Tutorials

Release 8.0

The AMUSE Team

May 23, 2013

Contents

1 Working with Units

The AMUSE framework provides a unit handling library. This library is used troughout the AMUSE framework. When interacting with a code all data has a unit, even scaled systems have units.

2 Quantities

The basic data object is a quantity, a quantity is made up of a value and a unit. The value can be a single number (a scalar quantity) or a multi-dimensional array (a vector quantity).

Quantities are created by the `'|(bar)'` operator. All quantities can be used like numbers and a lot of numpy functions also work on quantities.

```
>>> from amuse.units.si import *
>>> from amuse.units.core import named_unit
>>>
>>> weigth = 80 | kg
>>> persons = 10
>>> print "Total weight: ", persons * weigth
Total weight:  800 kg

>>> day = named_unit("day", "d", s * 60 * 60 * 24 )
>>> weigth_loss = (0.1 | kg) / (1 | day)
>>> print "Weigth loss: ", weigth_loss
Weigth loss:  0.1 1.15740740741e-05 * kg * s**-1
>>> print "Weigth loss: ", weigth_loss.as_quantity_in(kg/day)
Weigth loss:  0.1 kg / d
```

3 Working with arrays

A vector quantity can be used like a python list. Take care to only put quantities into a vector quantity.

```
>>> from amuse.units.units import MSun
>>>
>>> masses = [] | MSun
>>> for i in range(10):
...     masses.append(i**2 | MSun)
>>> print "Masses:", masses
Masses: [0.0, 1.0, 4.0, 9.0, 16.0, 25.0, 36.0, 49.0, 64.0, 81.0] MSun
```

Note: When working with arrays, some care must be taken to ensure that vector quantities are created and not arrays of quantities. The following code will create an array of quantities

```
>>> from amuse.units.units import MSun
>>>
>>> masses = []
>>> for i in range(2):
...     masses.append(i**2 | MSun)
>>> print "Masses:", masses
Masses: [quantity<0 MSun>, quantity<1 MSun>]
```

---

4 Estimating the typical mass of the most massive star in a cluster

In this tutorial we will estimate the typical mass of the most massive star for a cluster with a Salpeter initial mass function (IMF). This tutorial also illustrates that the units won't get in the way of calculations.

First we import numpy and set the seed of its random number generator (RNG). All random numbers used in AMUSE are drawn from this RNG. Seeding it is of course not necessary, but it will make the results reproducible.

```
>>> import numpy
>>> numpy.random.seed(123456)
```

We will also need units and the Salpeter IMF generator. The easiest way to import almost everything from the AMUSE toolbox is to do:

```
>>> from amuse.lab import *
```

but here we will import the required module and function manually for clarity:

```
>>> from amuse.units import units
>>> from amuse.ext.salpeter import new_salpeter_mass_distribution
```

Now, we can calculate the maximum stellar mass for each of *number_of_cluster_realizations* realizations of an *number_of_stars* star cluster. The Salpeter IMF runs from 0.1 to 125 solar mass with a slope of -2.35, by default. Suppose the maximum of 125 solar mass is a bit too high for our taste, so we set it to 100 solar mass.

```
>>> number_of_stars = 1000
>>> number_of_cluster_realizations = 100
>>> maxmasses = [] | units.MSun
>>> for i in range(number_of_cluster_realizations):
...     maxmasses.append(max(new_salpeter_mass_distribution(
...         number_of_stars,
...         mass_max = 100. | units.MSun
...     )))
...
```

Note that the way we initialize *maxmasses*, with the solar mass unit, forces it to be a VectorQuantity instead of a Python list of ScalarQuantities. This is always recommended, because it is much faster, and it will make sure that AMUSE always recognizes it as a Quantity.

If we want to know the mean mass of the *maxmasses* VectorQuantity, we simply use the *mean* function of VectorQuantities:

```
>>> print "mean:  ", maxmasses.mean()
mean:   27.4915750164 MSun
```

The same works for the median or the standard deviation of *maxmasses*.

```
>>> print "median:", maxmasses.median()
>>> print "stddev:", maxmasses.std()
median: 21.0983403429 MSun
stddev: 19.7149800906 MSun
```

Slightly slower, but giving the same result, we can use the numpy functions:

```
>>> print "mean:  ", numpy.mean( maxmasses)
>>> print "median:", numpy.median(maxmasses)
>>> print "stddev:", numpy.std(maxmasses)
mean:   27.4915750164 MSun
median: 21.0983403429 1.98892e+30 * kg
stddev: 19.7149800906 MSun
```

Something weird has happened to the unit of the median mass. The result is still correct but the unit is converted to SI units. This is usually caused by a multiplication of a Quantity, where AMUSE tries to simplify the result, cancelling out for example factors of kg / kg. There's no need to bother, but if it annoys you, it can easily be fixed by:

```
>>> print "median:", numpy.median(maxmasses).in_(units.MSun)
median: 21.0983403429 MSun
```

5 Working with Particle Sets

5.1 There are Sets, Subsets, Particles, Children and Parents...

The fundamental data structures used in AMUSE are particle sets. Based on attributes of the elements in the sets (particles), selections can be made using the selection method which return subsets. These subsets are views or scopes on the set and do not hold values of their own.

It is also possible to *add* structure to the set by defining parent-child relationships between **particles in a set**. These structures exist only in the set and are a property of it and have no meaning with respect to the particles outside the set.

5.2 Selecting stars in a plummer model

In this tutorial we generate a set of stars, a plummer model, and two subsets of stars. The first subset contains the stars within a certain radius from the center of mass of the plummer system and the second subset contains all the other stars.

The plummer module generates a particle set of the form data.Stars(N).

Imports

We start with some AMUSE imports in a python shell: The core module contains the set objects, and the units and nbody_system are needed for the correct assignment of properties to the particles, which **need to be quantities**.

```
>>> from amuse.support import data
>>> from amuse.units import nbody_system
>>> from amuse.units import units
>>> from amuse.ic.plummer import new_plummer_sphere
>>> import numpy as np
```

Model

Note: A quick way to look at the sets we are going to make is by using gnuplot. If you have gnuplot, you can install the **gnuplot-py** package to control gnuplot directly from your script.

To install **gnuplot-py**, open a shell and do:

```
easy_install gnuplot-py
```

Let's generate a plummer model consisting of 1000 stars,

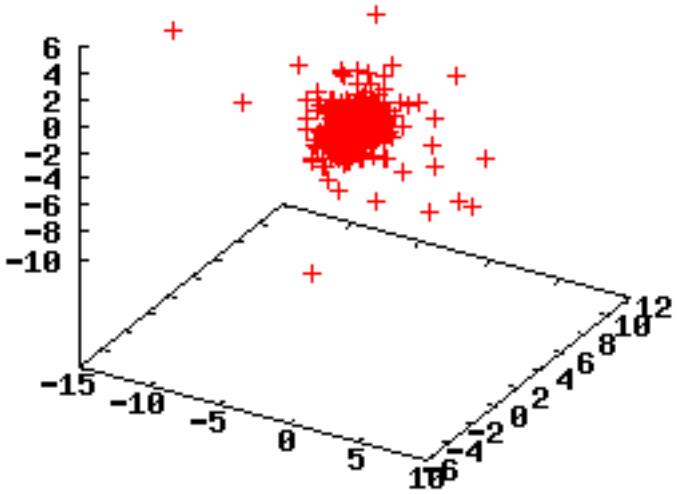
```
>>> convert_nbody = nbody_system.nbody_to_si(100.0 | units.MSun, 1 | units.parsec)
>>> plummer = new_plummer_sphere(1000, convert_nbody)
```

We can work on the new **plummer** particle set, but we want to keep this set unchanged for now. So, we copy all data to a working set:

```
>>> stars = plummer.copy()
```

Note: To look at the stars in gnuplot do:

```
>>> plotter = Gnuplot.Gnuplot()
>>> plotter.splot(plummer.position.value_in(units.parsec))
```



Selection

At this stage we select the subsets based on the distance of the individual stars with respect to the center of mass, being **1 parsec** in this example.

We need the center of mass, of course:

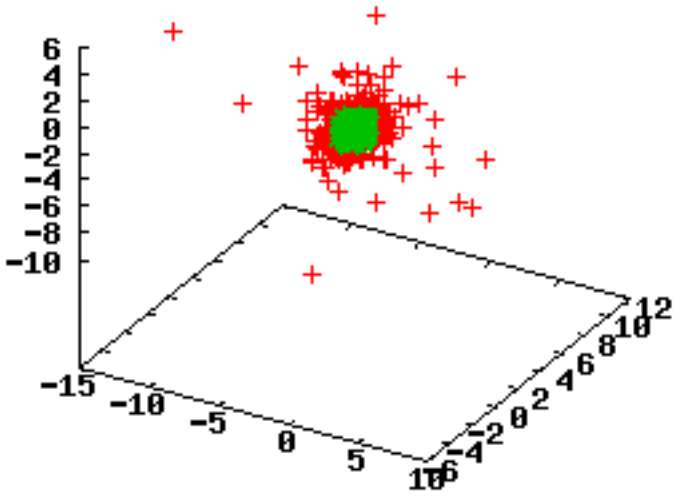
```
>>> center_of_mass = stars.center_of_mass()
```

and the selection of the sets:

```
>>> innersphere = stars.select(lambda r: (center_of_mass-r).length()<1.0 | units.parsec,["position
>>> outersphere = stars.select(lambda r: (center_of_mass-r).length()>=1.0 | units.parsec,["positi
```

Note: To look at the stars in gnuplot do:

```
>>> plotter = Gnuplot.Gnuplot()
>>> plotter.splot(outersphere.position.value_in(units.parsec), innersphere.position.value_in(units
```



We can achieve the same result in another way by using the fact that the outersphere is the difference of the innersphere set and the stars set:

```
>>> outersphere_alt = stars.difference(innersphere)
```

or using the particle subtraction '-' operator:

```
>>> outersphere_alt2 = stars - innersphere
```

The selections are all subsets as we can verify:

```
>>> outersphere
<amuse.datamodel.particles.ParticlesSubset object at ...>
```

Set operations

The result should be the same, but we'll check:

```
>>> hopefuly_empty = outersphere.difference(outersphere_alt)
>>> hopefuly_empty.is_empty()
True
>>> len(outersphere - outersphere_alt2)==0
True
```

From our selection criteria we would expect to have selected all stars, to check this we could do something like this:

```
>>> len(innersphere)+len(outersphere) == 1000
True
>>> len(innersphere)+len(outersphere_alt) == 1000
True
>>> len(innersphere)+len(outersphere_alt2) == 1000
True
```

The union of the innersphere and outersphere set should give the stars set, we can check:

```
>>> like_stars = innersphere.union(outersphere)
>>> stars.difference(like_stars).is_empty()
True
>>> (innersphere + outersphere_alt2 -stars).is_empty()
True
```

Iteration

We can iterate over sets and we will put that to use here to check whether we really selected the right stars in the outersphere:

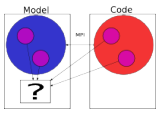
```
>>> should_not_be_there_stars = 0
>>> for star in outersphere:
...     if (center_of_mass-star.position).length()<1.0|units.parsec:
...         should_not_be_there_stars += 1
>>> should_not_be_there_stars
0
```

Indexation

Not very set like, but can-do.

5.3 Using codes

Channels



Imagine we evaluate stars in some MPI bound legacy code, and want to know what happened to the stars in our subsets. Each query for attributes in de code set invokes one MPI call, which is inefficient if we have many queries. Copying the entire set to the model, however, costs only one MPI call too and we can query from the model set at will without the MPI overhead. Copying the data from one existing set to another set *or* subset can be done via channels.

First we define a simple dummy legacy code object, with some typical gd interface methods:

```
>>> class DummyLegacyCode(object):
...     def __init__(self):
...         self.particles = data.Particles()
...     def add_particles(self, new_particles):
...         self.particles.add_particles(new_particles)
...     def update_particles(self, particles):
...         self.particles.copy_values_of_state_attributes_to(particles)
...         particles = self.particles.copy()
...     def evolve(self):
...         self.particles.position *= 1.1
```

We instantiate the code and use it to evolve (expand) our plummer model. We add a channel to our innersphere subset to track the changes:

```
>>> code = DummyLegacyCode()
>>> channel_to_innersphere = code.particles.new_channel_to(innersphere)
>>> code.add_particles(stars)
>>> code.evolve()
>>> r_inner_init = innersphere.position
```

```
>>> r_outer_init = outersphere.position
>>> channel_to_innersphere.copy()
>>> r_inner_fini = innersphere.position
>>> r_outer_fini = outersphere.position
```

Checking the changes by looking at the positions (all changed after the evolve), we will see that only the inner-sphere paricles are updated:

```
>>> np.all(r_inner_init[0]==r_inner_fini[0])
False
>>> np.all(r_outer_init[0]==r_outer_fini[0])
True
```

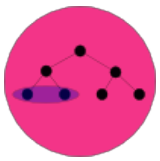
If we want to update all particles in our model, we can use:

```
>>> code.update_particles(stars)
```

and check:

```
>>> r_outer_fini = outersphere.position
>>> np.all(r_outer_init[0] == r_outer_fini[0])
False
```

5.4 Particle Hierarchies



Let us suppose that the zero-th star, stars[0] has a child and a grandchild star who do not belong to the plummer model:

```
>>> child_star = data.Particle()
>>> grandchild_star = data.Particle()
>>> child_star.mass = 0.001|units.MSun
>>> child_star.position = [0,0,0]|units.AU
>>> child_star.velocity = [0,0,0]|units.AUd
>>> grandchild_star.mass = 0.0001|units.MSun
>>> grandchild_star.position = [0,0.1,0]|units.AU
>>> grandchild_star.velocity = [0,0,0]|units.AUd
```

We can add them as child and grandchild etc.. to the set of plummer stars. But first we have to add them to the set as regular stars:

```
>>> child_star_in_set = stars.add_particle(child_star)
>>> grandchild_star_in_set = stars.add_particle(grandchild_star)
```

Now we can define the hierarchy:

```
>>> stars[0].add_child(child_star_in_set)
>>> child_star_in_set.add_child(grandchild_star_in_set)
```

The descendents of star 0 form a subset:

```
>>> stars[0].descendents()
<amuse.datamodel.ParticlesSubset object at ...>
>>> stars[0].children().mass.value_in(units.MSun)
array([ 0.001])
>>> stars[0].descendents().mass
quantity<[1.98892e+26, 1.98892e+27] kg>
```

5.5 Methods to retreive physical properties of the particles set

Particle sets have a number of functions for calculating physical properties that apply to the sets. Although some of these might be implemented in the legacy codes as well, using them via particle sets guarantees the applicability to *all* particles when multiple legacy codes are used. Furthermore, the particle set functions provide a uniform way of doing the calculations. Speed might be the downside.

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

- **smooting_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.

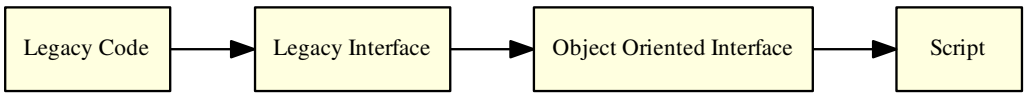
```
>>> from amuse.datamodel import Particles
>>> particles = Particles(2)
>>> particles.x = [0.0, 1.0] | units.m
>>> particles.y = [0.0, 0.0] | units.m
>>> particles.z = [0.0, 0.0] | units.m
>>> particles.mass = [1.0, 1.0] | units.kg
>>> particles[1].potential()
quantity<-6.67428e-11 m**2 * s**-2>
```

6 Integrate a C++ code

In this tutorial we will create an AMUSE interface to a C++ code. We will first define the legacy interface, then implement the code and finally build an object oriented interface on top of the legacy interface.

The legacy code will be a very simple and naive implementation to find 3 nearest neighbors of a particle.

The legacy code interface supports methods that transfer values to and from the code. The values do not have any units and no error handling is provided by the interface. We can add error handling, unit handling and more functions to the legacy interface by defining a subclass of a CodeInterface (this is the objected oriented interface).



The legacy code in this tutorial will be a very simple and naive implementation to find 3 nearest neighbors of a particle.

6.1 Two paths

When defining the interface will walk 2 paths:

1. Management of particles in AMUSE (python)
2. Management of particles in the code (C or Fortran)

The first path makes sense for legacy codes that perform a transformation on the particles, or analyse the particles state or do not store any internal state between function calls (all data is external). For every function of the code, data of every particle is send to the code. If we expect multiple calls, the code would incur a high communication overhead and we are better of choosing path 2.

The second path makes sense for codes that already have management of a particles (or grid) or were we want to call multiple functions of the code and need to send the complete model to code for every function call. The code is first given the data, then calls are made to the code to evolve it's model or perform reduction steps on the data, finally the updated data is retrieved from the code.

6.2 Procedure

The suggested procedure for creating a new interface is as follows:

0. **Legacy Interface.** Start with creating the legacy interface. Define functions on the interface to input and output relevant data. The CodeInterface code depends on the legacy interface code.
1. **Make a Class.** Create a subclass of the CodeInterface class
2. **Define methods.** In the legacy interface we have defined functions with parameters. In the code interface we need to define the units of the parameters and if a parameter or return value is used as an errorcode.
3. **Define properties.** Some functions in the legacy interface can be better described as a property of the code. These are read only variables, like the current model time.
4. **Define parameters.** Some functions in the legacy interface provide access to parameters of the code. Units and default values need to be defined for the parameters in this step
5. **Define sets or grids.** A code usually handles objects or gridpoints with attributes. In this step a generic interface is defined for these objects so that the interopability between codes increases.

6.3 Before we start

This tutorial asumes you have a working amuse environment. Please ensure that amuse is setup correctly by running 'nosetests' in the amuse directory.

Environment variables

To simplify the work in the coming sections, we first define the environment variable 'AMUSE_DIR'. This environment variable must point to the root directory of AMUSE (this is the directory containing the build.py script).

```
> export AMUSE_DIR=<path to the amuse root directory>
```

or in a c shell:

```
> setenv AMUSE_DIR <path to the amuse root directory>
```

After building the code, we want to run and test the code. Check if amuse is available in your python path by running the following code on the command line.

```
> python -c "import amuse"
Traceback (most recent call last):
File "<string>", line 1, in <module>
ImportError: No module named amuse
```

If this code ends in a "ImportError" as shown in the example, the PYTHONPATH environment variable must be extended with the src directory in AMUSE_DIR. We can do so by using one of the following commands.

```
> export PYTHONPATH=${PYTHONPATH}:${AMUSE_DIR}/src
```

or in a c shell:

```
> setenv AMUSE_DIR ${PYTHONPATH}:${AMUSE_DIR}/src
```

The name of our project

We will be writing a code to find the nearest neighbors of a particle, so let's call our project 'NearestNeighbor'.

