

Objective Molecular Dynamics User Guide

Yudi Rosandi*
Fachbereich Physik – TU Kaiserslautern

2005

*rosandi@physik.uni-kl.de

1 Installing guide

Do the following steps to install the Objective-MD (omd) library in your computer.

1. Extract the source file 'omd.zip':

```
$ unzip omd.zip
```

the source file will be stored in omd directory below the current directory.

2. Change directory to omd directory, and edit the file 'make.cfg'.

This file defines the installation directory, where the library and header files will be stored. By default omd will put the library at `omdlib` directory below the `$HOME` directory. To make it available globally you can change the setting to `/usr/include` and `/usr/lib` for include directory and library directory, respectively. But of course you need root permission to do this. Root permission is not needed when installing at your own directory.

3. Run:

```
$ make install
```

to install this library in the directory defined above.

4. Use:

```
$ make examples
```

to compile the example programs.

2 Overview

This Objective Molecular Dynamics program can be considered as the class library to create molecular dynamics simulation. It contains almost complete algorithms, stored in classes, to do general task in MD, such as the integrator, detector and some other algorithms. The program is written in C++.

The aim of this program is, to make it easier in performing simulation, without dealing with some unnecessary programming details. The program was designed as general as possible, to give the flexibility to users (you) in designing an MD simulation project. This is made possible by using object oriented programming. Nevertheless, it is not forbidden to re-implement the classes to fit to user need, using inheritance capability of C++ classes, without losing the functionality which is already available. In this case of course user will get into programming details, and must obey some regulations in order keep the mechanism work in the same scheme. The whole library consists of three main base classes:

- **AtomContainer** class, to store the atom data.
- **MDGadget** class, plug-able class to perform various operations.
- **SimSystem** class, which is actually a descendant of **AtomContainer**, as the main class container that performs the main program loop.

