9.33 Ab initio Molecular Dynamics Simulations

Recently, it was decided to include an *ab initio* molecular dynamics (AIMD) module in ORCA.[1] As a plethora of different electron structure methods with analytical gradients is already implemented, all these methods are now available also for MD simulations, offering a wide range of accuracy/performance trade-offs.

This MD module has a relatively short history inside of the ORCA package, and some features found in other MD codes are still missing. In future releases, many new features and methods will (hopefully) be added to this part of the program. However, we will do our best to keep a strict backward compatibility, such that the sample inputs from this section will remain valid in all future releases.

For some more information as well as input examples for the ORCA MD module, please visit

`https://brehm-research.de/orcamd`

9.33.1 Recent Changes

- Molecular dynamics simulation can now use Cartesian, distance, angle, and dihedral angle constraints. These are managed with the `constraint` command; see below.
- The MD module now features cells of several geometries (cube, orthorhombic, parallelepiped, sphere, ellipsoid), which can help to keep the system inside of a well-defined volume. The cells have repulsive harmonic walls.
- The cells can be defined as elastic, such that their size adapts to the system. This enables to run simulations under constant pressure.
- Trajectories can now be written in XYZ and PDB file format.
- A restart file is written in each simulation step. With this file, simulations can be restarted to seamlessly continue (useful for batch runs or if the job crashed). Restart is handled via the `restart` command; see below.
- Introduced regions (*i. e.*, subsets of atoms), which can be individually defined. Regions can be used to thermostat different parts of the system to different temperatures (*e. g.*, cold solute in hot solvent), or to write subset trajectories of selected atoms.
- The energy drift of the simulation is now displayed in every step (in units of Kelvin per atom). Large energy drift can be caused by poor SCF convergence, or by a time step length chosen too large.
- Physical units in the MD input are now connected to their numeric values via underscore, such as `350_pm`. A whitespace between value and unit is no longer acceptable. This slightly breaks backward compatibility – sorry.

[1] Strictly speaking, these simulations are Born–Oppenheimer molecular dynamics simulations (BOMD), because they approximately solve the time-independent Schrödinger equation to compute gradients and then move the atoms according to these gradients.

- Fixed a bug in the time integration of the equations of motion, which compromised energy conservation.
- Fixed crashes for semiempirics and if ECPs were employed. You can now run MD simulations with methods such as PM3 and with ECPs.

9.33.2 Input Format

The molecular dynamics module is activated by specifying "MD" in the simple input line. The actual MD input which describes the simulation follows in the "%md" section at some later position in the input file. The contents of this section will subsequently be referred to as "MD input".

```
! MD BLYP D3 def2-SVP
%md
  timestep 0.5_fs # This is a comment
  initvel 350_K
  thermostat berendsen 350_K timecon 10.0_fs
  dump position stride 1 filename "trajectory.xyz"
  run 200
end
* xyz 0 1
O         -4.54021        0.78439         0.09307
H         -3.64059        0.38224        -0.01432
H         -4.63463        1.39665        -0.67880
*
```

Please note that the MD input is not processed by ORCA's main parser, but by a dedicated parser in the MD module. Therefore, the MD input is not required to obey the general ORCA syntax rules. The syntax will be described in the following.

In contrast to general ORCA input, the MD input is not based on keywords, but on *commands*, which are executed consecutively on a line-by-line basis starting at the top (like, *e. g.*, in a shell script). This means that identical commands with different arguments may be given, coming into effect when the interpreter reaches the corresponding line. This enables to perform multiple simulations (*e. g.*, pre-equilibration and production run) within a single input file:

```
%md
  timestep 1.0_fs
  run 200
  timestep 2.0_fs
  run 500
end
```

Work is already under way to add variable definitions, loops, and conditional branching to the MD input.[2] This will enable even larger flexibility (*e. g.*, to run a simulation until a certain quantity has converged). The MD input is written in the SANscript language ("Scientific Algorithm Notation Script"), which is under development. A first glimpse can be found at

`https://brehm-research.de/sanscript`

As in standard ORCA input, **comments** in the MD input are initiated by a "#" sign and span to the end of the current line. Commands can be started both at the beginning of a line and after a command. The only place where a "#" is **not** treated as start of a comment is inside of a string literal (*e. g.*, in file names).

```
%md
  # Comment
  timestep 0.5_fs # Comment
  dump position filename "trajectory#1.xyz"
end
```

Some more MD input syntax rules:

- The MD input is generally **not** case-sensitive. The only exception are file names on platforms with case-sensitive file systems (Linux).
- Empty lines are allowed.
- Commands and options are separated by space or tabulator characters. Any combination of these characters may be used as separator.
- Both DOS and UNIX line break style is acceptable.

Commands

As already noted above, the central item of the MD input is a command. Each input line contains (at most) one command, and these commands are executed in the given order. A command typically takes one or more arguments, which are given behind the command name, separated by whitespaces, tabulator characters, or commas (optional). The order of the arguments for a command is fixed (see command list in section 9.33.3). Commands may have optional arguments, which are always specified at the end of the argument list, after the last non-optional argument. If there exist multiple optional arguments for a command, not all of them need to be specified; however, they need to be specified in the correct order and without gaps:

[2] Technically speaking, ORCA will then be a turing-complete script interpreter, such that *any* computational problem can be solved with ORCA :-)

```
%md
  command arg1 arg2 arg3                  # fine
  command arg1, arg2, arg3                # fine
  command arg1 arg2 arg3 optarg1          # fine
  command arg1 arg2 arg3 optarg1 optarg2  # fine
  command arg1 arg2 arg3 optarg2          # will not work
end
```

Apart from arguments and optional arguments, commands can also have *modifiers*. These can be considered as "sub-commands", which modify a given command, and may possess their own argument lists. Modifiers generally follow after all non-optional and optional arguments, and they may **not** possess optional arguments on their own. If a command has multiple modifiers, the order in which they are given is not important.

In the following input example, "mod1" and "mod2" are modifiers of "command". "mod1" takes one argument, "mod2" does not take arguments:

```
%md
  command arg1                            # fine
  command arg1 optarg1                    # fine
  command arg1 mod1 modarg1 mod2          # fine
  command arg1 mod2                       # fine
  command arg1 mod2 mod1 modarg1          # fine
  command arg1 optarg1 mod1 modarg1 mod2  # fine
end
```

To make this abstract definition a little more illustrative, please consider again one line from the input sample at the beginning of this section:

```
%md
  dump position stride 1 filename "trajectory.xyz"
end
```

Here, "dump" is the command, which takes one non-optional argument to specify which quantity shall be dumped – in this case, "position". The "dump" command has two modifiers, namely "stride" and "filename". The former takes one integer argument, the latter a string argument. Swapping the two modifiers (together with their respective arguments, of course) would not change the behavior.

