Enabling Grand-Canonical Monte Carlo: Extending the Flexibility of GROMACS Through the GromPy Python Interface Module

René Pool,[a,b]* Jaap Heringa,[a,b] Martin Hoefling,[c] Roland Schulz,[d] Jeremy C. Smith,[d] and K. Anton Feenstra[a,b]

We report on a python interface to the GROMACS molecular simulation package, GromPy (available at https://github.com/GromPy). This application programming interface (API) uses the ctypes python module that allows function calls to shared libraries, for example, written in C. To the best of our knowledge, this is the first reported interface to the GROMACS library that uses direct library calls. GromPy can be used for extending the current GROMACS simulation and analysis modes. In this work, we demonstrate that the interface enables hybrid Monte-Carlo/molecular dynamics (MD) simulations in the grand-canonical ensemble, a simulation mode that is currently not implemented in GROMACS. For this application, the interplay between GromPy and GROMACS requires only minor modifications of the GROMACS source code, not affecting the operation, efficiency, and performance of the GROMACS applications. We validate the grand-canonical application against MD in the canonical ensemble by comparison of equations of state. The results of the grand-canonical simulations are in complete agreement with MD in the canonical ensemble. The python overhead of the grand-canonical scheme is only minimal. © 2012 Wiley Periodicals, Inc.

DOI: 10.1002/jcc.22947

Introduction

The GROMACS molecular simulation package[1–4] is widely used in the field of (bio)molecular simulation. The most common setup of a simulation system in GROMACS, as in most other major molecular simulation software, assumes a system with a fixed composition of molecules. In addition, for each type of molecule, a fixed, predefined chemical connectivity is assumed. This setup does neither readily allow the breaking of chemical bonds nor readily allow the addition or removal of atoms or molecules to or from the system, for example, simulations in the grand-canonical ensemble. A common workaround implementation involves shell scripting to apply modifications to the simulated system setup, followed by (re)starting of the simulation engine. Needless to say, this adds significant overhead and subtracts from the overall efficiency. Moreover, such approaches show unfavorable scaling behavior with respect to computational efforts, for example, when increasing the amount of insertions/removals or when increasing the system size.

The grand-canonical ensemble can be used for studying systems where one is interested in the average number of molecules as a function of the external chemical potential and temperature. This renders it a suitable ensemble, for example, for exploring adsorption behavior of a given molecular species to the system of interest.[5] In a grand-canonical Monte Carlo (GCMC) simulation, one imposes the chemical potential μ_i of species i, the system volume V, and the temperature T. During simulation, particles of type i are removed or inserted as a result of the imposed chemical potential. At equilibrium, the amount of removals is equal to the amount of insertions, and one can sample the average number of molecules i. The main computational advantage with respect to molecular dynamics (MD) and *NVT* Monte Carlo (MC) is that equilibration times are drastically reduced as well as the sizes needed for the molecular systems.[5] It is also possible to combine MD with GCMC.[5,6] The result is a "hybrid" scheme that alternates short MD trajectories for particle translations of a system containing N_i particles of type i, with trial particle removals ($N_i \leftarrow N_i - 1$) or insertions ($N_i \leftarrow N_i + 1$).

The main application of GROMACS is as an engine to perform MD simulations. Based on such simulations, dynamic system properties of interest can be determined. In addition, the MD trajectories also contain nonequilibrium thermodynamic properties of molecular systems. For analyzing the simulation outcomes,

[a] *R. Pool, J. Heringa, K. A. Feenstra*
Centre for Integrative Bioinformatics Vrije Universiteit (IBIVU), VU University Amsterdam, De Boelelaan 1081a, 1081HV Amsterdam, The Netherlands
E-mail: r.pool@vu.nl

[b] *R. Pool, J. Heringa, K. A. Feenstra*
Netherlands Bioinformatics Centre, Geert Grooteplein 28, 6525GA Nijmegen, The Netherlands

[c] *M. Hoefling*
Department of Theoretical & Computational Biophysics, Max-Planck-Institute for Biophysical Chemistry, Am Fassberg 11, 37077 Goettingen, Germany

[d] *R. Schulz*
University of Tennessee/Oak Ridge National Laboratory, Center for Molecular Biophysics, 1 Bethel Valley, Oak Ridge, Tennessee

Contract grant sponsors: Netherlands Bioinformatics Centre (NBIC) BioRange programme (project number 2.3.1), supported by a BSIK grant through the Netherlands Genomics Initiative (NGI) (RP, JH and KAF); United States department of Energy LAB 08-19 Software Development Tools for Improved Ease-of-Use of Petascale Systems ERKPE92 Scalable Development Environment for Petascale Computing (RS and JCS).

GROMACS comes with a range of applications that facilitate this process. With respect to specific simulation options (e.g., GCMC) or with respect to data analysis, it would however be useful to have the GROMACS data structures accessible to the user via interpreted, high-level programming languages such as python.

In contrast to C or Fortran, python is suitable for rapid prototyping and is easy to read and learn. Moreover, the python user community is active and growing[7] and several python packages such as BioPython[8] and PyCogent[9] have become standards. A python interface would therefore extend the scope of users that can contribute to and use the flexibility of the GROMACS simulation package.

In this work, we describe an approach that makes the GROMACS data structures available to the user via the python module GromPy acting as an application programming interface (API) to the GROMACS C-library. The module allows access to the desired GROMACS data structures in memory from the python interpreter that can then be used to implement analysis tools and new simulation schemes. Here, we illustrate the use of the GromPy API by implementing a GCMC simulation scheme[5] for which we use GROMACS C-library functions to perform energy calculations.

Methods

The GromPy python interface

The GROMACS package is written in the C programming language. We base our development tree on GROMACS version 4.0.5 that will be ported to the latest development branch in the near future.

To implement the interface, we choose the python programming language. Python is a high-level, interpreted, object-oriented, and multiplatform programming language. It provides a large standard library and is easy to code. We use the free and open source CPython implementation of python.[10] Apart from the standard library, python has excellent extensions for numerical data analysis and data display.[11–16] CPython is written in C and compiles python programs into intermediate code that can be executed by a virtual machine. The CPython implementation also allows the implementation of modules in C and the interfacing of (precompiled) libraries.

