J Mol Model (2011) 17:2669–2685
DOI 10.1007/s00894-010-0948-5

ORIGINAL PAPER

Trajectory NG: portable, compressed, general molecular dynamics trajectories

**Daniel Spångberg · Daniel S. D. Larsson ·
David van der Spoel**

Received: 21 November 2010 / Accepted: 27 December 2010 / Published online: 26 January 2011

Abstract We present general algorithms for the compression of molecular dynamics trajectories. The standard ways to store MD trajectories as text or as raw binary floating point numbers result in very large files when efficient simulation programs are used on supercomputers. Our algorithms are based on the observation that differences in atomic coordinates/velocities, in either time or space, are generally smaller than the absolute values of the coordinates/velocities. Also, it is often possible to store values at a lower precision. We apply several compression schemes to compress the resulting differences further. The most efficient algorithms developed here use a block sorting algorithm in combination with Huffman coding. Depending on the frequency of storage of frames in the trajectory, either space, time, or combinations of space and time differences are usually the most efficient. We compare the efficiency of our algorithms with each other and with other algorithms present in the literature for various systems: liquid argon, water, a virus capsid solvated in 15 mM aqueous NaCl, and solid magnesium oxide. We perform tests to determine how much precision is necessary to obtain accurate structural and dynamic properties, as well as benchmark a parallelized implementation of the algorithms. We obtain compression ratios (compared to single precision floating point) of 1:3.3–1:35 depending on the frequency of storage of frames and the system studied.

Keywords Molecular dynamics · Trajectory · Compression

D. Spångberg (✉)
Uppsala Multidisciplinary Center for Advanced Computational Methods (UPPMAX) and Department of Materials Chemistry, Uppsala University,
Box 538, SE-751 21 Uppsala, Sweden
e-mail: daniels@mkem.uu.se

D. S. D. Larsson · D. van der Spoel
Department of Cell and Molecular Biology, Uppsala University,
Box 596, SE-751 24 Uppsala, Sweden

When large-scale molecular dynamics (MD) simulations are performed with highly optimized software packages such as GROMACS [1], Desmond [2], or NAMD [3], the files containing the resulting trajectories can become prohibitively large. The trajectories are usually stored on disk for later analysis and archival. The standard ways to store MD trajectories are either in text format (possibly compressed using standard lossless tools such as gzip [4] or bzip2 [5]) or by writing each value as a single-precision 32-bit floating-point number (lossless tools compress these files poorly). However, the accuracy of each value, even in single precision, is usually larger than necessary for the analysis. Therefore, by applying lossy compression, it is possible to store the trajectories more efficiently. Also, by correlating different parts of the trajectory, it is possible to obtain even more efficient storage. Trajectory formats that use these ideas are the XTC format [6] and the BS-VLC format [7]. In the XTC format, the fact that atoms belonging to the same molecules are usually stored in order is used, while in the BS-VLC format the fact that an atom in one frame of the trajectory (i.e., at time t) is usually close to itself in the next frame (at time $t+\Delta t$) is used. In the XTC format, it is—for an accuracy of about 10^{-2} Å—usually possible to store a value in 9–10 bits, which corresponds to compression ratios of 1:3.2–3.6 compared to 32-bit floating point. In the BS-VLC format, the compression ratio depends on the time between each frame, but for a common frame storage frequency of one frame per 1000 timesteps, compression ratios of up to 1:4 can be obtained.

Another approach, where compressing differences in the atomic coordinates is not performed directly, is to use the essential dynamics algorithm to compress trajectories [8]. In this algorithm, the covariance matrix of Cartesian atomic positional fluctuations is diagonalized, and the resulting eigenvectors and eigenvalues stored only if they make important contributions to the positional variance. In this approach, compression ratios of up to 1:71 can be obtained for DNA motion if errors (RMSD) in the coordinates of 0.6 Å can be tolerated. If even larger errors are acceptable, interpolation of the motion can give another factor of 10 in the compression ratio, albeit for coordinate errors of 0.8 Å. DNA motion is, however, rather uncomplicated compared to the systems studied here, which makes the essential dynamics approach less generic. Also, the errors obtained using this method are not acceptable for many applications.

The XTC and BS-VLC formats are used to store atomic positions only, not velocities. Here, we extend the ideas of those formats: first by using different algorithms depending on how frequently the frames are written to the trajectory file (i.e., how large Δt is); second by improving on the algorithms used themselves; and third by also applying similar algorithms for the lossy compression of atomic velocities.

Due to the discretization of the coordinates, the maximum size of the objects that can be stored with the new format depends on the precision of the coordinates stored. With a largest absolute deviation of 5×10^{-3} Å, the largest object that can be stored is about 0.5 mm in size.

We have implemented the compression algorithms in a library, and have made this library publicly available [9]. Appendices A and B contain additional information about our implementation choices. The routines in this library are callable from C, C++, and Fortran.

Method

Floating point inputs

Each coordinate or velocity value from a molecular dynamics simulation can be written as x_{ijk}, where i represents the dimension, j is the number of the coordinate (atom index) within each frame of the trajectory, and k is the number of the frame in the trajectory. Each such coordinate is stored during the molecular dynamics simulation as either a single-precision or a double-precision floating-point number (typically 32 and 64 bit, respectively).

Quantization

A user-defined precision, p, is defined for each coordinate/velocity value. A coordinate or velocity value, x_{ijk}, is converted to an integer, e_{ijk}, by rounding to the nearest multiple of the precision:

$$e(x) = \left[\frac{x}{p}\right]. \tag{1}$$

The largest error in each coordinate/velocity is $\frac{p}{2}$. By performing the quantization early, cumulative errors in later parts of the compression chain, which can occur in the BS-VLC format, are completely avoided. The quantization step is the only source of loss of precision in the sets of algorithms presented here. All further steps in the algorithms are completely lossless.

Delta coding

Three different integer sequences are generated in our compressed trajectory format—one is a direct sequence and two are delta-coded sequences:

Sequence 1 is a straightforward one-to-one application of Eq. 1. This sequence is termed "one-to-one" in the discussions below:

$$e_{ijk} = e\left(x_{ijk}\right). \tag{2}$$

Sequence 2 is a delta of integers within each frame (intraframe delta):

$$e_{ijk} = \begin{cases} e\left(x_{ijk}\right) & \text{if } j = 1 \\ e\left(x_{ijk}\right) - e_{i(j-1)k} & \text{if } j \neq 1 \end{cases} \tag{3}$$

Sequence 3 is a delta of integers between each frame (interframe delta):

$$e_{ijk} = \begin{cases} e\left(x_{ijk}\right) & \text{if } k = 1 \\ e\left(x_{ijk}\right) - e_{ij(k-1)} & \text{if } k \neq 1 \end{cases} \tag{4}$$

These sequences are then used in the subsequent compression steps.

Boundaries

The largest negative and largest positive integers in a sequence are determined. These values are used to convert to positive integers and to select appropriate bases that are large enough to store all values. The boundary values are computed for every frame or for a relatively small number of frames, so they need not be computed for the whole trajectory.

Converting to positive integers

In the original XTC format [6] and our BWLZH algorithm (cf. Sect. 1.6.5), positive integers are obtained from the negative integers by simply subtracting the largest negative value in each sequence:

$$e_{\text{positive}} = e - e_{\text{largest negative}}. \tag{5}$$

In some of the compression formats used here, a bias toward small positive integers is preferable, so positive integers are instead obtained for all algorithms (except BWLZH) as follows:

$$e_{\text{positive}} = \begin{cases} 1 + 2(e-1) & \text{if } e > 0 \\ 2 - 2(e+1) & \text{if } e \leq 0 \end{cases} \tag{6}$$

Compressing integer sequences

Five compression algorithms have been developed and tested here. They differ primarily in how the integer sequences are handled. In the stop-bits coding, each integer value is compressed separately. In the variable base coding and the XTC2 and XTC3 extensions of the XTC format, groups of three values (triplets, i.e., one atomic coordinate) are considered together. In the BWLZH coding, a large sequence of values (up to 200,000) are handled together. In the XTC3 format, additional compression is obtained by performing a BWLZH compression step as well. The different integer sequences are combined with different compression algorithms, resulting in a larger number of combinations. We have restricted our tests to the ten combinations that are defined below. The compression algorithms are each described in the sections below.

Algorithm 1: stop-bits coding

Stop-bits coding requires one parameter, n, which determines the initial value of how many bits are stored before the next stop bit. For each value, the following algorithm is applied:

1. The number of bits, m is set to the parameter n.
2. If the value fits into m bits, a zero-extended value m bits long is stored.
3. The just stored value is subtracted from the value, which is shifted m positions to the right.
4. If there are no more one bits in the value, a zero bit is stored and no further bits are stored for this value.
5. If there are still one bits in the value, a one bit is stored. A new number of stop bits, m, is computed as $\max\left(1, \left[\frac{m}{2}\right]\right)$, and steps 2, 3, etc. are repeated.

The number of bits required to store a value depends on the parameter n, but if n is chosen appropriately, this algorithm can be used to rather efficiently store large values and somewhat smaller values together. For instance, to store the value 5, when the parameter n is set to 3, the following bits are stored: 1010 (i.e., the value 5 followed by a zero bit, since the value 5 fits into three bits). To store a value of 9 with n set to 3, the following bits are stored: 001110 (i.e., the three least significant bits in the value 9, followed by a one bit signifying that more bits remain). The value m is then set to 1, and the final remaining one bit in 9 is stored, followed by a final zero bit, since there are no more remaining bits.

Algorithm 2: variable base coding

In variable-base coding, three values are stored together. An integer parameter, n, is first chosen to determine the smallest base that the three values are stored in. Two bits are reserved to choose the base for the three particular values, so four different bases can be used for different value triplets. Three of these bases are 2^n, 2^{n+1}, and 2^{n+2}. The last base, 2^m, where m is an integer, is chosen such that the largest possible integer can be stored. The use of two prefix bits results in an overhead of $\frac{2}{3}$ bits per stored value. For instance, if the value of the parameter n is 3 and the three values 5, 9, and 4 are to be stored, the three values fitted do not fit in the base 2^n, but they do fit in the base 2^{n+1}, so the following bit sequence is stored: 01010110010100. The first two bits indicate the base 2^{n+1}, and the three following quartets of bits encode the values 5, 9, and 4, respectively.