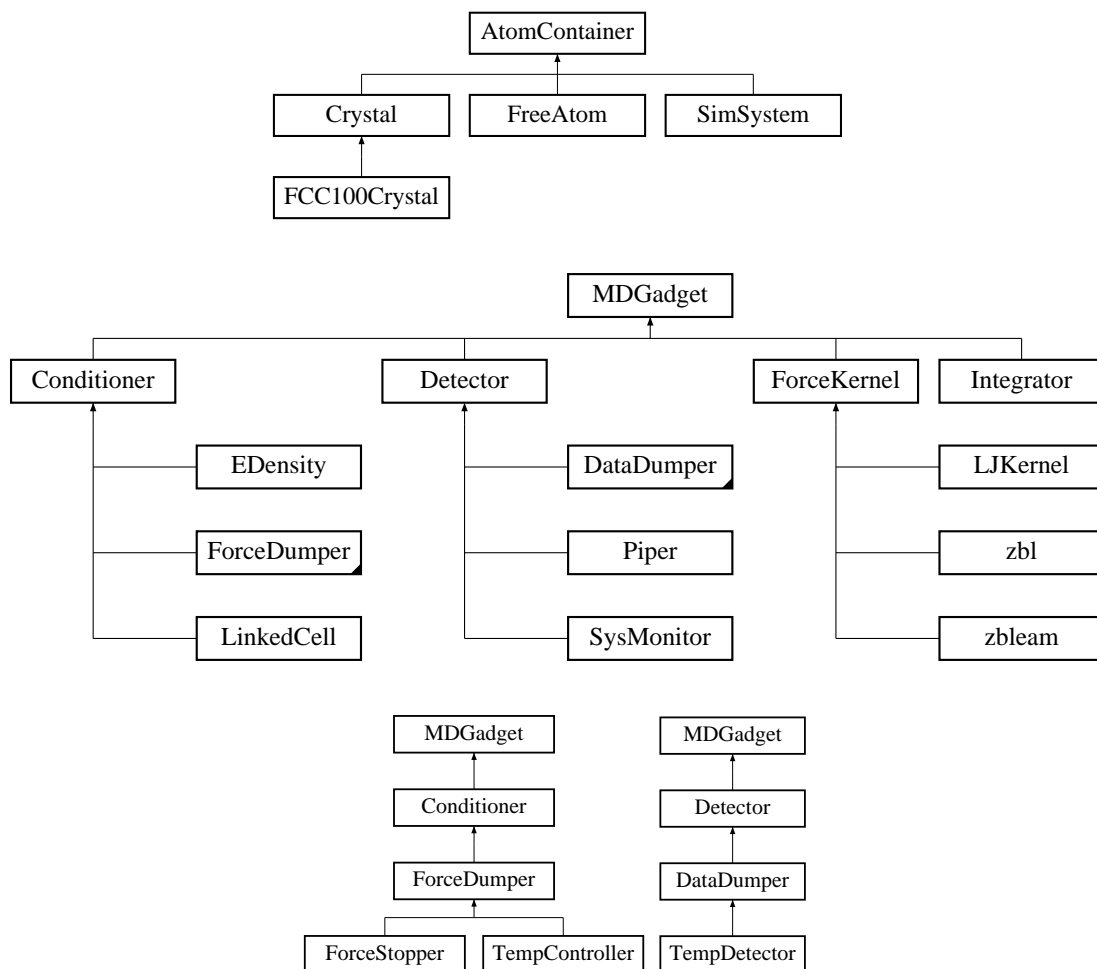


Figure 1: The class hierarchy of Objective-MD class library

One additional class **ConfigReader** is also provided to read the data from configuration file. This class is a free class which has no direct connection to the molecular dynamics algorithm. Figure 1 shows the class hierarchy of the main classes. To get more information about the capabilities and function of these classes, please see the reference manual. The scheme of doing programming with Objective-MD is that user can easily plug the objects to the program main class. The main class inherits **SimSystem** class. All other classes both the containers and the gadgets, can be attached or disposed easily to the MD program.

3 Making a simple MD application

To avoid too much details, let's stop talking about what and who are those classes, and be ready to have fun and take a small journey with C++ using this library. First of all I will tell you about how to do the compilation.

After installing the library you will have the library file (libomd.a) in your chosen path, and the header files in the 'OMD' directory right below that path. In following

explanation, I will assume that you use the default directory, which is `$HOME/omdlib` and `$HOME/omdlib/OMD`, respectively. Remember that `$HOME`, stands for your local home directory. Using this settings, you must do the compilation in the following way:

```
$ g++ mymd.cpp -o mymd -I$HOME/omdlib -L$HOME/omdlib -lomd -lm
```

It is not so complicated, isn't it? For more complex project, it would be better to use 'make' utility to compile the program. To do this you only have to make a small 'Makefile', as following:

```
-----cut-----
CXXFLAGS = -I$(HOME)/omdlib -L$(HOME)/omdlib -lomd -lm
MYPRG = mymd

all:
    $(CXX) $(MYPRG).cpp -o $(MYPRG) $(CXXFLAGS)
-----cut-----
```

With this Makefile in the same directory with the program, the compilation can be easily performed by a simple command:

```
$ make
```

3.1 Our first MD program

As you know now how to compile, lets make a simple program of Lennard-Jones system. Get ready with your favorite editor, and write the code in figure 2. Compile it, then run it after the compilation. You may also open the existing example program, available below the `example` directory, in the source directory.

This program will run the simulation of three particles, interacting each other by Lennard-Jones potential. We use two header files from our library, `simssystem.h` and `ljkernel.h` at line 5 and 6. Basically `simssystem.h` will also include another header file, but since `ljkernel.h` is the lowest implementation of the library (**ForceKernel**), it must be included explicitly here.

First, let us take a closer look at the main class declaration. We must have one main class which inherits **SimSystem** class (public inheritance), and at least implement two of its virtual functions **CreateSystem()** and **InitSystem()**, as at line 12 and 23. **CreateSystem()** is the function which prepares and creates the simulation system. It has a duty to add atom containers and some gadgets. **InitSystem()** is the function which has duty to do the system initialization. Here you must put all the initialization commands.

