An Emacs-Cairo Scrolling Bug due to Floating-Point Inaccuracy

Vincent Lefèvre

An Emacs-Cairo Scrolling Bug due to Floating-Point Inaccuracy

Vincent Lefèvre
Univ Lyon, EnsL, UCBL, CNRS, Inria, LIP, F-69342, LYON Cedex 07, France
Email: Vincent.Lefevre@inria.fr

***Abstract*—We study a bug that we found in the GNU Emacs text editor when built against the Cairo graphics library. We analyze both the Emacs code and the Cairo code, and we suggest what can be done to avoid unexpected results. This involves a particular case with a computation that can be reduced to the equivalent floating-point expression $((1/s) \cdot b) \cdot s$, where s and b are small positive integers such that $b < s$ and the basic operations are rounded to nearest. The analysis takes into account the values of s and b that can occur in practice, and the suggestions for workarounds must avoid handling this particular case in a separate branch or breaking the structure of the Cairo library (so that just returning b is not possible).**

***Index Terms*—floating-point, rounding, discontinuity**

I. Introduction

Floating-point arithmetic is used in many places nowadays, and it is sometimes the cause of surprising behaviors, in particular in contexts where the use of floating point is not expected by the user. Even though the IEEE 754 standard, first published in 1985 [1] and revised in 2008 and 2019 [2], has improved the floating-point behavior a lot and is nowadays well supported by the processors, developers still need to be careful when using floating-point numbers.

A new bug appeared in the GNU Emacs text editor under Linux (Debian/unstable) in October 2020, after an upgrade of the emacs-gtk package from version 26 to version 27: backward scrolling with the mouse wheel was almost inoperative; for instance, after 40 scrolling steps, the text was moved 24 lines downward instead of 200 lines. An analysis showed that this bug was related to issues in code using floating-point arithmetic (with a lack of specification in the software).

First, let us note that there exist long-standing bugs related to floating-point arithmetic with current compilers, here GCC. However, we could check that the code (at least the part that was analyzed) was not affected by such bugs. So, in the following, we may assume that the floating-point implementation is perfect and that the behavior strictly follows the IEEE 754 standard. The concerned operations were already specified in the 1985 version of the standard.

In Section II, this scrolling bug is described in more details, and the codes of the involved software are analyzed to find the causes of this bug. In Section III, we study the behavior of the floating-point computation that is at the root of this bug and of an alternate computation that can be used by slightly changing the code. We conclude in Section IV, with some discussion.

II. The Emacs-Cairo scrolling bug

Backward scrolling with the mouse wheel normally works as follows in GNU Emacs: for each scrolling step, the text is moved downward by 5 lines (default scroll amount).[1] If possible, the cursor remains at the same place in the text, thus is moved downward in the window, with the text; otherwise, to avoid the cursor moving out of the window, Emacs puts it on the last visible line. So, after several scrolling steps, the cursor normally stays on the last visible line.

However, under some particular conditions, backward scrolling appears to be much slower than expected. For instance, after 40 scrolling steps, the text is moved 24 lines downward instead of the expected 200 lines. From some testing, we have found that this issue occurs only with bitmap fonts of size 13, and only when Emacs was built against the Cairo graphics library (Cairo version at this time: 1.16.0) instead of the old Xft library, which is the case in Debian for the GTK+ Emacs 27 version. The cause of this issue could be found with a further study, as explained below.

As said, after some scrolling steps, the cursor should end up on the last visible text line, i.e., the bottom line in the window. But with the culprit font, for some reason that will be determined later, the cursor cannot always be put on this last visible text line. This triggers an unexpected cursor repositioning at the center of the window (called *recentering*), with the effect of scrolling forward, i.e., in the opposite direction. Thus, when scrolling backward, this recentering more or less cancels the scrolling done since the last time the cursor was near the center. This is not quite regular, but in average, the text still moves downward after a large number of scrolling steps.

At this moment, as the issue was occurring only with a bitmap font of a particular size (scalable fonts do not seem to be affected) and only with Cairo (which is able to do font scaling and rotations, thus internally uses floating-point arithmetic), we suspected something wrong with rounding.

A. The Emacs side

The place where the error was introduced could be found by backtracking. The test was done by evaluating the LISP expression `(move-to-window-line -1)`, whose effect is to move the cursor (actually, what is called *point* by

[1] https://git.savannah.gnu.org/cgit/emacs.git/tree/lisp/mwheel.el?h=emacs-27&id=b468b3d1ffa9

Emacs) to the beginning of the last fully visible screen line of the window. With font size 13, this triggers an unexpected recentering, while with the other font sizes (such as 14, chosen below), this does not. The values of some variables were traced with font size 13 and with font size 14, so that differences could be observed, i.e., one can notice what values are unexpected with size 13.

