length*length	smoothing parameter for gravity calculations >0!
ncrit_for_tree	1024	none	maximum number of p articles sharing an interaction list
opening_angle	0.75	none	opening angle, theta, for building the tree: between 0 and 1
stop- ping_conditions_number_of_steps	1	none	
stopping_conditions_timeout	4.0	seconds	
stop- ping_conditions_out_of_box_size	0.0	length	
time	0.0	time	current simulation time
dt_dia	1.0	time	time interval between diagnostics output

class `amuse.community.bhtree.interface.BHTree` (*convert_nbody=None, **options*)

parameters

`parameters.epsilon_squared`

smoothing parameter for gravity calculations (default value:0.125 length * length)

`parameters.timestep`

constant timestep for iteration (default value:0.015625 time)

`parameters.opening_angle`

opening angle, theta, for building the tree: between 0 and 1 (default value:0.75)

`parameters.use_self_gravity`

flag for usage of self gravity, 1 or 0 (true or false) (default value:1)

`parameters.ncrit_for_tree`

Ncrit, the maximum number of particles sharing an interaction list (default value:12)

`parameters.dt_dia`

time interval between diagnostics output (default value:1.0 time)

`parameters.stopping_conditions_timeout`

max wallclock time available for the evolve step (default value:4.0 s)

`parameters.stopping_conditions_number_of_steps`

max inner loop evals (default value:1.0)

`parameters.stopping_conditions_out_of_box_size`

size of cube (default value:0.0 length)

`parameters.stopping_condition_minimum_density`

minimum density of a gas particle (default value:-1.0 mass / (length**3))

`parameters.stopping_condition_maximum_density`

maximum density of a gas particle (default value:-1.0 mass / (length**3))

`parameters.stopping_condition_minimum_internal_energy`

minimum internal energy of a gas particle (default value:-1.0 length**2 / (time**2))

`parameters.stopping_condition_maximum_internal_energy`
 maximum internal energy of a gas particle (default value:-1.0 length**2 / (time**2))

example

```
>>> from amuse.community.bhtree.interface import BHTreeInterface, BHTree
>>> from amuse.units import nbody_system
>>> instance = BHTree(BHTree.NBODY)
>>> instance.parameters.epsilon_squared = 0.00001 | nbody_system.length**2
```

5.2.4 hermite0

Time-symmetric N-body integration module with shared but variable time step (the same for all particles but its size changing in time), using the Hermite integration scheme ². See also : [ACS](#)

Specifics

Parameters

name	default value	unit	description
pair factor	1.0	none	radius factor for pair detection
dt_param	0.03	none	timestep scaling factor
epsilon squared	0.0	length*length	smoothing parameter for gravity calculations
stopping_conditions_number_of_steps	1	none	
stopping_conditions_timeout	4.0	seconds	
stopping_conditions_out_of_box_size	0.0	length	
time	0.0	time	current simulation time
dt_dia	1.0	time	time interval between diagnostics output

`class amuse.community.hermite0.interface.Hermite` (*convert_nbody=None, **options*)

parameters

`parameters.epsilon_squared`
 smoothing parameter for gravity calculations (default value:0.0 length * length)

`parameters.dt_param`
 timestep scaling factor (default value:0.03)

`parameters.dt_dia`
 time interval between diagnostics output (default value:1.0 time)

`parameters.time`
 current simulation time (default value:0.0 time)

`parameters.pair_factor`
 radius factor for pair detection (default value:1.0)

`parameters.stopping_conditions_timeout`
 max wallclock time available for the evolve step (default value:4.0 s)

`parameters.stopping_conditions_number_of_steps`
 max inner loop evals (default value:1.0)

`parameters.stopping_conditions_out_of_box_size`
 size of cube (default value:0.0 length)

`parameters.stopping_condition_minimum_density`
 minimum density of a gas particle (default value:-1.0 mass / (length**3))

² Hut, P., Makino, J. & McMillan, S., 1995, *ApJL* **443**, L93.

```

parameters.stopping_condition_maximum_density
    maximum density of a gas particle (default value:-1.0 mass / (length**3))

parameters.stopping_condition_minimum_internal_energy
    minimum internal energy of a gas particle (default value:-1.0 length**2 / (time**2))

parameters.stopping_condition_maximum_internal_energy
    maximum internal energy of a gas particle (default value:-1.0 length**2 / (time**2))

```

5.2.5 phiGRAPE

phiGRAPE is a direct N-body code optimized for running on a parallel GRAPE cluster. See Harfst et al. ³ for more details. The Amusean version is capable of working on other platforms as well by using interfaces that mimic GRAPE hardware.

- **Sapporo** Sapporo is a library that mimics the behaviour of GRAPE hardware and uses the GPU to execute the force calculations ⁴.
- **Sapporo-light** This version of Sapporo is without multi-threading support and does not need C++. This makes it easier to integrate into fortran codes, but beware, it can only use one GPU device per application!
- **g6** Library which mimics the behavior of GRAPE and uses the CPU. Lowest on hw requirements.

Specifics

Hardware modes

Parameters

Amuse tries to build all implementations at compile time. In the phiGRAPE interface module the preferred mode can be selected with the mode parameter:

- **MODE_G6LIB = 'g6lib'** Just make it work, no optimizations, no special hw requirements
- **MODE_GPU = 'gpu'** Using sapporo, CUDA needed.
- **MODE_GRAPE = 'grape'** Using GRAPE hw.
- **MODE_PG = 'pg'** Phantom grape, optimized for x86_64 processors

```

>>> from amuse.community.phiGRAPE.interface import PhiGRAPEInterface, PhiGRAPE
>>> instance = PhiGRAPE(PhiGRAPE.NBODY, PhiGRAPEInterface.MODE_GPU)

```

The default is **MODE_G6LIB**.

³ Harfst, S., Gualandris, A., Merritt, D., Spurzem, R., Portegies Zwart, S., & Berczik, P. 2006, *NewAstron.* **12**, 357-377.

⁴ Gaburov, E., Harfst, S., Portegies Zwart, S. 2009, *NewAstron.* **14** 630-637.

name	default value	unit	description
initial-ize_gpu_once	0	none	set to 1 if the gpu must only be initialized once, 0 if it can be initialized for every call If you want to run multiple instances of the code on the same gpu this parameter needs to be 0 (default)
initial_timestep_parameter	0.0	none	parameter to determine the initial timestep
timestep_parameter	0.0	none	
epsilon_squared	0.0	length ³	smoothing parameter for gravity calculations >0!
stopping_conditions_number_of_steps	1	none	
stopping_conditions_timeout	4.0	sec-onds	
stopping_conditions_out_of_box_size	0.0	length	

```
class amuse.community.phigRAPE.interface.PhiGRAPE (convert_nbody=None,
mode='g6lib', use_gl=False,
**options)
```

parameters

`parameters.epsilon_squared`

smoothing parameter for gravity calculations (default value:0.0 length * length)

`parameters.timestep_parameter`

timestep parameter (default value:0.02)

`parameters.initial_timestep_parameter`

parameter to determine the initial timestep (default value:0.01)

`parameters.initialize_gpu_once`

set to 1 if the gpu must only be initialized once, 0 if it can be initialized for every call
If you want to run multiple instances of the code on the same gpu this parameter needs to be 0 (default)
(default value:0)

`parameters.stopping_conditions_timeout`

max wallclock time available for the evolve step (default value:4.0 s)

`parameters.stopping_conditions_number_of_steps`

max inner loop evals (default value:1.0)

`parameters.stopping_conditions_out_of_box_size`

size of cube (default value:0.0 length)

`parameters.stopping_condition_minimum_density`

minimum density of a gas particle (default value:-1.0 mass / (length**3))

`parameters.stopping_condition_maximum_density`

maximum density of a gas particle (default value:-1.0 mass / (length**3))

`parameters.stopping_condition_minimum_internal_energy`

minimum internal energy of a gas particle (default value:-1.0 length**2 / (time**2))

`parameters.stopping_condition_maximum_internal_energy`

maximum internal energy of a gas particle (default value:-1.0 length**2 / (time**2))

```
>>> instance.timestep_parameter = 0.1 |nbody_system.time
```

5.2.6 twobody

Semi analytical code based on Kepler ⁵. The particle set provided has length one or two. If one particle is given, the mass is assigned to a particle in the origin and the phase-coordinates are assigned to the other particle. This is usefull when $m1 \gg m2$.

Specifics

Parameters

5.2.7 smallN

Interface to the Kira Small-N Integrator and Kepler modules from Starlab. <http://www.ids.ias.edu/~starlab/>

You will need to download Starlab from the above site, make it, install it, and then set the STARLAB_INSTALL_PATH variable to be equal to the installation directory (typically something like ~/starlab/usr).

Starlab is available under the GNU General Public Licence (version 2), and is developed by:

- Piet Hut
- Steve McMillan
- Jun Makino
- Simon Portegies Zwart

Other Starlab Contributors:

- Douglas Heggie
- Kimberly Engle
- Peter Teuben

Specifics

Parameters

name	default value	unit	description
epsilon squared	0.0	length*length	smoothing parameter for gravity calculations >0!
number_of_particles	0.0	none	

Methods

5.2.8 Octgrav

Tree-code which runs on GPUs with NVIDIA CUDA architecture. ⁶

⁵ Bate, R.R, Mueller, D.D., White, J.E. "FUNDAMENTALS OF ASTRODYNAMICS" Dover 0-486-60061-0

⁶ Gaburov, E., Bedorf, J., Portegies Zwart S., 2010, "Gravitational tree-code on graphics processing units: implementations in CUDA", ICCS

Specifics

Parameters

name	default value	unit	description
opening_angle	0.8	none	opening angle for building the tree between 0 and 1
timestep	0.01	time	constant timestep for iteration
epsilon squared	0.01	length*length	smoothing parameter for gravity calculations
stop- ping_conditions_number_of_steps	1	none	
stopping_conditions_timeout	4.0	seconds	
stop- ping_conditions_out_of_box_size	0.0	length	

class `amuse.community.octgrav.interface.Octgrav` (*convert_nbody=None, **options*)

parameters

`parameters.epsilon_squared`
smoothing parameter for gravity calculations (default value:0.01 length * length)

`parameters.timestep`
constant timestep for iteration (default value:0.01 time)

`parameters.opening_angle`
opening angle for building the tree between 0 and 1 (default value:0.8)

`parameters.stopping_conditions_timeout`
max wallclock time available for the evolve step (default value:4.0 s)

`parameters.stopping_conditions_number_of_steps`
max inner loop evals (default value:1.0)

`parameters.stopping_conditions_out_of_box_size`
size of cube (default value:0.0 length)

`parameters.stopping_condition_minimum_density`
minimum density of a gas particle (default value:-1.0 mass / (length**3))

`parameters.stopping_condition_maximum_density`
maximum density of a gas particle (default value:-1.0 mass / (length**3))

`parameters.stopping_condition_minimum_internal_energy`
minimum internal energy of a gas particle (default value:-1.0 length**2 / (time**2))

`parameters.stopping_condition_maximum_internal_energy`
maximum internal energy of a gas particle (default value:-1.0 length**2 / (time**2))

Specifics

Parameters

name	default value	unit	description
opening_angle	0.8	none	opening angle for building the tree between 0 and 1
timestep	0.01	time	constant timestep for iteration
epsilon squared	0.01	length*length	smoothing parameter for gravity calculations
stop- ping_conditions_number_of_steps	1	none	
stopping_conditions_timeout	4.0	seconds	
stop- ping_conditions_out_of_box_size	0.0	length	

5.2.9 mercury

Mercury is a general-purpose N-body integration package for problems in celestial mechanics.

This package contains some subroutines taken from the Swift integration package by H.F. Levison and M.J. Duncan (1994) *Icarus*, vol 108, pp18. Routines taken from Swift have names beginning with *drift* or *orbel*.

The standard symplectic (MVS) algorithm is described in J. Widsom and M. Holman (1991) *Astronomical Journal*, vol 102, pp1528.

The hybrid symplectic algorithm is described in J.E. Chambers (1999) *Monthly Notices of the RAS*, vol 304, pp793.

Currently Mercury has an interface that differs from the other grav dyn interfaces. The class is called MercuryWayWard and will lose its predicate once the interface is standardized (work in progress). It handles two kinds of particles: *centre_particle* and *orbiters*. The *centre_particle* is restricted to contain only one particle and it should be the heaviest and much heavier than the orbiters. It is at the origin in phase space.

Apart from the usual phase space coordinate, particles have a spin in mercury and the centre particle has oblateness parameters expressed in moments 2, 4 and 6. Orbiters have density.

Furthermore, mercury does not use nbody units but instead units as listed in table.

Central particle

mass		MSun
radius		AU
oblateness	j2, j4, j6	AU ² etc.
angular momentum		MSun AU ² day ⁻¹

Orbiters

mass			MSun
density			g/cm ³
position		x, y, z	AU
velocity		vx, vy, vz	AU/day
celimit	close encounters		Hill radii, but units.none in amuse
angular momentum		Lx, Ly, Lz	MSun AU ² day ⁻¹

```
class amuse.community.mercury.interface.MercuryWayWard (convert_nbody=None,
**options)
```

parameters

`parameters.timestep`
current simulation time (default value:0.0 s)

`parameters.stopping_conditions_timeout`
max wallclock time available for the evolve step (default value:4.0 s)

`parameters.stopping_conditions_number_of_steps`
max inner loop evals (default value:1.0)

`parameters.stopping_conditions_out_of_box_size`
size of cube (default value:0.0 length)

`parameters.stopping_condition_minimum_density`
minimum density of a gas particle (default value:-1.0 mass / (length**3))

`parameters.stopping_condition_maximum_density`
maximum density of a gas particle (default value:-1.0 mass / (length**3))

`parameters.stopping_condition_minimum_internal_energy`
minimum internal energy of a gas particle (default value:-1.0 length**2 / (time**2))

`parameters.stopping_condition_maximum_internal_energy`
maximum internal energy of a gas particle (default value:-1.0 length**2 / (time**2))

5.2.10 Huayno

Hierarchically split-Up Approximately sYmplectic N-body sOlver (HUAYNO)

Inti Pelupessy - january 2011

short description

HUAYNO is a code to solve the astrophysical N-body problem. It uses recursive Hamiltonian splitting to generate multiple-timestep integrators which conserve momentum to machine precision. A number of different integrators are available. The code has been developed within the AMUSE environment. It can make use of GPUs - for this an OpenCL version can be compiled.

Use

Use of the code is the same as any gravity code within AMUSE. There are three parameters for the code: Smoothing length (squared) `eps2`, timestep parameter `eta` and a parameter to select the integrator:

`timestep_parameter(eta)`: recommended values `eta=0.0001 - 0.05 epsilon_squared (eps2)`: `eps2` can be zero or non-zero
`inttype_parameter(inttype)`: possible values for `inttype` are described below.

Miscellaneous:

- The code assumes $G=1$,
- Collisions are not implemented (needs rewrite),
- `workernp` option can be used for OpenMP parallelization,
- The floating point precision of the calculations can be changed by setting the `FLOAT` and `DOUBLE` definitions in `evolve.h`. `FLOAT` sets the precision of the calculations, `DOUBLE` the precision of the position and velocity reductions. They can be set to e.g. `float`, `double`, `long double` or `__float128`. It is advantageous to choose set `DOUBLE` at a higher precision than `FLOAT`. recommended is the combination: `double/ long double`
- the AMUSE interface uses double precision

- It is unlikely the integer types in evolve.h would need to be changed (INT should be able to hold particle number, LONG should be able to hold interaction counts)

OpenCL operation:

By compiling worker_cl it is possible to offload the force and timestep loops to the GPU. The implementation is based on the STDCL library (www.browndeertechnology.com) so this library should be compiled first. In the Makefile the corresponding directories should point to the installation directory of STDCL. The define in evolve_cl.h should be set appropriately for the OpenCL configuration: CLCONTEXT: stdgpu or stdcpu NTHREAD: 64 for GPU, 2 for CPU (actually 64 will also work for CPU just fine) BLOCKSIZE: number of particles stored in local memory (64 good starting guess) evolve_kern.cl contains the OpenCL kernels. Precision of the calculation is controlled by FLOAT(4) defines in evolve_kern.cl and CLFLOAT(4) in evolve.h. They should agree with each other (i.e. float and cl_float or double and cl_double)

5.2.11 Monte Carlo

Giersz, M. 1998, MNRAS, 298, 1239 Giersz, M. 2001, MNRAS, 324, 218 Giersz, M. 2006, MNRAS, 371, 484

Specifics

parameters

name	description
irun	initial sequence of random numbers
nt	total number of objects (stars and binaries) at T=0 ns - number of single stars, nb - number of binaries, nt ns+nb, nss -
istart	1 - initial model, .ne.1 - restart
ncor	number of stars to calculate the central parameters
nmin	minimum number of stars to calculate the central parameters
nz0	number of stars in each zone at T=0
nzone	minimum number of zones in the core
nminzo	minimum number of stars in a zone
ntwo	maximum index of 2
imodel	initial model: 1- uniform & isotropic, 2- Plummer, 3- King, 4 - M67
iprint	0- full diagnostic information, 1- diagnostic info. suppressed
ib3f	1 - Spitzer's, 2 - Heggie's formula for three-body binary interaction with field stars, 3 - use Pmax for interaction * prob
iexch	0 - no exchange in any interactions, 1 - exchange only in binary field star interactions, 2 - exchange in all interactions (b
tcrit	termination time in units of the crossing time
tcomp	maximum computing time in hours
qe	energy tolerance
alpha	power-law index for initial mass function for masses smaller than breake mass: -1 - equal mass case
alphah	power-law index for initial mass function for masses greater than breake mass. If alpha=alphah the IMF does not have
brakem	the mass in which the IMF is broken. If brakem is smaller * than the minimum mass (bodyn) than the break mass is as
bodyl	maximum particle mass before scaling (solar mass)
bodyn	minimum particle mass before scaling (solar mass)
fracb	primordial binary fraction by number. nb = fracb*nt, * ns = (1 - fracb)*nt, nss = (1 + fracb)*nt * fracb > 0 - primordial
amin	minimum semi-major axis of binaries (in solar units) * = 0 then amin = 2*(R1+R2), > 0 then amin = amin
amax	maximum semi-major axis of binaries (in solar units)
qvir	virial ratio (qvir = 0.5 for equilibrium)
rbar	tidal radius in pc, halfmass radius in pc for isolated * cluster. No scaling - rbar = 1
zmbar	total mass of the cluster in solar mass, * no scaling zmbar = 1
w0	king model parameter
bmin	minimum value of sin(beta^2/2)
bmax	maximum value of sin(beta^2/2)
tau0	time step for a complete cluster model
gamma	parameter in the Coulomb logarithm (standard value = 0.11)

Table 5.1 – continued from p

xtid	coefficient in the front of cluster tidal energy: $* -xtid*smt/rtid$
rplum	for M67 $rtid = rplum*rsplum$ ($rsplum$ - scale radius for $*$ plummer model)
dttp	time step (Myr) for profile output
dtte	time step (Myr) for mloss call for all objects
dtte0	time step (Myr) for mloss call for all objects for $tphys$ $*$ less then $tcrevo$. For $tphys$ greater then $tcrevo$ time step $*$ is eq
tcrevo	critical time for which time step for mloss call changes from $*$ dtte0 to dtte
xtau	call mloss for a particular object when $*$ $(uptime(im1) - olduptime(im1))/tau/tscale < xtau$
ytau	multiplication of $tau0$ ($tau = ytau*tau0$) after time $*$ greater than $tcrevo$
ybmin	multiplication of $bmin0$ ($bmin = ybmin*bmin0$) after time $*$ greater than $tcrevo$
zini	initial metalicity (solar $z = 0.02$, globular clusters $* M4 - z = 0.002$, NGC6397 - $z = 0.0002$)
ikroupa	0 - the initial binary parameters are picked up $*$ according Kroupa's eigenevolution and feeding algorithm $*$ (Kroupa 19
iflags	0 - no SN natal kiks for NS formation, 1 - SN natal kicks only for single NS formation, 2 - SN natal kick for single NS
iflagbh	0 - no SN natal kiks for BH formation, 1 - SN natal kicks $*$ only for single BH formation, 2 - SN natal kick for single $*$
nitesc	0 - no iteration of the tidal radius and induced mass loss $*$ due to stellar evolution, 1 - iteration of the tidal radius $*$ and

```
class amuse.community.mmc.interface.mmc
```

5.2.12 HiGPUs

HiGPUs is a parallel direct N-body code based on a 6th order Hermite integrator. The code has been developed by Capuzzo-Dolcetta, Punzo and Spera (Dep. of Physics, Sapienza, Univ. di Roma; see astrowww.phys.uniroma1.it/dolcetta/hpc/hpc.html) and uses, at the same time, MPI, OpenMP and CUDA libraries to fully exploit all the capabilities offered by hybrid supercomputing platforms. Moreover, it is implemented using block time steps (individual time stepping) such to be able to deal with stiff problems like highly collisional gravitational N-body problems.

Specifics

Parameters

name	default value	unit	description
eta_6	0.4	none	eta parameter for determining stars time steps
eta_4	0.01	none	eta parameter for initializing blocks
softening	0.001	length	smoothing parameter for gravity calculations
r_core_plummer	0.0	length	core radius for analytical plummer potential
mass_plummer	0.0	mass	total mass for analytical plummer potential
start_time	0.0	time	initial simulation time
min_step	-30.0	none	exponent which defines the minimum time step allowed for stars (2^{exponent})
max_step	-3.0	none	exponent which defines the maximum time step allowed for stars (2^{exponent})
n_Print	1000000	none	maximum number of snapshots
dt_Print	1.0	time	time interval between diagnostics output
n_gpu	2	none	number of GPUs per node
gpu_name	GeForce GTX	none	GPUs to use
Threads	480	none	number of gpus threads per block
out-put_path_name	../test_results/	none	path where HiGPUs output will be stored

For more information about parameters check the readme file in the docs folder. These are the maximum performance (in Gflops) reached using different single GPUs installed, one at a time, on a workstation equipped with 2 CPUs Intel Xeon X5650, 12 GB of ECC RAM memory 1333 MHz, Ubuntu Lucid 10.04 x86_64, motherboard Supermicro X8DTG-QF:

- TESLA C1060 : 107
- TESLA C2050 : 395
- TESLA M2070 : 391
- GeForce GTX 480 : 265
- GeForce GTX 580 : 311

class `amuse.community.higpus.interface.HiGPUs` (*convert_nbody=None*, *word_arguments*) ****key-**

parameters

`parameters.eta_4`
timestep parameter (default value:0.01 none)

`parameters.eta_6`
timestep parameter (default value:0.4 none)

`parameters.start_time`
start time (default value:0.0 time)

`parameters.r_core_plummer`
radius of Plummer potential (default value:0.0 length)

`parameters.mass_plummer`
mass of Plummer potential (default value:0.0 mass)

`parameters.softening`
softening (default value:0.001 length)

`parameters.Threads`
Threads per block (default value:128 none)

`parameters.n_Print`
start number to print file (default value:1000000 none)

`parameters.dt_Print`
time for snapshot (default value:1000000.0 time)

`parameters.max_step`
power of 2 for maximum time step (default value:-3.0 time)

`parameters.min_step`
power of 2 for minnum time step (default value:-30.0 time)

`parameters.gpu_name`
gpu name (default value:GeForce GTX 480 none)

`parameters.output_path_name`
output path name (default value:../test_results/ none)

`parameters.n_gpu`
number of gpus per node (default value:2 none)

5.3 Stellar Evolution

- [sse](#)
- [bse](#)
- [evtwin](#)
- [mesa](#)

5.3.1 sse

Stellar evolution is performed by the **rapid** single-star evolution (SSE) algorithm. This is a package of **analytical formulae** fitted to the detailed models of Pols et al. (1998) that covers all phases of evolution from the zero-age main-sequence up to and including remnant phases. It is valid for masses in the range 0.1-100 Msun and metallicity can be varied. The SSE package contains a prescription for mass loss by stellar winds. It also follows the evolution of rotational angular momentum for the star. Full details can be found in the SSE paper:

- “Comprehensive analytic formulae for stellar evolution as a function of mass and metallicity”
Hurley J.R., Pols O.R., Tout C.A., 2000, MNRAS, 315, 543

	min	max	unit
Mass	0.1	100	Msun
Metallicity	0.0001	0.03	fraction (0.02 is solar)

```
class amuse.community.sse.interface.SSE(**options)
```

parameters

```
parameters.metallicity
```

Metallicity of all stars (default value:0.02)

```
parameters.reimers_mass_loss_coefficient
```

Reimers mass-loss coefficient ($\eta \times 4 \times 10^{-13}$; 0.5 normally) (default value:0.5)

```
parameters.binary_enhanced_mass_loss_parameter
```

The binary enhanced mass loss parameter (inactive for single). (default value:0.0)

```
parameters.helium_star_mass_loss_factor
```

Helium star mass loss factor (default value:1.0)

```
parameters.SN_kick_speed_dispersion
```

The dispersion in the Maxwellian for the SN kick speed (190 km/s). (default value:190.0 km / s)

```
parameters.white_dwarf_IFMR_flag
```

ifflag > 0 uses white dwarf IFMR (initial-final mass relation) of HPE, 1995, MNRAS, 272, 800 (0). (default value:0)

```
parameters.white_dwarf_cooling_flag
```

wdfalg > 0 uses modified-Mestel cooling for WDs (0). (default value:1)

```
parameters.black_hole_kick_flag
```

bhflag > 0 allows velocity kick at BH formation (0). (default value:0)

```
parameters.neutron_star_mass_flag
```

nsflag > 0 takes NS/BH mass from Belczynski et al. 2002, ApJ, 572, 407 (1). (default value:1)

```
parameters.maximum_neutron_star_mass
```

The maximum neutron star mass (1.8, nsflag=0; 3.0, nsflag=1). (default value:3.0 MSun)

```
parameters.fractional_time_step_1
```

The timesteps chosen in each evolution phase as decimal fractions of the time taken in that phase: MS (0.05) (default value:0.05)

```
parameters.fractional_time_step_2
```

The timesteps chosen in each evolution phase as decimal fractions of the time taken in that phase: GB, CHeB, AGB, HeGB (0.01) (default value:0.01)

```
parameters.fractional_time_step_3
```

The timesteps chosen in each evolution phase as decimal fractions of the time taken in that phase: HG, HeMS (0.02) (default value:0.02)

5.3.2 bse

Binary evolution is performed by the **rapid** binary-star evolution (BSE) algorithm. Circularization of eccentric orbits and synchronization of stellar rotation with the orbital motion owing to tidal interaction is modelled in detail. Angular momentum loss mechanisms, such as gravitational radiation and magnetic braking, are also modelled. Wind accretion, where the secondary may accrete some of the material lost from the primary in a wind, is allowed with the necessary adjustments made to the orbital parameters in the event of any mass variations. Mass transfer also occurs if either star fills its Roche lobe and may proceed on a nuclear, thermal or dynamical time-scale. In the latter regime, the radius of the primary increases in response to mass-loss at a faster rate than the Roche-lobe of the star. Stars with deep surface convection zones and degenerate stars are unstable to such dynamical time-scale mass loss unless the mass ratio of the system is less than some critical value. The outcome is a common-envelope event if the primary is a giant star. This results in merging or formation of a close binary, or a direct merging if the primary is a white dwarf or low-mass main-sequence star. On the other hand, mass transfer on a nuclear or thermal time-scale is assumed to be a steady process. Prescriptions to determine the type and rate of mass transfer, the response of the secondary to accretion and the outcome of any merger events are in place in BSE and the details can be found in the BSE paper:

- “**Evolution of binary stars and the effect of tides on binary populations**” Hurley J.R., Tout C.A., & Pols O.R., 2002, MNRAS, 329, 897
- “**Comprehensive analytic formulae for stellar evolution as a function of mass and metallicity**” Hurley J.R., Pols O.R., Tout C.A., 2000, MNRAS, 315, 543

	min	max	unit
Mass	0.1	100	Msun
Metallicity	0.0001	0.03	fraction (0.02 is solar)
Period	all	all	
Eccentricity	0.0	1.0	

`class amuse.community.bse.interface.BSE (**options)`

parameters

`parameters.metallicity`

Metallicity of all stars (default value:0.02)

`parameters.reimers_mass_loss_coefficient`

Reimers mass-loss coefficient ($\eta_{\text{r}} \times 4 \times 10^{-13}$; 0.5 normally) (default value:0.5)

`parameters.binary_enhanced_mass_loss_parameter`

The binary enhanced mass loss parameter (inactive for single). (default value:0.0)

`parameters.helium_star_mass_loss_factor`

Helium star mass loss factor (default value:1.0)

`parameters.common_envelope_efficiency`

The common-envelope efficiency parameter (default value:1.0)

`parameters.common_envelope_binding_energy_factor`

The binding energy factor for common envelope evolution (default value:0.5)

`parameters.common_envelope_model_flag`

`ceflag > 0` activates spin-energy correction in common-envelope. `ceflag = 3` activates de Kool common-envelope model (0). (default value:0)

`parameters.tidal_circularisation_flag`

`tflag > 0` activates tidal circularisation (1). (default value:1)

`parameters.white_dwarf_IFMR_flag`

`ifflag > 0` uses white dwarf IFMR (initial-final mass relation) of HPE, 1995, MNRAS, 272, 800 (0). (default value:0)

`parameters.white_dwarf_cooling_flag`
wdflag > 0 uses modified-Mestel cooling for WDs (0). (default value:1)

`parameters.black_hole_kick_flag`
bhflag > 0 allows velocity kick at BH formation (0). (default value:0)

`parameters.neutron_star_mass_flag`
nsflag > 0 takes NS/BH mass from Belczynski et al. 2002, ApJ, 572, 407 (1). (default value:1)

`parameters.maximum_neutron_star_mass`
The maximum neutron star mass (1.8, nsflag=0; 3.0, nsflag=1). (default value:3.0 MSun)

`parameters.SN_kick_random_seed`
The random number seed used in the kick routine. (default value:29769)

`parameters.fractional_time_step_1`
The timesteps chosen in each evolution phase as decimal fractions of the time taken in that phase: MS (0.05) (default value:0.05)

`parameters.fractional_time_step_2`
The timesteps chosen in each evolution phase as decimal fractions of the time taken in that phase: GB, CHeB, AGB, HeGB (0.01) (default value:0.01)

`parameters.fractional_time_step_3`
The timesteps chosen in each evolution phase as decimal fractions of the time taken in that phase: HG, HeMS (0.02) (default value:0.02)

`parameters.SN_kick_speed_dispersion`
The dispersion in the Maxwellian for the SN kick speed (190 km/s). (default value:190.0 km / s)

`parameters.wind_velocity_factor`
The wind velocity factor: proportional to vwind**2 (1/8). (default value:0.125)

`parameters.wind_accretion_efficiency`
The wind accretion efficiency factor (1.0). (default value:1.0)

`parameters.wind_accretion_factor`
The Bondi-Hoyle wind accretion factor (3/2). (default value:1.5)

`parameters.nova_retained_accreted_matter_fraction`
The fraction of accreted matter retained in nova eruption (0.001). (default value:0.001)

`parameters.Eddington_mass_transfer_limit_factor`
The Eddington limit factor for mass transfer (1.0). (default value:1.0)

`parameters.Roche_angular_momentum_factor`
The angular momentum factor for mass lost during Roche (-1.0). (default value:-1.0)

5.3.3 evtwin

Evtwin is based on Peter Eggleton’s stellar evolution code, and actually solves the differential equations that apply to the interior of a star. Therefore it is more accurate, but also much slower than the analytic fits-based sse algorithm explained above. Binaries are not yet supported in the AMUSE interface to evtwin, neither is the work-around for the helium flash. Currently only solar metallicity.