Creating the system

The system creation takes place from line 12 to 21. First we add three atoms with **AddAtom()** function. **AddAtom()** accepts the pointer to an **AtomContainer**

```

-----cut-----

1 // file name: step1.cpp
2
3 #include <iostream>
4 #include <unistd.h>
5 #include <OMD/simsystem.h>
6 #include <OMD/ljkernel.h>
7
8 using std::cout;
9
10 class MySystem: public SimSystem {
11
12     void CreateSystem()
13     {
14         AddAtom(new FreeAtom( -0.7, 0., 0., 0., 0., 0., UNIT_MASS, 1));
15         AddAtom(new FreeAtom( 0.7, 0., 0., 0., 0., 0., UNIT_MASS, 1));
16         AddAtom(new FreeAtom( 0., 1.04, 0., 0., 0., 0., UNIT_MASS, 1));
17
18         SetForce(new Integrator)
19             ->AddForce(new LJKernel);
20         AddDetector(new SysMonitor("Output.md"));
21     }
22
23     void InitSystem()
24     {
25         Box.x0=-5.0;
26         Box.y0=-5.0;
27         Box.z0=-5.0;
28         Box.x1= 5.0;
29         Box.y1= 5.0;
30         Box.z1= 5.0;
31         TimeStep=0.0064;
32         MaxTime=5000.*TimeStep;
33         PBoundary=NONPERIODIC;
34     }
35 };
36
37 MD_MAIN_BEGIN
38     MySystem MSys;
39     MSys.DumpAtoms("Data.000");
40     MSys.Measure();
41
42     cout << "Crystal is created\n"
43         << "Number of atoms is: " << MSys.NAtom << '\n'
44         << "Boundary:\n"
45         << " x(" << MSys.Box.x0 << ", " << MSys.Box.x1 << ")\n"
46         << " y(" << MSys.Box.y0 << ", " << MSys.Box.y1 << ")\n"
47         << " z(" << MSys.Box.z0 << ", " << MSys.Box.z1 << ")\n"
48         << "Initialization finished...\n"
49         << "Initial potential energy : " << MSys.Potential << "\n"
50         << "          Force cut radius : "
51         << MSys.Forces->MaxCutRadius << "\n";
52
53     cout.flush();
54     system("sleep 2");
55     MSys.Run();
56 MD_MAIN_END
-----cut-----

```

Figure 2: First application in Objective-MD

class as its argument. The `new` command in C++, creates the instance of a class and calls automatically the constructor upon creation process. We use now the atom container `FreeAtom` to add free atoms to the system. Following is the syntax of the constructor of `FreeAtom` class:

```
FreeAtom(double x, double y, double z,
         double vx, double vy, double vz,
         double AMass, int za)
```

The arguments are the position, initial velocity, mass, and atomic number, respectively. Because in Lennard-Jones system we usually use reduced units, then here we set mass to `UNIT_MASS`. The atomic number is arbitrary, because it is not used in the calculation. One is not a bad choice at all in this case.

After adding some atoms to our system, now we must set the interaction between them. To do this we use `SetForce()` function which actually sets the integrator to system as `Integrator` class. This function `SetForce()` returns the pointer to the integrator, so we can directly access this integrator, right after its creation, as shown in line 18. We use this capability to directly add the `ForceKernel` to it, as at line 19.

The integrator must have force kernel that do the actual force calculation. We have one nice descendant of `ForceKernel` class in our class hierarchy, that calculates Lennard-Jones interaction, the `LJKernel` class. So, we use it now. `LJKernel` needs no parameter to the constructor. It can be created in the way shown at line 19. Actually it has one default parameter to set cut radius to the force. But for simplicity, here we use the default value, 2.5. Please keep in mind that adding force integrator must take place *before adding another gadget* class, i.e. before detectors and conditioners.

The final system creation stage is adding the detector, so we can see what is going on in our simulation system while it is running. We add the `SysMonitor` detector, which dumps the information of our system, both to the screen and file. The information is written in the following format,

```
"step(%d) Time(%0.5f) E_k(%0.51f) E_p(%0.51f) E(%0.51f)"
```

the simulation step, simulation time, kinetic energy, potential energy, and total energy, respectively, in screen printing, and

```
"%d %0.10f %0.101f %0.101f %0.101f"
```

in the same data sequence for file printing.

System initialization

After finished with simulation system creation, lets take a look at the system initialization, from line 23 to 34. The initialization code is straight forward. First it sets the boundary box (line 25 to 30), and then sets the time step and maximum simulation time. The last thing is setting the boundary condition of the system. The available predefined constants for this are:

- PERIODIC_X
- PERIODIC_Y
- PERIODIC_Z
- PERIODIC
- NONPERIODIC

The main program

Now, we are ready with our main class. We call this main class as **MySystem**. The only thing left is the main program. Yes, you don't see the usual C/C++ main program here. I have made the macros to simplify the work, `MD_MAIN_BEGIN` and `MD_MAIN_END`. And yes, you can make your normal `main()` program here, but you must take care, because the system must run inside the `try` and `catch` block. So please just accept these two cute macros, at least for now.

First, make the instance for our main class, and call it **MSys** (line 38). Actually, you only need to call the **Run()** function from the object (line 55), but before, let us print some information about our system. The **DumpAtoms()** function dumps the atom data directly to file whose name defined as its argument. Then the **Measure()** function measures the system quantities, such as potential and system total energy. After this, you can print some informations, as shown from line 42 to 51.

Now, we are ready to go. **Run()** function will do the rest, and execute the simulation main loop until the elapsed simulation time reaches maximum simulation time. After running the program, please evaluate the output file, 'Output.md', using your favorite graphics plotting program.