A first difference could be found in the return value of the `try_cursor_movement` function of `xdisp.c`. In short, this function tests whether recentering is needed. The difference comes from the fact that with size 13, the bottom pixel position of the last text line is one more the position of the last pixel line of the window (in the Emacs code, the test `MATRIX_ROW_BOTTOM_Y (row) > last_y` for this text line gives false, while it is expected to be true in this context). So this means that the last text line is not fully visible, hence the recentering.

All the text lines are actually shifted one pixel downward: `MATRIX_ROW_BOTTOM_Y (row)` is of the form $13n + 1$ instead of $13n$. This issue can actually be seen on screen (at least with a magnifier), but it is hardly noticeable when one is not aware of it.

Another difference between the two font sizes could be found in earlier code: the `compute_line_metrics` function, concerning the first text line. With size 13, its height is increased from 13 to 14 (hence the observed issue). This occurs because here, the physical ascent `row->phys_ascent` of the first line is larger than its logical ascent `row->ascent`; according to the comment associated with this code in Emacs, the reason is to make accented characters fully visible. However, in the present case, this is an inconsistency with size 13: both ascents should normally be equal in our context. In short, with size 13, `row->phys_ascent` should be 11 instead of 12. This was confirmed when rebuilding Emacs without Cairo, where one gets 11 for both ascents.

The source of this value for Cairo is the `ftcrfont_glyph_extents` function of `ftcrfont.c` (the FreeType font driver on Cairo), which does

```
cache->ascent = ceil (- extents.y_bearing);
```

where `extents.y_bearing` is of type `double`, i.e., it is a binary64 floating-point value. Here, its exact value is `-0x1.6000000000001p+3`, which is the binary64 number just below -11. Due to the use of the ceiling function on the negated value, one gets 12 instead of 11.

Note that among the available bitmap fonts, other non-integer `extents.y_bearing` values could be obtained with some font sizes, but they are above the integer instead of below, so that in these cases, the `ceil` function returns the expected value.

A possible workaround to take into account this (undocumented) inaccuracy in Cairo would be to consider that a value that is close enough to an integer would be regarded as being this integer. This has been the fix applied for GNU Emacs 28.1, following this suggestion:

```
cache->ascent =
  ceil (- extents.y_bearing - 1.0 / 256);
```

This could imply that for other kinds of fonts, something like a glyph not being fully visible in its cell, but the missing part would just be $1/256$-pixel tall, thus not noticeable.

Details about this bug on the Emacs side and suggested fix can be found in the GNU bug 44284 [2] and the bug fix in commit `33e2418a7cfd` [3]. We could check that this fix made the issue disappear and did not introduce new problems with other fonts.

Now, even though a workaround could be applied in Emacs, one may wonder the cause of the inaccurate value in Cairo.

B. The Cairo side

An analysis of the computations done on the Cairo side shows that one gets the result `y` of a transformation performed by the `cairo_matrix_transform_point` function, which does an affine transformation:

```
new_x = matrix->xx * x + matrix->xy * y;
new_y = matrix->yx * x + matrix->yy * y;
x = new_x + matrix->x0;
y = new_y + matrix->y0;
```

In our particular case, `matrix->xx` and `matrix->yy` are both equal to the font size (e.g., 13), and the other matrix components are 0. Though the above code does not provide correct rounding in general, here, it just reduces to a multiplication of the `y` input by the font size, which is thus correctly rounded (as the additions of ± 0 have no effect). The `y` input comes from

```
y = fs_metrics->y_bearing +
    fs_metrics->height * hm;
```

where `hm` is 0; thus is it just `fs_metrics->y_bearing`, and for size 13, its value is `-0x1.b13b13b13b13cp-1`, which is an approximation to $-11/13$ (but not the one to nearest). This value has been computed with

```
fs_metrics.y_bearing =
  DOUBLE_FROM_26_6 (-metrics->horiBearingY)
  * y_factor;
```

where

- the value of `metrics->horiBearingY` comes from the FreeType 2 font-engine library and is an integer representing a 26.6 fixed-point number (see the `FT_Glyph_Metrics` structure in `freetype.h`);
- the `DOUBLE_FROM_26_6` macro converts this integer to `double` (binary64), doing a division by $64 = 2^6$;
- here, `y_factor` is computed by `1 / unscaled->y_scale`, where `y_scale` is a binary64 value that corresponds to the font size (e.g., 13). In this context, it actually comes from the FreeType 2 library and is in the 26.6 fixed-point number format (thus in general, it does not necessarily represent an integer).

