MDorado

Documentation of Version 0.3.0

Jan Neumann,
June 10, 2022, Rostock

Contents

1 vectors

1.1 `norm_vecarray`

Given an array of vectors, normalizes each vector and returns the array of normalized vectors as well as the length of each vector.

`mdorado.vectors.norm_vecarray(vecarray)`

Parameters:

`vecarray`:	ndarray Array of shape N_{vec} (number of vectors), N_{dim} (dimensionality of the vectors) containing the vectors that will be normalized.

Returns: `unitvecarray`, `norm`: ndarray, ndarray
Returns two arrays, the first (unitvecarray) is of the same shape as the input array and contains the normalized vectors and the second (norm) is of the shape (N_{vec},) and contains the length of the corresponding original vector.

Example:

```python
1 # example_norm_vecarray.py
2 import numpy as np
3 from mdorado.vectors import norm_vecarray
4 
5 vectorarray = np.array([[11,4,-4], [4,1,8], [-6,-7, 2], [3,0, -1]])
6 unitvectors, lengths = norm_vecarray(vecarray=vectorarray)
7 print(unitvectors)
8 print(lengths)
```

The function requires an array of input vectors (line 5). These vectors will each be normalized and returned as unit vectors. Additionally, the length of the original vectors is returned:

```
$ python python example_norm_vecarray.py
[[ 0.88929729  0.32338083 -0.32338083]
 [ 0.44444444  0.11111111  0.88888889]
 [-0.63599873 -0.74199852  0.21199958]
 [ 0.9486833   0.         -0.31622777]]
[12.36931688  9.         9.43398113  3.16227766]
```

1.2 `pbc_vecarray`

Applies periodic boundary condition to an array of vectors so that the resulting array satisfies the minimum image convention for cuboid simulation boxes (all angles are 90°).

`mdorado.vectors.pbc_vecarray(vecarray, box)`

Parameters:

`vecarray`:	ndarray Array of shape N_{vec} (number of vectors), N_{dim} (dimensionality of the vectors) on which the periodic boundary condition will be applied.
`box`:	array-like One-dimensional array-like where the elements contain the length of the simulation box in the particular dimension.

Returns: ndarray
Array of the same shape as `vecarray` containing the vectors corrected for the minimum image convention.

Example:

```python
# example_pbc_vecarray.py
import numpy as np
from mdorado.vectors import pbc_vecarray
```

```
4
5 vectorarray = np.array([[11,4,-4], [4,1,8], [-6,-7, 2], [3,0, -1]])
6 box = [10,5,7]
7 pbc_vectorarray = pbc_vecarray(vecarray=vectorarray, box=box)
8 print(pbc_vectorarray)
```

The function requires an array containing vectors (line 5) as well as the length of each box dimension (line 6) as an input. At the moment, this only works for cuboid simulation boxes (all angles 90°). The printed array contains the corrected vectors:

```
$ python example_pbc_vecarray.py
[[ 1 -1  3]
 [ 4  1  1]
 [ 4 -2  2]
 [ 3  0 -1]]
```

1.3 vectormatrix

Given two sets of particle positions $\boldsymbol{A} = [\boldsymbol{A_1}, \boldsymbol{A_2}, \dots, \boldsymbol{A_n}]$ and $\boldsymbol{B} = [\boldsymbol{B_1}, \boldsymbol{B_2}, \dots, \boldsymbol{B_m}]$ computes a matrix $\boldsymbol{C}$ containing vectors for all combinations $\boldsymbol{C_{i,j}} = \boldsymbol{B_j} - \boldsymbol{A_i}$.

`mdorado.vectors.vectormatrix(apos, bpos)`

Parameters:

`apos`:	ndarray Array of shape (n, 3) containing the position vectors of all particles A.
`bpos`:	ndarray Array of shape (m, 3) containing the position vectors of all particles B.

Returns: ndarray
Returns an array (`abmat`) of shape (n, m, 3) containing all vectors $\overrightarrow{A_iB_j}$:
`abmat[i, j] = bpos[j] - apos[i]`.