3.2 Using MD-Gadgets

Previously I have introduced the Objective-MD gadgets, an integrator (**Integrator**), a force kernel (**LJKernel**), and a detector (**SysMonitor**) to you. Up to this point, you have not met one other type of gadget, the conditioner. But we will deal with it later.

Now I want to show you how to use another detector gadget. Isn't it boring to have only numbers in our screen? Well, lets do it in a better way. In our **MDGadget** class hierarchy, we have a **Piper**, which can communicate with an external program through a pipe¹. With this class we can run external program, and control it from inside our program while running.

Let us make a pipe connection with gnuplot! First, put this following code chunk right after adding the **SysMonitor** gadget, in the **CreateSystem()** function of our previous program,

¹I think, this class is the only class which totally incompatible with Windows environment. It only works fine under Linux. Anybody knows how to do piping under Windows?

```
-----cut-----
AddDetector(new Piper("gnuplot", 0.064, true))
    -> AddParameter("clear\nset xrange [-5:5]\nset yrange [-5:5]")
    -> AddParameter("plot 'Piper.out' u 1:2 w p pt 6 ps 3 notitle")
    -> AddParameter("quit");
-----cut-----
```

The **Piper** class has a constructor with three parameters, as following²,

```
Piper(const char *prg=NULL, double tm=0.0, bool dump=true)
```

Yes, all the argument has the default value. When using all of these defaults, then you will take responsible to connect the pipe by your self. You can use **Connect()** member function, though. The first parameter is the command string or the name of the external program, complete with all needed parameters. The second is the sample period, that defines how often the **Measure()** function is called by the system, and the last one is the switch to enable or disable dumping atom data to file when sample signal received. By default this class will print all the atom data to a file 'Piper.out', by calling **DumpAtoms()** function of **SimSystem** class (i.e. the main class). For current project we make a connection with 'gnuplot', the sample period is 0.064, and the data dumping is enabled.

The same way with **SetForce()**, **AddDetector()** function also returns the pointer to the added detector. Then we can use this to add more parameters to the piper. The **Piper** class may have three string parameters. First parameter will be sent through the pipe in the initialization stage. The second is the execution command, that will be sent in every sample time, and the last is the finalization command, sent right before the closing of pipe connection. Using **AddParameter()** function of **MDGadget**, we add these three commands. The initialization command is to clear the gnuplot screen and setting ranges. The execution command is to plot the data in 'Piper.out' file, and the finalization command is to close the gnuplot screen. By the way, the pipe will be closed automatically when our simulation ends, because gnuplot accept EOT³ character as a quit message. But anyway, it is a good practice to give 'quit' command here. Now, again, compile the program and run it. Enjoy the movie in gnuplot screen!

3.3 The conditioner

Now, it is the time for me to introduce the **Conditioner** class to you. As you can guess from the name, this branch of class has a duty to modify, change, or even manipulate the atom data or its environment, in the simulation system. I will categorize the conditioners into four category:

²Please see the reference manual for the detail about this class.

³stands for 'end of transmission'

1. The pre-integration conditioners, implement the **PreIntegration()** function, which is called by the main class before the integration takes place.
2. The pre-calculation conditioners, implement the **PreCalculation()** function, which is called by the integrator right before performing the force calculation loop.
3. The force modifier conditioners, implement the **ForceModifier()** function, which is called by integrator right after the force calculation loop.
4. The post-integration conditioners, implement the **PostIntegrator()** function, which is called by the main class before the detection takes place.

Of course one can create hybrid conditioners by implementing more than one functions above.

Now let us take one of these conditioners from the branch. Let us use the **Quencher** class to quench the atoms in our system. Following is the constructor,

```
Quencher(double QFactor=0.98, int per=0)
```

It takes two arguments, the quenching factor and the calling interval that defines in every how many steps the Quencher will execute it's function. This class is of the post-integration type, so it will be executed before the detection. Now, use the default value of arguments, and add this class anywhere in the system creation routine after setting the force⁴, by using **AddConditioner()** function, as follows,

```
-----cut-----  
  
AddConditioner(new Quencher);  
  
-----cut-----
```

Compile and run again your code, and see what happen. I believe you can predict it before running the program. Now play along with the atom positions, or you may add another. By luck, you will see the nice figures of equilibrium position of atoms in your simulation screen. Enjoy again, and when you find a bug, please send me message.