For the bitmap font of size 13, the `horiBearingY` component represents the integer value 11. So, in short, one gets $\mathrm{RN}(\mathrm{RN}(-11 \cdot \mathrm{RN}(1/13)) \cdot 13)$, where the rounding function RN corresponds to the default rounding mode, which is roundTiesToEven (rounding to nearest, ties to even).

[2] https://debbugs.gnu.org/cgi/bugreport.cgi?bug=44284

[3] https://git.savannah.gnu.org/cgit/emacs.git/commit/?id=33e2418a7cfd

One may wonder why Cairo does a multiplication by `1 / unscaled->y_scale` instead of just a division by `unscaled->y_scale`, which should be more accurate (this will be confirmed as shown in Section III). This could be regarded as a very minor optimization (probably not noticeable) as multiplication is faster than division (`y_factor` is used up to 3 times in a same branch), but this is not the reason. This code was introduced in 2005 (before version 1.0.0) in order to avoid a potential division by zero[4] as an attempt to fix a crash[5]. However, a value 0 may be caused by some other major issue, and this change did not make the crash disappear. Unfortunately, this new, less accurate code remained.

In the next section, we analyze the possible consequences of a revert of this change (when `y_scale` is not zero).

III. The FP expressions $((1/s) \cdot b) \cdot s$ and $(b/s) \cdot s$

We now compare the results of the floating-point expressions $((1/s) \cdot b) \cdot s$ and $(b/s) \cdot s$, where each arithmetic operation is rounded to nearest (roundTiesToEven), that is, more formally, $\mathrm{RN}(\mathrm{RN}(\mathrm{RN}(1/s) \cdot b) \cdot s)$ vs $\mathrm{RN}(\mathrm{RN}(b/s) \cdot s)$. We assume that the inputs s (for *size*) and b (for *bearing*, in absolute value) are positive integers such that $b \leqslant s$. In practice, they are exactly representable as floating-point numbers (*FP numbers*).

Note that the possible pairs (s, b) are limited in practice: bitmap fonts normally do not have large sizes, for which outline fonts are normally used instead, not all sizes up to the maximum are used, and the value of b is not far from the value of s (for instance, with the misc-fixed fonts, $b/s \geqslant 7/9$).

First, let us prove that if $\mathrm{RN}(\mathrm{RN}(\mathrm{RN}(1/s) \cdot b) \cdot s) = b$, then $\mathrm{RN}(\mathrm{RN}(b/s) \cdot s) = b$ too, i.e. that the change of the expression cannot introduce an error that was not present with the old expression; this is not obvious as rounding errors may partially compensate, if rounding occurs in different directions. Here, we just assume that the radix is 2, that s and b are arbitrary finite FP numbers such that $s \neq 0$, and that overflows or underflows cannot occur (this is the case here, where s and b are integers that cannot be very large).

Proof:

- Let $u = \mathrm{RN}(\mathrm{RN}(1/s) \cdot b)$ and $\varepsilon_u = u - b/s$. Then $u \cdot s = b + s \cdot \varepsilon_u$ and by hypothesis, $\mathrm{RN}(u \cdot s) = b$.
- Let $v = \mathrm{RN}(b/s)$ and $\varepsilon_v = v - b/s$. Then $v \cdot s = b + s \cdot \varepsilon_v$ and we want to prove that $\mathrm{RN}(v \cdot s) = b$.

Let us recall that except in case of overflow (excluded here), x being a real number, the correct rounding of x to nearest $\mathrm{RN}(x)$ minimizes the rounding error (in magnitude) among the FP numbers. This error is at most $\frac{1}{2}\,\mathrm{ulp}(x)$, where $\mathrm{ulp}(x)$ is the unit in the last place of x: if p denotes the precision and $|x| \in [2^e, 2^{e+1})$, then $\mathrm{ulp}(x) = 2^{e-p+1}$. Since u and v are FP numbers and v is the correct rounding of b/s, we have $|\varepsilon_v| \leqslant |\varepsilon_u|$. Moreover, since in radix 2, the quotient of FP numbers cannot be the midpoint between two consecutive normal FP numbers [3][4, p. 131], we have: if $|\varepsilon_v| = |\varepsilon_u|$, then $v = u$ (unicity of the FP number closest to b/s). Now, either $v \cdot s = u \cdot s$ or $|v \cdot s - b| = |s \cdot \varepsilon_v| < |s \cdot \varepsilon_u| = |u \cdot s - b|$. In the latter case, since $\mathrm{RN}(u \cdot s) = b$, we have $|u \cdot s - b| \leqslant \frac{1}{2}\,\mathrm{ulp}(b)$, so that $|v \cdot s - b| < \frac{1}{2}\,\mathrm{ulp}(b)$. If b is not a power of 2 (the radix), this means that $\mathrm{RN}(v \cdot s) = b$; if b is a power of 2, then $u = \mathrm{RN}(1/s) \cdot b = \mathrm{RN}(b/s) = v$. Thus in all cases, $\mathrm{RN}(v \cdot s) = b$. □