Relevant papers:

- “The evolution of low mass stars” Eggleton, P.P. 1971, MNRAS, 151, 351
- “Composition changes during stellar evolution” Eggleton, P.P. 1972, MNRAS, 156, 361
- “A numerical treatment of double shell source stars” Eggleton, P.P. 1973, MNRAS, 163, 279
- “An Approximate Equation of State for Stellar Material” Eggleton, P.P., Faulkner, J., & Flannery, B.P. 1973, A&A, 23, 325

- “A Possible Criterion for Envelope Ejection in Asymptotic Giant Branch or First Giant Branch Stars” Han, Z., Podsiadlowski, P., & Eggleton, P.P. 1994, MNRAS, 270, 121
- “Approximate input physics for stellar modelling” Pols, O.R., Tout, C.A., Eggleton, P.P., & Han, Z. 1995, MNRAS, 274, 964
- “The Braking of Wind” Eggleton, P.P. 2001, Evolution of Binary and Multiple Star Systems, 229, 157
- “A Complete Survey of Case A Binary Evolution with Comparison to Observed Algol-type Systems” Nelson, C.A., & Eggleton, P.P. 2001, ApJ, 552, 664
- “The Evolution of Cool Algols” Eggleton, P.P., & Kiseleva-Eggleton, L. 2002, ApJ, 575, 461
- For thermohaline mixing: Stancliffe, Glebbeek, Izzard & Pols, 2007 A&A
- For the OPAL 1996 opacity tables: Eldridge & Tout, 2004 MNRAS 348
- For enhancements to the solver: Glebbeek, Pols & Hurley, 2008 A&A

```
class amuse.community.evtwin.interface.Evtwin(**options)
```

```
    parameters
```

5.3.4 mesa

The software project MESA (Modules for Experiments in Stellar Astrophysics, <http://mesa.sourceforge.net/>), aims to provide state-of-the-art, robust, and efficient open source modules, usable singly or in combination for a wide range of applications in stellar astrophysics. Since the package is rather big (about 800 MB download, >2 GB built), this community code is optional and does not install automatically. Set the environment variable `DO_INSTALL_MESA` and run `make` to download and install it. The AMUSE interface to MESA can create and evolve stars using the MESA/STAR module. If you order a metallicity you haven't used before, starting models will be computed automatically and saved in the `mesa/src/data/star_data/starting_models` directory (please be patient...). All metallicities are supported, even the interesting case of $Z=0$. The supported stellar mass range is from about 0.1 to 100 Msun.

References:

- Paxton, Bildsten, Dotter, Herwig, Lesaffre & Timmes 2010, ApJS submitted, arXiv:1009.1622
- <http://mesa.sourceforge.net/>

```
class amuse.community.mesa.interface.MESA(**options)
```

```
    parameters
```

5.4 Hydrodynamics

- `athena` (grid code)
- `capreole` (grid code)
- `fi` (N-body/SPH code)
- `gadget2` (N-body/SPH code)

5.4.1 athena

Athena is a grid-based code for astrophysical hydrodynamics. Athena can solve magnetohydrodynamics (MHD) as well, but this is currently not supported from AMUSE. It was developed primarily for studies of the interstellar medium, star formation, and accretion flows.

The current version (Athena v4.0) implements algorithms for the following physics:

- compressible hydrodynamics and MHD in 1D, 2D, and 3D,
- ideal gas equation of state with arbitrary γ (including $\gamma = 1$, an isothermal EOS),
- an arbitrary number of passive scalars advected with the flow,
- self-gravity, and/or a static gravitational potential,
- Ohmic resistivity, ambipolar diffusion, and the Hall effect,
- both Navier-Stokes and anisotropic (Braginskii) viscosity,
- both isotropic and anisotropic thermal conduction,
- optically-thin radiative cooling.

In addition, Athena allows for the following grid and parallelization options:

- Cartesian or cylindrical coordinates,
- static (fixed) mesh refinement,
- shearing-box source terms, and an orbital advection algorithm for MHD,
- parallelization using domain decomposition and MPI.

A variety of choices are also available for the numerical algorithms, such as different Riemann solvers and spatial reconstruction methods.

The relevant references are:

- Gardiner & Stone 2005, JCP, 205, 509 (2D JCP Method)
- Gardiner & Stone 2007, JCP, 227, 4123 (3D JCP Method)
- Stone et al. 2008, ApJS, 178, 137 (Method)
- Stone & Gardiner 2009, NewA, 14, 139 (van Leer Integrator)
- Skinner & Ostriker 2010, ApJ, 188, 290 (Cylindrical Integrator)
- Stone & Gardiner 2010, ApJS, 189, 142 (Shearing Box Method)

```
class amuse.community.athena.interface.Athena (unit_converter=None, **options)
```

parameters

5.4.2 capreole

Capreole is a grid-based astrophysical hydrodynamics code developed by Garrelt Mellema. It works in one, two dimensions, and three spatial dimensions and is programmed in Fortran 90. It is parallelized with MPI. For the hydrodynamics it relies on the Roe-Eulderink-Mellema (REM) solver, which is an approximate Riemann solver for arbitrary metrics. It can solve different hydrodynamics problems. Capreole has run on single processors, but also on massively parallel systems (e.g. 512 processors on a BlueGene/L).

The reference for Capreole (original version):

- Mellema, Eulderink & Icke 1991, A&A 252, 718

```
class amuse.community.capreole.interface.Capreole (unit_converter=None, **options)
```

parameters

```
parameters.nx  
    number of cells in the x direction (default value:10)
```

```
parameters.ny  
    number of cells in the y direction (default value:10)
```

```

parameters.nz
    number of cells in the z direction (default value:10)

parameters.length_x
    length of model in the x direction (default value:10 length)

parameters.length_y
    length of model in the x direction (default value:10 length)

parameters.length_z
    length of model in the z direction (default value:10 length)

parameters.mesh_size
    number of cells in the x, y and z directions (default value:[10, 10, 10])

parameters.mesh_length
    length of the model in the x, y and z directions (default value:[10, 10, 10] length)

parameters.xbound1
    boundary conditions on first (inner, left) X boundary (default value:reflective)

parameters.xbound2
    boundary conditions on second (outer, right) X boundary (default value:reflective)

parameters.ybound1
    boundary conditions on first (inner, front) Y boundary (default value:reflective)

parameters.ybound2
    boundary conditions on second (outer, back) Y boundary (default value:reflective)

parameters.zbound1
    boundary conditions on first (inner, bottom) Z boundary (default value:reflective)

parameters.zbound2
    boundary conditions on second (outer, top) Z boundary (default value:reflective)

parameters.x_boundary_conditions
    boundary conditions for the X directorion (default value:['reflective', 'reflective'])

parameters.y_boundary_conditions
    boundary conditions for the Y directorion (default value:['reflective', 'reflective'])

parameters.z_boundary_conditions
    boundary conditions for the Z directorion (default value:['reflective', 'reflective'])

```

5.4.3 fi

FI is a parallel TreeSPH code for galaxy simulations. Extensively rewritten, extended and parallelized, it is a development from code from Jeroen Gerritsen and Roelof Bottema, which itself goes back to Treesph.

The relevant references are:

- Hernquist & Katz 1989, ApJS 70, 419
- Gerritsen & Icke 1997, A&A 325, 972
- Pelupessy, van der Werf & Icke 2004, A&A 422, 55
- Pelupessy, PhD thesis 2005, Leiden Observatory

```

class amuse.community.fi.interface.Fi (convert_nbody=None, mode='normal',
                                         use_gl=False, **options)

```

parameters

```

parameters.epsilon_squared
    smoothing parameter for gravity calculations (default value:0.0 length * length)

```

`parameters.timestep`
timestep for system (default value:1.0 time)

`parameters.radiation_flag`
Radiation flag. True means: radiation (i.e. radiative cooling/heating) is included. False means: no radiation, and implies no star formation. (default value:False)

`parameters.star_formation_flag`
Star-formation flag. True means: star formation is included. False means: no star formation included. (default value:False)

`parameters.use_hydro_flag`
Hydrodynamics flag. True means: SPH hydro included, False means: gravity only. (default value:True)

`parameters.square_root_timestep_flag`
Square-root-timestep flag. True means: use $\sqrt{\text{eps}/\text{acc}}$ timestep criterion. (default value:False)

`parameters.acc_timestep_flag`
Acceleration-timestep flag. True means: use $v_{\text{ref}}/\text{acc}$ timestep criterion. (default value:True)

`parameters.freeform_timestep_flag`
Freeform-timestep flag. True means: use freeform $(v/\text{freev})^{**\text{freevexp}} * (a/\text{freea})^{**\text{freeaexp}}$ timestep criterion. (default value:False)

`parameters.quadrupole_moments_flag`
Quadrupole-moments flag. True means: calculate and use quadrupole cell moments. (default value:False)

`parameters.direct_sum_flag`
Direct-summation flag. True means: direct N^{**2} gravity summation. (default value:False)

`parameters.self_gravity_flag`
Self-gravity flag. False means: self-gravity is not used, only external potentials. (default value:True)

`parameters.fixed_halo_flag`
Fixed-halo flag. True means: use fixed (spherical) potential. (default value:False)

`parameters.adaptive_smoothing_flag`
Adaptive-smoothing flag. True means: use of adaptive gravity smoothing for all particles. (default value:False)

`parameters.gadget_cell_opening_flag`
Gadget-cell-opening flag. True means: use of Gadget cell opening criterion. (default value:True)

`parameters.smooth_input_flag`
Smooth-input flag. True means: smooth input SPH properties. (default value:False)

`parameters.conservative_sph_flag`
Conservative-SPH flag. True means: use Springel & Hernquist conservative SPH form (currently the only option). (default value:True)

`parameters.sph_dens_init_flag`
SPH-density-init flag. True means: initialize sph density and h_{smooth} (most probably useless for AMUSE interface). (default value:True)

`parameters.integrate_entropy_flag`
Integrate-entropy flag. True means: integrate entropy, else: internal energy. (default value:True)

`parameters.isothermal_flag`
Isothermal flag. True means: isothermal gas (requires `integrate_entropy_flag == False`). (default value:False)

`parameters.eps_is_h_flag`
Eps-is-h flag. True means: set gas particles gravitational epsilon to h (SPH smoothing length). (default value:True)

`parameters.first_snapshot`
The number of the first snapshot. (default value:0)

`parameters.output_interval`
The number of steps between output. (default value:5)

`parameters.log_interval`
The number of steps between logs. (default value:5)

`parameters.maximum_time_bin`
The maximum time bin ($\text{dtime} * 2^{**} - \text{max_tbin} = \text{minimum time step}$). (default value:4096)

`parameters.minimum_part_per_bin`
The minimum number of particles per time bin. (default value:1)

`parameters.targetnn`
The target number of neighbour particles for variable gravitational eps. (default value:32)

`parameters.verbosity`
The level of terminal output (0=minimum). (default value:0)

`parameters.n_smooth`
The target number of SPH neighbours. (default value:64)

`parameters.periodic_box_size`
The size of simulation domain box (particles outside get deleted). (default value:300.0 length)

`parameters.code_mass_unit`
The code mass unit (in Msun, 10^9 standard). (default value:1000000000.0 MSun)

`parameters.code_length_unit`
The code length unit (in kpc, 1 standard). (default value:1.0 kpc)

`parameters.sqrt_timestep_crit_constant`
Square-root-timestep criterion constant (unitless, standard=1.). (default value:1.0)

`parameters.acc_timestep_crit_constant`
Acceleration-timestep criterion constant (unitless, standard=0.25). (default value:0.25)

`parameters.free_timestep_crit_constant_v`
Freeform-timestep criterion constant v. (default value:0.5)

`parameters.free_timestep_crit_constant_a`
Freeform-timestep criterion constant a. (default value:0.35)

`parameters.free_timestep_crit_constant_vexp`
Freeform-timestep criterion constant v_exp. (default value:0.0)

`parameters.free_timestep_crit_constant_aexp`
Freeform-timestep criterion constant a_exp. (default value:-1.0)

`parameters.opening_angle`
Opening angle, theta, for building the tree: between 0 and 1 (unitless, 0.5). (default value:0.5)

`parameters.gadget_cell_opening_constant`
Gadget-cell-openings criterion parameter (unitless, .01) (default value:0.01)

`parameters.nn_tol`
The fractional tolerance in nn_target (0.1). (default value:0.1)

`parameters.gas_epsilon`
The gas gravitational smoothing epsilon. (default value:0.005 length)

`parameters.gamma`
gas polytropic index (1.6666667) (default value:1.6666667)

`parameters.artificial_viscosity_alpha`
SPH artificial viscosity alpha parameter (0.5) (default value:0.5)

`parameters.beta`
SPH artificial viscosity beta parameter ($2*\alpha=1.0$) (default value:1.0)

`parameters.sph_artificial_viscosity_eps`
SPH artificial viscosity safety against divergence (0.01) (default value:0.01)

`parameters.courant`
SPH courant condition parameter (0.3) (default value:0.3)

`parameters.min_gas_part_mass`
minimum gas particle mass (fraction of initial (average) mass) (default value:0.25)

`parameters.sph_h_const`
SPH smoothing length if constant (default value:0.2 length)

`parameters.n_smooth_tol`
fractional tolerance in number of SPH neighbours (default value:0.1)

`parameters.grain_heat_eff`
FUV grain heating efficiency parameter (unitless, 0.05) (default value:0.05)

`parameters.zeta_cr_ion_rate`
primary cosmic ray ionization rate, zeta (in units of $1.8e-17 \text{ sec}^{-1}$, 1.) (default value: $3.6 \cdot 1.8e-17 * s^{-1}$)

`parameters.heat_par1`
additional heating 1 parameter (0.0) (default value:0.0)

`parameters.heat_par2`
additional heating 2 parameter (0.0) (default value:0.0)

`parameters.cool_par`
additional cooling parameter (1.0) (default value:1.0)

`parameters.optical_depth`
 $1/(\text{mean free path})$ for UV photons (code length $** -1$, 0.0) (default value:0.0)

`parameters.star_form_delay_fac`
star formation delay parameter (unitless, 1) (default value:1.0)

`parameters.star_form_mass_crit`
star formation cloud reference mass (Msun, $1.e5$) (default value:100000.0 MSun)

`parameters.star_form_eff`
gas particle mass fraction converted to stars (0.125) (default value:0.25)

`parameters.supernova_duration`
Supernova activity time, (Myr, $3.e7$) (default value:30000000.0 Myr)

`parameters.supernova_eff`
Supernova feedback coupling efficiency, (0.0) (default value:0.0)

`parameters.t_supernova_start`
Supernova feedback start time, (Myr, $3.e6$) (default value:3000000.0 Myr)

`parameters.max_density`
Maximum permissible density (code density units, 100) (default value:100.0)

`parameters.halofile`
Path to initial halo model file, relative to the Fi data directory (none) (default value:none)

`parameters.feedback`
feedback model (fuv, pres, kine, solo, solh) (default value:fuv)

`parameters.star_formation_mode`
star formation model (gerritsen, nieuw) (default value:gerritsen)

`parameters.h_update_method`
SPH smoothing length criterion (at the moment always 'mass') (default value:mass)

`parameters.sph_viscosity`
 SPH viscosity (sph,sphv, bulk). Note: not all may work. (default value:sph)

`parameters.fi_data_directory`
 Name of the Fi data directory (default value:)

`parameters.periodic_boundaries_flag`
 Periodic boundaries flag. True means: use periodic boundary conditions (read-only) (default value:False)

`parameters.stopping_conditions_timeout`
 max wallclock time available for the evolve step (default value:4.0 s)

`parameters.stopping_conditions_number_of_steps`
 max inner loop evals (default value:1.0)

`parameters.stopping_conditions_out_of_box_size`
 size of cube (default value:0.0 length)

`parameters.stopping_condition_minimum_density`
 minimum density of a gas particle (default value:-1.0 mass / (length**3))

`parameters.stopping_condition_maximum_density`
 maximum density of a gas particle (default value:-1.0 mass / (length**3))

`parameters.stopping_condition_minimum_internal_energy`
 minimum internal energy of a gas particle (default value:-1.0 length**2 / (time**2))

`parameters.stopping_condition_maximum_internal_energy`
 maximum internal energy of a gas particle (default value:-1.0 length**2 / (time**2))

5.4.4 gadget2

GADGET-2 computes gravitational forces with a hierarchical tree algorithm (optionally in combination with a particle-mesh scheme for long-range gravitational forces, currently not supported from the AMUSE interface) and represents fluids by means of smoothed particle hydrodynamics (SPH). The code can be used for studies of isolated systems, or for simulations that include the cosmological expansion of space, both with or without periodic boundary conditions. In all these types of simulations, GADGET follows the evolution of a self-gravitating collisionless N-body system, and allows gas dynamics to be optionally included. Both the force computation and the time stepping of GADGET are fully adaptive, with a dynamic range which is, in principle, unlimited.

The relevant references are:

- Springel V., 2005, MNRAS, 364, 1105 (GADGET-2)
- Springel V., Yoshida N., White S. D. M., 2001, New Astronomy, 6, 51 (GADGET-1)

```
class amuse.community.gadget2.interface.Gadget2 (unit_converter=None,
                                                mode='normal', **options)
```

parameters

`parameters.epsilon_squared`
 smoothing parameter for gravity calculations (default value:9.52140871968e+34 m**2)

`parameters.timestep`
 timestep for system, Gadget2 calculates this by itself, based on particle acceleration. (default value:3.085678e+16 s)

`parameters.no_gravity_flag`
 No-gravity flag. True means: gravitational forces are switched of for all particles (read-only: makefile option NOGRAVITY). (default value:False)

parameters.gadget_cell_opening_flag
Gadget-cell-opening flag. True means: use of Gadget cell opening criterion; Barnes-Hut otherwise (default value:True)

parameters.isothermal_flag
Isothermal flag. True means: isothermal gas, u is interpreted as c_s^2 (read-only: makefile option ISOTHERM_EQS). (default value:False)

parameters.eps_is_h_flag
Eps-is-h flag. True means: set gas particles gravitational epsilon to h (SPH smoothing length) (read-only: makefile option ADAPTIVE_GRAVSOFT_FORGAS). (default value:False)

parameters.n_smooth
The target number of SPH neighbours. (default value:50)

parameters.code_mass_unit
The code mass unit (in g/h, $1.989e43 \text{ g} = 10^{10} \text{ MSun}$ standard). (default value:1.989e+40 kg)

parameters.code_length_unit
The code length unit (in cm/h, $3.085678e21 \text{ cm} = 1 \text{ kpc}$ standard). (default value:3.085678e+19 m)

parameters.code_time_unit
The code time unit (in s/h, default: $3.085678e16 \text{ s} = (1 \text{ kpc}) / (1 \text{ km/s}) \sim 0.9778 \text{ Gyr}$). (default value:3.085678e+16 s)

parameters.code_velocity_unit
The code velocity unit (in cm/s, default: $1e5 \text{ cm/s} = 1 \text{ km/s}$). (default value:1000.0 m * s**-1)

parameters.opening_angle
Opening angle, theta, for building the tree: between 0 and 1 (unitless, 0.5). (default value:0.5)

parameters.gadget_cell_opening_constant
Gadget-cell-openings criterion parameter (unitless, 0.005) (default value:0.005)

parameters.gas_epsilon
The gas gravitational smoothing epsilon. (default value:3.085678e+17 m)

parameters.polytropic_index_gamma
gas polytropic index (1.6666667 or 1 for isothermal(read-only: makefile option ISOTHERM_EQS)). (default value:1.66666666667)

parameters.artificial_viscosity_alpha
SPH artificial viscosity alpha parameter (0.5) (default value:0.5)

parameters.courant
SPH courant condition parameter (0.3). Note that we follow conventional smoothing length definitions, implying a factor 2 difference with Gadget's CourantFac parameter (default value:0.3)

parameters.n_smooth_tol
fractional tolerance in number of SPH neighbours (default value:0.1)

parameters.gadget_output_directory
Name of the Gadget-2 OutputDir (default value:)

parameters.energy_file
The path to the Gadget-2 energy statistics output file. (default value:energy.txt)

parameters.info_file
The path to the Gadget-2 info output file. (default value:info.txt)

parameters.timings_file
The path to the Gadget-2 timings output file. (default value:timings.txt)

parameters.cpu_file
The path to the Gadget-2 cpu statistics output file. (default value:cpu.txt)

- parameters.time_limit_cpu**
The cpu-time limit. Gadget2 will stop once 85% of this (wall-clock) time has passed. (default value:36000 s)
- parameters.comoving_integration_flag**
Flag to do a cosmological run with comoving coordinates. (default value:False)
- parameters.type_of_timestep_criterion**
Timestep criterion to use. Can only be zero: timestep proportional to acceleration^{-0.5} (default value:0)
- parameters.time_begin**
The time at the start of the run. (default value:0.0 s)
- parameters.time_max**
The time at the end of the run. (default value:3.085678e+18 s)
- parameters.omega_zero**
Cosmological matter density parameter in units of the critical density at z=0. (default value:0.0)
- parameters.omega_lambda**
Cosmological vacuum energy density parameter in units of the critical density at z=0. (default value:0.0)
- parameters.omega_baryon**
Cosmological baryonic density parameter in units of the critical density at z=0. (default value:0.0)
- parameters.hubble_param**
The cosmological Hubble parameter. (default value:2.26854550277e-18 s⁻¹)
- parameters.timestep_accuracy_parameter**
Accuracy parameter used in timestep criterion. Actual timesteps are proportional to $\text{err_tol_int_accuracy}^{0.5}$ (default value:0.025)
- parameters.max_size_timestep**
The maximum size of the timestep a particle may take. (default value:3.085678e+14 s)
- parameters.min_size_timestep**
The minimum size of the timestep a particle may take. (default value:0.0 s)
- parameters.tree_domain_update_frequency**
The frequency with which the tree and domain decomposition are fully updated, in terms of (# force computations / # particles). (default value:0.05)
- parameters.time_between_statistics**
The time between statistics output written to the output files. (default value:3.085678e+15 s)
- parameters.min_gas_temp**
The minimum temperature of gas particles. (default value:0.0 K)
- parameters.min_gas_hsmooth_fractional**
The minimum smoothing length of gas particles relative to their softening lengths. (default value:0.0)
- parameters.softening_gas_max_phys**
The maximum physical softening of gas particles for comoving integrations. (default value:0.0 m)
- parameters.softening_halo_max_phys**
The maximum physical softening of dm particles for comoving integrations. (default value:0.0 m)
- parameters.periodic_box_size**
The size of the box in case of periodic boundary conditions. (default value:3.085678e+19 m)
- parameters.periodic_boundaries_flag**
Periodic boundaries flag. True means: use periodic boundary conditions (default value:False)

`parameters.interpret_kicks_as_feedback`
Flag telling Gadget2 whether to interpret external changes to particles' velocities as feedback (for timestepping). (default value:False)

`parameters.interpret_heat_as_feedback`
Flag telling Gadget2 whether to interpret external changes to particles' internal energy as feedback (for timestepping). (default value:True)

`parameters.stopping_conditions_timeout`
max wallclock time available for the evolve step (default value:4.0 s)

`parameters.stopping_conditions_number_of_steps`
max inner loop evals (default value:1.0)

`parameters.stopping_conditions_out_of_box_size`
size of cube (default value:0.0 m)

`parameters.stopping_condition_minimum_density`
minimum density of a gas particle (default value:-6.76991117829e-19 m**3 * kg)

`parameters.stopping_condition_maximum_density`
maximum density of a gas particle (default value:-6.76991117829e-19 m**3 * kg)

`parameters.stopping_condition_minimum_internal_energy`
minimum internal energy of a gas particle (default value:-1000000.0 m**2 * s**-2)

`parameters.stopping_condition_maximum_internal_energy`
maximum internal energy of a gas particle (default value:-1000000.0 m**2 * s**-2)

5.5 Radiative Transfer

- SimpleX (Delaunay triangulation based)

5.5.1 SimpleX

SimpleX computes the transport of radiation on an irregular grid composed of the Delaunay triangulation of a particle set. Radiation is transported along the vertices of the triangulation. The code can be considered as a particle based radiative transfer code: in this case particles sample the gas density, but can be both absorbers and sources of radiation. Calculation time with SimpleX scales linearly with the number of particles. At the moment the code calculates the transport of ionizing radiation in the grey (one frequency) approximation. It is especially well suited to couple with SPH codes.

Specifics

- particle sets send to SimpleX must have attributes x [pc], y[pc], z[pc], rho [amu/cm**3], flux [s**-1] and xion [none].
- care must be taken that the particle sets fit in the box_size
- the default hilbert_order should work for most particle distributions

`class amuse.community.simplex.interface.SimpleX(**options)`

parameters

References:

- Paardekooper J.-P., 2010, PhD thesis, University of Leiden
- Paardekooper J.-P., Kruip, C. J. H., Icke V., 2010, A&A, 515, 79 (SimpleX2)
- Ritzerveld, J., & Icke, V. 2006, Phys. Rev. E, 74, 26704 (SimpleX)

Work in progress

SUPPORTED FILE FORMATS

File input and output

AMUSE can read and write to/from several data formats. The main routines are `write_set_to_file` and `read_set_from_file`, explained below.

6.1 Introduction

The AMUSE framework provides a generic way of reading and writing sets of entities (these entities can be particles, gasclouds or gridpoints). AMUSE provides a function to write a set to a file and a function to read a set from a file. These functions need the name of the file and the format of the data in the file. We will describe these functions first in this chapter. The functions can throw 3 kinds of exceptions, these are described next.

6.2 Use

To write a data set to a space separated text file do:

```
>>> from amuse.io import write_set_to_file
>>> from amuse.datamodel.core import Particles
>>> from amuse.units import units
>>> x = Particles(100)
>>> x.mass = 10.0 | units.MSun
>>> x.radius = range(1.0,101.0) | units.RSun
>>> write_set_to_file(x, "test.csv", "txt", attribute_types = [units.MSun, units.RSun])
```

`amuse.io.write_set_to_file` (*set*, *filename*, *format='csv'*, ***for-*
mat_specific_keyword_arguments)

Write a set to the given file in the given format.

Parameters

- **filename** – name of the file to write the data to
- **format** – name of a registered format or a `FileFormatProcessor` subclass (must be a class and not an instance)

All other keywords are set as attributes on the fileformat processor. To determine the supported options for a processor call `get_options_for_format()`

Registered file formats:

amuse, Process an HDF5 file

csv, Process comma separated files

dyn, Process a Starlab binary structured file

gadget, Process (read) a Gadget binary data file (saving to Gadget files not supported)

hdf5, Process an HDF5 file

nemo, Process a NEMO binary structured file

starlab, Process a Starlab binary structured file

tsf, Process a NEMO binary structured file

txt, Process a text file containing a table of values separated by a predefined character

vts, Process a text file containing a table of values separated by a predefined character

vtu, Process a text file containing a table of values separated by a predefined character

`amuse.io.read_set_from_file` (*filename*, *format*='csv', ***format_specific_keyword_arguments*)
Read a set from the given file in the given format.

Parameters

- **filename** – name of the file to read the data from
- **format** – name of a registered format or a `FileFormatProcessor` subclass (must be a class and not an instance)

All other keywords are set as attributes on the fileformat processor. To determine the supported options for a processor call `get_options_for_format()`

Registered file formats:

amuse, Process an HDF5 file

csv, Process comma separated files

dyn, Process a Starlab binary structured file

gadget, Process (read) a Gadget binary data file (saving to Gadget files not supported)

hdf5, Process an HDF5 file

nemo, Process a NEMO binary structured file

starlab, Process a Starlab binary structured file

tsf, Process a NEMO binary structured file

txt, Process a text file containing a table of values separated by a predefined character

vts, Process a text file containing a table of values separated by a predefined character

vtu, Process a text file containing a table of values separated by a predefined character

`amuse.io.get_options_for_format` (*format*='csv')
Returns a list of tuples, each tuple contains the name of the option, a description of the option and the default values.

