A brief User Manual for EPSR++
V1.0: June 2019

Edited by
Changli Ma & He Cheng
at
Institute of High Energy Physics (IHEP)
Dongguan Neutron Science Center (CSNS)

June 2, 2019

Acronyms and Symbols

Acronyms

EPSR	Empirical Potential Structure Refinement
GPU	Graphics Processing Unit
PDF	Pair Correlation Function
RDF	Radial Distribution Function
RP	Reference Potential
EMP	Empirical Potential
NSP	Neutron Scattering Pattern
NSF	Neutron Structure Factor
MC	Monte-Carlo
MD	Molecular Dynamic
MPI	Message Passing Interface

Symbols

$S(Q)$	NSP of a sample measured in neutron scattering or diffraction experiment.
$\Delta S(Q)$	The difference $S(Q)$ between experiment and MC simulation.
$g_{\alpha\beta}(r)$	PDF or RDF of atom types α and β.
b_α	The neutron scattering length of isotope α.

1. Motivation

Empirical potential structure refinement (EPSR) [1] is a neutron scattering data analysis algorithm and a software package. It was developed by the British spallation neutron source (ISIS) Disordered Materials Group [2], aims to construct the most-probable all-atom structure of amorphous materials. In the past decades, it has been wildly used and generated reliable results for scientists. However, it is programmed in Fortran and implements shared memory architecture with OpenMP[3], so it can only run in parallel within a personal computer (PC) or a single computer cluster node. As a result, the EPSR is efficient at simulating systems with small molecules and contain smaller than 50 thousand atoms, but it is difficult to be used in a macromolecular or nano-particle system and it is very hard to define special molecules by users for their analysis. With the extensive construction of supercomputer clusters and the widespread use of GPU[4] acceleration technology, it is necessary to update the EPSR with distributed memory architecture and GPU acceleration supporting to improve its calculation speed. Accordingly, an open source framework EPSR++ is developed. It can be paralleled across nodes within a computer cluster and supports GPU acceleration. In addition, the framework is programmed in C++[5] object-oriented language, therefore users can define special simulation box, atoms, molecules and random motion patterns conveniently for their analysis.

[1] https://www.isis.stfc.ac.uk/Pages/Empirical-Potential-Structure-Refinement.aspx
[2] https://www.isis.stfc.ac.uk/Pages/Disordered-Materials.aspx
[3] https://www.openmp.org/
[4] https://en.wikipedia.org/wiki/Graphics_processing_unit
[5] http://www.cplusplus.com/

2. EPSR Principle and EPSR++ Algorithmic Flow

2.1 Brief Description of EPSR Principle

EPSR is essentially a Monte-Carlo energy minimization simulation method, but the difference from Metropolis Monte-Carlo is on the fact that the potential used in a simulation is based on experiment data rather than on theory. In EPSR, the atomic potential used in MC simulation is divided into two categories *i.e.* "reference potential (RP)" and "empirical potential (EMP)". The RP is similar to that used in MD simulations[6], so the potential form and parameters can be obtained from all-atom MD force field such as OPLS, AMBER, CHARMM and so on[7]. By contrast, EMP has no fixed form and is used to reflect the difference of neutron structure factor between experiments and MC simulations($\Delta S(Q)$). To be exact, EMP is the reverse Fourier transform of $\Delta S(Q)$. In the present form of EPSR, EMP is expressed to a list of Poisson distribution in real space and their corresponding Fourier transforms in Q space. In short, RP is used to keep molecules with reasonable shape, while EMP is used as a feedback parameter to lead the MC simulations in a consistently manner with diffraction experimental data. In the EPSR simulations, RP is used in MC simulation at the beginning, and the potential changes of the system(ΔU) is used as selection criteria of molecule or atom random movement. When the simulation reaches equilibrium, EMP is introduced to fit $\Delta S(Q)$ and is added to RP to continue the simulation. When the MC simulation with updated potential reaches equilibrium again, EPSR calculates EMP and updates simulation potential once more. This process is repeated until the EMP close to zero, *i.e.* $\Delta S(Q)$ is very little.

[6] https://en.wikipedia.org/wiki/Molecular_dynamics
[7] https://en.wikipedia.org/wiki/Force_field_(chemistry)

2.2 EPSR++ Algorithmic Flow

The schematic diagram of EPSR++ algorithmic flow is shown in Fig.2.1.

I. Define simulation box, molecules, atoms, select a parallel method, then recompile the EPSR++. Three basic C++ classes are defined in the framework for users to define special classes and objects:

• SimBox: It is used to generate a simulation box. It can be initialized with the molar mass of molecules, molecular numbers and density, etc. Users can define a spacial initial function for their experimental samples.