The XTC format

In the original XTC format [6], triplets of values are stored together. These triplets can be either large or small. The large values are stored in a base, which is predetermined to be large enough to store any value in the sequence. The three values, either large or small, are stored together as a single integer calculated as $b_x b_y z + b_x y + x$, where the bases b_x to b_z of the large values are different, to allow for efficient storage of anisotropic geometries. Storing the three values together in one integer allows for fewer bits to be used per value. The small value triplets, which are just changes in position, are stored with the same base $b_x = b_y = b_z$. The bases here are integer powers of $2^{1/3}$ rounded to the nearest integer, which allows storage with a resolution of $\frac{1}{3}$ bits. In the format, a large-value triplet is followed by a number of small-value triplets. The large values are one-to-one conversions of the atomic coordinates, while the small values are the position deltas with respect to the previous coordinate in the same frame. To allow even more efficient storage of water molecules, which is common in simulations of many chemical systems, and particularly in simulations of biological systems, the coordinates of the first two atoms in each large-small sequence are swapped. This is more efficient, since the distance between one hydrogen of the water molecule and the other hydrogen is longer than the distances between the oxygen and the hydrogen atoms.

Two integer variables are used to keep track of the number and base of small values. The first variable is the runlength, r, which can be between 0 and 8, allowing up to 9 (1 large+8 small) triplets to be stored together. The second variable, n, is an index for an array of allowed bases for the small values.

In this format, a single bit is used to indicate whether the *same* runlength and base as used previously should also be used for the small values, or whether a change is required. If the bit is zero, the values for runlength and base are kept the same as before. If the bit is one, the following 5-bit value, v, is used to alter the runlength and base. The new base index, n, is set to $n + [v \bmod 3] - 1$. The new runlength is calculated as $r = v - ([v \bmod 3] - 1)$. This format allows very efficient storage of regular, repeated structures, such as long sequences of water molecules.

Ideally, only one additional bit is used per molecule, unlike variable-base coding, where two bits are required per atom ($\frac{2}{3}$ bits per value). Also, up to $\frac{2}{3}$ bits per stored value can be saved by encoding several values into the same integer. In total, for a molecule with four atoms, about one bit per value can be saved by using this format rather than variable-base coding.

Algorithm 3: XTC2

The XTC2 coding developed here is based on the XTC format, so only the differences from the XTC format are described here.

In the XTC format, a large-value triplet is always followed by a number of small-value triplets (although the number of small-value triplets can be zero). In the bit stream there are two codes, allowing for two possibilities for the following sequence of large + small triplets: keep the previous settings (1 bit) or alter runlength and base (1+5 bits). Two integer variables (n and r) are used to track the runlength and base.

In the XTC2 coding, the codes have been modified and several other possible codes have been added. The largest possible runlength of small triplets is 6 rather than the 8 possible in the XTC format. A boolean variable, f, has been added to track whether the coordinates of the first two atoms should be swapped or not. The following codes are possible:

– A triplet of large values followed by runlength-encoded small triplets, no change in settings: one bit (1).
– Set base index and runlength: 2 bits (00)+4 bits (v). If v is 1111, the runlength is set to 6 and the base index is not changed. Otherwise the new base index is set to $n + (v \bmod 3) - 1$, and the new runlength is set to $1 + \left[\frac{v}{3}\right]$.
– A triplet of large values not followed by any small triplets, no change in settings: 4 bits (0100).
– Runlength-encoded small triplets, no change in settings: 4 bits (0101).
– Set base index: 4 bits (0110)+2 bits (v). The new base index is set to $n + [(v \bmod 2) + 1] \cdot -1^{[v/2]}$.
– Flip the boolean variable f that determines whether the first two atoms should be swapped or not: 5 bits (01110).
– Runlength-encoded large triplets, no change in settings: 5 bits (01111)+4 bits (v). The number of following triplets is v+3.

Algorithm 4: BWLZH

Burrows–Wheeler–Lempel–Ziv–Huffman (BWLZH) coding is based on the block-sorting algorithm of Burrows and Wheeler [10]. In our implementation, it is combined with Lempel–Ziv coding [11] and finally Huffman coding [12]. The algorithm has the following steps:

1. The integer sequences are formed in memory with the frame number, k, the fastest varying followed by the dimension, i, and finally the atom index, j.
2. For each integer sequence, the 32-bit integers are converted to a sequence of 16-bit integers. As long as there are nonzero bits left in the 32-bit integer: if the integer is larger than 32767 the least significant 15 bits+ 32768 are stored and the integer is shifted 15 positions to the right; otherwise the integer itself is stored. Thus, each 32-bit integer is converted to up to three 16-bit integers.
3. Each 16-bit integer sequence is divided into blocks of up to 200,000 values. This is a compromise between compression ratio and compression speed. The compression speed drops substantially for large blocks, primarily due to cache misses.
4. The Burrows–Wheeler transform (BWT) [10] is applied to the block of up to 200,000 values. Our implementation uses an initial step to detect repeating sequences of integers with a recurrence of up to 8 values. A merge sort with a comparison function using the information about repeating integer sequences is then applied to perform the block sorting. This works well for our data.
5. The sequence resulting from BWT is divided into two streams, one containing the most significant bytes, the other the least significant bytes. These streams are then processed individually. Some of the correlation in the data is lost due to this division, typically increasing the final file sizes by 1–2%. However, the move-to-front step in particular runs significantly faster, decreasing the compression and decompression times by a factor of three.
6. Move-to-front coding [13] is applied to each data stream.

7. In the original Burrows–Wheeler report, and in common implementations such as bzip2, a runlength-encoding (RLE) step is done at this stage. However, we find that our data often contain somewhat more complicated repetitive sequences, which is why we apply a Lempel–Ziv-77 (LZ77) compressor at this stage [11]. Our LZ77 compressor generates three data streams: table, length, and offset. The first is the table stream. A single nonrepetitive input data value is stored in the table stream as the value+2. A repetitive sequence with an offset of −1 is stored in the table stream as the value 0, and a length (up to 65535) is stored in the length stream. A repetitive sequence with an offset that is different from −1 (up to −65535 is supported) is stored in the table stream as the value 1, the length is stored in the length stream, and the offset in the offset stream. Note that in some cases the length stream or the length and offset streams may not contain any values.
8. Up to three streams (table, length, and offset) are encoded separately using a Huffman coder capable of handling values from 0 to 65537. The largest Huffman code length is restricted to 31 bits, to ensure that each value fits into an integer. It is, however, very unlikely that Huffman codes that long would result from common data.

Algorithm 5: XTC3

The XTC3 coding developed here is an extension of the XTC2 format and a combination with the BWLZH algorithm. In the XTC3 coding, the different kinds of data generated are separated into separate data streams. The data streams are all individually compressed with the BWLZH algorithm (steps 2 and forward), or by using base compression as in the XTC2 format. In XTC/XTC2 coding there are two ways to describe the coordinate data: either the actual coordinate of the atom (a large triplet) or intraframe delta coding (runlength-encoded small triplets). In XTC3 coding, the runlength-encoded small triplets are handled in the same way as in the XTC2 format, but for the large triplets there are three possible ways to describe the data: either the actual coordinate of the atom (the same as in the XTC/XTC2 formats), an intraframe delta to the previous atom (either a large or small triplet), or an interframe delta of the atomic position compared to the previous position of the same atom. These three different ways to store the large triplets are stored in three different data streams. Thus, there are in total six streams of data: the codes describing the actual data, the runlengths, the three different streams of the large triplets, and the stream of small triplets. A variable, s, has been added to keep track of the current type of large stream.

The following codes are used to describe the data:

0	A triplet of large values followed by runlength-encoded small triplets, no change in settings.
1	Set the runlength of small triplets. The runlength, r, is stored in a separate data stream. For performance reasons the largest runlength allowed has been set to 12.
2	A triplet of large values not followed by any small triplets, no change in settings.
3	Runlength-encoded small triplets, no change in settings.
4	Flip the boolean variable, f that determines whether the first two atoms should be swapped or not.
5	Runlength-encoded large triplets, no change in settings. The runlength is stored in a separate data stream. For performance reasons the largest runlength allowed has been set to 1024.
6	Change the variable s encoding the type of large triplet to "actual coordinate of atom."
7	Change the variable s encoding the type of large triplet to "intra frame delta coding."
8	Change the variable s encoding the type of large triplet to "inter frame delta coding."

The XTC3 format should be the best format when the atoms in the molecules have moved further (in time) than typical intramolecular distances, but when there are still correlations in atomic positions in time. Note that this may differ per atom in an inhomogeneous system. The streams of data are compressed here using either the BWLZH format or base compression; whichever is most efficient for each data stream. Among the tests performed here, most of the streams in the XTC3 format end up being compressed by BWLZH coding, although the stream with the actual directly stored coordinate data is usually slightly better compressed with base compression. The base compression is substantially faster than the BWLZH compression. For this reason we have, for some systems, also performed tests of the XTC3 algorithm using only base compression for the coordinate streams.

Sequences and algorithms

Table 1 shows the ten different combinations of integer sequences and compression algorithms tested here.

Blocks of frames

When the interframe delta sequence is used, time-dependent frames are generated; i.e., to decompress frame n, one needs to decompress all frames 1 to $n-1$. This is inconvenient, for example, if one is interested in the properties of only the last half of a trajectory. For this reason, the trajectories are divided into blocks of frames, where the first frame of every block is encoded with the

Table 1 Combinations of sequences and compression algorithms used for the compression of coordinates and velocities

Sequence, algorithm	Coordinates	Velocities
One-to-one, stop bits	No	Yes
One-to-one, variable base	No	Yes
One-to-one, XTC2	Yes	No
One-to-one, XTC3	Yes	No
One-to-one, BWLZH	No	Yes
Intraframe, variable base	Yes	No
Intraframe, BWLZH	Yes	No
Interframe, variable base	Yes	Yes
Interframe, stop bits	Yes	Yes
Interframe, BWLZH	Yes	Yes

one-to-one or the intraframe delta sequences. The file offset for the start of each block of frames is stored in the trajectory file or in a separate index file. This allows fast random access to different parts of the trajectory. The number of frames in each block is controlled by a parameter, C.