In our setup, we use the ctypes module[17] as interface between python and the GROMACS C-library. The ctypes module contains python equivalents for all basic C data types and allows the mapping of compound structures in C to python classes. As soon as the GROMACS data structures are accessible via ctypes, we can pass them to external GROMACS functions and access the result from the python interpreter during the execution with the GromPy module.

The initial GromPy implementation can be used for the analysis of trajectories, for example, using GROMACS' periodic boundary condition removal and structure fitting routines.[18] GromPy can also read in index groups and topologies and was applied in the prototyping of GROMACS tools, which were later implemented in C.[19] Recently, GromPy was applied to design a combined MD/MC approach to simulate FRET experiments and aid in the distance reconstruction.[20] This work involves extending GromPy by a GCMC simulation mode. The GromPy source code is publicly available at https://github.com/GromPy.

Hybrid GCMC/MD Simulations

In GCMC, the simulation box is in chemical equilibrium with an external bath. Hence, the chemical potential μ of both systems is equal. One therefore imposes the chemical potential of a particular molecular species upon which molecules are exchanged between the external reservoir and the simulation box.[5] In practice, this means that molecules are inserted into or removed from the simulation box during simulation. The MC acceptance rule for insertion of a molecule reads

$$P_{\mathrm{acc}}(N \to N+1) = \min\left[1, \frac{V}{\Lambda^3(N+1)} \exp\left(\beta[\mu - \Delta U]\right)\right], \quad (1)$$

where N is the number of molecules, V is the box volume, $\Lambda = \sqrt{h^2/(2\pi m k_B T)}$ is the thermal De Broglie wavelength (h denotes Planck's constant, m is the molecular mass, k_B Boltzmann's constant, and T is the temperature), $\beta = 1/(k_B T)$ is the inverse temperature, and $\Delta U = U(N+1) - U(N)$ is the energy difference of adding one molecule at a random position in the simulation box. For removal of a molecule, we use the following acceptance rule

$$P_{\mathrm{acc}}(N \to N-1) = \min\left[1, \frac{\Lambda^3 N}{V} \exp\left(-\beta[\mu + \Delta U]\right)\right], \quad (2)$$

where $\Delta U = U(N-1) - U(N)$ is the potential energy difference associated with the removal of a randomly selected molecule.

To simulate thermal motion, we apply several MD steps at constant *NVT* using the velocity rescale thermostat,[21] which generates a canonical ensemble, in between the GCMC moves. The nature of the MC move (i.e., a trial insertion/removal or an MD move) during a MC cycle is chosen at random based on a user-defined list of probabilities for each type of MC move.

Extending GromPy and modifying the GROMACS source code

This work involves an extension of GromPy, enabling GCMC using the GROMACS C-library. The general setup is shown in Figure 1. When used in GCMC mode, GromPy needs a starting configuration with a number of molecules $N_{i,\mathrm{start}}$ of type i in the form of a GROMACS tpr file stored on disk. Such a tpr file serves as input for a GROMACS simulation and contains all simulation parameters and a configuration of a system. The tpr file range $N_i \in [N_{i,\min}, N_{i,\max}]$ is generated in the preprocessing stage, where $N_{i,\min} \leqslant N_{i,\mathrm{start}}$ and $N_{i,\max} \geqslant N_{i,\mathrm{start}}$ are the extrema of the N_i sampling range. By imposing a chemical potential μ_i of this molecule type, GromPy samples the N_i range via the hybrid GCMC/MD algorithm.

All MC moves in our hybrid MD/GC MC module require having the current state $s_c : [N_{i,c}, \mathbf{r}^{N_{i,c}}, \mathbf{v}^{N_{i,c}}]$ and associated total potential energy U_c in memory, compare Figure 2. This state is a member of the grand-canonical ensemble and thus comprises the current number of molecules $N_{i,c}$ of type i, the coordinates

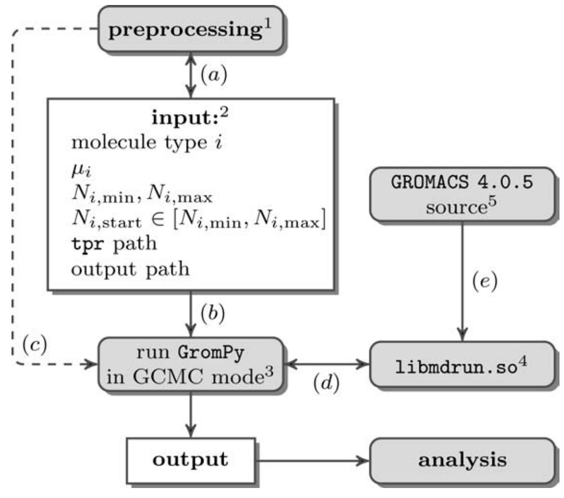

Figure 1. The GCMC simulation setup used in this work. The preprocessing stage (1) involves generating tpr files for each configuration $N_i \in [N_{i,\min}, N_{i,\max}]$ using grompp. The input (2) for GromPy comprises the molecule type i for which the chemical potential μ_i is imposed, a range of numbers of molecules $[N_{i,\min}, N_{i,\max}]$ of type i that can be sampled, a starting configuration $N_{i,\text{start}}$, the tpr path on disk, and the output path on disk. The preprocessing step requires prior knowledge of the input parameters (a). GromPy (3) reads the input parameters (b). Molecular insertions/removals requires tpr reads from disk (c). Once read, the associated data structures are kept in memory. The necessary energy evaluations are performed by the GROMACS library (4) with which GromPy communicates (d). This shared object library is compiled (e) from a slightly modified version (5) of GROMACS 4.0.5. The generated output can be further analyzed by the native GROMACS analysis suite, GromPy, or other software.