Parameters *format* – name of a registered format or a `FileFormatProcessor` subclass (must be a class and not an instance)

6.3 Exceptions

class `amuse.io.UnsupportedFormatException` (**arguments*)
Raised when the given format is not supported by AMUSE.

class `amuse.io.CannotSaveException` (**arguments*)
Raised when the given format cannot save data (only reading of data is supported for the format)

class `amuse.io.CannotLoadException` (**arguments*)
Raised when the given format cannot read data (only saving of data is supported for the format)

6.4 Text files

AMUSE has support for reading and writing text (`.txt`, `.csv`) files. You can specify the txt format by entering “txt” (for space separated files) or “csv” (for comma separated files) as the format:

```
>>> from amuse.support import io
>>> particles = io.read_set_from_file(
    'plummer.txt',
    'txt',
    attribute_types = (units.m, units.m, units.kms),
    attribute_names= ('x', 'y', 'vx')
)
>>> io.write_set_to_file(particles, 'plummer.txt', 'txt')
```

attribute_names

List of the names of the attributes to load or store

key_in_column Column for the key (default is -1, no key stored/loaded). Keys will be interleaved with the data if set other than 0. No attribute type or name can be given for the key.

attribute_types

List of the units of the attributes to store. If not given the units will be derived from the units stored in the attribute set. When derived from the set, the units will always be converted to the base units.

extra_attributes Extra attributes to store with the data set. Some formats (most notably the amuse native format) can store extra attributes with the set in file. The ‘write_set_to_file’ function will collect all keyword arguments that do not match to an option into the extra attributes dictionary.

column_separator

Separator between the columns

precision_of_number_output The precision is a decimal number indicating how many digits should be displayed after the decimal point

header_prefix_string

Lines starting with this character will be handled as part of the header

footer_prefix_string

Lines starting with this character will be handled as part of the footer

quantities

List of vector quantities, each vector quantity is one column in the text file. By default this list will be derived from the particles set. When this option is given it will override the particle set data.

float_format_string format specification string to convert numbers to strings, see `format_spec` in python documentation

By default, text files are stored with some unit information on a comment line in the header. Unfortunately, amuse cannot depend on this line to read the file. When reading a text file you always need to specify the units of every column in the file. For example, to read a file with particle positions where all positions are stored in parsec, do:

```
particles = io.read_set_from_file(
    'positions.txt',
    'txt',
    attribute_types = (units.parsec, units.parsec, units.parsec),
    attribute_names= ('x', 'y', 'z')
)
```

When writing a text file, you do not need to specify the units. If you don’t specify the units the `write_set_to_file` function will default to the units of the quantities being saved. For reliability and reproducibility we suggest to always specify the units and the names of the attributes to save when saving a text file:

```
particles = io.write_set_from_file(
    particles,
    'txt',
    attribute_types = (units.parsec, units.parsec, units.parsec),
    attribute_names= ('x', 'y', 'z')
)
```

The amuse text file routines are very generic and since version 6.0, you can also specify a list of vector quantities instead of a particle set:

```
particles = io.write_set_from_file(
    None,
    'txt',
    attribute_types = (units.parsec, units.parsec, units.parsec),
    attribute_names = ('x', 'y', 'z'),
    quantities = [
        [0.1,0.2] | units.parsec,
        [0.3,0.4] | units.parsec,
        [0.5,0.6] | units.parsec,
    ]
)
```

6.5 Starlab

AMUSE has support for reading and writing starlab (.dyn) files. You can specify the starlab format by entering “dyn” or “starlab” as the format:

```
>>> from amuse.support import io
>>> particles = io.read_set_from_file('plummer.dyn','starlab')
>>> io.write_set_to_file(particles, 'output.dyn', 'dyn')
```

The starlab format support several options, listed below. You can use these options by adding additional keyword arguments to the `read_set_from_file()` or `write_set_to_file()` functions. For example:

```
>>> from amuse.support import io
>>> particles = io.read_set_from_file('plummer.dyn','starlab', must_scale = False, return_children
```

extra_attributes Extra attributes to store with the data set. Some formats (most notably the amuse native format) can store extra attributes with the set in file. The ‘write_set_to_file’ function will collect all keyword arguments that do not match to an option into the extra attributes dictionary.

dynamics_time_units The time fields in the dynamics section of a starlab file can be in Myr or in scaled units, defaults to scaled units (nbody_system.time). When set to scaled units, AMUSE will convert the units if scaling parameters are also given in the file. See the ‘must_scale’ option to turn this scaling off.

dynamics_mass_units The m field in the dynamics section of a starlab file can be in MSun or in scaled units, defaults to scaled units (nbody_system.mass). When set to scaled units, AMUSE will convert the units if scaling parameters are also given in the file. See the ‘must_scale’ option to turn this scaling off

return_converter If True also return the converter when reading a set using `read_set_from_file`

dynamics_length_units The length fields in the dynamics section of a starlab file can be in parsec or in scaled units, defaults to scaled units (nbody_system.length) When set to scaled units, AMUSE will convert the units if scaling parameters are also given in the file. See the ‘must_scale’ option to turn this scaling off.

return_children If True returns the children of the root node, if False returns the root node (defaults to True)

must_scale If True use the scaling values from the file, if False do not scale the stellar dynamics properties. Only used when no nbody to si converter has been set.

nbody_to_si_converter Starlab datafiles store stellar dynamics properties in scaled nbody values, provide a converter to store si data (defaults to None). Value None means no converter, or use scaling values provided in the file

The units of the values in the star (stellar properties) section of a starlab file are always in derived S.I units (solar mass, million years, solar luminosity etc.).

The masses given in de the dynamics section of a starlab file are usually in *nbody* units. Some starlab tools will set the mass values in Solar mass units (for example the `makemass` tool will return the masses in solar mass units). To read these files you need to set the `dynamics_mass_units`.

```
> makeplummer -n 1000 > plummer1.dyn
> cat plummer1.dyn | makemass -f 1 -x -2.0 -l 0.1 -u 20 > plummer2.dyn
> cat plummer2.dyn | add_star -Q 0.5 -R 5 > plummer3.dyn
> cat plummer3.dyn | scale -s > plummer4.dyn
> cat plummer4.dyn | kira -S > plummer5.dyn
```

The `plummer1.dyn`, `plummer4.dyn` and `plummer5.dyn` files will provide masses (and all other dynamical properties) in scaled *nbody* units. The `plummer2.dyn` and `plummer3.dyn` files will have masses in solar masses. To read each file in AMUSE, and return the particles with S.I. units, you need to do:

```
>>> from amuse.support import io
>>> from amuse.units import nbody_system, units
>>> converter = nbody_system.nbody_to_si(1 | units.MSun, 1 | units.parsec)
>>> particles1 = io.read_set_from_file('plummer1.dyn', 'starlab', nbody_to_si_converter = converter)
>>> particles2 = io.read_set_from_file('plummer2.dyn', 'starlab', dynamics_mass_units = units.MSun)
>>> particles3 = io.read_set_from_file('plummer3.dyn', 'starlab', dynamics_mass_units = units.MSun)
>>> particles4 = io.read_set_from_file('plummer4.dyn', 'starlab')
>>> particles5 = io.read_set_from_file('plummer5.dyn', 'starlab')
```

Note: No `nbody` converter object is needed for the last files, as the scale factors given in the files will be used.

The `plummer1.dyn`, `plummer4.dyn` and `plummer5.dyn` can also be read in *nbody* units. In the following example the returned particles have dynamic attributes (mass, radius, velocity, acceleration) in *nbody* units:

```
>>> from amuse.support import io
>>> particles1 = io.read_set_from_file('plummer1.dyn', 'starlab')
>>> particles4 = io.read_set_from_file('plummer4.dyn', 'starlab', must_scale = False)
>>> particles5 = io.read_set_from_file('plummer5.dyn', 'starlab', must_scale = False)
```

6.6 NEMO

AMUSE has support for reading and writing `nemo` (`.tsf`) files. You can specify the starlab format by entering “`nemo`” or “`tsf`” as the format:

```
>>> from amuse.support import io
>>> particles = io.read_set_from_file('plummer.tsf', 'nemo')
>>> io.write_set_to_file(particles, 'output.tsf', 'tsf')
```

The `nemo` format support several options, listed below. You can use these options by adding additional keyword arguments to the `read_set_from_file()` or `write_set_to_file()` functions. For example:

```
>>> from amuse.support import io
>>> from amuse.units import nbody_system, units
>>> converter = nbody_system.nbody_to_si(1 | units.MSun, 1 | units.parsec)
>>> particles = io.read_set_from_file('plummer.nemo', 'tsf', nbody_to_si_converter = converter)
```

extra_attributes Extra attributes to store with the data set. Some formats (moste notably the `amuse` native format) can store extra attributes with the set in file. The ‘`write_set_to_file`’ function will collect all keyword arguments that do not match to an option into the extra attributes dictionary.

nbody_to_si_converter NEMO datafiles store `nbody` data, provide a converter to store `si` data (None means no converter)

6.7 Gadget

AMUSE has support for reading and writing gadget 2 files. You can specify the gadget format by entering “gadget” as the format:

```
>>> from amuse.support import io
>>> gas, halo, disk, bulge, stars, bndry = io.read_set_from_file('plummer.dat', 'gadget')
>>> io.write_set_to_file(particles, 'output.dat', 'gadget')
```

The gadget file format reader will return a tuple of gas, halo, disk, bulge, stars and boundary particles. This is different from other readers which only return one set. The `write_set_to_file` can take a particles set (will be saved as halo particles) or a tuple of gas, halo, disk, bulge, stars and boundary particles.

The gadget format support several options, listed below. You can use these options by adding additional keyword arguments to the `read_set_from_file()` or `write_set_to_file()` functions. For example (will read a gadget file with timestep information):

```
>>> from amuse.support import io
>>> particles = io.read_set_from_file('plummer.nemo', 'gadget', has_timestep = True)
```

is_initial_conditions_format Set to true if the file contains initial conditions. An initial conditions gadget file contains less data.

extra_attributes Extra attributes to store with the data set. Some formats (most notably the amuse native format) can store extra attributes with the set in file. The ‘`write_set_to_file`’ function will collect all keyword arguments that do not match to an option into the extra attributes dictionary.

endianness The endianness of the binary data stored in the file

has_rate_of_entropy_production Set to true if the file has a block with the rate of change of the entropic function of each gas particle

ids_are_long Set to true the ids will be written as longs in the gadget file

header_struct_format The format of the header structure of the gadget file.

has_potential_energy Set to true if the file has a potential energy block

ids_are_keys Set to True if the file contains correct keys. Set to False to generate the keys in amuse and provide an id attribute for the id’s in the gadget file

equal_mass_array If filled with an array with masses > 0.0 assume equal mass for the corresponding set

has_timestep Set to true if the file has a block with the individual timestep for each particle.

has_acceleration Set to true if the file has an acceleration block

REPORTING DURING A RUN

```
class amuse.io.ReportTable (filename, format='csv', **format_specific_keyword_arguments)
```


QUANTITIES AND UNITS

8.1 Introduction

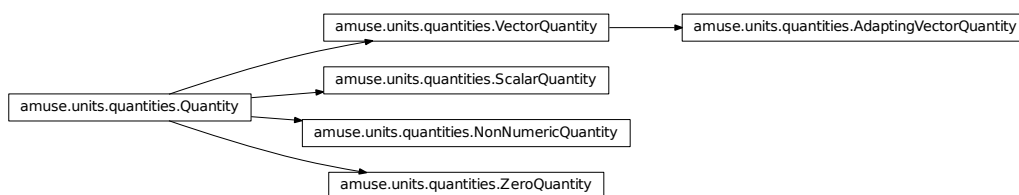
We want to be able to use physical quantities rather than just measures (represented by e.g. floats or integers on computers) in order to raise the ambiguity caused by the implicit choice of units. Serving this purpose, AMUSE comes with a **quantity** class. Like particle sets (fundamental data structure in AMUSE), quantities are fundamental variables. When interacting with code all data has units, even scaled systems. In handling quantities we regard units as being integral part of the mathematical description of the variable [D.C. Ipsen, Units, Dimension, and Dimensionless Numbers, 1960, McGraw-Hill Book Company], i.e. we can state things like:

1 AU = 149597870.691 km

```
>>> from amuse.units import units
>>> q1 = 1.0|units.MSun
>>> q2 = 1.98892e30|units.kg
>>> q1 == q2
True
```

Quantity objects have basic conversion ability, when different units belonging to the same dimension exist the quantity object will convert from one to the other. For more elaborate conversion facilities, like interfacing with a natural unit code, AMUSE provides the generic_unit_converter and the derived nbody_system modules.

8.2 Quantities



class `amuse.units.quantities.Quantity` (*unit*)

A Quantity objects represents a scalar or vector with a specific unit. Quantity is an abstract base class for VectorQuantity and ScalarQuantity.

Quantities should be constructed using *or-operator* (“|”), *new_quantity* or *unit.new_quantity*.

Quantities emulate numeric types.

Examples

```
>>> from amuse.units import units
>>> 100 | units.m
quantity<100 m>
>>> (100 | units.m) + (1 | units.km)
quantity<1100.0 m>
```

Quantities can be tested

```
>>> from amuse.units import units
>>> x = 100 | units.m
>>> x.is_quantity()
True
>>> x.is_scalar()
True
>>> x.is_vector()
False
>>> v = [100, 200, 300] | units.g
>>> v.is_quantity()
True
>>> v.is_scalar()
False
>>> v.is_vector()
True
```

Quantities can be converted to numbers

```
>>> from amuse.units import units
>>> x = 1000.0 | units.m
>>> x.value_in(units.m)
1000.0
>>> x.value_in(units.km)
1.0
>>> x.value_in(units.g) # but only if the units are compatible!
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
IncompatibleUnitsException: Cannot express m in g, the units do not have the same bases
```

as_quantity_in (*another_unit*)

Reproduce quantity in another unit. The new unit must have the same basic si quantities.

Parameters *another_unit* – unit to convert quantity to

Returns quantity converted to new unit

is_quantity ()

True for all quantities.

is_scalar ()

True for scalar quantities.

is_vector ()

True for vector quantities.

sqrt ()

Calculate the square root of each component

```
>>> from amuse.units import units
>>> s1 = 144.0 | units.m**2
>>> s1.sqrt()
quantity<12.0 m>
>>> v1 = [16.0, 25.0, 36.0] | units.kg
>>> v1.sqrt()
quantity<[4.0, 5.0, 6.0] kg**0.5>
```

value_in (*unit*)

Return a numeric value (for scalars) or array (for vectors) in the given unit.

A number is returned without any unit information. Use this function only to transfer values to other libraries that have no support for quantities (for example plotting).

Parameters `unit` – wanted unit of the value

Returns number in the given unit

```
>>> from amuse.units import units
>>> x = 10 | units.km
>>> x.value_in(units.m)
10000.0
```

class `amuse.units.quantities.ScalarQuantity` (*number, unit*)

A `ScalarQuantity` object represents a physical scalar quantity.

class `amuse.units.quantities.VectorQuantity` (*array, unit*)

A `VectorQuantity` object represents a physical vector quantity.

```
>>> from amuse.units import units
>>> v1 = [0.0, 1.0, 2.0] | units.kg
>>> v2 = [2.0, 4.0, 6.0] | units.kg
>>> v1 + v2
quantity<[2.0, 5.0, 8.0] kg>
>>> len(v1)
3
```

`__getitem__` (*index*)

Return the “index” component as a quantity.

Parameters `index` – index of the component, valid values for 3 dimensional vectors are:

[0, 1, 2]

Returns quantity with the same units

```
>>> from amuse.units import si
>>> vector = [0.0, 1.0, 2.0] | si.kg
>>> print vector[1]
1.0 kg
>>> print vector[0:2]
[0.0, 1.0] kg
>>> print vector[[0,2,]]
[0.0, 2.0] kg
```

`__setitem__` (*index, quantity*)

Update the “index” component to the specified quantity.

Parameters `index` – index of the component, valid values for 3 dimensional vectors are:

[0, 1, 2]

Quantity quantity to set, will be converted to the unit of this vector

```
>>> from amuse.units import si
>>> vector = [0.0, 1.0, 2.0] | si.kg
>>> g = si.kg / 1000
>>> vector[1] = 3500 | g
>>> print vector
[0.0, 3.5, 2.0] kg
```

amax (*axis=None*)

Return the maximum along an axis.

```
>>> from amuse.units import si
>>> v1 = [1.0, 2.0, 3.0] | si.kg
>>> v1.amax()
quantity<3.0 kg>
```

amin (*axis=None*)

Return the minimum value along an axis.

```
>>> from amuse.units import si
>>> v1 = [1.0, 2.0, 3.0] | si.kg
>>> v1.amin()
quantity<1.0 kg>
```

append (*scalar_quantity*)

Append a scalar quantity to this vector.

```
>>> from amuse.units import si
>>> vector = [1.0, 2.0, 3.0] | si.kg
>>> vector.append(4.0 | si.kg)
>>> print vector
[1.0, 2.0, 3.0, 4.0] kg
```

argsort (***options*)

Returns the indices that would sort an array.

```
>>> from amuse.units import si
>>> v1 = [3.0, 1.0, 2.0] | si.kg
>>> v1.argsort()
array([1, 2, 0])
```

cross (*other, axisa=-1, axisb=-1, axisc=-1, axis=None*)

Return the cross product of this vector quantity with the supplied vector (quantity).

extend (*vector_quantity*)

Concatenate the vector quantity to this vector. If the units differ, the *vector_quantity* argument is converted to the units of this vector.

```
>>> from amuse.units import units
>>> vector1 = [1.0, 2.0, 3.0] | units.kg
>>> vector2 = [1500, 2500, 6000] | units.g
>>> vector1.extend(vector2)
>>> print vector1
[1.0, 2.0, 3.0, 1.5, 2.5, 6.0] kg
```

length ()

Calculate the length of the vector.

```
>>> from amuse.units import units
>>> v1 = [0.0, 3.0, 4.0] | units.m
>>> v1.length()
quantity<5.0 m>
```

length_squared ()

Calculate the squared length of the vector.

```
>>> from amuse.units import units
>>> v1 = [2.0, 3.0, 4.0] | units.m
>>> v1.length_squared()
quantity<29.0 m**2>
```

lengths ()

Calculate the length of the vectors in this vector.

```
>>> from amuse.units import units
>>> v1 = [[0.0, 3.0, 4.0],[2.0, 2.0, 1.0]] | units.m
>>> v1.lengths()
quantity<[5.0, 3.0] m>
```

lengths_squared ()

Calculate the length of the vectors in this vector

```
>>> from amuse.units import units
>>> v1 = [[0.0, 3.0, 4.0],[4.0, 2.0, 1.0]] | units.m
>>> v1.lengths_squared()
quantity<[25.0, 21.0] m**2>
```

max (*other*)

Return the maximum of self and the argument.

```
>>> from amuse.units import si
>>> v1 = [1.0, 2.0, 3.0] | si.kg
>>> v2 = [0.0, 3.0, 4.0] | si.kg
>>> v1.max(v2)
quantity<[1.0, 3.0, 4.0] kg>
```

min (*other*)

Return the minimum of self and the argument.

```
>>> from amuse.units import si
>>> v1 = [1.0, 2.0, 3.0] | si.kg
>>> v2 = [0.0, 3.0, 4.0] | si.kg
>>> v1.min(v2)
quantity<[0.0, 2.0, 3.0] kg>
```

prepend (*scalar_quantity*)

Prepend the scalar quantity before this vector. If the units differ, the *scalar_quantity* argument is converted to the units of this vector.

```
>>> from amuse.units import units
>>> vector1 = [1.0, 2.0, 3.0] | units.kg
>>> vector1.prepend(0.0 | units.kg)
>>> print vector1
[0.0, 1.0, 2.0, 3.0] kg
```

prod (*axis=None, dtype=None*)

Calculate the product of the vector components

```
>>> from amuse.units import units
>>> v1 = [1.0, 2.0, 3.0] | units.m
>>> v1.prod()
quantity<6.0 m**3>
>>> v1 = [[2.0, 3.0], [2.0, 4.0], [5.0,3.0]] | units.m
>>> v1.prod()
quantity<720.0 m**6>
>>> v1.prod(0)
quantity<[20.0, 36.0] m**3>
>>> v1.prod(1)
quantity<[6.0, 8.0, 15.0] m**2>

>>> v1 = [[[2.0, 3.0], [2.0, 4.0]],[[5.0, 2.0], [3.0, 4.0]]] | units.m
>>> v1.prod()
quantity<5760.0 m**8>
>>> v1.prod(0)
quantity<[[ 10.  6.], [  6. 16.]] m**2>
>>> v1.prod(1)
quantity<[[ 4. 12.], [15.  8.]] m**2>
>>> v1.prod(2)
quantity<[[ 6.  8.], [10. 12.]] m**2>
```

sorted ()

Return a new vector with all items sorted.

```
>>> from amuse.units import si
>>> v1 = [3.0, 1.0, 2.0] | si.kg
```

```
>>> v1.sorted()
quantity<[1.0, 2.0, 3.0] kg>
```

sorted_with (*others)

Return a new vector with all items sorted. Perform all the same move operations on the other vectors.

Argument kind, the sort method for supported kinds see the numpy.sort documentation

```
>>> from amuse.units import si
>>> v1 = [3.0, 1.0, 2.0] | si.kg
>>> v2 = [2.0, 3.0, 2.0] | si.m
>>> v3 = [1.0, 4.0, 5.0] | si.s
>>> list(v1.sorted_with(v2, v3))
[quantity<[1.0, 2.0, 3.0] kg>, quantity<[3.0, 2.0, 2.0] m>, quantity<[4.0, 5.0, 1.0] s>]
```

sum (axis=None, dtype=None, out=None)

Calculate the sum of the vector components

```
>>> from amuse.units import units
>>> v1 = [0.0, 1.0, 2.0] | units.kg
>>> v1.sum()
quantity<3.0 kg>
```

x

The x axis component of a 3 dimensional vector. This is equivalent to the first component of vector.

Returns x axis component as a quantity

```
>>> from amuse.units import si
>>> vector = [1.0, 2.0, 3.0] | si.kg
>>> print vector.x
1.0 kg
```

y

The y axis component of a 3 dimensional vector. This is equivalent to the second component of vector.

Returns y axis component as a quantity

```
>>> from amuse.units import si
>>> vector = [1.0, 2.0, 3.0] | si.kg
>>> print vector.y
2.0 kg
```

z

The z axis component of a 3 dimensional vector. This is equivalent to the third component of vector.

Returns z axis component as a quantity

```
>>> from amuse.units import si
>>> vector = [1.0, 2.0, 3.0] | si.kg
>>> print vector.z
3.0 kg
```

class amuse.units.quantities.**NonNumericQuantity** (value, unit)

A Non Numeric Quantity object represents a quantity without a physical meaning.

These Quantity objects cannot be used in numeric operations (like addition or multiplication). Also, conversion to another unit is not possible.

Examples are string quantities or enumerated value quantities.

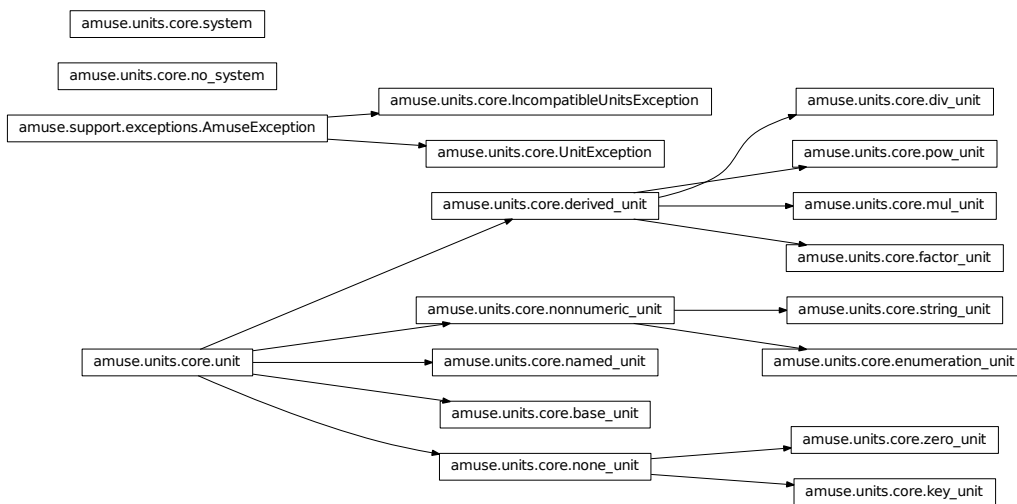
```
>>> from amuse.units.core import enumeration_unit
>>> my_unit = enumeration_unit(
...     "x",
...     "x",
...     [1, 3, 4],
```

```

...     ["first", "second", "third"])
...
>>> 3 | my_unit
quantity<3 - second>
>>> (3 | my_unit).value_in(my_unit)
3

```

8.3 Units



class `amuse.units.core.base_unit` (*quantity, name, symbol, system*)
`base_unit` objects are orthogonal, indivisible units of a system of units.

A system of units contains a set of base units

Parameters

- **quantity** – name of the base quantity, for example *length*
- **name** – name of the unit, for example *meter*
- **symbol** – symbol of the unit, for example *m*
- **system** – system of units object

```

>>> cgs = system("cgs")
>>> cm = base_unit("length", "centimetre", "cm", cgs)
>>> cm
unit<cm>

```

base

The base represented as a list of tuples. Each tuple consists of a power and a unit.

factor

The multiplication factor of a unit. For example, factor is 1000 for km.

class `amuse.units.core.derived_unit`

Abstract base class of derived units. New units can be derived from `base_units`. Each operation on a unit creates a new `derived_unit`.

class `amuse.units.core.div_unit` (*left_hand*, *right_hand*)

A `div_unit` object defines a unit multiplied by another unit. Do not call this method directly, `div_unit` objects are supposed to be created by dividing units.

Parameters

- **left_hand** – Left hand side of the multiplication.
- **right_hand** – Right hand side of the multiplication.

```
>>> from amuse.units import si
>>> speed = si.m / si.s
>>> speed
unit<m / s>
>>> speed_with_powers = si.m * si.s ** -1
>>> speed.as_quantity_in(speed_with_powers)
quantity<1.0 m * s**-1>
```

class `amuse.units.core.enumeration_unit` (*name*, *symbol*, *possible_values=None*, *names_for_values=None*)

Enumeration unit objects define a fixed set of quantities.

A quantity with a `enumeration_unit` can only have a value defined in the set of values of the `enumeration_unit`.

Parameters

- **possible_values** – A sequence or iterable with all the possible values. If `None` the possible values are integers ranging from 0 to the length of the `names_for_values` argument
- **names_for_values** – A sequence of strings defining a display name for each value. If `None` the names are the string vales of the values in the `possible_values` arguments

Examples

```
>>> my_unit = enumeration_unit('my_unit', 'my_unit', [1,2,5], ["star", "gas", "planet"])
>>> 2 | my_unit
quantity<2 - gas>
>>> list(my_unit.quantities())
[quantity<1 - star>, quantity<2 - gas>, quantity<5 - planet>]
>>> 3 | my_unit
Traceback (most recent call last):
...
AmuseException: <3> is not a valid value for unit<my_unit>
```

Or, with default values:

```
>>> my_unit = enumeration_unit('my_unit', 'my_unit', None, ["star", "gas", "planet"])
>>> 2 | my_unit
quantity<2 - planet>
>>> list(my_unit.quantities())
[quantity<0 - star>, quantity<1 - gas>, quantity<2 - planet>]
```

class `amuse.units.core.factor_unit` (*factor*, *unit*, *name=None*, *symbol=None*)

A `factor_unit` object defines a unit multiplied by a number. Do not call this method directly, `factor_unit` objects are supposed to be created by multiplying a number with a unit.

Parameters

- **unit** – The unit to derive from.
- **factor** – The multiplication factor.

```
>>> from amuse.units import si
>>> minute = 60.0 * si.s
>>> minute.as_quantity_in(si.s)
quantity<60.0 s>
>>> hour = 60.0 * minute
```

```
>>> hour
unit<60.0 * 60.0 * s>
>>> hour.as_quantity_in(si.s)
quantity<3600.0 s>
```

`amuse.units.core.k` (*unit*)

Create a new factor unit with factor 1000. Prepend *k* to the symbol of the unit.

This is a very simple utility function that only prepends a name and has no knowledge about anything else than kilo's.

Examples

```
>>> from amuse.units import si
>>> km = k(si.m)
>>> km
unit<km>
>>> kkg = k(si.kg)
>>> kkg
unit<kkg>
>>> kkg.as_quantity_in(si.kg)
quantity<1000.0 kg>
```

`class` `amuse.units.core.mul_unit` (*left_hand*, *right_hand*)

A `mul_unit` object defines a unit multiplied by another unit. Do not call this method directly, `mul_unit` objects are supposed to be created by multiplying units.

Parameters

- **left_hand** – Left hand side of the multiplication.
- **right_hand** – Right hand side of the multiplication.

```
>>> from amuse.units import si
>>> area = si.m * si.m
>>> area
unit<m * m>
>>> hectare = (100 * si.m) * (100 * si.m)
>>> hectare.as_quantity_in(area)
quantity<10000.0 m * m>
```

`class` `amuse.units.core.named_unit` (*name*, *symbol*, *unit*)

A `named_unit` object defines an alias for another unit. When printing a `named_unit`, the symbol is shown and not the unit parts. For all other operations the `named_units` works exactly like the aliased unit.

Parameters

- **name** – Long name or description of the unit
- **symbol** – Short name to show when printing units or quantities
- **unit** – The unit to alias

```
>>> from amuse.units import si
>>> 60 * si.s
unit<60 * s>
>>> minute = named_unit("minute", "min", 60*si.s)
>>> minute
unit<min>
>>> (20 | (60 * si.s)).as_quantity_in(minute)
quantity<20.0 min>
```

`class` `amuse.units.core.nonnumeric_unit` (*name*, *symbol*)

`nonnumeric_unit` objects are indivisible units not connected to any system of units.

`nonnumeric_units` cannot be used to derive new units from.

nonnumeric_units have no physical meaning.

class `amuse.units.core.pow_unit` (*power, unit*)

A `pow_unit` object defines a unit as another unit to a specified power.

Do not call this method directly, `pow_unit` objects are supposed to be created by taking powers of units.

Parameters

- **power** – Power of the unit
- **unit** – The unit to derive from

```
>>> from amuse.units import si
>>> area = si.m**2
>>> area
unit<m**2>
>>> area.as_quantity_in(si.m * si.m)
quantity<1.0 m * m>
>>> hectare = (100 * si.m) ** 2
>>> hectare.as_quantity_in(area)
quantity<10000.0 m**2>
```

class `amuse.units.core.string_unit` (*name, symbol*)

String unit objects define quantities with a string value. These have no physical meaning, but are needed for some legacy codes. For example the path of a file.

class `amuse.units.core.unit`

Abstract base class for unit objects.