• Molecule: It is used to define molecules, intra-molecule potentials and their movement patterns, such as translation, rotation, etc. Users can define arbitrary molecule classes and try any special inter- and intra-molecule movements. This class makes the program very flexible and user friendly.

• Atom: It is used to define atoms. The basic properties of an atom such as atom name, element name, isotope name, coordinate position, neutron scattering length, etc are defined in this class. The coordinate position is defined using Hep3Vector class of CLHEP[8] library because of Hep3Vector has abundant functions to perform transition, rotation, distance and angle calculation, etc. Users need to initialize atom objects in molecule class objects rather than define new atom classes.

Additionally, new atom potential and potential function can be added by editing the C++ head and source file 'NDA_parameter_pot.h' and 'NDA_func_pot.h'

After classes for special simulation are defined, users need to edit the 'Makefile' to select which parallel method will be used to accelerate the program *i.e.*, '-D USE_MPI', '-D USE_OMP', '-D USE_GPU' compiler options mean employ mpich2 API to make the program parallel among different nodes of a computer cluster, use OpenMP API to make the program parallel among many cores of a CPU and enable GPU hardware acceleration respectively.

II. Edit Input File, Run Program: a text input file needs to be offered to the program to initialize the simulation box and execute parameters, such as the molar mass of molecules, molecular numbers, density, maximal movement step, MC equilibrium criteria, EPMC finish criteria, etc. Users can also modify these parameters' values directly in the C++ source file before recompile EPSR++ rather than edit an input file.

