Z. Phys. Chem. **227** (2013) 345–352 / **DOI** 10.1524/zpch.2013.0311
© by Oldenbourg Wissenschaftsverlag, München

# Efficient Coding of the Minimum Image Convention

By Ulrich K. Deiters*

Institute of Physical Chemistry, University of Cologne, Luxemburger Str. 116, 50939 Köln, Germany

*Dedicated to Prof. Dr. Andreas Heintz on the occasion of his 65th birthday*

Received July 2, 2012; accepted November 30, 2012
(Published online January 28, 2013)

***Computer Simulation / Fluid Phases / Periodic Boundary Conditions / Code Optimization***

The computation of distances between interactions sites while observing the minimum image convention is one of the most frequently performed operations in computer simulations of fluids. In this work several ways of encoding the minimum image convention are discussed and their performances on different computers compared. It turns out that the optimal choice for the algorithm depends on the computer type as well as the optimization level of the compiler, and that a well adapted algorithm can achieve a significant reduction of the computation time.

## 1. Introduction

One of the most frequently performed calculations in computer simulations of fluids – Monte Carlo as well as molecular dynamics simulations – is the computation of the distance between two particles (or interaction sites), and evidently the efficiency of this computation can have a significant impact on the performance of a simulation program. This distance computation, however, is complicated by the fact that the particle numbers in typical molecular computer simulations are of the order of merely $10^2$–$10^4$, and that the "thermodynamic limit" (infinite sample size) is realized approximately by means of the periodic boundary conditions, *i.e.*, by surrounding a simulation ensemble with identical replicas, thus creating an infinite, periodic superlattice of simulation boxes. In such a superlattice, interactions are computed between the nearest replicas of particles only. This so-called minimum image convention requires additional computation steps.

Efficient code implementations of the minimum image convention had already been addressed by a publication in 1998 [1], but as computer architectures as well as compilers have changed over the past decade, it may now be worthwhile to reconsider this problem with modern computers.

* Corresponding author. E-mail: ulrich.deiters@uni-koeln.de

## 2. The algorithms

The Euclidian distance between a particle at the location $\mathbf{r}_0$ and one at $\mathbf{r}_1$ is

$$|\mathbf{r}_1 - \mathbf{r}_0| = \sqrt{(\Delta x)^2 + (\Delta y)^2 + (\Delta z)^2}\,, \tag{1}$$

with $\Delta x = x_1 - x_0$ and similar expressions for the other coordinates. In a system with periodic boundary conditions, the one-dimensional distance corresponds to a set of values

$$\Delta_k x = \Delta x + kb \quad \text{with } k = \ldots, -2, -1, 0, +1, +2, \ldots . \tag{2}$$

Here $b$ denotes the length of the simulation box or, in other words, the lattice constant of the superlattice.

The minimum image convention then corresponds to choosing $k$ in such a way that $|\Delta x|$ becomes minimal:

$$\Delta_{\text{mic}} x = \min(\Delta x + kb) \tag{3}$$

$\Delta_{\text{mic}} x$ is known as the remainder of $\Delta x$ with respect to the box length $b$.

The problem is now that the calculation of the remainder is not part of the instruction set of the X86 family of microprocessors [2], which nowadays seems to dominate the field.[1] Therefore the remainder calculation has to be performed otherwise, and we will explore now some ways to do so.

The best way to calculate remainders may depend on the way how the coordinates of the molecules are stored with respect to the periodic boundary conditions. There are three different concepts:

A. For all atoms or interaction sites, absolute coordinates are stored. If, in the course of the simulation, a site moves out of the primary simulation box, it is "folded back" by adding $\pm b$ to the relevant coordinate.
   Consequently, all interaction sites have coordinates within the primary simulation box. This simplifies the calculation of distances, but it makes it awkward to rotate a molecule.
B. For the center of each molecule (the center of gravity or simply a selected site), absolute coordinates are stored; for the other sites, relative coordinates with respect to the center are stored. If the center moves out of the primary simulation box, it is folded back.
   With this concept, shifting or rotating a molecule is easy. But now the interaction sites of a molecule may lie outside the primary box, even if its center is inside.
C. The molecules are never folded back, but can drift away from the primary simulation box unhindered.
   Consequently, the molecules can move to locations far outside the primary simulation box. With this storage concept it is possible to calculate mean squared displacements and other diffusion-related properties.

[1] The Motorola 68000 family had such an instruction (`frem`) [3], which could be put to good use in simulations [4], but that was about 25 years ago. Unfortunately, the X86 "partial remainder" instruction (`fprem1`) is not a valid substitute.

