Appendix B

Some implementation details

In this chapter we will present some implementation details. This is far from complete, but we deemed it necessary to clarify some things that would otherwise be hard to understand.

B.1 Single Sum Virial in GROMACS

The virial Ξ can be written in full tensor form as:

$$\Xi \;=\; -\frac{1}{2}\sum_{i<j}^{N} \boldsymbol{r}_{ij} \otimes \boldsymbol{F}_{ij} \tag{B.1}$$

where $\otimes$ denotes the *direct product* of two vectors.[1] When this is computed in the inner loop of an MD program 9 multiplications and 9 additions are needed.[2]

Here it is shown how it is possible to extract the virial calculation from the inner loop [166].

B.1.1 Virial

In a system with , the periodicity must be taken into account for the virial:

$$\Xi \;=\; -\frac{1}{2}\sum_{i<j}^{N} \boldsymbol{r}^{n}_{ij} \otimes \boldsymbol{F}_{ij} \tag{B.2}$$

where $\boldsymbol{r}^{n}_{ij}$ denotes the distance vector of the *nearest image* of atom i from atom j. In this definition we add a *shift vector* δ_i to the position vector $\boldsymbol{r}_i$ of atom i. The difference vector $\boldsymbol{r}^{n}_{ij}$ is thus equal to:

$$\boldsymbol{r}^{n}_{ij} \;=\; \boldsymbol{r}_i + \delta_i - \boldsymbol{r}_j \tag{B.3}$$

or in shorthand:

$$\boldsymbol{r}^{n}_{ij} \;=\; \boldsymbol{r}^{n}_{i} - \boldsymbol{r}_j \tag{B.4}$$

[1] $(\mathbf{u} \otimes \mathbf{v})^{\alpha\beta} \;=\; \mathbf{u}_\alpha \mathbf{v}_\beta$

[2] The calculation of Lennard-Jones and Coulomb forces is about 50 floating point operations.

In a triclinic system, there are 27 possible images of i; when a truncated octahedron is used, there are 15 possible images.

B.1.2 Virial from non-bonded forces

Here the derivation for the single sum virial in the *non-bonded force* routine is given. $i \neq j$ in all formulae below.

$$\begin{aligned}
\Xi &= -\frac{1}{2}\sum_{i<j}^{N} \boldsymbol{r}_{ij}^{n} \otimes \boldsymbol{F}_{ij} && \text{(B.5)}\\
&= -\frac{1}{4}\sum_{i=1}^{N}\sum_{j=1}^{N} (\boldsymbol{r}_i + \delta_i - \boldsymbol{r}_j) \otimes \boldsymbol{F}_{ij} && \text{(B.6)}\\
&= -\frac{1}{4}\sum_{i=1}^{N}\sum_{j=1}^{N} (\boldsymbol{r}_i + \delta_i) \otimes \boldsymbol{F}_{ij} - \boldsymbol{r}_j \otimes \boldsymbol{F}_{ij} && \text{(B.7)}\\
&= -\frac{1}{4}\left(\sum_{i=1}^{N}\sum_{j=1}^{N} (\boldsymbol{r}_i + \delta_i) \otimes \boldsymbol{F}_{ij} \; - \sum_{i=1}^{N}\sum_{j=1}^{N} \boldsymbol{r}_j \otimes \boldsymbol{F}_{ij}\right) && \text{(B.8)}\\
&= -\frac{1}{4}\left(\sum_{i=1}^{N} (\boldsymbol{r}_i + \delta_i) \otimes \sum_{j=1}^{N} \boldsymbol{F}_{ij} \; - \sum_{j=1}^{N} \boldsymbol{r}_j \otimes \sum_{i=1}^{N} \boldsymbol{F}_{ij}\right) && \text{(B.9)}\\
&= -\frac{1}{4}\left(\sum_{i=1}^{N} (\boldsymbol{r}_i + \delta_i) \otimes \boldsymbol{F}_i \; + \sum_{j=1}^{N} \boldsymbol{r}_j \otimes \boldsymbol{F}_j\right) && \text{(B.10)}\\
&= -\frac{1}{4}\left(2\sum_{i=1}^{N} \boldsymbol{r}_i \otimes \boldsymbol{F}_i + \sum_{i=1}^{N} \delta_i \otimes \boldsymbol{F}_i\right) && \text{(B.11)}
\end{aligned}$$