Separating Arguments

As shown above, the arguments which are passed to a command do not need to be separated by commas. However, it is allowed (and recommended) to still use commas. First, it can increase the readability of the input file. Secondly, there exist a few ambiguous cases in which commas (or parentheses) should be used to clarify the intended meaning. One of these cases is the arithmetic minus operator. It can either be used as binary operator (subtracting one number from another), or as unary operator (returning the negative of a number). By default, the minus operator will be considered as binary operator (if possible).

Consider the case in which you want to pass two integer arguments "`10`" and "`-10`" to a command. Without commas (or parentheses), the minus is mistreated as binary operator, and only one argument will be passed to the command:

```
command 10 -10    #  Pitfall: treated as "command (10 - 10)", i.e., "command 0"
command 10, -10   #  Two arguments, as intended
command 10 (-10)  #  Also works
```

Physical Units

In many cases, it is required to specify quantities which bear a physical unit in an input file (*e. g.*, temperature, time step lengths, ...). For many quantities, there are different units in widespread use, which always leads to some confusion (just consider the "kcal vs kJ" case). ORCA handles this problem by defining default units for each quantity and requiring that all quantities are given in their default unit. ORCA's default units are the atomic units, which are heavily used in the quantum chemistry community, but not so much in the molecular dynamics community. As an *ab initio* molecular dynamics module exists in the small overlap region of both communities, some "unit conflicts" might arise. To prevent those from the beginning, it is allowed to specify units of personal choice within ORCA's MD input.

Luckily, this is as simple and convenient as it sounds. The parser of the MD module checks if a unit is given after a numeric constant, and automatically converts the constant to the internal default unit. If no explicit unit is given, the default unit is assumed. Please note that the default units within the MD module are not necessarily atomic units (see table below). Units are connected to the preceding numerical value by an underscore:

```
%md
  timestep  1.0_fs
  timestep 41.3_au # identical
  timestep  1.0    # identical, as default time unit in MD module is fs
end
```

In the following, all units which are currently known to the MD module's parser are listed, sorted by physical quantities. The default unit for each quantity is printed in **bold letters**. Additive constant and factor are applied to convert a unit into the default unit. The additive constant is applied before the factor. A "-" sign means that the constant/factor is not applied. More units will be probably added in the future.

Unit Symbol	Additive constant	Factor
Length Units		
Angstrom	-	-
A	-	-
Bohr	-	0.5291
pm	-	0.01
nm	-	10.0
Time Units		
fs	-	-
ps	-	1000
au	-	0.02419
Temperature Units		
Kelvin	-	-
K	-	-
Celsius	273.15	-
C	273.15	-

Restarting Simulations

Ab initio molecular dynamics simulation are computationally expensive, and will typically run for a long time even in the case of medium-sized systems. Often, it is desirable to perform such a simulation as a combination of multiple short runs (*e. g.*, if the queuing system of the cluster imposes a maximum job time). The ORCA MD module writes a restart file in each simulation step, which allows for the seamless continuation of simulations. This restart file has the name "`basename.mdrestart`", where `basename` is the project's base name. To load an existing restart file, use the `restart` command *(see command list below).*

In the first run of a planned sequence of runs, no restart file exists yet. for this case, the `restart` command offers the `ifexists` modifier. The restart file is only loaded if it exists. If not, the restart is simply skipped, and no error is thrown. By using this modifier, you can have the `restart` command already in place in the first run of a sequence (where no restart file exists in the beginning), and do not need to modify the input after the first run has finished.

A similar case arises when considering the initialization of the velocities, which is performed by the `initvel` command. In the first run, you usually want to initialize the velocities. However, in all

following runs of the sequence, the velocities are read from the restart file, and you do not want to overwrite them by random velocities. To this aim, the `initvel` command can be used with the `no_overwrite` modifier. Velocities are only randomly initialized if they have not been defined before. If they have been defined (either by an earlier call to `initvel` or by a successful `restart` command), the initialization is simply skipped.

Concerning the `dump` command, it is good to know that trajectory files are appended (*not* overwritten) by default. If you ever want to overwrite an existing trajectory file by a `dump` command, use the `replace` modifier.

Please note that **only** the positions, velocities, and time step counters are restarted when executing a `restart` command. All other properties (thermostats, regions, trajectory dumps, constraints, cells, etc.) are **not** restarted. They should remain in the input file, as executed in the first run of a sequence.

In the end of this discussion, a short example is given. If the MD input file

```
%md
  timestep  0.5_fs
  restart ifexists
  initvel 300_K no_overwrite
  thermostat berendsen 300_K timecon 10.0_fs
  dump position stride 1 filename "trajectory.xyz"
  run 100
end
```

is subsequently executed ten times (without any modification), the resulting trajectory file will be exactly identical to that obtained if the following input is executed once:

```
%md
  timestep  0.5_fs
  initvel 300_K
  thermostat berendsen 300_K timecon 10.0_fs
  dump position stride 1 filename "trajectory.xyz"
  run 1000
end
```

Regions

In the ORCA MD module, **regions** can be defined. This concept does *not* refer to regions in space, but rather to subsets of atoms in the system. A region is nothing more than a list of atoms. Regions may overlap, *i. e.*, atoms can be part of more than one region at a time. The atoms which are part of a certain region remain the same until the region is manually re-defined, *i. e.*, regions are fixed and do not adapt to any changes in the system. There exist a few pre-defined regions which have a name. User-defined regions, in contrast, only carry an integer identifier. The following regions are pre-defined in any case:

- `all`: Contains all atoms of the system. This is the default if no region is specified in some command, so by default, the command will always act on the whole system.

- `active`: This region contains all movable ("non-frozen") atoms. By default, it is identical to the `all` region. Atoms inside of this region are updated by the time integration in a molecular dynamics run, and are considered for computing the kinetic energy.

