The GROMOS Software for (Bio)Molecular Simulation

Volume 6: Technical Details

August 24, 2012

Contents

CHAPTER 1

Outline of the GROMOS Code

1.1. PROMD outline

PROMD contains three major routines for performing a simulation:

1. *PROGRAM PROMD:* Starts program, reads input files, etc.
2. *SUBROUTINE DOINIT:* Initialises all variables needed for simulation.
3. *SUBROUTINE DORUN:* Simulation loop.

All simulation procedures will have to run through the steps defined in these routines, where each step is applied depending on the simulation procedure or simulation parameters defined. E.g. a step only to be applied for EM will be encapsulated by IF (MDSWITCH%L_EM) THEN.

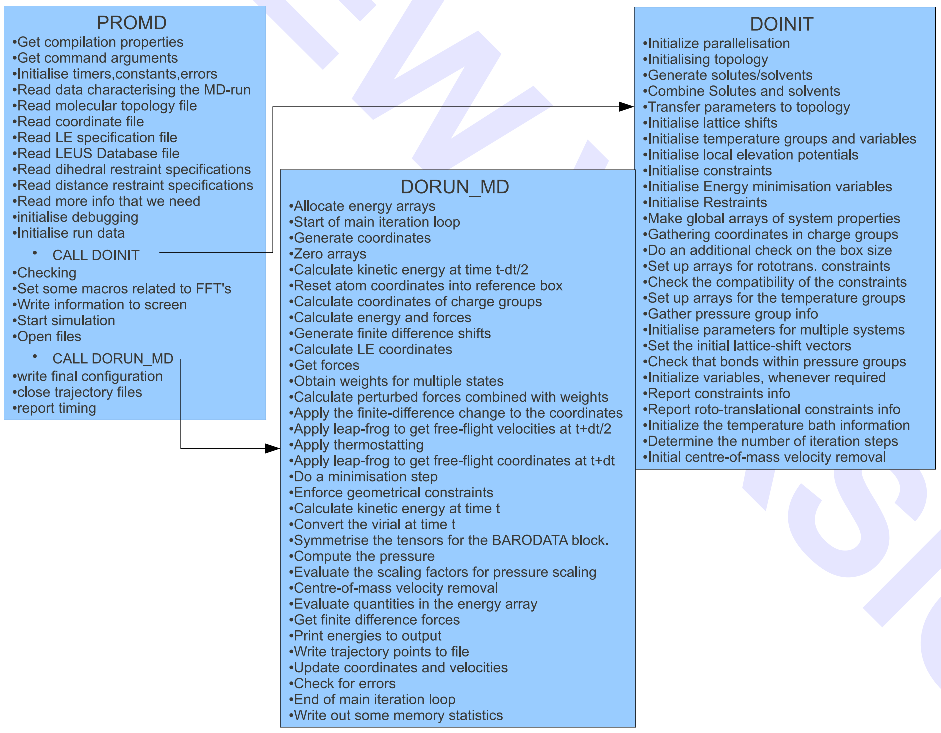
PROMD
•Get compilation properties
•Get command arguments
•Initialise timers,constants,errors
•Read data characterising the MD-run
•Read molecular topology file
•Read coordinate file
•Read LE specification file
•Read LEUS Database file
•Read dihedral restraint specifications
•Read distance restraint specifications
•Read more info that we need
•initialise debugging
•Initialise run data
• CALL DOINIT
•Checking
•Set some macros related to FFT's
•Write information to screen
•Start simulation
•Open files
• CALL DORUN_MD
•write final configuration
•close trajectory files
•report timing
DORUN_MD
•Allocate energy arrays
•Start of main iteration loop
•Generate coordinates
•Zero arrays
•Calculate kinetic energy at time t-dt/2
•Reset atom coordinates into reference box
•Calculate coordinates of charge groups
•Calculate energy and forces
•Generate finite difference shifts
•Calculate LE coordinates
•Get forces
•Obtain weights for multiple states
•Calculate perturbed forces combined with weights
•Apply the finite-difference change to the coordinates
•Apply leap-frog to get free-flight velocities at t+dt/2
•Apply thermostatting
•Apply leap-frog to get free-flight coordinates at t+dt
•Do a minimisation step
•Enforce geometrical constraints
•Calculate kinetic energy at time t
•Convert the virial at time t
•Symmetrise the tensors for the BARODATA block.
•Compute the pressure
•Evaluate the scaling factors for pressure scaling
•Centre-of-mass velocity removal
•Evaluate quantities in the energy array
•Get finite difference forces
•Print energies to output
•Write trajectory points to file
•Update coordinates and velocities
•Check for errors
•End of main iteration loop
•Write out some memory statistics
DOINIT
•Initialize parallelisation
•Initialising topology
•Generate solutes/solvents
•Combine Solutes and solvents
•Transfer parameters to topology
•Initialise lattice shifts
•Initialise temperature groups and variables
•Initialise local elevation potentials
•Initialise constraints
•Initialise Energy minimisation variables
•Initialise Restraints
•Make global arrays of system properties
•Gathering coordinates in charge groups
•Do an additional check on the box size
•Set up arrays for rototrans. constraints
•Check the compatibility of the constraints
•Set up arrays for the temperature groups
•Gather pressure group info
•Initialise parameters for multiple systems
•Set the initial lattice-shift vectors
•Check that bonds within pressure groups
•Initialize variables, whenever required
•Report constraints info
•Report roto-translational constraints info
•Initialize the temperature bath information
•Determine the number of iteration steps
•Initial centre-of-mass velocity removal