Creating the initial directory structure

First we need to create a directory for our project and put some files in it to help build the code. The fastest method to setup the directory is by using the build.py script.

```
> $AMUSE_DIR/build.py --type=c --mode=dir NearestNeighbor
```

The script will generate a directory with all the files needed to start our project. It also generates a very small example legacy code with only one function `'echo_int'`. We can build and test our new module:

```
> cd nearestneighbor/
> make all
> $AMUSE_DIR/amuse.sh -c 'from interface import NearestNeighbor; print NearestNeighbor().echo_int
OrderedDictionary({'int_out':10, '__result':0})
> nosetests -v
.
----------------------------------------------------------------------
Ran 1 test in 0.556s

OK
```

Note: The build.py script can be used to generate a range of files. To see what this file can do you can run the script with a `'--help'` parameter, like so:

```
> $AMUSE_DIR/build.py --help
```

6.4 The Legacy Code

Normally the legacy code already exists and our task is limited to defining and implementing an interface so that AMUSE scripts can access the code. For this tutorial we will implement our legacy code.

When a legacy code is integrated all interface code is put in one directory and all the legacy code is put in a **src** directory placed under this directory. The build.py script created a **src** directory for us, and we will put the nearest neighbor algorithm in this directory.

Go to the **src** directory and create a **code.cc** file, open this file in your favorite editor and copy and paste this code into it:

```
#include "code.h"
#include <math.h>

/**
 * return the distance between point0 and point1
 */
double distance_between_points(
    double x0, double y0, double z0,
    double x1, double y1, double z1)
{
    double dx = x1 - x0;
    double dy = y1 - y0;
    double dz = z1 - z0;
    return sqrt(dx * dx  + dy * dy + dz * dz);
```

```
}

class DistanceAndIndex{

public:
    double r;
    int index;

    DistanceAndIndex():r(-1),index(-1) {}
    DistanceAndIndex(DistanceAndIndex & original):r(original.r),index(original.index){}
};

/**
 * Find the nearest neigbors of all the points (specified with
 * x, y and z).
 *
 * Fills the n1, n2 and n3 arrays with the closest, second clostest
 * and third closests points.
 *
 **/
int find_nearest_neighbors(int npoints,
    double * x, double * y, double * z,
    int * n0, int * n1, int * n2)
{
    for(int i = 0; i < npoints; i++)
    {
        DistanceAndIndex found[3];

        for(int j = 0; j < npoints; j++)
        {
            if(j == i) {
                continue;
            }

            double x0 = x[i];
            double y0 = y[i];
            double z0 = z[i];

            double x1 = x[j];
            double y1 = y[j];
            double z1 = z[j];

            double r = distance_between_points(x0, y0, z0, x1, y1 , z1);

            for(int k = 0; k < 3; k++) {
                if(found[k].index == -1) {
                    found[k].index = j;
                    found[k].r = r;
                    break;
                } else {
                    if(r < found[k].r) {
                        for(int l = 2;l > k; l--) {
                            found[l] = found[l-1];
                        }
                        found[k].index = j;
                        found[k].r = r;
                        break;
                    }
                }
            }

        }
```

```
        n0[i] = found[0].index;
        n1[i] = found[1].index;
        n2[i] = found[2].index;
    }


    return 0;
}
```

Note: This algorithm is un-optimized and has N*N order. It is not meant as very efficient code but as a readable example.

Before we can continue we also need to alter the **Makefile** in the **src** directory, so that our **code.cc** file is included in the build. To do so, open an editor on the Makefile and change the line:

```
CODEOBJS = test.o
```

to:

```
CODEOBJS = test.o code.o
```

Test if the code builds. As we have not coupled our algorithm to the interface we (we have not even defined an interface) we do not have any new functionality. In the legacy interface directory (not the **src** directory) do:

```
> make all
> nosetests
.
----------------------------------------------------------------------
Ran 1 test in 0.427s

OK
```

It works, if the test fails for any reason please check that the C++ code is correct and that `worker_code` exists in your directory.

6.5 Path 1

Defining the legacy interface

We will first define a legacy interface so that we can call the **find_nearest_neighbors** function from python. AMUSE can interact with 2 classes of functions:

1. A function with all scalar input and output variables. All variables are simple, non-composite variables (like int or double). For example:

   ```
   int example1(double input, double *output)
   {
       *output = input;
       return 0;
   }
   ```

2. A function with all vector (or list) input and output variables and a length variable. The return value is a scalar value. For example:

   ```
   int example2(double *input, double *output, int n)
   {
       for(int i = 0; i < n; i++)
       {
           output[i] = input[i];
       }
   ```

```
        return 0;
    }
```

If you have functions that don't follow this pattern you need to define a convert function in C++ that provides an interface following one of the two patterns supported by AMUSE.

In our case the **find_nearest_neighbors** complies to pattern 2 and we do not have to write any code in C++ to convert the function to a compliant interface. We only have to specify the function in python. We do so by adding a **find_nearest_neighbors** method to the **NearestNeighborInterface** class in the **interface.py** file. Open and editor on the interfaces.py file and add the following method to the NearestNeighborInterface class:

```
class NearestNeighborInterface(CodeInterface):
    #...

    @legacy_function
    def find_nearest_neighbors():
        function = LegacyFunctionSpecification()
        function.must_handle_array = True
        function.addParameter(
            'npoints',
            dtype='int32',
            direction=function.LENGTH)
        function.addParameter(
            'x',
            dtype='float64',
            direction=function.IN)
        function.addParameter(
            'y',
            dtype='float64',
            direction=function.IN)
        function.addParameter(
            'z',
            dtype='float64',
            direction=function.IN)
        function.addParameter(
            'n0',
            dtype='int32',
            direction=function.OUT)
        function.addParameter(\
            'n1',
            dtype='int32',
            direction=function.OUT)
        function.addParameter(
            'n2',
            dtype='int32',
            direction=function.OUT)
        function.result_type = 'int32'
        return function
```

In the *find_nearest_neighbors* method we specify every parameter of the C++ function and the result type. For each parameter we need to define a name, data type and wether we will input, output (or input and output) data using this parameter. AMUSE knows only a limited amount of data types for parameters: float64, float32, int32, string, bool and int64. We also have a special parameter, with LENGTH as direction. This parameter is needed for all functions that follow pattern 2, it will be filled with the length of the input arrays. We also must specify that the function follows pattern 2 by setting `'function.must_handle_array = True'`.

Save the file and recompile the code.

```
> make clean
> make all
> nosetests
.
----------------------------------------------------------------------
```

```
Ran 1 test in 0.427s

OK
```

It works! But, how do we know the find_nearest_neighbors method really works? Let's write a test and find out. Open an editor on the *test_nearestneighbor.py* file and add the following method to the **NearestNeighborInterfaceTests** class:

```
def test2(self):
    instance = NearestNeighborInterface()
    x = [0.0, 1.0, 4.0, 7.5]
    y = [0.0] * len(x)
    z = [0.0] * len(x)

    n0, n1, n2, error = instance.find_nearest_neighbors(x,y,z)

    self.assertEquals(error[0], 0)
    self.assertEquals(n0, [2,1,2,3])
    self.assertEquals(n1, [3,3,4,2])
    self.assertEquals(n2, [4,4,1,1])

    instance.stop()
```

This test calls the *find_nearest_neighbors* method with 4 positions and checks if the nearest neighbors are determined correctly. Let's run the test, and see if everything is working:

```
> nosetests
..
----------------------------------------------------------------------
Ran 2 test in 0.491s

OK
```

We now have a simple interface that works, but we have to do our own indexing after the call and we could send data of any unit to the method, also we have to do our own error checking after the method. Let's define a object oriented interface to solve these problems

Defining the Object Oriented Interface

The object oriented interface sits on top of the legacy interface. It decorates this interface with sets, unit handling, state engine and more. We start creating the object oriented interfacy by inheriting from CodeInterface and writing the __init__ function. The build script has added this class to the interface.py file for us. Open an editor on interface.py and make sure this code is in the file (at the end of the file):

```
class NearestNeighbor(CodeInterface):

    def __init__(self, **options):
        CodeInterface.__init__(self,  NearestNeighborInterface(), **options)
```

Configuring the handlers

We configure the object oriented interfacy by implementing several methods. The object oriented interface is implement by several "handlers". Each handler provides support for a specific aspect of the interface. AMUSE defines a handler for the unit conversion, a handler for the interfacing with sets of particles, a handler to ensure the methods are called in the right order, etc. Each handler is very generic and needs to be configured before use. The handler are configured using the "Visitor" pattern. The following pseudo-code shows how the handlers are configured

```
class CodeInterface(object):
    #...
```

```
    def configure_handlers(self):
        #...
        for handler in self.get_all_handlers():
            handler.configure(self)

    def define_converter(self, handler):
        """ configure the units converter handler """

        handler.set_nbody_converter(...)

    def define_particle_sets(self, handler):
        """ configure sets of particles """

        handler.define_incode_particle_set(...)
        handler.set_getter(...)

class HandleConvertUnits(AbstractHandler):
    #...

    def configure(self, interface):
        interface.define_converter(self)

class HandleParticles(AbstractHandler):
    #...

    def configure(self, interface):
        interface.define_particle_sets(self)
```

Configuration of the handlers is optional, we only have to define those handler that we need in our interface. In our example we need to configure the "HandleMethodsWithUnits" handler (to define units and error handling) and the "HandleParticles" to define a particle set.

Defining methods with units

We first want to add units and error handling to the **find_nearest_neighbors**. We do this by creating a **define_methods** function on the **NearestNeighbor** class. Open an editor on *interface.py* and add this method to the class:

```
def define_methods(self, handler):

    handler.add_method(
        "find_nearest_neighbors",
        (
            generic_unit_system.length,
            generic_unit_system.length,
            generic_unit_system.length,
        ),
        (
            handler.INDEX,
            handler.INDEX,
            handler.INDEX,
            handler.ERROR_CODE
        )
    )
```

The **add_method** call expects the name of the function in the legacy interface as it's first, next it expects a list of the units of the input parameters and a list of the units of the output parameters. The return value of a function is always the last item in the list of output parameters. We specify a **generic_unit_system.length** unit for the x, y and z parameters. The output parameters are indices and an errorcode. The errorcode will be handled by the AMUSE interface (0 means success and < 0 means raise an exception).

Let's write a test to see if it works, open an editor on the test_nearestneighbor.py class and add this method:

```
def test3(self):
    instance = NearestNeighbor()
    x = [0.0, 1.0, 4.0, 7.5] | generic_unit_system.length
    y = [0.0] * len(x) | generic_unit_system.length
    z = [0.0] * len(x) | generic_unit_system.length

    n0, n1, n2 = instance.find_nearest_neighbors(x,y,z)

    self.assertEquals(n0, [2,1,2,3])
    self.assertEquals(n1, [3,3,4,2])
    self.assertEquals(n2, [4,4,1,1])

    instance.stop()
```

Note: This test looks a lot like test2, but we now have to define a unit and we do not need to handle the errorcode.

Now build and test the code:

```
> make clean; make all
> nosetests
...
----------------------------------------------------------------------
Ran 3 tests in 0.650s

OK
```

Note: Although we only edited python code we still need to run make. The code will check if the "worker_code" executable is up to date on every run. It cannot detect if the update broke the code but it will still demand that the code is rebuilt.

Defining the particle set

Partcle sets in AMUSE can be handled by python (we call these "inmemory") and by the legacy code (we call these "incode"). In our case the code does not handle the particles and we need to configure the particles handler to manage an inmemory particle set. Open an editor on *interface.py* and add this method to the **NearestNeighbor** class:

```
def define_particle_sets(self, object):
    object.define_inmemory_set('particles')
```

That's all we now have a "particles" attribute on the class and we can add, remove, delete particles from this set. But we are still missing a connection between the particles and the nearest neighbors. AMUSE provides no handler for this, instead, we will write a method to run the find_nearest_neighbors function and set the indices on the particles set.

Open an editor on *interface.py* and add this method to the **NearestNeighbor** class:

```
def run(self):
    indices0, indices1, indices2 = self.find_nearest_neighbors(
        self.particles.x,
        self.particles.y,
        self.particles.z
    )
    self.particles.neighbor0 = self.particles[indices0]
    self.particles.neighbor1 = self.particles[indices1]
    self.particles.neighbor2 = self.particles[indices2]
```

This function gets the “x”, “y” and “z” attributes from the particles set and sends these to the “find_nearest_neighbors” method. This methods returns 3 lists of indices and we need to find the particles with these indices.

Note: Particle sets have no given sequence, deletion and addition of particles will change the order of the particles in the set. It is therefor never a good idea to use the index of the particle in the set as a reference to that particle. However, in the “run” method we “own” the particle set, it cannot change between the find_nearest_neighbor call and the moment we find the particles in the set by index (using self.particles[indices0]), and in this case it is save to use index as a valid reference.

Let’s write a test and see if it works, open an editor on the test_nearestneighbor.py class and add this method:

```
def test4(self):
    instance = NearestNeighbor()

    particles = data.Particles(4)
    particles.x = [0.0, 1.0, 4.0, 7.5] | generic_unit_system.length
    particles.y = 0.0 | generic_unit_system.length
    particles.z = 0.0 | generic_unit_system.length

    instance.particles.add_particles(particles)
    instance.run()

    self.assertEqual(instance.particles[0].neighbor0, instance.particles[1])
    self.assertEqual(instance.particles[1].neighbor0, instance.particles[0])
    self.assertEqual(instance.particles[2].neighbor0, instance.particles[1])
    self.assertEqual(instance.particles[3].neighbor0, instance.particles[2])

    instance.stop()
```

Now, make and run the tests:

```
> make clean; make all
> nosetests
....
----------------------------------------------------------------------
Ran 4 tests in 0.797s

OK
```

We are done, we have defined an object oriented interface on the legacy interface. Only, if we look at our tests, the code seems to be more rather than less complex. But, remember we now have units and we are compatible with other parts of amuse. And we can make more complex scripts easier.

Let’s make a plummer model and find the nearest neighbors in this model.

First make a file with the following contents, let’s call this file **plummer2.py**:

```
from interface import NearestNeighbor
from amuse.lab import *
from amuse.io import text

if __name__ == '__main__':
    number_of_particles = 1000
    particles = new_plummer_sphere(1000)

    code = NearestNeighbor()
    code.particles.add_particles(particles)

    code.run()

    local_particles = code.particles.copy()
    delta = local_particles.neighbor1.position - local_particles.position
```

```
    local_particles.dx = delta[...,0]
    local_particles.dy = delta[...,1]
    local_particles.dz = delta[...,2]

    output = text.TableFormattedText("output.txt", set = local_particles)
    output.attribute_names = ['x','y','z', 'dx', 'dy','dz']
    output.store()
```

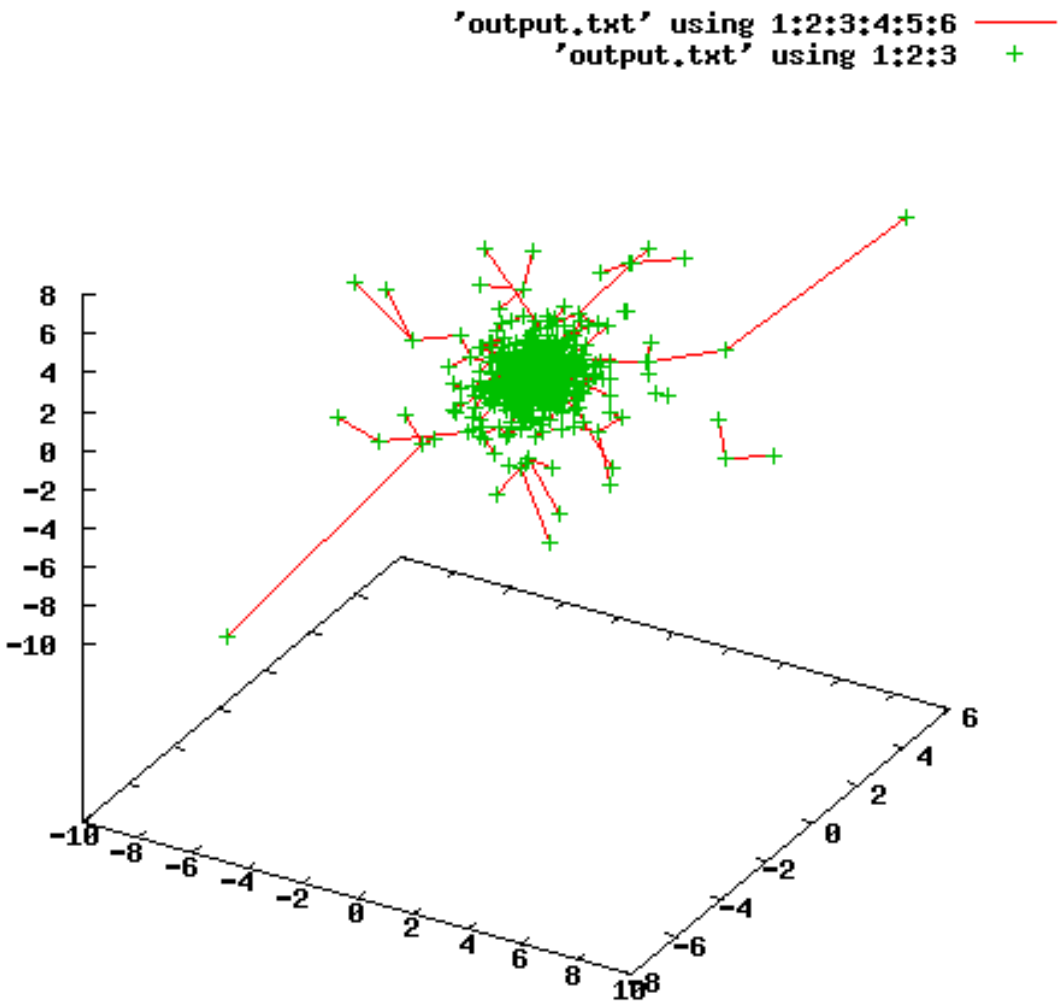
We can run this file with python:

```
.. code-block:: bash
```

    $AMUSE_DIR/amuse.sh plummer2.py

It will create an **output.txt** file and we can show this file with gnuplot.

```
gnuplot> splot 'output.txt' using 1:2:3:4:5:6 with vectors nohead, 'output.txt' using 1:2:3
gnuplot> #we can zoom into the center
gnuplot> set xr[-0.5:0.5]
gnuplot> set yr[-0.5:0.5]
gnuplot> set zr[-0.5:0.5]
gnuplot> splot 'output.txt' using 1:2:3:4:5:6 with vectors nohead, 'output.txt' using 1:2:3
```



6.6 Path 2

6.7 Defining the legacy interface

We define our code interface so that a user can add, update and delete particles, start the nearest neighbors finding algorithm and retrieve the ids of the nearest neighbors.

