# libgmxcpp Documentation

***Release 3.2***

**James W. Barnett**

**Feb 10, 2018**

# Contents

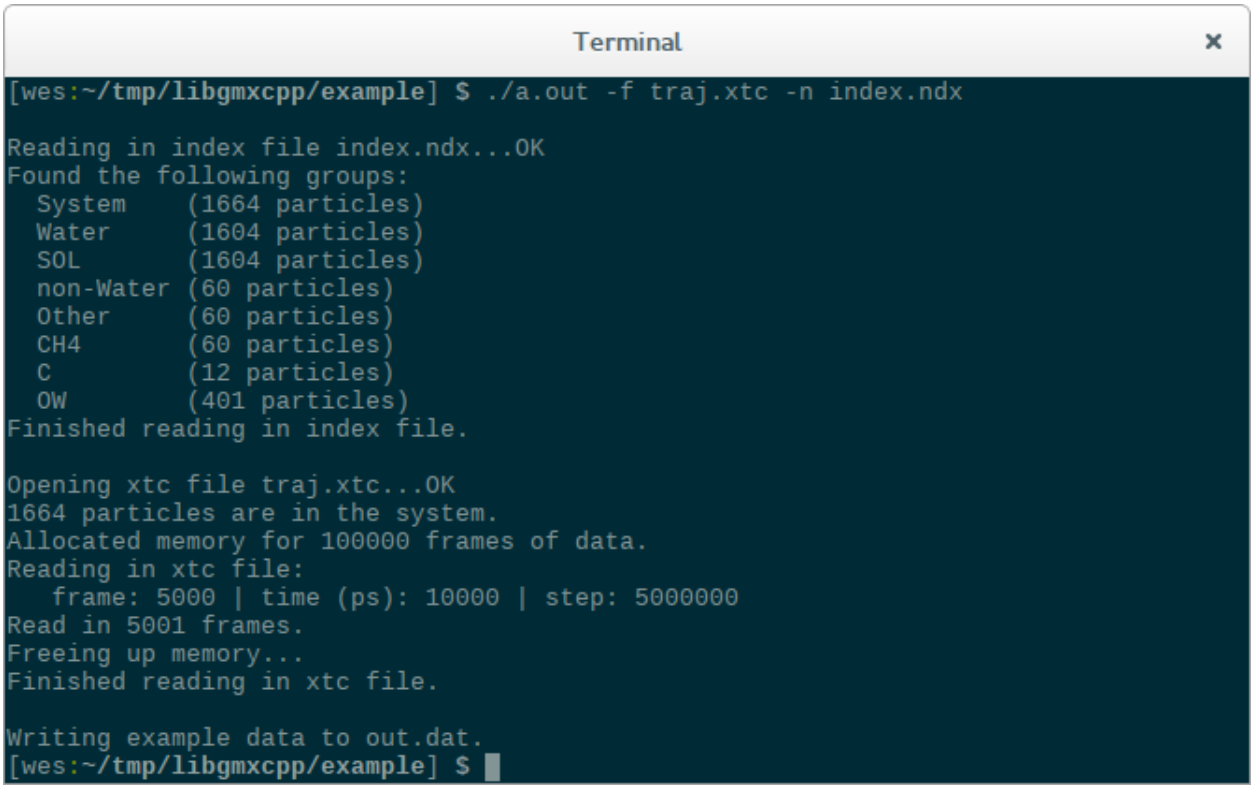
http://github.com/wesbarnett/libgmxcpp

This is a C++ toolkit used for reading in Gromacs files (.xtc, .ndx, and .tpr) for use in analyzing simulation results. This interfaces with libxdrfile and the GROMACS API and implements an object-oriented style. The main usage of the library is to be able to create a Trajectory object which reads in an XTC file along with an optional GROMACS index file such that the user only has to worry with implementing the actual analysis. Several functions which are repeatedly used in Molecular Dynamics analysis (periodic boundary condition calculations, distances, etc.) are also included.