Parallelization

The compression can take a long time, especially when the BWLZH algorithm is used. For this reason we have parallelized the compression routines. The parallelization is coarse grained, compressing each block of frames separately. When N processes are used in a simulation and there are C frames in each block, compression is performed once CN frames have been collected in memory. The parallelization is done in three steps:

1. *Collection of frames*. All frames, C in each block, are collected into the N processes with ranks numbered from 0 to $N-1$. A frame numbered i is collected at rank $\left[\frac{i}{C}\right]$. In a replicated data MD code, all ranks have identical information about all coordinates, so this is a simple local memory copy operation. In a nonreplicated data MD code where not all of the processes have identical information, such as when the domain-decomposition technique is used, a gather operation from all processes to rank $i \bmod C$ is required. Information about the global identity of each particle coordinate/velocity is also required in this case.
2. *Parallel compression*. All processes compress their individual blocks of frames.
3. *All processes send the compressed frames to rank 0, which writes the data to disk*. It is possible to perform this step in parallel with parallel I/O, but the relatively small size of the compressed data makes this operation quick compared to the compression time, meaning that parallel I/O is not usually necessary. However, for large systems, I/O can become the bottleneck; see for instance [14].

Results and discussion

Examples

The following molecular dynamics simulations have been performed:

1. *Liquid argon*: 10,000 argon atoms in a cubic cell. A standard Lennard–Jones potential was used [15] with a spherical cutoff of 9 Å. The NVT ensemble was used with a density of1.42 g cm^{-3}, and the temperature was kept at an average of 84 K using a Nosé–Hoover thermostat. The velocity Verlet integrator [16] was used with a 5 fs time step. The simulation was equilibrated for 10 ps and the production run was 0.5 ns.
2. *Liquid water*: 1000 TIP4P [17] water molecules in a cubic cell. The interactions were truncated using a spherical cutoff of 9 Å. The NVT ensemble was used with a density of 1 g cm^{-3}, and the temperature was kept at an average of 300 K using a Nosé–Hoover thermostat [18, 19]. The velocity Verlet integrator was used with a 1.5 fs time step. The RATTLE algorithm [20] was used to keep the water molecules rigid, and the method of Ciccotti [21] was used to treat the massless site of the TIP4P model. The simulation was equilibrated for 20 ps, and the production run was 150 ps.
3. *Solid magnesium oxide*: 1372 magnesium and 1372 oxygen ions modeled using the shell model of Harding and Harker [22]. The NμT ensemble (constant stress, constant temperature) was used with an initial density of 3.58 g cm^{-3}, and the temperature was kept at an average of 300 K using a Nosé–Hoover thermostat. The Cleveland modification [23] of the Parrinello–Rahman [24] equations kept the stress constant. Coulomb interactions were computed using Ewald lattice sums [15]. A spherical cutoff of 9 Å was used for the short-range interactions. The velocity Verlet integrator was used with a timestep of 0.5 fs. The method of adiabatic dynamics was used to solve for the shell positions [25]. The simulation was equilibrated for 5 ps and the production run was 75 ps.
4. *Virus capsid*: the capsid of the satellite tobacco necrosis virus [26] solvated in TIP3P [17] water with 15 mM NaCl. In total 1,005,865 atoms (198,480 protein atoms, 806,451 water atoms, 934 ions). The temperature was kept at an average of 300 K and the pressure at 1 bar using Berendsen weak coupling [27]. The integration time step was 5 fs and the SETTLE algorithm [28] was

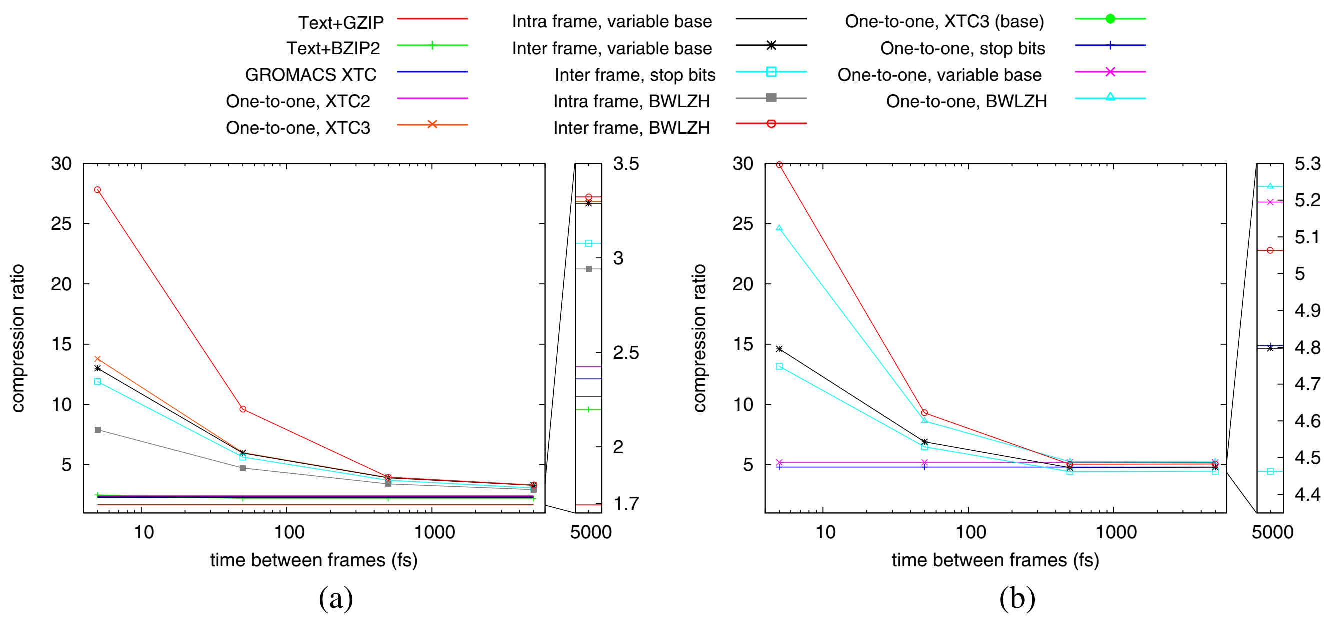

Fig. 1 **a**–**b** Compression ratio versus 32-bit single-precision values for the different compression algorithms for the liquid argon trajectory: **a** the coordinates rounded to the nearest 0.01 Å; **b** the velocities rounded to the nearest 0.1 Å/ps. In both **a** and **b**, the compression ratio when a frame is stored every 1000 steps is shown magnified

used to keep the water molecules rigid, and the P-LINCS algorithm [29] was used to keep all bond lengths constant. The position of the hydrogen bonds were calculated based on the heavy atoms or constrained with the P-LINCS algorithm in such a way that the bond angle was constant relative to the heavy atoms. Short-range non-bonded pair interactions (Lennard–Jones and electrostatic) were computed within a spherical cut-off of 1.15 nm, and long range electrostatic interactions were computed using Ewald lattice sums. The solvated starting structure was energy minimized, and the solvent was equilibrated, during which the atoms of the capsid were positionally restrained. The production run used for the analysis was calculated in parallel on 96 CPUs.

Compression

Figures 1, 2, 3 and 4 shows the compression ratios for the atomic coordinates and velocities for the different algorithms as a function of time between each frame. Appendix C contains detailed information about the compression results. The precision, p, is for the coordinates set to round to the

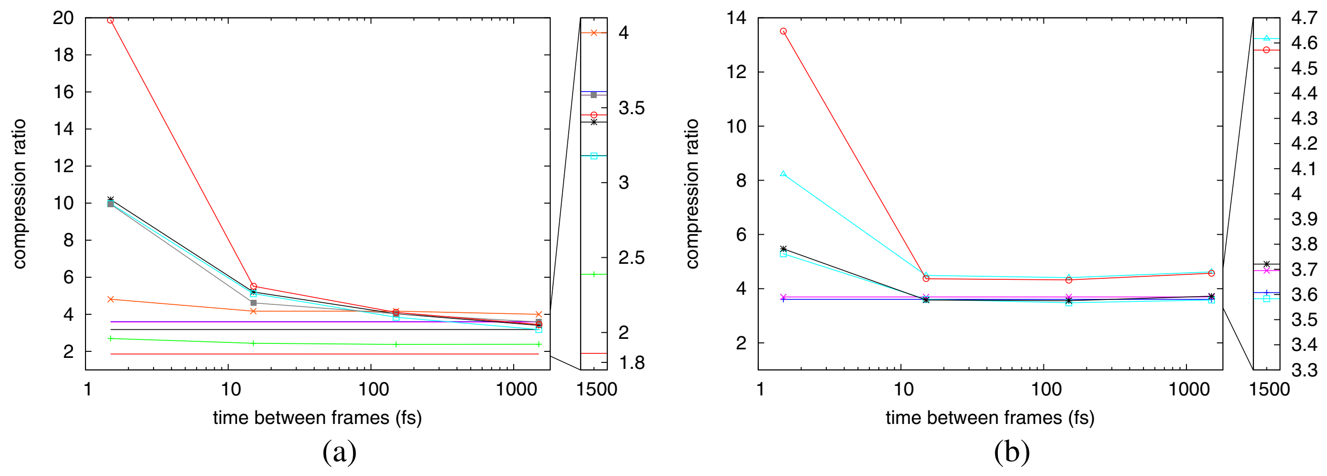

Fig. 2 **a**–**b** Compression ratio versus 32-bit single-precision values for the different compression algorithms for the liquid TIP4P water trajectory: **a** the coordinates rounded to the nearest 0.01 Å; **b** the velocities rounded to the nearest 0.1 Å/ps. See Fig. 1 for legend