To define the interface, open interface.py with your favorite editor and replace the contents of this file with:

```
from amuse.community import *

class NearestNeighborInterface(CodeInterface):

    include_headers = ['worker_code.h']

    def __init__(self, **keyword_arguments):
        CodeInterface.__init__(self, **keyword_arguments)

    @legacy_function
    def new_particle():
        function = LegacyFunctionSpecification()
        function.can_handle_array = True
        function.addParameter('index_of_the_particle', dtype='int32', direction=function.OUT)
        function.addParameter('x', dtype='float64', direction=function.IN)
        function.addParameter('y', dtype='float64', direction=function.IN)
        function.addParameter('z', dtype='float64', direction=function.IN)
        function.result_type = 'int32'
        return function

    @legacy_function
    def delete_particle():
        function = LegacyFunctionSpecification()
        function.addParameter('index_of_the_particle', dtype='int32', direction=function.IN)
        function.result_type = 'int32'
        return function

    @legacy_function
    def get_state():
        function = LegacyFunctionSpecification()
        function.can_handle_array = True
        function.addParameter('index_of_the_particle', dtype='int32', direction=function.IN)
        function.addParameter('x', dtype='float64', direction=function.OUT)
        function.addParameter('y', dtype='float64', direction=function.OUT)
        function.addParameter('z', dtype='float64', direction=function.OUT)
        function.result_type = 'int32'
        return function

    @legacy_function
    def set_state():
        function = LegacyFunctionSpecification()
        function.can_handle_array = True
        function.addParameter('index_of_the_particle', dtype='int32', direction=function.IN)
        function.addParameter('x', dtype='float64', direction=function.IN)
        function.addParameter('y', dtype='float64', direction=function.IN)
        function.addParameter('z', dtype='float64', direction=function.IN)
        function.result_type = 'int32'
        return function

    @legacy_function
    def find_nearest_neighbors():
        function = LegacyFunctionSpecification()
        function.result_type = 'int32'
        return function
```

```
    @legacy_function
    def get_close_neighbors():
        function = LegacyFunctionSpecification()
        function.can_handle_array = True
        function.addParameter('index_of_the_particle', dtype='int32', direction=function.IN)
        function.addParameter('index_of_first_neighbor', dtype='float64', direction=function.OUT)
        function.addParameter('index_of_second_neighbor', dtype='float64', direction=function.OUT)
        function.addParameter('index_of_third_neighbor', dtype='float64', direction=function.OUT)
        function.result_type = 'int32'
        return function

    @legacy_function
    def get_nearest_neighbor():
        function = LegacyFunctionSpecification()
        function.can_handle_array = True
        function.addParameter('index_of_the_particle', dtype='int32', direction=function.IN)
        function.addParameter('index_of_the_neighbor', dtype='float64', direction=function.OUT)
        function.addParameter('distance', dtype='float64', direction=function.OUT)
        function.result_type = 'int32'
        return function

    @legacy_function
    def get_number_of_particles():
        function = LegacyFunctionSpecification()
        function.can_handle_array = True
        function.addParameter('value', dtype='int32', direction=function.OUT)
        function.result_type = 'int32'
        return function


class NearestNeighbor(CodeInterface):

    def __init__(self):
        CodeInterface.__init__(self,  NearestNeighborInterface())
```

We can generate a stub from the interface code with:

```
> $AMUSE_DIR/build.py --type=c --mode=stub interface.py NearestNeighborInterface -o interface.cc
```

The generated **interface.cc** replaces the original file generated in the previous section.

The code builds, but does not have any functionality yet:

```
> make clean
> make all
```

Note: Compiling the interface code will result in a lot of warnings about unused dummy arguments. These warnings can be savely ignored for now.

The tests are broken (the echo_int function has been removed):

```
> nosetests
E
======================================================================
ERROR: test1 (nearestneighbor.test_nearestneighbor.NearestNeighborInterfaceTests)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "../src/amuse/test/amusetest.py", line 146, in run
    testMethod()
  File "nearestneighbor/test_nearestneighbor.py", line 11, in test1
    result,error = instance.echo_int(12)
AttributeError: 'NearestNeighborInterface' object has no attribute 'echo_int'
```

```
----------------------------------------------------------------------
Ran 1 test in 0.315s

FAILED (errors=1)
```

Let's create a working test by calling the new_particle method, open an editor on the test_nearestneighbor.py file and replace the test1 method with:

```
def test1(self):
    instance = NearestNeighborInterface()
    result,error = instance.new_particle(1.0, 1.0, 2.0)
    self.assertEquals(error, 0)
    self.assertEquals(result, 1)
    instance.stop()
```

As this is python code we do not need to rebuild the code, instead we can run the tests right after saving the code. Unfortunately, when we run the test, it still fails.

```
> nosetests
F
======================================================================
FAIL: test1 (nearestneighbor.test_nearestneighbor.NearestNeighborInterfaceTests)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "/src/amuse/test/amusetest.py", line 146, in run
    testMethod()
  File "/nearestneighbor/test_nearestneighbor.py", line 13, in test1
    self.assertEquals(result, 1)
  File "/src/amuse/test/amusetest.py", line 62, in failUnlessEqual
    self._raise_exceptions_if_any(failures, first, second, '{0} != {1}', msg)
  File "/src/amuse/test/amusetest.py", line 49, in _raise_exceptions_if_any
    raise self.failureException(msg or err_fmt_string.format(first, second, *args))
AssertionError: 0 != 1
-------------------- >> begin captured logging << --------------------
legacy: INFO: start call 'NearestNeighborInterface.new_particle'
legacy: INFO: end call 'NearestNeighborInterface.new_particle'
--------------------- >> end captured logging << ---------------------

----------------------------------------------------------------------
Ran 1 test in 0.319s
```

When you look closely at the output of the test you see that the result from the method is 0 and not the excpeted 1. We need to edit the fortan code to make this test work. Open an editor on interface.cc and go to the *new_particle* function.

```
int new_particle(int * index_of_the_particle, double x, double y,
    double z)
{
    *index_of_the_particle = 1;
    return 0;
}
```

Note: In AMUSE all interface functions return an errorcode. Any other return values must be passed through the arguments of the functions.

We need to rebuild the code, and after building run the tests.

```
> make all
> nosetests
.
----------------------------------------------------------------------
Ran 1 test in 0.427s
```

```
OK
```

The are tests work again! Only, we do not have any real working legacy code.

6.8 Filling the stubs

The implementation of the algorithm does not match the interface we defined and created. We need to write some glue code to connect the code with the interface. To do so we fill in the stubs generated earlier.

Open the **interface.cc** file in your favorite editor and change its contents to:

```
#include "worker_code.h"
#include "src/code.h"
#include <map>

class Particle{

public:
    int index;
    double x,y,z;
    int n0, n1, n2;

    Particle(int index, double x, double y, double z):index(index), x(x), y(y), z(z), n0(0), n1(0
    }
    Particle(const Particle & original):index(original.index), x(original.x), y(original.y), z(or
    }
};

typedef std::map<int, Particle *> ParticlesMap;
typedef std::map<int, Particle *>::iterator ParticlesMapIterator;

int highest_index = 0;
ParticlesMap particlesMap;

int find_nearest_neighbors(){
  std::size_t n = particlesMap.size();

  double * x = new double[n];
  double * y = new double[n];
  double * z = new double[n];
  int * n0 = new int[n];
  int * n1 = new int[n];
  int * n2 = new int[n];
  Particle ** particles = new Particle*[n];
  ParticlesMapIterator i;
  int c = 0;

  for(i = particlesMap.begin(); i != particlesMap.end(); i++) {
    Particle * p = (*i).second;
    particles[c] = p;
    x[c] = p->x;
    y[c] = p->y;
    z[c] = p->z;

    c++;
  }

  int errorcode = find_nearest_neighbors(n, x, y, z, n0, n1, n2);
  if(errorcode) {
    return errorcode;
  }
```

```
  for(std::size_t j = 0 ; j < n; j++) {
    Particle * p = particles[j];

    p->n0 = n0[j] >= 0 ? particles[n0[j]]->index : -1;
    p->n1 = n1[j] >= 0 ? particles[n1[j]]->index : -1;
    p->n2 = n1[j] >= 0 ? particles[n2[j]]->index : -1;
  }

  return 0;
}

int get_nearest_neighbor(int index_of_the_particle,
  double * index_of_the_neighbor, double * distance){

  if(index_of_the_particle > highest_index) {
    return -1;
  }

  Particle * p0 = particlesMap[index_of_the_particle];
  Particle * p1 = particlesMap[p0->n0];
  *index_of_the_neighbor  = p0->n0;
  *distance = distance_between_points(p0->x, p0->y, p0->z, p1->x, p1->y, p1->z);

  return 0;
}

int new_particle(int * index_of_the_particle, double x, double y,
  double z){

  *index_of_the_particle = highest_index;

  Particle * p = new Particle(highest_index, x, y, z);
  particlesMap[highest_index] = p;

  highest_index++;

  return 0;
}

int get_close_neighbors(
  int index_of_the_particle,
  double * index_of_first_neighbor,
  double * index_of_second_neighbor,
  double * index_of_third_neighbor
  ){

  if(index_of_the_particle > highest_index) {
    return -1;
  }

  Particle * p = particlesMap[index_of_the_particle];
  *index_of_first_neighbor  = p->n0;
  *index_of_second_neighbor = p->n1;
  *index_of_third_neighbor  = p->n2;

  return 0;
}

int delete_particle(int index_of_the_particle){

  if(index_of_the_particle > highest_index) {
    return -1;
  }
```

```
  particlesMap.erase(index_of_the_particle);

  return 0;
}

int set_state(int index_of_the_particle, double x, double y, double z){

  if(index_of_the_particle > highest_index) {
    return -1;
  }

  Particle * p = particlesMap[index_of_the_particle];

  p->x = x;
  p->y = y;
  p->z = z;

  return 0;
}

int get_state(int index_of_the_particle, double * x, double * y,
  double * z){

  if(index_of_the_particle > highest_index) {
    return -1;
  }

  Particle * p = particlesMap[index_of_the_particle];

  *x = p->x;
  *y = p->y;
  *z = p->z;

  return 0;
}

int get_number_of_particles(int * value){
  *value = (int) particlesMap.size();
  return 0;
}
```

Test if the code builds and try it out. In the legacy interface directory do:

```
> make clean
> make all
> nosetests
.
----------------------------------------------------------------------
Ran 1 test in 0.311s

OK
```

Let's check some more functionaly by adding another test

```
def test2(self):
    instance = NearestNeighborInterface()
    result,error = instance.new_particle(1.0, 1.0, 2.0)
    self.assertEquals(error, 0)
    self.assertEquals(result, 1)
    result,error = instance.new_particle(2.0, 3.0, 2.0)
    self.assertEquals(error, 0)
    self.assertEquals(result, 2)
    result,error = instance.new_particle(2.0, 3.0, 2.0)
```

```
    self.assertEquals(error, -1)
    error = instance.delete_particle(1)
    self.assertEquals(error, 0)
    result,error = instance.new_particle(2.0, 3.0, 2.0)
    self.assertEquals(error, 0)
    self.assertEquals(result, 1)
    instance.stop()
```

The tests should succeed:

```
> nosetests
..
----------------------------------------------------------------------
Ran 2 tests in 0.448s

OK
```

We now have done everything in Step 0 *Legacy Interface*. We have a legacy code and can access it in our python script. But, our interface is not very friendly to work with. We have to think about errorcodes and we have not information about units. To make our interface easier to works with we start defining methods, properties and parameters.

6.9 Defining methods

The object oriented interface is also defined in the **interface.py**. So, we continue by opening an editor on this file. We will be writing methods for the **NearestNeighbor** class, in your editor seek this code (at the end of the file):

```
class NearestNeighbor(CodeInterface):

    def __init__(self, **options):
        CodeInterface.__init__(self,  NearestNeighborInterface(), **options)
```

We will start by defining methods, we will do this by implementing the **define_methods** function, like so:

```
class NearestNeighbor(CodeInterface):

    def __init__(self, **options):
        CodeInterface.__init__(self,  NearestNeighborInterface(), **options)


    def define_methods(self, builder):

        builder.add_method(
            "new_particle",
            (generic_unit_system.length, generic_unit_system.length, generic_unit_system.length,),
            (builder.INDEX, builder.ERROR_CODE)
        )

        builder.add_method(
            "delete_particle",
            (builder.INDEX,),
            (builder.ERROR_CODE)
        )

        builder.add_method(
            "get_state",
            (builder.INDEX,),
            (generic_unit_system.length, generic_unit_system.length, generic_unit_system.length,
            public_name = "get_position"
        )

        builder.add_method(
```

```
            "set_state",
            (builder.INDEX, generic_unit_system.length, generic_unit_system.length, generic_unit_s
            (builder.ERROR_CODE),
            public_name = "set_position"
        )

        builder.add_method(
            "run",
            (),
            (builder.ERROR_CODE),
        )

        builder.add_method(
            "get_close_neighbors",
            (builder.INDEX,),
            (builder.INDEX, builder.INDEX, builder.INDEX, builder.ERROR_CODE),
        )

        builder.add_method(
            "get_nearest_neighbor",
            (builder.INDEX,),
            (builder.INDEX, generic_unit_system.length, builder.ERROR_CODE),
        )

        builder.add_method(
            "get_number_of_particles",
            (),
            (builder.NO_UNIT, builder.ERROR_CODE),
        )
```

With this code, we define the methods and specify how to interpret the arguments and return values. We get a special object (the **builder** object) that provides us with the **add_method** function to be able to this. The definition of the **add_method** function is as follows:

```
add_method(
    name of the original function in the legacy interface,
    list of arguments (unit or type),
    list of return values,
    public_name = name for the user of the class (optional)
)
```

For every argument or return value we can specify if it has a unit or if it is special. The special arguments are:

definition	description
builder.ERROR_CODE	the value is interpreted as an errorcode, zero means no error for all other values and Exception will be raise, only valid for one return value.
builder.NO_UNIT	the value has not unit (for example for the number of items in a list)
builder.INDEX	the value is interpreted as an index for object identifiers

Test if the code builds and try it out. In the legacy interface directory do:

```
> make clean
> make all
```

Let's add another test:

```
def test3(self):
    instance = NearestNeighbor()
    instance.set_maximum_number_of_particles(2)
    instance.commit_parameters()
    result = instance.new_particle(
        1.0 | generic_unit_system.length,
        2.0 | generic_unit_system.length,
        3.0 | generic_unit_system.length
```

```
    )
    self.assertEquals(result, 1)
    result = instance.new_particle(
        1.0 | generic_unit_system.length,
        1.0 | generic_unit_system.length,
        2.0 | generic_unit_system.length
    )
    self.assertEquals(result, 2)
    x,y,z = instance.get_position(1)
    self.assertEquals(1.0 | generic_unit_system.length, x)
    self.assertEquals(2.0 | generic_unit_system.length, y)
    self.assertEquals(3.0 | generic_unit_system.length, z)
    instance.stop()
```

And run the tests:

```
> nosetests
...
----------------------------------------------------------------------
Ran 3 tests in 0.664s

OK
```

As you can see our script is now a little simpler and we support units. We do not have to think about the errorcodes in this script, AMUSE will interpret the errorcodes and raise the right exceptions if needed. The units are also automatically converted to the right units for the code. But the script is still not very easy and we have to manage all the ids we get from the code. To make our code even easier to handle we will continue by defining a **set**.

Note: We skip defining parameters and properties, we will come back to this later in this tutorial.

6.10 Defining a set

We have made our interface a little easier but we still have to do a some management work in our script. We would like to work with objects and adding or removing these objects from the code. AMUSE supports this by defining **sets**. Each set is capable of storing specific attributes of the objects in the set. Our code is capable of storing the x, y and z position of an object. An object in AMUSE is called a *Particle* and the sets that contain these particles are called *ParticleSets* or shorter *Particles*.

We define our particle set by implementing a **define_particle_sets** function on our **NearestNeighbor** class like so:

```
class NearestNeighbor(CodeInterface):

    def __init__(self, **options):
        CodeInterface.__init__(self,  NearestNeighborInterface(), **options)


    def define_methods(self, builder):
        ...

    def define_particle_sets(self, builder):
        builder.define_set('particles', 'index_of_the_particle')
        builder.set_new('particles', 'new_particle')
        builder.set_delete('particles', 'delete_particle')
        builder.add_setter('particles', 'set_position')
        builder.add_getter('particles', 'get_position')
        builder.add_getter('particles', 'get_close_neighbors', names=('neighbor0', 'neighbor1', '
```

That's all, we now have defined a set called **particles**. Again, we get a builder object to use in defining our set. All methods have the name of the set as their first argument, this name can be any name you want, but in AMUSE

most codes provide a set called **particles**. For the **add_setter**, **add_getter**, **set_new** and **set_delete** functions, the second argument is the name of the method we defined in the previous step. Finally you can set the name of the attribute in the particles set with the **names** argument. This is optional for legacy functions, if not given the names of the attributes will be derived from the names of the arguments in the orginal calls. For example, the **get_position** call we specified earlier has parameter name **x**, **y** and **z**, these names are also used in the particles set.

Test if the code builds and try it out. In the legacy interface directory do:

```
> make clean
> make all
```

Let's add another test:

```
def test4(self):
    instance = NearestNeighbor()
    instance.set_maximum_number_of_particles(100)
    instance.commit_parameters()

    particles = data.Particles(4)
    particles.x = [0.0, 1.0, 4.0, 7.5] | generic_unit_system.length
    particles.y = 0.0 | generic_unit_system.length
    particles.z = 0.0 | generic_unit_system.length

    instance.particles.add_particles(particles)
    instance.run()

    self.assertEqual(instance.particles[0].neighbor0, instance.particles[1])
    self.assertEqual(instance.particles[1].neighbor0, instance.particles[0])
    self.assertEqual(instance.particles[2].neighbor0, instance.particles[1])
    self.assertEqual(instance.particles[3].neighbor0, instance.particles[2])

    instance.stop()
```

The support for a particle set means we can now also interact with other parts of AMUSE. Let's make a plummer model and find the nearest neighbors in this model.