In these formulae we introduced:

$$\begin{aligned}
\boldsymbol{F}_i &= \sum_{j=1}^{N} \boldsymbol{F}_{ij} && \text{(B.12)}\\
\boldsymbol{F}_j &= \sum_{i=1}^{N} \boldsymbol{F}_{ji} && \text{(B.13)}
\end{aligned}$$

which is the total force on i with respect to j. Because we use Newton's Third Law:

$$\boldsymbol{F}_{ij} = -\boldsymbol{F}_{ji} \qquad \text{(B.14)}$$

we must, in the implementation, double the term containing the shift δ_i.

B.1.3 The intra-molecular shift (mol-shift)

For the bonded forces and SHAKE it is possible to make a *mol-shift* list, in which the periodicity is stored. We simple have an array `mshift` in which for each atom an index in the `shiftvec` array is stored.

The algorithm to generate such a list can be derived from graph theory, considering each particle in a molecule as a bead in a graph, the bonds as edges.

1 Represent the bonds and atoms as bidirectional graph

2 Make all atoms white

3 Make one of the white atoms black (atom i) and put it in the central box

4 Make all of the neighbors of i that are currently white, gray

5 Pick one of the gray atoms (atom j), give it the correct periodicity with respect to any of its black neighbors and make it black

6 Make all of the neighbors of j that are currently white, gray

7 If any gray atom remains, go to [5]

8 If any white atom remains, go to [3]

Using this algorithm we can

- optimize the bonded force calculation as well as SHAKE
- calculate the virial from the bonded forces in the single sum method again

Find a representation of the bonds as a bidirectional graph.

B.1.4 Virial from Covalent Bonds

Since the covalent bond force gives a contribution to the virial, we have:

$$
\begin{aligned}
b &= \|\boldsymbol{r}_{ij}^n\| && \text{(B.15)}\\
V_b &= \frac{1}{2}k_b(b-b_0)^2 && \text{(B.16)}\\
\boldsymbol{F}_i &= -\nabla V_b && \text{(B.17)}\\
&= k_b(b-b_0)\frac{\boldsymbol{r}_{ij}^n}{b} && \text{(B.18)}\\
\boldsymbol{F}_j &= -\boldsymbol{F}_i && \text{(B.19)}
\end{aligned}
$$

The virial contribution from the bonds then is:

$$
\begin{aligned}
\Xi_b &= -\frac{1}{2}(\boldsymbol{r}_i^n \otimes \boldsymbol{F}_i \;+\; \boldsymbol{r}_j \otimes \boldsymbol{F}_j) && \text{(B.20)}\\
&= -\frac{1}{2}\boldsymbol{r}_{ij}^n \otimes \boldsymbol{F}_i && \text{(B.21)}
\end{aligned}
$$

B.1.5 Virial from SHAKE

An important contribution to the virial comes from shake. Satisfying the constraints a force **G** that is exerted on the particles "shaken." If this force does not come out of the algorithm (as in standard SHAKE) it can be calculated afterward (when using *leap-frog*) by:

$$\Delta \boldsymbol{r}_i = \boldsymbol{r}_i(t+\Delta t) - [\boldsymbol{r}_i(t) + \mathbf{v}_i(t-\frac{\Delta t}{2})\Delta t + \frac{\boldsymbol{F}_i}{m_i}\Delta t^2] \tag{B.22}$$

$$\mathbf{G}_i = \frac{m_i \Delta \boldsymbol{r}_i}{\Delta t^2} \tag{B.23}$$

This does not help us in the general case. Only when no periodicity is needed (like in rigid water) this can be used, otherwise we must add the virial calculation in the inner loop of SHAKE.