- `inactive`: This region contains all atoms which are *not* part of the `active` region. These atoms are "frozen"; they are ingnored by the time integration, and not considered for the computation of the kinetic energy. They simply remain on their initial positions. This is in principle identical to applying cartesian constraints to the atoms; however, it is much faster. As constraints have to be solved iteratively *(see below)*, cartesian constraints become quite computationally demanding if applied to thousands of atoms.

From these three pre-defined regions, only the `active` region can be re-defined manually. Changes in the composition of the `active` region automatically modify the `inactive` region. The `all` region obviously cannot be changed.

Regions can be useful for many purposes. For example, a "realistic" wall of atoms can be built around the system by defining the `active` region such that it only contains the non-wall atoms. The wall atoms will then be frozen. Apart from that, trajectories of regions can be written to disk, only containing the "interesting" part of a simulation. Furthermore, velocity initialization can be applied to regions, enabling to start a simulation in which different sets of atoms possess different initial temperatures. Thermostats can be attached to regions to keep different sets of atoms at different temperatures during the whole simulation. This allows for sophisticated simulation setups (cold solute in hot solvent, temperature gradient through the system, etc).

Regions are defined by the `define_region` command. Many other commands take regions as optional arguments. Please see the command list below.

9.33.3 Command List

In the following, an alphabetical list of all commands currently known to the MD module is given. The description of each command starts with a small box which contains the command's name and a table of arguments and modifiers. The last-but-one column in the table specifies the type of each argument. Possible types are "Integer", "Real", "String", and "Keyword". In the latter case, the last column contains a list of allowed keyword values in { `braces` }. If the type is "Real" and is a physical quantity with unit, the quantity is given in the last column in [square brackets]. This table is followed by a text description of the command.

Command Overview

cell				
Mandatory Arguments:		-		
Optional Arguments:		-		
Modifiers:	`cube`	...	...	*see text*
	`rect`	...	...	*see text*
	`rhomb`	...	...	*see text*
	`sphere`	...	...	*see text*
	`ellipsoid`	...	...	*see text*
	`none`	-	-	-
	`spring`	k	Real	*see text*
	`elastic`	t_{avg}	Real	[time]
		c_{response}	Real	*see text*
	`anisotropic`	-	-	-
	`pressure`	...	...	*see text*
	`fixed`	-	-	-

Defines a harmonic repulsive wall around the system. This helps to keep the molecules inside of a well-defined volume, or to keep a constant pressure in the system. In the latter case, the cell can be defined as elastic, such that it exerts a well-defined pressure *(see below)*. Please note that ORCA does not feature periodic boundary conditions, and therefore, all cells are non-periodic (just repulsive walls). There are several cell geometries available *(only one type of cell can be active at a time)*:

- **`cube`**: Defines a cubic cell. If two real values p_1 and p_2 are specified as coordinates, the cell ranges from (p_1, p_1, p_1) to (p_2, p_2, p_2). If only one real value p is supplied, the cell ranges from $(-\frac{p}{2}, -\frac{p}{2}, -\frac{p}{2})$ to $(\frac{p}{2}, \frac{p}{2}, \frac{p}{2})$, *i. e.* it is centered at the origin with edge length p.
- **`rect`**: Defines an orthorhombic cell. Six real values x_1, y_1, z_1, x_2, y_2, and z_2 have to be specified as coordinates *(in this order)*. The cell will range from (x_1, y_1, z_1) to (x_2, y_2, z_2).
- **`rhomb`**: Defines a parallelepiped-shaped cell (also termed as rhomboid sometimes). You have to specify twelve real values in total. The first three define one corner point p of the cell, and the remaining nine define three cell vectors v_1, v_2, and v_3, each given as cartesian vector components. The cell is then defined as the set of points $\{p + c_1 v_1 + c_2 v_2 + c_3 v_3 \mid 0 \leq c_1, c_2, c_3 \leq 1\}$ The vectors v_1, v_2, and v_3 do not need to be orthogonal to each other, but they may not all lie within one plane (cell volume would be zero).
- **`sphere`**: Defines a spherical cell. You need to specify four real values c_x, c_y, c_z, and r. The cell will then be defined as a sphere around the central point (c_x, c_y, c_z) with radius r.
- **`ellipsoid`**: Defines an ellipsoid-shaped cell. As first three arguments, you have to specify three real values c_x, c_y, c_z, which define the center of the ellipsoid to be (c_x, c_y, c_z). As fourth argument, a keyword has to follow, which may either be "`xyz`" or "`vectors`". In

the "`xyz`" case, three more real values r_x, r_y, and r_z have to be specified, which define the partial radii of the ellipsoid along the X, Y, and Z coordinate axes. If instead "`vectors`" was given, nine more real values v_x^1, v_y^1, v_z^1, v_x^2, v_y^2, v_z^2, v_x^3, v_y^3, v_z^3 have to follow after the keyword. These values define three vectors $v^1 := \left(v_x^1, v_y^1, v_z^1\right)$, $v^2 := \left(v_x^2, v_y^2, v_z^2\right)$, and $v^3 := \left(v_x^3, v_y^3, v_z^3\right)$, which are the principal axes of the ellipsoid. These vectors have to be strictly orthogonal to each other. The length of each vector defined the partial radius of the ellipsoid along the corresponding principal axis.

All cell types define a harmonic potential $E_{\text{cell}}(r) := k \cdot r^2$ which acts on all atoms in the system **outside of the cell**, where r is the closest distance from the atom's center to the defined cell surface. Atoms whose center is inside of the cell or directly on the cell surface do not experience any repulsive force. Following from the definition, the force which acts on an atom outside of the cell is always parallel to the normal vector of the cell surface at the point which is closest to the atom center. This is trivial in case of cubic, rectangular, rhombic, and spherical cells, but not so trivial for ellipsoid-shaped cells.

The spring constant k in the above equation (*i. e.*, the "steepness" of the wall) can be specified by the "`spring`" modifier, which expects one real value as argument. The spring constant has to be specified in the unit $\mathrm{kJ\,mol^{-1}\,Å^{-2}}$, other units cannot be specified here. The default value is $10\,\mathrm{kJ\,mol^{-1}\,Å^{-2}}$. Larger spring constants reduce the penetration depth of atoms into the wall, but may require shorter time steps to ensure energy conservation.

The command "`cell none`" disables any previously defined cell.