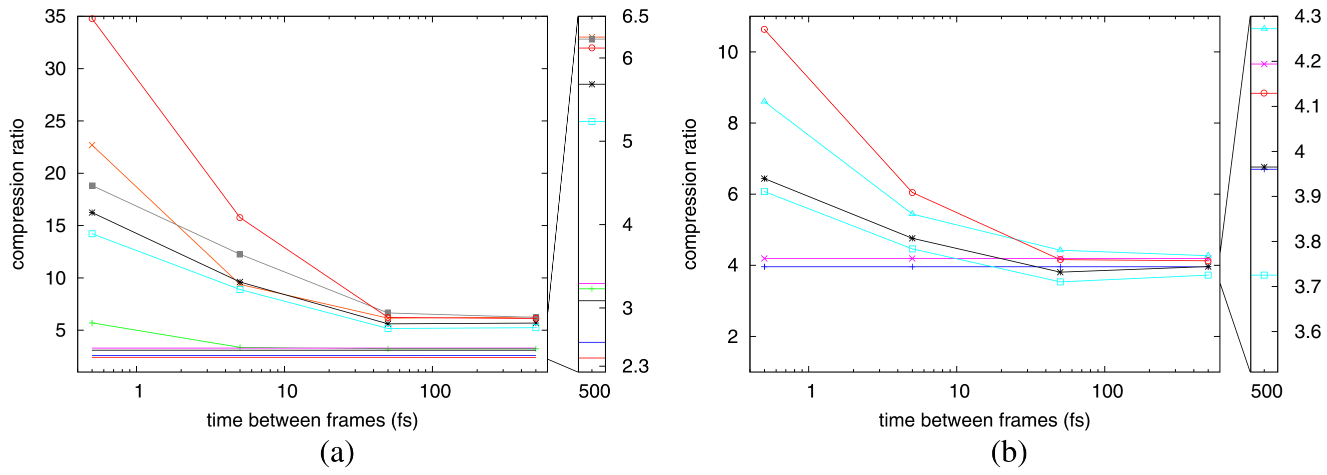

Fig. 3 **a**–**b** Compression ratio versus 32-bit single-precision values for the different compression algorithms for the solid MgO trajectory: **a** the coordinates rounded to the nearest 0.01 Å; **b** the velocities rounded to the nearest 0.1 Å/ps. See Fig. 1 for legend

nearest 0.01 Å and for the velocities to round to the nearest 0.1 Å/ps. The ratio is the compressed size against 32-bit single-precision floating-point values. For comparison, the results obtained when compressing text files, where the atomic coordinates are written with the same precision, using gzip and bzip2 at the highest supported compression (flag −9), are shown as well.

When storing a frame every timestep, the compression ratios for the coordinates range from 1:20 to 1:35. The interframe delta coding and the BWLZH algorithm is the best combination for this frequency. When storing frames more seldomly, the compression ratio goes down, and when a frame is stored every 1000 timesteps, the ratio is between 1:3.3 and 1:6.2. At this frequency, different algorithms are the most efficient for the different systems. For liquid argon, the interframe delta and the BWLZH algorithm are still the most efficient, for TIP4P the XTC3 coding is the most efficient, for MgO, the XTC3 coding and the intraframe coding together with the BWLZH algorithm are about as good as each other, while for the virus simulation the XTC3 coding is the most efficient, even when the XTC3 algorithm is restricted to base compression. For comparison, the ratios obtained with gzip range from 1:1.5 to 1:2.4, and for bzip2 from 1:1.9 to 1:5.7. For all cases studied here, at least one of the the algorithms developed here is better than both gzip and bzip2, and each of these is always worse than at least one of the algorithms developed here.

The compression ratios of the velocities range from 1:11 to 1:30 when a frame is stored every timestep with the best algorithm: the interframe delta and the BWLZH algorithm. When a frame is stored every 1000 steps, the ratio varies from 1:3.7 to 1:5.2, and the one-to-one coding and the BWLZH algorithm are the best. For the virus, the one-to-one coding and the variable base algorithm are the best.

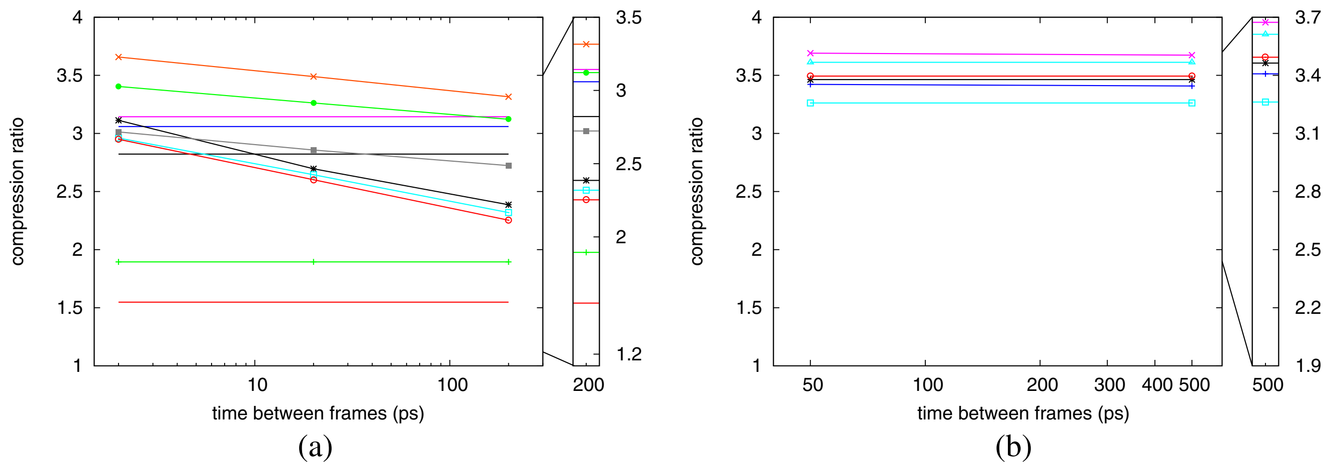

Fig. 4 **a**–**b** Compression ratio versus 32-bit single-precision values for the different compression algorithms for the virus-in-water trajectory: **a** the coordinates rounded to the nearest 0.01 Å; **b** the velocities rounded to the nearest 0.1 Å/ps. See Fig. 1 for legend

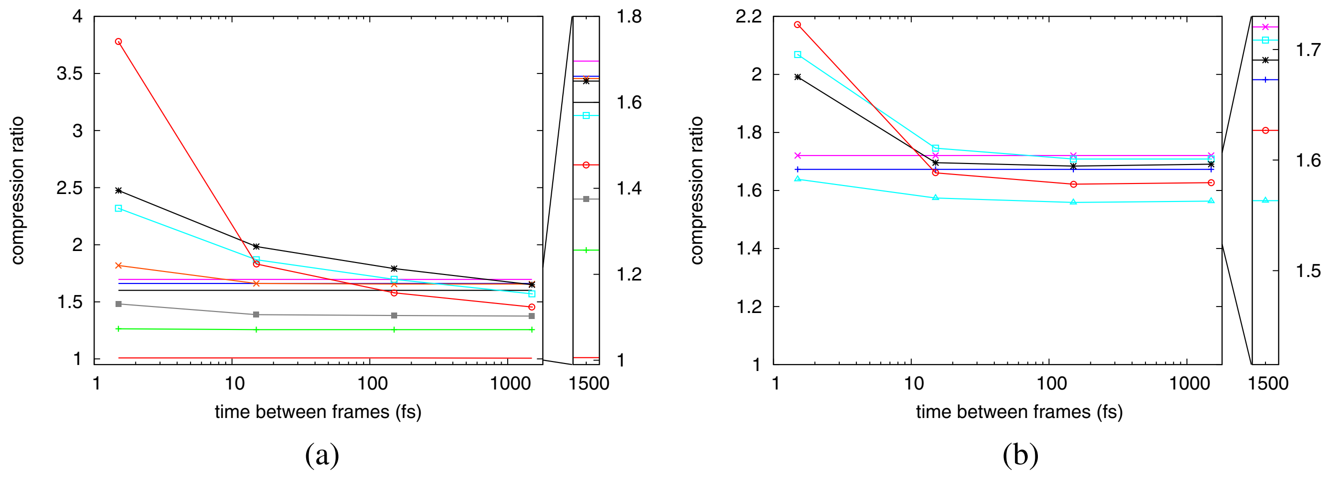

Fig. 5 Compression ratio versus 32-bit single-precision values for the liquid TIP4P water trajectory for the different compression algorithms when high precision is used to store the coordinates and velocities. The coordinates and velocities are rounded to the nearest 10^{-5} Å and 10^{-4} Å/ps, respectively. See Fig. 1 for legend

Figure 5 shows compression ratios for the simulation of TIP4P water when very high accuracy is used in storing the coordinates and velocities, although this is not usually needed; see Sect. 3 and Fig. 6. For the coordinates, compression ratios of between 1:1.7 and 1:3.8 can be obtained, while the precision obtained is about the same as the one with single precision. For the velocities, compression ratios of between 1:1.7 and 1:2.3 are obtained, while the precision is somewhat less than that obtained from single-precision floating point.

Considering all the tests performed for the different systems, the different accuracies used, and the different times between the frames, the only two algorithms among those developed here that are never the best are the intraframe delta sequence together with the variable-base coding and the one-to-one sequence together with the stop-bits coding. For each of the other eight algorithms, there is at least one case where it is the best choice. Appendix A contains information about how the optimization algorithms are chosen in our implementation.