In the following sections we will outline some ways to encode the minimum image convention, using C/C++ notation. In the algorithms, `box` denotes the box length $b$ and `dx` the raw difference of the coordinates of two sites ($\Delta x$), which is then replaced by the remainder. Moreover, we define

```
int k;
double box_r = 1.0 / box; // reciprocal box length
double box2 = 0.5 * box;
double box2_r = 1.0 / box2;
```

## 2.1 Class A algorithms

If all interaction sites are in the same simulation box, the distance between two arbitrary sites along any coordinate axis cannot exceed the box length, hence $-b \le \Delta x \le +b$.

A1: A simple, but efficient way to calculate remainders is then doing an implicit double-to-integer cast operation, which is usually rather fast ([1], Alg. 2):

```
k = (int) dx * box2_r;
dx -= k * box;
```

With the default settings of the compilers, such a cast operation is equivalent to a rounding towards zero.

Of course, any Class B or C algorithm can also be used here, too.

## 2.2 Class B algorithms

If the centers or primary sites of all molecules are in the same box, but the other sites can be away from the centers by a distance $L$, the distances between two arbitrary sites must be in the range $-b-L \le \Delta x \le +b+L$. We must demand that the algorithm works reliably over this range of distances.

B1: The remainder can then be calculated by an extension of Algorithm A1, namely by doing the "cast" step twice ([1], Alg. 3). The working range of this algorithm is $-1.5b \le \Delta x \le +1.5b$:

```
k = (int) dx * box2_r;
dx -= k * box;
k = (int) dx * box2_r;
dx -= k * box;
```

B2: But also a simple conditional clause can be used ([1], Alg. 0):

```
if (dx > box2)
   dx -= box;
else if (dx < box2)
   dx += box;
```

Here no multiplications are performed. The working range of this algorithm is the same as that of B1.

B3: As the sign of $\Delta x$ does not matter for the calculation of the Euclidian distance, the previous method can be simplified:

```
dx = fabs(dx);
if (dx > box2) dx -= box;
```

Again, the working range of this algorithm is the same as that of B1.

B4: Finally, the problem of dealing with negative distances can be eliminated by adding a multiple of the box length ([1], Alg. 4). So the following algorithm can be used for the range $-3.5b \le \Delta x \le +\infty$:

```
dx = dx * box_r + 3.0;
k = (int) dx + 0.5;
dx = (dx - k) * box;
```

Users should be aware that the Class B algorithms might be problematic in simulations with boxes of variable size (isobaric simulations), where the condition $b > 2r$ can possibly be violated.

## 2.3 Class C algorithms

Algorithms in this class calculate remainders at any distance.

C1: A straightforward way of implementing the remainder calculation is, of course, the invocation of the `remainder()` or `drem()` library function[2]:

```
dx = remainder(dx, box);
```

`remainder()` involves a floating-point division.

C2: Equation (3) seems to suggest the calculation the nearest integer, either this way:

```
dx -= box * nearbyint(dx * box_r);
```

C3: or this way:

```
dx -= box * round(dx * box_r);
```

The difference between `round()` and `nearbyint()` seems to be that the behaviour of the latter can be controlled by some external flags.

C4: Alternatively one may use the largest integer value below $\Delta x/b$:

```
dx -= box * floor(dx * box_r + 0.5);
```

C5: The following method makes use of a fast double-to-integer cast:

```
k = (int) dx * box_r + ((dx >= 0.0) ? 0.5 : -0.5);
dx -= k * box;
```

C6: As in Algorithm B3, the conditional expression can be eliminated if the sign of $\Delta x$ can be disregarded:

```
k = (int) fabs(dx * box_r) + 0.5;
dx -= k * box;
```

[2] We are referring to the Unix or Linux standard C library ([5], Sect. 3).