First make a file with the following contents, let's call this file **plummer2.py**:

```
from interface import NearestNeighbor
from amuse.lab import *
from amuse.io import text

if __name__ == '__main__':
    number_of_particles = 1000
    particles = new_plummer_sphere(1000)

    code = NearestNeighbor()
    code.particles.add_particles(particles)

    code.run()

    local_particles = code.particles.copy()
    delta = local_particles.neighbor1.position - local_particles.position

    local_particles.dx = delta[...,0]
    local_particles.dy = delta[...,1]
    local_particles.dz = delta[...,2]

    output = text.TableFormattedText("output.txt", set = local_particles)
    output.attribute_names = ['x','y','z', 'dx', 'dy','dz']
    output.store()
```

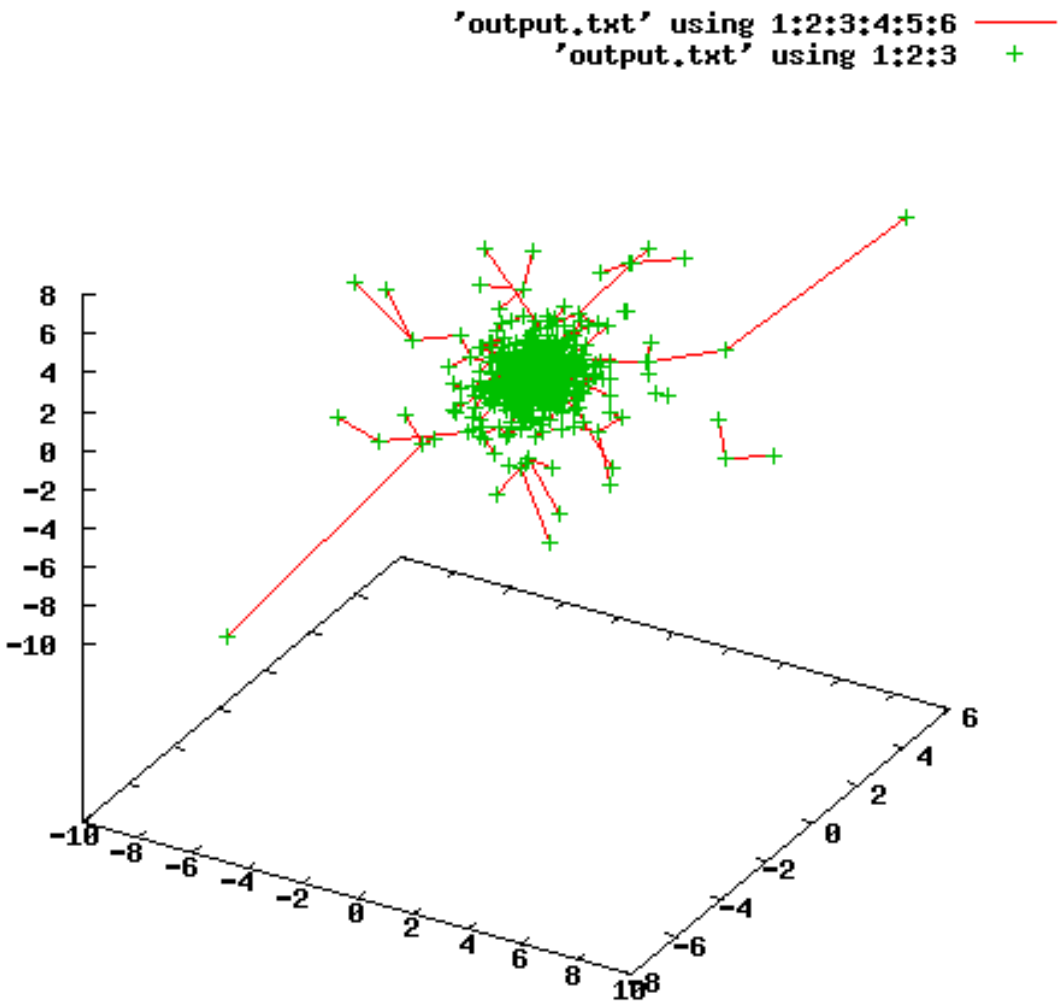
We can run this file with python:

```
.. code-block:: bash
```

$AMUSE_DIR/amuse.sh plummer2.py

It will create an **output.txt** file and we can show this file with gnuplot.

```
gnuplot> splot 'output.txt' using 1:2:3:4:5:6 with vectors nohead, 'output.txt' using 1:2:3
gnuplot> #we can zoom into the center
gnuplot> set xr[-0.5:0.5]
gnuplot> set yr[-0.5:0.5]
gnuplot> set zr[-0.5:0.5]
gnuplot> splot 'output.txt' using 1:2:3:4:5:6 with vectors nohead, 'output.txt' using 1:2:3
```

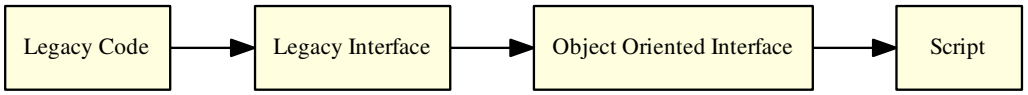


7 Integrate a Fortran 90 code

In this tutorial we will create an AMUSE interface to a fortran 90 code. We will first define the legacy interface, then implement the code and finally build an object oriented interface on top of the legacy interface.

The legacy code interface supports methods that transfer values to and from the code. The values do not have any units and no error handling is provided by the interface. We can add error handling, unit handling and more

functions to the legacy interface by defining a subclass of a CodeInterface (this is the objected oriented interface).



The legacy code in this tutorial will be a very simple and naive implementation to find 3 nearest neighbors of a particle.

7.1 Two paths

When defining the interface will walk 2 paths:

1. Management of particles in AMUSE (python)
2. Management of particles in the code (C or Fortran)

The first path makes sense for legacy codes that perform a transformation on the particles, or analyse the particles state or do not store any internal state between function calls (all data is external). For every function of the code, data of every particle is send to the code. If we expect multiple calls, the code would incur a high communication overhead and we are better of choosing path 2.

The second path makes sense for codes that already have management of a particles (or grid) or were we want to call multiple functions of the code and need to send the complete model to code for every function call. The code is first given the data, then calls are made to the code to evolve it's model or perform reduction steps on the data, finally the updated data is retrieved from the code.

7.2 Procedure

The suggested procedure for creating a new interface is as follows:

0. **Legacy Interface.** Start with creating the legacy interface. Define functions on the interface to input and output relevant data. The CodeInterface code depends on the legacy interface code.
1. **Make a Class.** Create a subclass of the CodeInterface class
2. **Define methods.** In the legacy interface we have defined functions with parameters. In the code interface we need to define the units of the parameters and if a parameter or return value is used as an errorcode.
3. **Define properties.** Some functions in the legacy interface can be better described as a property of the code. These are read only variables, like the current model time.
4. **Define parameters.** Some functions in the legacy interface provide access to parameters of the code. Units and default values need to be defined for the parameters in this step
5. **Define sets or grids.** A code usually handles objects or gridpoints with attributes. In this step a generic interface is defined for these objects so that the interopability between codes increases.

7.3 Before we start

This tutorial asumes you have a working amuse environment. Please ensure that amuse is setup correctly by running 'nosetests' in the amuse directory.

Environment variables

To simplify the work in the coming sections, we first define the environment variable 'AMUSE_DIR'. This environment variable must point to the root directory of AMUSE (this is the directory containing the build.py script).

```
> export AMUSE_DIR=<path to the amuse root directory>
```

or in a c shell:

```
> setenv AMUSE_DIR <path to the amuse root directory>
```

After building the code, we want to run and test the code. Check if amuse is available in your python path by running the following code on the command line.

```
> python -c "import amuse"
Traceback (most recent call last):
File "<string>", line 1, in <module>
ImportError: No module named amuse
```

If this code ends in a "ImportError" as shown in the example, the PYTHONPATH environment variable must be extended with the src directory in AMUSE_DIR. We can do so by using one of the following commands.

```
> export PYTHONPATH=${PYTHONPATH}:${AMUSE_DIR}/src
```

or in a c shell:

```
> setenv AMUSE_DIR ${PYTHONPATH}:${AMUSE_DIR}/src
```

The name of our project

We will be writing a code to find the nearest neighbors of a particle, so let's call our project 'NearestNeighbor'.

Creating the initial directory structure

First we need to create a directory for our project and put some files in it to help build the code. The fastest method to setup the directory is by using the build.py script.

```
> $AMUSE_DIR/build.py --type=f90 --mode=dir NearestNeighbor
```

The script will generate a directory with all the files needed to start our project. It has also generates a very small legacy code with only one function `'echo_int'`. We can build and test our new module:

```
> cd nearestneighbor/
> make all
> $AMUSE_DIR/amuse.sh -c 'from interface import NearestNeighbor; print NearestNeighbor().echo_int
OrderedDictionary({'int_out':10, '__result':0})
> nosetests -v
.
----------------------------------------------------------------------
Ran 1 test in 0.556s

OK
```

Note: The build.py script can be used to generate a range of files. To see what this file can do you can run the script with a –help parameter, like so:

```
> $AMUSE_DIR/build.py --help
```

7.4 The Legacy Code

Normally the legacy code already exists and our task is limited to defining and implementing an interface so that AMUSE scripts can access the code. For this tutorial we will implement our legacy code.

When a legacy code is integrated all interface code is put in one directory and all the legacy code is put in a **src** directory placed under this directory. The build.py script created a **src** directory for us, and we will put the nearest neighbor algorithm in this directory.

Go to the **src** directory and create a **code.f90** file, open this file in your favorite editor and copy and paste this code into it:

```
FUNCTION distance_between_points(x0,y0,z0,x1,y1,z1)
    IMPLICIT NONE
    DOUBLE PRECISION :: distance_between_points
    DOUBLE PRECISION :: x0,y0,z0,x1,y1,z1
    DOUBLE PRECISION :: dx,dy,dz
    dx = x1 - x0
    dy = y1 - y0
    dz = z1 - z0
    distance_between_points = SQRT(dx * dx  + dy * dy + dz * dz)
END FUNCTION


FUNCTION find_nearest_neighbors (npoints,x,y,z,n0,n1,n2)
    IMPLICIT NONE
    INTEGER, INTENT(IN):: npoints
    INTEGER :: i,j, k, kk
    INTEGER :: find_nearest_neighbors
    DOUBLE PRECISION :: x(npoints),y(npoints),z(npoints)
    INTEGER :: n0(npoints),n1(npoints),n2(npoints)
    DOUBLE PRECISION :: r
    DOUBLE PRECISION :: distance_between_points
    INTEGER :: nn_index(3)
    DOUBLE PRECISION :: nn_distance(3)

    DO i = 1, npoints, 1
        DO k = 1, 3
            nn_index(k) = 0
            nn_distance(k) = 0.0
        END DO

        DO j = 1, npoints, 1
            IF (i.NE.j) THEN
                r = distance_between_points(x(i), y(i), z(i), &
                 x(j), y(j), z(j))

                DO k = 1, 3
                    IF (nn_index(k).EQ.0) THEN
                        nn_index(k) = j
                        nn_distance(k) = r
                        EXIT
                    ELSE
                        IF (r.LT.nn_distance(k)) THEN
                            DO kk = 3, k, -1
                                nn_index(kk) = nn_index(kk-1)
                                nn_distance(kk) = nn_distance(kk-1)
                            END DO
                            nn_index(k) = j
                            nn_distance(k) = r
                            EXIT
                        END IF
                    END IF
                END DO
```

```
            END IF
        END DO

        n0(i) = nn_index(1)
        n1(i) = nn_index(2)
        n2(i) = nn_index(3)
    END DO

    find_nearest_neighbors = 0
END FUNCTION
```

Note: This algorithm is un-optimized and has N*N order. It is not meant as very efficient code but as a readable example.

Before we can continue we also need to alter the **Makefile** in the **src** directory, so that our **code.cc** file is included in the build. To do so, open an editor on the Makefile and change the line:

```
CODEOBJS = test.o
```

to:

```
CODEOBJS = test.o code.o
```

Test if the code builds. As we have not coupled our algorithm to the interface we (we have not even defined an interface) we do not have any new functionality. In the legacy interface directory (not the **src** directory) do:

```
> make all
> nosetests
.
----------------------------------------------------------------------
Ran 1 test in 0.427s

OK
```

It works, if the test fails for any reason please check that the fortran code is correct and that `worker_code` exists in your directory.

7.5 Path 1

Defining the legacy interface

We will first define a legacy interface so that we can call the **find_nearest_neighbors** function from python. AMUSE can interact with 2 classes of functions:

1. A function with all scalar input and output variables. All variables are simple, non-composite variables (like INTEGER or DOUBLE PRECISION). For example:

   ```
   FUNCTION example1(input, output)
       IMPLICIT NONE
       INTEGER example1
       DOUBLE PRECISION input, output
       output = input
       example1 = 0
   END FUNCTION
   ```

2. A function with all vector (or list) input and output variables and a length variable. The return value is a scalar value. For example:

   ```
   FUNCTION example2(input, output, N)
       IMPLICIT NONE
       INTEGER example2, N, i
       DOUBLE PRECISION input(N), output(N)
   ```

```
        DO i = 1, N
            output(i) = input(i)
        END DO
        example2 = 0
    END FUNCTION
```

If you have functions that don't follow this pattern you need to define a convert function in fortran that provides an interface following one of the two patterns supported by AMUSE.

In our case the **find_nearest_neighbors** complies to pattern 2 and we do not have to write any code in fortran to convert the function to a compliant interface. We only have to specify the function in python. We do so by adding a **find_nearest_neighbors** method to the **NearestNeighborInterface** class in the **interface.py** file. Open and editor on the interfaces.py file and add the following method to the NearestNeighborInterface class:

```
class NearestNeighborInterface(CodeInterface):
    #...

    @legacy_function
    def find_nearest_neighbors():
        function = LegacyFunctionSpecification()
        function.must_handle_array = True
        function.addParameter('npoints', dtype='int32', direction=function.LENGTH)
        function.addParameter('x', dtype='float64', direction=function.IN)
        function.addParameter('y', dtype='float64', direction=function.IN)
        function.addParameter('z', dtype='float64', direction=function.IN)
        function.addParameter('n0', dtype='int32', direction=function.OUT)
        function.addParameter('n1', dtype='int32', direction=function.OUT)
        function.addParameter('n2', dtype='int32', direction=function.OUT)
        function.result_type = 'int32'
        return function
```

In the *find_nearest_neighbors* method we specify every parameter of the fortran function and the result type. For each parameter we need to define a name, data type and wether we will input, output (or input and output) data using this parameter. AMUSE knows only a limited amount of data types for parameters: float64, float32, int32 and string. We also have a special parameter, with LENGTH as direction. This parameter is needed for all functions that follow pattern 2, it will be filled with the length of the input arrays. We also must specify that the function follows pattern 2 by setting `function.must_handle_array = True`.

Save the file and recompile the code.

```
> make clean
> make all
> nosetests
.
----------------------------------------------------------------------
Ran 1 test in 0.427s

OK
```

It works! But, how do we know the find_nearest_neighbors method really works? Let's write a test and find out. Open an editor on the *test_nearestneighbor.py* file and add the following method to the **NearestNeighborInterfaceTests** class:

```
def test2(self):
    instance = NearestNeighborInterface()
    x = [0.0, 1.0, 4.0, 7.5]
    y = [0.0] * len(x)
    z = [0.0] * len(x)

    n0, n1, n2, error = instance.find_nearest_neighbors(x,y,z)

    self.assertEquals(error[0], 0)
    self.assertEquals(n0, [2,1,2,3])
    self.assertEquals(n1, [3,3,4,2])
```

```
    self.assertEquals(n2, [4,4,1,1])

    instance.stop()
```

This test calls the *find_nearest_neighbors* method with 4 positions and checks if the nearest neighbors are determined correctly. Let's run the test, and see if everything is working:

```
> nosetests
..
----------------------------------------------------------------------
Ran 2 test in 0.491s

OK
```

We now have a simple interface that works, but we have to do our own indexing after the call and we could send data of any unit to the method, also we have to do our own error checking after the method. Let's define a object oriented interface to solve these problems

Defining the Object Oriented Interface

The object oriented interface sits on top of the legacy interface. It decorates this interface with sets, unit handling, state engine and more. We start creating the object oriented interfacy by inheriting from CodeInterface and writing the __init__ function. The build script has added this class to the interface.py file for us. Open an editor on interface.py and make sure this code is in the file (at the end of the file):

```
class NearestNeighbor(CodeInterface):

    def __init__(self, **options):
        CodeInterface.__init__(self,  NearestNeighborInterface(), **options)
```

Configuring the handlers

We configure the object oriented interfacy by implementing several methods. The object oriented interface is implement by several "handlers". Each handler provides support for a specific aspect of the interface. AMUSE defines a handler for the unit conversion, a handler for the interfacing with sets of particles, a handler to ensure the methods are called in the right order, etc. Each handler is very generic and needs to be configured before use. The handler are configured using the "Visitor" pattern. The following pseudo-code shows how the handlers are configured

```
class CodeInterface(object):
    #...

    def configure_handlers(self):
        #...
        for handler in self.get_all_handlers():
            handler.configure(self)

    def define_converter(self, handler):
        """ configure the units converter handler """

        handler.set_nbody_converter(...)

    def define_particle_sets(self, handler):
        """ configure sets of particles """

        handler.define_incode_particle_set(...)
        handler.set_getter(...)

class HandleConvertUnits(AbstractHandler):
    #...
```

```
    def configure(self, interface):
        interface.define_converter(self)

class HandleParticles(AbstractHandler):
    #...

    def configure(self, interface):
        interface.define_particle_sets(self)
```

Configuration of the handlers is optional, we only have to define those handler that we need in our interface. In our example we need to configure the "HandleMethodsWithUnits" handler (to define units and error handling) and the "HandleParticles" to define a particle set.

Defining methods with units

We first want to add units and error handling to the **find_nearest_neighbors**. We do this by creating a **define_methods** function on the **NearestNeighbor** class. Open an editor on *interface.py* and add this method to the class:

```
def define_methods(self, handler):

    handler.add_method(
        "find_nearest_neighbors",
        (
            generic_unit_system.length,
            generic_unit_system.length,
            generic_unit_system.length,
        ),
        (
            handler.INDEX,
            handler.INDEX,
            handler.INDEX,
            handler.ERROR_CODE
        )
    )
```

The **add_method** call expects the name of the function in the legacy interface as it's first, next it expects a list of the units of the input parameters and a list of the units of the output parameters. The return value of a function is always the last item in the list of output parameters. We specify a **generic_unit_system.length** unit for the x, y and z parameters. The output parameters are indices and an errorcode. The errorcode will be handled by the AMUSE interface (0 means success and < 0 means raise an exception).

Let's write a test to see if it works, open an editor on the test_nearestneighbor.py class and add this method:

```
def test3(self):
    instance = NearestNeighbor()
    x = [0.0, 1.0, 4.0, 7.5] | generic_unit_system.length
    y = [0.0] * len(x) | generic_unit_system.length
    z = [0.0] * len(x) | generic_unit_system.length

    n0, n1, n2 = instance.find_nearest_neighbors(x,y,z)

    self.assertEquals(n0, [2,1,2,3])
    self.assertEquals(n1, [3,3,4,2])
    self.assertEquals(n2, [4,4,1,1])

    instance.stop()
```

Note: This test looks a lot like test2, but we now have to define a unit and we do not need to handle the errorcode.

Now build and test the code:

```
> make clean; make all
> nosetests
...
----------------------------------------------------------------------
Ran 3 tests in 0.650s

OK
```

Note: Although we only edited python code we still need to run make. The code will check if the "worker_code" executable is up to date on every run. It cannot detect if the update broke the code but it will still demand that the code is rebuilt.

Defining the particle set

Partcle sets in AMUSE can be handled by python (we call these "inmemory") and by the legacy code (we call these "incode"). In our case the code does not handle the particles and we need to configure the particles handler to manage an inmemory particle set. Open an editor on *interface.py* and add this method to the **NearestNeighbor** class:

```
def define_particle_sets(self, object):
    object.define_inmemory_set('particles')
```

That's all we now have a "particles" attribute on the class and we can add, remove, delete particles from this set. But we are still missing a connection between the particles and the nearest neighbors. AMUSE provides no handler for this, instead, we will write a method to run the find_nearest_neighbors function and set the indices on the particles set.