[4] https://gitlab.freedesktop.org/cairo/cairo/-/commit/399b00a99b2b

[5] https://bugzilla.gnome.org/show_bug.cgi?id=311299

TABLE I

For each value of s from 3 to 40: the values of b from 1 to s such that $((1/s) \cdot b) \cdot s$ does not give b exactly; a + means a value larger than b, and a − means a value smaller than b. The values for which $(b/s) \cdot s$ does not give b exactly are those with a slash followed by a second + or −. For the font sizes s found on the system, we give the associated value of b.

```
 5: 3+
 6:                                          b =  5
 7: 5-                                       b =  6
 8:                                          b =  7
 9: 7-                                       b =  7
10: 3+ 6+ 7+                                 b =  8
12: 7-                                       b = 10
13: 7+ 11+                                   b = 11
14: 5- 10- 13-                               b = 12
15:                                          b = 12
17: 3- 6- 12-
18: 7- 11- 14- 15-                           b = 14
19: 13- 17-
20: 3+ 6+ 7+ 12+ 14+                         b = 16
21: 11- 15- 19-
22: 15+/-
23: 13-/-
24: 7- 14-
25: 7+/+ 14+/+
26: 7+ 11+ 14+ 15+/- 22+ 23+
27: 17- 21- 25-
28: 5- 10- 13- 19- 20- 26- 27-
29: 7- 14- 15+/+ 28-
31: 9- 13- 18- 26-
33: 11+ 15+ 22+ 30+
34: 3- 6- 12- 24- 25-
35: 7- 14- 27- 28- 29-/+
36: 7- 11- 14- 15- 21- 22- 28- 29- 30- 31-
37: 19+ 23+ 27+ 31+ 35+
38: 13- 17- 21-/+ 25- 26- 29- 34-
39: 15- 25-/+ 30- 31-/-
40: 3+ 6+ 7+ 12+ 14+ 23+ 24+ 28+ 29+
```

For better readability, we will omit RN in the floating-point expressions below, where each arithmetic operation is implicitly rounded to nearest.

For each value of s from 3 to 40, Table I lists the values of b from 1 to s such that $((1/s) \cdot b) \cdot s$ does not evaluate to b exactly; these values are followed by the sign of the error, telling whether the result is larger or smaller than b. When $(b/s) \cdot s$ does not evaluate to b exactly (which is possible only when this is already the case for $((1/s) \cdot b) \cdot s$, as proved above), this sign of this error of this second expression is shown too, after a slash. For our problem, only the sign of the error (zero, positive or negative) really matters; so we do not care about the error bound, which is tiny compared to 1 (but if need be for more general applications, a tight error analysis could be done, for instance based on [5], [6]). In the case of Emacs (without the fix), which just uses the ceiling function, only positive errors will actually give an unexpected value. As an example, this is the case of $((1/s) \cdot b) \cdot s$ with $(s = 13, b = 11)$, on which the bug was found. Even though changing the expression $((1/s) \cdot b) \cdot s$ to $(b/s) \cdot s$ will not

introduce an error, this may change the sign of the error, so that this may introduce an unexpected value when only one of the ceiling and floor function is used. For Emacs, this may be an issue when the error is changed from negative to positive, and the first case where this occurs is $(s = 35, b = 29)$. However, in practice, no bitmap fonts of size $s > 20$ could be found. So, in short, changing the expression fixes the issue with bitmap fonts of size 13 on the Cairo side, makes results more accurate in general, and does not seem to have any drawback in practice, as we could check by testing Emacs 27 (without the fix on its side).

IV. Conclusion

We have identified two issues in the GNU Emacs 27 and Cairo 1.16.0 software about their use of floating-point arithmetic. Though they occur in the same computation to get font metrics, these issues are unrelated; but their combination made a bug appear: a slightly incorrect cell height, which could be regarded only as a cosmetic problem by itself, but with annoying consequences, such as more or less preventing backward scrolling.