Example:

```python
# example_vectormatrix.py
import numpy as np
from mdorado.vectors import vectormatrix

vectorarray = np.array([[11,4,-4], [4,1,8], [-6,-7, 2], [3,0, -1]])
vecmat = vectormatrix(apos=vectorarray, bpos=vectorarray)
print(vecmat)
```

Here, we use the vectors in `vectorarray` as positional vectors of four different particles $\boldsymbol{A}, \boldsymbol{B}, \boldsymbol{C}, \boldsymbol{D}$. The resulting matrix then contains all vectors connecting the particles. The comments (everything after #) are of course not part of the output but should provide clarity over the structure of the resulting matrix:

```
$ python example_vectormatrix.py
[[[  0.    0.    0.]      #$\overrightarrow{AA}$
  [ -7.   -3.   12.]      #$\overrightarrow{AB}$
  [-17.  -11.    6.]      #$\overrightarrow{AC}$
  [ -8.   -4.    3.]]     #$\overrightarrow{AD}$

 [[  7.    3.  -12.]      #$\overrightarrow{BA}$
  [  0.    0.    0.]      #$\overrightarrow{BB}$
  [-10.   -8.   -6.]      #$\overrightarrow{BC}$
  [ -1.   -1.   -9.]]     #$\overrightarrow{BD}$

 [[ 17.   11.   -6.]      #$\overrightarrow{CA}$
  [ 10.    8.    6.]      #$\overrightarrow{CB}$
  [  0.    0.    0.]      #$\overrightarrow{CC}$
  [  9.    7.   -3.]]     #$\overrightarrow{CD}$

 [[  8.    4.   -3.]      #$\overrightarrow{DA}$
  [  1.    1.    9.]      #$\overrightarrow{DB}$
  [ -9.   -7.    3.]      #$\overrightarrow{DC}$
  [  0.    0.    0.]]]    #$\overrightarrow{DD}$
```

1.4 get_vectormatrix

Uses `mdorado.vectors.vectormatrix` to compute a vector matrix containing all vectors between all combinations of particles A and B for every timestep in a universe.

`mdorado.vectors.get_vectormatrix(universe, agrp, bgrp, pbc=True)`

Parameters:

`universe`:	MDAnalysis.Universe Universe containing the trajectory.
`agrp`:	AtomGroup from MDAnalysis AtomGroup containing all atoms of A.
`bgrp`:	AtomGroup from MDAnalysis AtomGroup containing all atoms of B.
`pbc`:	bool, optional Specifies whether periodic boundary conditions should be applied to find the shortest vector from A to an image of B. Calls the `mdorado.vectors.pbc_vecarray` function. Only works for cuboid boxes (all angles are 90°). Default is `True`.

Returns: ndarray
Array of the shape (N_A, N_B, N_steps, 3), where N_A the number of particles A, N_B the number of particles B, N_steps is the number of timesteps in the universe, and the last axis refers to the three directions in space x, y and z. For example, the array element AB[i, j, k, 2] refers to the z-component of the vector pointing from A_i to B_j at the k-th timestep of the simulation.

Example:

```python
# example_get_vectormatrix.py
import MDAnalysis
import mdorado.vectors as mvec
from mdorado.data.datafilenames import water_topology,
   ↪ water_trajectory

u = MDAnalysis.Universe(water_topology, water_trajectory)
```

```
 7 ogrp = u.select_atoms("name ow")
 8
 9 vecmat = mvec.get_vectormatrix(universe=u, agrp=ogrp, bgrp=ogrp,
   ↪   pbc=True)
10 print(vecmat.shape)
```

Using the SPC water simulation (`water_topology`, `water_trajectory`) included in MDorado, we create an `AtomGroup` containing all 125 oxygen atoms (line 7). The function `get_vectormatrix` then computes a timeseries for every oxygen-oxygen vector in the trajectory (line 9). With `pbc=True` (default), we make sure that we always consider the vector from an oxygen atom to the closest image of the other oxygen atom. Due to the size of the resulting array, we only print its shape here:

```
$ python example_get_vectormatrix.py
(125, 125, 1001, 3)
```

The first axis refers to the oxygen atom used as a starting point for the vector. The second axis refers to the second oxygen atom, the end point of the vector. The thrid axis refers to the timestep in the trajectory and the last axis to the three coordinates x, y, and z. The element `vecmat[6, 108, 515, 0]` refers to the x-component of the vector pointing from the 6th oxygen atom to the 108th oxygen atom at the 515th timestep. Due to the symmetry of the matrix in our example, it is the opposite of the element `vecmat[108, 6, 515, 0]`.

