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

Vincent LEFÈVRE

AriC, Inria Lyon / LIP, ENS-Lyon

31st IEEE International Symposium on Computer Arithmetic (ARITH 2024)

2024-06

The History

- GNU Emacs text editor under Linux: emacs-gtk Debian/unstable package.
- With its own GUI, built with GTK+ (default).
- After a system upgrade in October 2020: Emacs 26 → Emacs 27.

I quickly noticed a new bug: something wrong with the backward scrolling with the mouse wheel, more or less slow and not smooth.

- Normally: A scrolling step moves the text 5 lines downward (default).
- 40 scrolling steps: 24 lines downward instead of 200.

The cause...

- Normally: the cursor should remain at the bottom.
- New behavior: unexpected cursor repositioning (*recentering*).
  Consequence: The text is moved in the opposite direction.

→ Bug reports (to the Debian BTS, and several days later, upstream).

Further Tests...

I could see that the issue was occurring only with

- bitmap fonts of size 13 (scalable fonts unaffected);
- Emacs built with the Cairo graphics library.

(Here, only Unicode fonts are taken into account.)

It is not always possible to put the cursor on the last text line of the window.
→ An attempt causes the recentering.

Considering that

- the issue occurs only with particular values (here, 1 among 7 tested);
- Cairo uses floating-point arithmetic (for transformations: scale, rotate...);

at this point, I wondered: "is there some rounding involved?"
(`https://debbugs.gnu.org/cgi/bugreport.cgi?bug=44284#29`)

→ Find where unexpected data were introduced, by inspecting the source code.

A Look at the Source Code

A difficulty: the support of various backends. And the list of available backends depends on how the software has been built.

GNU Emacs:

- Without Cairo (on the X Window System):
  - `x` (the X core font driver);
  - `xft` (the Xft font driver);
  - `xfthb` (the Xft font driver with HarfBuzz text shaping);
- **With Cairo (on X):**
  - `x` (the X core font driver);
  - `ftcr` (the FreeType font driver on Cairo);
  - **`ftcrhb`** (the FreeType font driver on Cairo with HarfBuzz text shaping) if built with HarfBuzz (default);
- On MS-Windows and on Haiku: also various backends.

Cairo: several font backends (called *font types*), e.g. **FreeType**.

FP computations can spread throughout these components.

The Emacs Side

To trigger the bug: move the cursor to the last fully visible screen line.
Evaluate the LISP expression: `(move-to-window-line -1)`
→ Recentering with a bitmap font of size 13; no issues with other font sizes.

I did not know

- the expected values in the computations
  → compare what is obtained with sizes 13 and 14;
- the cause of the incorrect behavior
  → backtracking, i.e. start at some location in the source and go backward.

A guess: `xdisp.c` (about the window display), function `redisplay_window`, call to `try_cursor_movement` (which handles the change of the cursor position).

The Emacs Side [2]

```
/* Handle case where text has not changed, only point, and it has
   not moved off the frame, and we are not retrying after hscroll.
   (current_matrix_up_to_date_p is true when retrying.)  */
if (current_matrix_up_to_date_p
    && (rc = try_cursor_movement (window, startp, &temp_scroll_step),
        rc != CURSOR_MOVEMENT_CANNOT_BE_USED))
```

A difference for the return value `rc`:

- With font size 14: 0;
- With font size 13: 2 (`CURSOR_MOVEMENT_MUST_SCROLL`).

In `try_cursor_movement`: differences on code related to pixel lines.
In short: the last (fully visible) text line is actually not fully visible.
On screen (using a magnifier): a pixel line is missing, but this is hardly noticeable.

More precisely, after comparisons of size 13 with size 14, for size 13:

- All the text lines are actually shifted one pixel downward.
- The cause: The height of the first text line is increased from 13 to 14.

The Emacs Side [3]

This is due to the computed font metrics.

- With font size 14: `row->phys_ascent = 12 ; row->ascent = 12`
- With font size 13: `row->phys_ascent = 12 ; row->ascent = 11`

  expected values: `row->phys_ascent = 11 ; row->ascent = 11`
  (obtained when rebuilding GNU Emacs without Cairo).

Source of the incorrect value: `ftcrfont.c` (FreeType font driver on Cairo):

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

where

- `extents.y_bearing` is of type `double` (binary64 FP format);
- equal to `-0x1.6000000000001p+3` (binary64 number just below -11) while it should have been -11 exactly.

Among the other available bitmap fonts, `extents.y_bearing` is either an integer or just above the expected integer, so that `ceil` returns the expected value.

The Emacs Side [4]

- Undocumented inaccuracy in Cairo.
- Workaround: consider that a value that is close enough to an integer should be regarded as being this integer.

Fix applied for GNU Emacs 28.1 (commit `33e2418a7cfd`): change of

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

to

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

- I checked that this made the issue disappear.
- False positives for other kinds of fonts? $\rightarrow$ Probably not noticeable.

All the details in my bug report:

`https://debbugs.gnu.org/cgi/bugreport.cgi?bug=44284`

The Cairo Side

The question: Where does the inaccurate value come from?

