3-CLP Numerical Guarantees

by Steffen Schuldenzucker
August 23, 2022

We show that the calculations in the 3-CLP can't overflow and have low error given certain bounds on the balances and parameters. Thus, the numerical operations don't need to do overflow checks and we can use the `*U()` functions. We also show that the `divDownLarge()` functions only have small error. Most of the work is spent on the Newton iteration.

Contents:

1 Definitions

By a *scaled* value, we mean a raw uint256 in memory. By an *unscaled* value, we mean the value that this variable represents. Scaling is by 1e18, so we write `[x] := scale(x) := x * 1e18`. Unless explicitly indicated, we refer to *unscaled* values in this document.

Note that we only use unsigned 18-decimal `uint256` value in the 3-CLP code. So an unscaled value `x` is representable iff `[x] ≤ 2^256-1 ≤ 1.15e77` or equivalently `x ≤ 1.15e59`.

2 Overflow Bounds for Operations

The `add` and `sub` functions have the obvious over/underflow bounds: the result needs to be in the interval `[0, 1.15e59]`. This condition is usually rather weak and implied by any other bounds we have on these values (which make sure that, e.g., multiplication and division work). This is why we will not usually discuss addition and subtraction explicitly below. For subtraction, we need to ensure, of course, that it does not underflow at 0. See the comments in the code for this.

The multiplication and division functions have different bounds (this is easy to see; see also the documentation of these functions):

- `mulDown(a, b)`: `a * b ≤ 1.15e41`. (the actual bound is `(2^256-1)/1e18^2) ≥ 1.15e41`)
- `divDown(a, b)`: `a ≤ 1.15e41`. (the actual bound is the same as above)
- `mulDownLargeSmall(a, b)`: `a * b ≤ 1.15e59` and `b ≤ 1.15e41`.
- `divDownLarge(a, b, d, e)`: `a * [d] ≤ 1.15e59` and `b ≥ [e] * 1e-18`.
  - It is easier to think about the parameters d and e in their scaled form.
  - By default, `[d] = 1e9` so that we need `a ≤ 1.15e50`.

Clearly, the `*Large*` functions allow bigger values than their regular counterparts.

Note that we always have `divDown(a, b) ≤ a * 1e18`. This is because the smallest representable number is `b = 1e-18`. Checking for overflows in `divDown()` therefore also implies a (rather weak) bound on the result.

Also note that the multiplication and division functions interact as follows: at the end of computing a product, we divide by 1e18 whereas the beginning of a division inflates the dividend by 1e18. Therefore, when a product can be safely computed via `mulDown()`, we can also divide this product again via `divDown()` by some other value without worrying about overflow (as long as we add a small margin for numerical errors).

3 Error Bound for `divDownLarge()`

The "large" variant of multiplication, `mulDownLargeSmall()` has error on the order of 1e-18 like the other operations (but it uses twice the space and the number of operations). In contrast, `divDownLarge()` only has this order of error within some range of inputs.

Specifically, `divDownLarge()` is defined as follows:

```
[divDownLarge(a, b, d, e)] = Math.divDown([a] * [d], Math.divUp([b], [e]))
```

where `*` is regular integer multiplication and `Math.div{Up,Down}` are rounding-down and rounding-up version of integer division. Recall that we require `[d] * [e] = 1e18`, i.e., `d * e = 1e-18`. `[d] = 1e18` and `[e]=1` correspond to the regular `divDown` operation.

Let ε := 1e-18. The truncation in `Math.divUp([b], [e])` introduces an error of at most scaled `+[e]`, i.e., unscaled `+[e] * ε`, into what would be the real quotient `[b]/[e]`. The integer multiplication in the numerator does not introduce any error.

We now study the error in `divDownLarge()`. Write short $\oslash\downarrow$ for `Math.divDown` (integer division truncating at 1, rounding down) and $\oslash\uparrow$ for `Math.divUp` and note that multiplication is exact as long as it doesn't overflow. Write `/` for regular division without rounding, yielding a real number. Write "$\{+\varepsilon\}$" and "$\{-\varepsilon\}$" as one-sided versions of "±ε". We have

$$\begin{aligned}
[\text{divDownLarge}(a,b,d,e)] &= ([a]*[d]) \oslash\downarrow ([b] \oslash\uparrow [e]) \\
&= ([a]*[d]) \oslash\downarrow ([b]/[e]\ \{+1\}) \\
&= \frac{[a]*[d]}{[b]/[e]\ \{+1\}}\ \{-1\} \\
&= \frac{[a]*[1]}{[b]\ \{+[e]\}}\ \{-1\}
\end{aligned}$$