Two classes of units are defined:

base units The base units in a given system of units. For SI, these are meter, kilogram, second, ampere, kelvin, mole and candele. See the `si` module `amuse.units.si`

derived units Derived units are created by dividing or multiplying with a number or with another unit. For example, to get a velocity unit we can devine `vel = 1000 * m / s`

Units can also be named, by creating a named unit.

as_quantity_in (*unit*)

Express this unit as a quantity in the given unit

Parameters **unit** – The unit to express this unit in

Result A Quantity object

Examples

```
>>> from amuse.units import units
>>> ton = 1000 * units.kg
>>> ton.as_quantity_in(units.kg)
quantity<1000.0 kg>
```

has_same_base_as (*other*)

Determine if the base of *other* is the same as the base of self.

Parameters **other** – unit to compare base to

Result True, if bases are compatible.

```
>>> from amuse.units import units
>>> mps = units.m / units.s
>>> kph = units.km / units.hour
>>> mps.has_same_base_as(kph)
True
>>> mps.has_same_base_as(units.km)
False
```

in_(*x*)

Express this quantity in the given unit

Parameters **unit** – The unit to express this quantity in**Result** A Quantity object

Examples

```
>>> from amuse.units import units
>>> l = 1 | units.AU
>>> l.in_(units.km)
quantity<149597870.691 km>
```

new_quantity(*value*)

Create a new Quantity object.

Parameters **value** – numeric value of the quantity, can be a number or a sequence (list or ndarray)**Returns** new ScalarQuantity or VectorQuantity object with this unit**to_simple_form**()

Convert unit to a form with only one factor and powers

Result Unit with only a factor and power terms

```
>>> from amuse.units import units
>>> N = (units.m * units.kg) / (units.s * units.s)
>>> N
unit<m * kg / (s * s)>
>>> J = N * units.m
>>> J
unit<m * kg / (s * s) * m>
>>> J.to_simple_form()
unit<m**2 * kg * s**-2>
```

value_in(*unit*)

Return a numeric value of this unit in the given unit. Works only when the units are compatible, i.e. from tonnage to kg's.

A number is returned without any unit information.

Parameters **unit** – wanted unit of the value**Returns** number in the given unit

```
>>> from amuse.units import units
>>> x = units.km
>>> x.value_in(units.m)
1000.0
```

8.4 Unit systems and converters

The amuse framework gives you the ability to choose a unit system for your model through the 'generic_unit_converter' module. This enables you to work with e.g. natural units or n-body units within AMUSE.

The implementation makes use of a dimension-space, which is a vector-space where the chosen units form a base. For a detailed description of the method see e.g.: Maksymowicz, A, *American Journal of Physics*, Vol.44, No.3, 1976.

8.4.1 Generic units

The generic unit system knows the seven base quantities in the International System of Quantities, I.S.Q.

Base quantity	Name in generic unit	Name in S.I. unit
length	generic_system.length	units.m
time	generic_system.time	units.s
mass	generic_system.mass	units.kg
current	generic_system.current	units.A
temperature	generic_system.temperature	units.K
amount of substance	generic_system.amount_of_substance	units.mol
luminous intensity	generic_system.luminous_intensity	units.cd

8.4.2 N-body units

The n-body unit system knows the three base quantities in the International System of Quantities, I.S.Q. and defines the gravitational constant to be 1:

$$G = 1 \text{ l} / (\text{mass} * (\text{time}^{**2}))$$

Base quantity	Name in generic unit	Name in S.I. unit
length	generic_system.length	units.m
time	generic_system.time	units.s
mass	generic_system.mass	units.kg

Derived quantities

acceleration	length / (time ** 2)
potential	(length ** 2) / (time ** 2)
energy	mass * potential
specific_energy	potential
speed	length / time
volume	(length ** 3)
density	mass / volume
momentum_density	density * speed
energy_density	density * specific_energy

8.4.3 Converter

class `amuse.units.generic_unit_converter.ConvertBetweenGenericAndSiUnits` (**arguments_list*)

A `ConvertBetweenGenericAndSiUnits` object is a converter from arbitrary units which user gives, to si units (and vice versa).

The “`generic_unit_converter`” `ConvertBetweenGenericAndSiUnits` is the actual class through which you define the unit system. Upon instantiation you choose the base units. In the example below we chose the speed of light as a unit: $c = 1$ unit length/second, and the second as the unit of time.

Note that the system has two base dimensions, length and time. By the second argument we have assigned the unit second to time and by the requirement that unit length / second equals one, the new unit length will be $\{c\}$ meters in S.I. units.

Example:

```
>>> from amuse.units.generic_unit_converter import ConvertBetweenGenericAndSiUnits
>>> from amuse.units import units, constants
>>> converter = ConvertBetweenGenericAndSiUnits(constants.c, units.m)
```

to_generic (*value*)

Convert a quantity in S.I units to a quantity in generic units.

```
>>> from amuse.units.generic_unit_converter import ConvertBetweenGenericAndSiUnits
>>> from amuse.units import units, constants
>>> converter = ConvertBetweenGenericAndSiUnits(constants.c, units.s)
```

```
>>> print converter.to_generic(constants.c)
1.0 length * time**-1
```

to_si (*value*)

Convert a quantity in generic units to a quantity in S.I. units.

```
>>> from amuse.units.generic_unit_converter import ConvertBetweenGenericAndSiUnits
>>> from amuse.units import units, constants
>>> converter = ConvertBetweenGenericAndSiUnits(constants.c, units.s)
>>> print converter.to_si(length)
299792458.0 m
```

8.4.4 Use with codes

For convenience, the gravitational dynamics interface works with a unit-converter which converts between the units used by the code and the preferred units in the script (user's choice).

We show two examples. The first one uses the (derived from the generic units converter) `nbody_system` converter, which is the logical choice for dynamics (n-body) codes.

The second one uses the generic unit converter directly, this is just an example.

Usage: example 1

```
>>> from amuse.community.hermite0.interface import Hermite
>>> from amuse.units import nbody_system
>>> from amuse.units import constants, units
>>> convert_nbody = nbody_system.nbody_to_si(1.0 | units.MSun, 149.5e6 | units.km)
>>> hermite = Hermite(convert_nbody)
```

Usage: example 2

```
>>> from amuse.community.hermite0.interface import Hermite
>>> from amuse.units import generic_unit_converter as gc
>>> from amuse.units import constants, units
>>> converter = gc.ConvertBetweenGenericAndSiUnits(constants.G, 1.0 | units.MSun, 1 | units.AU)
>>> hermite = Hermite(converter)
```

Example

More examples can be found in the tutorial, *working_with_units* Stellar Dynamics Interface Definition

```
class amuse.community.interface.gd.GravitationalDynamics (legacy_interface,
                                                         unit_converter=None,
                                                         **options)
```


DATAMODEL

9.1 Introduction

The AMUSE datamodel is based on sets of particles.

Different astrophysical objects are all referred to as Particles in the AMUSE code. These objects can be:

- Star
- Black hole
- Gas Particle
- Dark Matter Particle

The astrophysical objects are all modelled using the `Particle` class. Objects of this class can have a varying set of attributes. These attributes can be defined by a script or a code.

9.2 Identity

All particles have a unique 64-bit key. This key is generated using a random number generator. The chances of duplicate keys using 64-bit integers are finite but very low. This chance of a duplicate key can be determined by a generalization of the birthday problem.

Duplicate keys::

```
>>> # given n random integers drawn from a discrete uniform distribution
>>> # with range [1,highest_integer], what is the probability
>>> # p(n;highest_integer) that at least two numbers are the same?
>>> import math
>>> number_of_bits = 64
>>> highest_integer = 2**number_of_bits
>>> number_of_particles = 1000000.0 # one million
>>> probability = 1.0 - math.exp( (-number_of_particles * (number_of_particles - 1.0)) / (2.0 *
>>> print probability
2.71050268896e-08
>>> # can also set the probability and determine the set size
>>> probability = 0.00001 # 0.001 percent
>>> number_of_particles = math.sqrt(2 * highest_integer * math.log(1 / (1.0 - probability)))
>>> print number_of_particles
19207725.6894
```

If you use large sets or want to load a lot of simulations with different particles into a script the probability of encountering a duplicate may be too high. An extension to more larger key sizes is planned but not yet implemented. You can check for duplicates in a set of particles by calling `has_duplicates` on a set.

9.3 Sets of particles

The AMUSE datamodel assumes all particles come in sets. The data of a particle is stored in the set. This module provides access to all set handling in AMUSE. The actual implementation is in the base, storage and particle modules.

class `amuse.datamodel.AbstractParticleSet` (*original=None*)

Abstract superclass of all sets of particles. This class defines common code for all particle sets.

Particle sets define dynamic attributes. Attributes can be set and retrieved on the particles using common python syntax. These attributes can only have values with units.

```
>>> particles = Particles(2)
>>> particles.mass = [10.0, 20.0] | units.kg
>>> particles.mass
quantity<[10.0, 20.0] kg>
```

```
>>> particles.mass = 1.0 | units.kg
>>> particles.mass
quantity<[1.0, 1.0] kg>
```

Particle sets can be iterated over.

```
>>> particles = Particles(2)
>>> particles.mass = [10.0, 20.0] | units.kg
>>> for particle in particles:
...     print particle.mass
...
10.0 kg
20.0 kg
```

Particle sets can be indexed.

```
>>> particles = Particles(3)
>>> particles.x = [10.0, 20.0, 30.0] | units.m
>>> particles[1].x
quantity<20.0 m>
```

Particle sets can be sliced.

```
>>> particles = Particles(3)
>>> particles.x = [10.0, 20.0, 30.0] | units.m
>>> particles[1:].x
quantity<[20.0, 30.0] m>
```

Particle sets can be copied.

```
>>> particles = Particles(3)
>>> particles.x = [10.0, 20.0, 30.0] | units.m
>>> copy = particles.copy()
>>> particles.x = 2.0 | units.m
>>> particles.x
quantity<[2.0, 2.0, 2.0] m>
>>> copy.x
quantity<[10.0, 20.0, 30.0] m>
```

Particle sets can be added together. Attribute values are not stored by the resulting subset. The subset provides a view on two or more sets of particles. Changing attributes of the sum of sets will also change the attributes of each original set, contrary to `copy()`.

```
>>> particles = Particles(4)
>>> particles1 = particles[:2]
>>> particles1.x = [1.0, 2.0] | units.m
>>> particles2 = particles[2:]
```

```

>>> particles2.x = [3.0, 4.0] | units.m
>>> new_set = particles1 + particles2
>>> print len(new_set)
4
>>> print new_set.x
[1.0, 2.0, 3.0, 4.0] m

```

Particle sets can be subtracted from each other. Like with addition, attribute values are not stored by the resulting subset.

```

>>> particles = Particles(4)
>>> particles.x = [1.0, 2.0, 3.0, 4.0] | units.m
>>> junk = particles[2:]
>>> new_set = particles - junk
>>> print len(new_set)
2
>>> print new_set.x
[1.0, 2.0] m
>>> print particles.x
[1.0, 2.0, 3.0, 4.0] m

```

Particle sets can have instance based or global vector attributes. A particle set stores a list of scalar values for each attribute. Some attributes are more naturally accessed as lists of vector values. Once defined, a particle set can convert the scalar values of 2 or more attributes into one vector attribute.

```

>>> from amuse.support.data import particle_attributes
>>> particles = Particles(2)
>>> particles.x = [1.0, 2.0] | units.m
>>> particles.y = [3.0, 4.0] | units.m
>>> particles.z = [5.0, 6.0] | units.m
>>> particles.add_vector_attribute("p", ["x", "y", "z"])
>>> particles.p
quantity<[[ 1.  3.  5.], [ 2.  4.  6.]] m>
>>> particles.p[0]
quantity<[1.0, 3.0, 5.0] m>
>>> particles.position # "position" is a global vector attribute, coupled to x,y,z
quantity<[[ 1.  3.  5.], [ 2.  4.  6.]] m>

```

`__add__` (*particles*)

Returns a particle subset, composed of the given particle(s) and this particle set. Attribute values are not stored by the subset. The subset provides a view on two or more sets of particles.

Parameters `particles` – (set of) particle(s) to be added to self.

```

>>> particles = Particles(4)
>>> particles1 = particles[:2]
>>> particles1.x = [1.0, 2.0] | units.m
>>> particles2 = particles[2:]
>>> particles2.x = [3.0, 4.0] | units.m
>>> new_set = particles1 + particles2
>>> new_set
<amuse.datamodel.particles.ParticlesSubset object at 0x...>
>>> print len(new_set)
4
>>> print new_set.x
[1.0, 2.0, 3.0, 4.0] m

```

`__sub__` (*particles*)

Returns a subset of the set without the given particle(s) Attribute values are not stored by the subset. The subset provides a view on two or more sets of particles.

Parameters `particles` – (set of) particle(s) to be subtracted from self.

```
>>> particles = Particles(4)
>>> particles.x = [1.0, 2.0, 3.0, 4.0] | units.m
>>> junk = particles[2:]
>>> new_set = particles - junk
>>> new_set
<amuse.datamodel.particles.ParticlesSubset object at 0x...>
>>> print len(new_set)
2
>>> print new_set.x
[1.0, 2.0] m
>>> print particles.x
[1.0, 2.0, 3.0, 4.0] m
```

add_particle (*particle*)

Add one particle to the set.

Parameters *particle* – particle to add

```
>>> particles = Particles()
>>> print len(particles)
0
>>> particle = Particle()
>>> particle.x = 1.0 | units.m
>>> particles.add_particle(particle)
<amuse.datamodel.particles.Particle object at ...>
>>> print len(particles)
1
>>> print particles.x
[1.0] m
```

add_particles (*particles*)

Adds particles from the supplied set to this set. Attributes and values are copied over.

Note: For performance reasons the particles are not checked for duplicates. When the same particle is part of both sets errors may occur.

Parameters *particles* – set of particles to copy values from

```
>>> particles1 = Particles(2)
>>> particles1.x = [1.0, 2.0] | units.m
>>> particles2 = Particles(2)
>>> particles2.x = [3.0, 4.0] | units.m
>>> particles1.add_particles(particles2)
<amuse.datamodel.particles.ParticlesSubset object at 0x...>
>>> print len(particles1)
4
>>> print particles1.x
[1.0, 2.0, 3.0, 4.0] m
```

as_set ()

Returns a subset view on this set. The subset will contain all particles of this set.

```
>>> particles = Particles(3)
>>> particles.x = [1.0, 2.0, 3.0] | units.m
>>> subset = particles.as_set()
>>> print subset.x
[1.0, 2.0, 3.0] m
>>> print particles.x
[1.0, 2.0, 3.0] m
```

copy_values_of_attribute_to (*attribute_name*, *particles*)

Copy values of one attribute from this set to the other set. Will only copy values for the particles in

both sets. See also `synchronize_to()`.

If you need to do this a lot, setup a dedicated channel.

```
>>> particles1 = Particles(2)
>>> particles1.x = [1.0, 2.0] | units.m
>>> particles2 = particles1.copy()
>>> print particles2.x
[1.0, 2.0] m
>>> p3 = particles1.add_particle(Particle())
>>> particles1.x = [3.0, 4.0, 5.0] | units.m
>>> particles1.copy_values_of_attribute_to("x", particles2)
>>> print particles2.x
[3.0, 4.0] m
```

difference (*other*)

Returns a new subset containing the difference between this set and the provided set.

```
>>> particles = Particles(3)
>>> particles.mass = [10.0, 20.0, 30.0] | units.kg
>>> particles.x = [1.0, 2.0, 3.0] | units.m
>>> subset = particles.select(lambda x : x > 15.0 | units.kg, ["mass"])
>>> less_than_15kg = particles.difference(subset)
>>> len(subset)
2
>>> len(less_than_15kg)
1
```

has_duplicates ()

Returns True when a set contains a particle with the same key more than once. Particles with the same key are interpreted as the same particles.

```
>>> particles = Particles()
>>> p1 = particles.add_particle(Particle(1))
>>> p2 = particles.add_particle(Particle(2))
>>> particles.has_duplicates()
False
>>> p3 = particles.add_particle(Particle(1))
>>> particles.has_duplicates()
True
>>> p3 == p1
True
```

remove_particle (*particle*)

Removes a particle from this set.

Result is undefined if particle is not part of the set

Parameters *particle* – particle to remove from this set

```
>>> particles1 = Particles(2)
>>> particles1.x = [1.0, 2.0] | units.m
>>> particles1.remove_particle(particles1[0])
>>> print len(particles1)
1
>>> print particles1.x
[2.0] m
```

remove_particles (*particles*)

Removes particles from the supplied set from this set.

Parameters *particles* – set of particles to remove from this set

```
>>> particles1 = Particles(2)
>>> particles1.x = [1.0, 2.0] | units.m
>>> particles2 = Particles()
```

```
>>> particles2.add_particle(particles1[0])
<amuse.datamodel.particles.Particle object at ...>
>>> particles1.remove_particles(particles2)
>>> print len(particles1)
1
>>> print particles1.x
[2.0] m
```

reversed()

Returns a subset with the same particles, but with reversed sequential order (the first particle will become last)

```
>>> particles = Particles(3)
>>> particles.radius = [1.0, 2.0, 3.0] | units.m
>>> r = particles.reversed()
>>> print r.radius
[3.0, 2.0, 1.0] m
```

select (*selection_function, attributes*)

Returns a subset view on this set. The subset will contain all particles for which the selection function returned True. The selection function is called with scalar quantities defined by the attributes parameter

```
>>> particles = Particles(3)
>>> particles.mass = [10.0, 20.0, 30.0] | units.kg
>>> particles.x = [1.0, 2.0, 3.0] | units.m
>>> subset = particles.select(lambda x : x > 15.0 | units.kg, ["mass"])
>>> print subset.mass
[20.0, 30.0] kg
>>> print subset.x
[2.0, 3.0] m
```

select_array (*selection_function, attributes=()*)

Returns a subset view on this set. The subset will contain all particles for which the selection function returned True. The selection function is called with a vector quantities containing all the values for the attributes parameter.

This function can be faster than the select function as it works on entire arrays. The selection_function is called once.

```
>>> particles = Particles(3)
>>> particles.mass = [10.0, 20.0, 30.0] | units.kg
>>> particles.x = [1.0, 2.0, 3.0] | units.m
>>> subset = particles.select_array(lambda x : x > 15.0 | units.kg, ["mass"])
>>> print subset.mass
[20.0, 30.0] kg
>>> print subset.x
[2.0, 3.0] m
```

```
>>> particles = Particles(1000)
>>> particles.x = units.m.new_quantity(numpy.arange(1,1000))
>>> subset = particles.select_array(lambda x : x > (500 | units.m), ("x",) )
>>> print len(subset)
499
```

sorted_by_attribute (*attribute, kind='mergesort'*)

Returns a subset with the same particles, but sorted using the given attribute name

Argument kind, the sort method for supported kinds see the numpy.sort documentation

```
>>> particles = Particles(3)
>>> particles.mass = [2.0, 3.0, 1.0] | units.kg
>>> particles.radius = [1.0, 2.0, 3.0] | units.m
>>> sorted = particles.sorted_by_attribute('mass')
>>> print sorted.mass
```

```
[1.0, 2.0, 3.0] kg
>>> print sorted.radius
[3.0, 1.0, 2.0] m
```

sorted_by_attributes (*attributes)

Returns a subset with the same particles, but sorted using the given attribute names. The last attribute name in the call is used for the primary sort order, the second-to-last attribute name for the secondary sort order, and so on. See also `numpy.lexsort`

```
>>> particles = Particles(4)
>>> particles.mass = [2.0, 3.0, 1.0, 4.0] | units.kg
>>> particles.radius = [3.0, 2.0, 1.0, 2.0] | units.m
>>> sorted = particles.sorted_by_attributes('mass', 'radius')
>>> print sorted.radius
[1.0, 2.0, 2.0, 3.0] m
>>> print sorted.mass
[1.0, 3.0, 4.0, 2.0] kg
```

synchronize_to (other_particles)

Synchronize the particles of this set with the contents of the provided set.

After this call the *other_particles* set will have the same particles as this set.

This call will check if particles have been removed or added it will not copy values of existing particles over.

Parameters *other_particles* – particle set which has to be updated

```
>>> particles = Particles(2)
>>> particles.x = [1.0, 2.0] | units.m
>>> copy = particles.copy()
>>> new_particle = Particle()
>>> new_particle.x = 3.0 | units.m
>>> particles.add_particle(new_particle)
<amuse.datamodel.particles.Particle object at ...>
>>> print particles.x
[1.0, 2.0, 3.0] m
>>> print copy.x
[1.0, 2.0] m
>>> particles.synchronize_to(copy)
>>> print copy.x
[1.0, 2.0, 3.0] m
```

to_string (attributes_to_show=None)

Display string of a particle set.

```
>>> p0 = Particle(10)
>>> p1 = Particle(11)
>>> particles = Particles()
>>> particles.add_particle(p0)
<amuse.datamodel.particles.Particle object at ...>
>>> particles.add_particle(p1)
<amuse.datamodel.particles.Particle object at ...>
>>> particles.x = [4.0 , 3.0] | units.m
>>> particles.y = [5.0 , 2.0] | units.km
>>> print particles
```

key	x	y
-	m	km
10	4.000e+00	5.000e+00
11	3.000e+00	2.000e+00

```
class amuse.datamodel.Particles (size=0, storage=None, keys=None, keys_generator=None,
                                particles=None, **attributes)
```

A set of particles. Attributes and values are stored in a private storage model. This storage model can store the values in the python memory space, in the memory space of the code or in a HDF5 file. By default the storage model is in memory.

class `amuse.datamodel.ParticlesSubset` (*particles, keys*)

A subset of particles. Attribute values are not stored by the subset. The subset provides a limited view to the particles.

Particle subset objects are not supposed to be created directly. Instead use the `to_set` or `select` methods.

add_particles_to_store (*keys, attributes=[]*, *values=[]*)

Adds particles from to the subset, also adds the particles to the superset

remove_particles_from_store (*keys*)

Removes particles from the subset, and removes particles from the super set

union (*other*)

Returns a new subset containing the union between this set and the provided set.

```
>>> particles = Particles(3)
>>> particles.mass = [10.0, 20.0, 30.0] | units.kg
>>> subset1 = particles.select(lambda x : x > 25.0 | units.kg, ["mass"])
>>> subset2 = particles.select(lambda x : x < 15.0 | units.kg, ["mass"])
>>> union = subset1.union(subset2)
>>> len(union)
2
>>> sorted(union.mass.value_in(units.kg))
[10.0, 30.0]
```

class `amuse.datamodel.ParticlesWithUnitsConverted` (*particles, converter*)

A view on a particle sets. Used when to convert values between incompatible sets of units. For example to convert from si units to nbody units.

The converter must have implement the `ConverterInterface`.

```
>>> from amuse.units import nbody_system
>>> particles_nbody = Particles(2)
>>> particles_nbody.x = [10.0, 20.0] | nbody_system.length
>>> convert_nbody = nbody_system.nbody_to_si(10 | units.kg, 5 | units.m)
>>> particles_si = ParticlesWithUnitsConverted(
...     particles_nbody,
...     convert_nbody.as_converter_from_si_to_nbody())
...
>>> print particles_nbody.x
[10.0, 20.0] length
>>> print particles_si.x
[50.0, 100.0] m
>>> particles_si.x = [200.0, 400.0] | units.m
>>> print particles_nbody.x
[40.0, 80.0] length
```

class `ConverterInterface`

Interface definition for the converter.

source The source quantity is in the units of the user of a `ParticlesWithUnitsConverted` object

target The target quantity must be in the units of the internal particles set.

from_source_to_target (*quantity*)

Converts the quantity from the source units to the target units.

Parameters *quantity* – quantity to convert

from_target_to_source (*quantity*)

Converts the quantity from the target units to the source units.

Parameters *quantity* – quantity to convert

9.3.1 object

class `amuse.datamodel.Particle` (*key=None, particles_set=None, **keyword_arguments*)

A physical object or a physical region simulated as a physical object (cloud particle).

All attributes defined on a particle are specific for that particle (for example mass or position). A particle contains a set of attributes, some attributes are *generic* and applicable for multiple modules. Other attributes are *specific* and are only applicable for a single module.

`__add__` (*particles*)

Returns a particle subset, composed of the given particle(s) and this particle. Attribute values are not stored by the subset. The subset provides a view on the particles.

Parameters `particles` – particle(s) to be added to self.

```
>>> particles = Particles(2)
>>> particle1 = particles[0]
>>> particle1.x = 1.0 | units.m
>>> particle2 = particles[1]
>>> particle2.x = 2.0 | units.m
>>> new_set = particle1 + particle2
>>> new_set
<amuse.datamodel.particles.ParticlesSubset object at 0x...>
>>> print len(new_set)
2
>>> print new_set.x
[1.0, 2.0] m
```

`__sub__` (*particles*)

Raises an exception: cannot subtract particle(s) from a particle.

`as_set` ()

Returns a subset view on the set containing this particle. The subset view includes this particle and no other particles.

```
>>> particles = Particles(2)
>>> particles.x = [1.0, 2.0] | units.m
>>> particle2 = particles[1]
>>> print particle2.x
2.0 m
>>> particles_with_one_particle = particle2.as_set()
>>> len(particles_with_one_particle)
1
>>> print particles_with_one_particle.x
[2.0] m
```

9.3.2 Methods to retrieve physical properties of the particles set

`amuse.datamodel.particle_attributes.center_of_mass` (*particles*)

Returns the center of mass of the particles set. The center of mass is defined as the average of the positions of the particles, weighted by their masses.

```
>>> from amuse.datamodel import Particles
>>> particles = Particles(2)
>>> particles.x = [-1.0, 1.0] | units.m
>>> particles.y = [0.0, 0.0] | units.m
>>> particles.z = [0.0, 0.0] | units.m
>>> particles.mass = [1.0, 1.0] | units.kg
>>> particles.center_of_mass()
quantity<[0.0, 0.0, 0.0] m>
```

`amuse.datamodel.particle_attributes.center_of_mass_velocity` (*particles*)

Returns the center of mass velocity of the particles set. The center of mass velocity is defined as the average

of the velocities of the particles, weighted by their masses.

```
>>> from amuse.datamodel import Particles
>>> particles = Particles(2)
>>> particles.vx = [-1.0, 1.0] | units.ms
>>> particles.vy = [0.0, 0.0] | units.ms
>>> particles.vz = [0.0, 0.0] | units.ms
>>> particles.mass = [1.0, 1.0] | units.kg
>>> particles.center_of_mass_velocity()
quantity<[0.0, 0.0, 0.0] m * s**-1>
```

`amuse.datamodel.particle_attributes.kinetic_energy` (*particles*)
Returns the total kinetic energy of the particles in the particles set.

```
>>> from amuse.datamodel import Particles
>>> particles = Particles(2)
>>> particles.vx = [-1.0, 1.0] | units.ms
>>> particles.vy = [0.0, 0.0] | units.ms
>>> particles.vz = [0.0, 0.0] | units.ms
>>> particles.mass = [1.0, 1.0] | units.kg
>>> particles.kinetic_energy()
quantity<1.0 m**2 * kg * s**-2>
```

`amuse.datamodel.particle_attributes.potential_energy` (*particles*, *smoothing_length_squared=quantity<zero>*, *G=quantity<6.67428e-11 m**3 * kg**-1 * s**-2>*)

Returns the total potential energy of the particles in the particles set.

Parameters

- **smoothing_length_squared** – the smoothing length is added to every distance.
- **G** – gravitational constant, need to be changed for particles in different units systems

```
>>> from amuse.datamodel import Particles
>>> particles = Particles(2)
>>> particles.x = [0.0, 1.0] | units.m
>>> particles.y = [0.0, 0.0] | units.m
>>> particles.z = [0.0, 0.0] | units.m
>>> particles.mass = [1.0, 1.0] | units.kg
>>> particles.potential_energy()
quantity<-6.67428e-11 m**2 * kg * s**-2>
```

`amuse.datamodel.particle_attributes.particle_specific_kinetic_energy` (*set*, *particle*)
Returns the specific kinetic energy of a particle.

```
>>> from amuse.datamodel import Particles
>>> particles = Particles(2)
>>> particles.vx = [0.0, 1.0] | units.ms
>>> particles.vy = [0.0, 0.0] | units.ms
>>> particles.vz = [0.0, 0.0] | units.ms
>>> particles.mass = [1.0, 1.0] | units.kg
>>> particles[1].specific_kinetic_energy()
quantity<0.5 m**2 * s**-2>
```

`amuse.datamodel.particle_attributes.particle_potential` (*set*, *particle*, *smoothing_length_squared=quantity<zero>*, *gravitationalConstant=quantity<6.67428e-11 m**3 * kg**-1 * s**-2>*)

Returns the potential energy of a particle.

```
>>> from amuse.datamodel import Particles
>>> particles = Particles(2)
>>> particles.x = [0.0, 1.0] | units.m
>>> particles.y = [0.0, 0.0] | units.m
>>> particles.z = [0.0, 0.0] | units.m
>>> particles.mass = [1.0, 1.0] | units.kg
>>> particles[1].potential()
quantity<-6.67428e-11 m**2 * s**-2>
```


PARTICLE ATTRIBUTES

Particle attributes are defined in the `amuse.datamodel.particle_attributes`. These attributes can be accessed in two ways, on the particle(s) or as a function imported from the module. When accessed on the particles, the first parameter (usually called `'particles'`) must not be given.

