GROMACS - Bug #2562

Clarify how to do float->int rounding

06/27/2018 08:18 PM - Roland Schulz

Status:	Closed		
Priority:	Normal		
Assignee:			
Category:			
Target version:	2019		
Affected version - extra info:		**Difficulty:**	uncategorized
Affected version:	git master		

Description

In most places we round float->int using (int)f+0.5. That has multiple problems:
- incorrect for 0.5-eps (gives 1 instead of 0)
- incorrect for negative numbers (without any check/assert and easy to miss in code review - I suspect multiple real issues in code)
- unnecessary slow (compared to some possible solutions)
- inconsistent with rounding used for float->float (round half away from zero vs round half to even)

What should the default rounding mode be?
- Round half up: Convention usually used outside of computers. Inefficient. No C++ function.
- Round half away from zero: Matches lround(). Somewhat efficient.
- Round half to even: Matches fp rounding and lrint(). Most efficient.

Possible solutions:
1) Round half away from zero with fast option: Create fastround() for performance critical code for positive inputs. It assert that the input is positive and uses +.5 otherwise. Document that it is wrong for 0.5-eps. Use lround/lroundf for non performance critical code.
2) Round half away from zero always correct: Always use lround/lroundf.
3) Round half to even: Always use lrint
4) Mix modes. Use lrint for performance critical code or where consistency with fp rounding is better. Use lround/lroundf otherwise.

Associated revisions

Revision 88754544 - 08/27/2018 10:44 AM - Roland Schulz

Enable more clang-tidy checks for new code

Some of the clang-tidy checks would require too many code
changes when enabled globally. To get the benefits from
the checks without changing all legacy code add a second
configuration. Which configuration is used is selected by
directory: Any new module should use the new configuration.

Additionally it is recommended to use the new configuration
on a per file or change basis (clang-tidy-diff.py) manually.

This change contains the proposed configuration for new code and
and enables it for three folders which contain new code.

Checks enabled relative to base configuration:
clang-analyzer-security.insecureAPI.strcpy
readability-inconsistent-declaration-parameter-name
readability-function-size
cppcoreguidelines-owning-memory
cppcoreguidelines-no-malloc
cppcoreguidelines-pro-type-member-init
cppcoreguidelines-pro-type-union-access

The reasons that the remaining checks are still disabled (don't
know a good place to document because json doesn't allow comments):
misc-incorrect-roundings: #2562
readability-else-after-return: I don't think it is something we want to follow
cppcoreguidelines-pro-type-*cast: While it is best to not do those cast there
are valid reasons for them. And because they are already easy to spot,
mandating a NOLINT for those is probably not helpful.

cppcoreguidelines-special-member-functions: conflicts with Wunused-member
cppcoreguidelines-pro-type-vararg: #2570
cppcoreguidelines-pro-bounds-constant-array-index: While I think it would be
nice to have a compile mode with bound checking enabled. It is so ugly
to have to write at(v, n) instead of v[n].
cppcoreguidelines-pro-bounds-array-to-pointer-decay: This would be particular
nice to have But it makes it very hard to write warning free code with
legacy APIs which uses e.g. rvec. Those have to be modernized first.
cppcoreguidelines-pro-bounds-pointer-arithmetic: This would also be very nice.
But it requires that depending APIs use ArrayRef rather than pointers
for non-owning array passing.

Change-Id: I891a576d2c185ef6587224a1a19324f1a8967237

Revision d854e03e - 09/10/2018 09:41 AM - Roland Schulz

Add roundToInt

- Leaves the rounding of halfway cases implementation defined.
Reason: It simplifies implementation and has no impact for
anything where the input can have at least 1ulp error.
- Uses rint(f) because it is optimized in all commonly used
compilers avoiding the use of intrinsics.

Change-Id: Id01c2a37d3e1c5000858e78332fa72beb5b3b58d
Fixes: #2562

History

#1 - 06/28/2018 07:20 PM - Eric Irrgang

Roland Schulz wrote:

> 4) Mix modes. Use lrint for performance critical code or where consistency with fp rounding is better. Use lround/lroundf otherwise.

It seems like the default should be that the function does what it says it does, but that fast routines should be available. Doesn't something similar come up for evaluating exponentials? Should the developer pick and choose which version they want for each math operation used with explicitly incompatible signatures or should they just toggle all implementations by using either `math` or `fastmath` with the same API? I'm inclined to the latter. For one thing, it would make for easier testing of whether performance or precision were more important in a given translation unit or even leave it to the user to decide with a CMake option.

#2 - 06/28/2018 09:47 PM - Roland Schulz

I agree clarity is the most important thing. And the current approach of static_cast<int>(x+0.5) isn't clear at all.
Do you mean we shouldn't use the standard rounding functions directly but have a gmx_round function which can be called both as gmx_round(x) and gmx_round<MathOptimization::Unsafe>(x)? gmx_round would call lround/lroundf I assume? What would gmx_round<MathOptimization::Unsafe> do? Would it use static_cast<int>(x+0.5) or lrint?
Note that this isn't just about safe and unsafe. I would consider lrint safe. It never gives incorrect results. But it uses a different rounding model which can be better or worse depending on the use case.

I think static_cast<int>(x+0.5) has no advantage over lrint. It is inaccurate and slower.
Maybe we should have
gmx_round<FastToEven>: Uses even rounding and is fast. Calls rlrint
gmx_round<SlowToZero>: Uses to zero rounding. Call lround/lroundf

Not sure which of the to should be the default (or whether there should be default). But would be clearer than using lround/lrint because those names are not very clear. Disadvantage is that it is non-std and thus unfamiliar to people new to GROMACS. Also it would be very verbose especially if we choose not to have a default.

#3 - 06/29/2018 09:53 AM - Berk Hess

Doesn't lrint cause overhead because it returns a long, while we nearly always use int?