Parallelization

Table 2 shows the compression timings for the compression of the liquid argon model containing 10,000 argon atoms for 100,000 timesteps simulated using a domain decomposition code and written every timestep. The resulting trajectory file is about 900 MB in size, which should be

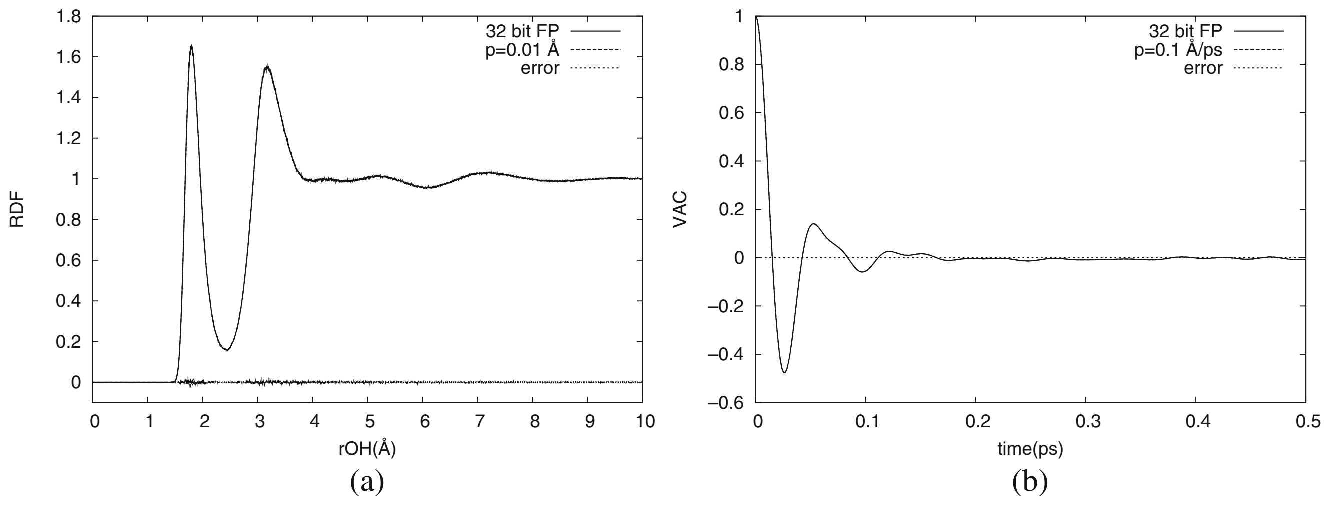

Fig. 6 a–b Comparison of properties: the O...H radial distribution function (**a**) and the hydrogen velocity autocorrelation function (**b**) for the TIP4P water trajectory. The properties were computed from the original single-precision trajectory and the trajectory compressed with lossy compression with coordinate values rounded to the nearest 0.01 Å and velocity values rounded to the nearest 0.1 Å /ps. The difference in the properties is also shown

Table 2 The compression times per frame (t) as a function of the number of cores (N) for the compression of both positions and velocities for 100,000 frames each containing 10,000 argon atoms. The interframe delta and the BWLZH algorithm were used for both the positions and the velocities. The scaling results are also shown, where $S=t/t(N=1)$

N	$t(s)$	S
1	0.11	1.0
8	0.015	7.3
16	0.0088	12.5
32	0.0049	22.4
64	0.0024	45.8

compared to the ~22 GB file that would result from storing the positions and velocities in single precision. The tests were run on a cluster of dual quad-core Intel X5520 nodes with an Infiniband DDR interconnect.

Properties from compressed trajectories

The compression algorithms presented here are lossy (i.e., the least significant digits are removed from each value; see Sect. 1.2). The accuracy to choose for the compressed trajectory depends on which properties are to be computed from the trajectory. Here, we present two examples of properties computed from the compressed trajectories: the radial distribution function, and the velocity autocorrelation function. Figure 6 shows the intermolecular O...H radial distribution function and the hydrogen velocity autocorrelation function for the TIP4P water trajectory, stored to a precision of 0.01 Å for the coordinates and 0.1 Å/ps for the velocities, as compared to the single-precision trajectory (a precision of about 10^{-5} Å or Å/ps). The difference (i.e., the error) is shown as well. The error is very small in both cases. Thus, storing trajectories in compressed form has no practical effect on the properties calculated here.

Conclusions

We have developed and tested different trajectory compression algorithms and performed tests of these for four different MD simulations, from solid magnesium oxide over liquid argon and water to biomolecular systems. Which compression algorithm is optimal depends on the chemical system, how often the frames are stored, as well as the precision required. The compression ratio strongly depends on how often the frames are stored. If frames are stored seldomly, the algorithms that use differences in atomic coordinates within each frame are the most optimal, and the lower bound of the compression ratio is determined by this case. When the precision in positions is set to 0.01 Å and the precision in velocities is 0.1 Å/ps, the compression ratios obtained for the coordinates and velocities when frames are rarely stored lie between 1:3.3 to 1:6.2. When one frame is stored every timestep, the ratio goes up to between 1:20 and 1:35 for the coordinates and between 1:11 and 1:30 for the velocities. The accuracies of these trajectories are found to be large enough to not have any practical impact on the results for the radial distribution function and hydrogen velocity autocorrelation function. Even when a very high accuracy is desired, compression ratios of at least 1:1.7 can be obtained.

Acknowledgements The computations were performed on resources provided by the Uppsala Multidisciplinary Center for Advanced Computational Science (UPPMAX), and resources provided by SNIC through the National Supercomputer Centre (NSC).

Appendix

A. Automatic selection of optimal compression algorithms

The optimal compression algorithm to use depends on the system simulated and the frequency with which frames are written to the trajectory file. The first time a block of frames is to be compressed and written to disk, we run a test of all compression algorithms and choose the one that gives the smallest compressed size. This test is performed only once, so all subsequent blocks are compressed using the same compression algorithm as initially determined. The selection of algorithms to include in the test is controlled by a parameter to the library routines.

B. Portable storage

Our implementation writes all integers with the least significant byte first, making the file format essentially little endian. However the file format is completely portable, since all external (I/O) references in our implementation are done using individual bytes only. This means that any system endianness—either big, little, or mixed—is handled portably. Also, we never store floating point values, only properly scaled fixed point numbers (integers). All text stored in the file is written as ASCII (automatic conversion to/from the source encoding is performed).

C. File sizes

Tables 3, 4, 5, 6, 7 and 8 show the raw file sizes from the simulation trajectories compressed with the different algorithms.

Table 3 Trajectory file sizes in bytes from the liquid argon simulation trajectory. The results for different compression algorithms are shown. For comparison, the uncompressed trajectories where values are stored as 32-bit floats are also shown

Frames in each block	Position algorithm	Velocity algorithm	Time between each frame	Number of frames	Position precision (Å)	Velocity precision (Å/ps)	File size (10^6B)
–	32-bit float	–	5.0 fs	1000	$\approx 10^{-5}$	–	120
–	32-bit float	32-bit float	5.0 fs	1000	$\approx 10^{-5}$	$\approx 10^{-5}$	240
–	Text	–	5.0 fs	1000	0.01	–	176
–	Text, gzip	–	5.0 fs	1000	0.01	–	70.9
–	Text, bzip2	–	5.0 fs	1000	0.01	–	48.1
1	XTC	–	5.0 fs	1000	0.01	–	50.8
1	Intraframe, var. base (n=11)	–	5.0 fs	1000	0.01	–	52.9
1	One-to-one, XTC2	–	5.0 fs	1000	0.01	–	49.5
1	Intraframe, BWLZH	–	5.0 fs	1000	0.01	–	54.9
1	One-to-one, XTC3	–	5.0 fs	1000	0.01	–	49.5
1	One-to-one, XTC2	One-to-one, stop bits (n=4)	5.0 fs	1000	0.01	0.1	74.5
1	One-to-one, XTC2	One-to-one, var. base (n=4)	5.0 fs	1000	0.01	0.1	72.6
1	One-to-one, XTC2	One-to-one, BWLZH	5.0 fs	1000	0.01	0.1	72.5
100	Interframe, stop bits (n=1)[a]	–	5.0 fs	1000	0.01	–	10.1
100	Interframe, var. base (n=1)[a]	–	5.0 fs	1000	0.01	–	9.21
100	Intraframe, BWLZH[a]	–	5.0 fs	1000	0.01	–	15.2
100	Interframe, BWLZH[a]	–	5.0 fs	1000	0.01	–	15.2
100	Interframe, BWLZH[a]	–	5.0 fs	1000	0.01	–	4.30
100	One-to-one, XTC3[a]	–	5.0 fs	1000	0.01	–	8.71
100	Interframe, BWLZH[a]	Interframe, stop bits (n=1)[b]	5.0 fs	1000	0.01	0.1	13.4
100	Interframe, BWLZH[a]	Interframe, var. base (n=4)[b]	5.0 fs	1000	0.01	0.1	12.5
100	Interframe, BWLZH[a]	One-to-one, BWLZH[b]	5.0 fs	1000	0.01	0.1	9.18
100	Interframe, BWLZH[a]	Interframe, BWLZH[b]	5.0 fs	1000	0.01	0.1	8.31
–	Text, gzip	–	50 fs	1000	0.01	–	70.9
–	Text, bzip2	–	50 fs	1000	0.01	–	54.5
100	Interframe, stop bits (n=3)[a]	–	50 fs	1000	0.01	–	21.3
100	Interframe, var. base (n=4)[a]	–	50 fs	1000	0.01	–	20.1
100	Intraframe, BWLZH[a]	–	50 fs	1000	0.01	–	25.4
100	Interframe, BWLZH[a]	–	50 fs	1000	0.01	–	12.5
100	One-to-one, XTC3[a]	–	50 fs	1000	0.01	–	20.0
100	Interframe, BWLZH[a]	Interframe, stop bits (n=3)[b]	50 fs	1000	0.01	0.1	31.0
100	Inter frame, BWLZH[a]	Interframe, var. base (n=3)[b]	50 fs	1000	0.01	0.1	29.9
100	Interframe, BWLZH[a]	One-to-one, BWLZH[b]	50 fs	1000	0.01	0.1	26.4
100	Interframe, BWLZH[a]	Interframe, BWLZH[b]	50 fs	1000	0.01	0.1	25.4
–	Text, gzip	–	500 fs	1000	0.01	–	70.9
–	Text, bzip2	–	500 fs	1000	0.01	–	54.6
100	Interframe, stop bits (n=5)[a]	–	500 fs	1000	0.01	–	32.3
100	Interframe, var. base (n=6)[a]	–	500 fs	1000	0.01	–	30.7
100	Intraframe, BWLZH[a]	–	500 fs	1000	0.01	–	35.2
100	Inter-frame, BWLZH[a]	–	500 fs	1000	0.01	–	30.2
100	One-to-one, XTC3[a]	–	500 fs	1000	0.01	–	30.5
100	Interframe, BWLZH[a]	Interframe, stop bits (n=5)[b]	500 fs	1000	0.01	0.1	57.3
100	Interframe, BWLZH[a]	Interframe, var. base (n=5)[b]	500 fs	1000	0.01	0.1	55.4
100	Interframe, BWLZH[a]	One-to-one, BWLZH[b]	500 fs	1000	0.01	0.1	53.0
100	Interframe, BWLZH[a]	Interframe, BWLZH[b]	500 fs	1000	0.01	0.1	54.0
–	Text, gzip	–	5.0 ps	100	0.01	–	7.09