To access these functions directly do:

```
from amuse.datamodel.particle_attributes import *
from amuse.lab import new_plummer_sphere
```

```
particles = new_plummer_sphere(100)
print kinetic_energy(particles)
```

To access these functions as an attribute do:

```
from amuse.lab import new_plummer_sphere
```

```
particles = new_plummer_sphere(100)
```

```
print particles.kinetic_energy()
```

```
amuse.datamodel.particle_attributes.LagrangianRadii (stars, cm=None, mf=[0.01,
0.02, 0.05, 0.1, 0.2, 0.5, 0.75,
0.9, 1])
```

Calculate lagrangian radii. Output is radii, mass fraction

```
>>> import numpy
>>> from amuse.ic.plummer import new_plummer_sphere
>>> numpy.random.seed(1234)
>>> parts=new_plummer_sphere(100)
>>> lr,mf=parts.LagrangianRadii()
>>> print lr[5]
0.856966667972 length
```

```
amuse.datamodel.particle_attributes.center_of_mass (particles)
```

Returns the center of mass of the particles set. The center of mass is defined as the average of the positions of the particles, weighted by their masses.

```
>>> from amuse.datamodel import Particles
>>> particles = Particles(2)
>>> particles.x = [-1.0, 1.0] | units.m
>>> particles.y = [0.0, 0.0] | units.m
>>> particles.z = [0.0, 0.0] | units.m
>>> particles.mass = [1.0, 1.0] | units.kg
>>> particles.center_of_mass()
quantity<[0.0, 0.0, 0.0] m>
```

```
amuse.datamodel.particle_attributes.center_of_mass_velocity (particles)
```

Returns the center of mass velocity of the particles set. The center of mass velocity is defined as the average of the velocities of the particles, weighted by their masses.

```
>>> from amuse.datamodel import Particles
>>> particles = Particles(2)
>>> particles.vx = [-1.0, 1.0] | units.ms
>>> particles.vy = [0.0, 0.0] | units.ms
>>> particles.vz = [0.0, 0.0] | units.ms
>>> particles.mass = [1.0, 1.0] | units.kg
>>> particles.center_of_mass_velocity()
quantity<[0.0, 0.0, 0.0] m * s**-1>
```

`amuse.datamodel.particle_attributes.densitycentre_coreradius_coredens` (*particles*)
calculate position of the density centre, coreradius and coredensity

```
>>> import numpy
>>> from amuse.ic.plummer import new_plummer_sphere
>>> numpy.random.seed(1234)
>>> particles=new_plummer_sphere(100)
>>> pos,coreradius,coredens=particles.densitycentre_coreradius_coredens()
>>> print coreradius
0.286582946447 length
```

`amuse.datamodel.particle_attributes.distances_squared` (*particles, field_particles*)
Returns the total potential energy of the particles in the particles set.

Parameters `field_particles` – the external field consists of these (i.e. potential energy is calculated relative to the field particles)

```
>>> from amuse.datamodel import Particles
>>> field_particles = Particles(2)
>>> field_particles.x = [0.0, 2.0] | units.m
>>> field_particles.y = [0.0, 0.0] | units.m
>>> field_particles.z = [0.0, 0.0] | units.m
>>> particles = Particles(3)
>>> particles.x = [1.0, 3.0, 4] | units.m
>>> particles.y = [0.0, 0.0] | units.m
>>> particles.z = [0.0, 0.0] | units.m
>>> distances_squared(particles, field_particles)
quantity<[[ 1.  1.], [ 9.  1.], [ 16.  4.]] m**2>
```

`amuse.datamodel.particle_attributes.get_binaries` (*particles, hardness=10, G=quantity<6.67428e-11 m**3 * kg**-1 * s**-2>*)

returns the binaries in a particleset. binaries are selected according to a hardness criterion [hardness=10]
This function returns the binaries as a list of i,j particles. Triple detection is not done.

```
>>> from amuse import datamodel
>>> m = [1,1,1] | units.MSun
>>> x = [-1,1,0] | units.AU
>>> y = [0,0,1000] | units.AU
>>> z = [0,0,0] | units.AU
>>> vx = [0,0,0] | units.kms
>>> vy = [1.,-1.,0] | units.kms
>>> vz = [0,0,0] | units.kms
>>> particles = datamodel.create_particle_set( mass=m,x=x,y=y,z=z,vx=vx,vy=vy,vz=vz )
>>> binaries = particles.binaries()
>>> print len(binaries)
1
```

`amuse.datamodel.particle_attributes.kinetic_energy` (*particles*)
Returns the total kinetic energy of the particles in the particles set.

```
>>> from amuse.datamodel import Particles
>>> particles = Particles(2)
>>> particles.vx = [-1.0, 1.0] | units.ms
>>> particles.vy = [0.0, 0.0] | units.ms
```

```
>>> particles.vz = [0.0, 0.0] | units.ms
>>> particles.mass = [1.0, 1.0] | units.kg
>>> particles.kinetic_energy()
quantity<1.0 m**2 * kg * s**-2>
```

`amuse.datamodel.particle_attributes.move_to_center` (*particles*)

Move the particle positions to the center of mass and move the particle velocity to the center of mass velocity.

Implemented as:

```
particles.position -= particles.center_of_mass()
particles.velocity -= particles.center_of_mass_velocity()
```

`amuse.datamodel.particle_attributes.particle_potential` (*set*, *particle*, *smoothing_length_squared=quantity<zero>*, *gravitationalConstant=quantity<6.67428e-11 m**3 * kg**-1 * s**-2>*)

Returns the potential energy of a particle.

```
>>> from amuse.datamodel import Particles
>>> particles = Particles(2)
>>> particles.x = [0.0, 1.0] | units.m
>>> particles.y = [0.0, 0.0] | units.m
>>> particles.z = [0.0, 0.0] | units.m
>>> particles.mass = [1.0, 1.0] | units.kg
>>> particles[1].potential()
quantity<-6.67428e-11 m**2 * s**-2>
```

`amuse.datamodel.particle_attributes.particle_specific_kinetic_energy` (*set*, *particle*)

Returns the specific kinetic energy of a particle.

```
>>> from amuse.datamodel import Particles
>>> particles = Particles(2)
>>> particles.vx = [0.0, 1.0] | units.ms
>>> particles.vy = [0.0, 0.0] | units.ms
>>> particles.vz = [0.0, 0.0] | units.ms
>>> particles.mass = [1.0, 1.0] | units.kg
>>> particles[1].specific_kinetic_energy()
quantity<0.5 m**2 * s**-2>
```

`amuse.datamodel.particle_attributes.particleset_potential` (*particles*, *smoothing_length_squared=quantity<zero>*, *G=quantity<6.67428e-11 m**3 * kg**-1 * s**-2>*)

Returns the potential of the particles in the particles set.

Parameters

- **smoothing_length_squared** – the smoothing length is added to every distance.
- **G** – gravitational constant, need to be changed for particles in different units systems

```
>>> from amuse.datamodel import Particles
>>> particles = Particles(2)
>>> particles.x = [0.0, 1.0] | units.m
>>> particles.y = [0.0, 0.0] | units.m
>>> particles.z = [0.0, 0.0] | units.m
>>> particles.mass = [1.0, 1.0] | units.kg
```

```
>>> particles.potential()
quantity<[-6.67428e-11, -6.67428e-11] m**2 * s**-2>
```

```
amuse.datamodel.particle_attributes.potential_energy(particles, smoothing_length_squared=quantity<zero>,
                                                    G=quantity<6.67428e-11 m**3 * kg**-1 * s**-2>)
```

Returns the total potential energy of the particles in the particles set.

Parameters

- **smoothing_length_squared** – the smoothing length is added to every distance.
- **G** – gravitational constant, need to be changed for particles in different units systems

```
>>> from amuse.datamodel import Particles
>>> particles = Particles(2)
>>> particles.x = [0.0, 1.0] | units.m
>>> particles.y = [0.0, 0.0] | units.m
>>> particles.z = [0.0, 0.0] | units.m
>>> particles.mass = [1.0, 1.0] | units.kg
>>> particles.potential_energy()
quantity<-6.67428e-11 m**2 * kg * s**-2>
```

```
amuse.datamodel.particle_attributes.potential_energy_in_field(particles, field_particles,
                                                             smoothing_length_squared=quantity<zero>,
                                                             G=quantity<6.67428e-11 m**3 * kg**-1 * s**-2>)
```

Returns the total potential energy of the particles in the particles set.

Parameters

- **field_particles** – the external field consists of these (i.e. potential energy is calculated relative to the field particles)
- **smoothing_length_squared** – the smoothing length is added to every distance.
- **G** – gravitational constant, need to be changed for particles in different units systems

```
>>> from amuse.datamodel import Particles
>>> field_particles = Particles(2)
>>> field_particles.x = [0.0, 2.0] | units.m
>>> field_particles.y = [0.0, 0.0] | units.m
>>> field_particles.z = [0.0, 0.0] | units.m
>>> field_particles.mass = [1.0, 1.0] | units.kg
>>> particles = Particles(2)
>>> particles.x = [1.0, 3.0] | units.m
>>> particles.y = [0.0, 0.0] | units.m
>>> particles.z = [0.0, 0.0] | units.m
>>> particles.mass = [1.0, 1.0] | units.kg
>>> particles.potential_energy_in_field(field_particles)
quantity<-2.22476e-10 m**2 * kg * s**-2>
```

```
amuse.datamodel.particle_attributes.scale_to_standard(particles, convert_nbody=None,
                                                    smoothing_length_squared=quantity<zero>)
```

Scale the particles to a standard NBODY model with **total mass=1**, **kinetic energy=0.25** and **potential energy=0.5** (or **virial radius=1.0**)

Parameters

- **convert_nbody** – the scaling is in nbody units, when the particles are in si units a `convert_nbody` is needed
- **smoothing_length_squared** – needed for calculating the potential energy correctly.

`amuse.datamodel.particle_attributes.specific_kinetic_energy` (*particles*)
Returns the specific kinetic energy of a particle.

```
>>> from amuse.datamodel import Particles
>>> particles = Particles(2)
>>> particles.vx = [1.0, 1.0] | units.ms
>>> particles.vy = [0.0, 0.0] | units.ms
>>> particles.vz = [0.0, 0.0] | units.ms
>>> particles.mass = [1.0, 1.0] | units.kg
>>> particles.specific_kinetic_energy()
quantity<[0.5, 0.5] m**2 * s**-2>
```

`amuse.datamodel.particle_attributes.thermal_energy` (*particles*)
Returns the total internal energy of the (gas) particles in the particles set.

```
>>> from amuse.datamodel import Particles
>>> particles = Particles(2)
>>> particles.u = [0.5, 0.5] | units.ms**2
>>> particles.mass = [1.0, 1.0] | units.kg
>>> particles.thermal_energy()
quantity<1.0 m**2 * kg * s**-2>
```

`amuse.datamodel.particle_attributes.total_angular_momentum` (*particles*)
Returns the total angular momentum of the particles set.

```
>>> from amuse.datamodel import Particles
>>> particles = Particles(2)
>>> particles.x = [-1.0, 1.0] | units.m
>>> particles.y = [0.0, 0.0] | units.m
>>> particles.z = [0.0, 0.0] | units.m
>>> particles.vx = [0.0, 0.0] | units.ms
>>> particles.vy = [-1.0, 1.0] | units.ms
>>> particles.vz = [0.0, 0.0] | units.ms
>>> particles.mass = [1.0, .5] | units.kg
>>> particles.total_angular_momentum()
quantity<[0.0, 0.0, 1.5] m**2 * kg * s**-1>
```

`amuse.datamodel.particle_attributes.total_mass` (*particles*)
Returns the total mass of the particles set.

```
>>> from amuse.datamodel import Particles
>>> particles = Particles(3)
>>> particles.mass = [1.0, 2.0, 3.0] | units.kg
>>> particles.total_mass()
quantity<6.0 kg>
```

`amuse.datamodel.particle_attributes.total_momentum` (*particles*)
Returns the total momentum of the particles set.

```
>>> from amuse.datamodel import Particles
>>> particles = Particles(2)
>>> particles.vx = [-1.0, 1.0] | units.ms
>>> particles.vy = [0.0, 0.0] | units.ms
>>> particles.vz = [0.0, 0.0] | units.ms
>>> particles.mass = [1.0, 1.0] | units.kg
>>> particles.total_momentum()
quantity<[0.0, 0.0, 0.0] m * kg * s**-1>
```

`amuse.datamodel.particle_attributes.velocity_diff_squared` (*particles*,
field_particles)

Returns the total potential energy of the particles in the particles set.

Parameters `field_particles` – the external field consists of these (i.e. potential energy is calculated relative to the field particles)

```
>>> from amuse.datamodel import Particles
>>> field_particles = Particles(2)
>>> field_particles.vx = [0.0, 2.0] | units.m
>>> field_particles.vy = [0.0, 0.0] | units.m
>>> field_particles.vz = [0.0, 0.0] | units.m
>>> particles = Particles(3)
>>> particles.vx = [1.0, 3.0, 4] | units.m
>>> particles.vy = [0.0, 0.0] | units.m
>>> particles.vz = [0.0, 0.0] | units.m
>>> velocity_diff_squared(particles, field_particles)
quantity<[[ 1.  1.], [ 9.  1.], [ 16.  4.]] m**2>
```

`amuse.datamodel.particle_attributes.virial_radius` (*particles*)

Returns the virial radius of the particles set. The virial radius is the inverse of the average inverse distance between particles, weighted by their masses.

```
>>> from amuse.datamodel import Particles
>>> particles = Particles(2)
>>> particles.x = [-1.0, 1.0] | units.m
>>> particles.y = [0.0, 0.0] | units.m
>>> particles.z = [0.0, 0.0] | units.m
>>> particles.mass = [1.0, 1.0] | units.kg
>>> particles.virial_radius()
quantity<4.0 m>
```

FROM CODES TO DATA

11.1 Introduction

The framework contains two distinct levels on which interaction with the codes takes place. The first level interacts directly with the code and is implemented as a set of functions on a class. The second level is built on top of the first level and abstracts the function calls to handling objects and datasets. Often multiple function-calls on the first level can be abstracted to a single statement (assignment or query) on the second level.

11.2 The first level

The first level is a direct interface to the code, in this chapter the kind of functions supported by the first level interface code will be briefly described. All functions that can be defined on the first level fall in two categories, those that handle scalars (a single value for each parameter) or those that handle 1-D vectors (a list of values for each parameter). For functions that handle 1-D vectors each vector must be of the same length.

Note: Not every function will fit in the two categories, but it is usually possible to rewrite a function or create an interfacing function that do fit into one of the two categories. Supporting only these two categories keeps the communication layer simpler and allows for some optimizations in the communication between python and C/Fortran codes.

An example of using the first level with scalars and vectors:

```
from amuse.community.codes.athena.interface import AthenaInterface

# create an instance of the code (will start an application
# in the background to handle all requests)
hydro = AthenaInterface()

# set parameters needed by the code
# these are functions handling one scalar input
# parameter
hydro.set_gamma(1.6666666666666667)
hydro.set_courant_friedrichs_levy_number(0.8)

# define a grid having 5 cells in all directions and
# with the total length of the grid in each direction
# is 1.0
# this is a function handling multiple
# scalar input parameters
hydro.setup_mesh(5, 5, 5, 1.0, 1.0, 1.0)

# setup boundary conditions
# (can be periodic, reflective, outflow)
hydro.set_boundary(
```

```
"periodic", "periodic",
"periodic", "periodic",
"periodic", "periodic"
)

# let the code do some work
# (athena will allocate the grid)
# this is a function handling no
# scalar input parameters and having
# 1 scalar output parameter
hydro.commit_parameters()

# lets print the center position
# of one grid point
print hydro.get_position_of_index(1,2,3)

# all calls so far have been to functions handling scalar values
# the next calls will be to functions handling vectors of values

# lets print the center positions
# of all grid points on one line
print hydro.get_position_of_index(range(0,5), [0] * 5, [0] * 5)
```

In the previous example we used functions with scalar parameters and vector parameters. The functions handling vectors often can also handle scalars, the framework will take care of the necessary conversions.

All first level functions are not actual python functions, these functions are instances of a special Python class that implements function call handling. To continue our example:

```
# let's take a look at the kind of functions
# on the first level
print hydro.get_position_of_index

# you can ask the specification of a
# first level function
print hydro.get_position_of_index.specification
```

11.2.1 Adding units

The first step after defining a first level function is to specify the units of the in- and output-parameters of the first level function.

SETS AND GRIDS IN CODES

This module defines the classes needed to map functions defined by the codes into particle sets and grids.

The attribute values of Particles or Gridpoints are stored in Particle Sets or Grids. These sets or grids manage:

1. The storage allocation (deletion and removal of particles)
2. The attribute access (getting or setting the value(s) of attribute(s))
3. Queries or selections of particles (selection of subsets of particles)

All 3 functions can be provided by a code. The classes in this module provide a mapping from the functions in a code to the datamodel used in AMUSE.

When a code manages a particular set all the data of that set is stored in the memory space of that code. The code needs to provide functions to access the data in the set.

Note: Most codes already implement a particle set or a grid. The only extra requirement for AMUSE is to provide functions to access this set. When a code does not have any knowledge of sets or grids, the management will take place in AMUSE and only some data transfer code is needed

All incode storage is build on mapping attributes to functions. These mappings are provided by a number of helper classes:

setter/getter

ParticleGetAttributesMethod Given particle indices or gridpoints (i,j,k) return a vector quantity for each attribute

ParticleSetAttributesMethod Send values to the code given particle indices or gridpoints (i,j,k) and a vector quantities for each attribute.

new/delete

NewParticleMethod Given vector quantities for attributes return the indices of newly allocated particles

function

ParticleMethod Given particle indices or gridpoints (i,j,k) and optional arguments return one or more vector quantities

selection

ParticleSpecificSelectMethod Given a particle return a subset of particles. For links between particles (nearest neighbor, subparticle)

ParticleQueryMethod Retrieve indices from the code and return a subset of particles. For selection of a limited number of particles by the code (get the escaper)

ParticleSetSelectSubsetMethod Like ParticleQueryMethod but can handle larger subsets of particles, the code can provide a special function the return the number of particles in the set.

The InCode storage system is based on a number of classes:

AbstractInCodeAttributeStorage Handle attribute set/get functionality but no particle or grid management

InCodeAttributeStorage Subclass of AbstractInCodeAttributeStorage, manages particles

InCodeGridAttributeStorage Subclass of AbstractInCodeAttributeStorage, manages grids

```
class amuse.datamodel.incode_storage.AbstractInCodeAttributeStorage (code_interface,  
                                                                    setters,  
                                                                    getters,  
                                                                    ex-  
                                                                    tra_keyword_arguments_for_getters)
```

Bases: `amuse.datamodel.base.AttributeStorage`

Abstract base storage for incode attribute storage. It provides functions to handle getters and setters of attributes but not for creating or deleting of particles as this differs between grids and particle sets.

`get_defined_attribute_names ()`

`select_getters_for (attributes)`

`select_setters_for (attributes)`

```
class amuse.datamodel.incode_storage.InCodeAttributeStorage (code_interface,  
                                                                    new_particle_method,  
                                                                    delete_particle_method,  
                                                                    num-  
                                                                    ber_of_particles_method,  
                                                                    setters, getters,  
                                                                    name_of_the_index)
```

Bases: `amuse.datamodel.incode_storage.AbstractInCodeAttributeStorage`

Manages sets of particles stored in codes.

Maps indices returned by the code to keys defined in AMUSE.

`add_particles_to_store (keys, attributes=[], values=[])`

`can_extend_attributes ()`

`get_all_indices_in_store ()`

`get_all_keys_in_store ()`

`get_indices_of (keys)`

`get_key_indices_of (keys)`

`get_values_in_store (keys, attributes, by_key=True)`

`has_key_in_store (key)`

`remove_particles_from_store (keys)`

`set_values_in_store (keys, attributes, values, by_key=True)`

```
class amuse.datamodel.incode_storage.InCodeGridAttributeStorage (code_interface,  
                                                                    get_range_method,  
                                                                    setters,  
                                                                    getters, ex-  
                                                                    tra_keyword_arguments_for_getters)
```

Bases: `amuse.datamodel.incode_storage.AbstractInCodeAttributeStorage`

Manages grids stored in codes. Grids are currently assumed to be three dimensional.

`add_particles_to_store (keys, attributes=[], quantities=[])`

`copy ()`

`get_all_keys_in_store ()`

`get_defined_attribute_names ()`

get_values_in_store (*indices, attributes*)

has_key_in_store (*key*)

remove_particles_from_store (*keys*)

set_values_in_store (*indices, attributes, quantities*)

storage_shape ()

class `amuse.datamodel.incode_storage.NewParticleMethod` (*method, at-tribute_names=None*)

Bases: `amuse.datamodel.incode_storage.ParticleSetAttributesMethod`

Instances wrap a method to create particles. The method may take attributes values to set initial values on the created particles.

The new particle functions work a lot like the set attribute methods, only the new particle function is supposed to return an array of the indices of the created particles.

```
indices = instance.new_particle(x, y, z)
```

add_entities (*attributes, values*)

class `amuse.datamodel.incode_storage.ParticleGetAttributesMethod` (*method, at-tribute_names=None*)

Bases: `amuse.datamodel.incode_storage.ParticleMappingMethod`

Instances wrap other methods and provide mappings from attribute names to results.

Simple attribute getter methods take an array of indices and return a tuple with arrays of result values.

```
x, y, z = instance.get_xyz(indices)
```

Instances of this class make it possible to access the return values by their attribute names.

For this it employs two strategies:

1. It uses the provided array of names and maps each name to the positional output.
2. If no array of names is provided it asks the wrapped method for all the names of the output parameters (this scheme only works for legacy functions or for wrapped legacy functions)

attribute_names

check_arguments (*storage, attributes_to_return, *indices*)

convert_return_value (*return_value, storage, attributes_to_return*)

get_attribute_values (*storage, attributes_to_return, *indices*)

class `amuse.datamodel.incode_storage.ParticleGetIndexMethod`

Bases: `object`

Instances return the index of a particle in the code

attribute_names

get_attribute_values (*storage, attributes_to_return, *indices*)

class `amuse.datamodel.incode_storage.ParticleMappingMethod` (*method, at-tribute_names=None*)

Bases: `amuse.support.methods.AbstractCodeMethodWrapper`

name_of_the_indexing_parameter

class `amuse.datamodel.incode_storage.ParticleMethod` (*method, public_name=None*)

Bases: `amuse.support.methods.AbstractCodeMethodWrapper`

Instances wrap a function that returns quantities given particle indices and optional arguments. Instances have a lot in common with attribute getters, but can take extra arguments.

```
pressure = instance.get_pressure(index, gamma)
```

```
apply_on_all (particles, *list_arguments, **keyword_arguments)
```

```
apply_on_one (set, particle, *list_arguments, **keyword_arguments)
```

```
class amuse.datamodel.incode_storage.ParticleQueryMethod (method, names=(),  
                                                    public_name=None,  
                                                    query_superset=False)
```

Bases: object

Instances wrap a function that can take one or more arguments and returns an index (or a list of indices, if the arguments are lists). This method is most useful to select one particle form all particles in the set

```
index = instance.get_escaper()
```

The idex or indices are converted to a particle subset.

```
apply_for_superset (particles, *args, **kwargs)
```

```
apply_normal (particles, *args, **kwargs)
```

```
class amuse.datamodel.incode_storage.ParticleSetAttributesMethod (method, at-  
                                                                tribute_names=None)
```

Bases: `amuse.datamodel.incode_storage.ParticleMappingMethod`

Instances wrap other methods and provide mappings from attribute names to input parameters.

Simple attribute setter methods take an array of indices and one or more arrays of new values.

```
instance.set_xyz(indices, x, y, z)
```

Instances of this class make it possible to access the possitional parameters with attribute names.

Note: the index argument is assumed to always come first!

For this it employs two strategies:

- 1.It uses the provided array of names and maps each name to the positional output.
- 2.If no array of names is provided it asks the wrapped method for all the names of the input parameters (this scheme works for legacy functions and sometimes for python native functions (if they have named arguments))

attribute_names

```
convert_attributes_and_values_to_list_and_keyword_arguments (attributes,  
                                                                values)
```

names_to_index

optional_attribute_names

```
set_attribute_values (storage, attributes, values, *indices)
```

```
class amuse.datamodel.incode_storage.ParticleSetSelectSubsetMethod (method,  
                                                                set_query_arguments_method=No  
                                                                get_number_of_particles_in_set_r  
                                                                pub-  
                                                                lic_name=None)
```

Bases: object

Generic method to query and retrieve particles from the set. This selection can have up to tree stages:

- 1.start the query given a number of optional arguments
- 2.get the number of selected particles
- 3.get the index of each particle

The pseudo-code for this selection is:

```
set_selection_criteria(r = 10.0 | units.m)
n = get_number_of_selected_particles()
for i in range(n):
    particle_index = get_index_of_selected_particle(i)
```

The first and second step are optional. If no number of particles method is provided the class assumes the selection only returns 1 particle.

Generalisation of ParticleQueryMethod

apply_on_all (*particles*, **list_arguments*, ***keyword_arguments*)

```
class amuse.datamodel.incode_storage.ParticleSpecificSelectMethod (method,
                                                                    names=(),
                                                                    pub-
                                                                    lic_name=None)
```

Bases: object

Instances wrap a function that can take a particle index and returns one or more indices (but a limited and fixed number of indices). This method is most useful to return links between particles (subparticles or nearest neighbors)

```
output_index = instance.get_nearest_neighbord(input_index)
```

The idex or indices are converted to a particle subset.

apply_on_all (*particles*)

apply_on_one (*set*, *particle*)

```
class amuse.datamodel.incode_storage.ParticleSpecificSelectSubsetMethod (method,
                                                                           get_number_of_particles_
                                                                           pub-
                                                                           lic_name=None)
```

Bases: object

Instances wrap a function that can take a particle index, plus a list offset and returns one index. This method is most useful to return links between particles (subparticles or nearest neighbors). Instances also need a function to get the number of links.

```
output_index = instance.get_nearest_neighbors(index_of_the_particle, input_index)
```

The index or indices are converted to a particle subset.

apply_on_all (*particles*)

apply_on_one (*set*, *particle*)

```
class amuse.datamodel.incode_storage.ParticlesAddedUpdateMethod (get_number_of_particles_added_meth
                                                                    get_id_of_added_particles_method=N
```

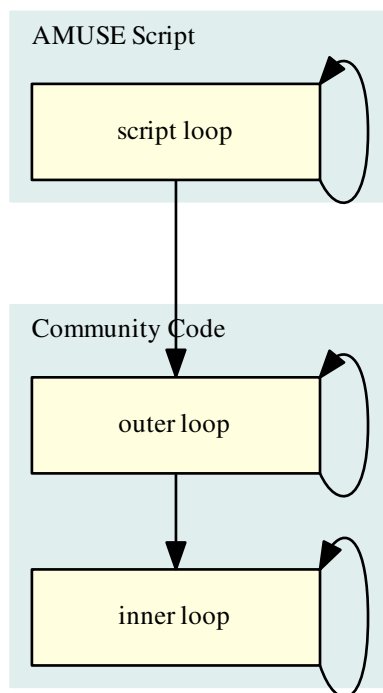
Bases: object

apply_on_all (*particles*, **list_arguments*, ***keyword_arguments*)

STOPPING CONDITIONS

13.1 Introduction

Codes in AMUSE evolve their models in three loops, the “inner”, the “outer” loop, and the “script” loop. The “inner” loop is controlled by the code and evolves the model in a sufficiently small steps to limit errors and be able to simulate important physics. The “outer” loop invokes the “inner” loop until a condition is met specified by the AMUSE script. The AMUSE script (or the “script” loop) interacts with the “outer” loop.



In AMUSE the outer loop is always limited by the model time. Every `evolve_model()` call has one argument, the model time. When the simulation is about to go beyond this time (or is at this time), control is returned to the caller. This process is shown in the following code example.

```
// example outer loop
void outer_loop(double end_time)
{
```

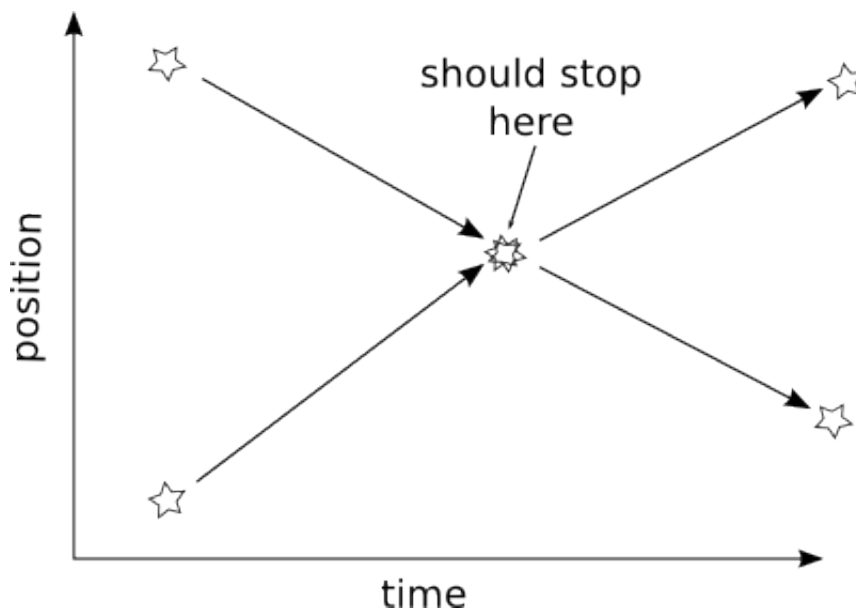
```
while(model_time < end_time)
{
    ...
    inner_loop()
    ...
    model_time += delta_time_for_this_step
}
return
}
```

The model time is often enough to control the loop but some situations call for an earlier break in the loop when the model time has not reached the end time. To be able to break the outer loop before the end time is reached, the framework supports **stopping conditions**. The following code example shows how these conditions might work in a C code.

```
// example outer loop with stopping conditions
void outer_loop(double end_time)
{
    int special_condition_was_met = 0;

    while(model_time < end_time)
    {
        ...
        inner_loop()
        ...
        check_if_special_conditions_are_met();
        ...
        model_time += delta_time_for_this_step
        ...
        if(special_condition_was_met) {
            break;
        }
    }
    return
}
```

For example, a code might be able to detect a collision of two stars and the user wants to handle this event. If the code would proceed to the end time, the collision is long over and the user has no way to know about the collision and cannot handle it.



The **stopping conditions** framework in AMUSE provides functionality for using and implementing stopping conditions. This framework uses simple tables in the codes and translates these tables to objects in the python

scripts. In the next section the use of this framework is described. In the final section the implementation of the framework in a community code is described.

13.2 Using stopping conditions

All interaction with the *stopping conditions* of a *code* is via the `stopping_conditions` attribute of a `CodeInterface` class. When the `stopping_conditions` attribute is queried it returns a `StoppingConditions` object. All `StoppingConditions` objects have an attribute for each stopping condition supported by AMUSE. The attributes have the following names:

collision_detection The outer loop breaks when two stars connect

pair_detection The outer loop breaks when two stars pair, possibly creating a binary star.

escaper_detection The outer loop breaks when a star escapes the simulation (is no longer gravitationally bound to the other stars)

timeout_detection The outer loop breaks when a certain amount of computer (wall-clock) time has elapsed.

number_of_steps_detection The outer loop breaks when a certain amount of steps (iterations) is reached.