$\mathbf{r}^{N_{i,c}}$, and the velocities $\mathbf{v}^{N_{i,c}}$ (we use the $\mathbf{r}^N$ and $\mathbf{v}^N$ short hand notation for the coordinate and velocity arrays consisting of N elements). The GCMC module uses two MC move types: one that performs several MD steps on s_c to simulate thermal motion of the molecular system and one that performs a GCMC move that tries to modify s_c by inserting or removing a molecule. For computational efficiency, the MD move is always accepted as the resulting configuration is already part of the correct statistical mechanical ensemble. After the MD move, we update the coordinates, velocities, and total potential energy. Inside the GCMC move, we select either the removal or the insertion of a molecule with a probability of $P_{\text{insert}} = P_{\text{remove}} = \frac{1}{2}$. For insertion, we generate a trial state s_t that has $N_{i,t} = N_{i,c} + 1$ molecules. The first $N_{i,c}$ elements of the coordinate and velocity arrays are copied from s_c. The last element is filled by a random molecular position $\mathbf{r}'$ inside the box and by a molecular velocity $\mathbf{v}'$ chosen at random from the Maxwell–Boltzmann velocity distribution associated with the imposed temperature T, respectively. This step requires having s_t in memory. If this is not the case, we first read a tpr file with $N_i = N_{i,t}$ from disk. A molecular removal involves generating a trial state s_t that has $N_{i,t} = N_{i,c} - 1$ molecules. We randomly select a molecule (k) from the list and copy the $N_{i,c}$ elements of the coordinate and velocity arrays from s_c to s_t, while excluding the kth element. Again, we require having s_t in memory and read from disk otherwise. Trial insertions or removals with associated U_t are accepted according to eq. (1) or eq. (2) (where $\Delta U = U_t - U_c$),

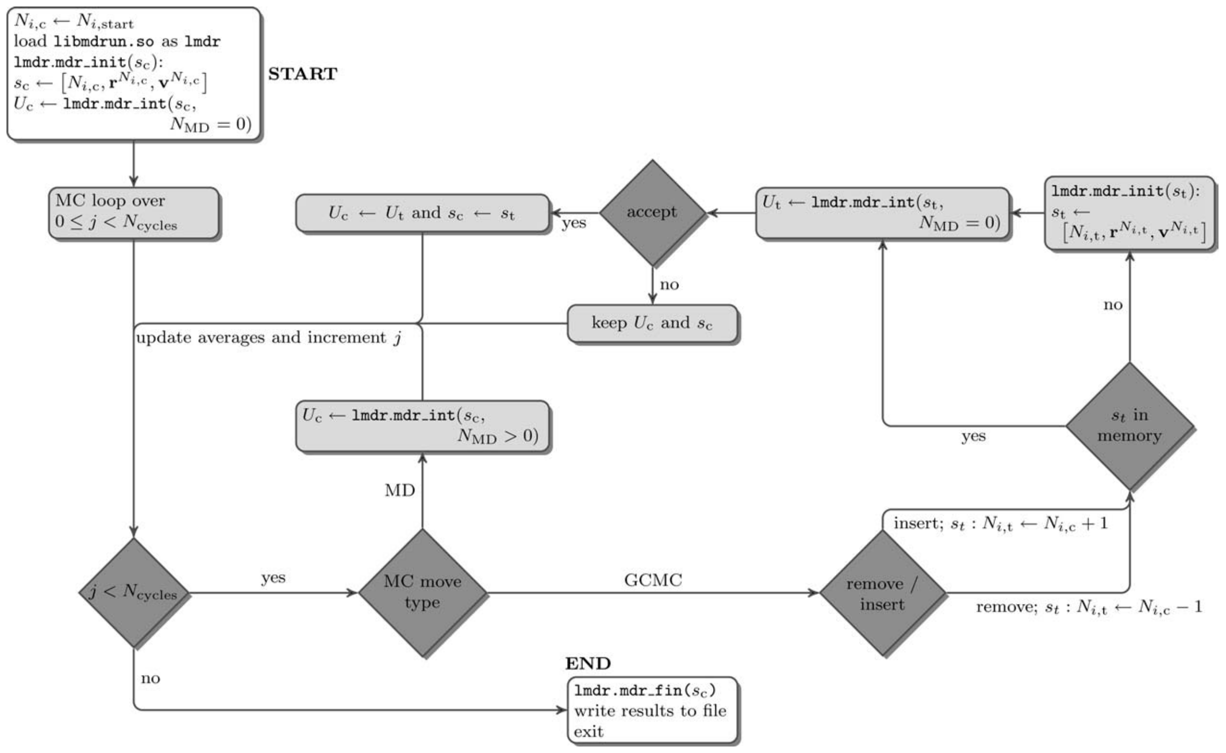
$N_{i,c} \leftarrow N_{i,\text{start}}$
load libmdrun.so as lmdr
lmdr.mdr_init(s_c):
$s_c \leftarrow [N_{i,c}, \mathbf{r}^{N_{i,c}}, \mathbf{v}^{N_{i,c}}]$
$U_c \leftarrow$ lmdr.mdr_int(s_c, $N_{MD} = 0$)
START
MC loop over $0 \le j < N_{\text{cycles}}$
$U_c \leftarrow U_t$ and $s_c \leftarrow s_t$
yes
accept
$U_t \leftarrow$ lmdr.mdr_int(s_t, $N_{MD} = 0$)
lmdr.mdr_init(s_t): $s_t \leftarrow [N_{i,t}, \mathbf{r}^{N_{i,t}}, \mathbf{v}^{N_{i,t}}]$
no
keep U_c and s_c
update averages and increment j
no
$U_c \leftarrow$ lmdr.mdr_int(s_c, $N_{MD} > 0$)
yes
s_t in memory
MD
insert; $s_t : N_{i,t} \leftarrow N_{i,c} + 1$
$j < N_{\text{cycles}}$
yes
MC move type
GCMC
remove / insert
remove; $s_t : N_{i,t} \leftarrow N_{i,c} - 1$
END
no
lmdr.mdr_fin(s_c)
write results to file
exit