Open an editor on *interface.py* and add this method to the **NearestNeighbor** class:

```
def run(self):
    indices0, indices1, indices2 = self.find_nearest_neighbors(
        self.particles.x,
        self.particles.y,
        self.particles.z
    )
    self.particles.neighbor0 = self.particles[indices0 - 1]
    self.particles.neighbor1 = self.particles[indices1 - 1]
    self.particles.neighbor2 = self.particles[indices2 - 1]
```

This function gets the "x", "y" and "z" attributes from the particles set and sends these to the "find_nearest_neighbors" method. This methods returns 3 lists of indices and we need to find the particles with these indices. As this is fortran code (indices start with 1) and we use python (indices start with 0) we need to subtract 1 from the array of indices and use this to find the particles.

Note: Particle sets have no given sequence, deletion and addition of particles will change the order of the particles in the set. It is therefor never a good idea to use th index of the particle in the set as a reference to that particle. However, in the "run" method we "own" the particle set, it cannot change between the find_nearest_neighbor call and the moment we find the particles in the set by index (using self.particles[indices0-1]), and in this case it is save to use index as a valid reference.

Let's write a test and see if it works, open an editor on the test_nearestneighbor.py class and add this method:

```
def test4(self):
    instance = NearestNeighbor()

    particles = data.Particles(4)
    particles.x = [0.0, 1.0, 4.0, 7.5] | generic_unit_system.length
```

```
    particles.y = 0.0 | generic_unit_system.length
    particles.z = 0.0 | generic_unit_system.length

    instance.particles.add_particles(particles)
    instance.run()

    self.assertEqual(instance.particles[0].neighbor0, instance.particles[1])
    self.assertEqual(instance.particles[1].neighbor0, instance.particles[0])
    self.assertEqual(instance.particles[2].neighbor0, instance.particles[1])
    self.assertEqual(instance.particles[3].neighbor0, instance.particles[2])

    instance.stop()
```

Now, make and run the tests:

```
> make clean; make all
> nosetests
....
----------------------------------------------------------------------
Ran 4 tests in 0.797s

OK
```

We are done, we have defined an object oriented interface on the legacy interface. Only, if we look at our tests, the code seems to be more rather than less complex. But, remember we now have units and we are compatible with other parts of amuse. And we can make more complex scripts easier.

Let's make a plummer model and find the nearest neighbors in this model.

First make a file with the following contents, let's call this file **plummer2.py**:

```
from interface import NearestNeighbor
from amuse.lab import *
from amuse.io import text

if __name__ == '__main__':
    number_of_particles = 1000
    particles = new_plummer_sphere(1000)

    code = NearestNeighbor()
    code.particles.add_particles(particles)

    code.run()

    local_particles = code.particles.copy()
    delta = local_particles.neighbor1.position - local_particles.position

    local_particles.dx = delta[...,0]
    local_particles.dy = delta[...,1]
    local_particles.dz = delta[...,2]

    output = text.TableFormattedText("output.txt", set = local_particles)
    output.attribute_names = ['x','y','z', 'dx', 'dy','dz']
    output.store()
```

We can run this file with python:

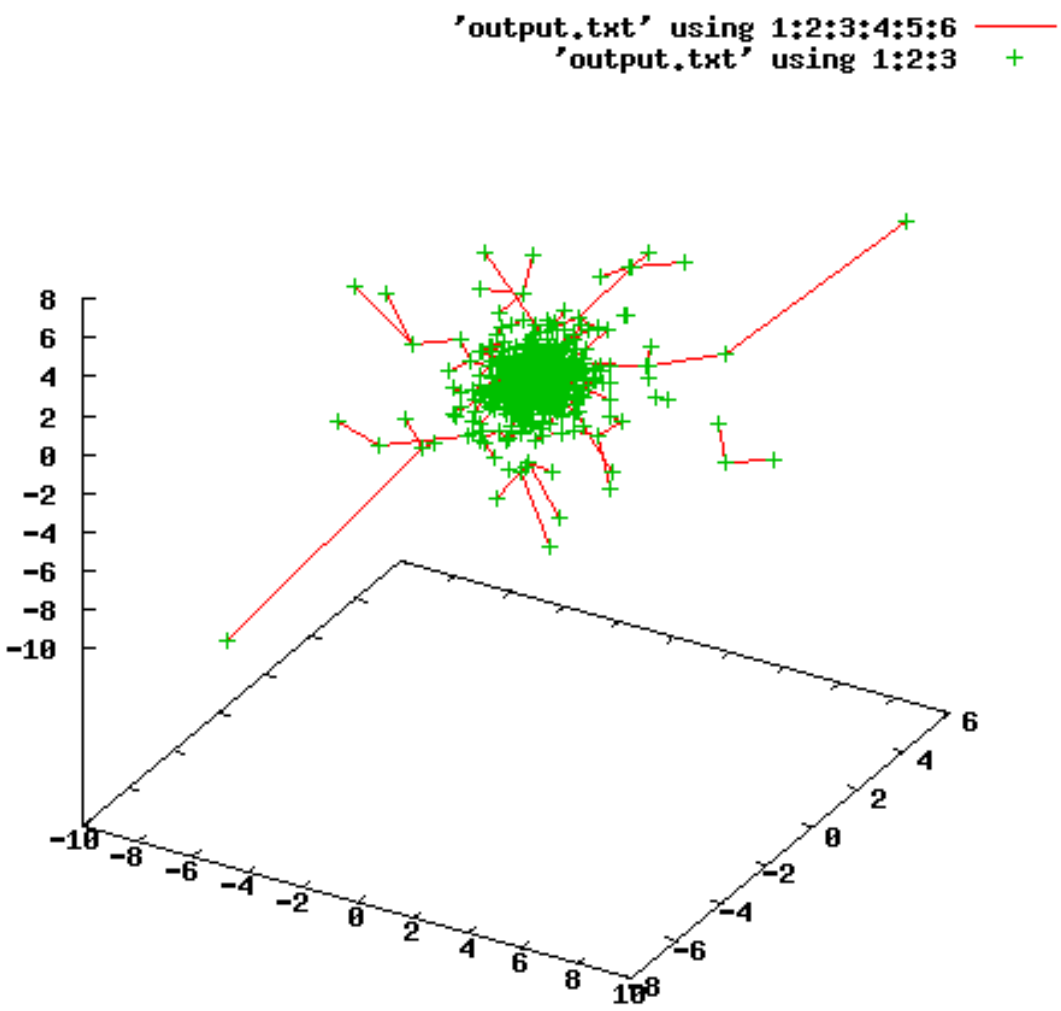
```
.. code-block:: bash
```

    $AMUSE_DIR/amuse.sh plummer2.py

It will create an **output.txt** file and we can show this file with gnuplot.

```
gnuplot> splot 'output.txt' using 1:2:3:4:5:6 with vectors nohead, 'output.txt' using 1:2:3
gnuplot> #we can zoom into the center
```

```
gnuplot> set xr[-0.5:0.5]
gnuplot> set yr[-0.5:0.5]
gnuplot> set zr[-0.5:0.5]
gnuplot> splot 'output.txt' using 1:2:3:4:5:6 with vectors nohead, 'output.txt' using 1:2:3
```



7.6 Path 2

7.7 Defining the legacy interface

We define our code interface so that a user can add, update and delete particles, start the nearest neighbors finding algorithm and retrieve the ids of the nearest neighbors.

To define the interface, open interface.py with your favorite editor and replace the contents of this file with:

```
from amuse.community import *

class NearestNeighborInterface(CodeInterface):

    use_modules = ['NN']
```

```
def __init__(self, **keyword_arguments):
    CodeInterface.__init__(self, **keyword_arguments)

@legacy_function
def commit_parameters():
    function = LegacyFunctionSpecification()
    function.result_type = 'int32'
    return function

@legacy_function
def set_maximum_number_of_particles():
    function = LegacyFunctionSpecification()
    function.addParameter('value', dtype='float64', direction=function.IN)
    function.result_type = 'int32'
    return function

@legacy_function
def get_maximum_number_of_particles():
    function = LegacyFunctionSpecification()
    function.addParameter('value', dtype='float64', direction=function.OUT)
    function.result_type = 'int32'
    return function

@legacy_function
def new_particle():
    function = LegacyFunctionSpecification()
    function.can_handle_array = True
    function.addParameter('index_of_the_particle', dtype='int32', direction=function.OUT)
    function.addParameter('x', dtype='float64', direction=function.IN)
    function.addParameter('y', dtype='float64', direction=function.IN)
    function.addParameter('z', dtype='float64', direction=function.IN)
    function.result_type = 'int32'
    return function

@legacy_function
def delete_particle():
    function = LegacyFunctionSpecification()
    function.addParameter('index_of_the_particle', dtype='int32', direction=function.IN)
    function.result_type = 'int32'
    return function

@legacy_function
def get_state():
    function = LegacyFunctionSpecification()
    function.can_handle_array = True
    function.addParameter('index_of_the_particle', dtype='int32', direction=function.IN)
    function.addParameter('x', dtype='float64', direction=function.OUT)
    function.addParameter('y', dtype='float64', direction=function.OUT)
    function.addParameter('z', dtype='float64', direction=function.OUT)
    function.result_type = 'int32'
    return function

@legacy_function
def set_state():
    function = LegacyFunctionSpecification()
    function.can_handle_array = True
    function.addParameter('index_of_the_particle', dtype='int32', direction=function.IN)
    function.addParameter('x', dtype='float64', direction=function.IN)
    function.addParameter('y', dtype='float64', direction=function.IN)
    function.addParameter('z', dtype='float64', direction=function.IN)
    function.result_type = 'int32'
    return function
```

```
    @legacy_function
    def run():
        function = LegacyFunctionSpecification()
        function.result_type = 'int32'
        return function

    @legacy_function
    def get_close_neighbors():
        function = LegacyFunctionSpecification()
        function.can_handle_array = True
        function.addParameter('index_of_the_particle', dtype='int32', direction=function.IN)
        function.addParameter('index_of_first_neighbor', dtype='float64', direction=function.OUT)
        function.addParameter('index_of_second_neighbor', dtype='float64', direction=function.OUT)
        function.addParameter('index_of_third_neighbor', dtype='float64', direction=function.OUT)
        function.result_type = 'int32'
        return function

    @legacy_function
    def get_nearest_neighbor():
        function = LegacyFunctionSpecification()
        function.can_handle_array = True
        function.addParameter('index_of_the_particle', dtype='int32', direction=function.IN)
        function.addParameter('index_of_the_neighbor', dtype='float64', direction=function.OUT)
        function.addParameter('distance', dtype='float64', direction=function.OUT)
        function.result_type = 'int32'
        return function

    @legacy_function
    def get_number_of_particles():
        function = LegacyFunctionSpecification()
        function.can_handle_array = True
        function.addParameter('value', dtype='int32', direction=function.OUT)
        function.result_type = 'int32'
        return function


class NearestNeighbor(CodeInterface):

    def __init__(self):
        CodeInterface.__init__(self,  NearestNeighborInterface())
```

We can generate a stub from the interface code with:

```
> $AMUSE_DIR/build.py --type=f90 --mode=stub interface.py NearestNeighborInterface -o interface.f
```

The generated **interface.f90** replaces the original file generated in the previous section.

We will be implementing the interface.f90 file as a module and we need to add the module definition to this file. We do this by adding a `MODULE NN` line to the beginning of the file and a `END MODULE` line to the end of the file. To do so open the interface.90 file and append/prepend the following code:

```
MODULE NN

CONTAINS

!.. original code

END MODULE
```

The code builds, but does not have any functionality yet:

```
> make clean
> make all
```

Note: Compiling the interface code will result in a lot of warnings about unused dummy arguments. These warnings can be savely ignored for now.

The tests are broken (the echo_int function has been removed):

```
> nosetests
E
======================================================================
ERROR: test1 (nearestneighbor.test_nearestneighbor.NearestNeighborInterfaceTests)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "../src/amuse/test/amusetest.py", line 146, in run
    testMethod()
  File "nearestneighbor/test_nearestneighbor.py", line 11, in test1
    result,error = instance.echo_int(12)
AttributeError: 'NearestNeighborInterface' object has no attribute 'echo_int'

----------------------------------------------------------------------
Ran 1 test in 0.315s

FAILED (errors=1)
```

Let's create a working test by calling the new_particle method, open an editor on the test_nearestneighbor.py file and replace the test1 method with:

```
def test1(self):
    instance = NearestNeighborInterface()
    result,error = instance.new_particle(1.0, 1.0, 2.0)
    self.assertEquals(error, 0)
    self.assertEquals(result, 1)
    instance.stop()
```

As this is python code we do not need to rebuild the code, instead we can run the tests right after saving the code. Unfortunately, when we run the test, it still fails.

```
> nosetests
F
======================================================================
FAIL: test1 (nearestneighbor.test_nearestneighbor.NearestNeighborInterfaceTests)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "/src/amuse/test/amusetest.py", line 146, in run
    testMethod()
  File "/nearestneighbor/test_nearestneighbor.py", line 13, in test1
    self.assertEquals(result, 1)
  File "/src/amuse/test/amusetest.py", line 62, in failUnlessEqual
    self._raise_exceptions_if_any(failures, first, second, '{0} != {1}', msg)
  File "/src/amuse/test/amusetest.py", line 49, in _raise_exceptions_if_any
    raise self.failureException(msg or err_fmt_string.format(first, second, *args))
AssertionError: 0 != 1
-------------------- >> begin captured logging << --------------------
legacy: INFO: start call 'NearestNeighborInterface.new_particle'
legacy: INFO: end call 'NearestNeighborInterface.new_particle'
--------------------- >> end captured logging << ---------------------

----------------------------------------------------------------------
Ran 1 test in 0.319s
```

When you look closely at the output of the test you see that the result from the method is 0 and not the excpeted 1. We need to edit the fortan code to make this test work. Open an editor on interface.f90 and go to the *new_particle* function.

```
FUNCTION new_particle(index_of_the_particle, x, y, z)
  INTEGER :: index_of_the_particle
  DOUBLE PRECISION :: x, y, z
  index_of_the_particle = 1
  new_particle=0
END FUNCTION
```

Note: In AMUSE all interface functions return an errorcode. Any other return values must be passed through the arguments of the functions.

We need to rebuild the code, and after building run the tests.

```
> make all
> nosetests
.
----------------------------------------------------------------------
Ran 1 test in 0.427s

OK
```

The are tests work again! Only, we do not have any real working legacy code.

7.8 Filling the stubs

The implementation of the algorithm does not match the interface we defined and created. We need to write some glue code to connect the code with the interface. To do so we fill in the stubs generated earlier.

Open the **interface.f90** file in your favorite editor and change its contents to:

```
MODULE NN

integer, DIMENSION(:), ALLOCATABLE :: identifiers
integer, DIMENSION(:,:), ALLOCATABLE :: nearest_neighbor
DOUBLE PRECISION, DIMENSION(:), ALLOCATABLE :: particle_y
DOUBLE PRECISION, DIMENSION(:), ALLOCATABLE :: particle_x
DOUBLE PRECISION, DIMENSION(:), ALLOCATABLE :: particle_z
DOUBLE PRECISION, DIMENSION(:), ALLOCATABLE :: distances
integer :: maximum_number_of_particles
integer :: number_of_particles_allocated

CONTAINS

FUNCTION get_nearest_neighbor(index_of_the_particle, index_of_the_neighbor,  &
    distance)
  IMPLICIT NONE
  INTEGER :: index_of_the_particle
  DOUBLE PRECISION :: index_of_the_neighbor, distance

  INTEGER :: get_nearest_neighbor

  distance = distances(index_of_the_particle)
  index_of_the_neighbor = nearest_neighbor(index_of_the_particle, 0)

  get_nearest_neighbor=0
END FUNCTION

FUNCTION new_particle(index_of_the_particle, x, y, z)
  IMPLICIT NONE
  INTEGER :: index_of_the_particle
  DOUBLE PRECISION :: x, y, z
```

```
  INTEGER :: new_particle, i

  index_of_the_particle = -1
  DO i = 1, maximum_number_of_particles
    IF (identifiers(i).EQ.-1) THEN
        identifiers(i) = i
        index_of_the_particle = i
        number_of_particles_allocated = number_of_particles_allocated + 1
        EXIT
    END IF
  END DO

  IF (index_of_the_particle.EQ.-1) THEN
      new_particle = -1
  ELSE
      particle_x(index_of_the_particle) = x
      particle_y(index_of_the_particle) = y
      particle_z(index_of_the_particle) = z
      new_particle = 0
  END IF
END FUNCTION

FUNCTION get_close_neighbors(index_of_the_particle,  &
    index_of_first_neighbor, index_of_second_neighbor,  &
    index_of_third_neighbor)
  IMPLICIT NONE
  INTEGER :: index_of_the_particle
  DOUBLE PRECISION :: index_of_first_neighbor, index_of_second_neighbor
  DOUBLE PRECISION :: index_of_third_neighbor

  INTEGER :: get_close_neighbors

  index_of_first_neighbor = nearest_neighbor(index_of_the_particle, 1)
  index_of_second_neighbor = nearest_neighbor(index_of_the_particle, 2)
  index_of_third_neighbor = nearest_neighbor(index_of_the_particle, 3)

  get_close_neighbors=0
END FUNCTION

FUNCTION delete_particle(index_of_the_particle)
  IMPLICIT NONE
  INTEGER :: index_of_the_particle

  INTEGER :: delete_particle

  identifiers(index_of_the_particle) = -1
  number_of_particles_allocated = number_of_particles_allocated - 1

  delete_particle=0
END FUNCTION

FUNCTION set_state(index_of_the_particle, x, y, z)
  IMPLICIT NONE
  INTEGER :: index_of_the_particle
  DOUBLE PRECISION :: x, y, z

  INTEGER :: set_state

  particle_x(index_of_the_particle) = x
  particle_y(index_of_the_particle) = y
  particle_z(index_of_the_particle) = z

  set_state=0
```

```
END FUNCTION

FUNCTION get_state(index_of_the_particle, x, y, z)
  IMPLICIT NONE
  INTEGER :: index_of_the_particle
  DOUBLE PRECISION :: x, y, z

  INTEGER :: get_state

  x = particle_x(index_of_the_particle)
  y = particle_y(index_of_the_particle)
  z = particle_z(index_of_the_particle)

  get_state=0
END FUNCTION

FUNCTION get_number_of_particles(value)
  IMPLICIT NONE
  INTEGER :: value

  INTEGER :: get_number_of_particles
  value = number_of_particles_allocated
  get_number_of_particles=0
END FUNCTION

FUNCTION run()
  IMPLICIT NONE
  INTEGER :: run
  INTEGER, DIMENSION(:), ALLOCATABLE :: index_to_table
  DOUBLE PRECISION, DIMENSION(:), ALLOCATABLE :: x
  DOUBLE PRECISION, DIMENSION(:), ALLOCATABLE :: y
  DOUBLE PRECISION, DIMENSION(:), ALLOCATABLE :: z
  INTEGER, DIMENSION(:), ALLOCATABLE :: nn1
  INTEGER, DIMENSION(:), ALLOCATABLE :: nn2
  INTEGER, DIMENSION(:), ALLOCATABLE :: nn3
  INTEGER :: i, j, find_nearest_neighbors

  ALLOCATE(x(number_of_particles_allocated))
  ALLOCATE(y(number_of_particles_allocated))
  ALLOCATE(z(number_of_particles_allocated))
  ALLOCATE(nn1(number_of_particles_allocated))
  ALLOCATE(nn2(number_of_particles_allocated))
  ALLOCATE(nn3(number_of_particles_allocated))
  ALLOCATE(index_to_table(number_of_particles_allocated))

  j = 0
  DO i = 1, maximum_number_of_particles
    IF (identifiers(i).NE.-1) THEN
        j = j + 1
        x(j) = particle_x(i)
        y(j) = particle_y(i)
        z(j) = particle_z(i)
        index_to_table(j) = i
    END IF
  END DO

  run = find_nearest_neighbors(number_of_particles_allocated, x, y, z, nn1, nn2, nn3)

  DO j = 1, number_of_particles_allocated
    i = index_to_table(j)
    nearest_neighbor(i, 1) = index_to_table(nn1(j))
    nearest_neighbor(i, 2) = index_to_table(nn2(j))
    nearest_neighbor(i, 3) = index_to_table(nn3(j))
```

```
  END DO

  DEALLOCATE(x)
  DEALLOCATE(y)
  DEALLOCATE(z)
  DEALLOCATE(nn1)
  DEALLOCATE(nn2)
  DEALLOCATE(nn3)
  DEALLOCATE(index_to_table)
END FUNCTION

FUNCTION set_maximum_number_of_particles(value)
  IMPLICIT NONE
  DOUBLE PRECISION :: value

  INTEGER :: set_maximum_number_of_particles
  maximum_number_of_particles = value
  set_maximum_number_of_particles=0
END FUNCTION

FUNCTION commit_parameters()
  IMPLICIT NONE

  INTEGER :: commit_parameters, i

  ALLOCATE(particle_x(maximum_number_of_particles))
  ALLOCATE(particle_y(maximum_number_of_particles))
  ALLOCATE(particle_z(maximum_number_of_particles))
  ALLOCATE(distances(maximum_number_of_particles))
  ALLOCATE(nearest_neighbor(maximum_number_of_particles, 3))
  ALLOCATE(identifiers(maximum_number_of_particles))

  DO i = 1, maximum_number_of_particles
    identifiers(i) = -1
  END DO

  number_of_particles_allocated = 0
  commit_parameters=0
END FUNCTION

FUNCTION get_maximum_number_of_particles(value)
  IMPLICIT NONE
  DOUBLE PRECISION :: value

  INTEGER :: get_maximum_number_of_particles
  value = maximum_number_of_particles
  get_maximum_number_of_particles=0
END FUNCTION

END MODULE
```