Note: `CodeInterface` objects provide a object oriented interface to each code.

All `StoppingConditions` objects provide a string representation describing the supported conditions and their states.

```
>>> from amuse.lab import *
>>> code = Hermite()
>>> print code.stopping_conditions
Stopping conditions of a 'Hermite' object
* supported conditions: collision_detection, pair_detection, timeout_detection
* enabled conditions: none
* set conditions: none
>>> code.stop()
```

For every attribute the user can determine if it is supported, if it was enabled or if it was hit during the `evolve()` loop.

```
>>> from amuse.lab import *
>>> code = Hermite()
>>> print code.stopping_conditions.collision_detection.is_supported()
True
>>> print code.stopping_conditions.collision_detection.is_enabled()
False
>>> print code.stopping_conditions.collision_detection.is_set()
False
>>> code.stop()
```

When a stopping condition was hit during the inner or outer loop evaluation the user can query which particles were involved. The particles involved are stored in columns. For pair wise detections (collection and pair) the condition provides two columns. The first column contains all the first particles of the pairs and the second column contains all the second particles of the pairs. These columns can be queried with the `particles()` method on a stopping condition attribute. This method takes one argument, the column index and returns a particle subset with the involved particles.

Pair	particles(0)	particles(1)
1 + 2	1	2
5 + 11	5	11
12 + 10	12	10

```

>>> from amuse.lab import *
>>> code = Hermite()
>>> sc = code.stopping_conditions.collusion_detection
>>> sc.enable()
>>> code.evolve_model(0.1 | nbody_system.time)
>>> print sc.is_set()
True
>>> print sc.particles(0)
...
>>> pair = sc.particles(0)[0], sc.particles(1)[0]
>>> code.stop()

```

13.3 Implementing stopping conditions

All codes supporting stopping conditions must implement the `StoppingConditionInterface` interface. This can be easily done by inheriting from the `StoppingConditionInterface` and implementing all the functions defined in the interface:

```

class MyCodeInterface(CodeInterface, StoppingConditionInterface):
    ...

```

This `StoppingConditionInterface` interface models the interface to a simple data model, best described as two tables:

Defined stopping conditions table This table list the types of stopping conditions and if these are supported by the code, and if so, if these are enabled by the user.

type int	supported int read only	enabled int read/write
0	0	0
1	1	0
2	0	0
3	1	0

type The type of the stopping condition, integer between 0 and N. defined by the framework

supported 1 if the stopping condition is supported, 0 otherwise. Needs to be set by the implementor

enabled 1 if enabled by the user, 0 otherwise. Implementor must provide a mechanism to set and unset this field.

Set stopping conditions table This table has a row for each stopping condition set during the outer and inner loop evaluations. Every row lists the type of condition set and which particles were involved (if any). The id of the last particle in the list must be -1.

type int	particle[0] int	particle(1) int	... int	particle(n) int
1	1	2	-1	-1
1	10	12	-1	-1
3	-1	...		
1	11	20	...	
...				
3	-1			

type The type of the stopping condition, integer between 0 and N. defined by the framework. Set by the code

particle(0..n) Index of the particle involved in the stopping condition. The index of the last particle must be -1

```

class amuse.support.codes.stopping_conditions.StoppingConditionInterface

```

disable_stopping_condition

Will disable the stopping if it is supported

```
int32 disable_stopping_condition(int32 type);
```

```
FUNCTION disable_stopping_condition(type)
  INTEGER :: type
  INTEGER :: disable_stopping_condition
END FUNCTION
```

Parameters *type* (*int32*, *IN*) – The index of the stopping condition

Returns 0 - OK -1 - ERROR

enable_stopping_condition

Will enable the stopping if it is supported

```
int32 enable_stopping_condition(int32 type);
```

```
FUNCTION enable_stopping_condition(type)
  INTEGER :: type
  INTEGER :: enable_stopping_condition
END FUNCTION
```

Parameters *type* (*int32*, *IN*) – The type index of the stopping condition

Returns 0 - OK -1 - ERROR

get_number_of_stopping_conditions_set

Return the number of stopping conditions set, one condition can be set multiple times.

Stopping conditions are set when the code determines that the conditions are met. The objects or or information about the condition can be retrieved with the `get_stopping_condition_info` method.

```
int32 get_number_of_stopping_conditions_set(int32 * result);
```

```
FUNCTION get_number_of_stopping_conditions_set(result)
  INTEGER :: result
  INTEGER :: get_number_of_stopping_conditions_set
END FUNCTION
```

Parameters *result* (*int32*, *OUT*) – > 1 if any stopping condition is set

Returns 0 - OK -1 - ERROR

get_stopping_condition_info

Generic function for getting the information connected to a stopping condition. Index can be between 0 and the result of the `method:'get_number_of_stopping_conditions_set'` method.

```
int32 get_stopping_condition_info(int32 index, int32 * type,
  int32 * number_of_particles);
```

```
FUNCTION get_stopping_condition_info(index, type, number_of_particles)
  INTEGER :: index, type, number_of_particles
  INTEGER :: get_stopping_condition_info
END FUNCTION
```

Parameters

- **index** (*int32*, *IN*) – Index in the array[0,number_of_stopping_conditions_set>
- **type** (*int32*, *OUT*) – Kind of the condition, can be used to retrieve specific information
- **number_of_particles** (*int32*, *OUT*) – Number of particles that met this condition

Returns 0 - OK -1 - ERROR

get_stopping_condition_maximum_density_parameter

```
int32 get_stopping_condition_maximum_density_parameter(float64 * value);
```

```
FUNCTION get_stopping_condition_maximum_density_parameter(value)
  DOUBLE PRECISION :: value
  INTEGER :: get_stopping_condition_maximum_density_parameter
END FUNCTION
```

Parameters *value (float64, OUT)* –

Returns

get_stopping_condition_maximum_internal_energy_parameter

```
int32 get_stopping_condition_maximum_internal_energy_parameter(
  float64 * value);
```

```
FUNCTION get_stopping_condition_maximum_internal_energy_parameter(value)
  DOUBLE PRECISION :: value
  INTEGER :: get_stopping_condition_maximum_internal_energy_parameter
END FUNCTION
```

Parameters *value (float64, OUT)* –

Returns

get_stopping_condition_minimum_density_parameter

```
int32 get_stopping_condition_minimum_density_parameter(float64 * value);
```

```
FUNCTION get_stopping_condition_minimum_density_parameter(value)
  DOUBLE PRECISION :: value
  INTEGER :: get_stopping_condition_minimum_density_parameter
END FUNCTION
```

Parameters *value (float64, OUT)* –

Returns

get_stopping_condition_minimum_internal_energy_parameter

```
int32 get_stopping_condition_minimum_internal_energy_parameter(
  float64 * value);
```

```
FUNCTION get_stopping_condition_minimum_internal_energy_parameter(value)
  DOUBLE PRECISION :: value
  INTEGER :: get_stopping_condition_minimum_internal_energy_parameter
END FUNCTION
```

Parameters *value (float64, OUT)* –

Returns

get_stopping_condition_number_of_steps_parameter

Retrieve max inner loop evaluations.

```
int32 get_stopping_condition_number_of_steps_parameter(int32 * value);
```

```
FUNCTION get_stopping_condition_number_of_steps_parameter(value)
  INTEGER :: value
  INTEGER :: get_stopping_condition_number_of_steps_parameter
END FUNCTION
```

Parameters *value* (*int32*, *OUT*) – Current number of available inner loop evaluations

Returns 0 - OK -1 - ERROR

get_stopping_condition_out_of_box_parameter

Get size of box

```
int32 get_stopping_condition_out_of_box_parameter(float64 * value);
```

```
FUNCTION get_stopping_condition_out_of_box_parameter(value)
  DOUBLE PRECISION :: value
  INTEGER :: get_stopping_condition_out_of_box_parameter
END FUNCTION
```

Parameters *value* (*float64*, *OUT*) – Size of box

Returns 0 - OK -1 - Value out of range

get_stopping_condition_particle_index

For collision detection

```
int32 get_stopping_condition_particle_index(int32 index,
  int32 index_of_the_column, int32 * index_of_particle);
```

```
FUNCTION get_stopping_condition_particle_index(index, &
  index_of_the_column, index_of_particle)
  INTEGER :: index, index_of_the_column, index_of_particle
  INTEGER :: get_stopping_condition_particle_index
END FUNCTION
```

Parameters

- **index** (*int32*, *IN*) – Index in the array[0,number_of_stopping_conditions_set>
- **index_of_the_column** (*int32*, *IN*) – Column index involved in the condition (for pair collisions 0 and 1 are possible)
- **index_of_particle** (*int32*, *OUT*) – Set to the identifier of particle[index_of_the_column][index]

Returns 0 - OK -1 - ERROR

get_stopping_condition_timeout_parameter

Retrieve max computer time available (in seconds).

```
int32 get_stopping_condition_timeout_parameter(float64 * value);
```

```
FUNCTION get_stopping_condition_timeout_parameter(value)
  DOUBLE PRECISION :: value
  INTEGER :: get_stopping_condition_timeout_parameter
END FUNCTION
```

Parameters *value* (*float64*, *OUT*) – Current value of available wallclock time in seconds

Returns 0 - OK -1 - ERROR

has_stopping_condition

Return 1 if the stopping condition with the given index is supported by the code, 0 otherwise.

```
int32 has_stopping_condition(int32 type, int32 * result);
```

```
FUNCTION has_stopping_condition(type, result)
  INTEGER :: type, result
  INTEGER :: has_stopping_condition
END FUNCTION
```

Parameters

- **type** (*int32, IN*) – The type index of the stopping condition
- **result** (*int32, OUT*) – 1 if the stopping condition is supported

Returns 0 - OK -1 - ERROR

is_stopping_condition_enabled

Return 1 if the stopping condition with the given index is enabled,0 otherwise.

```
int32 is_stopping_condition_enabled(int32 type, int32 * result);
```

```
FUNCTION is_stopping_condition_enabled(type, result)
  INTEGER :: type, result
  INTEGER :: is_stopping_condition_enabled
END FUNCTION
```

Parameters

- **type** (*int32, IN*) – The index of the stopping condition
- **result** (*int32, OUT*) – 1 if the stopping condition is enabled

Returns 0 - OK -1 - ERROR

is_stopping_condition_set

Return 1 if the stopping condition with the given index is enabled,0 otherwise.

```
int32 is_stopping_condition_set(int32 type, int32 * result);
```

```
FUNCTION is_stopping_condition_set(type, result)
  INTEGER :: type, result
  INTEGER :: is_stopping_condition_set
END FUNCTION
```

Parameters

- **type** (*int32, IN*) – The index of the stopping condition
- **result** (*int32, OUT*) – 1 if the stopping condition is enabled

Returns 0 - OK -1 - ERROR

set_stopping_condition_maximum_density_parameter

```
int32 set_stopping_condition_maximum_density_parameter(float64 value);
```

```
FUNCTION set_stopping_condition_maximum_density_parameter(value)
  DOUBLE PRECISION :: value
  INTEGER :: set_stopping_condition_maximum_density_parameter
END FUNCTION
```

Parameters *value* (*float64, IN*) –

Returns**set_stopping_condition_maximum_internal_energy_parameter**

```
int32 set_stopping_condition_maximum_internal_energy_parameter(
    float64 value);
```

```
FUNCTION set_stopping_condition_maximum_internal_energy_parameter(value)
    DOUBLE PRECISION :: value
    INTEGER :: set_stopping_condition_maximum_internal_energy_parameter
END FUNCTION
```

Parameters *value* (*float64*, *IN*) –

Returns**set_stopping_condition_minimum_density_parameter**

```
int32 set_stopping_condition_minimum_density_parameter(float64 value);
```

```
FUNCTION set_stopping_condition_minimum_density_parameter(value)
    DOUBLE PRECISION :: value
    INTEGER :: set_stopping_condition_minimum_density_parameter
END FUNCTION
```

Parameters *value* (*float64*, *IN*) –

Returns**set_stopping_condition_minimum_internal_energy_parameter**

```
int32 set_stopping_condition_minimum_internal_energy_parameter(
    float64 value);
```

```
FUNCTION set_stopping_condition_minimum_internal_energy_parameter(value)
    DOUBLE PRECISION :: value
    INTEGER :: set_stopping_condition_minimum_internal_energy_parameter
END FUNCTION
```

Parameters *value* (*float64*, *IN*) –

Returns**set_stopping_condition_number_of_steps_parameter**

Set max inner loop evaluations.

```
int32 set_stopping_condition_number_of_steps_parameter(int32 value);
```

```
FUNCTION set_stopping_condition_number_of_steps_parameter(value)
    INTEGER :: value
    INTEGER :: set_stopping_condition_number_of_steps_parameter
END FUNCTION
```

Parameters *value* (*int32*, *IN*) – Available inner loop evaluations

Returns 0 - OK -1 - Value out of range

set_stopping_condition_out_of_box_parameter

Set size of box.

```
int32 set_stopping_condition_out_of_box_parameter(float64 value);
```

```
FUNCTION set_stopping_condition_out_of_box_parameter(value)
  DOUBLE PRECISION :: value
  INTEGER :: set_stopping_condition_out_of_box_parameter
END FUNCTION
```

Parameters *value* (*float64*, *IN*) – Size of box

Returns 0 - OK -1 - Value out of range

set_stopping_condition_timeout_parameter

Set max computer time available (in seconds).

```
int32 set_stopping_condition_timeout_parameter(float64 value);
```

```
FUNCTION set_stopping_condition_timeout_parameter(value)
  DOUBLE PRECISION :: value
  INTEGER :: set_stopping_condition_timeout_parameter
END FUNCTION
```

Parameters *value* (*float64*, *IN*) – Available wallclock time in seconds

Returns 0 - OK -1 - Value out of range

This interface contains a lot of functions and implementing it might be a time consuming task. To help implementing the interface, AMUSE provides a C library that does most of the table management, this library is described in the next section.

13.4 Using the stopping conditions library

The stopping conditions tables are implemented as a static C library. This library is part of the AMUSE distribution and can be found in the ``lib/stopcond`` directory.

The library implements *all* functions of the `StoppingConditionInterface` interface and provides a number of functions for the code implementer to actually set and support stopping conditions. The library does not implement any stopping condition detection routines. These have to be implemented on a per code basis. The functionality of the library is limited to managing the stopping conditions tables and the library has to be controlled by the code so that stopping conditions are set.

The library implements the defined stopping conditions table as three long integers. These integers are used as bitmaps (the state of individual bits is used to determine if a condition is true or false). The individual bits can be tested with the following macros:

COLLISION_DETECTION_BITMAP

PAIR_DETECTION_BITMAP

ESCAPER_DETECTION_BITMAP

TIMEOUT_DETECTION_BITMAP

NUMBER_OF_STEPS_BITMAP

The three variables are not accessible directly from FORTRAN codes. The stopping condition library has special functions for FORTRAN to access them.

The three global variables are:

long **enabled_conditions**

Individual bits are set when the user enables a stopping condition. Access in FORTRAN through:

```

INCLUDE "../..../lib/stopcond/stopcond.inc"
INTEGER:: is_stopping_condition_enabled
INTEGER:: is_collision_detection_enabled
INTEGER:: error
error = is_stopping_condition_enabled(COLLISION_DETECTION, is_collision_detection_enabled)

```

long **set_conditions**

Individual bits are set when the a condition is detected by the code. (the `set_stopping_condition_info()` will fill this bitmap). Access in FORTRAN:

```

INCLUDE "../..../lib/stopcond/stopcond.inc"
INTEGER:: is_stopping_condition_set
INTEGER:: is_collision_detection_set
INTEGER:: error
error = is_stopping_condition_set(COLLISION_DETECTION, is_collision_detection_set)

```

long **supported_conditions**

```
set_supported_conditions()
```

Individual bits are set for supported conditions. This global must be defined by the code.

For a code that support collision detection and timeout detection the variable can be defined in c as:

```

set_support_for_condition(COLLISION_DETECTION);
set_support_for_condition(PAIR_DETECTION);

```

or in FORTRAN as:

```

INCLUDE "../..../lib/stopcond/stopcond.inc"

set_support_for_condition(COLLISION_DETECTION)
set_support_for_condition(PAIR_DETECTION)

```

To refer to the type of a stopping condition, the library defines the following macros:

COLLISION_DETECTION

PAIR_DETECTION

ESCAPER_DETECTION

TIMEOUT_DETECTION

NUMBER_OF_STEPS_DETECTION

The following four functions can be used to set and reset the stopping conditions.

int **reset_stopping_conditions()**

Resets all stopping conditions. Needs to be called just before entering the `outer_loop`

```

void outer_loop(double end_time)
{
    reset_stopping_conditions()
    while(model_time < end_time)
    {
        ...
    }
}

```

int **next_index_for_stopping_condition()**

Reserves a row on the stopping conditions table. Needs to be called after detecting a stopping condition so that the code can fill in the stopping condition information.

```

if(COLLISION_DETECTION_BITMAP & enabled_conditions) {
    int row = next_index_for_stopping_condition();
}

```

```
    set_stopping_condition_info(row, COLLISION_DETECTION);  
    ...  
}
```

int set_stopping_condition_info (int *index*, int *type*)

Sets the type of stopping condition encountered for the given row index. Use one of the predefined type macros for the type argument

```
set_stopping_condition_info(row, COLLISION_DETECTION);
```

int set_stopping_condition_particle_index (int *index*, int *column_index*, int *index_of_particle*)

Sets the id of a particles involved in the stopping condition (only needed for stopping conditions with particles)

```
set_stopping_condition_info(row, COLLISION_DETECTION);  
set_stopping_condition_particle_index(stopping_index, 0, ident[i]);  
set_stopping_condition_particle_index(stopping_index, 1, ident[j]);
```

13.4.1 Stopping conditions with parameters

The *timeout* and *number of steps* stopping conditions are parameter based, i.e. the conditions are met if the outer loop took a certain time or a reached a certain number of steps (iterations). The parameters are accessed through the following procedures:

int set_stopping_condition_timeout_parameter (double *value*);

Sets the computer (wall-clock) time parameter.

get_stopping_condition_timeout_parameter (double **value*);

Gets the computer (wall-clock) time parameter.

int set_stopping_condition_number_of_steps_parameter (int *value*);

Sets the number of steps parameter.

int get_stopping_condition_number_of_steps_parameter (int **value*);

Gets the number of steps parameter.

For MPI enabled codes, the stopping conditions must be distributed and collected. The following three functions can be used to manage the MPI communication.

int mpi_setup_stopping_conditions ()

Setups some globals for MPI communication, this needs to be called once per code execution (for example in a `initialize_code()` function)

int mpi_distribute_stopping_conditions ()

Distributes the state of the conditions, so each process knows which conditions are enabled by the user. Needs to be called when the parallel code is entered (and all other data is distributed).

int mpi_collect_stopping_conditions ()

Collects the individual tables of the set conditions and combines these to one table on the root process (MPI rank == 0). Needs to be called at the end of the parallel part of the code.

The procedure for adding stopping conditions is explained by the following steps.

1. Add compiler and linker flags to the build script or Makefile.

To use the `'stopcond'` library, the compile and link steps of the community code must be changed. The following compiler flags need to be added:

```
-I$(AMUSE_DIR)/lib/stopcond
```

For the link step the following linker flags are needed:

```
-L$(AMUSE_DIR)/lib/stopcond -lstopcondmpi
```

2. Include the header file and defining the globals

The name of the header file is “stopcond.h” and this header file needs to be included in all files referring to the stopping conditions.

```
#include <stopcond.h>
```

The code should take care of the setting of the `supported_conditions` bitmap in its initialization, e.g.:

```
set_support_for_condition(COLLISION_DETECTION);
set_support_for_condition(PAIR_DETECTION);
```

In FORTRAN include “stopcond.inc” in subroutines that require the constants.

```
INCLUDE "../lib/stopcond/stopcond.inc"
set_support_for_condition(COLLISION_DETECTION)
set_support_for_condition(PAIR_DETECTION)
```

3. Check for a condition when it is enabled and set the condition if found. In c:

```
int is_collision_detection_enabled;
is_stopping_condition_enabled(COLLISION_DETECTION, &is_collision_detection_enabled);

if(is_collision_detection_enabled)
{
    real sum_of_the_radii = radius_of_star[i] + radius_of_star[j];
    if (distance_between_stars <= sum_of_the_radii)
    {
        int stopping_index = next_index_for_stopping_condition();
        set_stopping_condition_info(stopping_index, COLLISION_DETECTION);
        set_stopping_condition_particle_index(
            stopping_index,
            0,
            identifier_of_star[i]
        );
        set_stopping_condition_particle_index(
            stopping_index,
            1,
            identifier_of_star[j]
        );
    }
}
```

and in FORTRAN:

```
INCLUDE "../lib/stopcond/stopcond.inc"
INTEGER::is_stopping_conditions_enabled
INTEGER::is_collision_detection_enabled, stopping_index
INTEGER::error

error = is_stopping_condition_enabled(COLLISION_DETECTION, is_collision_detection_enabled)
!...
IF is_collision_detection_enabled.EQ.1 THEN
    IF there_is_a_collision THEN
        stopping_index = next_index_for_stopping_conditions()
        error = set_stopping_condition_info(stopping_index, COLLISION_DETECTION)
    !...
    ENDIF
ENDIF
```

4. Break the outer loop when any condition is set. The `set_conditions` can be tested:

```
while(...) {
    ...
```

```
    if(set_conditions)
    {
        break;
    }
}

INCLUDE "../..lib/stopcond/stopcond.inc"
INTEGER::is_any_condition_set
DO WHILE ( )
!...
    IF (is_any_condition_set().EQ.1) EXIT
END DO
```

A good example of a code implementing the stopping conditions is the “hermite0” community code. The code can be found in the ‘src/amuse/community/hermite0’ directory.

INPUT / OUTPUT FRAMEWORK

14.1 Introduction

The reading and writing of the files is done by subclasses of the `FileFormatProcessor` class.

14.2 Extending

class `amuse.io.FileFormatProcessor` (*filename=None, set=None, format=None*)
Abstract base class of all fileformat processors

All classes providing loading or storing of files should be subclasses of this base class.

Every subclass must support the *filename*, *set* and *format* arguments. The arguments must all be optional.

Parameters

- **filename** – name of the file the read the data from
- **set** – set (of particles or entities) to store in the file
- **format** – format of the file, will be a string or class

Attribute provided_formats list of strings of the formats provided by the processor

extra_attributes

Extra attributes to store with the data set. Some formats (most notably the `amuse` native format) can store extra attributes with the set in file. The `'write_set_to_file'` function will collect all keyword arguments that do not match to an option into the extra attributes dictionary.

get_description_of_options ()

Yields tuples, each tuple contains the name of the option, a description of the option and the default values

load ()

Loads the set from the file and returns the set.

classmethod register ()

Register this class, so that it can be found by name in the `write_set_to_file()` and `read_set_from_file()` functions.

store ()

Stores the set in the file. The set and the file are both properties of the processor.

class `amuse.io.FullTextFileFormatProcessor` (*filename=None, set=None, format=None*)

Abstract base class of all fileformat processors that process their data by first reading the complete text string

Subclasses need to implement the `store_string()` and `load_string()` methods.

load_string (*string*)

Return a particle set, read from the string

store_string ()

Return a string representation of the particle set

OPTIONS

15.1 Introduction

The AMUSE framework provides a generic way to handle optional attributes for any class. When an optional attribute is requested the class can read the value from an configuration file or return a default value when nothing is found in the file. The values for all options are stored in one configuration file.

15.2 Configuration file location

The AMUSE framework will search for the configuration file in following locations:

1. First, the framework tries to find a file named **amuserc** in the current working directory
2. Second, the framework tries to find a hidden file named **.amuserc** in the home directory.
3. Last, the framework tries to find a file named **amuserc** in the amuse installation directory

When no file is found all options will refer to the default-values.

15.3 Configuration file format

The configuration file is formatted similar to Microsoft Windows INI files. The configuration file consists of sections, led by a [section] header and followed by name: value entries.

For example:

```
[channel]
redirection=none
debugger=xterm
```

15.4 Option values

Values for optional attributes are determined in four different ways:

- the attribute is set on an instance of the object:

```
channel.debugger = "ddd"
```

- given when creating an object of the class:

```
channel = MessageChannel(debugger = "ddd")
```

- given in the ini file:

```
[channel]
debugger = ddd
```

- set as default value:

```
@option
def debugger(self):
    return "ddd"
```

15.5 Sections lookup

Options can be set in different sections. To determine the value of an option the framework first searches the sections of a class and then the sections of the option.

15.6 Use

class `amuse.support.options.option` (*function=None, type='string', name=None, sections=(), choices=(), global_options=None*)

Decorator to define an option

Parameters

- **type** – Type of the value, used when reading from the configuration file. Can be “string”, “int”, “float” or “boolean”. Defaults to “string”
- **sections** – Sections in the configuration file to search for the option value, must be an array of strings
- **choices** – When given will check if the value of the option is in the array (must be a list or set of objects)
- **name** – By default the name of the option in the configuration file is the same as the name of the function, use this argument to use a different name (not recommended)

Options can only be defined on subclasses of `OptionalAttributes`

class `amuse.support.options.OptionalAttributes` (***optional_keyword_arguments*)

Abstract superclass for all classes supporting optional attributes.

To support optional attributes a class must inherit (directly or indirectly) from this class.

To support setting the attributes when an object is created the class must define a *catch-all* keyword argument in the `__init__` function and send this argument to the `__init__` of the superclass.

The values of options are first searched for in the sections given in the `option_sections` attribute of the class (empty by default). Next the sections of the option are searched.

For example:

```
class MyInterface(OptionalAttributes):
    option_sections = ('mysection',)

    def __init__(self, **options):
        OptionalAttributes.__init__(self, **options)

    @option(type="int", choices=(5,10,15), sections=('try',))
    def number_of_tries(self):
        "number of times to try to connect"
        return 5
```

To code will first search for the value of the option in the *mysection*, if no value is found the *try* section is searched. For the following configuration file the **number_of_tries** attribute will be 10 as the *mysection* section is searched first.

```
:: ini
```

```
[mysection] number_of_tries = 10 [try] number_of_tries = 5
```

The value of the option can be overridden by specifying it when creating an object of the class.

```
:: python
```

```
x = MyInterface(number_of_tries = 15) print x.number_of_tries 15
```


DIRECTORY STRUCTURE

The amuse source-code is separated into 3 directories:

- `src` - source code, implementation of the environment.
- `test` - applications, examples and unittests.
- `support` - build system, test system.

Under the `src` directories all code needed to run AMUSE can be found. One can view this code as an *library* that can be used to create *applications* to do numerical astrophysical experiments. This code will contain the building blocks needed to interface with codes, import and export data, do unit conversions, and all other AMUSE functionality.

Under the `test` directories all application and test code can be found. This directory tree will contain scripts to do a complete astrophysical experiment. Also all unit-tests can be found here. These unit tests each cover only a small part (unit) of the functionality of AMUSE. For example a test to check the import of a file to AMUSE data format.

Under the `support` directories all support code for the building system can be found.

16.1 The `src` directories

The directories under the `src` directory are further split into:

- `community` - contains the source code of existing astrophysical applications and *glue* code to the AMUSE interface classes. in other words this directory contains the implementation of the interfaces.
- `support` - contains the AMUSE generic code, defines the data representation and input/output routines and also provides the generic unit handling code. Code in the `interface` and `community` directories use these functions and classes to provide their functionality.

16.2 The `test` directories

The directories under the `test` directory are further split into:

- `unit_tests` - All unit testing code. These tests are coded using the standard unit testing framework that is included in the Python distribution (`unittest`). See python module documentation for further information: <http://docs.python.org/library/unittest.html>.
- `application` - contains the source code of published applications.
- `examples` - contains documented example codes.

16.3 The `support` directories

The directories under the `support` directory are further split into:

- `test` - Scripts to support the testing of AMUSE code.
- `build` - Scripts used by the building system of AMUSE.

SETTING UP GPGPU WITH CUDA

17.1 Introduction

Here we provide help for setting up general-purpose computing on graphics processing units (GPGPU) using CUDA. Performing (part of) the calculations on a graphics card can result in a significant speed-up. Currently two codes in AMUSE support or require GPGPU: phi-GRAPE (using Sapporo) and Octgrav.

17.2 Self-help script

In the AMUSE root directory a self-help script can be found. If building or testing any of the codes mentioned above fails and you wonder why, it will hopefully provide you with helpful suggestions. From a command line run the bash script `cuda_self_help`:

```
> ./amuse-x.0/cuda_self_help
```

17.3 Step-by-step

- *Check that your computer has a CUDA-capable Nvidia graphics card*
- *Check that you have installed the CUDA Toolkit (TK) and software development kit (SDK)*
- *Set the `CUDA_TK` and `CUDA_SDK` environment variables*
- *Testing*

17.3.1 Check that your computer has a CUDA-capable Nvidia graphics card

First determine the model of your GPU.

On Linux:

```
> nvidia-settings -q gpus
```

On Mac:

1. Click on “Apple Menu”
2. Click on “About this Mac”
3. Click on “More Info”
4. Select “Graphics/Displays” under Contents list

Check whether your GPU model is listed among [Nvidia’s CUDA-enabled GPUs](#).

17.3.2 Check that you have installed the CUDA Toolkit (TK) and software development kit (SDK)

If not, download and install it from [CUDA Development Tools](#).

17.3.3 Set the CUDA_TK and CUDA_SDK environment variables

After installing the CUDA TK and SDK, make sure the environment variables CUDA_TK and CUDA_SDK are set correctly. For shell (bash) you need to do:

```
export CUDA_TK=/path/to/cuda_tk
export CUDA_SDK=/path/to/cuda_sdk
```

'/path/to/cuda_tk' should hold directories named 'include', 'lib', and 'lib64' (where libcuda.so is located)

'/path/to/cuda_sdk' should hold a directory named 'common/inc' (where various header files are located)

We recommend you add these lines to your '.bashrc' file so that the variables are set correctly for all sessions. If you have a C shell you need to do a *setenv* and edit the '.cshrc' file.