[8] http://proj-clhep.web.cern.ch/proj-clhep/

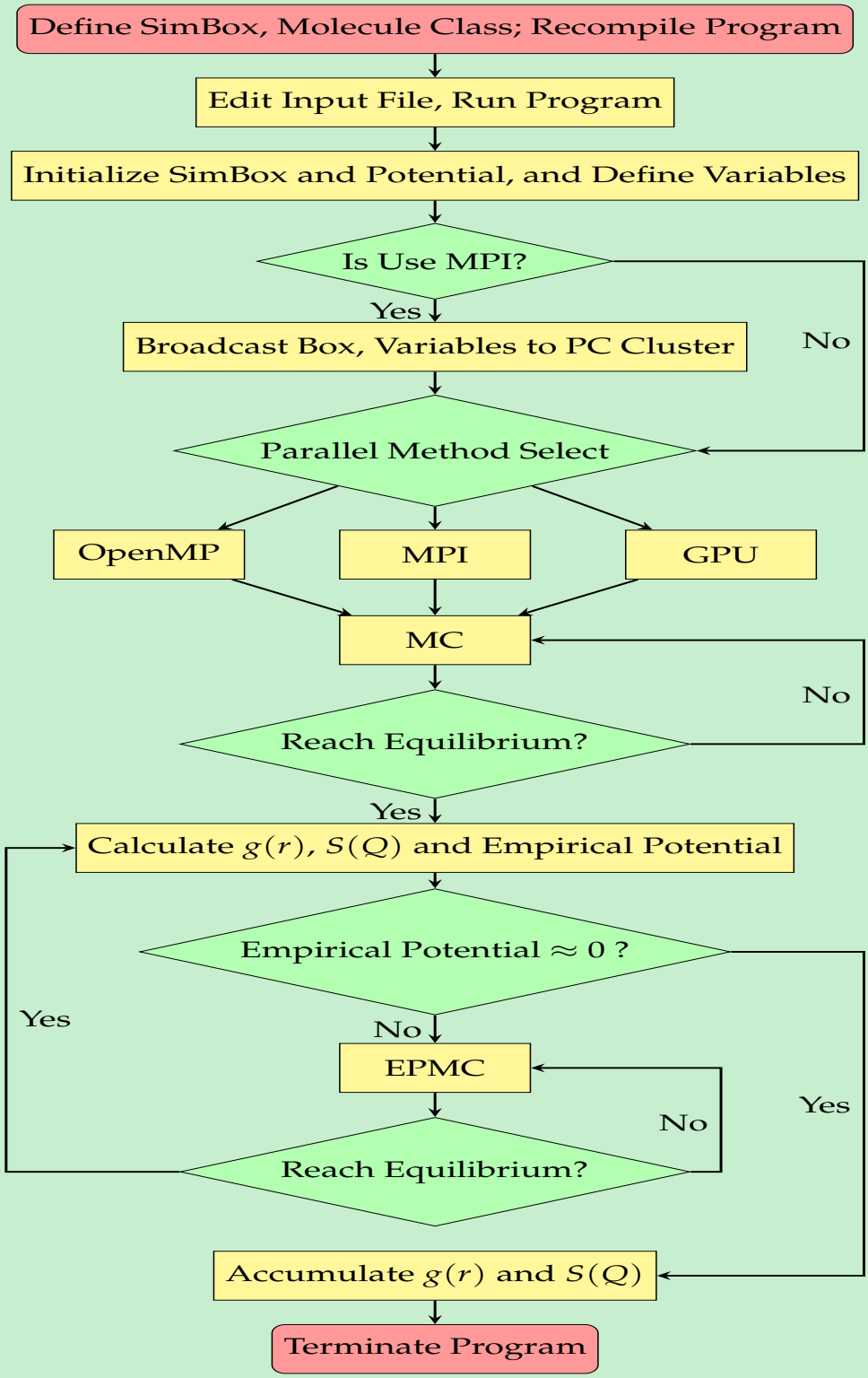

Fig. 2.1: Schematic of arithmetic flow of EPSR++.

III. Initialize SimBox and Potential, and Define Variables: Program initializes simulation box, reference potential arrays of all atom type pairs with the input parameters. The reference potential parameters are defined in a C++ head file 'NDA_parameter_pot.h', users can edit or define these parameters conveniently. Potential functions can also be redefined by users, including L–J potential, electric field, and potential truncation function.

    If the program needs to be paralleled among different nodes of a computer cluster, the initialized simulation box and variables will be broadcast to different nodes with 'MPI_Bcast()' function of mpich2 API.

IV. MC: The program performs MC simulation with reference potentials until reaches to equilibrium. The program moves molecules or atoms in sequence or randomly. The potential energy change of the simulation box ($\Delta U = U_{after} - U_{before}$) is used as movement accept criterion. If $\Delta U < 0$ the movement is accepted. If $\Delta U > 0$, the movement is accepted with a probability $e^{-\Delta U/kT}$.

    The compute resource consumption of $g(r)$ is in direct proportion to the second order of atom amount and it needs to be recalculated after every MC simulation step. We designed three selectable parallel methods, *i.e.* OpenMP, MPI, and and GPU, to accelerate the calculation. Users could select a suitable method according to their hardware and speed requirements.

V. Calculate $g(r)$, $S(Q)$ and EMP: When MC simulation reaches equilibrium, the program calculates the difference neutron structure factor $\Delta S(Q)$ between simulated $S_{sim}(Q)$ and experimental $S_{exp}(Q)$. EMP is calculated by performing Fourier transition to $\Delta S(Q)$. The program adds EMP and RP together as updated potential to perform the subsequent MC simulation, which is named as EPMC.

    EPMC is very similar with MC except that: when the simulation reaches a equilibrium with EMP and RP, the program calculates EMP again, and adds it to the previous potential, and performs the simulation with the updated potential. When EMP gets to a very low value, the program stops to update simulation potential.

VI. Accumulate $g(r)$ and $S(Q)$: When the simulated $S(Q)$ is very similar with that from experiment and EMP is near zero, program will still perform EPMC to accumulate simulation data to improve statistics until obtain smooth $g(r)$ and $S(Q)$ curves. Except $g(r)$ and $S(Q)$, the program outputs a coordinate file which contains all atoms's coordinates in text format as used in GROMACS (with .gro as suffix)[9] or as used in LAMMPS (with .xyz as suffix)[10], this file

[9] http://www.gromacs.org/
[10] https://lammps.sandia.gov/

can be used in GROMACS or LAMMPS to calculate enthalpy, entropy, etc, and also can be visualized with visual molecular dynamics (VMD)[11]. Users can define, calculate and output any interesting variables which are related directly with the atomic structure of the simulation systems by programming new algorithms and output functions.

3. EPSR++ Using Guides

3.1 System requirement

EPSR++ can be compiled and executed at servers, personal computers or computer clusters with Linux operation system. GCC, make[12], and CLHEP[13] have to be installed in the system. If users need the program parallel across different computer cluster nodes and accelerated with GPU hardware, mpicxx/mpic++ (mpich2 API)[14] and NVCC[15] should be installed respectively.

3.2 Units

The units of length, momentum transfer Q, potential, and electric quantity used in the program are angstrom (Å), reciprocal of angstrom (1/Å, or $Å^{-1}$), kJ/mol, and elementary charge respectively. The unit of neutron scattering lengths b_n is fermi (fm, 1fm= 10^{-15}m). Accordingly, the unit of neutron structure factor [S(Q)] is fm^2 ($10^{-30}m^2$). In general, the neutron scattering data is in the unit of barn ($10^{-28}m^2$). Therefore users need to multiply the raw neutron data by 100 before they are read by EPSR++.