Table 3 (continued)

Frames in each block	Position algorithm	Velocity algorithm	Time between each frame	Number of frames	Position precision (Å)	Velocity precision (Å/ps)	File size (10^6B)
–	Text, bzip2	–	5.0 ps	100	0.01	–	5.46
100	Interframe, stop bits (n=9)[a]	–	5.0 ps	100	0.01	–	3.90
100	Interframe, var. base (n=8)[a]	–	5.0 ps	100	0.01	–	3.65
100	Intraframe, BWLZH[a]	–	5.0 ps	100	0.01	–	4.08
100	Interframe, BWLZH[a]	–	5.0 ps	100	0.01	–	3.61
100	One-to-one, XTC3[a]	–	5.0 ps	100	0.01	–	3.64
100	One-to-one, BWLZH	Interframe, stop bits (n=5)[b]	5.0 ps	100	0.01	0.1	6.30
100	One-to-one, BWLZH	Interframe, var. base (n=5)[b]	5.0 ps	100	0.01	0.1	6.11
100	One-to-one, BWLZH	One-to-one, BWLZH[b]	5.0 ps	100	0.01	0.1	5.90
100	One-to-one, BWLZH	Interframe, BWLZH[b]	5.0 ps	100	0.01	0.1	5.98

[a] One-to-one, XTC2 is used for the first frame in each block

[b] One-to-one, BWLZH is used for the first frame in each block

Table 4 Trajectory file sizes in bytes from the liquid water simulation trajectory. The results for different compression algorithms are shown. For comparison, the uncompressed trajectories where values are stored as 32-bit floats are shown

Frames in each block	Position algorithm	Velocity algorithm	Time between each frame	Number of frames	Position precision (Å)	Velocity precision (Å/ps)	File size (10^6B)
–	32-bit float	–	1.5 fs	1000	$\approx 10^{-5}$	–	48
–	32-bit float	32-bit float	1.5 fs	1000	$\approx 10^{-5}$	$\approx 10^{-5}$	96
–	Text	–	1.5 fs	1000	0.01	–	68.2
–	Text, gzip	–	1.5 fs	1000	0.01	–	25.8
–	Text, bzip2	–	1.5 fs	1000	0.01	–	17.8
1	XTC	–	1.5 fs	1000	0.01	–	13.3
1	Intraframe, var. base (n=7)	–	1.5 fs	1000	0.01	–	15.1
1	One-to-one, XTC2	–	1.5 fs	1000	0.01	–	13.4
1	Intraframe, BWLZH	–	1.5 fs	1000	0.01	–	16.9
1	One-to-one, XTC3	–	1.5 fs	1000	0.01	–	13.5
1	One-to-one, XTC2	One-to-one, stop bits (n=5)	1.5 fs	1000	0.01	0.1	26.7
1	One-to-one, XTC2	One-to-one, var. base (n=8)	1.5 fs	1000	0.01	0.1	26.4
1	One-to-one, XTC2	One-to-one, BWLZH	1.5 fs	1000	0.01	0.1	26.8
100	Interframe, stop bits (n=1)[a]	–	1.5 fs	1000	0.01	–	4.82
100	Interframe, var. base (n=2)[a]	–	1.5 fs	1000	0.01	–	4.71
100	Intraframe, BWLZH[a]	–	1.5 fs	1000	0.01	–	4.83
100	Interframe, BWLZH[a]	–	1.5 fs	1000	0.01	–	2.42
100	One-to-one, XTC3[a]	–	1.5 fs	1000	0.01	–	9.98
100	Interframe, BWLZH[a]	Interframe, stop bits (n=3)[b]	1.5 fs	1000	0.01	0.1	11.5
100	Interframe, BWLZH[a]	Interframe, var. base (n=4)[b]	1.5 fs	1000	0.01	0.1	11.2
100	Interframe, BWLZH[a]	One-to-one, BWLZH[b]	1.5 fs	1000	0.01	0.1	8.25
100	Interframe, BWLZH[a]	Interframe, BWLZH[b]	1.5 fs	1000	0.01	0.1	5.98
–	Text, gzip	–	15 fs	1000	0.01	–	25.8
–	Text, bzip2	–	15 fs	1000	0.01	–	19.7
100	Interframe, stop bits (n=4)[a]	–	15 fs	1000	0.01	–	9.38
100	Interframe, var. base (n=5)[a]	–	15 fs	1000	0.01	–	9.23

Table 4 (continued)

Frames in each block	Position algorithm	Velocity algorithm	Time between each frame	Number of frames	Position precision (Å)	Velocity precision (Å/ps)	File size (10^6B)
100	Intraframe, BWLZH[a]	–	15 fs	1000	0.01	–	10.4
100	Interframe, BWLZH[a]	–	15 fs	1000	0.01	–	8.70
100	One-to-one, XTC3[a]	–	15 fs	1000	0.01	–	11.5
100	Interframe, BWLZH[a]	Interframe, stop bits (n=5)[b]	15 fs	1000	0.01	0.1	22.0
100	Interframe, BWLZH[a]	Interframe, var. base (n=7)[b]	15 fs	1000	0.01	0.1	22.1
100	Interframe, BWLZH[a]	One-to-one, BWLZH[b]	15 fs	1000	0.01	0.1	19.4
100	Interframe, BWLZH[a]	Interframe, BWLZH[b]	15 fs	1000	0.01	0.1	19.7
–	Text, gzip	–	150 fs	1000	0.01	–	25.8
–	Text, bzip2	–	150 fs	1000	0.01	–	20.2
100	Interframe, stop bits (n=5)[a]	–	150 fs	1000	0.01	–	12.5
100	Interframe, var. base (n=6)[a]	–	150 fs	1000	0.01	–	11.9
100	Intraframe, BWLZH[a]	–	150 fs	1000	0.01	–	11.9
100	Interframe, BWLZH[a]	–	150 fs	1000	0.01	–	11.7
100	One-to-one, XTC3[a]	–	150 fs	1000	0.01	–	11.5
100	Interframe, BWLZH[a]	Interframe, stop bits (n=5)[b]	150 fs	1000	0.01	0.1	25.5
100	Interframe, BWLZH[a]	Interframe, var. base (n=7)[b]	150 fs	1000	0.01	0.1	25.2
100	Interframe, BWLZH[a]	One-to-one, BWLZH[b]	150 fs	1000	0.01	0.1	22.6
100	Interframe, BWLZH[a]	Interframe, BWLZH[b]	150 fs	1000	0.01	0.1	22.8
–	Text, gzip	–	1.5 fs	100	0.01	–	2.58
–	Text, bzip2	–	1.5 fs	100	0.01	–	2.02
100	Interframe, stop bits (n=8)[a]	–	1.5 fs	100	0.01	–	1.51
100	Interframe, var. base (n=9)[a]	–	1.5 fs	100	0.01	–	1.41
100	Intraframe, BWLZH[a]	–	1.5 fs	100	0.01	–	1.34
100	Interframe, BWLZH[a]	–	1.5 fs	100	0.01	–	1.39
100	One-to-one, XTC3[a]	–	1.5 fs	100	0.01	–	1.20
100	One-to-one, XTC2	Interframe, stop bits (n=5)[b]	1.5 fs	100	0.01	0.1	2.73
100	One-to-one, XTC2	Interframe, var. base (n=7)[b]	1.5 fs	100	0.01	0.1	2.68
100	One-to-one, XTC2	One-to-one, BWLZH[b]	1.5 fs	100	0.01	0.1	2.43
100	One-to-one, XTC2	Interframe, BWLZH[b]	1.5 fs	100	0.01	0.1	2.44

[a] One-to-one, XTC2 is used for the first frame in each block

[b] One-to-one, BWLZH is used for the first frame in each block

Table 5 Trajectory file sizes in bytes from the liquid water simulation trajectory stored with high accuracy. The results for different compression algorithms are shown. For comparison, the uncompressed trajectories where values are stored as 32-bit floats are shown

Frames in each block	Position algorithm	Velocity algorithm	Time between each frame	Number of frames	Position precision (Å)	Velocity precision (Å/ps)	File size (10^6B)
–	32-bit float	–	1.5 fs	1000	$\approx 10^{-5}$	–	48
–	32-bit float	32-bit float	1.5 fs	1000	$\approx 10^{-5}$	$\approx 10^{-5}$	96
–	Text	–	1.5 fs	1000	10^{-5}	–	68.2
–	Text, gzip	–	1.5 fs	1000	10^{-5}	–	47.6
–	Text, bzip2	–	1.5 fs	1000	10^{-5}	–	38.0
1	XTC	–	1.5 fs	1000	10^{-5}	–	28.9
1	Intraframe, var. base (n=17)	–	1.5 fs	1000	10^{-5}	–	30.0
1	One-to-one, XTC2	–	1.5 fs	1000	10^{-5}	–	28.3

Table 5 (continued)