- The first issue, in Cairo, is an additional inaccuracy introduced only to prevent a division by zero, which should probably have never occurred, that is, such a division by zero seemed to have been the consequence of another bug. This issue could easily be solved, probably without any drawback due to the second issue: in particular, we proved that in the context of bitmap fonts at a fixed size, the new code cannot introduce an error that was not present; it can change the sign of the error, which may potentially break code, but the possibility that this may occur in practice seems improbable (and let us note that the code without this fix is much more problematic anyway). This fix, provided by commit `6e5e4bd978b7`[6], was merged in the Cairo repository[7] after version 1.18.0; at the time of writing, no new Cairo versions with this fix have been released yet.
- The second issue is that an error can be introduced in some computation while it may seem obvious that the result should always be exact. This happens because some particular case is handled by generic code. The fact that such an unexpected result is provided by a library and that this is not documented (though people do not always fully read the documentation) makes the problem worse: the caller could not know that a workaround was needed. Such a fix has been applied for GNU Emacs 28.1 at one place, but we do not know whether other parts of Emacs or other applications are affected by the inaccuracies in Cairo (with or without the above fix).

We should note that more generally, it is not always needed to implement algorithms that work for all existing values of the inputs, but that are slower and potentially imply less maintainable code, if the code can actually be executed only on a subset of these values in practice. Here, the fix in Cairo is sufficient for the 11 cases found on the system, as shown by Table I, though we do not know whether other cases are possible. Implementations should document the expected range of their inputs. But this does not seem to be done here in the analyzed code of GNU Emacs and Cairo.

Let us add that even though some inaccuracies in numerical code could be avoided in theory, the fact that software often split tasks (computations) into separate functions regarded as black boxes, which is generally regarded as good programming practice, can make inaccuracies unavoidable (even more if functions are part of different software components, such as different libraries). This can be acceptable if applications can handle such inaccuracies. For instance, in the present case of the Cairo graphics library, inaccuracies cannot be visible on screen, assuming that anti-aliasing is used. In such a case, there remain issues when discontinuous mathematical functions come into play, such as the floor and ceiling functions, conversions to integer (usually with truncation), and comparisons. But the API specifications need to be clear on whether unexpected inaccuracies may occur; the above bug in GNU Emacs gives a good example of this problem.

In particular, there may still be cases in GNU Emacs where conversions to integer are not done correctly (with visible or invisible consequences), and a further analysis should be done. It would also be interesting to have some tools to detect values that are very close to integers at run time in applications and libraries, in particular when a discontinuous mathematical function is applied on them. This could be done just with instrumentation provided by the compiler, or even by replacing library functions such as `floor` and `ceil` without recompiling the code (assuming that they were not replaced by the compiler via optimizations). Some generic floating-point analysis tools could also be considered, but they do not seem to be usable on already existing code without a lot of work, in particular if the code is split into different components like here.

[6] https://gitlab.freedesktop.org/cairo/cairo/-/commit/6e5e4bd978b7

[7] https://gitlab.freedesktop.org/cairo/cairo/-/merge_requests/533

References

[1] IEEE, *IEEE Standard for Binary Floating-Point Arithmetic, ANSI/IEEE Standard 754-1985*. New York: Institute of Electrical and Electronics Engineers, 1985. [Online]. Available: https://doi.org/10.1109/IEEESTD.1985.82928

[2] ——, "IEEE standard for floating-point arithmetic," *IEEE Std 754-2019*, pp. 1–84, 2019. [Online]. Available: https://doi.org/10.1109/IEEESTD.2019.8766229

[3] C.-P. Jeannerod, N. Louvet, J.-M. Muller, and A. Panhaleux, "Midpoints and exact points of some algebraic functions in floating-point arithmetic," *IEEE Transactions on Computers*, vol. 60, no. 2, pp. 228–241, Feb. 2011. [Online]. Available: https://doi.org/10.1109/TC.2010.144

[4] J.-M. Muller, N. Brunie, F. de Dinechin, C.-P. Jeannerod, M. Joldes, V. Lefèvre, G. Melquiond, N. Revol, and S. Torres, *Handbook of Floating-Point Arithmetic*, 2nd ed. Birkhäuser, Cham, 2018. [Online]. Available: https://www.springer.com/book/9783319765259

[5] C.-P. Jeannerod and S. M. Rump, "On relative errors of floating-point operations: Optimal bounds and applications," *Mathematics of Computation*, vol. 87, pp. 803–819, 2018. [Online]. Available: https://doi.org/10.1090/mcom/3234

[6] S. M. Rump, "Error bounds for computer arithmetics," in *Proceedings of the IEEE 26th Symposium on Computer Arithmetic*. Kyoto, Japan: IEEE, Jun. 2019. [Online]. Available: https://doi.org/10.1109/ARITH.2019.00011