1.5 `get_vecarray`

Given to AtomGroups agrp and bgrp, it computes the time evolution of every vector bgrp[i]-agrp[i] for the whole trajectory. Can account for periodic boundary conditions for cuboid boxes.

`mdorado.vectors.get_vecarray(universe, agrp, bgrp, pbc=True)`

Parameters:

`universe`:	MDAnalysis.Universe Universe containing the trajectory.
`agrp`:	AtomGroup from MDAnalysis

	AtomGroup containing all atoms of type A for which a vector $\overrightarrow{AB}$ should be computed.
`bgrp`:	AtomGroup from MDAnalysis AtomGroup containing all atoms of type B for which a vector $\overrightarrow{AB}$ should be computed.
`pbc`:	bool, optional Specifies whether periodic boundary conditions should be applied to find the shortest vector from A to an image of B. Calls the `mdorado.vectors.pbc_vecarray` function. Only works for cuboid boxes (all angles are 90°). Default is `True`.

Returns: ndarray
Array containing the trajectory of every vector bgrp[i]-agrp[i] of the shape N_{vec} (number of vectors), N_{ts} (number of timesteps in the universe), N_{dim} (number of dimensions).

Example:

```
# example_get_vecarray.py
import MDAnalysis
import mdorado.vectors as mvec
from mdorado.data.datafilenames import water_topology,
↪   water_trajectory

u = MDAnalysis.Universe(water_topology, water_trajectory)
ogrp = u.select_atoms("name ow")
hgrp = u.select_atoms("name hw")[::2]

vectorarray = mvec.get_vecarray(universe=u, agrp=ogrp, bgrp=hgrp,
↪   pbc=False)
print(vectorarray.shape)
```

From the example SPC water simulation, we create two `AtomGroups`, one containing all oxygen atoms and the other containing every second hydrogen atom (one per water molecule). Using these and the `get_vecarray` function, we can compute a timeseries of one OH-vector per water molecule in our simulation.

The trajectories of our example simulation are “repaired” in such a way, that atoms belonging to the same molecule are an the same side of the simulation box as the center-of-mass of the molecule. This has the advantage, that intramolecular vectors are not broken up by the periodic boundary condition and we can set `pbc=False`.

The resulting array is of shape (125, 1001, 3):

```
$ python example_get_vecarray.py
(125, 1001, 3)
```

The first axis references the 125 different OH-vectors in our simulation, one per water molecule. The second axis references the timestep of the trajectory and the third axis references the three coordinates x, y, and z. The array element `vectorarray[49][957][2]` references the z-component of one OH-vectors of the 49th water molecule at the timestep number 957.

1.6 get_normal_vecarray

Computes given to AtomGroups agrp, bgrp and cgrp, it computes the normal vectors

$$\boldsymbol{n}_i(t) = [\boldsymbol{b}_i(t) - \boldsymbol{a}_i(t)] \times [\boldsymbol{c}_i(t) - \boldsymbol{a}_i(t)] \tag{1.1}$$

for every triple of atoms positions $(\boldsymbol{a}_i, \boldsymbol{b}_i, \boldsymbol{c}_i)$ and for every timestep t of the trajectory.

```
mdorado.vectors.get_normal_vecarray(universe, agrp, bgrp, cgrp,
    pbc=True)
```

Parameters:

`universe`:	MDAnalysis.Universe Universe containing the trajectory.
`agrp`:	AtomGroup from MDAnalysis AtomGroup containing all atoms of type A used to define the plane containing the atoms A_i, B_i and C_i.
`bgrp`:	AtomGroup from MDAnalysis AtomGroup containing all atoms of type B used to define the plane containing the atoms A_i, B_i and C_i.
`cgrp`:	AtomGroup from MDAnalysis

AtomGroup containing all atoms of type C used to define the plane containing the atoms A_i, B_i and C_i.

`pbc`: bool, optional
Specifies whether periodic boundary conditions should be applied to find the shortest vector from A to an image of B and C. Calls the `mdorado.vectors.pbc_vecarray` function. Only works for cuboid boxes (all angles are 90°). Default is `True`.

Returns: ndarray
An ndarray of shape N_{vec} (number of normal vectors $\vec{n}$), N_{steps} (number of timesteps in `universe`), N_{dim} (number of dimensions) containing the time evolution of all normal vectors $\vec{n} = \overrightarrow{AB} \times \overrightarrow{AC}$ in the trajectory.