**Table 1.** CPU time [ms] required for the calculation of all site–site distances of an ensemble of 256 diatomic molecules, for various computers and algorithms.

| algorithm | machine/optimization | | | | | | |
|---|---|---|---|---|---|---|---|
| | 1 | | 2 | | 3 | | 4 |
| | −O3 | −Ofast | −O3[1] | −fast[2] | −O3 | −Ofast | +O4 |
| A1 | 0.9 | 0.9 | 1.3 | 1.3 | 3.0 | 3.0 | 9.0 |
| B1 | 1.5 | 1.4 | 2.3 | 2.0 | 7.2 | 6.9 | 30.9 |
| B2 | 1.1 | 1.1 | 1.9 | 1.9 | 2.7 | 2.8 | 19.8 |
| B3 | 1.1 | 0.8 | 1.6 | 2.0 | 2.4 | 1.9 | 100.3 |
| B4 | 1.1 | 1.1 | 1.8 | 1.6 | 4.2 | 3.5 | 24.4 |
| C1 | 7.2 | 5.7 | 10.0 | 10.5 | 6.3 | 5.3 | 326.4 |
| C2 | 53.0 | 1.2 | 49.6 | 1.1 | 105.4 | 3.0 | |
| C3 | 2.8 | 1.4 | 6.0 | 6.0 | 7.0 | 4.4 | |
| C4 | 3.5 | 1.6 | 6.7 | 1.2 | 8.7 | 5.8 | 252.3 |
| C5 | 1.2 | 1.1 | 1.8 | 2.1 | 4.2 | 4.0 | 30.2 |
| C6 | 1.0 | 1.0 | 1.6 | 1.5 | 4.1 | 3.5 | 107.3 |

[1] g++ compiler, "fast math" optimization had no significant effect
[2] icpc compiler

## 3. Application

The algorithms from the previous section were used to compute a complete site–site distance table for an ensemble of 256 diatomic molecules. The configuration of this ensemble was the result of an $NpT$ Monte Carlo simulation. Within the usual statistical uncertainties, the configuration had neither preferential orientations nor any long-distance order. The simulation program employed the Class C storage concept, *i.e.*, the molecules could move to coordinates outside the primary simulation box. For the test of the Class A and B algorithms, the coordinates of the "escaped" molecules were folded back to the primary box.

The computers used for the comparisons were as follows:

1. Linux workstation, processor: Intel Core i5-2500K (clock frequency 3.3 GHz), compiler: g++ (GNU C++ compiler, v. 4.6)
2. CHEOPS cluster of the computer center of the University of Cologne, processor: Intel Xeon X5550 (2.66 GHz), compiler: g++ or ipcp (Intel C++ compiler, v. 12.1)
3. Linux server (Transtec Calleo), processor: AMD Opteron 6218 (2.0 GHz), compiler: g++
4. HP-UX workstation (Hewlett-Packard Visualize C3000), processor: HP-PA 8500 (400 MHz), compiler: cc (Hewlett-Packard C compiler); we include this machine in the comparison not only for nostalgic reasons, but also because it has a different processor architecture (HP-PA RISC)

In all cases, the test programs were compiled with high optimization levels. The effect of a "fast math" optimization, which is possible with some compilers, was tested, too. Only one processor core was used. As with some processors the clock frequency

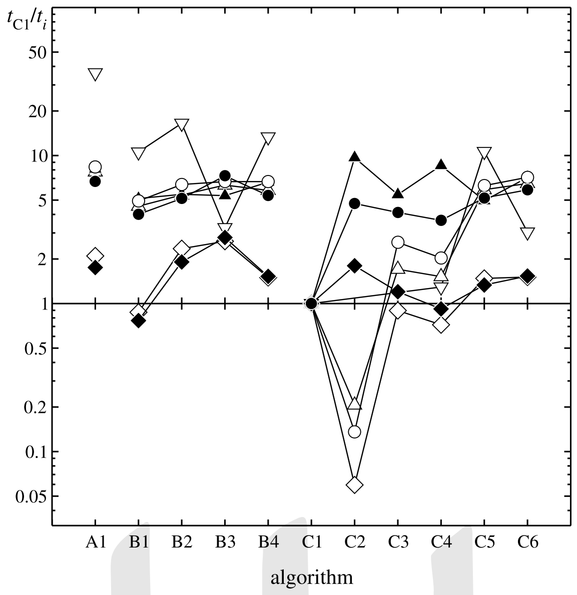

**Fig. 1.** Relative speed improvement (ratio of CPU times) relative to algorithm C1 for several algorithms for the evaluation of the minimum image convention, based on Table 1. ○: machine 1, △: machine 2, ◇: machine 3, ▽: machine 4; open symbols: optimization `-O3`, filled symbols: "fast math" (on machine 2: Intel compiler). The lines were added for better clarity. Note that the speed scale is logarithmic.

depends on temperature and system activity, care was taken to run the test programs at a low and constant overall system load. The results are shown in Table 1.