where in the last line, we expanded the fraction by $[e]$ and exploited the fact that we require $[e]*[f] = 1\text{e}18 = [1]$. By dividing by $[1]$, we transition to the unscaled version and receive

$$\begin{aligned}
\text{divDownLarge}(a,b,d,e) &= \frac{[a]}{[b]\ \{+[e]\}}\{-\varepsilon\} \\
&= \frac{a}{b\ \{+e\}}\{-\varepsilon\}
\end{aligned}$$

where the second line is by canceling a factor `e-18` on both sides of the fraction. We ignore the (additive) error $\{-\varepsilon\}$ (which is on the order of the other numerical operations) for now and consider the error introduced by the $\{+e\}$ in the numerator. Obviously, this error is larger when e is large, i.e., when d is small. Note that there is thus a trade-off between the maximum possible size of a and the accuracy of the operation.

Note: Formally, our error estimation is slightly loose because the error of the integer division operations "$\oslash\downarrow$" and "$\oslash\uparrow$" is not always as big as 1. For example, if $[e] = 1$, the error is 0 because integer division by 1 is a no-op, and if $[e] = 5$, the error is at most 0.5, not 1, because the dividend of "$\oslash\uparrow$" is integer. We ignore this effect because we receive an upper bound on the error in any case and also the effect is insignificant for larger values of e, which are the relevant values here.

Note that `divDownLarge` only has a downwards and no upwards error and the maximum absolute error due to the e error is

$$\frac{a}{b} - \frac{a}{b+e} = \frac{a}{b} \cdot \frac{e}{b+e} \leq \frac{a}{b^2} \cdot e$$

Here we have bounded $b + e \geq b$, which will be a very tight bound in all situations we consider.

The value that results from the `divDownLarge()` function is equal to the real value computed by the function above, truncated to 18 decimals. Thus, in principle, if the error above is less then 1e-18, it should disappear in the final result. However, in practice, we need to keep in mind that all operations are performed in binary, not decimal arithmetics. Therefore, the real threshold where the error vanishes is slightly smaller than 1e-18.

4 Assumptions

We require that the following hold at all times (unscaled values). This is ensured in the code prior to any operation that uses these values. Here l refers to any (starting or intermediate or final) value of the invariant that occurs over the course of the algorithm:

- $\alpha \leq 0.9999$
  - This implies $a = 1 - \alpha \geq 1e-4$
- $\alpha \geq 0.004$
- $x, y, z \leq 1e11$
- $l \leq 4.86e16$
  - This bound is not in principle needed because all L values that actually occur are bounded by 4.01e15 as a consequence of other bounds (see below). It's purely a safety measure.
  - To be safe, the code uses the bound 1e16.
- $l / L+ \geq 1.3$
  - This is also not in principle needed because we should always have $l / L+ \geq 1.5$ (see below). So this is also purely a safety measure.
  - This condition will ensure that `dfRootEst` is not too small, which could make l too large and (more importantly, b/c l itself is checked anyways) also make `divDownLarge` imprecise when l is very large.

In addition, we change how some operations are done when `l > 2e13`. This is slightly below the threshold where the computation of `l^3` as `l.mulDown(l).mulDown(l)` does not overflow (which is about `4.87e13`). Note that we could've chosen `4.87e13` as this threshold but we leave some safety margin. The threshold cannot be too small because, for smaller values of l, the `divDownLarge()` operation may become imprecise.

4.1 Derived Bounds

The following bounds easily follow from our conditions:

- `mb = (x + y + z) * α^2/3 ≤ 3e11`
- `mc = (xy + yz + zx) * α^1/3 ≤ 3e22`

- `md = xyz ≤ 1e33`
- `L+ ≤ 2e15`
  - This is a bound on the numerically computed value for x, y, z, α at their respective allowed maximum. L+ is monotonic in these values, which can be seen directly from the computation formula.
- `L_0 ≤ 2 * L+ ≤ 4e15`
- `L_root ≤ 3e15` where `L_root` is the root of the polynomial that we are computing and the invariant we want to calculate.
  - This is a bound on the numerically computed value for x, y, z, α at their respective allowed maximum.
  - `L_root` is monotonic in these values. To see this, note that for any fixed L, f(L) is monotonically decreasing in each of these values. Thus, if f(L) < 0 then still f(L) < 0 if we increase these values. Thus, the root of f must be moving to the right.