CHAPTER 1

# Advantages

- Only one object construction needs to be called to read in both .xtc and .ndx files.
- Index groups can be used by name within the program to get a desired atom's coordinates.
- Custom classes for atomic coordinates and simulation box allow overloading of operators to simplify coding.
- Common functions such as distance, magnitude, and cross product are built-in.
- Analysis loops can easily be parallelized with class getter functions, since all data frames are initially read in and can be accessed simultaneously.
- No other libraries needed (the relevant parts of libxdrfile are included with this project).




## 1.1 Installation

### 1.1.1 Requirements

`cmake` is required for building the library. Gromacs 2016+ is required, since the library links to some of its functions. xdrfile is required.

### 1.1.2 Installing

A typical install consists of downloading the most recent tarball and extracting it. Enter the source directory. Then do:

```
mkdir build
cd build
cmake ..
make
make install
```

You may need superuser privileges for the last step, or you may need to specify a different installation directory (like your home folder) with the *cmake* option `-DCMAKE_INSTALL_PREFIX` above.

Alternatively if you are running Arch you can install it from the AUR.

### 1.1.3 Classes for use with AVX Instructions

Some classes are provided for use with SIMD intrinsics, specifically the AVX set. To compile with these classes available, add `-DAVX=ON` to your cmake call when installing. This is experimental, since tests have not been implemented

for these classes yet.

Unfortunately I don't have time to cover all instruction sets, so I'm focusing on those most useful to myself. If you're interested in adding more, please file a pull request.

A good example of this in practice, is my test particle insertion code found here. Specifically look at the CalcPE function in Atomtype.cpp.

### 1.1.4 Turning off banner

By default a banner is printed to stderr every time a Trajectory object is created. If this annoys you, use the cmake flag `-DBANNER=OFF` to turn it off.

### 1.1.5 Location

Header files will be installed within a folder named `gmxcpp`.

### 1.1.6 Testing the build

To test your build you can run `make test` in the build directory (see above).

Automated tests were performed via Travis when new commits were pushed, but a newer compiler is required than available. Specifically, "<random>" is used in some utilities.

### 1.1.7 Documentation

If you want to have a local copy of the documentation, do `make docs` in the build directory. The html files will be placed in `docs/html` in your build directory. `sphinx`, `breathe`, and `doxygen` are required to build the documentation. Install `doxygen` with your package manager (*e.g.,* `sudo apt-get install doxygen`). Install `sphinx` and `breathe` with:

```
sudo pip install sphinx
sudo pip install breathe
```

Additionally the source code is well-documented, containing more detail than the generated documentation.

## 1.2 Usage

The basic idea of the library is two-fold and contains two main aspects: 1) Reading in Gromacs files into memory using constructors and using getters to access their information in an analysis program, and 2) a set of basic analysis functions (see next section). Currently libgmxcpp can read in .xtc, .ndx, and .tpr files (tpr files are limited currently to mass and charge). Below is an example workflow which contains both of these aspects. The next two sections contain the API details for the classes and functions.

### 1.2.1 Workflow

This is a suggested workflow for using this library in constructing one's analysis program. As an example this tutorial will walk through creating a program that calculates the center of mass of a group of atoms from a Gromacs simulation.

Let's say you have simulated several methanes in water. In the case of calculating the center of mass of the methanes we'll need the .xtc file (having the coordinates), the .ndx file (grouping the atoms), and the .tpr file (having the masses).

The first thing to do is to construct an object associated for each file type. First we'll read in the index file, since we'll be using it to locate the methanes in the trajectory::

```
Index ndx("index.ndx");
```

Then we'll read in both the .xtc and .tpr files and associate the Index object with it. This is optional, but we want to do it in this case since we can easily find the methanes by our index groups::

```
Trajectory trj("traj.xtc",ndx);
Topology top("topol.tpr",ndx);
```

Now all information from the simulation is available to us using object getters from `trj` and `top`. Since `ndx` is now associated with both of these object we don't have to worry about calling anything from it directly. The first thing you should do is either read in the entire trajectory, or read in some frames. To read in the entire xtc file do:

```
trj.read();
```

To read in only one frame do:

```
trj.read_next();
```

To read in the next 10 frames do:

```
trj.read_next(10);
```

`read_next` is useful in a loop and returns the actual number of frames read in, so you know when you are at the end of the file. It does not close the xtc file like `read` does. To do so simply call:

```
trj.close();
```

In most cases `read()` should be enough unless you are dealing with a large system and run out of memory.