Test if the code builds and try it out. In the legacy interface directory do:

```
> make clean
> make all
```

The tests will still fail as we need to set the “maximum_number_of_particles” before allocating any new particles. Let’s update the test, open an editor on test_nearestneighbor.py file and update the test1 method to:

```
def test1(self):
    instance = NearestNeighborInterface()
    instance.set_maximum_number_of_particles(100)
    instance.commit_parameters()
    result,error = instance.new_particle(1.0, 1.0, 2.0)
```

```
    self.assertEquals(error, 0)
    self.assertEquals(result, 1)
    instance.stop()
```

Now the tests should succeed:

```
> nosetests
.
----------------------------------------------------------------------
Ran 1 test in 0.311s

OK
```

Let's check some more functionaly by adding another test

```
def test2(self):
    instance = NearestNeighborInterface()
    instance.set_maximum_number_of_particles(2)
    instance.commit_parameters()
    result,error = instance.new_particle(1.0, 1.0, 2.0)
    self.assertEquals(error, 0)
    self.assertEquals(result, 1)
    result,error = instance.new_particle(2.0, 3.0, 2.0)
    self.assertEquals(error, 0)
    self.assertEquals(result, 2)
    result,error = instance.new_particle(2.0, 3.0, 2.0)
    self.assertEquals(error, -1)
    error = instance.delete_particle(1)
    self.assertEquals(error, 0)
    result,error = instance.new_particle(2.0, 3.0, 2.0)
    self.assertEquals(error, 0)
    self.assertEquals(result, 1)
    instance.stop()
```

Now the tests should succeed:

```
> nosetests
..
----------------------------------------------------------------------
Ran 2 tests in 0.448s

OK
```

We now have done everything in Step 0 *Legacy Interface*. We have a legacy code and can access it in our python script. But, our interface is not very friendly to work with. We have to think about errorcodes and we have not information about units. To make our interface easier to works with we start defining methods, properties and parameters.

7.9 Defining methods

The object oriented interface is also defined in the **interface.py**. So, we continue by opening an editor on this file. We will be writing methods for the **NearestNeighbor** class, in your editor seek this code (at the end of the file):

```
class NearestNeighbor(CodeInterface):

    def __init__(self, **options):
        CodeInterface.__init__(self,  NearestNeighborInterface(), **options)
```

We will start by defining methods, we will do this by implementing the **define_methods** function, like so:

```
class NearestNeighbor(CodeInterface):

    def __init__(self, **options):
```

```
        CodeInterface.__init__(self,  NearestNeighborInterface(), **options)


    def define_methods(self, builder):

        builder.add_method(
            "new_particle",
            (generic_unit_system.length, generic_unit_system.length, generic_unit_system.length,),
            (builder.INDEX, builder.ERROR_CODE)
        )

        builder.add_method(
            "delete_particle",
            (builder.INDEX,),
            (builder.ERROR_CODE)
        )

        builder.add_method(
            "get_state",
            (builder.INDEX,),
            (generic_unit_system.length, generic_unit_system.length, generic_unit_system.length,
            public_name = "get_position"
        )

        builder.add_method(
            "set_state",
            (builder.INDEX, generic_unit_system.length, generic_unit_system.length, generic_unit_
            (builder.ERROR_CODE),
            public_name = "set_position"
        )

        builder.add_method(
            "run",
            (),
            (builder.ERROR_CODE),
        )

        builder.add_method(
            "get_close_neighbors",
            (builder.INDEX,),
            (builder.INDEX, builder.INDEX, builder.INDEX, builder.ERROR_CODE),
        )

        builder.add_method(
            "get_nearest_neighbor",
            (builder.INDEX,),
            (builder.INDEX, generic_unit_system.length, builder.ERROR_CODE),
        )

        builder.add_method(
            "get_number_of_particles",
            (),
            (builder.NO_UNIT, builder.ERROR_CODE),
        )
```

With this code, we define the methods and specify how to interpret the arguments and return values. We get a special object (the **builder** object) that provides us with the **add_method** function to be able to this. The definition of the **add_method** function is as follows:

```
add_method(
    name of the original function in the legacy interface,
    list of arguments (unit or type),
    list of return values,
```

```
    public_name = name for the user of the class (optional)
)
```

For every argument or return value we can specify if it has a unit or if it is special. The special arguments are:

definition	description
builder.ERROR_CODE	the value is interpreted as an errorcode, zero means no error for all other values and Exception will be raise, only valid for one return value.
builder.NO_UNIT	the value has not unit (for example for the number of items in a list)
builder.INDEX	the value is interpreted as an index for object identifiers

Test if the code builds and try it out. In the legacy interface directory do:

```
> make clean
> make all
```

Let's add another test:

```
def test3(self):
    instance = NearestNeighbor()
    instance.set_maximum_number_of_particles(2)
    instance.commit_parameters()
    result = instance.new_particle(
        1.0 | generic_unit_system.length,
        2.0 | generic_unit_system.length,
        3.0 | generic_unit_system.length
    )
    self.assertEquals(result, 1)
    result = instance.new_particle(
        1.0 | generic_unit_system.length,
        1.0 | generic_unit_system.length,
        2.0 | generic_unit_system.length
    )
    self.assertEquals(result, 2)
    x,y,z = instance.get_position(1)
    self.assertEquals(1.0 | generic_unit_system.length, x)
    self.assertEquals(2.0 | generic_unit_system.length, y)
    self.assertEquals(3.0 | generic_unit_system.length, z)
    instance.stop()
```

And run the tests:

```
> nosetests
...
----------------------------------------------------------------------
Ran 3 tests in 0.664s

OK
```

As you can see our script is now a little simpler and we support units. We do not have to think about the errorcodes in this script, AMUSE will interpret the errorcodes and raise the right exceptions if needed. The units are also automatically converted to the right units for the code. But the script is still not very easy and we have to manage all the ids we get from the code. To make our code even easier to handle we will continue by defining a **set**.

Note: We skip defining parameters and properties, we will come back to this later in this tutorial.

7.10 Defining a set

We have made our interface a little easier but we still have to do a some management work in our script. We would like to work with objects and adding or removing these objects from the code. AMUSE supports this by defining **sets**. Each set is capable of storing specific attributes of the objects in the set. Our code is capable of storing the

x, y and z position of an object. An object in AMUSE is called a *Particle* and the sets that contain these particles are called *ParticleSets* or shorter *Particles*.

We define our particle set by implementing a **define_particle_sets** function on our **NearestNeighbor** class like so:

```
class NearestNeighbor(CodeInterface):

    def __init__(self, **options):
        CodeInterface.__init__(self,  NearestNeighborInterface(), **options)


    def define_methods(self, builder):
        ...

    def define_particle_sets(self, builder):
        builder.define_set('particles', 'index_of_the_particle')
        builder.set_new('particles', 'new_particle')
        builder.set_delete('particles', 'delete_particle')
        builder.add_setter('particles', 'set_position')
        builder.add_getter('particles', 'get_position')
        builder.add_getter('particles', 'get_close_neighbors', names=('neighbor0', 'neighbor1', '
```

That's all, we now have defined a set called **particles**. Again, we get a builder object to use in defining our set. All methods have the name of the set as their first argument, this name can be any name you want, but in AMUSE most codes provide a set called **particles**. For the **add_setter**, **add_getter**, **set_new** and **set_delete** functions, the second argument is the name of the method we defined in the previous step. Finally you can set the name of the attribute in the particles set with the **names** argument. This is optional for legacy functions, if not given the names of the attributes will be derived from the names of the arguments in the orginal calls. For example, the **get_position** call we specified earlier has parameter name **x**, **y** and **z**, these names are also used in the particles set.

Test if the code builds and try it out. In the legacy interface directory do:

```
> make clean
> make all
```

Let's add another test:

```
def test4(self):
    instance = NearestNeighbor()
    instance.set_maximum_number_of_particles(100)
    instance.commit_parameters()

    particles = data.Particles(4)
    particles.x = [0.0, 1.0, 4.0, 7.5] | generic_unit_system.length
    particles.y = 0.0 | generic_unit_system.length
    particles.z = 0.0 | generic_unit_system.length

    instance.particles.add_particles(particles)
    instance.run()

    self.assertEqual(instance.particles[0].neighbor0, instance.particles[1])
    self.assertEqual(instance.particles[1].neighbor0, instance.particles[0])
    self.assertEqual(instance.particles[2].neighbor0, instance.particles[1])
    self.assertEqual(instance.particles[3].neighbor0, instance.particles[2])

    instance.stop()
```

The support for a particle set means we can now also interact with other parts of AMUSE. Let's make a plummer model and find the nearest neighbors in this model.

First make a file with the following contents, let's call this file **plummer2.py**:

```
from interface import NearestNeighbor
from amuse.lab import *
from amuse.io import text

if __name__ == '__main__':
    number_of_particles = 1000
    particles = new_plummer_sphere(1000)

    code = NearestNeighbor()
    code.particles.add_particles(particles)

    code.run()

    local_particles = code.particles.copy()
    delta = local_particles.neighbor1.position - local_particles.position

    local_particles.dx = delta[...,0]
    local_particles.dy = delta[...,1]
    local_particles.dz = delta[...,2]

    output = text.TableFormattedText("output.txt", set = local_particles)
    output.attribute_names = ['x','y','z', 'dx', 'dy','dz']
    output.store()
```

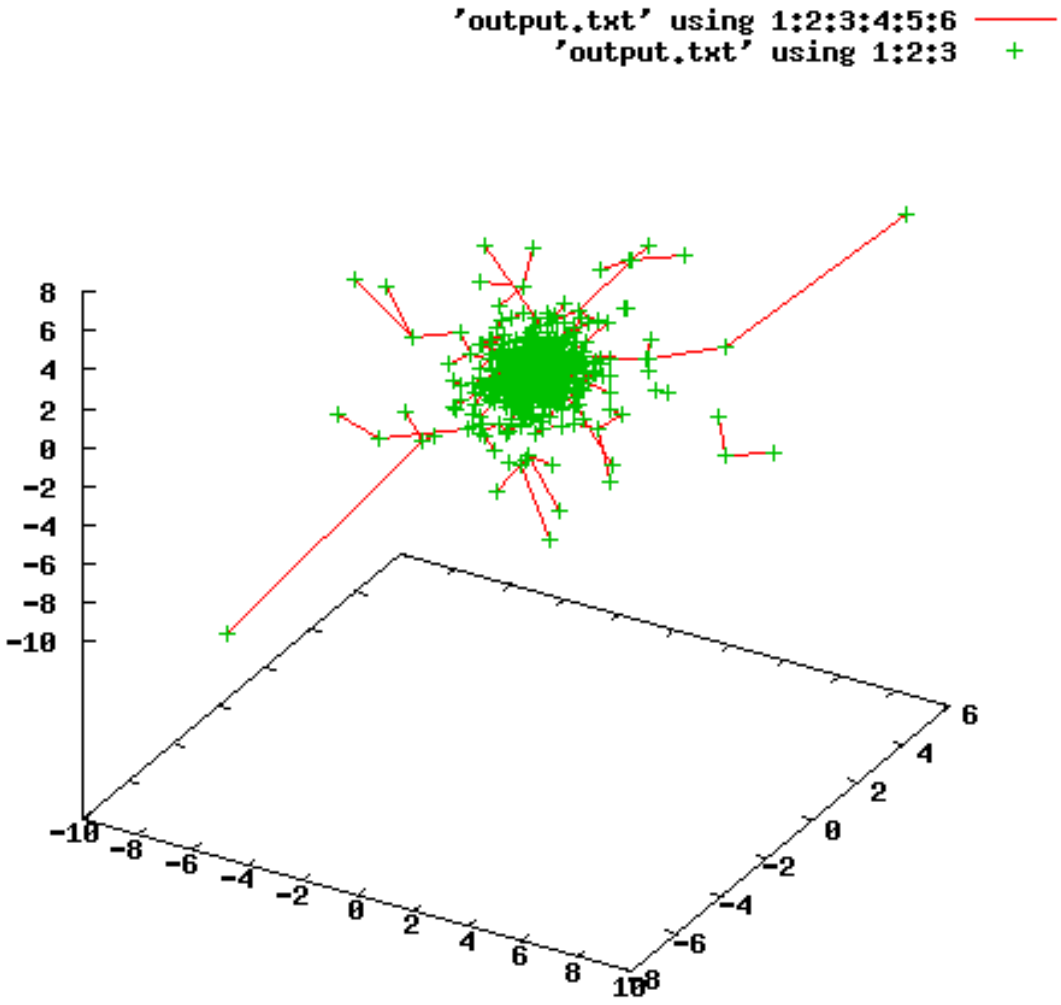
We can run this file with python:

```
.. code-block:: bash
```

$AMUSE_DIR/amuse.sh plummer2.py

It will create an **output.txt** file and we can show this file with gnuplot.

```
gnuplot> splot 'output.txt' using 1:2:3:4:5:6 with vectors nohead, 'output.txt' using 1:2:3
gnuplot> #we can zoom into the center
gnuplot> set xr[-0.5:0.5]
gnuplot> set yr[-0.5:0.5]
gnuplot> set zr[-0.5:0.5]
gnuplot> splot 'output.txt' using 1:2:3:4:5:6 with vectors nohead, 'output.txt' using 1:2:3
```



8 Adding a Gravitational Dynamics Code

In this tutorial we explain the steps to take when integrating a new gravitational dynamics code.

> **Warning:** This tutorial will show the steps but will not finish in a working product. It will highlight the steps to take, we recommend you first read these tutorials *Integrate a C++ code* and *Integrate a Fortran 90 code*.

8.1 Environment variables

To simplify the work in the coming sections, we first define the environment variable 'AMUSE_DIR'. This environment variable must point to the root directory of AMUSE (this is the directory containing the build.py script).

```
> export AMUSE_DIR=<path to the amuse root directory>
```

or in a c shell:

```
> setenv AMUSE_DIR <path to the amuse root directory>
```

After building the code, we want to run and test the code. Check if amuse is available in your python path by running the following code on the command line.

```
> python -c "import amuse"
Traceback (most recent call last):
File "<string>", line 1, in <module>
ImportError: No module named amuse
```

If this code ends in a "ImportError" as shown in the example, the PYTHONPATH environment variable must be extended with the src directory in AMUSE_DIR. We can do so by using one of the following commands.

```
> export PYTHONPATH=${PYTHONPATH}:${AMUSE_DIR}/src
```

or in a c shell:

```
> setenv PYTHONPATH ${PYTHONPATH}:${AMUSE_DIR}/src
```

8.2 Creating an initial directory structure

First we need to create a directory for our project and put some files in it to help build the code. The fastest method to setup the directory is by using the build.py script.

```
> # for C/C++ codes:
> $AMUSE_DIR/build.py --type=c --mode=dir CodeName
> # for fortran codes:
> $AMUSE_DIR/build.py --type=f90 --mode=dir CodeName
```

Note: The `CodeName` should be an unique name for your code. It should follow python rules for class names, start with an uppercase letter and have an uppercase letter for each word in the name (camelcase).

This script will create a direcory called `codename`. The script will populate the directory with a make file (`Makefile`), a python interface definition file (`interface.py`) and a C or fortran stub file (containing methods implementing the definitions in interface.py, `interface.cc` or `interface.f90`).

8.3 The src directory

The build.py script creates a `src` directory in the `codename` directory. The `src` directory should contain a make file and the sources of the code to integrate. (The build.py script also creates some files in the `src` directory as an example, but you can remove and overwrite everything in this directory)

The generated build system will call the make file with the `all` target (to build an stand-alone executable, if available) and the `libcodename.a` target (to build a static library with all necessary object files).

Note: If your code has a different build system or if the makefile is not in the main directory of the code you need to edit the generated make file in the `codename` directory. The AMUSE codebase contains several codes that have a different build system, please look into the code for `evtwin` (uses cmake) or `athena` (downloads and patches code) for hints.

If you already have a make file, you might need to add the `libcodename.a` target. You can define this target by adding the following code to your make file at the appropraite locations in the makefile.

... code-block:: Makefile CODELIB = libcodename.a

CODEOBJS = ...list of object files...

AR = ar ruv RANLIB = ranlib

$(CODELIB): $(CODEOBJS) $(RM) -f $@ $(AR) $@ $(CODEOBJS) $(RANLIB) $@

9 Create an low-level Interface to a Code

In this tutorial we will create an interface to a code.

> **Warning:** This tutorial does not use the build script provided with AMUSE. All files and directories need to be created "by hand". Use this tutorial if you want to get a deeper understanding of how the build process works and which steps are involved in creating a low level interface. To learn how to create an interface to a code we recommend *Integrate a C++ code* and *Integrate a Fortran 90 code*.