FIGURE 1.1. Outline of the most important subroutines in PROMD.

See Fig. 1.1 for a detailed description of the different steps in these routines. These three routines call all other necessary routines. Important routines are described in Sec. 1.1.2.

PROMD uses MODULES for storing global data and related subroutines. The different modules are topic-based.

1.1.1. Important PROMD modules and data types.

`LOGICALSMOD`	defines a set of logicals defining the simulation run. These direct the application of steps in DOINIT and DORUN.

MDMOD	contains the data read from the input file and related subroutines
COORDMOD	contains data such as coordinates, velocities, forces, specifications and related subroutines for reading and writing.
TOPOMOD	contains the topologies read, the internal topology and related subroutines.
NONBONDEDMOD	contains subroutines for nonbonded interaction calculations (pair list generation and interaction evaluation)
COVALENTMOD	contains subroutines for covalent interaction evaluations
DEBUGMOD	contains subroutines and arrays for debug calls
ERRORMOD	contains subroutines and arrays for error handling
FILEIOMOD	contains subroutines for IO operations
MATHMOD	contains subroutines for mathematical operations
TIMINGMOD	contains subroutines for timing
PRECISIONMOD	contains variable precision definition and numerical constants

1.1.2. Important PROMD routines.

FORCE	This subroutine calculates the energy and the force of the current configuration
CONSTR	Apply constraints
SORT_X	Sort charge groups into grid cells
EXPAND_X	Duplicate box sides
MAKE_MASK	Generate grid pair list mask
INIT_LIST_XX	Generate grid pair list (XX may be IR or SR for intermediate or short range pair list)
NONB_C_XX	Calculate nonbonded interactions based on a pair list (XX varies according to the chosen pair list and long range correction)
CONTRACT_X	Reduce forces of an expanded system to the original system
COV_XX	Calculate covalent interactions (XX has the value BND for bonds, ANG for angles, IMP for improper dihedrals and DIH for proper dihedrals)

1.1.3. Internal topology in PROMD. The internal topology is the set of parameters actually used in the force calculations and is generated from the topologies read according to the input file.

The internal topology has three basic components

FFPARAMS	(TYPE_TOPO_PARAM) All force-field parameters used in the simulation ordered by types ($\neq$ to types defined in input topology, e.g. if only three different sets of Lennard-Jones parameters are used in a simulation, these are stored as type 1 to 3)
SOLUTE	(TYPE_TOPO_SOLUTE) All force-field types defined for the solute (bond types, charge types, exclusions, etc.)
SOLUTE_PERT	(TYPE_TOPO_SOLUTE_PERT) All force-field types defined for the perturbed part of the solute
SOLVENT	(TYPE_TOPO_SOLVENT) All force-field types defined for the solvent (restraint types, charge types, etc.)

In many cases it is useful to operate with only one topology for the whole system instead of a set of solute and solvent topologies. A common topology for the whole system is therefore generated.