Now that we've called our constructors, we can get any information we want from these objects such as atomic coordinates and masses, which is what we need for getting the center of mass. There is a provided analysis function in the library which gets the center of mass for a group of atoms, removing the periodic boundary condition. For this function we need the atomic coordinates of the atoms in the group we're interested in, the masses of those atoms, and the simulation box for the particular frame we're interested in. Here I know that my simulation is using a cubic box so I am using the `cubicbox` class instead of the `triclinicbox` class. Here's how we can get that info for the methanes from the first frame, where we have an index group with the methanes labeled as `CH4`::

```
vector <coordinates> atom;
vector <double> mass;
cubicbox box;

atom = trj.GetXYZ(0,"CH4");
box = trj.GetCubicBox(0);
mass = top.GetMass("CH4");
```

These getters are described in this documentation on the `Trajectory` and `Topology` class pages. Now to get the center of mass we just call our analysis function::

```
coordinates com;

com = center_of_mass(atom,mass,box);
```

This only works for frame 0 (the first frame), so to do this for each frame we would put this into a loop::

```
coordinates com;
vector <coordinates> atom;
vector <double> mass;
cubicbox box;

Index ndx("index.ndx");
Trajectory trj("traj.xtc",ndx);
trj.read();
Topology top("topol.tpr",ndx);

for (int i = 0; i < trj.GetNFrames(); i++)
{
    atom = trj.GetXYZ(i,"CH4");
    box = trj.GetCubicBox(i);
    mass = top.GetMass("CH4");
    com = center_of_mass(atom,mass,box);
}
```

At this point outputting the data or averaging it, further analysis is up to you. Note that we would have to include the appropriate header files to be able to do this. Additionally the `for` loop can possibly be parallelized depending on the analysis. A full program might be::

```
#include <vector>
#include "gmxcpp/Index.h"
#include "gmxcpp/Topology.h"
#include "gmxcpp/Trajectory.h"
#include "gmxcpp/Utils.h"
using namespace std;

int main()
{

    coordinates com;
    vector <coordinates> atom;
    vector <double> mass;
    triclinicbox box;

    Index ndx("index.ndx");
    Trajectory trj("traj.xtc",ndx);
    trj.read();
    Topology top("topol.tpr",ndx);

    for (int i = 0; i < trj.GetNFrames(); i++)
    {
        atom = trj.GetXYZ(i,"CH4");
        box = trj.GetBox(i);
        mass = top.GetMass("CH4");
        com = center_of_mass(atom,mass,box);
    }

    return 0;
}
```

### 1.2.2 Compiling a Program

Say you have written the above program and saved it to `com.cpp`. To compile you need to link your program to libgmxcpp. Additionally if the headers for your Gromacs installation are in a non-standard installation, which they most probably are, you need to add that path to the `CPLUS_INCLUDE_PATH` environmental variable.

For example:

```
export CPLUS_INCLUDE_PATH=$CPLUS_INCLUDE_PATH:/usr/local/gromacs/include
g++ com.cpp -lgmxcpp
```

The first line needs to be changed depending on your Gromacs installation and can be included in your bash profile so you don't have to add it every time you compile a new program.

### 1.2.3 Other Examples

There is an example program in the `example` directory. Use `make` to compile it and test it out on an .xtc and .ndx file from a recent simulation.

Additionally there is an example program which calculates the radial distribution function using this library.

An example of using `read_next()` in a loop along with using OpenMP for parallelization is found here.

## 1.3 Reading in Files

Below are the three main classes for reading in and accessing information from Gromacs simulation files. Each class contains its own header file in the *gmxcpp* directory which should be included in your own program. See the previous section on some example usages.