4 Re-implementing gadgets

Well, now we will learn how to implement a gadget in our library. The gadgets which are common to re-implement are force kernel, conditioner, and the detector. I think you will not need to re-implement the integrator, unless you want to apply a very different MD application. Lets assume that the **Integrator** class is the blue print in our class library tree. Now I will start to tell you how to create a new detector.

⁴I would prefer to add these gadgets in the sequence: force integrator, conditioner, and then detectors. But the main thing is to put the force integrator setting at the first position.

```

-----cut-----
1 class LJKernel: public ForceKernel {
2     double rsq, rp2, rp6, rp12, ff;
3     double dx,  dy,  dz, RCSQ;
4
5     public:
6     LJKernel(double R_CUT=2.5)
7     {
8         CutRadius=R_CUT;
9         RCSQ=CutRadius*CutRadius;
10    }
11
12    void Compute(int at, int to)
13    {
14        rsq = CalcSqrDistance(at, to, dx, dy, dz);
15        if (rsq < RCSQ) {
16            rp2=1.0/rsq;
17            rp6=rp2*rp2*rp2;
18            rp12=rp6*rp6;
19            potential = rp12-rp6;
20            virial = rp12+rp12-rp6;
21            ff = rp2*virial;
22            RETURN_FORCE(dx*ff, dy*ff, dz*ff);
23        }
24    }
25
26    void Correction() {
27        MDSystem->Potential *= 4.0;
28        MDSystem->Kinetic *= 24.0;
29    }
30 };
-----cut-----

```

Figure 3: The implementation of Lennard-Jones force kernel

4.1 Creating force kernel

The force kernel is a small⁵ gadget, that is very important to do the calculation. I call it kernel because without it our program will be very dumb. Here I will not rewrite another kernel, but will show you what is inside our **LJKernel** class. Please take a look at figure 3. I think now you believe me that it is small.

First, we must inherit the **ForceKernel** (line 1), again, with public⁶ inheritance. We can put all of the needed variable inside the class. Don't put it outside, unless you really know what you are doing! Here we have private⁷ variables as can be seen at line 2 and 3. These variables are private, that means they are available only for this class.

Line 6 to 10 is the constructor body. C++ has a very nice behavior, that it permits you to call parent's constructor directly in the child constructor declaration. The parent (**ForceKernel**), has two arguments with default values, as follows,

```
ForceKernel(int vfrom=0, int vto=0)
```

⁵Well, yes some time it is not so small, but I need to motivate you though!

⁶Read C++ book to understand this public inheritance. But in most cases we only need public inheritance

⁷By default a variable declared inside the class is private

These parameters define between which type of atoms this force will be used. These are the id numbers of **AtomContainer** class. But now we will only create force kernel that only work between the same type of atoms. So, don't mention it for now, and let's just use the defaults. We only set two variable in the constructor, as you can see at line 8 and 9, the cut radius of the force, and the square of it. Don't forget, here we also give the default value for the cut radius in its argument.

From line 12 to 24 lies the main algorithm of force calculation in our kernel. Please forget about the force calculation loop, which you usually see in every MD program code. Our **Integrator** class will take care of it for you. Just concentrate to the simplest way of calculating force between two particles. You must point out to line 22, where our kernel give the result to the integrator. Actually, we have to put the calculation result in two structure data, **dF0** and **dF1**, which stand for the force acting on the atom pointing by argument 'at' and 'to', respectively. These structure have three elements, x, y, and z. One must set here $dF1 = -dF0$. But don't worry, the macro **RETURN_FORCE()** will do everything for you. All you need to do is putting the elements (dF_x, dF_y, dF_z) as the macro arguments. You also must concern that the integrator uses basic form of verlet algorithm, i.e.,

$$x_{t+1} = v_t \delta t + \frac{1}{2} \frac{f_t}{m} \delta t^2$$

$$v_{t+1} = v_t + \frac{\delta t}{2} \left(\frac{f_t}{m} + \frac{f_{t+1}}{m} \right)$$

According to this, you must return the *real* force and velocity to the integrator⁸.

The last thing left is the correction. By implementing **Correction()** function, we can re-manipulate the data after the whole force loop in the system is done. This function will be called by the integrator after it calculates all of the forces and kinetic energy of system. In our case, instead of multiplying the potential and the kinetic energy with the same number in every calculation loop, it would be nicer if we do that after. Then we write the small corrections as seen at line 27 and 28. Here we multiply the system potential energy by four, and the kinetic energy by 24, which is common in every Lennard-Jones system algorithms⁹.

Now everything is ready, and we can use the object, which is actually did in our first MD program. In some force calculations, we need collaboration between force kernel and the conditioners. The **zbleam** class is the good example to this case, but for other pair potentials, the implementation is almost the same.