If you want to perform simulations under constant pressure, you can define an elastic cell. Then, ORCA accumulates the force which the cell exerts on the atoms in each time step, and divides this total force by the cell surface area to obtain a pressure. As this momentarily pressure heavily fluctuated, a running average is used to smooth this quantity. If the averaged pressure is larger than the external pressure which was specified, the cell will slightly grow; if it was smaller, the cell will slightly shrink. In the beginning of a simulation, the cell size will not vary until at least the running average history depth of steps have been performed.

An elastic cell is enabled by using the "`elastic`" modifier after the cell geometry definition. Subsequently, two real values t_{avg} and c_{response} are required. While t_{avg} defines the length of the running average to smooth the pressure *(in units of physical time,* ***not*** *time steps)*, the c_{response} constant controls how fast the cell size will change at most. More specific, c_{response} is the fraction of the cell volume growth per time step if the ratio of averaged and external pressure would be infinite, and at the same time the fraction of the cell volume reduction per step if the aforementioned ratio is zero. Put into mathematical form, the cell volume change

per time step is

$$V_{\text{new}} := \begin{cases} V_{\text{old}} \cdot \frac{c_{\text{response}} \cdot \frac{\langle p \rangle}{p_{\text{ext}}} + 1}{c_{\text{response}} + 1} & \text{if } \frac{\langle p \rangle}{p_{\text{ext}}} \leq 1, \\ V_{\text{old}} \cdot \frac{(c_{\text{response}} + 1) \cdot \frac{\langle p \rangle}{p_{\text{ext}}}}{\frac{\langle p \rangle}{p_{\text{ext}}} + c_{\text{response}}} & \text{if } \frac{\langle p \rangle}{p_{\text{ext}}} > 1, \end{cases}$$

where $\langle p \rangle$ represents the averaged pressure the system exerts on the walls, and p_{ext} is the specified external pressure. Good starting points are $t_{\text{avg}} = 100\,\text{fs}$ and $c_{\text{response}} = 0.001$. An already defined fixed cell can be switched to elastic by the command "`cell elastic ...`" (the dots represent the two real arguments).

By default, the size change of an elastic cell due to pressure is performed *isotropically*, *i. e.*, the cell is scaled as a whole, and exactly retains its aspect ratio. By specifying the "`anisotropic`" modifier after switching on an elastic cell, the cell pressure is broken down into individual components, and the size of the cell is allowed to change independently in the individual directions. This, of course, only makes sense for the cell geometries `rect`, `rhomb`, and `ellipsoid`. An already defined isotropic cell can be switched to anisotropic by simply executing "`cell anisotropic`".

In case of an elastic cell, the external pressure is defined by the modifier "`pressure`", which expects either one or three real values as arguments. If one argument is given, this is the isotropic external pressure. If three arguments are supplied, these are the components of the pressure in X, Y, and Z direction *(in case of orthorhombic cells)* or along the direction of the three specified vectors *(in case of parallelepiped-shaped and ellipsoid-shaped cells)*. This allows for anisotropic external pressure (probably only useful for solid state computations). Both the pressure and the pressure components have to specified in units of bar ($= 10^5\,\text{N}\,\text{m}^{-2}$), other units cannot be used. If this modifier is not used, the default pressure will be set to 1.0 bar (isotropic) if an elastic cell is used. The external pressure of an already defined cell can be changed by the command "`cell pressure ...`" (the dots represent the real argument(s)).

As all cells are non-elastic by default, there is no keyword to explicitly request this at the time of cell definition. However, possible applications might require to use an elastic cell during equilibration period, and then "freeze" this cell at the final geometry for the production run. This can be achieved by using the "`cell fixed`" command (without any additional arguments).

If the cell is elastic, there is a volume work term which contributes to the total energy of the system. ORCA computes this term in every step and adds it to the potential energy. Without this contribution, the conserved quantity would drift excessively in elastic cell runs.

To completely switch off a previously defined cell, simply use "`cell none`".

Please note that cells are not automatically restarted by using the "`restart`" command.

Examples:

Cubic cell with edge length 10 Å centered around origin:

```
cell cube 10
```

Spherical cell with radius 5 Å centered around origin and 20 kJ mol^{-1} Å^{-2} wall steepness:

```
cell sphere 0, 0, 0, 5 spring 20
```

Elastic orthorhombic cell from $(-2, -2, 0)$ to $(12, 12, 10)$, $t_{\mathrm{avg}} = 100$ fs, $c_{\mathrm{response}} = 0.001$:

```
cell rect -2, -2, 0, 12, 12, 10 elastic 100, 0.001
```

Ellipsoid-shaped cell centered on origin with partial radii 5, 10, 15 Å along the X, Y, Z axes:

```
cell ellipsoid 0, 0, 0 xyz 5, 10, 15
```

The commas are optional, but make sure to use them with negative numbers. By default, the minus operator will act as binary operator *(see discussion above)*.

---

constraint				
Mandatory Arguments:		*operation*	Keyword	{ `add`, `remove` }
		type	Keyword	{ `cartesian`, `distance`, `angle`, `dihedral` }
		atom(s)	Integer	-
Optional Arguments:		-		
Modifiers:	`target`	*value(s)*	Real	-

Manages constraints in the molecular dynamics simulation. Unlike restraints, constraints are geometric relations which are strictly enforced at every time (*i. e.*, they do not fluctuate around their target value).

The simplest possiblity is to constrain the cartesian position of an atom to some value. A zero-based atom index is required. The command `constraint add cartesian 3` would fix the fourth atom in the simulation at its current position in space. If the desired position shall be explicitly given, it can be specified via the `target` modifier, *e. g.*, `constraint add cartesian 3 target 5.0 1.0 1.0`. To determine which dimensions to fix, one of the `xyz`, `xy`, `xz`, `yz`, `x`, `y`, or `z` modifiers can be added. For example, `constraint add cartesian 3 x target 1.0` would constrain the x coordinate of atom 3 to the absolute value 1.0, but would not influence movement along the y and z coordinate at all.

By using the `distance` keyword, distances between atoms can be fixed. The command `constraint add distance 3 5` would fix the distance between atom 3 and 5 to its current value. You need to specify exactly two atom indices; multiple distance constraints are entered via multiple `constraint` commands. Also here, a desired distance value can be given via the `target` modifier, such as `constraint add distance 3 5 target 350_pm`.