Figure 2. Flowchart of GromPy in GCMC mode. Each MC move is based on the current state $s_c : [N_{i,c}, \mathbf{r}^{N_{i,c}}, \mathbf{v}^{N_{i,c}}]$ and associated total potential energy U_c, kept in memory. State s_c is defined by the number of molecules $N_{i,c}$ of type i, their coordinates $\mathbf{r}^{N_{i,c}}$ and velocities $\mathbf{v}^{N_{i,c}}$. GromPy uses two MC move types: an MD move of several MD steps and a GC MC move. After the MD move, the coordinates, velocities, and total potential energy are updated. The GCMC move involves removal or insertion of a molecule selected with a probability $P_{\text{insert}} = P_{\text{remove}} = \frac{1}{2}$. Insertion or removal requires having s_t in memory. If this is not the case, we first read a tpr file with $N_i = N_{i,t}$ from disk. Insertions or removals are accepted with the probabilities in eq. (1) and eq. (2), respectively. If accepted, s_t becomes s_c and U_t becomes U_c. Otherwise, we keep s_c and U_c. After each MC move we update the averages and increment the MC loop iterator j.

respectively. If accepted, we update s_t to s_c and the associated potential energy U_t becomes U_c. Otherwise, we keep s_c and U_c. After each MC move, we update the averages and increment the MC loop iterator.

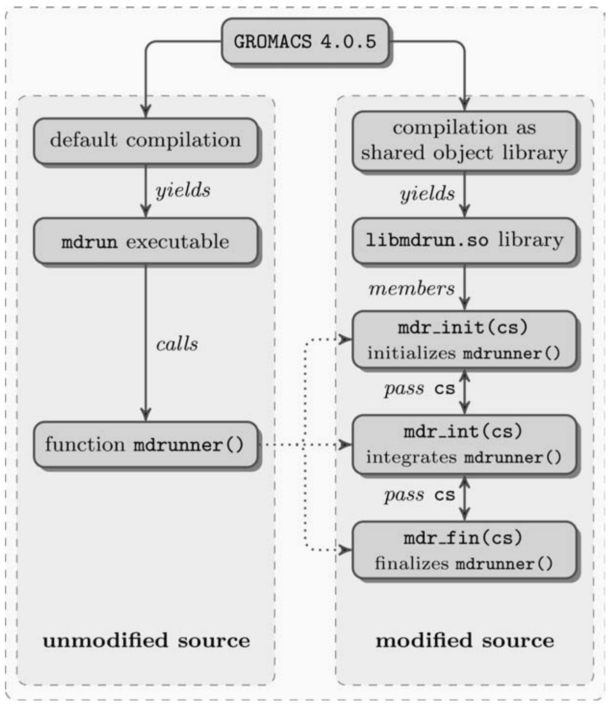

Figure 3. Modification of the source code of GROMACS version 4.0.5. Left: default compilation yields the mdrun executable (among others). This program calls function mdrunner() that is the calculation engine for MD simulations. Right: compilation of the mdrun executable as the shared object library libmdrun.so and splitting up function mdrunner() into an initialization stage, an integration stage, and a finalization stage. Communication between the stages is achieved through the new cs data structure. Library libmdrun.so is loaded into the GromPy module where mdr_init(cs), mdr_int(cs) and mdr_fin() are called for performing MD moves and GCMC trial moves by manipulating cs before each MC move.

As described above, the GCMC module uses the current and trial states (s_c and s_t) to sample the grand-canonical ensemble. For this, energy evaluations are needed to obtain U_c and U_t that serve as input for the acceptance rules for insertion [eq. (1)] and removal [eq. (2)]. At run time, the states are stored in memory by interfacing with specific GROMACS library functions. The associated energies U_c and U_t are computed by calls to the GROMACS library. Both operations are performed using the python ctypes module. To achieve the interfacing, we modified the GROMACS 4.0.5 source code as shown in Figure 3. Although the modifications were performed for the serial implementation of GROMACS, we intend to make the modifications compatible with the parallel parts of the code. We expect that this will require relatively little effort. The GROMACS function mdrunner() loads a tpr file and can perform an MD simulation on a given system. This function is called by the GROMACS mdrun executable. As ctypes can load only shared object libraries, we compile the mdrun executable as a shared object library: libmdrun.so. During a GCMC run, we generate trial states s_t by copying the current state s_c to s_t and adding a trial position (and velocity) for insertion or excluding a randomly selected molecule for removal. To achieve this flexibility, we have split up the mdrunner() function into three parts: mdr_init(cs), mdr_int(cs), and mdr_fin(cs). We added a new data structure cs for the current state that enables communication between the subfunctions. For our purposes, the most important member of cs is the state s. By subsequently calling the three separate functions (and without modifying cs in between), the behavior of the original mdrunner() function is reproduced exactly. Function mdr_init(cs) reads a tpr file from disk and stores the state s in cs. Function mdr_int(cs) performs an MD calculation of N_{MD} steps. N_{MD} is also a member of cs and can be set from within GromPy. For an MD move, the number of MD steps is set to $N_{MD} > 0$ and for energy evaluations in a GCMC move it is set to $N_{MD} = 0$ (which results in a single point energy calculation). Computational performance of the simulation is calculated by function mdr_fin(cs). The gain in total computational time is realized by keeping cs in python memory once initialized by a disk read. In this way, cs can be (re)used efficiently for MD or GCMC moves.

Note that the $N_{i,\mathrm{start}}$ configuration should be an equilibrated one. However, this is not a precondition for all other $N_i \neq N_{i,\mathrm{start}}$ tpr files that the user wishes to use for sampling, as this tpr file is merely used to fill the coordinate and velocity arrays in a trial move. During simulation, s_c will always be part of the correct ensemble.

To summarize, once in memory cs can be manipulated for whatever intended purpose and can serve as input for mdr_int(cs). Our purpose is GCMC and we therefore need to manipulate the cs members s and N_{MD}. Obviously, the same behavior can be achieved by executing a shell script that calls the necessary GROMACS executables, that is, grompp and mdrun. The downside of such an approach is that most of the time the GCMC shell will perform file I/O and/or system calls, mainly invoked by the necessary consecutive execution of the GROMACS grompp and mdrun applications. Having the relevant GROMACS data structures in memory, combined with the modified GROMACS source code drastically reduces the time spent on file I/O and renders GromPy an efficient GCMC application, with less than 6% of run time spent in overhead. This overhead involves logging to disk, reading of tpr files from disk, iterating over the MC loop, replacing the $\mathbf{r}^N$ and $\mathbf{v}^N$ arrays for trial insertions/removals, and associated evaluations of eqs. (1) and (2).