→ Result `y` of 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;
```

Here,

```
new_x = font_size * x + 0 * y;
new_y = 0 * x + font_size * y;
x = new_x + 0;
y = new_y + 0;
```

In short: multiplication of the `y` input by the font size.

Note: Even though this transformation is not correctly rounded in general, here, it reduces to just a multiplication, thus correctly rounded.

The Cairo Side [2]

The `y` input comes from

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

where `hm` is 0. Thus this is just `y = fs_metrics->y_bearing`.

For size 13, this value is `-0x1.b13b13b13b13cp-1` $\approx -11/13$
(not the binary64 value nearest $-11/13$). Computed with

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

where

- `metrics->horiBearingY` is an integer from the FreeType 2 library.
- the `DOUBLE_FROM_26_6` macro interprets this integer as a 26.6 fixed-point number: FP division by $64 = 2^6$ $\rightarrow$ binary64 value (exact operation).
  For the considered fonts of size 13, one gets 11.
- `y_factor` is here computed by `1 / unscaled->y_scale`, where `y_scale` is a binary64 value = 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).

The Cairo Side [3]

In short, for the considered fonts of size 13:

$$\mathrm{RN}(\mathrm{RN}(-11 \cdot \mathrm{RN}(1/13)) \cdot 13),$$

where the rounding function RN is the default rounding mode, roundTiesToEven (rounding to nearest, ties to even).

Why a multiplication by `1 / unscaled->y_scale` instead of just a division by `unscaled->y_scale` (which would be more accurate; see next slides)?

- For an (unnoticeable) optimization, as multiplication is faster than division (`y_factor` is used up to 3 times in a same branch)? $\rightarrow$ No.
- Code introduced in 2005 in order to avoid a potential division by zero, but apparently caused by a cache issue (incorrect data in memory?), and this change did not fix anything.
  Unfortunately, this new, less accurate code remained.

Fixed in the master branch on 2024-03-21 with my merge request:

`https://gitlab.freedesktop.org/cairo/cairo/-/merge_requests/533`

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

- New 2005 code: $\mathrm{RN}(\mathrm{RN}(\mathrm{RN}(1/s) \cdot b) \cdot s)$.
- Proposed code: $\mathrm{RN}(\mathrm{RN}(b/s) \cdot s)$, i.e. like the pre-2005 code.

Inputs s (for *size*) and b (for *bearing*, in absolute value) are positive integers such that $b \leqslant s$, both rather small, thus representable in binary64.

The possible pairs (s, b) are limited in practice:

- bitmap fonts normally do not have large sizes s;
- not all sizes up to the maximum font size are used;
- the value of b is not far from the value of s (e.g., $b/s \geqslant 7/9$).

The FP expressions $((1/s) \cdot b) \cdot s$ and $(b/s) \cdot s$ [2]

Moreover, a property: 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. one cannot introduce an error with the proposed code. About the proof:

- Analysis based on the errors (for each expression) made before the last operation, i.e. the multiplication by s.
- Needs radix 2: the quotient of FP numbers cannot be the midpoint between two consecutive normal FP numbers.

Only the sign of the error (zero, positive or negative) for each expression matters.

- An error cannot appear.
- An error can disappear (this will be denoted $b+$ and $b-$).
- An error can remain, and its sign may change ($b+/+$, $b+/-$, $b-/-$, $b-/+$).

The FP expressions $((1/s) \cdot b) \cdot s$ and $(b/s) \cdot s$ [3]

Sign of the error for each value of s from 3 to 40, and b from 1 to s:

```
 5: 3+
 6:
 7: 5-
 8:
 9: 7-
10: 3+ 6+ 7+
12: 7-
13: 7+ 11+
14: 5- 10- 13-
15:
17: 3- 6- 12-
18: 7- 11- 14- 15-
19: 13- 17-
20: 3+ 6+ 7+ 12+ 14+
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+
```

```
Values of (s,b) found on the system:
b =  5  (zero)
b =  6  (zero)
b =  7  (zero)
b =  7  (negative error / zero)
b =  8  (zero)
b = 10  (zero)
b = 11  (positive error / zero)
b = 12  (zero)
b = 12  (zero)

b = 14  (negative error / zero)

b = 16  (zero)
```

```
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+
```

Conclusion

Bug caused by the combination of 2 FP, unrelated issues:

- one in the Cairo library (at least up to 1.18.0) $\rightarrow$ accuracy improved (so applications should still take inaccuracies into account);
- one in GNU Emacs 27 $\rightarrow$ now fixed.

Notes:

- For graphics libraries like Cairo, inaccuracies on binary64 values cannot normally be visible on screen.
- But potential issues with discontinuous mathematical functions (such as the floor and ceiling functions, conversions to integer...).
- The API specifications need to be clear on the expected accuracy.

Conclusion [2]

Various remarks:

- FP arithmetic can be used in unexpected places.
- The consequences of rounding errors can be surprising.
- Open-source / free software is great to be able to find the cause of bugs that the developers cannot reproduce (or do not have the time to work on).
- Difficulties for the analysis due to the structure of the source code, while regarded as good programming practice.
  Could affect tools designed to detect bugs or make the code more robust.
- The lack of specification of the interface, such as the undocumented accuracy, might have been partly a cause of the bug.
  Not the first time (e.g., Maple vs old intpak package).
- Need for easy-to-use tools to detect FP-related bugs in existing software for end users (different from testing numerical algorithms).
- In GNU Emacs, still suspicious values (very close to integers), though there are no known issues. $\rightarrow$ To be checked.

Other examples of bugs with some similarities

- Still about fonts, decisions related to rounding can be complex.
  A bug I had reported against FreeType in 2017 (never fixed): a change of the rounding direction (done on purpose) caused a blank line between characters.
  `https://bugs.debian.org/cgi-bin/bugreport.cgi?bug=866960`
  `https://savannah.nongnu.org/bugs/index.php?52165`
- Neopolis game (2 bugs): some specific values displayed incorrectly.
  The source was not available, but I could guess the FP algorithms.
  The affected values are related to the FP representation.
- The Minecraft boat crash/break (MC-119369), bug open since 2017, found by various players.
  Also related to FP values that are not exactly integers due to rounding.
  Also occurs on specific values, but derived from math reasoning (i.e. the values do not depend on the precision of the FP format).
  Matt Parker's video "The Minecraft boat-drop mystery":
  `https://www.youtube.com/watch?v=ei58gGM9Z8k`