I think we can use gmx_round to round to nearest, fast.

Do we ever need round to zero for negative numbers? If not, we can simply use static_cast for (positive) numbers, or?

#4 - 06/30/2018 03:08 PM - Eric Irrgang

Roland Schulz wrote:

> gmx_round<FastToEven>: Uses even rounding and is fast. Calls rlrint
> gmx_round<SlowToZero>: Uses to zero rounding. Call lround/lroundf

On a related note, what are the thoughts on how to make such a helper easily discoverable in the documentation. It seems like the availability and encouragement to use such functions should be within a couple of clicks of http://manual.gromacs.org/documentation/current/dev-manual/index.html but I don't see an appropriate place to stick it. Clicking through the Doxygen documentation, it is hard to find conventional GROMACS alternatives even if you know to look for them. (In the "library" documentation build, you can click through and discover the "math" header directory, but most of the files don't expose documentation there.)

My question is just: how strongly developers would be encouraged to use such functions and what documentation, if any, would support that intention, or whether this would be just a quietly internally documented function for use in a handful of cherry-picked places?

#5 - 06/30/2018 07:23 PM - Roland Schulz

I think the best enforcement is clang-tidy. That way we don't have to catch things in code-review and no one has to read+remember it. The reason I looked at it was part of my clang-tidy effort. It has a check for the (int)(x+0.5) pattern: http://releases.llvm.org/6.0.0/tools/clang/tools/extra/docs/clang-tidy/checks/misc-incorrect-roundings.html. I think enabling it would be good. But of course before we enable it we need to decide what to replace it with. If we also want to have a strict rule to only use gmx_round and not lround/rint/... we could add a custom check for that.

#6 - 07/03/2018 04:58 AM - Roland Schulz

> Doesn't lrint cause overhead because it returns a long, while we nearly always use int?

64bit shouldn't be an issue. Doing operations in 64bit should be as fast on supported HW (we don't support 32bit HW anymore) and 64bit->32bit conversion is free.

```
#include <cmath>
int f(float f) {
    return lrintf(f);
}
```

gets compiled to a single instruction in GCC (-O3 -ffast-math, >=4.81, X86-64). In comparison (int)(f+.5) is 3 instructions (float->double, add, trunc) and (int)(f+.5f) is 2 instructions (add, trunc). But other compilers are being stupid and turn this into a function call (but no extra instruction for 64->32). For llvm this is a known bug https://bugs.llvm.org/show_bug.cgi?id=22944 . Thus we probably would need to actually implement it ourselves if we wanted to not have performance regression on other compilers :(. Or use lrint for gcc and (int)(f+.5f) for other compilers.

> Do we ever need round to zero for negative numbers? If not, we can simply use static_cast for (positive) numbers, or?

Looking at the warning given by clang-tidy misc-incorrect-roundings checks, there were multiple ones which looked suspicious and I assumed have a possibility to be negative. But I'm not sure. But I do think it is easy to accidentally get wrong.
I'm also not sure whether the error at 0.5-eps matters somewhere.

#7 - 08/06/2018 06:43 AM - Gerrit Code Review Bot

Gerrit received a related patchset '6' for Issue #2562.
Uploader: Roland Schulz (roland.schulz@intel.com)
Change-Id: gromacs~master~I891a576d2c185ef6587224a1a19324f1a8967237
Gerrit URL: https://gerrit.gromacs.org/8130

#8 - 08/06/2018 10:48 AM - Mark Abraham

Considering all the thoughts above, I suggest convenience functions in real.h that provide

- gmx_round defaults to gmx_round<SlowToZero>, implemented with lround/lroundf
- gmx_round<FastToEven> is implemented as lrint/lrintf, after asserting the input is non-negative, quirks documented. If needed, we call the functions on gcc and the addition of 0.5 elsewhere, but if we can copy the glibc implementation of lrintf that might be best (even an intrinsic if we have to).

A preliminary commit calls the default one in most parts of the code e.g. where performance is less critical (preprocessing, analysis, mdrun outside the loop), and a follow-up commit fixes the remaining ones where reviewers can help judge the merits of the two functions when performance is a consideration.

#9 - 08/07/2018 12:01 AM - Roland Schulz

I'm OK with the suggestion. Just one question: Does it really make sense to fall back to int(x+.5) for non-GCC and non-intrinsic? We don't know whether that is actually faster than lrintf for those compiler+HW and even if it is, it is likely a minor optimization compared to other things and is trivially to add an intrinsic. The advantage, if we always use lrintf for non-intrinsic (including non-GCC), is that we don't need the assert and we always have the same rounding behavior.

#10 - 08/07/2018 09:24 PM - Gerrit Code Review Bot

Gerrit received a related patchset '1' for Issue #2562.

Uploader: Roland Schulz (roland.schulz@intel.com)
Change-Id: gromacs~master~I107a2322fd9c0055ba72688f5711f06568a41d88
Gerrit URL: https://gerrit.gromacs.org/8150

#11 - 08/09/2018 01:36 AM - Gerrit Code Review Bot

Gerrit received a related patchset '1' for Issue #2562.
Uploader: Roland Schulz (roland.schulz@intel.com)
Change-Id: gromacs~master~Id01c2a37d3e1c5000858e78332fa72beb5b3b58d
Gerrit URL: https://gerrit.gromacs.org/8154

#12 - 09/10/2018 09:45 AM - Roland Schulz

- Status changed from New to Resolved

Applied in changeset d854e03ef1f5ad62705bfc941552eca6a6d3907c.

#13 - 11/28/2018 03:34 PM - Paul Bauer

- Status changed from Resolved to Closed

#14 - 11/28/2018 03:41 PM - Mark Abraham

- Target version set to 2019