### 1.3.1 Index

**class Index**

Class containing index file info.

Contains all information from an index file. When constructed the index file is read in. The names of each group are stored in headers. The locations for each group are stored in the locations vector.

#### Public Functions

**Index**()

Blank constructor for *Index* class.

**Index**(string *ndxfile*)

Constructor which specifies index file.

When constructed the index file is read into the corresponding data elements of the object and can be retrieved with getter functions below.

**Parameters**

- `ndxfile`: Name of index file to be read in.

int **GetGroupSize**(string *groupName*) **const**

Gets the size of an index group.

**Return** Size of the group.

**Parameters**

- `groupName`: Name of group for which size is desired.

int **GetLocation**(string *groupName*, int *atomNumber*) **const**
Gets the index location of the atom in the group specified.

This returns in the index location of an atom relative to the entire system. That is, if you know a specific atom's location relative to an index group, i.e., it is the second atom in a group, then this gives the index number for it for the entire system, i.e., the second atom in a group might be the 300th atom in the system. Look at how an index file is formatted to understand more thoroughly.

**Parameters**

- `groupName`: Name of group where at is located.
- `atomNumber`: The location of the atom in the group.

string **GetFilename**() **const**
Gets the filename associated with this object.

### 1.3.2 Topology

**class Topology**
The main class in reading Gromacs .tpr files.

Class which stores information from a Gromacs topology (tpr) file. Currently just stores the atomic charges and masses in vectors which can be retrieved by getters.

#### Public Functions

**Topology**(string *tprfile*)
Constructor which reads in a GROMACS tpr file.

Constructor which reads in the tpr file. Currently only reads charges and masses of each atom into memory.

**Parameters**

- `tprfile`: Name of the Gromacs tpr file to be read in.

**Topology**(string *tprfile*, *Index* *index*)
Constructor which reads in a GROMACS tpr file and associates an index file with it.

Constructor which reads in the tpr file and associates an index file with it. Currently only reads charges and masses of each atom into memory.

**Parameters**

- `index`: *Index* object to associate with this topology.
- `tprfile`: Name of the Gromacs tpr file to be read in.

double **GetCharge**(int *atom*) **const**
Gets the electric charge of the specified atom.

**Return** The charge (units specified in Gromacs manual)

**Parameters**

- `atom`: The atom

double **GetCharge**(int *atom*, string *group*) **const**
Gets the electric charge of the specified atom in an index group.

**Return** The charge (units specified in Gromacs manual)

**Parameters**

- `atom`: The atom
- `group`: *Index* group

vector<double> **GetCharge**() **const**
Gets the electric charge of all atoms in the system.

**Return** The charge of all atoms in the system (units specified in Gromacs manual)

vector<double> **GetCharge**(string *group*) **const**
Gets the electric charge of the specified index group.

**Return** The charge of all atoms in the index group (units specified in Gromacs manual)

**Parameters**

- `group`: *Index* group

double **GetMass**(int *atom*) **const**
Gets the mass of the specified atom.

**Return** The mass (units specified in Gromacs manual)

**Parameters**

- `atom`: The atom

double **GetMass**(int *atom*, string *group*) **const**
Gets the mass of the specified atom in an index group.

**Return** The mass (units specified in Gromacs manual)

**Parameters**

- `group`: *Index* group

vector<double> **GetMass**() **const**
Gets the mass of all atoms in the system.

**Return** The mass of all atoms in the system (units specified in Gromacs manual)

vector<double> **GetMass**(string *group*) **const**
Gets the mass of the specified index group.

**Return** The mass of all atoms in the inde group (units specified in Gromacs manual)

**Parameters**

- `group`: *Index* group

string **GetElem**(int *atom*)
Gets the element name of an atom.

**Return** Name of the element

**Parameters**

- `atom`: The atom number

string **GetElem**(int *atom*, string *group*)
Gets the element name of an atom in a specified group.

**Return** Name of the element

**Parameters**

- `atom`: The atom number
- `group`: *Index* group of which the atom belongs

string **GetAtomName**(int *atom*)
Gets the atom name of an atom.