Frames in each block	Position algorithm	Velocity algorithm	Time between each frame	Number of frames	Position precision (Å)	Velocity precision (Å/ps)	File size (10^6B)
1	Intraframe, BWLZH	–	1.5 fs	1000	10^{-5}	–	39.6
1	One-to-one, XTC3	–	1.5 fs	1000	10^{-5}	–	29.2
1	One-to-one, XTC2	One-to-one, stop bits (n=11)	1.5 fs	1000	10^{-5}	10^{-4}	57.0
1	One-to-one, XTC2	One-to-one, var. base (n=16)	1.5 fs	1000	10^{-5}	10^{-4}	56.2
1	One-to-one, XTC2	One-to-one, BWLZH	1.5 fs	1000	10^{-5}	10^{-4}	64.8
100	Interframe, stop bits (n=7)[a]	–	1.5 fs	1000	10^{-5}	–	20.7
100	Interframe, var. base (n=11)[a]	–	1.5 fs	1000	10^{-5}	–	19.4
100	Intraframe, BWLZH[a]	–	1.5 fs	1000	10^{-5}	–	32.4
100	Interframe, BWLZH[a]	–	1.5 fs	1000	10^{-5}	–	12.7
100	One-to-one, XTC3[a]	–	1.5 fs	1000	10^{-5}	–	26.4
100	Interframe, BWLZH[a]	Interframe, stop bits (n=9)[b]	1.5 fs	1000	10^{-5}	10^{-4}	35.9
100	Interframe, BWLZH[a]	Interframe, var. base (n=5)[b]	1.5 fs	1000	10^{-5}	10^{-4}	36.8
100	Interframe, BWLZH[a]	One-to-one, BWLZH[b]	1.5 fs	1000	10^{-5}	10^{-4}	42.0
100	Interframe, BWLZH[a]	Interframe, BWLZH[b]	1.5 fs	1000	10^{-5}	10^{-4}	34.8
–	Text, gzip	–	15 fs	1000	10^{-5}	–	47.6
–	Text, bzip2	–	15 fs	1000	10^{-5}	–	38.2
100	Interframe, stop bits (n=3)[a]	–	15 fs	1000	10^{-5}	–	25.7
100	Interframe, var. base (n=4)[a]	–	15 fs	1000	10^{-5}	–	24.2
100	Intraframe, BWLZH[a]	–	15 fs	1000	10^{-5}	–	34.6
100	Interframe, BWLZH[a]	–	15 fs	1000	10^{-5}	–	26.2
100	One-to-one, XTC3[a]	–	15 fs	1000	10^{-5}	–	28.9
100	Interframe, var. base (n=14)[a]	Interframe, stop bits (n=11)[b]	15 fs	1000	10^{-5}	10^{-4}	51.7
100	Interframe, var. base (n=14)[a]	Interframe, var. base (n=17)[b]	15 fs	1000	10^{-5}	10^{-4}	52.5
100	Interframe, var. base (n=14)[a]	One-to-one, BWLZH[b]	15 fs	1000	10^{-5}	10^{-4}	54.7
100	Interframe, var. base (n=14)[a]	Interframe, BWLZH[b]	15 fs	1000	10^{-5}	10^{-4}	53.1
–	Text, gzip	–	150 fs	1000	10^{-5}	–	47.6
–	Text, bzip2	–	150 fs	1000	10^{-5}	–	38.2
100	Interframe, stop bits (n=17)[a]	–	150 fs	1000	10^{-5}	–	28.3
100	Interframe, var. base (n=16)[a]	–	150 fs	1000	10^{-5}	–	26.8
100	Intraframe, BWLZH[a]	–	150 fs	1000	10^{-5}	–	34.8
100	Interframe, BWLZH[a]	–	150 fs	1000	10^{-5}	–	30.4
100	One-to-one, XTC3[a]	–	150 fs	1000	10^{-5}	–	29.0
100	Interframe, var. base (n=16)[a]	Interframe, stop bits (n=11)[b]	150 fs	1000	10^{-5}	10^{-4}	54.9
100	Interframe, var. base (n=16)[a]	Interframe, var. base (n=17)[b]	150 fs	1000	10^{-5}	10^{-4}	55.3
100	Interframe, var. base (n=16)[a]	One-to-one, BWLZH[b]	150 fs	1000	10^{-5}	10^{-4}	57.6
100	Interframe, var. base (n=16)[a]	Interframe, BWLZH[b]	150 fs	1000	10^{-5}	10^{-4}	56.4
–	Text, gzip	–	1.5 fs	100	10^{-5}	–	4.77
–	Text, bzip2	–	1.5 fs	100	10^{-5}	–	3.82
100	Interframe, stop bits (n=19)[a]	–	1.5 fs	100	10^{-5}	–	3.06
100	Interframe, var. base (n=17)[a]	–	1.5 fs	100	10^{-5}	–	2.91
100	Intraframe, BWLZH[a]	–	1.5 fs	100	10^{-5}	–	3.49
100	Interframe, BWLZH[a]	–	1.5 fs	100	10^{-5}	–	3.30
100	One-to-one, XTC3[a]	–	1.5 fs	100	10^{-5}	–	2.90
100	One-to-one, XTC2	Interframe, stop bits (n=11)[b]	1.5 fs	100	10^{-5}	10^{-4}	5.64
100	One-to-one, XTC2	Interframe, var. base (n=13)[b]	1.5 fs	100	10^{-5}	10^{-4}	5.67
100	One-to-one, XTC2	One-to-one, BWLZH[b]	1.5 fs	100	10^{-5}	10^{-4}	5.90
100	One-to-one, XTC2	Interframe, BWLZH[b]	1.5 fs	100	10^{-5}	10^{-4}	5.78

[a] One-to-one, XTC2 is used for the first frame in each block

[b] One-to-one, BWLZH is used for the first frame in each block

Table 6 Trajectory file sizes in bytes from the solid magnesium oxide simulation trajectory. The results for different compression algorithms are shown. For comparison, the uncompressed trajectories where values are stored as 32-bit floats are shown

Frames in each block	Position algorithm	Velocity algorithm	Time between each frame	Number of frames	Position precision (Å)	Velocity precision (Å/ps)	File size (10^6B)
–	32-bit float	–	0.5 fs	1000	$\approx 10^{-5}$	–	49.4
–	32-bit float	32-bit float	0.5 fs	1000	$\approx 10^{-5}$	$\approx 10^{-5}$	98.8
–	Text	–	0.5 fs	1000	0.01	–	69.7
–	Text, gzip	–	0.5 fs	1000	0.01	–	20.5
–	Text, bzip2	–	0.5 fs	1000	0.01	–	8.66
1	XTC	–	0.5 fs	1000	0.01	–	19.1
1	Intraframe, var. base (n=8)	–	0.5 fs	1000	0.01	–	16.0
1	One-to-one, XTC2	–	0.5 fs	1000	0.01	–	15.0
1	Intraframe, BWLZH	–	0.5 fs	1000	0.01	–	9.53
1	One-to-one, XTC3	–	0.5 fs	1000	0.01	–	10.5
1	Intraframe, BWLZH	One-to-one, stop bits (n=5)	0.5 fs	1000	0.01	0.1	22.0
1	Intraframe, BWLZH	One-to-one, var. base (n=6)	0.5 fs	1000	0.01	0.1	21.3
1	Intraframe, BWLZH	One-to-one, BWLZH	0.5 fs	1000	0.01	0.1	21.3
100	Interframe, stop bits (n=1)[a]	–	0.5 fs	1000	0.01	–	3.47
100	Interframe, var. base (n=1)[a]	–	0.5 fs	1000	0.01	–	3.04
100	Intraframe, BWLZH[a]	–	0.5 fs	1000	0.01	–	2.62
100	Interframe, BWLZH[a]	–	0.5 fs	1000	0.01	–	1.42
100	One-to-one, XTC3[a]	–	0.5 fs	1000	0.01	–	2.18
100	Interframe, BWLZH[a]	Interframe, stop bits (n=2)[b]	0.5 fs	1000	0.01	0.1	9.56
100	Interframe, BWLZH[a]	Interframe, var. base (n=2)[b]	0.5 fs	1000	0.01	0.1	9.09
100	Interframe, BWLZH[a]	One-to-one, BWLZH[b]	0.5 fs	1000	0.01	0.1	7.16
100	Interframe, BWLZH[a]	Interframe, BWLZH[b]	0.5 fs	1000	0.01	0.1	6.07
–	Text, gzip	–	0.5 fs	1000	0.01	–	20.5
–	Text, bzip2	–	0.5 fs	1000	0.01	–	14.7
100	Interframe, stop bits (n=2)[a]	–	0.5 fs	1000	0.01	–	5.56
100	Interframe, var. base (n=2)[a]	–	0.5 fs	1000	0.01	–	5.14
100	Intraframe, BWLZH[a]	–	0.5 fs	1000	0.01	–	4.03
100	Interframe, BWLZH[a]	–	0.5 fs	1000	0.01	–	3.13
100	One-to-one, XTC3[a]	–	0.5 fs	1000	0.01	–	5.26
100	Interframe, BWLZH[a]	Interframe, stop bits (n=5)[b]	0.5 fs	1000	0.01	0.1	14.2
100	Interframe, BWLZH[a]	Interframe, var. base (n=5)[b]	0.5 fs	1000	0.01	0.1	13.5
100	Interframe, BWLZH[a]	One-to-one, BWLZH[b]	0.5 fs	1000	0.01	0.1	12.2
100	Interframe, BWLZH[a]	Interframe, BWLZH[b]	0.5 fs	1000	0.01	0.1	11.3
–	Text, gzip	–	50 fs	1000	0.01	–	20.5
–	Text, bzip2	–	50 fs	1000	0.01	–	15.3
100	Interframe, stop bits (n=4)[a]	–	50 fs	1000	0.01	–	9.56
100	Interframe, var. base (n=4)[a]	–	50 fs	1000	0.01	–	8.81
100	Intraframe, BWLZH[a]	–	50 fs	1000	0.01	–	7.43
100	Interframe, BWLZH[a]	–	50 fs	1000	0.01	–	7.94
100	One-to-one, XTC3[a]	–	50 fs	1000	0.01	–	8.02
100	Intraframe, BWLZH[a]	Interframe, stop bits (n=5)[b]	50 fs	1000	0.01	0.1	21.4
100	Intraframe, BWLZH[a]	Interframe, var. base (n=7)[b]	50 fs	1000	0.01	0.1	20.4
100	Intraframe, BWLZH[a]	One-to-one, BWLZH[b]	50 fs	1000	0.01	0.1	18.6
100	Intraframe, BWLZH[a]	Interframe, BWLZH[b]	50 fs	1000	0.01	0.1	19.3
–	Text, gzip	–	500 fs	100	0.01	–	2.06
–	Text, bzip2	–	500 fs	100	0.01	–	1.53
100	Interframe, stop bits (n=4)[a]	–	500 fs	100	0.01	–	0.943
100	Interframe, var. base (n=4)[a]	–	500 fs	100	0.01	–	0.869
100	Intraframe, BWLZH[a]	–	500 fs	100	0.01	–	0.794
100	Interframe, BWLZH[a]	–	500 fs	100	0.01	–	0.807