Similarly, angles and dihedral angles between atoms can be fixed with the `angle` and `dihedral` keywords. Angles are defined by three atom indices, and dihedral angles by four atom indices. Also here, target values may be specified. Any combination of cartesian, distance, angle, and dihedral constraints may be used simultaneously, and may even be applied to the same group of atoms. A molecule can be made completely rigid by constraining all its bonds, angles, and torsions. Please make sure that your constraints are not over-determined, and do not contradict each other. Otherwise, they can't be enforced and the simulation will print warnings or crash.

Constraints are removed with the `remove` keyword. It is not possible to overwrite/update already existing constraints with new constraints; these need to be removed first. You can either remove single constraints, *e. g.*, `constraint remove distance 3 5`, or groups of similar constraints. To remove all angle constraints, use `constraint remove angle all`. To remove all restraints, enter `constraint remove all`.

Please note that each constraint decreases the number of the system's degrees of freedom (dof) by one. This effect is included, *e. g.*, in the temperature computation, where the dof count enters.

It is computationally inefficient to define a large number of cartesian constraints if a subset of atoms simply shall be fixed. A more efficient approach is to define an active region which only contains the atoms which shall be movable (see `define_region` command). All atoms outside of the active region will not be subject to time integration and therefore keep their positions. However, please note that these atoms may not be involved in any other (distance, angle, dihedral) constraint.

---

define_region			
Mandatory Arguments:	*identifier*	Keyword/Integer	...
	atomlist	Integer(s)	-
Optional Arguments:	-		
Modifiers:	-		

Defines or re-defines regions. Regions are just subsets of atoms from the system *(see section above)*.

As written in the section above, there exist several pre-defined regions which are identified by names. The only such pre-defined region which can be re-defined by the user is the `active` region. All atoms in this region are subject to time integration in molecular dynamics runs. All other atoms are simply ignored and remain on their initial positions.

To re-define the `active` region, use the command "`define_region active 1 5 7 ...`". The integer arguments after `active` are the numbers of the atoms to be contained in the region,

in the order given in the ORCA input file. Atom numbers are generally zero-based in ORCA, *i. e.*, counting starts with 0.

Apart from that, user-defined regions are supported. These are identified with an integer number instead of a name. The integer numbers do not need to be sequential, *i. e.*, it is fine to define region 2 without defining region 1. To give an example, the command "`define_region 1 17 18 19`" defines region 1, and adds atoms 17, 18, and 19 to this newly defined region.

If you want to specify a range of atoms, you can use the syntax "`a..b`" to include all atom numbers from `a` to `b`. If you want only, *e. g.*, every third atom in a range, you can use "`a..b..i`" to add the range from `a` to `b` with increment `i`. As an example, "`2..10..3`" will expand to the list `2, 5, 8`. You can mix atom numbers and ranges, as shown in the following example *(as always, the commas are optional)*:

```
define_region active 1, 4, 5..11, 14, 17..30..2
```

Please make sure not to add the same atom multiple times to the list. Otherwise, the command will fail and the run will be aborted.

---

dump				
Mandatory Arguments:		*quantity*	Keyword	{ `position`, `velocity`, `force` }
Optional Arguments:		-		
Modifiers:	`format`	*fmt*	Keyword	{ `xyz`, `pdb` }
	`stride`	*n*	Integer	-
	`filename`	*fname*	String	-
	`region`	*region*	...	...
	`replace`	-		
	`none`	-		

Specifies how to write the output trajectory of the simulation. The *quantity* argument can be one of the keywords `position`, `velocity`, and `force`.

The `stride` modifier specifies to write only every n-th time step to the output file (default is $n = 1$, *i. e.*, every step).

The `format` modifier sets the format of the output file. Currently, only the `xyz` and `pdb` formats are implemented. If not specified, ORCA tries to deduce the format from the file extension of the specified file name. If also no file name is given, trajectories will be written in xyz format by default.

The `filename` modifier gives the output file name. If not specified, the default file name will have the form "`proj-qty-rgn.ext`", where `proj` is the base name of the ORCA project, `qty` is one of `postrj`, `veltrj`, or `frctrj`, `rgn` specifies the name or number of the region for which the dump is active, and `ext` is the file extension selected by the `format` modifier.

If the trajectory file already exists at the beginning of a `run` command, new frames will be appended to its end by default. If you want to overwrite the existing file instead, use the `replace` modifier. The old existing file is erased only once after a dump with this modifier has been specified. If multiple `run` commands follow after the dump definition, the trajectory will **not** be replaced before each of these runs, only before the first one among them. To overwrite the file another time, simply re-define the dump with the `replace` modifier. If the file does not yet exist at the beginning of a run, this modifier has no effect.

With the `region` modifier, the trajectory output can be restricted to a specific region (*i. e.*, subset of atoms). This modifier expects one argument, which is either the name of a pre-defined region or the number of a user-defined region *(see above)*. If not specified, the trajectory of the whole system will be written. Multiple dump commands for multiple regions can be active at the same time, but each pair of region and quantity *(position/velocity/force)* can have only one attached dump command at a time (re-defining will overwrite the dump settings).

Use the `none` modifier to disable writing this quantity to an output file. The command "`dump position none`" will disable writing of all position trajectories for all regions. To disable only the dump for a specific region, use "`dump position region r none`", where `r` is the name or number of the region.

The default is to write a position trajectory with `stride 1` and `format xyz` to a file named "proj-postrj-all.xyz", where "proj" is the base name of the ORCA project. If you want to create no output trajectory at all, use "`dump position none`" as described above.

---

initvel				
Mandatory Arguments:		*temp*	Real	[temperature]
Optional Arguments:		-		
Modifiers:	`region`	*region*	...	...
	`no_overwrite`	-		

Initializes the velocities of the atoms by random numbers based on a Maxwell–Boltzmann distribution, such that the initial temperature matches *temp* (see also section 9.33.4.2). Please note that this overwrites all velocities, so do not call this command when your system is already equilibrated (*e. g.*, to change temperature – use a thermostat instead).

The total linear momentum of the initial configuration is automatically removed, such that the system will not start to drift away when the simulation begins. This only concerns the initial configuration. Total linear momentum might build up during the simulation due to numeric effects.

With the `region` modifier, the initialization of velocities can be performed for a specific region (*i. e.*, subset of atoms). This modifier expects one argument, which is either the name of a pre-defined region or the number of a user-defined region *(see above)*. If not specified, the command acts on the whole system.