**Return** Name of the atom

**Parameters**

- `atom`: The atom number

string **GetAtomName**(int *atom*, string *group*)
Gets the element name of an atom in a specified group.

**Return** Name of the element

**Parameters**

- `atom`: The atom number
- `group`: *Index* group of which the atom belongs

string **GetResName**(int *atom*)
Gets the residue name of an atom.

**Return** Name of the residue

**Parameters**

- `atom`: The atom number

string **GetResName**(int *atom*, string *group*)
Gets the residue name of an atom in a specified group.

**Return** Name of the residue

**Parameters**

- `atom`: The atom number

- `group`: *Index* group of which the atom belongs

### 1.3.3 Trajectory

**class Trajectory**

The main class in reading Gromacs files.

A *Trajectory* object contains a vector of Frame objects, plus other info on the simulation (number of atoms). It also contains the special xd pointer that libxdrfile needs to open the xtc file, as well as the number of atoms in the system, the number of frames read in, and an *Index* object.

#### Public Functions

**Trajectory**(string *xtcfile*)

Constructor where only XTC file is read.

Constructor of *Trajectory* object, with no index file specified

**Parameters**

- `xtcfile`: Name of the Gromacs XTC file to be read in. file.

**Trajectory**(string *xtcfile*, *Index index*)

Constructor which sets both the XTC file and incorporates a previously read in *Index* object.

When this constructor is used, both the Gromacs XTC file is saved in the vector of Frame objects, and the group names and index numbers from an *Index* object are copied into the *Trajectory* object.

**Parameters**

- `xtcfile`: Name of the Gromacs XTC file to be read in.
- `index`: The *Index* object which has already had its index file read in.

**Trajectory**(string *xtcfile*, string *ndxfile*)

Constructor that sets both the XTC file and a GROMACS index file.

When this constructor is used, both the Gromacs XTC file is saved in the vector of Frame objects, and the group names and index numbers for the index file are saved in an *Index* object.

**Parameters**

- `xtcfile`: Name of the Gromacs XTC file to be read in.
- `ndxfile`: Name of the Gromacs index file to be read in.

int **read**(int *b* = 0, int *s* = 1, int *e* = -1)

Reads in simulation frames into memory and then closes the file.

**Return** Number of frames read in.

**Parameters**

- `b`: First frame to be read in. By default, starts at the first frame (frame 0).
- `s`: Read in every sth frame.
- `e`: Stop reading at this frame. -1 means read until the end of the

int **read_next** (int *n* = 1)
Reads in n simulations frames into memory and keeps the file open.

Frames are saved into the frameArray object, overwriting previously saved frames

**Return** Number of frames actually read in.

**Parameters**

- `n`: Number of frames to read into memory.

int **skip_next** (int *n* = 1)
Skip n frames.

**Return** Number of frames actually skipped

**Parameters**

- `n`: Number of frames to skip

int **GetNAtoms** () **const**
Gets the number of atoms in a system.

**Return** Number of atoms.

int **GetNAtoms** (string *groupName*) **const**
Gets the number of atoms in an index group.

**Return** number of atoms in the group specified.

**Parameters**

- `groupName`: Name of group for which number of atoms is returned.

int **GetNFrames** () **const**
Gets the number of frames that were saved.

**Return** Number of frames.

float **GetTime** (int *frame*) **const**
Gets the time at frame specified.

**Return** Time in picoseconds.

**Parameters**

- `frame`: Number corresponding with the frame for which time should be returned.

int **GetStep** (int *frame*) **const**
Gets the step at frame specified.

**Return** Step number.

**Parameters**

- `frame`: Number corresponding with the frame for which step should be returned.

coordinates **GetXYZ**(int *frame*, int *atom*) **const**
Gets the coordinates of a specific atom in the entire system.

Gets the cartesian coordinates for the atom specified at the frame specified and returns it as a vector

**Return** Vector with X, Y, and Z coordinates of the atom specified.

**Parameters**

- `atom`: The number corresponding with the atom in the entire system.
- `frame`: Number of the frame desired.

coordinates **GetXYZ**(int *frame*, string *groupName*, int *atom*) **const**
Gets the coordinates for a specific atom in a group.