Table 6 (continued)

Frames in each block	Position algorithm	Velocity algorithm	Time between each frame	Number of frames	Position precision (Å)	Velocity precision (Å/ps)	File size (10^6B)
100	One-to-one, XTC3[a]	–	500 fs	100	0.01	–	0.790
100	Intraframe, BWLZH	Interframe, stop bits (n=5)[b]	500 fs	100	0.01	0.1	2.12
100	Intraframe, BWLZH	Interframe, var. base (n=6)[b]	500 fs	100	0.01	0.1	2.04
100	Intraframe, BWLZH	One-to-one, BWLZH[b]	500 fs	100	0.01	0.1	1.95
100	Intraframe, BWLZH	Interframe, BWLZH[b]	500 fs	100	0.01	0.1	1.99

[a] Intraframe, BWLZH is used for the first frame in each block

[b] One-to-one, BWLZH is used for the first frame in each block

Table 7 Trajectory file sizes in bytes from the virus-in-water simulation trajectory. The results for different compression algorithms are shown. For comparison, the uncompressed trajectories where values are stored as 32-bit floats are shown

Frames in each block	Position algorithm	Velocity algorithm	Time between each frame	Number of frames	Position precision (Å)	Velocity precision (Å/ps)	File size (10^6B)
–	32-bit float	–	2 ps	15	$\approx 10^{-4}$	–	181
–	Text	–	2 ps	15	0.01	–	298
–	Text, gzip	–	2 ps	15	0.01	–	117
–	Text, bzip2	–	2 ps	15	0.01	–	95.6
1	XTC	–	2 ps	15	0.01	–	59.2
1	Intraframe, var. base (n=8)	–	2 ps	15	0.01	–	64.2
1	One-to-one, XTC2	–	2 ps	15	0.01	–	57.6
1	Intraframe, BWLZH	–	2 ps	15	0.01	–	72.5
1	One-to-one, XTC3	–	2 ps	15	0.01	–	55.9
1	One-to-one, XTC3 (base only)	–	2 ps	15	0.01	–	58.3
–	Text	–	2 ps	15	0.01	–	298
–	Text, gzip	–	2 ps	15	0.01	–	117
–	Text, bzip2	–	2 ps	15	0.01	–	95.3
15	Interframe, stop bits (n=6)[a]	–	2 ps	15	0.01	–	61.2
15	Interframe, var. base (n=8)[a]	–	2 ps	15	0.01	–	58.2
15	Intraframe, BWLZH[a]	–	2 ps	15	0.01	–	60.1
15	Interframe, BWLZH[a]	–	2 ps	15	0.01	–	61.4
15	One-to-one, XTC3[a]	–	2 ps	15	0.01	–	49.5
15	One-to-one, XTC3 (base only)[a]	–	2 ps	15	0.01	–	53.2
15	Interframe, stop bits (n=7)[a]	–	20 ps	15	0.01	–	68.5
15	Interframe, var. base (n=10)[a]	–	20 ps	15	0.01	–	67.2
15	Intraframe, BWLZH[a]	–	20 ps	15	0.01	–	63.4
15	Interframe, BWLZH[a]	–	20 ps	15	0.01	–	69.6
15	One-to-one, XTC3[a]	–	20 ps	15	0.01	–	51.9
15	One-to-one, XTC3 (base only)[a]	–	20 ps	15	0.01	–	55.5
15	Interframe, stop bits (n=7)[a]	–	200 ps	15	0.01	–	78.1
15	Interframe, var. base (n=10)[a]	–	200 ps	15	0.01	–	75.9
15	Intraframe, BWLZH[a]	–	200 ps	15	0.01	–	66.5
15	Interframe, BWLZH[a]	–	200 ps	15	0.01	–	80.4
15	One-to-one, XTC3[a]	–	200 ps	15	0.01	–	54.6
15	One-to-one, XTC3 (base only)[a]	–	200 ps	15	0.01	–	58.0

[a] One-to-one, XTC3 coding is used for the first frame in each block

Table 8 Trajectory file sizes in bytes from the virus-in-water simulation trajectory. The results for different compression algorithms are shown. For comparison, the uncompressed trajectories where values are stored as 32-bit floats are shown

Frames in each block	Position algorithm	Velocity algorithm	Time between each frame	Number of frames	Position precision (Å)	Velocity precision (Å/ps)	File size (10^6B)
–	32-bit float	–	50 ps	7	$\approx 10^{-4}$	–	84.5
–	32-bit float	32-bit float	50 ps	7	$\approx 10^{-4}$	$\approx 10^{-5}$	169
7	One-to-one, XTC2	–	50 ps	7	0.01	–	27.7
7	One-to-one, XTC2	One-to-one, stop bits (n=5)	50 ps	7	0.01	0.1	52.4
7	One-to-one, XTC2	One-to-one, var. base (n=7)	50 ps	7	0.01	0.1	50.6
7	One-to-one, XTC2	One-to-one, BWLZH	50 ps	7	0.01	0.1	51.1
7	One-to-one, XTC2	One-to-one, stop bits (n=7)[a]	50 ps	7	0.01	0.1	53.6
7	One-to-one, XTC2	One-to-one, var. base (n=8)[a]	50 ps	7	0.01	0.1	52.1
7	One-to-one, XTC2	One-to-one, BWLZH[a]	50 ps	7	0.01	0.1	51.9
7	One-to-one, XTC2	One-to-one, stop bits (n=5)	500 ps	7	0.01	0.1	52.5
7	One-to-one, XTC2	One-to-one, var. base (n=7)	500 ps	7	0.01	0.1	50.7
7	One-to-one, XTC2	One-to-one, BWLZH	500 ps	7	0.01	0.1	51.1
7	One-to-one, XTC2	One-to-one, stop bits (n=7)[a]	500 ps	7	0.01	0.1	53.6
7	One-to-one, XTC2	One-to-one, var. base (n=8)[a]	500 ps	7	0.01	0.1	52.1
7	One-to-one, XTC2	One-to-one, BWLZH[a]	500 ps	7	0.01	0.1	51.9

[a] One-to-one, var. base (n=7) is used for the first frame in each block

References

1. Hess B, Kutzner C, van der Spoel D, Lindahl E (2008) J Chem Theory Comput 4(3):435
2. Bowers KJ, Chow E, Xu H, Dror RO, Eastwood MP, Gregersen BA, Klepeis JL, Kolossváry I, Moraes MA, Sacerdoti FD, Salmon JK, Shan Y, Shaw DE (2006) SC ’06: Proceedings of the ACM/IEEE Conference on Supercomputing. ACM, New York
3. Phillips JC, Braun R, Wang W, Gumbart J, Tajkhorshid E, Villa E, Chipot C, Skeel RD, Kale L, Schulten K (2005) J Comp Chem 26:1781
4. Gailly J et al (2010) GZIP version 1.4. http://ftp.gnu.org/gnu/gzip/
5. Seward J (2008) BZIP2 version 1.0.5. http://www.bzip.org/
6. Green D, Meacham KE, Surridge M, van Hoesel F, Berendsen HJC (1995) Methods and techniques in computational chemistry: METECC-95. STEF, Cagliari, p 435
7. Melo A, Puga AT, Gentil F, Brito N, Alves AP, Ramos MJ (2000) J Chem Inf Comput Sci 40:559
8. Meyer T, Ferrer-Costa C, Pérez A, Rueda M, Bidon-Chanal A, Luque FJ, Laughton A, Oronzco M (2006) J Chem Theory Comput 2:251
9. Uppsala Universitet (2010)TrajNG—trajectory compression library. http://www.uppmax.uu.se/Members/daniels/trajng-trajectory-compression-library
10. Burrows M, Wheeler DJ (1994) SRC research report. Digital Equipment Corporation, Palo Alto
11. Ziv J, Lempel A (1977) IEEE Trans Inf Theory IT23:337
12. Huffman DV (1952) IRE 40:1098
13. Bentley J, Sleator D, Tarjan R, Wei V (1986) Commun ACM 29 (4):320
14. Schulz R, Lindner B, Petridis L, Smith J (2009) J Chem Theory Comput 5:2798
15. Allen MP, Tildesley DJ (1987) Computer simulation of liquids. Clarendon, Oxford
16. Swope WC, Andersen HC, Berens PH, Wilson KR (1982) J Chem Phys 76:637
17. Jorgensen WL, Chandrasekhar J, Madura JD, Impey RW, Klein ML (1983) J Chem Phys 79:926
18. Nosé S (1984) Mol Phys 52:255
19. Hoover WG (1985) Phys Rev A 31:1695
20. Andersen HC (1983) J Comput Phys 52:24
21. Cicotti G, Ferrario M, Ryckaert J (1982) Mol Phys 47(6):1253
22. Harding JH, Harker AH (1985) Phil Mag B 25(3):119
23. Cleveland CL (1988) J Chem Phys 89(8):4987
24. Parrinello M, Rahman A (1981) J App Phys 52(12):7182
25. Mitchell PJ, Fincham D (1993) J Phys Condens Matter 5:1031
26. Jones T, Liljas L (1984) J Mol Biol 177:735
27. Berendsen HJC, Postma JPM, van Gunsteren WF, Nola AD, Haak JR (1984) J Chem Phys 81:3684
28. Miyamoto S, Kollman P (1992) J Comput Chem 13:952
29. Hess B (2008) J Chem Theory Comput 4:116