[11] http://www.ks.uiuc.edu/Research/vmd/
[12] https://www3.ntu.edu.sg/home/ehchua/programming/cpp/gcc_make.html
[13] http://proj-clhep.web.cern.ch/proj-clhep/
[14] http://www.mpich.org/
[15] https://docs.nvidia.com/cuda/cuda-compiler-driver-nvcc/index.html

Table 3.1: The files and subfolders in EPSR++V1.0 directory.

Name	Property	Description
EPSR++.cu	file	C++ main program source file of EPSR++.
Makefile	file	Control makefile of the program.
README.txt	file	A brief description of the program.
make.sh	file	Shell script to call shell commands and GNU make to compile the program.
source	folder	Contains all C++ header and source files.
objects	folder	Contains all objects files, dependent files and a sub-makefile.
test_data	folder	Contains GROMACS MD simulation neutron data and ISIS experimental data of H_2O.
EPSR++_CPU	file	The executable program only support serial algorithm in a CPU core.
EPSR++_OMP	file	The executable program support parallel algorithm with OpenMP in a multi-core CPU.
EPSR++_MPI	file	The executable program support parallel algorithm with mpich2 in a multi-core CPU or across different nodes of a computer cluster.
EPSR++_GPU	file	The executable program support GPU acceleration.
run_isis.sh	file	Run shell script for ISIS H_2O samples.
run_simu.sh	file	Run shell script for GROMACS MD H_2O samples.

3.3 Software package folder's architecture

The top folder of the package is named EPSR++V1.0. The files and subfolders in it are listed in Tab. 3.1. Users can edit main C++ source file, Makefiles, etc, to make the program suit for their analysis.

3.4 Header and Source Files of EPSR++

In this section, the files that may need to be modified by users will be introduced, so users can modify them according this manual and standard C++ syntax for. The files in EPSR++V1.0 directory:

a. make.sh: A bash shell script. Because gcc and mpicxx/mpic++ cannot recognize a file with .cu extension as a C++ source file, EPSR++.cu needs to be copied to a file named EPSR++.cxx before compiling. Users need only to modify EPSR++.cu, then recompile the program by run this script by '> sh make.sh'.

   `in make.sh:`

```
cp  EPSR++.cu   EPSR++.cxx
make clean
make all
rm  EPSR++.cxx
```

b. Makefile: The general Makefile. Users can modify it for their analysis according to their computer system, speed requirement, and the detailed comments in the file.

c. Makefile in objects directory: This is the sub makefile for compiling object files.

d. EPSR++.cu: This is the C++ source file of the main program. Users can modify it according to their simulation. All codes which are likely need to be modified by users are commented and are started and finished with lines
`// maybe need to be modified by users VVVVV !!!` and
`// maybe need to be modified by users AAAAA !!!`
respectively as shown below:

```
// maybe need to be modified by users VVVVV !!!
// Program configuration
//=================================================
// Some basic settings
// True means use movement times, false means use accept
// ratio as update pot criteria
bool    is_use_step_to_update_pot = true;
bool    is_accumulate_data  = false;

// If    is_use_step_to_update_pot = true,  edit below:
int      num_step_start_update_pot = 5000;
int      num_step_update_pot = 5000;

// If    is_use_step_to_update_pot = false,  edit below:
double   accept_ratio_move_cut = 0.05;
accept_ratio_move_cut = 0.1;
// maybe need to be modified by users AAAAA !!!
```

The files in ‘source’ directory are C++ header and source files used in the program. The files with ‘_gpu.cu/h’ or ‘_omp.cxx/h’ suffix are the corresponding CUDA or OpenMP version respectively. We omit ‘.h’ and ‘.cxx’ suffixes of the files’ names in in this section.

a. atomtables: The neutron scattering lengths b_n list of all isotopic nuclides in nature. These files are the same as used in Debyer.[16]

b. NDA_atom: The files for atom class. This class is almost complete, so users don't need to modify it any more. The 'name_atom' and 'name_nuc' must be as same as used in atomtables.cxx.

c. NDA_molecule: The files for molecule class. Users can define their own molecule, molecule movement patterns etc. There are some important functions for reference to define new functions according to users' requirement:

   'Initialize_H2O()' is a initialize function that makes the molecule to be a water molecule. 'CalculateEnergyMol_H2O()' and 'CalculateEnergyMol_H2O_Temp()' is the functions that are used to calculate the intra-molecule energy of H_2O molecules. Users can refer to these functions to defined new molecules.