Gets the cartesian coordinates for the atom specified in the specific index group for this frame.

**Return** Vector with X, Y, and Z coordinates of the atom specified.

**Parameters**

- `frame`: Number of the frame desired.
- `groupName`: Name of index group in which atom is located.
- `atom`: The number corresponding with the atom in the index group. Note that this is **not** the same number corresponding with the system. That is, the atom may be the 5th atom in the system, but it may be the 2nd atom in the group. This is where it is located in the group.

vector<coordinates> **GetXYZ**(int *frame*) **const**
Gets all of the coordinates for the system for a specific frame.

**Return** A two dimensional vector with all cartesian coordinates for the system at this frame. The first dimension is the atom number. The second dimension contains the X, Y, and Z positions.

**Parameters**

- `frame`: Number of the frame desired.

vector<coordinates> **GetXYZ**(int *frame*, string *groupName*) **const**
Gets all of the coordinates for an index group for a specific frame.

**Return** A two dimensional vector with all cartesian coordinates for the system at this frame. The first dimension is the atom number in the group. The second dimension contains the X, Y, and Z positions.

**Parameters**

- `frame`: Number of the frame desired.
- `groupName`: Name of index group in which atom is located.

triclinicbox **GetBox**(int *frame*) **const**
Gets the triclinic box dimensions for a frame.

**Return** Two-dimensional array with three elements in each dimension, corresponding to a triclinic box.

**Parameters**

- `frame`: Number of the frame desired.

double **GetBoxVolume**(int *frame*) **const**
Gets the volume of the box at a specific frame.

**Return** Box volume.

**Parameters**

- `frame`: Number of the frame desired.

## 1.4 Analysis Functions

In addition to being able to read in trajectories and index files, some basic analysis functions are included in the API. These are not intended to be exhaustive of all possible analytical tools. Instead, this is a simple framework the analyst can use in writing his own programs. All of these are currently found in `gmxcpp/Utils.h`, except for the clustering routines, which are found in `gmxcpp/Clusters.h`.

### 1.4.1 Bond vector

**Warning:** doxygenfunction: Unable to resolve multiple matches for function "bond_vector" with arguments (coordinates, coordinates, triclinicbox)) in doxygen xml output for project "libgmxcpp" from directory: ./doxyxml. Potential matches:

```
- Analysis some basic analysis functions are included in the API These are not␣
↪intended to be exhaustive of all possible analytical tools this is a simple␣
↪framework the analyst can use in writing his own programs All of these are␣
↪currently found in gmxcpp Utils except for the clustering which are found in␣
↪gmxcpp Clusters h Bond vector doxygenfunction::bond_vector(coordinates,␣
↪coordinates, triclinicbox)
- coordinates bond_vector(coordinates, coordinates, cubicbox)
```

### 1.4.2 Bond angle

**Warning:** doxygenfunction: Unable to resolve multiple matches for function "bond_angle" with arguments (coordinates, coordinates, triclinicbox)) in doxygen xml output for project "libgmxcpp" from directory: ./doxyxml. Potential matches:

```
- double bond_angle(coordinates, coordinates, coordinates, cubicbox)
- double bond_angle(coordinates, coordinates, coordinates, triclinicbox)
```

### 1.4.3 Center a group of atoms around a point

**Warning:** doxygenfunction: Unable to resolve multiple matches for function "do_center_group" with arguments () in doxygen xml output for project "libgmxcpp" from directory: ./doxyxml. Potential matches:

```
- void do_center_group(vector<coordinates>&, coordinates, cubicbox)
- void do_center_group(vector<coordinates>&, coordinates, triclinicbox)
- void do_center_group(vector<coordinates>&, cubicbox&)
```

### 1.4.4 Center of mass

**Warning:** doxygengroup: Cannot find namespace "center_of_mass" in doxygen xml output for project "libgmxcpp" from directory: ./doxyxml

### 1.4.5 Clustering

**class Clusters**

Class containing clustering functions.