`SYSTEM`	(TYPE_TOPO_SOLUTE) All force-field types defined for the whole system (solutes + solvents)

It is also useful to have the force-field parameters available in the same format as the corresponding types found in SOLUTE, SOLVENT and SYSTEM, thus these parameters are copied from FFPARAMS with the subroutine PARAMS_TO_SOLUTE and PARAMS_TO_SOLVENT. The topology format is designed such that the parameters defined in FFPARAMS are the dominant ones, thus the subroutines PARAMS_TO_*** should be rerun if FFPARAMS changes. This ensures that all topology data structures are consistent within the program.

1.1.4. Compact arrays. For performance reasons it is important to store large arrays as compactly as possible. PROMD therefore uses a special data structure for the pairlist and for the exclusions which only requires one 32-bit integer per pair. The pairlist (TYPE_G96PLIST) consists of four arrays

`CGID_A(`N_{CG}`)`	The set of charge groups
`N_NEIGH(`N_{CG}`)`	The number of neighbours of each charge group, I, in CGID_A
`N_START(`N_{CG}`)`	The starting position in CGID_B of the neighbours of a given charge group I
`CGID_B(`N_{pairs}`)`	The neighbours of each charge group in CGID_A. The neighbours of charge group I are given by CGID_B(N_START(I) + 1:N_START(I) + N_NEIGH(I))

The memory usage of this storage scales as

$$3N_{CG} + N_{pairs} \approx N_{pairs} = N_{CG}N_{cut} \tag{1.1}$$

where N_{CG}, N_{pairs} and N_{cut} are the number of charge groups, neighbouring pairs and neighbours per charge group, respectively. The ordering of array CGID_A depends on the pair list generation routine.

The exclusions are stored in a similar data structure (TYPE_EXCLUSION) and consist of

`N_EX(`N_{atoms}`)`	The number of exclusions that a given atom, I, has with atoms of higher atom number
`N_START(`N_{atoms}`)`	The starting position in EXCLUSIONS of the excluded atoms for a given atom I
`EXCLUSIONS(`$N_{exclusions}$`)`	The atom index of the excluded atoms for each atom in N_EX. The indices of the excluded atoms of atom I are given in EXCLUSIONS(N_START(I) + 1:N_START(I) + N_EX(I))

Note that only exclusions to higher atom numbers are listed (to prevent double counting) and that this array (unlike the pairlist array) is sorted such that exclusions are listed in increasing order.

1.2. MD++ outline

The code is split into two parts, the first one being an MD library containing basic functions necessary to run an MD simulation, the second one being the actual MD program. This second part is very small. It is therefore easy to write other specialised MD programs that make use of a subset of the functions provided in the library or apply them in a different order. The source code of the library is in turn split up into nine different parts: *math*, *simulation*, *topology*, *configuration*, *algorithm*, *interaction*, *io*, *util* and *check* (represented as C++ *namespaces*).

- *math* contains classes for vectors, matrices and vector arrays, mathematical operations, physical constants and periodic boundary treatment.
- *simulation* contains the simulation parameters supplied to run an MD or SD simulation or an EM.
- *topology* contains the topology of the simulated system, possibly also including a perturbation topology.
- *configuration* contains the state of a system: its coordinates, velocities, forces, restraints data and so on.
- *algorithm* contains classes that use information from *simulation* and *topology* to act upon a *configuration*. All steps during an MD or SD simulation or EM can be carried out using an *algorithm*.

- *interaction* contains the largest algorithm: the energy, forces and virial evaluation. Here, all interaction terms and their parameters are defined. Because of its size, *interaction* is a separate part, though it formally belongs to *algorithm*. The *interaction* part is further split into *bonded*, *nonbonded* and *special* interactions.
- *io* contains classes to read in or write out information. All file access is block oriented and human readable.
- *util* contains a few extra classes that are necessary to set up a simulation but which do not exactly belong to it. Parsing of command line arguments, generation of initial velocities or setting of debug levels are examples of classes found herein.
- *check* contains test routines. Testing includes the automatic calculation of energies under different conditions as well as the calculation of forces, virial tensor and energy λ-derivatives and their comparison to values obtained by finite difference calculations.