The `no_overwrite` modifier only initializes the velocities if no atom velocities have been defined/read before. This is useful in combination with the `restart` command: After reading an existing restart file, the velocities are already known, and the initialization will be skipped if this modifier is used. The following combination of commands in a MD input would initialize the velocities only upon first execution, and restart the positions and velocities on all following executions of the same input:

```
restart ifexist
initvel 350_k no_overwrite
```

If neither the `initvel` command nor a `restart` command is not invoked before a "`run`" call, the atom velocities will be initialized to zero before starting the run.

---

printlevel			
Mandatory Arguments:	*value*	Keyword	{ `low`, `medium`, `high`, `debug` }
Optional Arguments:	-		
Modifiers:	-		

Controls the amount of information which is printed to the screen during the simulation. `debug` should be used only in rare cases, because it might slow the simulation down heavily.

The default value is `medium`.

randomize				
Mandatory Arguments:	-			
Optional Arguments:	*seed*	Integer	-	
Modifiers:	-			

There are a few algorithms in the ORCA MD module which rely on random numbers, *e. g.*, the initialization of atom velocities with the `initvel` command. These random numbers are so-called "pseudorandom numbers", produced by a deterministic generator. This generator has a *state*, which is simply an integer number. If initialized to the same state, the generator will always create the same sequence of "random" numbers. This sounds like a deficiency at first thought, but is a very important feature for scientific reproducibility and for debugging purposes. If you start the same MD input file with "random" velocity initialization a couple of times, the trajectory will be exactly identical in all runs.

However, there are cases in which this behavior is not desired, *e. g.*, if you want to average a property over multiple trajectories of the same system. In these cases, call the `randomize` command in the beginning of the input. If no argument is given, the random number generator is initialized with the current system time as a seed. MD runs started at different times will have different random velocities in the beginning. If you want more control over this process, you can also specify a positive integer number as argument, which is used as initial random seed. Simulations started with the same seed argument will have identical initial random velocities (if all other system parameters such as atom count, atom types, ... remain identical).

Without a call to `randomize`, a seed of 1 is always used.

---

restart				
Mandatory Arguments:	-			
Optional Arguments:	*fname*	String	-	
Modifiers:	`ifexist`	-		

Reads a restart file to continue a previous molecular dynamics run. Such a restart file is written after every simulation step, such that a crashed simulation may easily be recovered. The file name of the restart file may be given via *fname*; otherwise, it is deduced from the project's base name as `<basename>.mdrestart`.

If the `ifexist` modifier is specified, a restart is only performed if the restart file exists. The error and abort that would normally occur in case of a non-existent restart file are suppressed by this flag. This is useful in the first of a series of batch runs, where the restart file does not yet exist in the beginning.

Please note that the following quantities are stored to/loaded from restart files:

- Atom Positions
- Atom Velocities
- Simulation step number
- Elapsed physical time

All other quantities (timestep, regions, thermostat, constraints, cells, etc.) are **not** restarted and need to be set in the input file.

---

run			
Mandatory Arguments:	*n*	Integer	-
Optional Arguments:	-		
Modifiers:	-		

Performs a simulation run over n time steps with the current settings, applying the Velocity Verlet algorithm to solve the equations of motion (see section 9.33.4.1). You might want to call commands like "`timestep`", "`initvel`", "`thermostat`", and "`dump`" before.

Please note that only atoms within the `active` region will be subject to time integration. All other atoms will be skipped, and will therefore retain their initial positions.

If no call to "`initvel`" occurred before this command, the atom velocities are initialized to zero.

If no call to "`timestep`" occurred before this command, a default time step of 0.5 fs is set.

---

scflog			
Mandatory Arguments:	*value*	Keyword	{ `discard`, `last`, `append`, `each` }
Optional Arguments:	-		
Modifiers:	-		

Controls how/if the detailed output from the electron structure calculation (*i. e.*, integrals, scf, gradient, ...) will be written to log files. `discard` completely discards the output. `last` only keeps the last output for each program call (useful to read error message if simulation aborts). `append` redirects all the output into one single log file, appending each step at the end of the file. `each` writes the output for each step and each program to different log files, which have the step number in their file names.

To print the detailed output of the electron structure calculation to the screen instead, see the `printlevel` command.

The default value is `append`. Note that this can lead to large log files in long runs.

thermostat				
Mandatory Arguments:		*type*	Keyword	{ **berendsen**, **none** }
Optional Arguments:		*temperature*	Real	[temperature]
Modifiers:	**timecon**	*dt*	Real	[time]
	region	*region*	...	...
	massive	-		

Changes the atom thermostat settings for subsequent simulation runs. **type** sets the thermostat type. Currently, only the Berendsen thermostat is implemented. Use **none** as type to disable the thermostat.

The **temperature** argument sets the target temperature to which the system is thermostated. If this argument is omitted, the temperature from the last call to the **initvel** command is used (if no such call was invoked before, the simulation is aborted).

The **timecon** modifier sets the coupling strength of the thermostat (large time constants correspond to weak coupling). The default value is 10 fs, which is a relatively strong coupling. Values in the range of 10 ... 100 fs are reasonable (see also section 9.33.4.3).

The **massive** modifier activates *massive thermostating*, which means that each degree of freedom is assigned to an independent thermostat. This is useful for pre-equilibration runs (helps to reach energy equipartition) and should not be used during production runs, as it heavily distorts the dynamics.

With the **region** modifier, the thermostat can be attached to a specific region (*i. e.*, subset of atoms). This modifier expects one argument, which is either the name of a pre-defined region or the number of a user-defined region *(see above)*. If not specified, the thermostat acts on the whole system. Multiple thermostats for multiple regions can be active at the same time, but each region can have only one attached thermostat at a time (re-defining will overwrite the thermostat settings).

The command "**thermostat none**" will remove all thermostats from all regions. If you want to disable a thermostat for a specific region only, use "**thermostat none region r**", where **r** is the name or number of the region.

Please note that a Berendsen thermostat will show no effect (or unexpected effects) if the system's temperature is close to 0 K, as it works by multiplying the velocities with a factor.

timestep			
Mandatory Arguments:	*dt*	Real	[time]
Optional Arguments:	-		
Modifiers:	-		

Sets the simulation time step Δt used to integrate the equations of motion for all following runs to *dt*. If your system contains hydrogen atoms, a time step not above 0.5 fs is recommended. If only heavier atoms are present, a larger time step may be chosen. A good estimate for a time step that still allows for an accurate simulation is $\Delta t = \sqrt{m} \cdot 0.5\,\text{fs}$, where m is the mass of the lightest atom in the system (in a.m.u.).