Validation of the GCMC module

We aim to validate the GROMACS-GCMC scheme by comparing equations of state (EOS) determined by GCMC and *NVT* MD. For this, it is necessary to simulate a single phase. We therefore choose to simulate supercritical fluids. The validation is performed for two model systems. The first system consists of single Lennard-Jones (LJ) particles of the same type. For this, we use water particles of the MARTINI coarse-grained force field[22] that are modeled as single LJ particles. For this system type, we approximate the critical properties by Gibbs ensemble simulation results.[5] For the second system, with polar SPC water,[23] we also need to account for charges and insertions/removals of multi-atomic molecules, rendering it a more complicated and challenging test case. The critical properties for the SPC model are taken from the literature[24]: $T_{c,\mathrm{SPC}} = 587\,\mathrm{K}$ and $\rho_{c,\mathrm{SPC}} = 15\,\mathrm{mol/l}$.

In the LJ simulations, we use a shift potential for the nonbonded interactions with a switch radius of $r_s = 0.9\,\mathrm{nm}$. The nonbonded interactions were truncated at $r_c = 1.2\,\mathrm{nm}$.[22] In the SPC simulations, all nonbonded interactions were calculated up to a cut-off distance of $r_c = 0.9\,\mathrm{nm}$ (corrections to the total energy and pressure due to truncation are taken into account) and the Coulombic interactions are calculated by the particle mesh Ewald method[25] with a spacing of the Fourier grid of 0.12 nm.

The *NVT* EOS for both systems are determined at $T = 773\,\mathrm{K}$ and $T = 900\,\mathrm{K}$. The simulation parameters are summarized in Table 1. For each density ρ, we perform a separate simulation of which the ranges are the *x*-values in Figures 4a and 4b for the LJ and SPC models, respectively. These density ranges are obtained by changing the box volume, while keeping the amount of molecules constant. A pilot experiment showed that *NVT* results are consistent when varying the box volume *V* at constant *N* or varying *N* at constant *V*. We average the total pressure *p* and hence obtain a pressure profile as a function of density ρ.

Table 1. MD parameters used in this work for the LJ and SPC models.

	Model	
	LJ	SPC
$N_{\mathrm{molecules}}$	400	500
Δt (ps)	2×10^{-2}	2×10^{-3}
t_{NVT} (ps)	2×10^{3}	2×10^{3}
$t_{\mathrm{MD},\mu VT}$ (ps)	2	0.2
$t_{\mathrm{GCMC},\mu VT}$ (ps)	0	0
$\tau^{-1}_{\mathrm{thermostat}}$ (ps^{-1})	0.1	0.1

The MD time step is denoted by Δt, the total simulation time for each *NVT* simulation by t_{NVT}, the simulation time per MD move in each μVT simulation by $t_{\mathrm{MD},\mu VT}$, and the "simulation time" for a single point energy calculation needed for a GCMC trial move by $t_{\mathrm{GCMC},\mu VT}$. We apply the velocity rescale thermostat[21] that ensures a canonical ensemble. The associated coupling frequency is represented by $\tau^{-1}_{\mathrm{thermostat}}$.

The μVT EOSs at $T = 773\,\mathrm{K}$ and $T = 900\,\mathrm{K}$ are obtained by imposing a range of chemical potentials μ to fixed volume systems of either LJ particles or SPC water molecules. The simulation parameters of the μVT simulations can be found in Table 2. The MD parameters used in MD moves are listed in Table 1. For the density ranges studied, compare the *x*-values