   'Translation()', 'Rotate()', 'TranslationSingleAtom()', 'RotateSingleAtom()' and their corresponding Attempt functions are used to perform whole molecule or single atom translation and rotation. User can define any similar movement patterns refer to these functions.

```
void  Initialize_H2O(bool is_set_coord);

void  Translation(CLHEP::Hep3Vector xyz_delta, double boxsize);
void  Rotate(CLHEP::Hep3Vector & axis, double delta,
             double boxsize);

void  TranslationAttempt(CLHEP::Hep3Vector xyz_delta,
                  double boxsize);
void  RotateAttempt(CLHEP::Hep3Vector & axis, double delta,
                    double boxsize);

void  ConfirmAtomMove(bool is_rotate);

void  TranslationSingleAtom(int index_atom, CLHEP::Hep3Vector
                            xyz_delta, double boxsize);
void  RotateSingleAtom(int index_atom, CLHEP::Hep3Vector & axis,
                       double delta, double boxsize);

void  TranslationAttemptSingleAtom(int index_atom,
                       CLHEP::Hep3Vector xyz_delta, double boxsize);
```

[16] https://debyer.readthedocs.io/en/latest/

```
void  RotateAttemptSingleAtom(int index_atom, CLHEP::Hep3Vector &
                                axis,  double delta, double boxsize);

void  ConfirmAtomMoveSingleAtom(int index_atom, bool is_rotate);

inline void  UpdateEnergyMol() { energy_mol = energy_mol_temp; }

void   CalculateEnergyMol();
void   CalculateEnergyMol_H2O();
void   CalculateEnergyMol_H2O_Temp(int index_atom, CLHEP::Hep3Vector
                                   delta_coord_atom);
```

d. NDA_func_array: Source files define all functions about C++ array's calculation.

e. NDA_func_file: Source files define functions for files' reading and writing.

f. NDA_func_matrix: Source files for matrix calculation.

g. NDA_func_nsf: Source files for neutron scattering pattern calculation.

h. NDA_func_pdf: Source files for pair correlation function calculation.

i. NDA_func_pot: Functions about simulation potential, including L–J potential, electric potential, repulsive potential, potential truncate functions, etc.

j. NDA_func_sim: Functions to treat simulation box in MC and EPSR simulation.

k. NDA_func_temp: Some useful C++template functions.

l. NDA_parameter_pot: Define the parameter of potential used in the simulation. The example of of H2O sample used are list here. The atom name here must is as same as used as 'name_atom_inepsr' of NDA_atom.

```
const unsigned num_para_pot_lj = 5;
std::map <string, double*>  para_pot_lj;

double O_H2O_lj[num_para_pot_lj] = {0.65, 3.166};
para_pot_lj.insert(make_pair("O_H2O", O_H2O_lj));

double H_H2O_lj[num_para_pot_lj] = {0.0, 0.0};
para_pot_lj.insert(make_pair("H_H2O", H_H2O_lj));
```

```
const unsigned num_para_pot_ep = 5;
std::map <string, double*>  para_pot_ep;

double O_H2O_ep[num_para_pot_ep] = {-0.8476};
para_pot_ep.insert(make_pair("O_H2O", O_H2O_ep));

double H_H2O_ep[num_para_pot_ep] = {0.4238};
para_pot_ep.insert(make_pair("H_H2O", H_H2O_ep));
```

m. NDA_symbox: Files of simulation box class. There are some basic functions for calculate simulation box. 'Initialize_H2O()' and 'Initialize_H2O_Sample()' are initialize function with H_2O molecules, users can define similar functions to define their own simulation box. 'PrintBoxToGro()' function is used to output the simulation box conformation to a '.GRO' file that can be recognized by GROMACS and VMD. Users can modify it or define new function to print the conformation in other format. Other functions in this class no need to be modified.

```
void Initialize_H2O(
            int     & num_mol_input,
            double     mol_wt_input,
            double    density_input,
            bool          set_coord,
            bool     grid_or_random,
            bool          use_shell,
            double          r_shell
            );

void Initialize_H2O_Sample(int  num_mol_input,
    string name_sample,  bool  set_nuc_name);

void PrintBoxToGro(std::string name_dir, std::string name_file);
```

There is a Makefile in 'objects' directory, and it is the only file that users need to modify in this folder.

Postscript Note:

Anyway, we try our best to add detailed comment statements in the readme file and source files. If you have any problems in using the program, please contact with us by email at machangli@ihep.ac.cn or chenghe@ihep.ac.cn .