17.3.4 Testing

Now try building for example Octgrav and run the nosetests (from AMUSE root directory), but first re-initialize mpd (or it will remember its original environment):

```
mpdallexit
mpd &
make octgrav.code
nosetests ./test/codes_tests/test_octgrav.py
```

If this fails, please contact us through the 'amusecode' [google group](#), or on IRC at the #amuse channel on [irc.freenode.net](#).

DISTRIBUTED AMUSE

It is possible to run AMUSE on multiple machines simultaneously. The AMUSE script itself always runs on a users' local machine, while workers for codes can be "send out" to remote machines such as workstations, clusters, etc.

18.1 Installation

Deploying workers on remote machines requires a full installation of AMUSE on each machine. For each code used the `sockets` worker needs to be built. This is done by default, and should be available for all codes.

On the local machine, the Ibis library also needs to be built. Ibis requires a Java Development Kit (JDK), preferably Sun Java version 6. The `configure` script tries to locate the JDK, but you may need to specify it by hand. For details, see:

```
./configure --help
```

AMUSE will build the Ibis library by default if a JDK is found. However, it needs to download some code to build it, so the `DOWNLOAD_CODES=1` option needs to be specified as well. Building the Ibis library manually can be done by running:

```
make ibis.code DOWNLOAD_CODES=1
```

in the root of AMUSE.

18.2 Specifying resources using Jungle files

In order to use a remote machine (resource) for a worker, AMUSE needs to know information about this resource such as hostname, type of machine, username to gain access, etc. This is specified in so-called Jungle files. AMUSE comes with a set of jungle files for the Observatory machines, the LGM, the DAS, etc, in `lib/ibis`.

Users should not need to change the jungle files themselves. On exception is the `user.name` option, which may need to be set for each machine. By default, AMUSE assumes the username on each machine is equal to the users' username on the local machine.

The format of the jungle files is still under active development, and changes in the format will occur from time to time. Currently supported options are listed at the beginning of each file.

18.3 Starting Ibis Deploy

To run workers on remote machine a special support process needs to be started on the local machine. The script to start this is created automatically by AMUSE, and is called `ibis-deploy.sh`. It has several options, see:

```
./ibis-deploy.sh --help
```

Jungle files can be passed to the deployment program with the `--jungle-file` option.

If preferred, the status of the deployment application can also be monitored by specifying the `--gui` option. This will create a window showing monitoring information.

18.4 Running workers remotely

To run a worker on a remote machine, some additional options need to be given to AMUSE when creating a worker. First the special *ibis* channel implementation needs to be specified. Also the name of the resource, number of nodes, and number of processes to start in total needs to be specified. For instance, to run an instant of Gadget on node06 of the LGM cluster, and using 8 processes for this one worker, use:

```
instance = Gadget2Interface(channel_type='ibis', hostname='lgm06',
                             number_of_nodes=1, number_of_processes=8)
```

An easy way to start all workers remotely on some resource is to use the generic configuration file system of AMUSE, see *Options*.

AMUSE STYLE GUIDE

This Style Guide covers the sourcecode written in the AMUSE project. The source of the existing codes, integrated into AMUSE, do not have to follow this guide. Existing codes are not rewritten.

19.1 Python Code

All python code should be consistent with the [Python Style Guide](#). We have defined some deviations from the python style guide. These are listed in this document.

Maximum Line Length Limit *most* lines to a maximum of 79 characters.

Some lines may be longer than 79 characters, especially when this increased readability. When you need to define a class with a long name, having a superclass with a long name, a line with more than 79 characters is OK.

For example:

```
class NeedsALongNameToDescribeItsFunction (AComplicatedNameForTheSuperClass):
```

Is better than:

```
class NeedsALongNameToDescribeItsFunction \
    (AComplicatedNameForTheSuperClass):
```

Or:

```
class NeedsALongNameToDescribeItsFunction (
    AComplicatedNameForTheSuperClass):
```

Naming Conventions Naming the classes and methods in the code is very important. However, do not spend too much time on naming. Use long names. Use a longer name when a shorter name cannot be determined quickly. When you've defined a very long name, it's use may be harder. When it is used a lot, a refactoring to shorter name will present itself. When it is not used a lot, at least the class or method is described better in the longer name.

Do not use abbreviations in names. Especially, do not use the first three or four letters of a word (for example, *dir* for directory or *ref* for reference). Also, do not remove the vowels to shorten a name (for example *tmp* for temp).

The following one letter variables are allowed.

- Loop variables, when used as `for x in ...`, names allowed are 'x' and 'y'. Example:

```
for x in stars:
    print x.mass
```

- Loop index variables, when used as `for i in range(10) :`, only name allowed is 'i'. Thus when writing a loop in a loop, you need to use longer names. For example, to iterate over every item in a 10 by 5 table, do:s

for row in range(10):

for column in range(5): print table[row][column]

These names are only allowed in short loops of maximum 6 lines. Longer loops need better names. Also, for looping over row indices and column indices use *row* and *column*.

19.2 C / C++ code

All C and C++ code should be consistent with the [Google C++ style guid](#)

File Names All C++ files should have a `.cc` suffix.

Naming Conventions We follow the Python style for naming function names. Function names are all lowercase, with underscores between words.

19.3 Fortran code

All Fortran code written in AMUSE should be Fortran 90. We will follow the coding standard given in [Fortran Coding Standard for the Community Atmospheric Model](#)

File Names All fortran 90 files should have a `.f90` suffix.

Layout Source code should follow the “free form” fortran 90. Do not use the “fixed form” fortran 77 layout.

Indentation Use indentation to highlight the block structure of your code:

```
FUNCTION example(arg1)
  REAL :: arg1, example
  example = arg1 * 2.0
END FUNCTION
```

Use 4 character to indent the code.

SOURCE CODE MANAGEMENT

20.1 Committing Code

20.2 SVN repository location

The AMUSE source code repository can be found at:

<http://castle.strw.leidenuniv.nl/projects/amuse/trunk>

GLOSSARY

Application A script that combines modules into a multi-physical simulator.

Binary star Two stars in close orbit

Code A computer program to solve or approximate the equations of state of a physical domain. Codes are usually written in Fortran or C/C++. For example 'sse' is a code to perform stellar evolution.

Data-model A single representation model for data of different physical codes.

Evolve Work out / develop the physical model equations. Usually, small steps in time are taken, the new state of the system is calculated from the previous state.

Grid A collection of identifiable locations or regions (for example a box of interstellar medium with a density, molecular composition). Every value in the collection can be seen as belonging to a specific location (x,y,z point).

Interstellar Cloud Generic name for an accumulation of gas, plasma and dust. A denser region of the interstellar medium

Interstellar Medium The matter between the stars in a galaxy.

Main sequence star A star that derives its energy from the conversion of hydrogen into helium in its core.

Metallicity Proportion of star matter made up of chemical elements other than hydrogen and helium.

Module Interface to a code in AMUSE. Also, file containing python definitions. These definitions can be imported into the main Python program

Molecular Cloud A type of interstellar cloud whose density and size permits the formation of molecules,

Physical Domain A region in space, time and objects (stars / gas clouds) that can be described by a set of physical models. A physical domain is a limited description of the physics of a stellar system. For example in the gravity domain the gravitational forces are taken into account but not the evolution of stars or the hydrodynamics of the interstellar medium

Production Code A code used for modelling stellar systems and described in published articles.

Run During a run the physical models are evolved to a wanted end state. A run can end when a certain model time has passed. For example, one can run a script until the age is 12 million years.

Set A collection of identifiable objects (for example all stars in a globular cluster). Every value in the set can be seen as belonging to an object.

SPH, smoothed particle hydrodynamics A computational method used for simulating fluid flows

Star Cluster A group of stars. The mean distance between the stars is smaller than the mean distance between stars in the galaxy disks. Two types of star clusters exist: globular clusters, open clusters

Step An identifiable stage in a run or evolve. A code can take multiple steps during one evolve step. A step can be the integrator step, force calculation step.

Toy Code A code with limited accuracy but that is easy to use.

Unit test Automatic test case for a part of the code. For example one can define a unit tests that checks the result of one method given a specified argument.

ZAMS, zero age main sequence Start of the main sequence in the evolution of a star. Stable stage after formation of the star from a collapsing gas cloud

PYTHON MODULE INDEX

a

- amuse.community.athena.interface, 52
- amuse.community.bhtree.interface, 37
- amuse.community.bse.interface, 49
- amuse.community.capreole.interface, 52
- amuse.community.evtwin.interface, 51
- amuse.community.fi.interface, 53
- amuse.community.gadget2.interface, 57
- amuse.community.hermite0.interface, 38
- amuse.community.higpus.interface, 47
- amuse.community.interface.gd, 83
- amuse.community.mercury.interface, 43
- amuse.community.mesa.interface, 51
- amuse.community.mmc.interface, 46
- amuse.community.octgrav.interface, 42
- amuse.community.phiGRAPE.interface, 40
- amuse.community.simplex.interface, 60
- amuse.community.sse.interface, 48
- amuse.datamodel, 86
- amuse.datamodel.incode_storage, 105
- amuse.io, 63
- amuse.rfi.core, 33
- amuse.support.core, 29
- amuse.support.options, 127
- amuse.units.core, 77
- amuse.units.generic_unit_converter, 82
- amuse.units.generic_unit_system, 81
- amuse.units.nbody_system, 82
- amuse.units.quantities, 71

INDEX

Symbols

`__add__()` (amuse.datamodel.AbstractParticleSet method), 87
`__add__()` (amuse.datamodel.Particle method), 93
`__add__()` (amuse.support.core.print_out method), 29
`__getitem__()` (amuse.units.quantities.VectorQuantity method), 73
`__setitem__()` (amuse.units.quantities.VectorQuantity method), 73
`__sub__()` (amuse.datamodel.AbstractParticleSet method), 87
`__sub__()` (amuse.datamodel.Particle method), 93

A

`AbstractInCodeAttributeStorage` (class in amuse.datamodel.incode_storage), 106
`AbstractParticleSet` (class in amuse.datamodel), 86
`acc_timestep_crit_constant` (amuse.community.fi.interface.Fi.parameters attribute), 55
`acc_timestep_flag` (amuse.community.fi.interface.Fi.parameters attribute), 54
`adaptive_smoothing_flag` (amuse.community.fi.interface.Fi.parameters attribute), 54
`add_entities()` (amuse.datamodel.incode_storage.NewParticleMethod method), 107
`add_particle()` (amuse.datamodel.AbstractParticleSet method), 88
`add_particles()` (amuse.datamodel.AbstractParticleSet method), 88
`add_particles_to_store()` (amuse.datamodel.incode_storage.InCodeAttributeStorage method), 106
`add_particles_to_store()` (amuse.datamodel.incode_storage.InCodeGridAttributeStorage method), 106
`add_particles_to_store()` (amuse.datamodel.ParticlesSubset method), 92
`addParameter()` (amuse.rfi.core.LegacyFunctionSpecification method), 34
`amax()` (amuse.units.quantities.VectorQuantity method), 73
`amin()` (amuse.units.quantities.VectorQuantity method), 73

amuse.community.athena.interface (module), 52
amuse.community.bhtree.interface (module), 37
amuse.community.bse.interface (module), 49
amuse.community.capeole.interface (module), 52
amuse.community.evtwin.interface (module), 51
amuse.community.fi.interface (module), 53
amuse.community.gadget2.interface (module), 57
amuse.community.hermite0.interface (module), 38
amuse.community.higpus.interface (module), 47
amuse.community.interface.gd (module), 83
amuse.community.mercury.interface (module), 43
amuse.community.mesa.interface (module), 51
amuse.community.mmc.interface (module), 46
amuse.community.octgrav.interface (module), 42
amuse.community.phiGRAPE.interface (module), 40
amuse.community.simplex.interface (module), 60
amuse.community.sse.interface (module), 48
amuse.datamodel (module), 86
amuse.datamodel.incode_storage (module), 105
amuse.datamodel.particle_attributes (module), 93, 97
amuse.io (module), 63
amuse.rfi.core (module), 33
amuse.support.core (module), 29
amuse.support.options (module), 127
amuse.units.core (module), 77
amuse.units.generic_unit_converter (module), 82
amuse.units.generic_unit_system (module), 81
amuse.units.nbody_system (module), 82
amuse.units.quantities (module), 71
`append()` (amuse.units.quantities.VectorQuantity method), 74
Application, 141
`apply_for_superset()` (amuse.datamodel.incode_storage.ParticleQueryMethod method), 108
`apply_normal()` (amuse.datamodel.incode_storage.ParticleQueryMethod method), 108
`apply_on_all()` (amuse.datamodel.incode_storage.ParticleMethod method), 108
`apply_on_all()` (amuse.datamodel.incode_storage.ParticlesAddedUpdateMethod method), 109
`apply_on_all()` (amuse.datamodel.incode_storage.ParticleSetSelectSubsetMethod method), 109
`apply_on_all()` (amuse.datamodel.incode_storage.ParticleSpecificSelectMethod method), 109
`apply_on_all()` (amuse.datamodel.incode_storage.ParticleSpecificSelectMethod method), 109

[apply_on_one\(\) \(amuse.datamodel.incode_storage.ParticleMethod\)](#), 108
[apply_on_one\(\) \(amuse.datamodel.incode_storage.ParticleSetAttributesMethod\)](#), 109
[apply_on_one\(\) \(amuse.datamodel.incode_storage.ParticleSetAttributesMethod\)](#), 109
[argsort\(\) \(amuse.units.quantities.VectorQuantity method\)](#), 74
[artificial_viscosity_alpha \(amuse.community.fi.interface.Fi.parameters attribute\)](#), 55
[artificial_viscosity_alpha \(amuse.community.gadget2.interface.Gadget2.parameters attribute\)](#), 58
[as_quantity_in\(\) \(amuse.units.core.unit method\)](#), 80
[as_quantity_in\(\) \(amuse.units.quantities.Quantity method\)](#), 72
[as_set\(\) \(amuse.datamodel.AbstractParticleSet method\)](#), 88
[as_set\(\) \(amuse.datamodel.Particle method\)](#), 93
[Athena \(class in amuse.community.athena.interface\)](#), 52
[attribute_names \(amuse.datamodel.incode_storage.ParticleAttribute\)](#), 107
[attribute_names \(amuse.datamodel.incode_storage.ParticleSetAttribute\)](#), 107
[attribute_names \(amuse.datamodel.incode_storage.ParticleSetAttribute\)](#), 108

B

[base \(amuse.units.core.base_unit attribute\)](#), 77
[base_unit \(class in amuse.units.core\)](#), 77
[beta \(amuse.community.fi.interface.Fi.parameters attribute\)](#), 55
[BHTree \(class in amuse.community.bhtree.interface\)](#), 37
[Binary star](#), 141
[binary_enhanced_mass_loss_parameter \(amuse.community.bse.interface.BSE.parameters attribute\)](#), 49
[binary_enhanced_mass_loss_parameter \(amuse.community.sse.interface.SSE.parameters attribute\)](#), 48
[black_hole_kick_flag \(amuse.community.bse.interface.BSE.parameters attribute\)](#), 50
[black_hole_kick_flag \(amuse.community.sse.interface.SSE.parameters attribute\)](#), 48
[BSE \(class in amuse.community.bse.interface\)](#), 49

C

[can_extend_attributes\(\) \(amuse.datamodel.incode_storage.InCodeAttributeStorage method\)](#), 106
[CannotLoadException \(class in amuse.io\)](#), 64
[CannotSaveException \(class in amuse.io\)](#), 64
[Capreole \(class in amuse.community.capreole.interface\)](#), 52

[CodeFunction \(class in amuse.rfi.core\)](#), 33
[CodeInterface \(class in amuse.rfi.core\)](#), 33
[COLLISION_DETECTION \(built-in variable\)](#), 121
[COLLISION_DETECTION_BITMAP \(built-in variable\)](#), 120
[CollisionMethod \(amuse.community.interface.hydro.HydrodynamicsInterfaceAttribute\)](#), 17
[CollisionMethod \(amuse.community.interface.gd.GravitationalDynamicsInterfaceAttribute\)](#), 12
[common_envelope_binding_energy_factor \(amuse.community.bse.interface.BSE.parameters attribute\)](#), 49
[common_envelope_efficiency \(amuse.community.bse.interface.BSE.parameters attribute\)](#), 49
[common_envelope_model_flag \(amuse.community.bse.interface.BSE.parameters attribute\)](#), 49
[comoving_integration_flag \(amuse.community.gadget2.interface.Gadget2.parameters attribute\)](#), 59
[conservative_sph_flag \(amuse.community.fi.interface.Fi.parameters attribute\)](#), 54
[convert_attributes_and_values_to_list_and_keyword_arguments\(\) \(amuse.datamodel.incode_storage.ParticleSetAttributesMethod method\)](#), 108
[convert_return_value\(\) \(amuse.datamodel.incode_storage.ParticleGetAttributesMethod method\)](#), 107
[ConvertBetweenGenericAndSiUnits \(class in amuse.units.generic_unit_converter\)](#), 82
[cool_par \(amuse.community.fi.interface.Fi.parameters attribute\)](#), 56
[copy\(\) \(amuse.datamodel.incode_storage.InCodeGridAttributeStorage method\)](#), 106
[copy_values_of_attribute_to\(\) \(amuse.datamodel.AbstractParticleSet method\)](#), 88

- courant (amuse.community.fi.interface.Fi.parameters attribute), 56
- courant (amuse.community.gadget2.interface.Gadget2.parameters attribute), 58
- cpu_file (amuse.community.gadget2.interface.Gadget2.parameters attribute), 58
- cross() (amuse.units.quantities.VectorQuantity method), 74
- ## D
- Data-model, 141
- dedent() (amuse.support.core.print_out method), 29
- delete_particle (amuse.community.interface.gd.GravitationalDynamicsInterface attribute), 6
- densitycentre_coreradius_coredens() (in module amuse.datamodel.particle_attributes), 98
- derived_unit (class in amuse.units.core), 77
- difference() (amuse.datamodel.AbstractParticleSet method), 89
- direct_sum_flag (amuse.community.fi.interface.Fi.parameters attribute), 54
- disable_stopping_condition (amuse.support.codes.stopping_conditions.StoppingConditionsInterface attribute), 114
- distances_squared() (in module amuse.datamodel.particle_attributes), 98
- div_unit (class in amuse.units.core), 77
- dt_dia (amuse.community.bhtree.interface.BHTree.parameters attribute), 37
- dt_dia (amuse.community.hermite0.interface.Hermite.parameters attribute), 38
- dt_param (amuse.community.hermite0.interface.Hermite.parameters attribute), 38
- dt_Print (amuse.community.higpus.interface.HiGPUs.parameters attribute), 47
- ## E
- Eddington_mass_transfer_limit_factor (amuse.community.bse.interface.BSE.parameters attribute), 50
- enable_stopping_condition (amuse.support.codes.stopping_conditions.StoppingConditionsInterface attribute), 115
- enabled_conditions (C variable), 120
- energy_file (amuse.community.gadget2.interface.Gadget2.parameters attribute), 58
- enumeration_unit (class in amuse.units.core), 78
- eps_is_h_flag (amuse.community.fi.interface.Fi.parameters attribute), 54
- eps_is_h_flag (amuse.community.gadget2.interface.Gadget2.parameters attribute), 58
- epsilon_squared (amuse.community.bhtree.interface.BHTree.parameters attribute), 37
- epsilon_squared (amuse.community.fi.interface.Fi.parameters attribute), 53
- epsilon_squared (amuse.community.gadget2.interface.Gadget2.parameters attribute), 57
- epsilon_squared (amuse.community.hermite0.interface.Hermite.parameters attribute), 38
- epsilon_squared (amuse.community.octgrav.interface.Octgrav.parameters attribute), 42
- epsilon_squared (amuse.community.phiGRAPE.interface.PhiGRAPE.parameters attribute), 40
- ESCAPER_DETECTION (built-in variable), 121
- ESCAPER_DETECTION_BITMAP (built-in variable), 120
- eta_4 (amuse.community.higpus.interface.HiGPUs.parameters attribute), 47
- eta_6 (amuse.community.higpus.interface.HiGPUs.parameters attribute), 47
- Evolve, 141
- Evtwin (class in amuse.community.evtwin.interface), 51
- example_function (amuse.community.interface.example.ExampleInterface attribute), 1
- ExampleInterface (class in amuse.community.interface.example), 1–3
- extend() (amuse.units.quantities.VectorQuantity method), 74
- extra_attributes (amuse.io.FileFormatProcessor attribute), 125
- ## F
- factor (amuse.units.core.base_unit attribute), 77
- factor_unit (class in amuse.units.core), 78
- feedback (amuse.community.fi.interface.Fi.parameters attribute), 56
- Fair class in amuse.community.fi.interface), 53
- fi_data_directory (amuse.community.fi.interface.Fi.parameters attribute), 57
- FileFormatProcessor (class in amuse.io), 125
- first_snapshot (amuse.community.fi.interface.Fi.parameters attribute), 54
- fixed_halo_flag (amuse.community.fi.interface.Fi.parameters attribute), 54
- fractional_time_step_1 (amuse.community.bse.interface.BSE.parameters attribute), 50
- fractional_time_step_1 (amuse.community.sse.interface.SSE.parameters attribute), 48
- fractional_time_step_2 (amuse.community.bse.interface.BSE.parameters attribute), 50
- fractional_time_step_2 (amuse.community.sse.interface.SSE.parameters attribute), 48
- fractional_time_step_3 (amuse.community.bse.interface.BSE.parameters attribute), 50
- fractional_time_step_3 (amuse.community.sse.interface.SSE.parameters attribute), 48

- free_timestep_crit_constant_a
(amuse.community.fi.interface.Fi.parameters attribute), 55
- free_timestep_crit_constant_aexp
(amuse.community.fi.interface.Fi.parameters attribute), 55
- free_timestep_crit_constant_v
(amuse.community.fi.interface.Fi.parameters attribute), 55
- free_timestep_crit_constant_vexp
(amuse.community.fi.interface.Fi.parameters attribute), 55
- freeform_timestep_flag
(amuse.community.fi.interface.Fi.parameters attribute), 54
- from_source_to_target()
(amuse.datamodel.ParticlesWithUnitsConverted.ConverterInterface method), 92
- from_target_to_source()
(amuse.datamodel.ParticlesWithUnitsConverted.ConverterInterface method), 92
- FullTextFileFormatProcessor (class in amuse.io), 125
- ## G
- Gadget2 (class in amuse.community.gadget2.interface), 57
- gadget_cell_opening_constant
(amuse.community.fi.interface.Fi.parameters attribute), 55
- gadget_cell_opening_constant
(amuse.community.gadget2.interface.Gadget2.parameters attribute), 58
- gadget_cell_opening_flag
(amuse.community.fi.interface.Fi.parameters attribute), 54
- gadget_cell_opening_flag
(amuse.community.gadget2.interface.Gadget2.parameters attribute), 57
- gadget_output_directory
(amuse.community.gadget2.interface.Gadget2.parameters attribute), 58
- gamma (amuse.community.fi.interface.Fi.parameters attribute), 55
- gas_epsilon (amuse.community.fi.interface.Fi.parameters attribute), 55
- gas_epsilon (amuse.community.gadget2.interface.Gadget2.parameters attribute), 58
- get_acceleration (amuse.community.interface.gd.GravitationalDynamicsInterface attribute), 11
- get_all_indices_in_store()
(amuse.datamodel.incode_storage.InCodeAttributeStorage method), 106
- get_all_keys_in_store()
(amuse.datamodel.incode_storage.InCodeAttributeStorage method), 106
- get_all_keys_in_store()
(amuse.datamodel.incode_storage.InCodeGridAttributeStorage method), 106
- get_attribute_values() (amuse.datamodel.incode_storage.ParticleGetAttribute method), 107
- get_attribute_values() (amuse.datamodel.incode_storage.ParticleGetIndex method), 107
- get_binaries() (in module amuse.datamodel.particle_attributes), 98
- get_center_of_mass_position
(amuse.community.interface.gd.GravitationalDynamicsInterface attribute), 14
- get_center_of_mass_velocity
(amuse.community.interface.gd.GravitationalDynamicsInterface attribute), 13
- get_data_directory() (amuse.rfi.core.CodeInterface method), 33
- get_defined_attribute_names()
(amuse.datamodel.incode_storage.AbstractInCodeAttributeStorage method), 106
- get_defined_attribute_names()
(amuse.datamodel.incode_storage.InCodeGridAttributeStorage method), 106
- get_description_of_options()
(amuse.io.FileFormatProcessor method), 125
- get_eps2 (amuse.community.interface.gd.GravitationalDynamicsInterface attribute), 4
- get_example_parameter
(amuse.community.interface.example.ExampleInterface attribute), 2
- get_gravity_at_point (amuse.community.interface.gd.GravitationalDynamicsInterface attribute), 15
- get_grid_state (amuse.community.interface.hydro.HydrodynamicsInterface attribute), 20
- get_index_of_first_particle
(amuse.community.interface.gd.GravitationalDynamicsInterface attribute), 7
- get_index_of_next_particle
(amuse.community.interface.gd.GravitationalDynamicsInterface attribute), 7
- get_index_of_position (amuse.community.interface.hydro.HydrodynamicsInterface attribute), 18
- get_index_range_inclusive()
(amuse.community.interface.hydro.HydrodynamicsInterface method), 17
- get_indices_of() (amuse.datamodel.incode_storage.InCodeAttributeStorage method), 106
- get_key_indices_of() (amuse.datamodel.incode_storage.InCodeAttributeStorage method), 106
- get_kinetic_energy (amuse.community.interface.gd.GravitationalDynamicsInterface attribute), 13
- get_mass (amuse.community.interface.gd.GravitationalDynamicsInterface attribute), 9
- get_number_of_particles
(amuse.community.interface.gd.GravitationalDynamicsInterface attribute), 6
- get_number_of_stopping_conditions_set
(amuse.support.codes.stopping_conditions.StoppingConditionInterface attribute), 115
- get_options_for_format() (in module amuse.io), 64

get_output_directory() (amuse.rfi.core.CodeInterface GravitationalDynamicsInterface (class in
 method), 33 amuse.community.interface.gd), 4, 5, 8,
 get_position (amuse.community.interface.gd.GravitationalDynamicsInterface, 15
 attribute), 10 Grid, 141
 get_position_of_index (amuse.community.interface.hydro.HydrodynamicsInterface
 attribute), 18
 get_potential (amuse.community.interface.gd.GravitationalDynamicsInterface (amuse.community.fi.interface.Fi.parameters
 attribute), 12 attribute), 56
 get_potential_energy (amuse.community.interface.gd.GravitationalDynamicsInterface (amuse.community.fi.interface.Fi.parameters
 attribute), 13 attribute), 56
 get_state (amuse.community.interface.gd.GravitationalDynamicsInterface) (amuse.datamodel.AbstractParticleSet
 attribute), 8 method), 89
 get_stopping_condition_info has_key_in_store() (amuse.datamodel.incode_storage.InCodeAttributeStorage
 (amuse.support.codes.stopping_conditions.StoppingConditionInterface
 attribute), 115 method), 106
 get_stopping_condition_maximum_density_parameter has_key_in_store() (amuse.datamodel.incode_storage.InCodeGridAttributeStorage
 (amuse.support.codes.stopping_conditions.StoppingConditionInterface (amuse.units.core.unit method),
 attribute), 116 80
 get_stopping_condition_maximum_internal_energy_parameter has_key_in_store() (amuse.support.codes.stopping_conditions.StoppingConditionInterface
 (amuse.support.codes.stopping_conditions.StoppingConditionInterface
 attribute), 116 attribute), 117
 get_stopping_condition_minimum_density_parameter heat_par1 (amuse.community.fi.interface.Fi.parameters
 (amuse.support.codes.stopping_conditions.StoppingConditionInterface
 attribute), 116 attribute), 56
 get_stopping_condition_minimum_internal_energy_parameter heat_par2 (amuse.community.fi.interface.Fi.parameters
 (amuse.support.codes.stopping_conditions.StoppingConditionInterface
 attribute), 116 attribute), 56
 get_stopping_condition_number_of_steps_parameter pingConditionInterface (amuse.community.bse.interface.BSE.parameters
 (amuse.support.codes.stopping_conditions.StoppingConditionInterface
 attribute), 116 attribute), 49
 get_stopping_condition_out_of_box_parameter pingConditionInterface (amuse.community.sse.interface.SSE.parameters
 (amuse.support.codes.stopping_conditions.StoppingConditionInterface
 attribute), 117 attribute), 48
 get_stopping_condition_particle_index pingConditionInterface (amuse.community.hermite0.interface),
 (amuse.support.codes.stopping_conditions.StoppingConditionInterface
 attribute), 117 38
 get_stopping_condition_timeout_parameter HiGPUs (class in amuse.community.higpus.interface),
 (amuse.support.codes.stopping_conditions.StoppingConditionInterface
 attribute), 117 47
 get_time (amuse.community.interface.gd.GravitationalDynamicsInterface) hubble_param (amuse.community.gadget2.interface.Gadget2.parameters
 attribute), 13 attribute), 59
 get_time (amuse.community.interface.hydro.HydrodynamicsInterface pingConditionInterface (class in
 attribute), 22 amuse.community.interface.hydro), 17,
 get_total_mass IN (amuse.rfi.core.LegacyFunctionSpecification
 (amuse.community.interface.gd.GravitationalDynamicsInterface
 attribute), 14 attribute), 3
 get_total_radius (amuse.community.interface.gd.GravitationalDynamicsInterface in_() (amuse.units.core.unit method), 80
 (amuse.community.interface.gd.GravitationalDynamicsInterface
 attribute), 15 onCodeAttributeStorage (class in
 amuse.datamodel.incode_storage), 106
 get_values_in_store() (amuse.datamodel.incode_storage.InCodeAttributeStorage (class in
 method), 106 amuse.datamodel.incode_storage), 106
 get_values_in_store() (amuse.datamodel.incode_storage.InCodeGridAttributeStorage (amuse.support.core.print_out method), 30
 method), 106 indent_characters() (amuse.support.core.print_out
 method), 30
 gpu_name (amuse.community.higpus.interface.HiGPUs.parameters method), 30
 attribute), 47 info_file (amuse.community.gadget2.interface.Gadget2.parameters
 attribute), 58
 grain_heat_eff (amuse.community.fi.interface.Fi.parameters attribute), 56
 attribute), 56 initial_timestep_parameter
 GravitationalDynamics (class in (amuse.community.phiGRAPE.interface.PhiGRAPE.parameters
 amuse.community.interface.gd), 83 attribute), 40