One step of an MD or SD simulation or EM consists of several `Algorithms` (List. 1.1) applied to the `Configuration` in the right order.

```
class Algorithm {
  public:
  Algorithm(string name) : name(name) {}
  ~Algorithm() {}
  virtual int init(Topology & topo,
                       Configuration & conf,
                   Simulation & sim) = 0;

  virtual int apply(Topology & topo,
                    Configuration & conf,
                    Simulation & sim) = 0;

  string name;
};
```

Listing 1.1. Interface of the `Algorithm` class

The `Algorithm_Sequence` class (List. 1.2) is a container for all these algorithms. When a simulation is set up, they are inserted in the correct order into the `Algorithm_Sequence`. Before the start of a simulation, all algorithms will be initialised (by calling the `init()` function). During an MD step (`Algorithm_Sequence::run()`), the algorithms are applied (by calling `Algorithm::apply()`).

```
class Algorithm_Sequence : public vector<Algorithm *> {
  public:
  Algorithm_Sequence();
  ~Algorithm();

  int init(Topology & topo,
           Configuration & conf,
           Simulation & sim);

  int run(Topology & topo,
          Configuration & conf,
          Simulation & sim);

  Algorithm * algorithm(string name);
};
```

Listing 1.2. Interface of the `Algorithm_Sequence` class. It is a container for `Algorithm` objects which provides methods to initialise and run the contained algorithms. It further provides access by name.

The force-field itself is also an `algorithm`, which, when applied, calculates the energies, forces and virial contribution of all force-field terms for the complete system. The force-field terms themselves are `Interaction` classes. The `Forcefield` is therefore a container to store the different `Interaction` objects (in analogy to the `Algorithm_Sequence` and `Algorithm` classes). When the force-field is applied, it calls `calculate_interactions()` on all interaction objects. There are distinct interaction objects for the covalent interactions (bond-length, bond-angle, improper-dihedral and torsional-dihedral interactions), the non-bonded interactions (pairlist construction, long-range interactions and short-range interactions) and the non-physical interactions (atom-position, atom-distance, dihedral-angle, NOE or ^3J-value restraints). It is very easy to add a custom `Interaction` class to calculate a non-standard interaction. An overview of the (non-bonded) interaction classes is given in Fig. 1.2. The `Nonbonded_Sets` contain independent subsets of the non-bonded interactions. Their `calculate_interactions()` method may be called in parallel (using either *shared* or *distributed* memory parallelisation). The `Nonbonded_Sets` share (through the `Nonbonded_Interaction`) a pairlist construction algorithm, which they call to create the part of the complete pairlist relevant to them. These different parts of the pairlist stay together with the `Nonbonded_Set` and need never be assembled into the complete pairlist. To gain flexibility, the calculation of the individual atom - atom pair interaction is further split up into a `Nonbonded_Outerloop` (loops over the atom - atom pairs), a `Nonbonded_Innerloop` (prepares the parameters necessary to calculate the interaction) and a `Nonbonded_Term` (calculates the atom - atom pair interaction energy, force and virial contribution). The `Storage` class provides directly accessible (local) memory for each `Nonbonded_Set`.