Example:

```python
1  # example_get_normal_vecarray.py
2  import MDAnalysis
3  import mdorado.vectors as mvec
4  from mdorado.data.datafilenames import water_topology,
     ↪  water_trajectory
5
6  u = MDAnalysis.Universe(water_topology, water_trajectory)
7  ogrp = u.select_atoms("name ow")
8  h1grp = u.select_atoms("name hw")[::2]
9  h2grp = u.select_atoms("name hw")[1::2]
10
11 normalvectorarray = mvec.get_normal_vecarray(universe=u, agrp=ogrp,
     ↪  bgrp=h2grp, cgrp=h2grp, pbc=False)
12 print(normalvectorarray.shape)
```

To compute a vector perpendicular to the molecular plane for every water molecule (normal vector) from our example SPC water simulation at every timestep, we first have to create three `AtomGroups`. The first (line 7) contains all oxygen atoms, the second (line 8) contains every second hydrogen atom starting with the 0th (“the first” of each water molecule), an the second (line 9) also contains every second hydrogen atom but starting with the 1st (“the second” of each water molecule).

The trajectories of our example simulation are "repaired" in such a way, that atoms belonging to the same molecule are an the same side of the simulation box as the center-of-mass of the molecule. This has the advantage, that intramolecular vectors are not broken up by the periodic boundary condition and we can set `pbc=False`.

The shape of the resulting array is (125, 1001, 3):

```
$ python example_get_normal_vecarray.py
(125, 1001, 3)
```

The first axis references the 125 different water molecules in our simulation. The second axis references the timestep of the trajectory and the third axis references the three coordinates x, y, and z. The array element `normalvectorarray[120][531][0]` references the x-component of a vector perpendicular to the plane containing the three atoms of the 120th water molecule at the timestep number 531.

2 Dipolar NMR Relaxation and Correlations in the Structure and Dynamics of Molecular Liquids

The dipolar relaxation rate of an NMR active nucleus is determined by its magnetic dipolar interaction with all the surrounding nuclei. It is therefore subject to the time-dependent spatial correlations in the liquid and is affected by both the molecular structure and the dynamics of the liquid. For the NMR relaxation rate of nuclear spins with $I = 1/2$, the magnetic dipole-dipole interaction represents the most important contribution. The relaxation rate, i.e. the rate at which the nuclear spin system approaches thermal equilibrium, is determined by the time dependence of the magnetic dipole-dipole coupling. For two like spins, it is [14]

$$\begin{aligned} T_1^{-1} &= 2\gamma^4\hbar^2 I(I+1)(\mu_0/4\pi)^2 \\ & \left\{ \int_0^\infty \left\langle \sum_j^N \frac{D_{0,1}[\Omega_{ij}(0)]}{r_{ij}^3(0)} * \frac{D_{0,1}[\Omega_{ij}(t)]}{r_{ij}^3(t)} \right\rangle e^{i\omega t} \mathrm{d}t \right. \\ &+ \left. 4\int_0^\infty \left\langle \sum_j^N \frac{D_{0,2}[\Omega_{ij}(0)]}{r_{ij}^3(0)} * \frac{D_{0,2}[\Omega_{ij}(t)]}{r_{ij}^3(t)} \right\rangle e^{i2\omega t} \mathrm{d}t \right\}, \end{aligned} \tag{2.1}$$

where $D_{k,m}[\Omega]$ is the k, m-Wigner rotation matrix element of rank 2. The Eulerian angles $\Omega(0)$ and $\Omega(t)$ at time zero and time t specify the dipole-dipole vector relative to the laboratory fixed frame of a pair of spins and r_{ij} denotes their separation distance and μ_0 specifies the permittivity of free space. The sum indicates summation of all j interacting like spins in the entire system. For the case of an isotropic fluid and in the extreme narrowing limit Eq. (2.1) simplifies to [15]

$$T_1^{-1} = 2\gamma^4\hbar^2 I(I+1)\left(\frac{\mu_0}{4\pi}\right)^2 \int_0^\infty G_2(t)\,\mathrm{d}t\,. \tag{2.2}$$

The dipole-dipole correlation function here is abbreviated as $G_2(t)$ and is available through [15, 16]

$$G_2(t) \quad = \quad \left\langle \sum_j r_{ij}^{-3}(0)\, r_{ij}^{-3}(t) P_2\left[\cos\theta_{ij}(t)\right] \right\rangle \ , \tag{2.3}$$

where $\cos\theta_{ij}(t)$ is the angle between the vectors $\vec{r}_{ij}$ joining spins i and j at time 0 and at time t [15] and P_2 is the second Legendre polynomial.