The overall decrease of the CPU time requirements from machine 4 to 1 reflects the progress in microprocessor technology. It is evidently advisable to use modern computers for simulations. Of course, this insight is neither new nor the topic of this work. More suprising is the effect of the optimization level: For some algorithms it is rather small; the performance of algorithm C2, however, is simply disastrous when compiled at level `-O3`, and good at level `-Ofast`. This underscores the importance of selecting the correct optimization level of the compiler.

For a comparison of the various algorithms it is perhaps better to look a relative CPU times. Figure 1 contains a logarithmic representation of the relative speeds of the algorithms, $t_{C1}/t_i$, where $t_{C1}$ is the CPU time required for the C1 algorithm (which invokes the `remainder()` library function).

Figure 1 shows that the CPU time requirements of the various algorithms depend on the processor architecture, which confirms our earlier results [6]. Some observations can be made:

- Of the Class B methods, B3 is best (except for the HP-PA processor). At the highest optimization level, it can even be faster than A1.
- C1 is never very good, although on the AMD processors the speed differences are not as pronounced as on the Intel processors.

**Table 2.** CPU times for an NVT-Monte Carlo simulation of a Lennard-Jones fluid on machine 1, relative to the time required for Algorithm C1 at optimization `-O3`.

| algorithm | optimization | |
|---|---|---|
| | −O3 | −Ofast |
| C1 | 1.00 | 1.01 |
| C2 | 3.92 | 0.73 |
| C3 | 0.92 | 0.73 |
| C4 | 0.93 | 0.74 |
| C5 | 0.70 | 0.71 |
| C6 | 0.70 | 0.71 |

- As mentioned above, the performance of algorithm C2 can be outright disastrous if the optimization level is too low.
- Of the Class C algorithms, C5 and C6 can be generally recommended. They usually give good performances irrespective of the optimization level.
- "Fast math" optimization brings the performances of the different algorithms to a similar level. But even so, significant differences remain.

One should be aware, however, that the "fast math" optimization of the GNU as well as the Intel C++ compiler involves unsafe optimizations which may possibly lead to wrong numerical results [7,8]. No adverse effects were observed for our test programs nor for our simulation programs[3], but this may be different for other programs.

*Note to Fortran users:* Most of the C library functions used in this article have Fortran counterparts; type cast operations are handled by modern Fortran compilers implicitly. Therefore most of the algorithms discussed here can easily be translated to Fortran. A Fortran77 version of our program based on Algorithm C3, but using the Fortran library function `NINT()` to compute the nearest integer (as proposed, for instance, in Algorithm 5 in [9]), was slower than the fastest algorithms in Table 1 by a factor of 1.5–2.4 for X86-based processors, depending on the optimization level. On machine 4, the slowdown amounted to a factor of 30.

We conclude that generally the algorithms C5 and C6 give very good performances for most computers, compilers, and optimization levels. But as microprocessors as well as compilers are undergoing a rapid evolution, programmers of computer simulations should be encouraged to make their own performance tests.

In a full simulation program, the computation of distances takes up a part of the total CPU time only; larger shares of the CPU time are required for the calculation of molecule movements and the evaluation of interaction energies and forces. A comparison of some Monte Carlo simulations for a Lennard-Jones fluid shows, however, that the time savings achieved by choosing an efficient encoding of the minimum image convention can still be significant (Table 2).

[3] *mc++* educational Monte-Carlo simulation package, available from the author upon request

## References

1. M. Hloucha and U. K. Deiters, Mol. Simul. **20** (1998) 239.
2. K. Lejska a.k.a "MazeGen", X86 opcode and instruction reference, rev. 1.11 (2009), http://ref.x86asm.net/.
3. Motorola Inc., *MC68881 Floating-Point Coprocessor User's Manual*, 1st edn. (1985).
4. U. K. Deiters, Mol. Simul. **3** (1989) 343.
5. The Linux Man-Pages Project, release of 2012, http://www.kernel.org/doc/man-pages/.
6. M. Hloucha and U. K. Deiters, Fluid Phase Equilib. **149** (1998) 41.
7. Free Software Foundation, Inc., *GNU compiler collection manual*, available at http://gcc.gnu.org/onlinedocs/gcc-4.6.3/gcc/ (2012).
8. Intel Corp., *Intel® C++ Compiler XE 12.1 User and Reference Guides*, available at http://software.intel.com/sites/products/documentation/hpc/composerxe/en-us/2011Update/cpp/lin/index.htm (2011).
9. D. Frenkel and B. Smit, *Understanding Molecular Simulation*, 2nd edn., Academic Press, London (2002).