4.2 Creating detector

To create a new detector, one must inherit the **Detector** class. This class is an *abstract* class, which means it cannot be used unless you implement all of it's abstract functions. **Detector** has one abstract function, the **Measure()** function. Implementing the function is mandatory.

⁸In some MD code, people usually put $f\delta t/2$ in force, and $v\delta t$ in velocity variables. Now in our library we put force and velocity as is.

⁹All gadget classes can access the simulation main class as **MDSystem**, which is the pointer to it.

```

-----cut-----
1 class SysMonitor: public Detector {
2     ofstream fout;
3     bool     usescr;
4     bool     usefile;
5
6     public:
7
8     SysMonitor(const char* fn=NULL, bool _usescr=true)
9     {
10        if (fn!=NULL)
11            {fout.open(fn);usefile=true;}
12            else usefile=false;
13            usescr=_usescr;
14        }
15
16        ~SysMonitor(){fout.close();}
17
18        void Measure()
19        {
20            if (usescr) {
21                sprintf(st, SCRFMT,
22                    MDSystem->Step,
23                    MDSystem->ElapsedTime,
24                    MDSystem->Kinetic,
25                    MDSystem->Potential-MDSystem->BasePotential,
26                    MDSystem->Energy);
27                cout << st; cout.flush();
28            }
29
30            if (usefile) {
31                sprintf(st, FFMT,
32                    MDSystem->Step,
33                    MDSystem->ElapsedTime,
34                    MDSystem->Kinetic,
35                    MDSystem->Potential-MDSystem->BasePotential,
36                    MDSystem->Energy);
37                fout << st; fout.flush();
38            }
39        }
40 };
-----cut-----

```

Figure 4: The implementation of a detector

`Measure()` function will be called in every sample period. So, user can write the *measuring* commands inside the function. The default sample period, defined in the constructor of **Detector** class, is zero. This means by default the system will call the function every time step. **Detector** class, as all other gadgets, has right to access system atom data via the **MDSystem** pointer. We can read this data and calculate some values from it according to our need. Figure 4 is the implementation of our previously used detector, the **SysMonitor** detector. It reads some values from the system, and print it both to the screen and file.

Now, let us see the class variables. Our system monitor has three variables, the output file stream, and two boolean variables that switch on and off the screen printing and file printing capability. Again, these variables are **private**.

The constructor of this class (line 8) is straight forward. It has two parameters, the first is the file name which must be provided when you want to print the data to file, and the second is the switch to enable or disable screen printing. The constructor

will open the output file stream (line 11) when the file name argument is provided. Otherwise it will print only to screen. The last thing is the **Measure()** function (line 18 to 39), which is also straight forward. It reads the data from the system, and print it. The printing formats are defined in **SCRFMT** and **FFMT**, for screen and file output format, respectively. One can redefine these macros, before including the **force.h** to the program¹⁰. Finally we must close the file connection. We do it in the destructor function, as shown at line 16.

Now you can implement also your own new force kernels, conditioners, and detectors. But please read the reference manual to see what this class library has provided for you.

5 Dealing with bigger simulation

Previously we only used individual atoms in our simulation system. Our number of atom was small. How about making a bigger simulation? Do you want to put your atoms one by one in the code? I don't think so! Now, Back to figure 1, if you see the **AtomContainer** library tree, you will find another atom container called the **Crystal**. OK, now you can read my mind, it is the class for creating crystal structures. **Crystal** class is the base class of all the crystal structures.

Crystal class it self is the complete class whose capability is to create the FCC(111) crystal structure¹¹. It also inherits the capability to import data from file from it ancestor, **AtomContainer**. We can import data from file with or without velocity data file¹². The constructor has declaration as follows,

```
Crystal(  
    int XMLayers=0,  
    int YMLayers=0,  
    int ZMLayers=0,  
    double mass=195.078,  
    int z=78,  
    double lc=3.924  
)
```

The arguments are the monolayers, mass, atomic number, and the lattice constant, respectively. The monolayers is set by default to zero. In this case, when you give no other values, then you must import the crystal from file. All other arguments have default value for *platinum*. One difference in using the class, compared to **FreeAtom**, is that we must create or import the crystal before using. It is done either by calling **Create()** function explicitly, without any parameters, or import the data from file. **Create()** function returns **AtomContainer** pointer, as in the original implementation of **AtomContainer** class. We can use the returned pointer to call another function after calling.