To calculate the integral, we separate the correlation function $G_2(t)$ into an r^{-6}-prefactor, which is sensitive to the structure of the liquid (average internuclear distances) and a correlation time τ_2, which is obtained as the time-integral of the normalized correlation function $\hat{G}_2(t)$, and which is sensitive to the mobility of the molecules in the liquid,

$$\int_0^\infty G_2(t)\,\mathrm{d}t \quad = \quad \left\langle \sum_j r_{ij}^{-6}(0) \right\rangle \tau_2 \ . \tag{2.4}$$

The correlation function G_2 and hence T_1 can be calculated directly from MD-simulation trajectory data. From the definition of the dipole-dipole correlation function in Eq. (2.3) it follows directly that the relaxation time T_1 is affected by both, reorientational and translational motions in the liquid. Moreover, it is obvious that it also depends strongly on the average distance between the spins and is hence sensitive to changing inter- and intramolecular pair distribution functions [17, 18]. In addition, the r^{-6}-weighting introduces a particular sensitivity to changes occurring at short distances. For convenience, one may divide the spins j into different classes according to whether they belong to the same molecule as spin i, or not, thus arriving at an *inter*- and *intramolecular* contribution to the relaxation rate

$$T_1^{-1} = T_{1,\mathrm{inter}}^{-1} + T_{1,\mathrm{intra}}^{-1}, \tag{2.5}$$

which are determined by corresponding intra- and intermolecular dipole-dipole correlation functions $G_{2,\mathrm{intra}}$ and $G_{2,\mathrm{inter}}$. The intramolecular contribution is basically due to molecular reorientations and conformational changes and has been used extensively to study the reorientational motions, such as that of the H-H-vector in CH_3-groups in molecular liquids and crystals [19]. The intermolecular contributions is affected by the translational mobility (i.e. diffusion) within the liquid and the preferential aggregation or interaction between particular sites, as expressed by intermolecular pair correlation functions.

Intermolecular Contributions The structure of the liquid can be expressed in terms of the intermolecular site-site pair correlation function $g_{ij}(r)$, describing the probability of finding a second atom of type j in a distance r from a reference site of type i according to [20]

$$g_{ij}(r) = \frac{1}{N_i\,\rho_j} \left\langle \sum_{k=1}^{N_i} \sum_{l=1}^{N_j} \delta(\vec{r} - \vec{r}_{kl}) \right\rangle \ , \tag{2.6}$$

where ρ_j is the number density of atoms of type j. The prefactor of the intermolecular dipole-dipole correlation function is hence related to the pair distribution function via an r^{-6} integral of the pair correlation function

$$\left\langle \sum_j r_{ij}^{-6}(0) \right\rangle = \rho_j \int_0^\infty r^{-6}\; g_{ij}(r)\; 4\pi\, r^2\, dr \ . \tag{2.7}$$

Since the process association in a molecular solution is equivalent with an increase of the nearest neighbor peak in the radial distribution function, Eq. 2.7 establishes a quantitative relationship between the degree of intermolecular association and the intermolecular dipolar nuclear magnetic relaxation rate.

The integral in Equation 2.7, of course, contains all the structural correlations affecting the spin pairs. Averaged intermolecular distances between two spins α and β are represented by the integral

$$I_{\alpha\beta} = 4\pi \int_0^\infty r^{-6}\, g_{\alpha\beta}(r)\, r^2 \mathrm{d}r \ . \tag{2.8}$$

Following a convention in the literature, the size of the integral $I_{\alpha\beta}$ is conviently described by a “distance of closest approach” $d_{\alpha\beta}$, which represents an integral of the same size, but over a step-like unstructured pair correlation function according to

$$I_{\alpha\beta} = 4\pi \int_{d_{\alpha\beta}}^\infty r^{-6} \cdot 1 \cdot r^2 \mathrm{d}r = \frac{4\pi}{3} \cdot \frac{1}{d_{\alpha\beta}^3} \tag{2.9}$$