9.1 Work directory

We start by making a directory for our code. This directory should be a subdirectory of the "src/amuse/community" directory. It also will be a python package and we need to create the file "__init__.py" in this directory. So, let's open a shell and go to the AMUSE root directory. To create the code directory we then do:

```
>> cd src/amuse/community
>> mkdir mycode
>> cd mycode
>> touch __init__.py
>> pwd
../src/amuse/community/mycode
>> ls
__init__.py
```

9.2 The code

To create an interface we first need the code. For this example we will use a very simple code do some calculations on two numbers.

The contents of the code.c file:

```
#include "code.h"

double sum(double x, double y) {
  return x + y;
}

int divide(double x, double y, double * result) {
  if(y == 0.0) {
      return -1;
  } else {
      *result = x / y;
      return 0;
  }
}
```

We need to access these function from another C file, so we need to define a header file.

The contents of the code.h file:

```
double sum(double x, double y);

int divide(double x, double y, double * result);
```

9.3 The interface code

Now we can define the interface class for our code in python. An interface needs to inherit from the class "CodeInterface".

```
1  from amuse.community import *
2
3  class MyCode(CodeInterface):
4      include_headers = ['code.h']
5
6      def __init__(self):
7          CodeInterface.__init__(self)
```

In this example we first import names from the `amuse.community` module on line 1. The `amuse.community` module defines the typical classes and function needed to write a legacy interface. On line 3 we define our class and inherit from `CodeInterface`. The class will be used to generate a C++ file. In this file we will need the definitions of our legacy functions. So, on line 4 we specify the necessary include files in a array of strings. Each string will be converted to an include statement.

9.4 Building the code

Building the code takes a couple of steps, first generating the C file and then compiling the code. We will construct a makefile to automate the build process.

```
1  ifndef AMUSE_DIR
2      AMUSE_DIR=../../../..
3  endif
4
5  CODE_GENERATOR = $(AMUSE_DIR)/build.py
6
7  CXXFLAGS = -Wall -g -DTOOLBOX  $(MUSE_INCLUDE_DIR)
8  LDFLAGS = -lm $(MUSE_LD_FLAGS)
9
10 OBJS = code.o
11
12 all: worker_code
13
14 cleanall: clean
15     $(RM) worker_code *~
16
17 clean:
18     rm -f *.so *.o *.pyc worker_code.cc
19
20 worker_code.cc: interface.py
21     $(CODE_GENERATOR) --type=c interface.py MyCode -o $@
22
23 worker_code: worker_code.cc $(OBJS)
24     mpicxx $@.cc $(OBJS) -o $@
25
26 .cc.o: $<
27     g++ $(CXXFLAGS) -c -o $@ $<
28
29 .c.o: $<
30     g++ $(CXXFLAGS) -c -o $@ $<
```

Let's start make and build the `worker_code` application

```
>> make clean
>> make
...
mpicxx worker_code.cc code.o -o worker_code
>> ls
... worker_code ...
```

9.5 Running the code

Before we run the code we need to start the MPI daemon process ‘’mpd’‘. This daemon process manages the start of child processes.

```
>> mpd &
```

We can use `amuse.sh` and try the interface.

```
>>> from amuse.community.mycode import interface
>>> instance = interface.MyCode()
>>> instance
<amuse.community.mycode.interface.MyCode object at 0x7f57abfb2550>
>>> del instance
```

We have not defined any methods and our interface class is not very useful. We can only create an instance of the code. When we create this instance the “worker_code” application will start to handle all the function calls. We can see the application running when we do “ps x | grep worker_code”

9.6 Implementing a method

Now we will define the `sum` method. We will add the definition to the MyCode class.

```
1  from amuse.community import *
2
3  class MyCode(CodeInterface):
4      include_headers = ['code.h']
5
6      def __init__(self):
7           CodeInterface.__init__(self)
8
9      @legacy_function
10     def sum():
11         function = LegacyFunctionSpecification()
12         function.addParameter('x', 'd', function.IN)
13         function.addParameter('y', 'd', function.IN)
14         function.result_type = 'd'
15         return function
```

The new code starts from line 9. On line 9 we specify a decorator. This decorator will convert the following function into a specification that can be used to call the function and generate the C++ code. On line 10 we give the function the same name as the function in our code. This function may not have any arguments. On line 11 we create an instance of the “LegacyFunctionSpecification” class, this class has methods to specify the intercase. On line 12 and 13 we specify the parameters for out functions. Parameters have a name, type and direction. The type is specified with a single character *type code*. The following type codes are defined:

Type code	C type	Fortran type
‘i’	int	integer
‘d’	double	double precision
‘f’	float	single precision

The direction of the parameter can be `IN`, `OUT` or `INOUT`. On line 14 we define the return type, this can be a *type code* or `None`. The default value is `None`, specifying no return value (void function).

Let’s rebuild the code.

```
>> make clean
>> make
...
mpicxx worker_code.cc code.o -o worker_code
```

We can now start ‘`amuse.sh`‘ again and do a simple sum

```
>>> from amuse.community.mycode import interface
>>> instance = interface.MyCode()
>>> instance.sum(40.5, 10.3)
50.799999999999997
>>> 40.5 + 10.3
50.799999999999997
>>> del instance
```

And we see that our interface correctly sums two numbers.

9.7 A method with an OUT parameter

We can complete out interface by defining the `divide` function.

```
1  from amuse.community import *
2
3  class MyCode(CodeInterface):
4      include_headers = ['code.h']
5
6      def __init__(self):
7          CodeInterface.__init__(self)
8
9      @legacy_function
10     def sum():
11         function = LegacyFunctionSpecification()
12         function.addParameter('x', 'd', function.IN)
13         function.addParameter('y', 'd', function.IN)
14         function.result_type = 'd'
15         return function
16
17     @legacy_function
18     def divide():
19         function = LegacyFunctionSpecification()
20         function.addParameter('x', 'd', function.IN)
21         function.addParameter('y', 'd', function.IN)
22         function.addParameter('result', 'd', function.OUT)
23         function.result_type = 'i'
24         return function
```

On line 22 we define the parameter "result" as an OUT parameter. In python we do not have to provide this parameter as an argument to our function. It After rebuilding we can try this new function.

```
>>> from amuse.community.mycode import interface
>>> instance = interface.MyCode()
>>> (result, error) =  instance.divide(10.2, 30.2)
>>> result
0.33774834437086093
>>> error
0
>>> del instance
```

We see that the function returns two values, the OUT parameter and also the return value of the function.

9.8 Working with arrays

Some functions will be called to perform on the elements of an array. For example:

```
>>> from amuse.community.mycode import interface
>>> instance = interface.MyCode()
>>> x_values = [1.0, 2.0, 3.0, 4.0, 5.0]
>>> y_values = [10.3, 20.3, 30.4 , 40.4, 50.6]
```

```
>>> results = []
>>> for x , y in map(None, x_values, y_values):
...     results.append(instance.sum(x,y))
...
>>> print results
[11.300000000000001, 22.300000000000001, 33.399999999999999,
44.399999999999999, 55.600000000000001]
```

The MPI message passing overhead is incurred for every call on the method. We can change this by specifing the function to be able to handle arrays.

```
1  from amuse.community import *
2
3  class MyCode(CodeInterface):
4      include_headers = ['code.h']
5
6      def __init__(self):
7          CodeInterface.__init__(self)
8
9      @legacy_function
10     def sum():
11         function = LegacyFunctionSpecification()
12         function.addParameter('x', 'd', function.IN)
13         function.addParameter('y', 'd', function.IN)
14         function.result_type = 'd'
15         function.can_handle_array = True
16         return function
```

On line 15 we specify that the function can be called with an array of values. The function will be called for every element of the array. The array will be send in one MPI message, reducing the overhead.

Let's rebuild the code and run an example.

```
>>> from amuse.community.mycode import interface
>>> instance = interface.MyCode()
>>> x_values = [1.0, 2.0, 3.0, 4.0, 5.0]
>>> y_values = [10.3, 20.3, 30.4 , 40.4, 50.6]
>>> results = instance.sum(x_values, y_values)
>>> print results
[ 11.3  22.3  33.4  44.4  55.6]
```

9.9 Other interfaces

The community codes directory contains a number of codes. Please look at these codes to see how the interfaces are defined.

10 Using Blender for visualising Amuse results

10.1 Blender

Blender is a free open source 3D content creation suite, *GPL-ed*. As it is specialised in visualisation of 3D content *and* scriptable in python it provides handy a tool for visualisation of Amuse data.

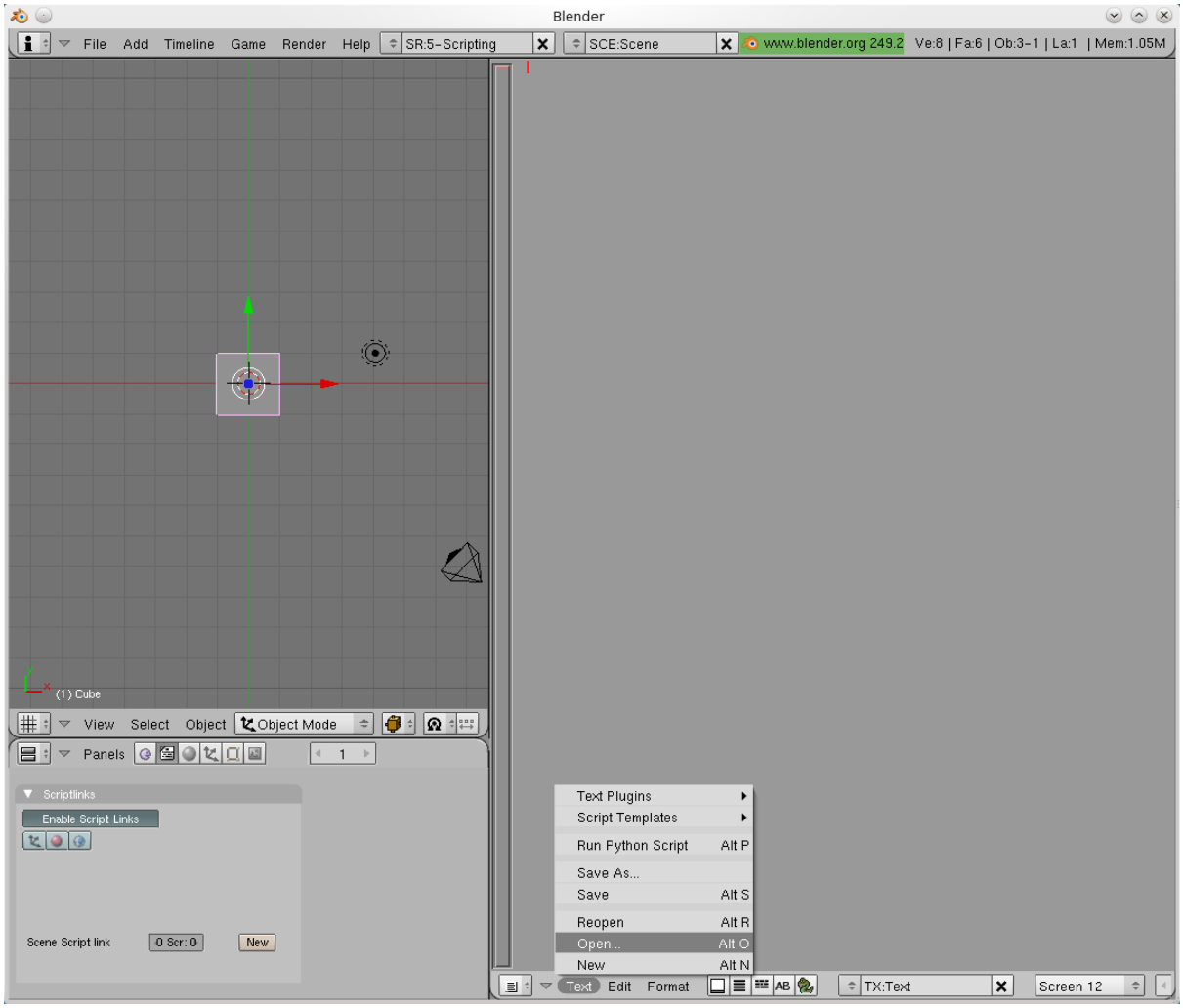
More information on Blender can be found on http://www.blender.org/

10.2 Starting Blender

We start blender from the command line, assuming all environment variables are set for amuse, and make sure mpd is running.

```
>>blender &
```

After blender opened we press **CTRL-RIGHTCURSOR** three times to open a text panel next to our 3D view. In the text panel we use the Text menu item to open a text file:



This way we can retrieve scripts we had written before. Now, we are going to write a new script, but use our favourite editor, and once finished we will load the script using the method described above.

Note: By default, blender places a cube in the scene. You can delete it else it will obscure the sun in our example.

The script is based on the sun-earth test in Hermite. We start with that amuse script and add a couple of lines to communicate with blender, which are marked by comments in the example below.

10.3 Amuse blender API

To simplify communication with blender, amuse has the module amuse.ext.blender, which contains a very basic API. In our example we start with importing it on top of the regular stellar dynamics imports:

```
from amuse.ext.blender import blender
```

To create a sphere, e.g. the sun, with 32 segments, 32 rings and a radius of 1.0 in the current scene we type:

```
sun = blender.Primitives.sphere(segments = 32, rings = 32, radius = 1.0)
```

We can move and rotate our object using:

```
x,y,z = 1.0, 0.0, 0.0
alpha = 0.2
sun.loc = (x, y, z)
sun.rotZ = alpha
```

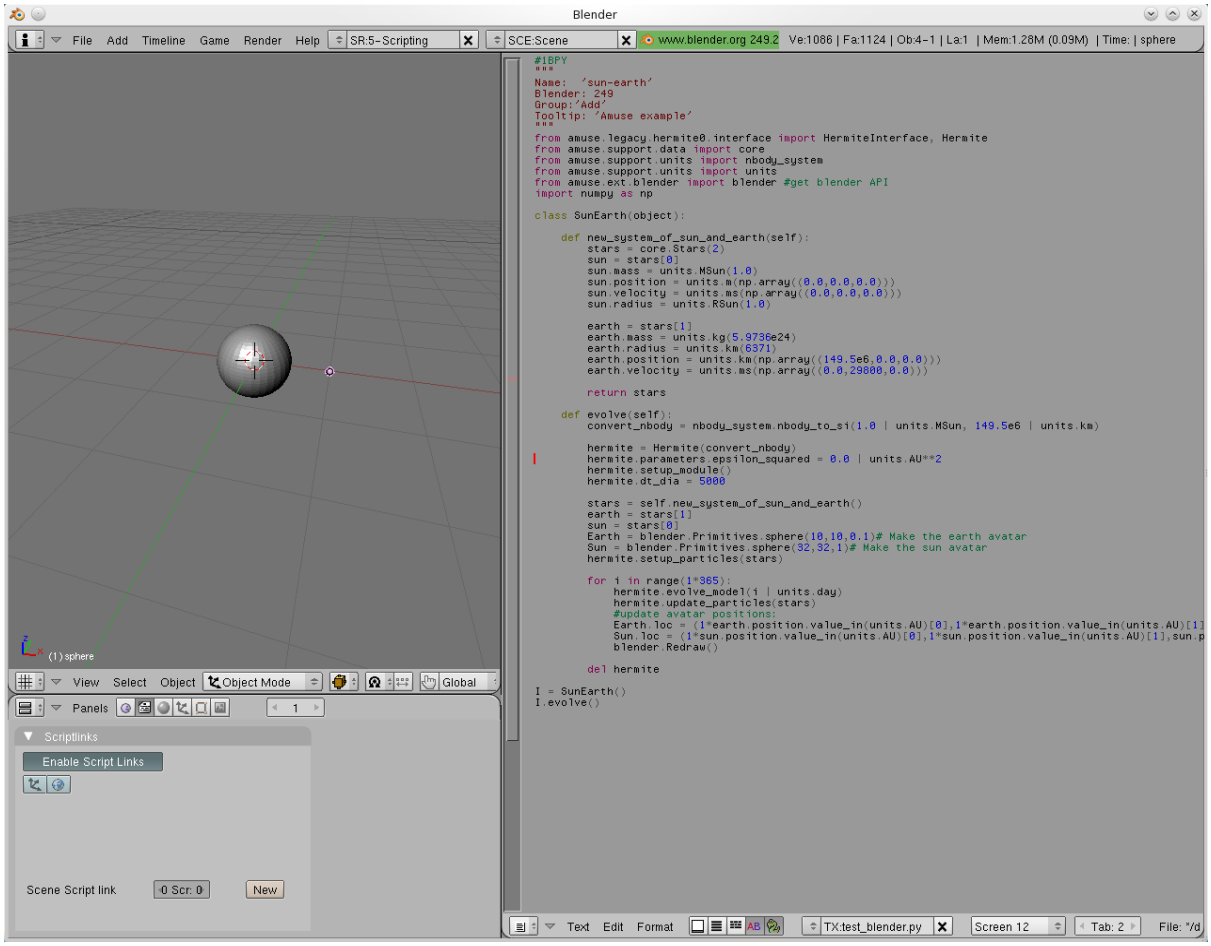
The Hermite Sun-Earth model with blender visualisation will become like this:

```
#1BPY
"""
Name:  'sun-earth'
Blender: 249
Group:'Add'
Tooltip: 'Amuse example'
"""
from amuse.community.hermite0.interface import HermiteInterface, Hermite
from amuse.units import nbody_system
from amuse.units import units
from amuse.ext.blender import blender #get blender API
import numpy as np

from amuse import datamodel
class SunEarth(object):

    def new_system_of_sun_and_earth(self):
        stars = datamodel.Stars(2)
        sun = stars[0]
        sun.mass = units.MSun(1.0)
        sun.position = units.m(np.array((0.0,0.0,0.0)))
        sun.velocity = units.ms(np.array((0.0,0.0,0.0)))
        sun.radius = units.RSun(1.0)

        earth = stars[1]
        earth.mass = units.kg(5.9736e24)
        earth.radius = units.km(6371)
        earth.position = units.km(np.array((149.5e6,0.0,0.0)))
        earth.velocity = units.ms(np.array((0.0,29800,0.0)))

        return stars

    def evolve_model(self):
        convert_nbody = nbody_system.nbody_to_si(1.0 | units.MSun, 149.5e6 | units.km)

        hermite = Hermite(convert_nbody)
        hermite.initialize_code()

        hermite.parameters.epsilon_squared = 0.0 | units.AU**2

        stars = self.new_system_of_sun_and_earth()
        earth = stars[1]
        sun = stars[0]
        Earth = blender.Primitives.sphere(10,10,0.1)# Make the earth avatar
        Sun = blender.Primitives.sphere(32,32,1)# Make the sun avatar
        hermite.particles.add_particles(stars)

        for i in range(1*365):
            hermite.evolve_model(i | units.day)
```

```
            hermite.particles.copy_values_of_all_attributes_to(stars)
            #update avatar positions:
            Earth.loc = (1*earth.position.value_in(units.AU)[0],1*earth.position.value_in(units.AU
            Sun.loc = (1*sun.position.value_in(units.AU)[0],1*sun.position.value_in(units.AU)[1],s
            blender.Redraw()

        hermite.print_refs()
        hermite.stop()

if __name__ == '__main__':
    I = SunEarth()
    I.evolve_model()
```

The path to this example code is {amusedir}/trunk/examples/applications/test_blender.py

We save this file as *myname.py*, open it in blender and run it by typing **ALT-P** (cursor in editor):



11 Plotting with amuse

11.1 matplotlib

Matplotlib is a python plotting library capable of working with many graphical user interface toolkits. It is not required by AMUSE, but if installed then AMUSE provides extended plot functionality. If a plot is made, axis labels will be made automatically yielding the concerning units.