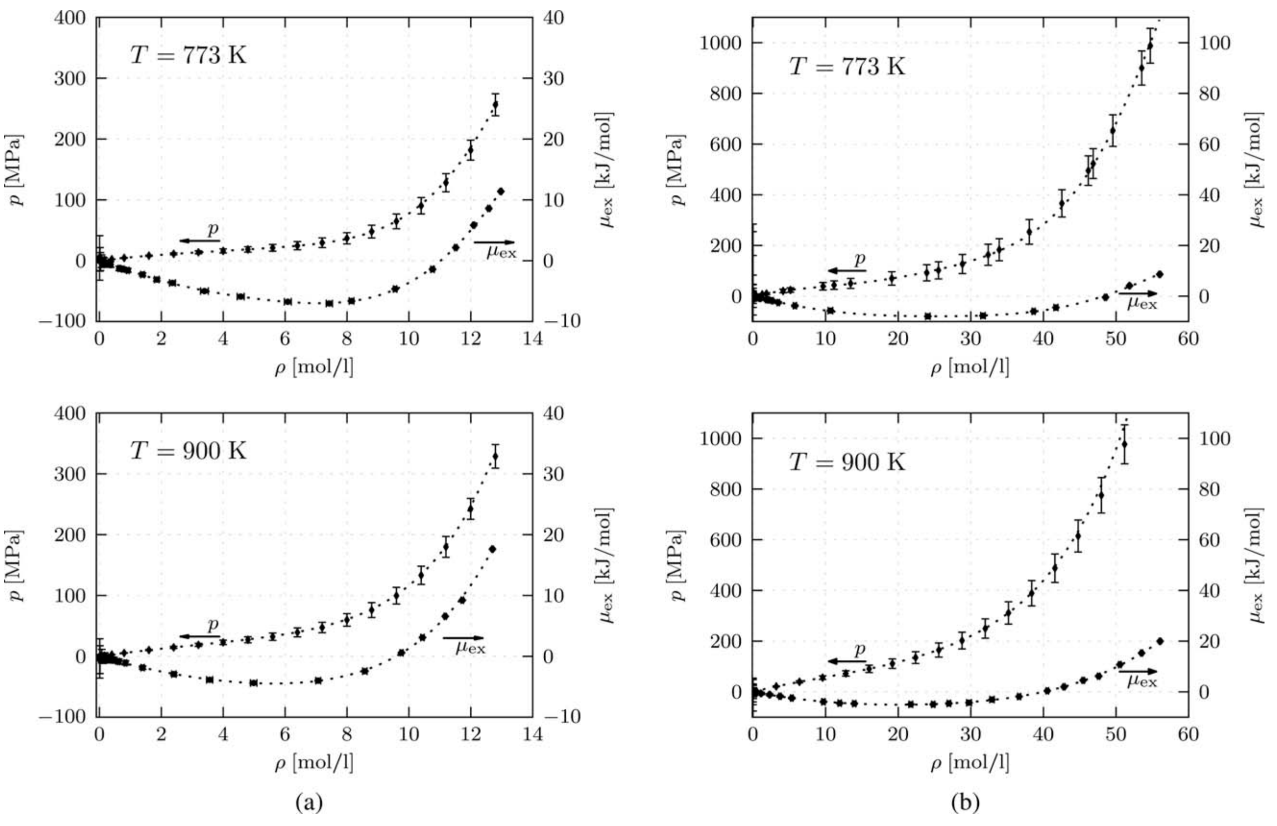

Figure 4. EOS for the LJ model (a) and the SPC water model (b) at $T = 773\,\mathrm{K}$ (top) and $T = 900\,\mathrm{K}$ (bottom). The data points on the $p(\rho)$ line are determined by MD at *NVT* (the error bars indicate the standard deviation of the pressure fluctuations). The points on the $\mu_{\mathrm{ex}}(\rho)$ line are determined by grand-canonical MC using GromPy in GCMC mode. The standard deviations of μ_{ex} and of ρ for the μVT data are shown as error bars (at 1σ) and were calculated by conventional error propagation rules. The least-squares fit of a sixth degree polynomial to these μ_{ex} points was transformed into a $p(\rho)$ curve using eq. (4). Both results are shown as dotted lines in each plot.

Table 2. GCMC parameters used in this work for the LJ and SPC models.

	Model	
	LJ	SPC
b (nm)	3.64	2.70
N_{cycles}	2500	2500
N_{moves}	42	42
P_{MD}	0.05	0.05
P_{GCMC}	0.95	0.95

The length of the cubic simulation box is denoted by b and the number of MC cycles is denoted by N_{cycles}. Each MC cycle consists of N_{moves} trial MC moves where the MC move type is chosen randomly with probabilities P_{MD} and P_{GCMC} for an MD move and GCMC move, respectively.

Table 3. Insertion/removal acceptance probabilities $P_{\text{acc,GCMC}}$ at various densities for the LJ and SPC models.

Model	$\langle\rho\rangle$ (mol/l)	$P_{\text{acc,GCMC}}$
LJ	0.22	0.81
LJ	7.4	0.10
LJ	13.0	0.002
SPC	0.32	0.76
SPC	10.7	0.32
SPC	56.0	0.004

Note that at equilibrium $P_{\text{acc}}(N \to N+1) = P_{\text{acc}}(N \to N-1) = P_{\text{acc,GCMC}}$.

in Figures 4a and 4b for the LJ and SPC models, respectively. For this type of simulation, we obtain a density profile as a function of μ.

The Gibbs–Duhem equation is used to validate the μVT results

$$\rho\left[\frac{\partial\mu}{\partial\rho}\right]_T = \left[\frac{\partial p}{\partial\rho}\right]_T, \tag{3}$$

from which the pressure profile

$$p(\rho) = \rho k_B T + \int_{\rho'=0}^{\rho'=\rho} d\rho' \left(\rho' \left[\frac{\partial\mu_{\text{ex}}}{\partial\rho'}\right]_T\right) \tag{4}$$

is derived. The excess part of the chemical potential μ_{ex} is calculated as

$$\begin{aligned}\mu_{\text{ex}} &= \mu - \mu_{\text{id}}\\ &= \mu - k_B T \ln(\Lambda^3\rho),\end{aligned} \tag{5}$$

where μ_{id} is the ideal part of the chemical potential. The pressure as a function of density $p(\rho)$ in eq. (4) is determined from a numerical least-squares fit of a sixth degree polynomial to the μVT data of $N_{\text{dat}} = 1000$ data points.

Results and Discussion

A supercritical LJ system

For the supercritical LJ system, we used a system of single particle MARTINI[22] water (W) molecules. A system consisting of just this molecule type, involves nonbonded LJ interactions only and therefore renders it a relatively simple test system. We calculated the critical temperature of this system as $T_{c,\text{W}} = 647.2\,\text{K}$ and its associated critical density as $\rho_{c,\text{W}} = 4.99\,\text{mol/l}$ by Gibbs ensemble simulations.[5]

The *NVT* results are shown in Figure 4a (left and bottom axes). The GCMC insertion/removal acceptance probabilities are listed in Table 3. We examined if the *NVT* EOS is different when varying the number of molecules compared with varying the simulation box volume. This was found not to be the case. The results of the LJ μVT simulations are shown in Figure 4a (right and bottom axes). The *NVT* and μVT EOS are completely equivalent.

A supercritical SPC water system

Apart from nonbonded LJ interactions between the water oxygen atoms, the SPC model[23] involves Coulomb interactions between the partially charged hydrogen and oxygen atoms. The relative orientation of the hydrogen and oxygen atoms within a water molecule is assumed constant, that is, bond stretching and bond bending is constrained during simulation using the SETTLE algorithm.[26]

In Figure 4b (left and bottom axes), we show the *NVT* results. The GCMC insertion/removal acceptance probabilities are listed in Table 3. We again validated the *NVT* EOS when varying the number of molecules compared against the *NVT* EOS when varying the simulation box volume. The results of the μVT simulations are shown in Figure 4b (right and bottom axes).

As we can see from Figure 4b, the μVT data at $T = 773\,\text{K}$ are in excellent agreement with the *NVT* results. The μVT data at $T = 900\,\text{K}$ also agrees with the *NVT* data. Although still within the *NVT* error bars, the μVT $p(\rho)$ profile at the highest particle densities slightly overestimates the *NVT* one. This could be explained by GCMC sampling difficulties at extreme simulation conditions. SPC molecule insertions are performed by generating a random position in the simulation box for the oxygen atom, followed by randomly orienting the hydrogens while meeting the bond angular and bond length constraints. A more efficient sampling at higher densities could be achieved by applying the configurational bias MC (CBMC) technique.[5] In CBMC, one selects the most favorable insertion configuration from a set of trial configurations and appropriately corrects for this bias. It should be kept in mind that for both temperatures, the conditions at higher densities can be considered extreme, for example, pressures of over 5×10^8 Pa.