This class is used for clustering molecules based off of a cutoff distance between the various atomic sites on each molecule in question. Before clustering can be performed, the object must be constructed. Then "do_clustering" can be called for each frame one desires to cluster together. Initially each molecule is in its own cluster of size one. After performing the clustering routine one can get which molecules are part of a cluster, get the cluster for which a molecule belongs, and get the size of the cluster. The functions are only appropriate for clustering molecules of the same type.

#### Public Functions

**Clusters**(int *mol_n*, int *atoms_per_mol*)

Constructor for a *Clusters* object.

**Parameters**

- `mol_n`: Total number of molecules that are going to be processed.
- `atoms_per_mol`: Number of atoms in each molecule that are going to be processed.

void **do_clustering**(int *frame*, *Trajectory* &*traj*, double *rcut2*)

Perform clustering on all molecules in xtc file.

This version performs clustering on all molecules in the *Trajectory* object. This is useful when, say, only the solutes are in the trajectory file that was read in. After this function is called one can get information on the clusters using the getters in this class.

**Parameters**

- `frame`: The frame number to do clustering on.
- `traj`: The trajectory object with the molecules
- `rcut2`: The cutoff length squared for determining if molecules are in the same cluster. The cutoff is measured between atomic sites on each molecule. If any two sites are within the cutoff the two molecules are in the same cluster.

void **do_clustering**(int *frame*, *Trajectory* &*traj*, string *group*, double *rcut2*)

Perform clustering on a specific index group.

This version only performs the clustering routine on a specific index group. After this function is called one can get information on the clusters using the getters in this class.

**Parameters**

- `frame`: The frame number to do clustering on.
- `traj`: The trajectory object with the molecules

- `group`: The index group to do clustering on.
- `group`: The index group to do clustering on.
- `rcut2`: The cutoff length squared for determining if molecules are in the same cluster. The cutoff is measured between atomic sites on each molecule. If any two sites are within the cutoff the two molecules are in the same cluster.

int **`get_size`** (int *clust*)
Get the size of the cluster.

This return the number of molecules in a cluster given the cluster number. This should only be performed after 'do_clustering' has been done for the frame. Otherwise each cluster will be of size one. After doing 'do_clustering' several clusters will be of size zero, since initially each molecule is in it's own cluster.

**Return** The cluster size, indicating the number of molecules in a cluster.

**Parameters**

- `clust`: The cluster number.

int **`get_index`** (int *mol*)
Get the cluster number given a molecule.

This should only be called after performing 'do_clustering' for a frame. Initially each molecule will be in its own cluster.

**Return** The cluster number to which the molecule belongs.

**Parameters**

- `mol`: The number indicating the molecule of interest, corresponding to the order in the trajectory object.

vector<int> **`get_mol_numbers`** (int *clust*)
Find out which molecules belong to a cluster.

**Return** A vector of numbers indicating which molecules are part of this cluster.

**Parameters**

- `clust`: The cluster number.

### 1.4.6 Cross product

coordinates **`cross`** (coordinates *a*, coordinates *b*)
Calculates the cross product.

Gets the cross product between vectors a and b and returns it.

**Return** The resultant vector of the cross of a and b.

**Parameters**

- `a`: First vector to be crossed.
- `b`: Second vector to be crossed.

### 1.4.7 Dihedral angle

double **dihedral_angle** (coordinates *atom1*, coordinates *atom2*, coordinates *atom3*, coordinates *atom4*, triclinicbox *box*)

Calculates the torsion / dihedral angle from four atoms' positions.

Source: Blondel and Karplus, J. Comp. Chem., Vol. 17, No. 9, 1 132-1 141 (1 996). Note that it returns in radians and that the atoms should be in order along their connections.