If this command is not invoked before a "**run**" call, a default time step of 0.5 fs will be set before starting the run.

9.33.4 Scientific Background

In this section, some of the methods and algorithms used within ORCA's MD module are described in some more depth, with a focus on the scientific background.

9.33.4.1 Time Integration and Equations of Motion

The central concept of molecular dynamics simulations is to solve Newton's equations of motion (at least as long as the atom cores are treated classically). These read

$$\ddot{x}_i\left(t\right) = \frac{F_i\Big(\vec{x}\left(t\right)\Big)}{m_i}, \quad i = 1 \ldots N, \tag{9.1}$$

where $x_i\left(t\right)$ denotes the position of the i-th degree of freedom at time t, m the corresponding mass, and F_i the force acting upon this degree of freedom. As the force may depend on all positions, this is a coupled system of N ordinary differential equations (ODEs). In the general case, it is not possible to obtain an analytical solution of this system, and therefore numerical solution methods are applied. These are almost always based on discretizing the time variable and approximately solving the system by taking finite time steps.

Of all different methods to numerically solve coupled systems of ODEs, the *symplectic integration schemes* for Hamiltonian systems attained special attention in the field of molecular dynamics. They possess a very good conservation of energy. In contrast to many other methods, they show a reasonable behavior when investigating the long-term evolution of chaotic Hamiltonian systems (like, *e. g.*, MD simulations). Three popular such symplectic integration schemes are the *Leapfrog* algorithm, the *Verlet* method, and the *Velocity Verlet* integrator. Despite their different names, they are very similar. It can be easily seen that the Verlet and Velocity Verlet methods are algebraically equivalent (by eliminating the velocities from the Velocity Verlet algorithm), and it can be shown that, eventually, all three methods are identical.[3] All three methods are explicit integration methods with a global error of order 2, and therefore one order better than the semi-implicit Euler method, which is also a symplectic integration scheme. As the Velocity Verlet algorithm is the only of these three methods which yields velocities and positions at the same point in time, many popular molecular dynamics packages (CP2k, CPMD, LAMMPS) use this scheme. For the same reasons, the ORCA MD module uses the Velocity Verlet algorithm as time integration method.

The general equations of the Velocity Verlet scheme read

$$\vec{x}\left(t + \Delta t\right) = \vec{x}\left(t\right) + \vec{v}\left(t\right)\Delta t + \frac{1}{2}\vec{a}\left(t\right)\Delta t^2, \tag{9.2}$$

$$\vec{v}\left(t + \Delta t\right) = \vec{v}\left(r\right) + \frac{\vec{a}\left(t\right) + \vec{a}\left(t + \Delta t\right)}{2}\Delta t. \tag{9.3}$$

[3] Hairer, Lubich, Wanner, "Geometric Numerical Integration", Springer 2006.

By inserting

$$\vec{a}_i\,(t) = \frac{\vec{F}_i\,(t)}{m_i}, \quad i = 1 \ldots N, \tag{9.4}$$

one arrives at the two-step method

$$\vec{x}_i\,(t + \Delta t) = \vec{x}_i\,(t) + \vec{v}_i\,(t)\,\Delta t + \frac{\vec{F}_i\,(t)}{2m_i}\Delta t^2, \quad i = 1 \ldots N, \tag{9.5}$$

$$\vec{v}_i\,(t + \Delta t) = \vec{v}_i\,(r) + \frac{\vec{F}_i\,(t) + \vec{F}_i\,(t + \Delta t)}{2m_i}\Delta t, \quad i = 1 \ldots N, \tag{9.6}$$

which is implemented in ORCA's MD module.

9.33.4.2 Velocity Initialization

In the beginning of a MD simulation, it is often the case that only the initial positions of the atoms are known, but not the velocities. As MD simulations are performed at some finite temperature, it is a good idea to initialize the velocities in a way such that the desired simulation temperature is already present in the beginning. In statistical mechanics, it is often assumed that the velocity distribution of atoms is given by a Maxwell–Boltzmann distribution (which is strictly only the case in idealized gases). Therefore, it is a reasonable choice to initialize the atom's velocities according to the Maxwell–Boltzmann equation in the beginning of a MD simulation. The goal is to find an initial velocity distribution in which each degree of freedom possesses a similar amount of energy, such that the equipartition theorem is approximately fulfilled.

The scalar Maxwell–Boltzmann velocity distribution (leaving out the normalization factor) at temperature T is given by

$$f\,(v) = v^2 \exp\Big(-\frac{mv^2}{2k_B T}\Big). \tag{9.7}$$

To initialize the particle's velocities such that this distribution function is fulfilled, one starts with a series of normal-distributed random numbers with mean 0 and variance 1, denoted by $\mathcal{N}\,(0, 1)$. The cartesian velocity components for each atom are then computed by

$$v_{i,\alpha} := \sqrt{\frac{k_B T}{m_i}}\,\mathcal{N}\,(0, 1)\,, \quad \alpha \in \{x, y, z\}, \quad i = 1 \ldots N. \tag{9.8}$$

As the C++98 standard does not offer a platform-independent way of obtaining normal-distributed random numbers, these are internally computed from uniformly distributed random numbers by applying the *Box–Muller transform*: Assuming that u_1 and u_2 are two uniformly distributed random

numbers from the interval $[0, 1]$, the equations

$$z_1 := \sqrt{-2\log(u_1)}\cos(2\pi u_2), \tag{9.9}$$

$$z_2 := \sqrt{-2\log(u_1)}\sin(2\pi u_2) \tag{9.10}$$

yield two new random numbers z_1 and z_2 which obey a normal distribution with mean 0 and variance 1.

After the velocities have been initialized, the total linear momentum of the system will probably have some finite value other than zero. As the linear momentum is (approximately) conserved within a molecular dynamics simulation, this would result in the system drifting away into one direction during the course of the simulation, which is probably not desired. Therefore, the total momentum is explicitly set to zero after the Maxwell–Boltzmann initialization:

$$\vec{P}_{\text{tot}} := \sum_{i=1}^{N} m_i \vec{v}_{i,\text{old}}, \tag{9.11}$$

$$\vec{v}_{i,\text{new}} := \vec{v}_{i,\text{old}} - \frac{\vec{P}_{\text{tot}}}{m_i N}, \quad i = 1 \ldots N. \tag{9.12}$$

This, of course, might change the initial temperature. Therefore, a final step is performed, in which all velocity vectors are multiplied with a factor that is determined such that the initial temperature exactly matches the target value.