When it *is* applicable the virial can be calculated in the single sum way:

$$\Xi = -\frac{1}{2}\sum_i^{N_c} \boldsymbol{r}_i \otimes \boldsymbol{F}_i \tag{B.24}$$

where N_c is the number of constrained atoms.

B.2 Optimizations

Here we describe some of the algorithmic optimizations used in GROMACS, apart from parallelism. One of these, the implementation of the 1.0/sqrt(x) function is treated separately in sec. B.3. The most important other optimizations are described below.

B.2.1 Inner Loops for Water

GROMACS uses special inner loops to calculate non-bonded interactions for water molecules with other atoms, and yet another set of loops for interactions between pairs of water molecules. There highly optimized loops for two types of water models. For three site models similar to SPC [81], *i.e.*:

1. There are three atoms in the molecule.
2. The whole molecule is a single charge group.
3. The first atom has Lennard-Jones (sec. 4.1.1) and Coulomb (sec. 4.1.3) interactions.
4. Atoms two and three have only Coulomb interactions, and equal charges.

These loops also works for the SPC/E [167] and TIP3P [125] water models. And for four site water models similar to TIP4P [125]:

1. There are four atoms in the molecule.
2. The whole molecule is a single charge group.

3. The first atom has only Lennard-Jones (sec. 4.1.1) interactions.
4. Atoms two and three have only Coulomb (sec. 4.1.3) interactions, and equal charges.
5. Atom four has only Coulomb interactions.

The benefit of these implementations is that there are more floating-point operations in a single loop, which implies that some compilers can schedule the code better. However, it turns out that even some of the most advanced compilers have problems with scheduling, implying that manual tweaking is necessary to get optimum performance. This may include common-sub-expression elimination, or moving code around.

B.2.2 Fortran Code

Unfortunately, on a few platforms Fortran compilers are still better than C-compilers. For some machines (*e.g.* SGI Power Challenge) the difference may be up to a factor of 3, in the case of vector computers this may be even larger. Therefore, some of the routines that take up a lot of computer time have been translated into Fortran and even assembly code for Intel and AMD x86 processors. In most cases, the Fortran or assembly loops should be selected automatically by the `configure` script when appropriate, but you can also tweak this by setting options to the `configure` script.

B.3 Computation of the 1.0/sqrt function

B.3.1 Introduction

The GROMACS project started with the development of a $1/\sqrt{x}$ processor that calculates:

$$Y(x) = \frac{1}{\sqrt{x}} \tag{B.25}$$

As the project continued, the Intel *i*860 processor was used to implement GROMACS, which now turned into almost a full software project. The $1/\sqrt{x}$ processor was implemented using a Newton-Raphson iteration scheme for one step. For this it needed look-up tables to provide the initial approximation. The $1/\sqrt{x}$ function makes it possible to use two almost independent tables for the exponent seed and the fraction seed with the IEEE floating-point representation.

B.3.2 General

According to [168] the $1/\sqrt{x}$ function can be evaluated using the Newton-Raphson iteration scheme. The inverse function is:

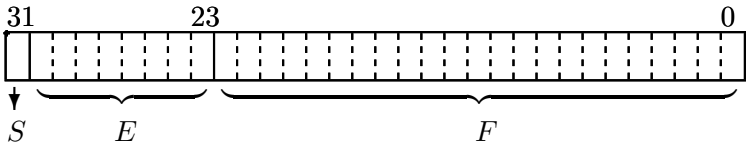
$$X(y) = \frac{1}{y^2} \tag{B.26}$$

So instead of calculating:

$$Y(a) = q \tag{B.27}$$

the equation:

$$X(q) - a = 0 \tag{B.28}$$



$$Value = (-1)^S(2^{E-127})(1.F)$$

Figure B.1: IEEE single-precision floating-point format

can now be solved using Newton-Raphson. An iteration is performed by calculating:

$$y_{n+1} = y_n - \frac{f(y_n)}{f'(y_n)} \tag{B.29}$$

The absolute error ε, in this approximation is defined by:

$$\varepsilon \equiv y_n - q \tag{B.30}$$

Using Taylor series expansion to estimate the error results in:

$$\varepsilon_{n+1} = -\frac{\varepsilon_n^2}{2}\frac{f''(y_n)}{f'(y_n)} \tag{B.31}$$

according to [168] equation (3.2). This is an estimation of the absolute error.