- initialize_code (amuse.community.interface.example.ExampleInterface attribute), 3
- initialize_code (amuse.community.interface.gd.GravitationalDynamicsInterface attribute), 5
- initialize_gpu_once (amuse.community.phiGRAPE.interface.PhiGRAPE.parameters attribute), 40
- initialize_grid (amuse.community.interface.hydro.HydrodynamicsInterface attribute), 22
- INOUT (amuse.rfi.core.LegacyFunctionSpecification attribute), 33
- integrate_entropy_flag (amuse.community.fi.interface.Fi.parameters attribute), 54
- interpret_heat_as_feedback (amuse.community.gadget2.interface.Gadget2.parameters attribute), 60
- interpret_kicks_as_feedback (amuse.community.gadget2.interface.Gadget2.parameters attribute), 59
- Interstellar Cloud, 141
- Interstellar Medium, 141
- is_quantity() (amuse.units.quantities.Quantity method), 72
- is_scalar() (amuse.units.quantities.Quantity method), 72
- is_stopping_condition_enabled (amuse.support.codes.stopping_conditions.StoppingConditionInterface attribute), 118
- is_stopping_condition_set (amuse.support.codes.stopping_conditions.StoppingConditionInterface attribute), 118
- is_vector() (amuse.units.quantities.Quantity method), 72
- isothermal_flag (amuse.community.fi.interface.Fi.parameters attribute), 54
- isothermal_flag (amuse.community.gadget2.interface.Gadget2.parameters attribute), 58
- ## K
- k() (in module amuse.units.core), 79
- kinetic_energy() (in module amuse.datamodel.particle_attributes), 98
- ## L
- LagrangianRadii() (in module amuse.datamodel.particle_attributes), 97
- late (class in amuse.support.core), 29
- legacy_function (in module amuse.rfi.core), 34
- legacy_global (in module amuse.rfi.core), 34
- LegacyFunctionSpecification (class in amuse.rfi.core), 33
- LENGTH (amuse.rfi.core.LegacyFunctionSpecification attribute), 33
- length() (amuse.units.quantities.VectorQuantity method), 74
- length_squared() (amuse.units.quantities.VectorQuantity method), 74
- length_x (amuse.community.cacreole.interface.Cacreole.parameters attribute), 53
- length_y (amuse.community.cacreole.interface.Cacreole.parameters attribute), 53
- length_z (amuse.community.cacreole.interface.Cacreole.parameters attribute), 53
- lengths_squared() (amuse.units.quantities.VectorQuantity method), 74
- lengths_x (amuse.community.cacreole.interface.Cacreole.parameters attribute), 53
- lengths_y (amuse.community.cacreole.interface.Cacreole.parameters attribute), 53
- lengths_z (amuse.community.cacreole.interface.Cacreole.parameters attribute), 53
- log_interval (amuse.community.fi.interface.Fi.parameters attribute), 55
- log_noindent() (amuse.support.core.print_out method), 30
- load() (amuse.io.FileFormatProcessor method), 125
- load_string() (amuse.io.FullTextFileFormatProcessor method), 125
- log_interval (amuse.community.fi.interface.Fi.parameters attribute), 55
- ## M
- Main sequence star, 141
- mass_plummer (amuse.community.higpus.interface.HiGPUs.parameters attribute), 47
- max() (amuse.units.quantities.VectorQuantity method), 75
- max_density (amuse.community.fi.interface.Fi.parameters attribute), 56
- max_size_timestep (amuse.community.gadget2.interface.Gadget2.parameters attribute), 59
- max_step (amuse.community.higpus.interface.HiGPUs.parameters attribute), 47
- maximum_neutron_star_mass (amuse.community.bse.interface.BSE.parameters attribute), 50
- maximum_neutron_star_mass (amuse.community.sse.interface.SSE.parameters attribute), 48
- maximum_time_bin (amuse.community.fi.interface.Fi.parameters attribute), 55
- MercuryWayWard (class in amuse.community.mercury.interface), 43
- MESA (class in amuse.community.mesa.interface), 51
- mesh_length (amuse.community.cacreole.interface.Cacreole.parameters attribute), 53
- mesh_size (amuse.community.cacreole.interface.Cacreole.parameters attribute), 53
- Metallicity, 141
- metallicity (amuse.community.bse.interface.BSE.parameters attribute), 49
- metallicity (amuse.community.sse.interface.SSE.parameters attribute), 48
- min() (amuse.units.quantities.VectorQuantity method), 75
- min_gas_hsmooth_fractional (amuse.community.gadget2.interface.Gadget2.parameters attribute), 59
- min_gas_part_mass (amuse.community.fi.interface.Fi.parameters attribute), 56
- min_gas_temp (amuse.community.gadget2.interface.Gadget2.parameters attribute), 59

- min_size_timestep (amuse.community.gadget2.interface.Gadget2.parameters attribute), 59
- min_step (amuse.community.higpus.interface.HiGPUs.parameters attribute), 47
- minimum_part_per_bin (amuse.community.fi.interface.Fi.parameters attribute), 55
- mmc (class in amuse.community.mmc.interface), 46
- Module, 141
- Molecular Cloud, 141
- move_to_center() (in module amuse.datamodel.particle_attributes), 99
- mpi_collect_stopping_conditions (C function), 122
- mpi_distribute_stopping_conditions (C function), 122
- mpi_setup_stopping_conditions (C function), 122
- mul_unit (class in amuse.units.core), 79
- ## N
- n() (amuse.support.core.print_out method), 30
- n_gpu (amuse.community.higpus.interface.HiGPUs.parameters attribute), 47
- n_Print (amuse.community.higpus.interface.HiGPUs.parameters attribute), 47
- n_smooth (amuse.community.fi.interface.Fi.parameters attribute), 55
- n_smooth (amuse.community.gadget2.interface.Gadget2.parameters attribute), 58
- n_smooth_tol (amuse.community.fi.interface.Fi.parameters attribute), 56
- n_smooth_tol (amuse.community.gadget2.interface.Gadget2.parameters attribute), 58
- name_of_the_indexing_parameter (amuse.datamodel.incode_storage.ParticleMappingMethod attribute), 107
- named_unit (class in amuse.units.core), 79
- names_to_index (amuse.datamodel.incode_storage.ParticleSetAttributesMethod attribute), 108
- ncrit_for_tree (amuse.community.bhtree.interface.BHTree.parameters attribute), 37
- neutron_star_mass_flag (amuse.community.bse.interface.BSE.parameters attribute), 50
- neutron_star_mass_flag (amuse.community.sse.interface.SSE.parameters attribute), 48
- new_particle (amuse.community.interface.gd.GravitationalDynamicsInterface attribute), 5
- new_quantity() (amuse.units.core.unit method), 81
- NewParticleMethod (class in amuse.datamodel.incode_storage), 107
- next_index_for_stopping_condition (C function), 121
- nn_tol (amuse.community.fi.interface.Fi.parameters attribute), 55
- no_gravity_flag (amuse.community.gadget2.interface.Gadget2.parameters attribute), 57
- nonnumeric_unit (class in amuse.units.core), 79
- NonNumericQuantity (class in amuse.units.quantities), 76
- ordered_matter_fraction (amuse.community.bse.interface.BSE.parameters attribute), 50
- NUMBER_OF_STEPS_BITMAP (built-in variable), 120
- NUMBER_OF_STEPS_DETECTION (built-in variable), 121
- nx (amuse.community.capeole.interface.Capeole.parameters attribute), 52
- ny (amuse.community.capeole.interface.Capeole.parameters attribute), 52
- nz (amuse.community.capeole.interface.Capeole.parameters attribute), 52
- ## O
- Octgrav (class in amuse.community.octgrav.interface), 42
- omega_baryon (amuse.community.gadget2.interface.Gadget2.parameters attribute), 59
- omega_lambda (amuse.community.gadget2.interface.Gadget2.parameters attribute), 59
- omega_zero (amuse.community.gadget2.interface.Gadget2.parameters attribute), 59
- opening_angle (amuse.community.bhtree.interface.BHTree.parameters attribute), 37
- opening_angle (amuse.community.fi.interface.Fi.parameters attribute), 55
- opening_angle (amuse.community.gadget2.interface.Gadget2.parameters attribute), 58
- opening_angle (amuse.community.octgrav.interface.Octgrav.parameters attribute), 42
- optical_depth (amuse.community.fi.interface.Fi.parameters attribute), 56
- option (class in amuse.support.options), 128
- optional_attribute_names (amuse.datamodel.incode_storage.ParticleSetAttributesMethod attribute), 108
- OptionalAttributes (class in amuse.support.options), 128
- OrderedDictionary (class in amuse.support.core), 30
- OUT (amuse.rfi.core.LegacyFunctionSpecification attribute), 33
- output_interval (amuse.community.fi.interface.Fi.parameters attribute), 55
- output_path_name (amuse.community.higpus.interface.HiGPUs.parameters attribute), 47
- ## P
- PAIR_DETECTION (built-in variable), 121
- PAIR_DETECTION_BITMAP (built-in variable), 120
- pair_factor (amuse.community.hermite0.interface.Hermite.parameters attribute), 38
- parameters (amuse.community.athena.interface.Athena attribute), 52
- parameters (amuse.community.bhtree.interface.BHTree attribute), 37
- parameters (amuse.community.bse.interface.BSE attribute), 49

parameters (amuse.community.capreole.interface.Capreole.periodic_boundaries_flag attribute), 52
 parameters (amuse.community.evtwin.interface.EVtwin attribute), 51
 parameters (amuse.community.fi.interface.Fi attribute), 53
 parameters (amuse.community.gadget2.interface.Gadget2.periodic_box_size (amuse.community.fi.interface.Fi.parameters attribute), 57
 parameters (amuse.community.gadget2.interface.Gadget2.periodic_boundaries_flag (amuse.community.gadget2.interface.Gadget2.parameters attribute), 59
 parameters (amuse.community.gadget2.interface.Gadget2.periodic_box_size (amuse.community.gadget2.interface.Gadget2.parameters attribute), 55
 parameters (amuse.community.hermite0.interface.Hermite.periodic_box_size (amuse.community.gadget2.interface.Gadget2.parameters attribute), 59
 parameters (amuse.community.higpus.interface.HiGPUs.PhiGRAPE (class in amuse.community.phiGRAPE.interface), 47
 parameters (amuse.community.mercury.interface.Mercury.WayWard40 Physical Domain, 141
 parameters (amuse.community.mesa.interface.MESA.polytropic_index_gamma (amuse.community.gadget2.interface.Gadget2.parameters attribute), 58
 parameters (amuse.community.octgrav.interface.Octgrav.potential_energy() (in module amuse.datamodel.particle_attributes), 100
 parameters (amuse.community.phiGRAPE.interface.PhiGRAPE.potential_energy_in_field() (in module amuse.datamodel.particle_attributes), 100
 parameters (amuse.community.simplex.interface.SimpleX.pow_unit (class in amuse.units.core), 80
 parameters (amuse.community.sse.interface.SSE.prepend() (amuse.units.quantities.VectorQuantity method), 75
 Particle (class in amuse.datamodel), 93
 particle_potential() (in module amuse.datamodel.particle_attributes), 99
 particle_specific_kinetic_energy() (in module amuse.datamodel.particle_attributes), 99
 ParticleGetAttributesMethod (class in amuse.datamodel.incode_storage), 107
 ParticleGetIndexMethod (class in amuse.datamodel.incode_storage), 107
 ParticleMappingMethod (class in amuse.datamodel.incode_storage), 107
 ParticleMethod (class in amuse.datamodel.incode_storage), 107
 ParticleQueryMethod (class in amuse.datamodel.incode_storage), 108
 Particles (class in amuse.datamodel), 91
 ParticlesAddedUpdateMethod (class in amuse.datamodel.incode_storage), 109
 particleset_potential() (in module amuse.datamodel.particle_attributes), 99
 ParticleSetAttributesMethod (class in amuse.datamodel.incode_storage), 108
 ParticleSetSelectSubsetMethod (class in amuse.datamodel.incode_storage), 108
 ParticleSpecificSelectMethod (class in amuse.datamodel.incode_storage), 109
 ParticleSpecificSelectSubsetMethod (class in amuse.datamodel.incode_storage), 109
 ParticlesSubset (class in amuse.datamodel), 92
 ParticlesWithUnitsConverted (class in amuse.datamodel), 92
 ParticlesWithUnitsConverted.ConverterInterface (class in amuse.datamodel), 92
 periodic_boundaries_flag (amuse.community.fi.interface.Fi.parameters attribute), 57
 periodic_boundaries_flag (amuse.community.gadget2.interface.Gadget2.parameters attribute), 59
 periodic_box_size (amuse.community.gadget2.interface.Gadget2.parameters attribute), 55
 periodic_box_size (amuse.community.gadget2.interface.Gadget2.parameters attribute), 59
 PhiGRAPE (class in amuse.community.phiGRAPE.interface), 47
 Physical Domain, 141
 polytropic_index_gamma (amuse.community.gadget2.interface.Gadget2.parameters attribute), 58
 potential_energy() (in module amuse.datamodel.particle_attributes), 100
 potential_energy_in_field() (in module amuse.datamodel.particle_attributes), 100
 pow_unit (class in amuse.units.core), 80
 prepend() (amuse.units.quantities.VectorQuantity method), 75
 print_out (class in amuse.support.core), 29
 prod() (amuse.units.quantities.VectorQuantity method), 75
 Production Code, 141
 PythonCodeInterface (class in amuse.rfi.core), 34

Q

quadrupole_moments_flag (amuse.community.fi.interface.Fi.parameters attribute), 54
 Quantity (class in amuse.units.quantities), 71

R

r_core_plummer (amuse.community.higpus.interface.HiGPUs.parameters attribute), 47
 radiation_flag (amuse.community.fi.interface.Fi.parameters attribute), 54
 read_set_from_file() (in module amuse.io), 64
 redirection (amuse.rfi.core.CodeInterface attribute), 33
 register() (amuse.io.FileFormatProcessor class method), 125
 reimers_mass_loss_coefficient (amuse.community.bse.interface.BSE.parameters attribute), 49
 reimers_mass_loss_coefficient (amuse.community.sse.interface.SSE.parameters attribute), 48
 remove_particle() (amuse.datamodel.AbstractParticleSet method), 89
 remove_particles() (amuse.datamodel.AbstractParticleSet method), 89
 remove_particles_from_store() (amuse.datamodel.incode_storage.InCodeAttributeStorage method), 106

- remove_particles_from_store()
(amuse.datamodel.incode_storage.InCodeGridAttributesStorage method), 107
- remove_particles_from_store()
(amuse.datamodel.ParticlesSubset method), 92
- ReportTable (class in amuse.io), 69
- reset_stopping_conditions (C function), 121
- reversed() (amuse.datamodel.AbstractParticleSet method), 90
- Roche_angular_momentum_factor
(amuse.community.bse.interface.BSE.parameters attribute), 50
- Run, 141
- ## S
- ScalarQuantity (class in amuse.units.quantities), 73
- scale_to_standard() (in module amuse.datamodel.particle_attributes), 100
- select() (amuse.datamodel.AbstractParticleSet method), 90
- select_array() (amuse.datamodel.AbstractParticleSet method), 90
- select_getters_for() (amuse.datamodel.incode_storage.AbstractInCodeAttributesStorage method), 106
- select_setters_for() (amuse.datamodel.incode_storage.AbstractInCodeAttributesStorage method), 106
- self_gravity_flag (amuse.community.fi.interface.Fi.parameters attribute), 54
- Set, 141
- set_acceleration (amuse.community.interface.gd.GravitationalDynamicsInterface attribute), 11
- set_attribute_values() (amuse.datamodel.incode_storage.ParticleSetAttributesStorage method), 108
- set_boundary() (amuse.community.interface.hydro.HydrodynamicsInterface method), 18
- set_conditions (C variable), 121
- set_eps2 (amuse.community.interface.gd.GravitationalDynamicsInterface attribute), 5
- set_example_parameter
(amuse.community.interface.example.ExampleInterface attribute), 2
- set_grid_density (amuse.community.interface.hydro.HydrodynamicsInterface attribute), 20
- set_grid_energy_density
(amuse.community.interface.hydro.HydrodynamicsInterface attribute), 21
- set_grid_momentum_density
(amuse.community.interface.hydro.HydrodynamicsInterface attribute), 21
- set_grid_state (amuse.community.interface.hydro.HydrodynamicsInterface attribute), 19
- set_mass (amuse.community.interface.gd.GravitationalDynamicsInterface attribute), 9
- set_position (amuse.community.interface.gd.GravitationalDynamicsInterface attribute), 10
- set_state (amuse.community.interface.gd.GravitationalDynamicsInterface attribute), 8
- set_stopping_condition_info (C function), 122
- set_stopping_condition_maximum_density_parameter
(amuse.support.codes.stopping_conditions.StoppingConditionInfo attribute), 118
- set_stopping_condition_maximum_internal_energy_parameter
(amuse.support.codes.stopping_conditions.StoppingConditionInfo attribute), 119
- set_stopping_condition_minimum_density_parameter
(amuse.support.codes.stopping_conditions.StoppingConditionInfo attribute), 119
- set_stopping_condition_minimum_internal_energy_parameter
(amuse.support.codes.stopping_conditions.StoppingConditionInfo attribute), 119
- set_stopping_condition_number_of_steps_parameter
(amuse.support.codes.stopping_conditions.StoppingConditionInfo attribute), 119
- set_stopping_condition_out_of_box_parameter
(amuse.support.codes.stopping_conditions.StoppingConditionInfo attribute), 119
- set_stopping_condition_particle_index (C function), 122
- set_stopping_condition_timeout_parameter
(amuse.support.codes.stopping_conditions.StoppingConditionInfo attribute), 119
- set_values_in_store() (amuse.datamodel.incode_storage.InCodeAttributesStorage method), 106
- set_values_in_store() (amuse.datamodel.incode_storage.InCodeGridAttributesStorage method), 107
- setup_mesh() (amuse.community.interface.hydro.HydrodynamicsInterface method), 18
- SimpleX (class in amuse.community.simplex.interface), 60
- smooths_input_flag (amuse.community.fi.interface.Fi.parameters attribute), 54
- SN_kick_random_seed
(amuse.community.bse.interface.BSE.parameters attribute), 50
- SN_kick_speed_dispersion
(amuse.community.bse.interface.BSE.parameters attribute), 50
- SN_kick_speed_dispersion
(amuse.community.sse.interface.SSE.parameters attribute), 48
- softening (amuse.community.higpus.interface.HiGPUs.parameters attribute), 47
- softening_gas_max_phys
(amuse.community.gadget2.interface.Gadget2.parameters attribute), 59
- softening_halo_max_phys
(amuse.community.gadget2.interface.Gadget2.parameters attribute), 59
- sorted() (amuse.units.quantities.VectorQuantity method), 75
- sorted_by_attribute() (amuse.datamodel.AbstractParticleSet method), 90
- sorted_by_attributes() (amuse.datamodel.AbstractParticleSet method), 91

sorted_with() (amuse.units.quantities.VectorQuantity method), 76

specific_kinetic_energy() (in module amuse.datamodel.particle_attributes), 101

SPH, smoothed particle hydrodynamics, 141

sph_artificial_viscosity_eps (amuse.community.fi.interface.Fi.parameters attribute), 56

sph_dens_init_flag (amuse.community.fi.interface.Fi.parameters attribute), 54

sph_h_const (amuse.community.fi.interface.Fi.parameters attribute), 56

sph_viscosity (amuse.community.fi.interface.Fi.parameters attribute), 56

sqrt() (amuse.units.quantities.Quantity method), 72

sqrt_timestep_crit_constant (amuse.community.fi.interface.Fi.parameters attribute), 55

square_root_timestep_flag (amuse.community.fi.interface.Fi.parameters attribute), 54

SSE (class in amuse.community.sse.interface), 48

Star Cluster, 141

star_form_delay_fac (amuse.community.fi.interface.Fi.parameters attribute), 56

star_form_eff (amuse.community.fi.interface.Fi.parameters attribute), 56

star_form_mass_crit (amuse.community.fi.interface.Fi.parameters attribute), 56

star_formation_flag (amuse.community.fi.interface.Fi.parameters attribute), 54

star_formation_mode (amuse.community.fi.interface.Fi.parameters attribute), 56

start_time (amuse.community.higpus.interface.HiGPUs.parameters attribute), 47

StellarEvolution (class in amuse.community.interface.se), 16

Step, 141

stop_interfaces() (in module amuse.rfi.core), 34

stopping_condition_maximum_density (amuse.community.bhtree.interface.BHTree.parameters attribute), 37

stopping_condition_maximum_density (amuse.community.fi.interface.Fi.parameters attribute), 57

stopping_condition_maximum_density (amuse.community.gadget2.interface.Gadget2.parameters attribute), 60

stopping_condition_maximum_density (amuse.community.hermite0.interface.Hermite.parameters attribute), 38

stopping_condition_maximum_density (amuse.community.mercury.interface.MercuryWayWard.parameters attribute), 44

stopping_condition_maximum_density (amuse.community.octgrav.interface.Octgrav.parameters attribute), 42

stopping_condition_maximum_density (amuse.community.phiGRAPE.interface.PhiGRAPE.parameters attribute), 40

stopping_condition_maximum_density (amuse.community.bhtree.interface.BHTree.parameters attribute), 37

stopping_condition_maximum_density (amuse.community.fi.interface.Fi.parameters attribute), 57

stopping_condition_maximum_density (amuse.community.gadget2.interface.Gadget2.parameters attribute), 60

stopping_condition_maximum_density (amuse.community.hermite0.interface.Hermite.parameters attribute), 39

stopping_condition_maximum_density (amuse.community.phiGRAPE.interface.PhiGRAPE.parameters attribute), 40

stopping_condition_maximum_internal_energy (amuse.community.bhtree.interface.BHTree.parameters attribute), 37

stopping_condition_maximum_internal_energy (amuse.community.fi.interface.Fi.parameters attribute), 57

stopping_condition_maximum_internal_energy (amuse.community.gadget2.interface.Gadget2.parameters attribute), 60

stopping_condition_maximum_internal_energy (amuse.community.hermite0.interface.Hermite.parameters attribute), 39

stopping_condition_maximum_internal_energy (amuse.community.mercury.interface.MercuryWayWard.parameters attribute), 44

stopping_condition_maximum_internal_energy (amuse.community.octgrav.interface.Octgrav.parameters attribute), 42

stopping_condition_maximum_internal_energy (amuse.community.phiGRAPE.interface.PhiGRAPE.parameters attribute), 40

stopping_condition_minimum_density (amuse.community.bhtree.interface.BHTree.parameters attribute), 37

stopping_condition_minimum_density (amuse.community.fi.interface.Fi.parameters attribute), 57

stopping_condition_minimum_density (amuse.community.gadget2.interface.Gadget2.parameters attribute), 60

stopping_condition_minimum_density (amuse.community.hermite0.interface.Hermite.parameters attribute), 38

stopping_condition_minimum_density (amuse.community.mercury.interface.MercuryWayWard.parameters attribute), 44

stopping_condition_minimum_density (amuse.community.octgrav.interface.Octgrav.parameters attribute), 42

stopping_condition_minimum_density (amuse.community.phiGRAPE.interface.PhiGRAPE.parameters attribute), 40

stopping_condition_minimum_internal_energy (amuse.community.bhtree.interface.BHTree.parameters attribute), 37

stopping_condition_minimum_internal_energy (amuse.community.fi.interface.Fi.parameters attribute), 57

stopping_condition_minimum_internal_energy (amuse.community.gadget2.interface.Gadget2.parameters attribute), 60

stopping_condition_minimum_internal_energy (amuse.community.hermite0.interface.Hermite.parameters attribute), 39

stopping_condition_minimum_internal_energy (amuse.community.mercury.interface.MercuryWayWard.parameters attribute), 44
 stopping_condition_minimum_internal_energy (amuse.community.octgrav.interface.Octgrav.parameters attribute), 42
 stopping_condition_minimum_internal_energy (amuse.community.phiGRAPE.interface.PhiGRAPE.parameters attribute), 40
 stopping_conditions_number_of_steps (amuse.community.bhtree.interface.BHTree.parameters attribute), 37
 stopping_conditions_number_of_steps (amuse.community.fi.interface.Fi.parameters attribute), 57
 stopping_conditions_number_of_steps (amuse.community.gadget2.interface.Gadget2.parameters attribute), 60
 stopping_conditions_number_of_steps (amuse.community.hermite0.interface.Hermite.parameters attribute), 38
 stopping_conditions_number_of_steps (amuse.community.mercury.interface.MercuryWayWard.parameters attribute), 44
 stopping_conditions_number_of_steps (amuse.community.octgrav.interface.Octgrav.parameters attribute), 42
 stopping_conditions_number_of_steps (amuse.community.phiGRAPE.interface.PhiGRAPE.parameters attribute), 40
 stopping_conditions_number_of_steps StoppingConditionInterface (class in amuse.support.codes.stopping_conditions), 114
 stopping_conditions_number_of_steps storage_shape() (amuse.datamodel.incode_storage.InCodeGridAttributes method), 107
 stopping_conditions_number_of_steps store() (amuse.io.FileFormatProcessor method), 125
 stopping_conditions_number_of_steps store_string() (amuse.io.FullTextFileFormatProcessor method), 126
 stopping_conditions_number_of_steps string (amuse.support.core.print_out attribute), 30
 stopping_conditions_number_of_steps string_unit (class in amuse.units.core), 80
 stopping_conditions_number_of_steps sum() (amuse.units.quantities.VectorQuantity method), 76
 stopping_conditions_number_of_steps supernova_duration (amuse.community.fi.interface.Fi.parameters attribute), 56
 stopping_conditions_number_of_steps supernova_eff (amuse.community.fi.interface.Fi.parameters attribute), 56
 stopping_conditions_out_of_box_size (amuse.community.bhtree.interface.BHTree.parameters attribute), 37
 stopping_conditions_out_of_box_size (amuse.community.fi.interface.Fi.parameters attribute), 57
 stopping_conditions_out_of_box_size (amuse.community.gadget2.interface.Gadget2.parameters attribute), 60
 stopping_conditions_out_of_box_size (amuse.community.hermite0.interface.Hermite.parameters attribute), 38
 stopping_conditions_out_of_box_size (amuse.community.mercury.interface.MercuryWayWard.parameters attribute), 44
 stopping_conditions_out_of_box_size (amuse.community.octgrav.interface.Octgrav.parameters attribute), 42
 stopping_conditions_out_of_box_size (amuse.community.phiGRAPE.interface.PhiGRAPE.parameters attribute), 40
 stopping_conditions_timeout (amuse.community.bhtree.interface.BHTree.parameters attribute), 37
 stopping_conditions_timeout (amuse.community.fi.interface.Fi.parameters attribute), 57
 stopping_conditions_timeout (amuse.community.gadget2.interface.Gadget2.parameters attribute), 60
 stopping_conditions_timeout (amuse.community.hermite0.interface.Hermite.parameters attribute), 38
 stopping_conditions_timeout (amuse.community.mercury.interface.MercuryWayWard.parameters attribute), 44
 stopping_conditions_timeout (amuse.community.octgrav.interface.Octgrav.parameters attribute), 42
 stopping_conditions_timeout (amuse.community.phiGRAPE.interface.PhiGRAPE.parameters attribute), 40
 stopping_conditions_timeout time_between_statistics (amuse.community.gadget2.interface.Gadget2.parameters attribute), 59
 stopping_conditions_timeout time_limit_cpu (amuse.community.gadget2.interface.Gadget2.parameters attribute), 58
 stopping_conditions_timeout time_max (amuse.community.gadget2.interface.Gadget2.parameters attribute), 59
 stopping_conditions_timeout (amuse.community.gadget2.interface.Gadget2.parameters attribute), 60
 stopping_conditions_timeout (amuse.community.hermite0.interface.Hermite.parameters attribute), 38
 stopping_conditions_timeout (amuse.community.mercury.interface.MercuryWayWard.parameters attribute), 44
 stopping_conditions_timeout (amuse.community.octgrav.interface.Octgrav.parameters attribute), 42
 stopping_conditions_timeout (amuse.community.phiGRAPE.interface.PhiGRAPE.parameters attribute), 40
 stopping_conditions_timeout (amuse.support.codes.stopping_conditions), 114
 storage_shape() (amuse.datamodel.incode_storage.InCodeGridAttributes method), 107
 store() (amuse.io.FileFormatProcessor method), 125
 store_string() (amuse.io.FullTextFileFormatProcessor method), 126
 string (amuse.support.core.print_out attribute), 30
 string_unit (class in amuse.units.core), 80
 sum() (amuse.units.quantities.VectorQuantity method), 76
 supernova_duration (amuse.community.fi.interface.Fi.parameters attribute), 56
 supernova_eff (amuse.community.fi.interface.Fi.parameters attribute), 56
 supported_conditions (C variable), 121
 synchronize_to() (amuse.datamodel.AbstractParticleSet method), 91

T

t_supernova_start (amuse.community.fi.interface.Fi.parameters attribute), 56
 targetnn (amuse.community.fi.interface.Fi.parameters attribute), 55
 thermal_energy() (in module amuse.datamodel.particle_attributes), 101
 Threads (amuse.community.higpus.interface.HiGPUs.parameters attribute), 47
 tidal_circularisation_flag (amuse.community.bse.interface.BSE.parameters attribute), 49
 time (amuse.community.hermite0.interface.Hermite.parameters attribute), 38
 time_begin (amuse.community.gadget2.interface.Gadget2.parameters attribute), 59
 time_between_statistics (amuse.community.gadget2.interface.Gadget2.parameters attribute), 59
 time_limit_cpu (amuse.community.gadget2.interface.Gadget2.parameters attribute), 58
 time_max (amuse.community.gadget2.interface.Gadget2.parameters attribute), 59

`zeta_cr_ion_rate` (`amuse.community.fi.interface.Fi.parameters`
attribute), [56](#)