9.33.4.3 Thermostats

After the initial velocities have been initialized to some finite temperature, it might be assumed that one can simply start the time integration of the dynamical system (equivalent to the *NVE ensemble*), and the starting temperature would be approximately preserved. In a real system, however, there are (at least) two reasons why the temperature will strongly deviate from the initial value already after a few steps. First, the initial velocity distribution only considers the kinetic energy of the particles, but some amount of energy will be exchanged with the potential energy contribution (*e. g.*, bond stretching) immediately, altering the temperature. Secondly, the numerical errors introduced due to the finite time step (and in case of *ab initio* MD, also due to the approximate forces) will lead to a drift in energy and therefore in temperature. To counter these effects, it is often desirable to have a temperature control during the course of the simulation (which then runs in the *NVT ensemble*), which is called a thermostat.

There exist many different kinds of thermostats, ranging from simple expressions up to highly complex dynamical systems on their own. But all of them share a common issue: If the thermostat is coupled only weakly to the system, the temperature will change anyway. However, if the thermostat is coupled more strongly to the system (*i. e.*, intervenes stronger), then the dynamics of the simulation

will change, no longer resembling the undisturbed original dynamics which one wants to investigate. Therefore, it is always a tradeoff between temperature stability and disturbed dynamics to decide how strong a thermostat should be coupled to the system.

At the moment, only the simple Berendsen thermostat is implemented in the ORCA MD module. More thermostats (*e. g.*, the widely used Nosé–Hoover thermostat) will follow in a future release.

Berendsen Thermostat

The Berendsen thermostat is similar to the simple velocity rescaling scheme, but enhanced by a time constant τ to control the coupling strength. Let T_0 be the desired target temperature and T the current temperature of the system. Then the temperature gradient caused by the thermostat can be expressed as

$$\frac{dT}{dt} = \frac{T_0 - T}{\tau}. \tag{9.13}$$

Considering the fact that discrete time steps Δt are used, the correction factor for the velocities in each time step is determined by

$$f := \sqrt{1 + \frac{\Delta t\,(T_0 - T)}{T\tau}} \tag{9.14}$$

The new velocities are then easily obtained as

$$\vec{v}_{i,\mathrm{new}} := f \cdot \vec{v}_{i,\mathrm{old}}, \quad i = 1 \ldots N. \tag{9.15}$$

Let's consider some special cases. If $\tau = \Delta t$, the whole temperature deviation from T_0 is corrected immediately, such that the temperature is always exactly kept at the target value. This is identical to simple velocity rescaling (without any time constant), which is known to work poorly for most systems (a single harmonic oscillator would, *e. g.*, simply explode). With a larger time constant $\tau > \Delta T$, the coupling strength is reduced, leading to reasonable results. Typically, a value of τ in the range of $20 \ldots 200 \cdot \Delta T$ will be applied. For $\tau \to \infty$, the coupling strength goes to zero, such that the thermostat is no longer active. Values of $\tau < \Delta T$ are not allowed.

From the formula, it becomes clear that a Berendsen thermostat will have no effect if the system has a temperature of 0 K (or in the "massive" case: if the considered degree of freedom has 0 K), because it is based on multiplying the velocities by a factor to modify the temperature. Therefore, this type of thermostat can't be used to heat a system up starting from 0 K.

9.33.4.4 Constraints

Unlike restraints, constraints are geometric relations which are strictly enforced at every time (*i. e.*, they do not fluctuate around their target value). Many molecular dynamics techniques make use of geometric constraints (*e. g.*, to keep water molecules rigid, or to fix some reaction coordinate). Standard BOMD describes the nuclei as point charges in space, such that the motion of the atoms is governed by the laws of classical mechanics. Systems in classical mechanics can be described by the Lagrange formalism, which contains a well established sub-formalism for holonomic constraints, namely the method of Lagrange multipliers.

However, molecular dynamics discretizes time to solve the equations of motions with finite time steps, often using a Verlet integrator. With discretized time, it is slightly more involved to enforce and keep exact constraints. Within the last decades, algorithms have been developed to do so. One famous among them is the SHAKE algorithm. However, it comes with the disadvantage of only enforcing the constraints in the positions, not in the velocities. This may lead to problems such as artificially high temperature values due to "hidden" velocities along the constrained directions. An extension of SHAKE which also enforces the constraints for the velocities is the RATTLE algorithm, which is implemented in the AIMD module of ORCA.

The RATTLE scheme is a generalization of the Velocity Verlet integrator to allow for constraints. This means that RATTLE is not applied in addition to the Velocity Verlet integrator, but replaces it. In case of no active constraints, both methods are identical. A system of coupled constraints cannot be solved exactly in one step, and RATTLE uses an iterative approach to enforce all constraints simultaneously. This is often a matter of concern with respect to performance. However, in AIMD, the energy and gradient calculations typically take seconds or even minutes per step, such that the additional computation time for iteratively solving the constraints can be totally neglected.

As an iterative procedure, RATTLE is not able to give exact solutions, but only converged up to a given tolerance. In the ORCA MD module, the tolerance is currently set to 10^{-2} pm for distances, and 10^{-4} degree for angles and dihedral angles. This tolerance is typically reached within a few dozen iterations. In some cases, it might happen that the RATTLE iterations do not converge to the required tolerance. This is typically the case if the set of constraints is over-determined or contradictory.

The mathematical and technical details of RATTLE are not described here, they can be found in the literature. The general concept of RATTLE was suggested by Andersen. [1] The original article only covered distance constraints. A follow-up work describes how to handle any holonomic constraints, in particular how to constrain angles and dihedral angles. [2] The Wilson vectors (*i. e.*, derivatives of angles and dihedral angles with respect to cartesian atom positions) are taken from Wilson's original work. [3]

Bibliography

[1] Andersen, H. C. (**1983**) *Journal of Computational Physics*, 52(1), 24 .

[2] Kutteh, R. (**1998**) *CCP5 Newslett.*, 9.

[3] Wilson, E. B.; Decius, J. C.; Cross, P. C. (**1955**) *Molecular Vibrations – The Theory of Infrared and Raman Vibrational Spectra.*