Computational performance and accuracy

To get an impression of the computational performance of GromPy in GCMC mode, we again determined the EOS for the LJ system at $T = 773\,\text{K}$. For the GCMC case, simulation parameters are the same as above. For each data point in Figure 5, the number of particles in the *NVT* simulation was taken the same as the average number of particles, calculated from the μVT simulation series. In this way, a fair comparison can be made between the two simulation modes. Per μVT or *NVT* simulation,

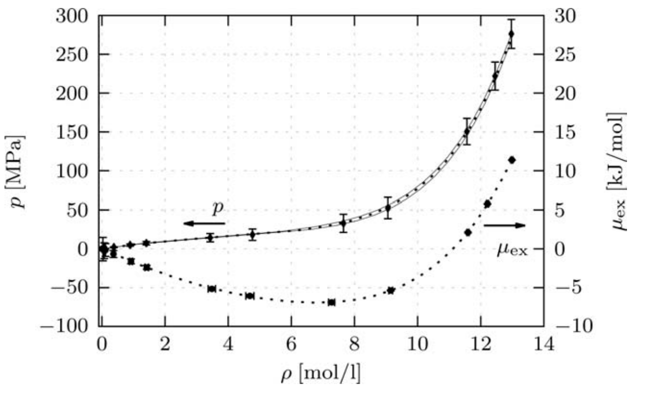

Figure 5. EOS for the LJ model at $T = 773$ K. The data points on the $p(\rho)$ line are determined by MD at *NVT*. The points on the $\mu_{ex}(\rho)$ line are determined by GC MC using GromPy in GCMC mode. The standard deviations of μ_{ex} and of ρ for the μVT data were calculated by conventional error propagation rules. The least-squares fit of a sixth degree polynomial to these μ_{ex} points was transformed into a $p(\rho)$ curve using eq. (4). Both results are shown as dotted lines. Note that the data points for both simulation types are obtained by an equal number of integration steps. The error bandwidth in the pressure profile based on the uncertainty in the μVT data is the area between the thin full lines.

we used a total of 749,700 integration steps of which the first 16.7% was used for equilibration.

Both EOSs were determined on a 32-bit Linux machine with the applications running on a single CPU. The total simulation time for the *NVT* EOS is 2400 s (8 s spent on system calls and 2392 s spent on "real" CPU time). The total simulation time for the μVT EOS is 2547 s (149 s spent on system calls and 2397 s spent on real CPU time). The ~150 s difference between the two simulation modes comes from the limited amount of time spent on system calls and can be considered as "python overhead" as described in Extending GromPy and modifying the GROMACS source code section. Note that this also involves the evaluations of eqs. (1) and (2) in python.

The μVT and *NVT* EOSs are completely equivalent, compare Figure 5. The uncertainty bandwidth in the pressure profile based on the standard deviations in the μVT data is the area between the thin solid lines in Figure 5. The μVT EOS uncertainty is well within the error bars of the pressure sampled in the *NVT* ensemble.

To illustrate the file I/O overhead problem in a shell approach that does not use direct calls to the GROMACS library, we implemented such a shell that can also sample the grand-canonical ensemble but uses the GROMACS executables to perform the necessary MC moves. We simulated the LJ system at $T = 900$ K at a chemical potential yielding an average number of $\langle N \rangle \approx 377$ ($\langle \rho \rangle \approx 13$ mol/l) using both GCMC approaches. For both simulations, the parameters are listed in Table 2 (and Table 1 for the parameters of the MD moves). The "shell" GCMC module requires 10,800 s for 2500 MC cycles, whereas GromPy in GCMC mode does the same 14 times faster (771 s). It thus turns out that the "shell" approach spends over 90% of the computation time on system calls and disk operations at this particular chemical potential. For this LJ system, we found that the computation time t_{CPU} scales as $t_{CPU} \propto N^{1.48}$. Now, assuming that the file I/O overhead remains constant, the shell approach would be 1.01 times slower if we would study systems of size $\langle N \rangle \approx 11,000$ and it would be two times slower for system sizes of $\langle N \rangle \approx 2700$. We tested this assumption and found that for a system of size $\langle N \rangle \approx 11,850$, again $\langle \rho \rangle \approx 13$ mol/l, the shell implementation is 39 times slower compared with GromPy/GCMC. Moreover, t_{CPU} for GromPy/GCMC scales with N via the above relation, whereas the shell approach shows a more unfavorable scaling behavior of t_{CPU} with N. Clearly, grompp and mdrun contribute unfavorably to the scaling behavior of t_{CPU} with system size and render the shell GCMC implementation unfeasible for large system sizes, due to the large number of grompp and mdrun executions (and associated file I/O) needed. GromPy/GCMC does not suffer from this effect as the necessary data structures are kept in memory at run-time.

Conclusions

We have successfully implemented and extended the GromPy module (available at https://github.com/GromPy) and enabled simulations in grand-canonical ensemble using the GROMACS C-library. To this end, only minor modifications to the GROMACS source code needed to be applied, and these do not in any way affect the operation, efficiency, and/or performance of the GROMACS applications built with the GROMACS source. To the best of our knowledge, GromPy is the first reported interface to the GROMACS library and MD engine that uses direct library calls. It can be used for further extending the current GROMACS simulation and analysis modes.

We validated our grand-canonical scheme for two system types. For the simplest one that involves only LJ interactions, the μVT results are in complete agreement with those of *NVT* MD simulations performed with GROMACS. For a second, more complicated, system that also involves Coulombic interactions and insertions of multi-atomic molecules, the μVT results agree completely with the *NVT* results at $T = 773$ K, but seem to slightly overestimate the high density region of the *NVT* EOS at $T = 900$ K (although still within the *NVT* error). This deviation is explained by sampling difficulties at this high temperature and density. Sampling efficiency might be enhanced by implementing CBMC for multiatomic molecules.