¹⁰See line 5 in fig. 2. in this code **force.h** is called implicitly via **simsystem.h**.

¹¹By default, I give the parameters for Pt(111) crystal.

¹²Well I am tired keep doing this, but again I suggest you to read the reference manual.

5.1 Plugging crystal to the simulation

Now lets do a rather big modification to our previous simulation program. Let us start from the original program in figure 2. The first step is to delete all the atom addition lines from the program, and replace it with the following,

```
-----cut-----
    AddAtom(new Crystal(12, 6, 9, UNIT_MASS, 1, 1.6))
        -> Create();
-----cut-----
```

I put 1.6 to the lattice constant argument, merely corresponds to the equilibrium distance for Lennard-Jones's dimmer as the nearest neighbor distance ($2^{1/6}$). The **SimSystem** class will find the boundary of the system atoms when creating them. But, you must take care that the boundary values are only the minimum and maximum coordinate values. According to this you must take responsible to re-adjust them in the *initialization* part. Now, replace the **Box** settings in the initialization code, like this,

```
-----cut-----
    Box.x0-=0.8;
    Box.y0-=0.8;
    Box.z0-=0.8;
    Box.x1+=0.8;
    Box.y1+=0.8;
    Box.z1+=0.8;
-----cut-----
```

The numbers are supposed to be the half lattice constant. Yes, it is not totally true for FCC(111) crystal, but I just want to avoid physical details here. You may provide the better numbers, though. What we do is making the system boundary box half lattice constant wider then before.

It is not a good idea to use pipe to 'gnuplot' this time, because our system atom number is too big. Now we can pick another detector class from the hierarchy tree, the **DataDumper**. This class has capability to dump the atom data to some files, with special extension name denoting the dumping sequence. We can interpret later this extension name as the sampling time of our data. Following is the constructor declaration of our new class:

```
    DataDumper(double      tm=0.0,
               const char* fn=NULL,
               int         extlen=3)
```

The argument `tm` is for the sample time, `fn` the file name, and `extlen` is the extension length of the files. When we give default values to second and third arguments, we will have the files with name format `'Data.XXX'`, where `'XXX'` is the sequence number. To plug this class to the simulation, you can do simply by adding to the initialization code,

```
-----cut-----
                AddDetector(new DataDumper(0.64));
-----cut-----
```

Now take a cup of coffee and run the program. Evaluate again the output (`Output.md`) with your plotter program. Since you have a lot of data files¹³, you can convert them also to a movie if you like.

5.2 The linked cell

It took about 45 minutes to run the last program in my computer. I hope you have a faster computer than mine. There are two nice algorithms in this world to help making MD calculation faster. The famous *linked cell* and *verlet list* algorithm. And fortunately we have these algorithms implemented in our **LinkedCell** class. It has both.

It is a pre-integration conditioner class, that maintains the verlet list for our simulation. The class is applicable both in periodic and non periodic boundary condition environment. The constructor have only one argument to define the update period of verlet's neighbor table. The following is the declaration of the constructor,

```
LinkedCell(int up=5)
```

I made the integrator in such a way that it knows by it self when there is a **LinkedCell** class attached to the system or not. So, feel free to attach and dispose this class to the system. If you decide to attach it, then add this following code line to your initialization code. Remember! Do this after setting the force.

```
-----cut-----
                AddConditioner(LinkedCell);
-----cut-----
```

By default, it will refresh the neighbor list every 5 steps. Now, the **LinkedCell** is on board, and ready to fly. Compile, run the program, and compare the time taken by simulation with the previous one. I believe you will not need a cup of coffee anymore this time.

5.3 Using the configuration reader

As I told you before, that we have one independent class, that can read the configuration items from a file. The class is **ConfReader**. The capabilities are to read the

¹³The data format inside these files is compatible with **impact** program from our group. You can use our **BuckyBall** program to view it and making movie.

tags and their values, store these tags and values in a list, give the values by calling the tag name to the program, and of course skipping comments. To use this class you must include `cfgreader.h` and create an instance of the class. Following is the declaration of the constructor,

```
ConfigReader(const char* cfg_fl);
```

The argument '`cfg_fl`' is the name of configuration file to read. In this configuration file contains tags and the corresponding values separated each other with equal (=) sign. The values can be integer, double, or boolean strings (`true` and `false`). We can read these tags anywhere in the program, using member functions, `GetDouble()`, `GetInt()`, `GetBool()`, `GetString()`¹⁴ and `GetVector()`. Except `GetVector()` the parameter, the argument of these functions is the tag name (`const char*`). For `GetVector()` we have two forms, as follows,

```
double GetVector(const char* tag, int elem)
double GetVector(const char* tag, double &x, double &y, double &z)
```

The first argument is always the tag name. The first form returns the value of the $elem^{th}$ element of the vector, while the second form gives the values directly to `x`, `y` and `z`, as pass by reference parameters¹⁵. The vector elements are always double precision number. When these functions fail to find the tag, it will throw an exception. By default the program will print the error message and then stops.