**Return** dihedral angle in radians

**Parameters**

- `atom1`: First atom in angle
- `atom2`: Second atom in angle
- `atom3`: Third atom in angle
- `atom4`: Fourth atom in angle
- `box`: Simulation box

### 1.4.8 Distance

**Warning:** doxygenfunction: Unable to resolve multiple matches for function "distance" with arguments (coordinates, coordinates, triclinicbox)) in doxygen xml output for project "libgmxcpp" from directory: ./doxyxml. Potential matches:

```
- double distance(coordinates, coordinates)
- double distance(coordinates, coordinates, triclinicbox)
```

### 1.4.9 Distance squared

**Warning:** doxygenfunction: Unable to resolve multiple matches for function "distance2" with arguments (coordinates, coordinates, triclinicbox)) in doxygen xml output for project "libgmxcpp" from directory: ./doxyxml. Potential matches:

```
- Distance squared doxygenfunction::distance2(coordinates, coordinates,␣
↪triclinicbox)
- __m256 distance2(coordinates8, coordinates8, cubicbox8)
- __m256 distance2(coordinates8, coordinates8, cubicbox_m256)
- double distance2(coordinates, coordinates)
- double distance2(coordinates, coordinates, cubicbox)
```

### 1.4.10 Dot product

**Warning:** doxygenfunction: Unable to resolve multiple matches for function "dot" with arguments (coordinates, coordinates)) in doxygen xml output for project "libgmxcpp" from directory: ./doxyxml. Potential matches:

```
- Dot product doxygenfunction::dot(coordinates, coordinates)
- double dot(coordinates)
```

### 1.4.11 Geometric center

coordinates **center_of_geometry** (vector<coordinates> &*atom*, cubicbox &*box*)

Gets the geometric of a group of atoms.

Gets the gemetric of a group of atoms, taking into account the periodic boundary condition. *

**Return** Geometric center.

**Parameters**

- `atom`: The positions of the atoms. Note this only works for a cubic box at the moment.
- `atom`: The positions of the atoms.
- `box`: The simulation box.

### 1.4.12 Periodic boundary condition

coordinates **pbc** (coordinates *a*, triclinicbox *box*)

Adjusts for periodic boundary condition.

User passes a vector, most likely a vector pointing from one atom to another in the simulation. This function adjusts the vector such that if it is longer than 1/2 the box size it accounts for the periodic boundary.

**Return** Vector after pbc accounted for.

**Parameters**

- `a`: Vector to be passed.
- `box`: The box dimensions (can be either triclinicbox or cubicbox).

### 1.4.13 Random points in a box

**Warning:** doxygengroup: Cannot find namespace "gen_rand_box_points" in doxygen xml output for project "libgmxcpp" from directory: ./doxyxml

### 1.4.14 Random point on sphere

coordinates **gen_sphere_point** ()

Generates a random point on a unit sphere at the origin.

**Return** The coordinates of the random point.

### 1.4.15 Surface area

double **get_surf_area** (vector<coordinates> *sites*, double *r*, double *rand_n*, triclinicbox *box*)

Gets the surface area of a group of atoms.

Gets the surface area of a group of atoms (could be a molecule) defined by vector of coordinates. Randomly generated points on a sphere of radius r are used at each site in order to get an acceptance ratio. The surface area

contributed from each site is simply the surface area of a sphere multiplied by the acceptance ratio for that site. The total surface area is the sum of the surface areas for each site.

**Parameters**

- `sites`: The coordinates of sites in the group / molecule. For example, the carbons in an alkane.
- `r`: The radius to be used in determining the surface area. For example, to determine the SASA use the appropriate radius.
- `rand_n`: The number of randomly generated points to be used for each site.
- `box`: The box dimensions for the frame in question.

### 1.4.16 Vector magnitude

double **`magnitude`** (coordinates *x*)
Calculates the magnitude of a vector.

**Return** Magnitude

**Parameters**

- `x`: Vector for which magnitude is desired

### 1.4.17 Volume of Box

double **`volume`** (triclinicbox *box*)
Calculates the volume of simulation box.

**Return** Volume of box

**Parameters**

- `box`: Box dimensions

## 1.5 License



I hope you find this library useful. There is no paper associated with this project to cite as is common in some projects. However, if you do use this code in a published work I humbly ask that you acknowledge it in some way.

# Index