The computational performance of GromPy in GCMC mode is comparable to the GROMACS mdrun executable. The accuracy of the μVT data is well within that of conventional MD in the *NVT* ensemble.

Our work is compatible with the 4.0.7 version of GROMACS, and only minor modifications are needed for the 4.5 version and higher versions. For the near future, we plan to merge the necessary changes on the code to the main development tree, which will make our GCMC compatible with the latest GROMACS releases, of course in consultation with the GROMACS developers community. In addition, our minor modifications to the serial implementation of the GROMACS source code should be made compatible with the parallel implementation. We expect that GCMC and hybrid MD/MC is of interest to the GROMACS users community. Our modifications to the source code are only minor and do not stand in the way of "normal" use of the MD engine. Additionally, a python interface to GROMACS will contribute significantly to the flexibility of the package.

Acknowledgments

This work is dedicated to Wilfred van Gunsteren for his pioneering work in biomolecular simulations and being a personal inspiration to many in the field.

RP, JH and KAF would like to thank Mohammed El-Kebir and Nicola Bonzanni of the IBIVU group for their contributions to modifying the GROMACS source code and Sanne Abeln for scientific ideas and discussions.

This work was part of the BioRange programme (project number: 2.3.1) of the Netherlands Bioinformatics Centre (NBIC), which is supported by a BSIK grant through the Netherlands Genomics Initiative (NGI).

Keywords: grand-canonical Monte Carlo • GROMACS • python • API • trajectory analysis

How to cite this article: R. Pool, J. Heringa, M. Hoefling, R. Schulz, J. C. Smith, and K. A. Feenstra, *J. Comput. Chem.* **2012**, *33*, 1207–1214. DOI: 10.1002/jcc.22947

[1] H. J. C. Berendsen, D. van der Spoel, R. Van Drunen, *Comput. Phys. Commun.* **1995**, *91*, 43.
[2] E. Lindahl, B. Hess, D. van der Spoel, *J. Mol. Mod.* **2001**, *7*, 306.
[3] D. van der Spoel, E. Lindahl, B. Hess, G. Groenhof, A. E. Mark, H. Berendsen, *J. Comput. Chem.* **2005**, *26*, 1701.
[4] B. Hess, C. Kutzner, D. van der Spoel, E. Lindahl, *J. Chem. Theory Comput.* **2008**, *4*, 435.
[5] D. Frenkel, B. Smit, Understanding Molecular Simulation: From Algorithms to Applications; 2nd ed., Academic Press: San Diego, **2002**.
[6] R. Pool, P. G. Bolhuis, *J. Phys. Chem. B* **2005**, *109*, 6650.
[7] F. Pérez, B. E. Granger, J. D. Hunter, *Comput. Sci. Eng.* **2011**, *13*, 13.
[8] P. J. A. Cock, T. Antao, J. T. Chang, B. A. Chapman, C. J. Cox, A. Dalke, I. Friedberg, T. Hamelryck, F. Kauff, B. Wilczynski, M. J. L. de Hoon, *Bioinformatics* **2009**, *25*, 1422.
[9] R. Knight, P. Maxwell, A. Birmingham, J. Carnes, J. G. Caporaso, B. C. Easton, M. Eaton, M. Hamady, H. Lindsay, Z. Liu, C. Lozupone, D. McDonald, M. Robeson, R. Sammut, S. Smit, M. J. Wakefield, J. Widmann, S. Wikman, S. Wilson, H. Ying, G. A. Huttley, *Genome Biol.* **2007**, *8*, R171.
[10] The Python Software Foundation, Python Programming Language — Official Website, http://www.python.org/.
[11] P. F. Dubois, K. Hinsen, J. Hugunin, *Comput. Phys.* **1996**, *10*, 262.
[12] D. Ascher, P. F. Dubois, K. Hinsen, J. Hugunin, T. Oliphant, *Numer. Python tech.* report UCRL-MA-128569, Lawrence Livermore National Laboratory, **2001**.
[13] E. Jones, T. Oliphant, P. Peterson, SciPy: Open Source Scientific Tools for Python, Available at http://www.scipy.org, **2001**.
[14] T. E. Oliphant, Guide to NumPy, Trelgol Publishing: USA, **2006**.
[15] T. E. Oliphant, *Comput. Sci. Eng.* **2007**, *9*, 10.
[16] J. D. Hunter, *Comput. Sci. Eng.* **2007**, *9*, 90.
[17] T. Heller, The ctypes package, Available at http://pypi.python.org/pypi/ctypes (Accessed: July 1, 2011).
[18] M. Hoefling, K. E. Gottschalk, *J. Struct. Biol.* **2010**, *171*, 52.
[19] M. G. Wolf, M. Hoefling, C. Aponte-Santamaría, H. Grubmüller, G. Groenhof, *J. Comput. Chem.* **2010**, *31*, 2169.
[20] M. Hoefling, N. Lima, D. Haenni, C. A. M. Seidel, B. Schuler, H. Grubmüller, *PloS One* **2011**, *6*, e19791.
[21] G. Bussi, D. Donadio, M. Parrinello, *J. Chem. Phys.* **2007**, *126*, 014101.
[22] S. J. Marrink, H. J. Risselada, S. Yefimov, D. P. Tieleman, A. H. de Vries, *J. Phys. Chem. B* **2007**, *111*, 7812.
[23] H. J. C. Berendsen, J. P. M. Postma, W. F. Van Gunsteren, J. Hermans, In Intermolecular Forces; B. Pullman, Ed., Vol. 11; D. Reidel Publishing Company: Dordrecht, the Netherlands, **1981**; pp. 313-–338.
[24] J. J. de Pablo, J. M. Prausnitz, H. J. Strauch, P. T. Cummings, *J. Chem. Phys.* **1990**, *93*, 7355.
[25] U. Essmann, L. Perera, M. L. Berkowitz, *J. Chem. Phys.* **1995**, *103*, 8577.
[26] S. Miyamoto, P. A. Kollman, *J. Comput. Chem.* **1992**, *13*, 952.

Received: 12 July 2011
Revised: 21 December 2011
Accepted: 9 January 2012
Published online on 28 February 2012