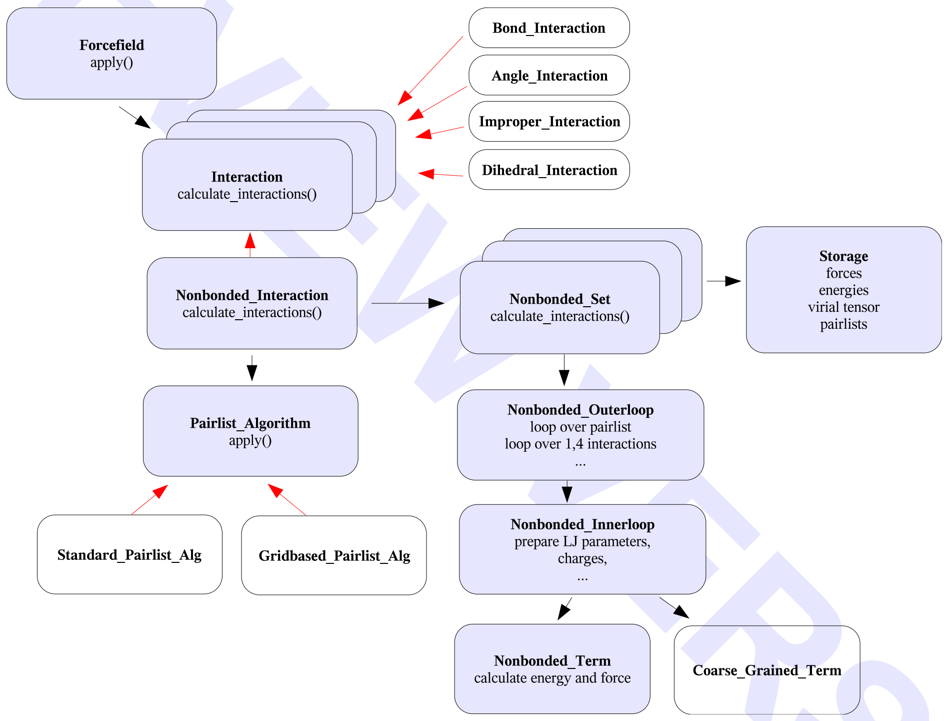

FIGURE 1.2. Illustration of the `Interaction` classes in MD++. The red arrows denote a *is-a* relationship, the black arrows *has-a*. All `Interaction` classes inherit from `Interaction` and, therefore, can be stored in the `Forcefield`, which is a `vector` of `Interaction` classes. The `Nonbonded_Interaction` consists of a `Pairlist_Algorithm` (either a `Standard_Pairlist_Algorithm` or a `Grid_Pairlist_Algorithm`) and (depending on parallelisation) one or more `Nonbonded_Sets`. Those, in turn, consist of `Storage` (to locally store forces, energies, virial tensor and pair lists) and an `Outerloop` (to calculate the interactions). The `Outerloop` relies on the `Innerloop` and on `Term` to calculate the interactions.

1.2.1. Efficiency. The main goal for writing a new C++ MD engine was to further improve on modularity (using some object-oriented features) and extendability (using clear and common interfaces between the modules). Nevertheless, a simulation code has to be reasonably efficient to be of practical use. The complete code is written in standard C++[1] with no language extensions or machine-specific parts, resulting in a highly portable program. This means that the compiler has to do all machine specific optimisations. We believe that the absence of any machine specific parts of code, which require duplication to be able to run on different machines, facilitates future modification. Furthermore, current compilers are getting ever better at producing fast programs, making use of the specific features available on the machine. In the inner loops of the interaction calculation, templates are used to generate specialised code. There are, for instance, specialised periodicity classes for the different implemented types of periodic boundary conditions (vacuum, rectangular, truncated octahedral and triclinic). The `Innerloop` methods are called with the boundary type as a template argument. Thus the compiler will generate a different specialised version of the inner loops for different boundary conditions automatically. In the same manner, the interaction function term of the non-bonded interaction can also be chosen (*e.g.* with or without switching function for non-bonded interactions) without any *if* statement required in the compiled inner loop. Example code fragments are shown in List. 1.3 and List. 1.4. The same technique is used to implement perturbation simulations and different definitions of the virial tensor.

```
enum boundary_type {vacuum, rectangular, triclinic};
template<boundary_type boundary>
class Periodicity;

template<>
class Periodicity<vacuum>{
public:
  void nearest_image(Vec const & ri, Vec const & rj, Vec & rij);
};

template<>
class Periodicity<rectangular>{
public:
  void nearest_image(Vec const & ri, Vec const & rj, Vec & rij);
};

template<>
class Periodicity<triclinic>{
public:
  void nearest_image(Vec const & ri, Vec const & rj, Vec & rij);
};

template<boundary_type boundary>
class Interaction{
public:
  virtual int calculate_interactions(Topology const & topology,
                                     Configuration & configuration,
                                     Simulation const & simulation) {

    Vec r;
    Periodicity<boundary> periodicity(configuration.current().box);

    periodicity.nearest_image(
            configuration.current().pos(0),
            configuration.current().pos(1),
            r);
    const double r2 = math::abs2(r);
    // and so on
    return 0;
  }
};
```

LISTING 1.3. Specialzed code generation using templates.

```
int main(int argc, char **argv) {
  Interaction<triclinic> interaction;
  interaction.calculate_interactions(
           topology, configuration, simulation);
  return 0;
}
```

LISTING 1.4. The usage of periodic boundary condition specific templates demonstrated on the `Interaction` class.