Hence the “distance of closest approach” can be determined with the knowledge of $I_{\alpha\beta}$ as

$$d_{\alpha\beta} = \left[\frac{4\pi}{3} \cdot \frac{1}{I_{\alpha\beta}} \right]^{1/3} \ . \tag{2.10}$$

Intramolecular Contributions: Intramolecular correlations are computed directly over all involved spin pairs of type α and β.

$$\left\langle r_{\alpha\beta}^{-6} \right\rangle^{-1/6} = \left\langle \frac{1}{N_\alpha N_\beta} \sum_i \sum_j r_{ij}^{-6} (1 - \delta_{ij}) \right\rangle^{-1/6} \ . \tag{2.11}$$

Here δ_{ij} ensures that contributions from identical spins for the case of $\alpha = \beta$ are not counted. Note that for the special case of $\alpha = \beta$ also the normalisation has to modified accordingly: $N_\beta = N_\alpha - 1$.

2.1 `dipol_correl`

Computes the dipol-dipol correlation function $G_2(t)$ for one or multiple vector trajectories.

`mdorado.dipol_relax..dipol_correl(vecarray, dt, outfilename=False)`

Parameters:

`vecarray`:	ndarray Array of shape N_{vec} (number of vectors), N_{steps} (number of steps in the trajectory), N_{dim} (dimensionality of the vectors) containing all vectors of interest for the dipolar relaxation rate .
`dt`:	int or float The difference in time between steps of the trajectory.
`outfilename`	str, optional If specified an xy-file with the name `str(outfilename)` containing the timestep and corresponding value of the correlation function. If `False` (default), no file will be written.

Returns: `timesteps`, `allcorrel`: ndarray, ndarray
Returns two arrays, the first containing information about the timestep t and the second containing the dipolar relaxation correlation function.

Bibliography

[1] D. Chandler, *J. Chem. Phys.* **1978**, *68*, 2959.

[2] C. H. Bennett in *Algorithms for Chemical Computations*, Chapter 4, p. 63.

[3] S. Gehrke, M. von Domaros, R. Clark, O. Hollóczki, M. Brehm, T. Welton, A. Luzar, B. Kirchner, *Faraday Discuss.* **2018**, *206*, 219.

[4] S. Gehrke, B. Kirchner, *J. Chem. Eng. Data* **2019**, *65*, 1146.

[5] J. Neumann, D. Paschek, A. Strate, R. Ludwig, *J. Phys. Chem. B* **2021**, *125*, 281.

[6] J. Neumann, R. Ludwig, D. Paschek, *J. Phys. Chem. B* **2021**, *125*, 5132.

[7] A. Luzar, D. Chandler, *Phys. Rev. Lett.* **1996**, *76*, 928.

[8] A. Luzar, D. Chandler, *Nature* **1996**, *379*, 55.

[9] A. Luzar, *J. Chem. Phys.* **2000**, *113*, 10663.

[10] J. Busch, J. Neumann, D. Paschek, *J. Chem. Phys.* **2021**, *154*, 214501.

[11] R. Kumar, J. R. Schmidt, J. L. Skinner, *J. Chem. Phys.* **2007**, *126*, 204107.

[12] D. Trzesniak, A.-P. E. Kunz, W. F. van Gunsteren, *ChemPhysChem* **2007**, *8*, 162.

[13] Calandrini, V., Pellegrini, E., Calligari, P., Hinsen, K., Kneller, G.R., *JDN* **2011**, *12*, 201.

[14] A. Abragam, *The Principles of Nuclear Magnetism*, Oxford University Press, **1961**.

[15] P. Westlund, R. Lynden-Bell, *J. Magn. Reson.* **1987**, *72*, 522–531.

[16] M. Odelius, A. Laaksonen, M. H. Levitt, J. Kowalewski, *J. Magn. Reson. Ser. A* **1993**, *105*, 289–294.

[17] H. G. Hertz, *Prog. NMR Spec.* **1967**, *3*, 159–230.

[18] H. G. Hertz, R. Tutsch, *Ber. Bunsenges. Phys. Chem.* **1976**, *80*, 1268–1278.

[19] E. O. Stejksal, D. E. Woessner, T. C. Farrar, H. S. Gutowsky, *J. Chem. Phys.* **1959**, *31*, 55–65.

[20] P. A. Egelstaff, *An Introduction to the Liquid State*, 2., Oxford University Press, Oxford, **1992**.