To use matplotlib within AMUSE with the extended plot functionality you need the following import:

```
>>> from amuse.plot import *
```

The native matplotlib plot functions are still available in the native_plot namespace, e.g.:

```
>>> native_plot.subplot(2,2,1)
```

matplotlib documentation

install matplotlib

Either use a pre-packaged version or install from source. If you install from source and you have installed the prerequisites in a user directory make sure the `PATH` settings are correct.

The source can be found here

Installation instructions can be found here

Latex support

Latex support for labels can be enabled by issuing:

```
>>> latex_support()
```

This command will temporarily change the matplotlibrc settings:

```
rc('text', usetex=True)
```

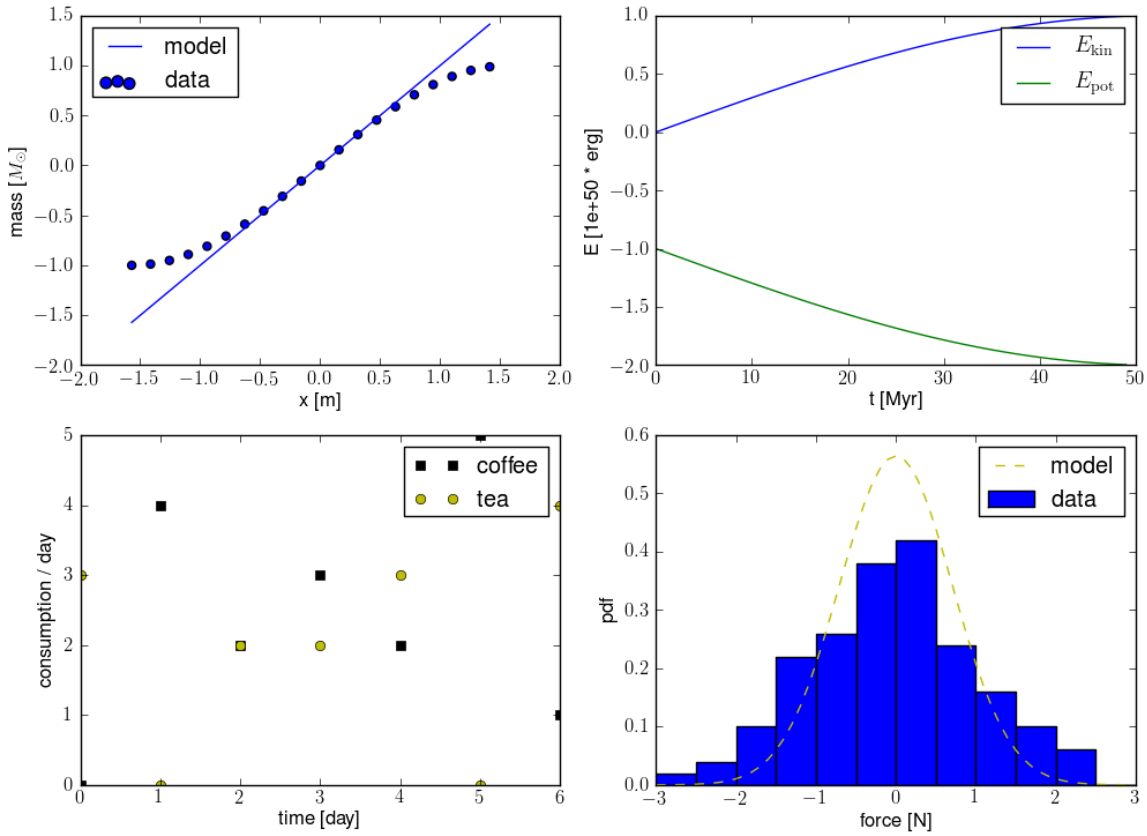
Mathtext

Latex support, while flexible and rich in features, might be slow and requires latex, dvipng and Ghostscript to be installed. You can use matplotlib's builtin TeX expression parser, mathtext instead. This is a subset TeX markup usable in any matplotlib text string by placing it inside a pair of dollar signs ($). See mathtext for details.

Supported functions

- plot
- semilogx
- semilogy
- loglog
- scatter
- hist
- xlabel
- ylabel

Example code



12 Setting values for grid boundaries

Note: At the time of writing of this tutorial (september 2012), only the Athena and Capreole hydrodynamics codes support the user defined boundaries.

In AMUSE, you can specify how a hydrodynamics grid code handles the boundary conditions. Hydrodynamic grid codes define a number of special grid cells at all ends of the grid. These cells are often called boundary or ghost cells and are handled differently by the code. The flow-variables in these cells are not evolved but calculated from other cells or from a user defined function.

12.1 Supported boundary conditions

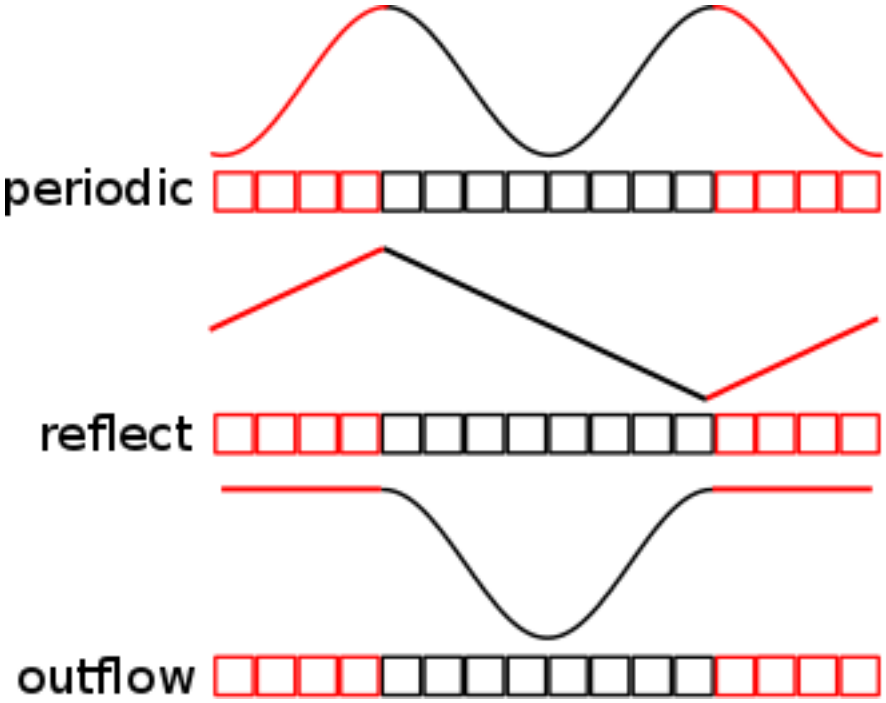
Each code in amuse will need to provide at least 3 kinds of boundary conditions:

periodic To simulate periodically repeating problems. Periodic boundary conditions simulates as if the grid is connected to a copy of itself, often implemented by copying over the data at the opposite site of the grid to the boundary cells.

reflect Mirrors all grid variables in the boundary cells, often implemented by copying the data of the cells connected to the boundary into the boundary cells.

outflow Sets the derivative of the flow variables with respect to the direction of the boundary to zero.

The following picture shows all three boundary conditions for a one dimensional grid with 4 boundary cells:



Boundary conditions are defined by setting the a boundary parameter for each boundary of the grid of a code. These boundaries are defined:

xbound1 The x-axis boundary on the left side of the grid, often called the inflow boundary

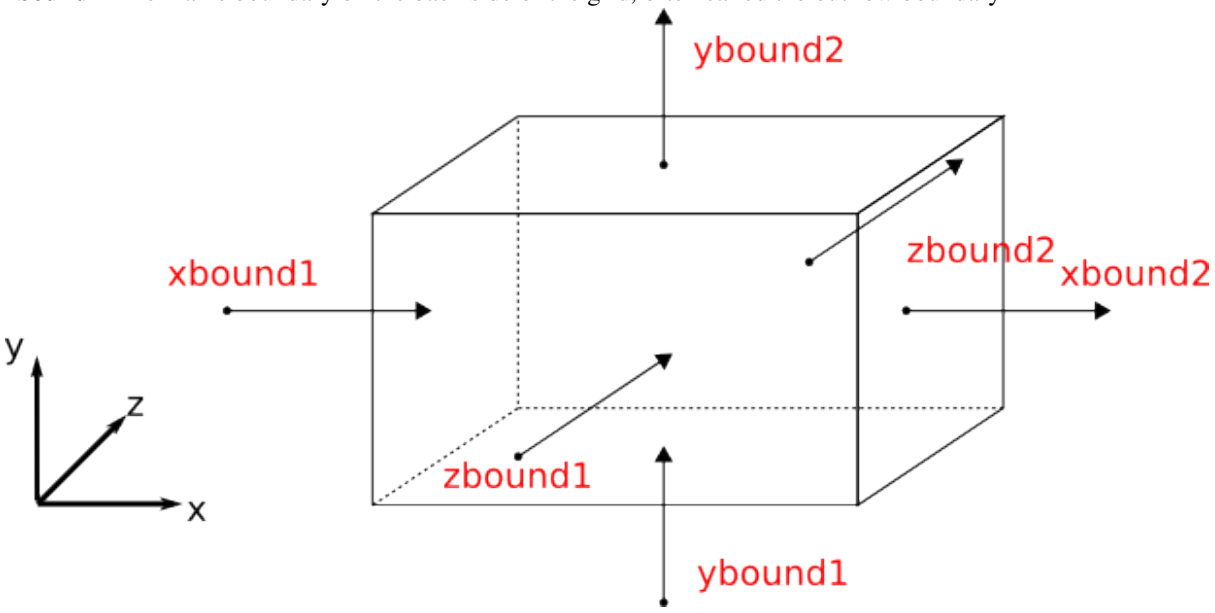
xbound2 The x-axis boundary on the right side of the grid, often called the outflow boundary

ybound1 The y-axis boundary on the bottom side of the grid, often called the inflow boundary

ybound2 The y-axis boundary on the top side of the grid, often called the outflow boundary

zbound1 The z-axis boundary on the front side of the grid, often called the inflow boundary

zbound2 The z-axis boundary on the back side of the grid, often called the outflow boundary




```
from amuse.lab import *

code = Athena()
```

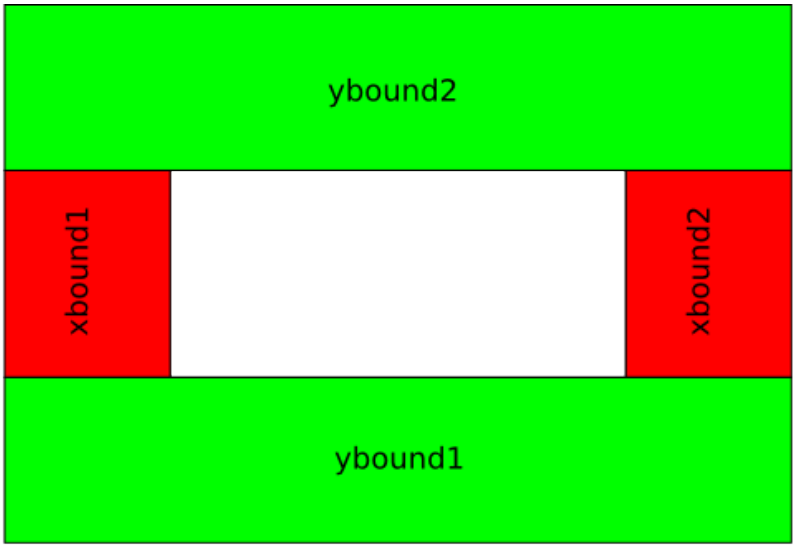
```
code.parameters.xbound1 = "periodic"
code.parameters.ybound1 = "reflect"
```

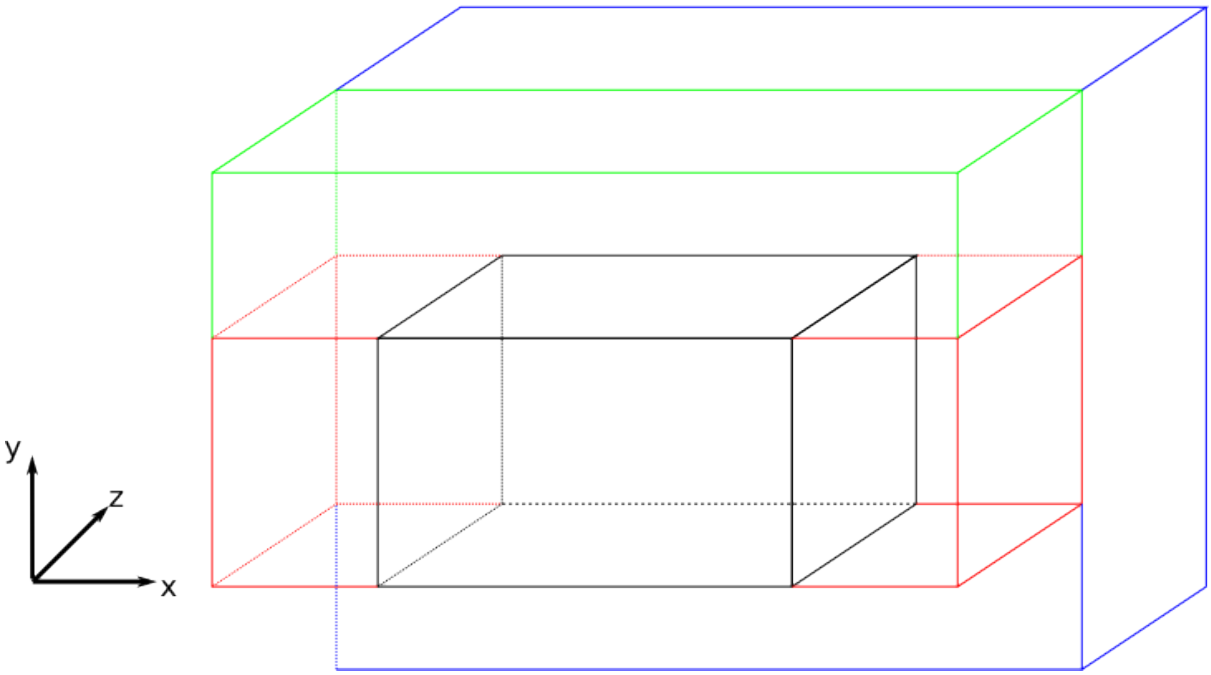
You can also set the boundary per axis as an `(inflow, outflow)` parameter with the `x_`, `y_`, `z_boundary_condition` parameters:

```
from amuse.lab import *

code = Athena()

instance.parameters.x_boundary_conditions = ("periodic","periodic")
instance.parameters.y_boundary_conditions = ("periodic","periodic")
instance.parameters.z_boundary_conditions = ("periodic","periodic")
```

12.2 Custom boundary conditions

You can specify a custom boundary condition by setting the boundary condition parameters to `interface`. After these parameters have been set and commited you can change the boundary values by filling in the correct boundary grid. These boundary grids do not overlap but patch together to form a larger cuboid around the grid cuboid. The x-boundary conditions have N boundary cells in the x direction and the same amount of cells in the y and z direction as the grid. The y-boundary conditions have N boundary cells in the y direction and in the x direction it has 2 * N + the number of cells in the x direction of the grid, in the z direction it has the same amount of cells as the grid. Finally the z-boundary conditions have N boundary cells in the z direction and in the x and y directions it has 2 * N + the number of cells in the x or y direction of the grid.





You can get the boundary grid of a specific boundary by calling *get_boundary_grid* on the code.

Note: You fill the returned grid by copying over data from a memory grid, as you can only set all attributes of the grid in one go (you cannot set the individual attributes yet at we did not implement the required methods, september 2012)

En example of filling the boundary grid:

```
from amuse.lab import *

instance=Athena()
instance.parameters.mesh_size = (10 , 20, 10)
instance.parameters.mesh_length = [1.0, 1.0, 1.0] | generic_unit_system.length
instance.parameters.x_boundary_conditions = ("interface", "outflow")

# request the boundary grid
# it will have shape of 4 x 20 x 10 (x, y, z)
# as Athena has a 4 cell boundary depth
xbound1_grid = instance.get_boundary_grid('xbound1')
```

```
# copy the grid to memory, so we can manipulate it easier
memxbound1 = xbound1_grid.copy()

# just set all cells in the grid to the same values
memxbound1.rho = 0.02 | density
memxbound1.rhovx = 0.2 | momentum
memxbound1.rhovy = 0.0 | momentum
memxbound1.rhovz = 0.0 | momentum
memxbound1.energy =  p / (instance.parameters.gamma - 1)
memxbound1.energy += 0.5 * (
        memxbound1.rhovx ** 2 +
        memxbound1.rhovy ** 2 +
        memxbound1.rhovz ** 2
    ) / memxbound1.rho

# copy over the data so that the code has the correct
# boundary values
channel = memxbound1.new_channel_to(xbound1_grid)
channel.copy()
```

A boundary grid connects to the grid in a very specific way. The general rule for the *1* or *inflow* boundaries is that the last cell of the boundary will connect to the first cell of the grid. The general rule for the *2* or *outflow* boundaries is that the first cell of the boundary will connect to the last cell of the grid.

For example for xbound1:

xbound1 The left, inflow boundary on the x-axis. Will have cells `[0, # boundary cells - 1]` in the x direction. The last cell of the boundary will connect to the first cell of the grid.

ybound1. The right, outflow boundary on the x-axis. Will have cells `[0, # boundary cells - 1]` in the x direction. The first cell of the boundary will connect to the last cell of the grid.

A boundary grid will have correct x, y and z positions that match the grid. So, if a 1D grid starts at 0.0 and the cell size is 1.0, the last grid cell on the left(the inflow boundary) will have position -0.5.

```
from amuse.lab import *

instance=Athena()
instance.parameters.mesh_size = (10 , 1, 1)
instance.parameters.mesh_length = [1.0, 1.0, 1.0] | generic_unit_system.length
instance.parameters.x_boundary_conditions = ("interface", "interface")

# request the boundary grid on both sides
# it will have shape of 4 x 1 x 1 (x, y, z)
# as Athena has a 4 cell boundary depth
xbound1_grid = instance.get_boundary_grid('xbound1')
xbound2_grid = instance.get_boundary_grid('xbound2')

print xbound1_grid[...,0,0].x
print instance.grid[...,0,0].x
print xbound2_grid[...,0,0].x
```

Python Module Index

a

Index

A