Some algorithms do rely on information from the previous integration step. To help implementing those kinds of algorithms, the complete current and old state (positions, velocities, forces, energies, restraint and constraint data, averages, and so on) of the simulation are stored. During the leap-frog algorithm, the current state becomes the old state and the updated information is stored in the new current state. This transfer is done by a simple and fast pointer exchange. This slightly increases memory usage, but the required space is still small compared to that used to store the pairlists.

1.2.2. Debugging information. It is often difficult to figure out what is going on during an MD or SD simulation or an EM and users tend to use the program as a *black box*. MD++ tries to improve this situation by enabling the user to select a tuneable amount of information to be printed out during the simulation. Every output or debugging message is associated with a debugging level, and the message is printed only if the requested debugging level is high enough. Additionally, every code section belongs to a *module* and a *submodule*. Different debug levels can be specified for all combinations of *modules* and *submodules*. In that way, fine grained control is achieved on how much information from which part of the MD++ code should be printed. For example, running MD++ like this

```
~/> md @f md.args @verb interaction:special:4
```

will print all debug messages in the interaction/special part of the code with a level lower than four. Additional information on debugging can be found in the `doxygen` documentation.

1.2.3. In-code documentation. All classes, structures and enumerations are documented *in-code* using the `doxygen` documentation tool. This documentation contains descriptions of the classes of MD++ and their usage. Inheritance diagrams, function call relationships and interactive links to other classes are automatically generated by the tool. The documentation further contains a brief description of the current input formats used in the given version of MD++. See Sec. 8-3.1 on how to generate the `doxygen` documentation during the compilation procedure of MD++.

1.3. GROMOS++ outline

GROMOS++ is a software package providing the user with tools to prepare all the needed input files for a standard simulation using MD++ or PROMD, e.g. the generation of the molecular topology, initial coordinates of randomly distributed molecules (solvent) or initial coordinates derived from a pdb file (solute), the solvation of a solute in the solvent and the split up of a simulation in multiple jobs with constant or changing simulation parameters over the job sequence. Furthermore, there are multiple programs to analyse the simulations performed. The following is a list of the most important GROMOS++ programs and the corresponding tasks. A complete list is available via the documentation tool, see Sec. 8-3.1 for more information:

- *com_top* combines multiple topology files into one file.
- *dssp* monitors secondary structure elements of a protein, based on the rules defined by Kabsch and Sander[2].
- *ene_ana* analyses (energy) trajectories.
- *frameout* writes out individual configurations or movies from a molecular trajectory file.
- *hbond* monitors the occurrence of hydrogen bonds.
- *ion* replaces water molecules by ions (to get an overall neutral box).
- *make_top* creates molecular topologies from building block and force-field parameter files.
- *mk_script* generates (multiple) script files to run simulations.
- *noe* analyses NOE distances over a trajectory.
- *pdb2g96* converts coordinate files from pdb to the GROMOS file format.
- *ran_box* creates a condensed phase system of any composition (randomly distributed molecules).
- *sim_box* solvates a solute in a box of pre-equilibrated solvent.
- *tser* calculates time series of properties which may be specified flexibly by the user (distances, angles, dihedral angles, intersection angles with planes, . . .).

As mentioned before, this list is not complete and a lot of more specific analyses can be done using multiple programs in the right order.

Besides all the programs listed above there is a *contrib* collection of programs, a folder containing some GROMOS++ programs which are not of general use but treat a very specific topic or programs which were replaced by newer versions. A list and short explanation of these programs is available via the documentation tool, see Sec. 8-3.1 for more information.

1.3.1. GROMOS++ source code and in-code documentation. The GROMOS++ source code is divided into two major parts, one containing the programs and *contrib* programs, the other one collecting the tools (classes, structures and enumerations) used within the programs. The second part is in turn split up into eight different parts: *gromos*, *gcore*, *gmath*, *gio*, *bound*, *fit*, *args* and *utils* (represented as C++ *namespaces*):