B.3.3 Applied to floating-point numbers

Floating-point numbers in IEEE 32 bit single-precision format have a nearly constant relative error of $\Delta x/x = 2^{-24}$. As seen earlier in the Taylor series expansion equation (eqn. B.31), the error in every iteration step is absolute and in general dependent of y. If the error is expressed as a relative error ε_r the following holds:

$$\varepsilon_{r_{n+1}} \equiv \frac{\varepsilon_{n+1}}{y} \tag{B.32}$$

and so:

$$\varepsilon_{r_{n+1}} = -(\frac{\varepsilon_n}{y})^2 y\frac{f''}{2f'} \tag{B.33}$$

For the function $f(y) = y^{-2}$ the term $yf''/2f'$ is constant (equal to $-3/2$) so the relative error ε_{r_n} is independent of y.

$$\varepsilon_{r_{n+1}} = \frac{3}{2}(\varepsilon_{r_n})^2 \tag{B.34}$$

The conclusion of this is that the function $1/\sqrt{x}$ can be calculated with a specified accuracy.

B.3.4 Specification of the look-up table

To calculate the function $1/\sqrt{x}$ using the previously mentioned iteration scheme, it is clear that the first estimation of the solution must be accurate enough to get precise results. The requirements for the calculation are

- Maximum possible accuracy with the used IEEE format
- Use only one iteration step for maximum speed

The first requirement states that the result of $1/\sqrt{x}$ may have a relative error ε_r equal to the ε_r of a IEEE 32 bit single-precision floating-point number. From this, the $1/\sqrt{x}$ of the initial approximation can be derived, rewriting the definition of the relative error for succeeding steps (eqn. B.34):

$$\frac{\varepsilon_n}{y} = \sqrt{\varepsilon_{r_{n+1}} \frac{2f'}{yf''}} \tag{B.35}$$

So for the look-up table the needed accuracy is:

$$\frac{\Delta Y}{Y} = \sqrt{\frac{2}{3} 2^{-24}} \tag{B.36}$$

which defines the width of the table that must be ≥ 13 bit.

At this point the relative error, ε_{r_n}, of the look-up table is known. From this the maximum relative error in the argument can be calculated as follows. The absolute error Δx is defined as:

$$\Delta x \equiv \frac{\Delta Y}{Y'} \tag{B.37}$$

and thus:

$$\frac{\Delta x}{Y} = \frac{\Delta Y}{Y} (Y')^{-1} \tag{B.38}$$

and thus:

$$\Delta x = constant \frac{Y}{Y'} \tag{B.39}$$

For the $1/\sqrt{x}$ function, $Y/Y' \sim x$ holds, so $\Delta x/x = constant$. This is a property of the used floating-point representation as earlier mentioned. The needed accuracy of the argument of the look-up table follows from:

$$\frac{\Delta x}{x} = -2 \frac{\Delta Y}{Y} \tag{B.40}$$

So, using the floating-point accuracy (eqn. B.36):

$$\frac{\Delta x}{x} = -2 \sqrt{\frac{2}{3} 2^{-24}} \tag{B.41}$$

This defines the length of the look-up table which should be ≥ 12 bit.

B.3.5 Separate exponent and fraction computation

The used IEEE 32 bit single-precision floating-point format specifies that a number is represented by a exponent and a fraction. The previous section specifies for every possible floating-point number the look-up table length and width. Only the size of the fraction of a floating-point number defines the accuracy. The conclusion from this can be that the size of the look-up table is length of look-up table, earlier specified, times the size of the exponent ($2^{12}2^8, 1Mb$). The $1/\sqrt{x}$ function has the property that the exponent is independent of the fraction. This becomes clear if the floating-point representation is used. Define:

$$x \equiv (-1)^S(2^{E-127})(1.F) \tag{B.42}$$

See Fig. B.1, where $0 \leq S \leq 1$, $0 \leq E \leq 255$, $1 \leq 1.F < 2$ and S, E, F integer (normalization conditions). The sign bit (S) can be omitted because $1/\sqrt{x}$ is only defined for $x > 0$. The $1/\sqrt{x}$ function applied to x results in:

$$y(x) = \frac{1}{\sqrt{x}} \tag{B.43}$$

or:

$$y(x) = \frac{1}{\sqrt{(2^{E-127})(1.F)}} \tag{B.44}$$

This can be rewritten as:

$$y(x) = (2^{E-127})^{-1/2}(1.F)^{-1/2} \tag{B.45}$$

Define:

$$(2^{E'-127}) \equiv (2^{E-127})^{-1/2} \tag{B.46}$$

$$1.F' \equiv (1.F)^{-1/2} \tag{B.47}$$

Then $\frac{1}{\sqrt{2}} < 1.F' \leq 1$ holds, so the condition $1 \leq 1.F' < 2$, which is essential for normalized real representation, is not valid anymore. By introducing an extra term, this can be corrected. Rewrite the $1/\sqrt{x}$ function applied to floating-point numbers (eqn. B.45) as:

$$y(x) = (2^{\frac{127-E}{2}-1})(2(1.F)^{-1/2}) \tag{B.48}$$

and:

$$(2^{E'-127}) \equiv (2^{\frac{127-E}{2}-1}) \tag{B.49}$$

$$1.F' \equiv 2(1.F)^{-1/2} \tag{B.50}$$

Then $\sqrt{2} < 1.F \leq 2$ holds. This is not the exact valid range as defined for normalized floating-point numbers in eqn. B.42. The value 2 causes the problem. By mapping this value on the nearest representation < 2, this can be solved. The small error that is introduced by this approximation is within the allowable range.

The integer representation of the exponent is the next problem. Calculating $(2^{\frac{127-E}{2}-1})$ introduces a fractional result if $(127 - E) = odd$. This is again easily accounted for by splitting up the calculation into an odd and an even part. For $(127 - E) = even$ E' in equation (eqn. B.49) can be exactly calculated in integer arithmetic as a function of E.

$$E' = \frac{127 - E}{2} + 126 \tag{B.51}$$

For $(127 - E) = odd$ equation (eqn. B.45) can be rewritten as:

$$y(x) = (2^{\frac{127-E-1}{2}})(\frac{1.F}{2})^{-1/2} \tag{B.52}$$

Thus:

$$E' = \frac{126 - E}{2} + 127 \tag{B.53}$$

which also can be calculated exactly in integer arithmetic. **Note** that the fraction is automatically corrected for its range earlier mentioned, so the exponent does not need an extra correction.

The conclusions from this are:

- The fraction and exponent look-up table are independent. The fraction look-up table exists of two tables (odd and even exponent) so the odd/even information of the exponent (lsb bit) has to be used to select the right table.
- The exponent table is an 256 x 8 bit table, initialized for odd and $even$.

B.3.6 Implementation

The look-up tables can be generated by a small C program, which uses floating-point numbers and operations with IEEE 32 bit single-precision format. Note that because of the $odd/even$ information that is needed, the fraction table is twice the size earlier specified (13 bit i.s.o. 12 bit).

The function according to eqn. B.29 has to be implemented. Applied to the $1/\sqrt{x}$ function, equation eqn. B.28 leads to:

$$f = a - \frac{1}{y^2} \tag{B.54}$$

and so:

$$f' = \frac{2}{y^3} \tag{B.55}$$

so:

$$y_{n+1} = y_n - \frac{a - \frac{1}{y_n^2}}{\frac{2}{y_n^3}} \tag{B.56}$$

or:

$$y_{n+1} = \frac{y_n}{2}(3 - ay_n^2) \tag{B.57}$$

Where y_0 can be found in the look-up tables, and y_1 gives the result to the maximum accuracy. It is clear that only one iteration extra (in double precision) is needed for a double-precision result.