Let us make the configuration file for the previous code. Now, we will control some variables from out side of program, in order to avoid recompilation when we want to change them. The following is the content of configuration file,

```
-----cut-----
#####
# Objective-MD Configuration file
#
SimulationName = Objective-MD_Tutorial
FileName = Omd
Monolayers = (40, 20, 15)
LatticeConstant = 1.6
DumpTime = 0.64
Steps = 5000
WaitBeforeRun = true
OnlyCheck = false
-----cut-----
```

¹⁴The string value, and also tag, cannot accept space character. Use underscore if you want to add space.

¹⁵It also returns the square of the absolute value of the vector.

This sample configuration file represents all of the data type described above, double, integer, boolean, and also string. See program in figure 5 for the application. Actually that is the modification of program in figure 2.

Write the configuration in the file named 'omd.cfg'. Now you can see how it works. May be this is the last time I ask you to do compilation. After you compile the code, you don't have to recompile it again. Almost everything can be controlled from the configuration file. You can also, simply add more new tags to this configuration file, and access it immediately from the program, as in the example.

6 What next?

I have told you the basic of using this library. Actually that was almost everything. All you have to do next is to explore the class library tree, and try the available classes. You can also re-implement them for your special purpose. It is all up to you. I believe you will find something new there, or at least a bug.

This new born class library is not complete yet. It will be very helpful if someone can help in implementing new potential kernels. Until today we only have the *eam* and *ZBL* potential for platinum, although the *ZBL* potential is also applicable to another atoms. By the way, I hope this class library will be useful.

Yudi Rosandi

Kaiserslautern, 2005

```

-----cut-----

1 #include <iostream>
2 #include <unistd.h>
3 #include <OMD/simsystem.h>
4 #include <OMD/linkedcell.h>
5 #include <OMD/ljkernel.h>
6 #include <OMD/cfgreader.h>
7
8 using std::cout;
9
10 ConfigReader Cf("omd.cfg");
11
12 class MySystem: public SimSystem {
13     void CreateSystem() {
14         double xml, yml, zml;
15         Cf.GetVector("Monolayers", xml, yml, zml);
16         AddAtom(new Crystal((int)xml, (int)yml, (int)zml, UNIT_MASS, 1,
17             Cf.GetDouble("LatticeConstant"))->Create());
18         SetForce(new Integrator)->AddForce(new LJKernel);
19         AddConditioner(new LinkedCell);
20         AddDetector(new SysMonitor("Output.md"));
21         AddDetector(new DataDumper(Cf.GetDouble("DumpTime")));
22     }
23
24     void InitSystem()
25     {
26         Box.x0-=0.8; Box.y0-=0.8; Box.z0-=0.8;
27         Box.x1+=0.8; Box.y1+=0.8; Box.z1+=0.8;
28         TimeStep=0.0064;
29         MaxTime =Cf.GetInt("Steps")*TimeStep;
30         MaxPath =0.04;
31         PBoundary=NONPERIODIC;
32     }
33 };
34
35 MD_MAIN_BEGIN
36     MySystem MSys;
37     MSys.DumpAtoms("Data.Crystal");
38     MSys.Measure();
39
40     cout << "Simulation name: " << Cf.GetString("SimulationName") << "\n"
41         << "Number of atoms is: " << MSys.NAtom << '\n'
42         << "Boundary:\n"
43         << " x(" << MSys.Box.x0 << ", " << MSys.Box.x1 << ")\n"
44         << " y(" << MSys.Box.y0 << ", " << MSys.Box.y1 << ")\n"
45         << " z(" << MSys.Box.z0 << ", " << MSys.Box.z1 << ")\n"
46         << "Initialization finished...\n"
47         << "Initial potential energy : " << MSys.BasePotential << "\n"
48         << "          Force cut radius : "
49         << MSys.Forces->MaxCutRadius << "\n";
50
51     cout.flush();
52     if (Cf.GetBool("WaitBeforeRun")) system("sleep 2");
53     if (!Cf.GetBool("OnlyCheck")) MSys.Run();
54 MD_MAIN_END
-----cut-----

```

Figure 5: Second application in Objective-MD