- *gromos* handles the gromos exceptions (error messages).
- *gcore* contains all the classes that store the information about the molecular system, e.g. angles, bonds, atom properties, Lennard-Jones parameters, information and coordinates of the solvent and many more.
- *gmath* contains the tools of the basic vector and matrix algebra, handles time correlation functions and distributions for a series of values as well. There is also a class to handle a kind of pocket calculations read from a string (useful to mathematically interpret a program input parameter defining some specific properties or calculations).
- *gio* contains the tools to read in data or write them out. The read or written data may for example be a topology, coordinates, building block or input parameter files or any kind of trajectory.
- *bound* contains the classes to handle periodic boundary conditions (rectangular, triclinic, truncated octahedron and vacuum).
- *fit* is the namespace that contains code for translational superpositioning and rotational fitting of configurations.
- *args* contains classes to handle the different command line arguments needed by the programs.
- *utils* is the biggest and most manifold namespace. It contains a class which may perform some basic tests on a molecular topology, classes which provide the tools to observe hydrogen bonds or define secondary structure elements within the backbone of a protein using the rules defined by Kabsch and Sander[2], and many other classes. One of the most used classes within this namespace is probably the class AtomSpecifier: it defines and implements a general form to access atoms in a system. It is used to look over a specific set of atoms, possibly spanning different molecules. An AtomSpecifier is basically a string defining one or a group of atoms, used as an input parameter of a program. More detailed information about the exact format is given in the documentation tool (see Sec. 8-3.1).

All the classes, structures and enumerations of the eight namespaces used in GROMOS++ are documented *in-code* and available via the `doxygen` documentation tool. This also contains a description of all programs together with some example input parameters. Interactive links to other classes are automatically generated and help to understand the specific parts and functions of the code.

CHAPTER 2

Error Messages

Error checking is done in GROMOS with respect to three *types of inconsistencies.*

1. The *array sizes* defined in the header files may not be sufficiently large to cope with the *size of the molecular system* (solute, solvent, restraints, etc.) as specified in the input files. This type of inconsistency is signalled by an error message indicating the subroutine producing the error and that the value of an (input) variable is larger than the array size parameter MAX.... to be found in the header files. So, either the former should be reduced or the latter enlarged.

2. The *files* from which data are read by a program may contain data or data types that are *incompatible with* the *expectations of the program.* This type of inconsistency is signalled by an error message as described under Pt. 1.

3. The *control switches* governing the action of a program may be set such that *incompatible options* or *program actions* are selected. This type of inconsistency is signalled by an error message that specifies the incompatible conditions that have been selected.

The philosophy with respect to error checking in GROMOS is that the *user should be allowed to do silly things*, since what is silly in one case, may be useful in another. This means that only inconsistencies of the first type mentioned above are rigorously checked. GROMOS *error messages state the inconsistency*, so what's wrong, *not what's to be done* to remove the inconsistency. It is up to the user to think of and select the appropriate action to avoid the error message.

With respect to the inconsistencies of the types mentioned above, the error message indicates the line of the file where the error occurs and the name of the program or subroutine. This allows the user to identify and analyse the inconsistency in case the printed error message is not sufficiently informative.

CHAPTER 3

Machine Compatibility

The GROMOS programs, class libraries and subroutines have been written in *standard C++*[1] and *standard Fortran 95*[3]. This means that GROMOS should compile and run on any machine for which standard C++ and Fortran 95 compilers are available.

PROMD contains Fortran versions for the *numerical* and *mathematical functions* it needs. These may be but need not be replaced by machine specific versions, e.g. to increase the speed of computation, see Chap. 8-3. Whether PROMD should be used in *32 bit or 64 bit precision* is up to the user, and must be decided upon compilation, see Chap. 8-3. For many types of calculations 32 bit (single) precision is sufficient, especially when considering the intrinsic accuracy of the physical or chemical data used in the calculation and the fact that in statistical mechanics one is interested in ensemble averages of various physical quantities, not in the individual atomic trajectories. On the other hand, 64 bit (double) precision may be a more safe choice when averaging over many atomic configurations.

MD++ and GROMOS++ require a set of libraries to carry out numerical calculations. These libraries are written in the C programming language and can be compiled with the same compilers as MD++ and GROMOS++ themselves, See Chap. 8-2. To maximize operating system and compiler compatibility configuration is carried out by a *GNU Autotools* generated configuration script which generates the *Makefile*s and takes care of correct linking of the libraries. All calculations are carried out in 64 bit (double) precision only. Single precision may be available (by some compilers through their options) but is not recommended.