Note that these results imply no statement about `L_root / L_0` yet. This is what we will look at next.

We have seen computationally (see `cpmmv3-cubic-experiments.nb` Mathematica notebook) regarding `ζ := L_root / L+`:

- ζ is monotonically decreasing in α.
- ζ ≥ 1.5 at all times and $\lim_{\alpha \to 1} \zeta = 1.5$ for all values of x, y, z.
- ζ is invariant under scaling of the vector (x, y, z) and for fixed α, ζ is maximized when x=y=z.
- The value of ζ for α=0.004 (our lower bound) and x=y=z, i.e., the maximum value of ζ we will ever see, is 2.79... ≤ 2.8.
- ⇒ The values of ζ we will see are contained in the interval `[1.5, 2.8]`.
  This also implies that the values of `l/L+` we will see throughout the iteration are ≥ 1.5.

Note that the first Newton step may move *upwards* and all following Newton steps move downwards (because of convexity). For the first Newton step, $-f(L_0)/f'(L_0)$, we have seen computationally (for $L_0 = \gamma_0 L_+$ where $\gamma_0 \in [1.5, 2]$, i.e., the values we use):

- For fixed γ_0, this first Newton step is monotonically decreasing in α and monotonically increasing in each of x, y, z.
- For α=0.004 (= the minimum value allowed) and x=y=z=1e11 (= the maximum value allowed), its value is ≤ 1.68e11.
- ⇒ The values of l we will see are ≤ `L_0 + 1.68e11 ≤ 4e15 + 1.68e11 ≤ 4.01e15`.

5 Operations outside `calcNewtonStep()`

It is easy to see that all operations that lie outside `calcNewtonStep()` do not overflow given our assumptions.

6 `calcNewtonStep()`

We walk through the function, line by line, to show that none of the operations overflow and the error due to `divDownLarge()` is vanishingly small. Line numbers refer to calculation lines (excluding comments, empty lines, and control flow) of a given function or block in commit `e8cb79fe5add` (23 Aug 2022).

We write short `l := rootEst`.

6.1 L1: `dfRootEst = rootEst.mulDown(rootEst)`

Since we assume `l ≤ 4.86e16`, `l*l ≤ 2.37e33 < 1.15e41`, so this operation is safe with a large margin. It would still be safe if we only assumed `l ≤ 3.83e18`.

6.2 L2: Multiplication by 3

This is obviously safe because we have a large margin above.

6.3 L3

We calculate `dfRootEst * root3Alpha * root3Alpha * root3Alpha` using mulDown. Since `root3Alpha ≤ 1` and `dfRootEst = 3 l^2` (see above), `dfRootEst * root3Alpha ≤ 1.15e41`, so this is safe.

The subtraction is safe because we assume `l ≥ lmin`, so `f'(l)` (which is being calculated) will be positive. Note that we consider a margin and order operations such that errors are not being amplified, so numerical errors will not violate this condition.

6.4 L4

`rootEst.mulDown(mb)`

We have and `mb ≤ 3e11` (see Derived Bounds above) so that `rootEst * mb ≤ 4.86e16 * 3e11 = 1.45e28 ≤ 1.15e41` with a large margin.

Other operations

Multiplication by 2 won't overflow because we have a margin above. The sub operations won't underflow because we are in a region where `f'(l) > 0`.

We now distinguish two cases based on where l lies compared to `_L_THRESHOLD_SIMPLE_NUMERICS = 2e13`.

6.5 Case of small l

For the lines in the if branch we have `l ≤ 2e13`.

L1

We have `l^2 * l = l^3 ≤ 8e39 ≤ 1.15e41`, so that this operation is safe.

L2

To see that the `mulDown` chain is safe, we again note that `root3Alpha ≤ 1` and so it is sufficient that `l^3 * 1 ≤ (2^256-1) / 1e18^2`, which we have seen in the previous line.

The subtractions won't underflow because we subtract a smaller value, because `root3Alpha ≤ 1`.

L3

When we enter the line, `deltaMinus ≤ l^3` and, again by the above `l^3 ≤ (2^256-1) / 1e18^2`. Note that there is some slack with the bound we're actually using for l.

L4

`l^2 * mb ≤ 2e13^2 * 3e11 = 1.2e38 ≤ 1.15e41`

L5

For the multiplication we have
`l * mc ≤ 2e13 * 3e22 = 6e36 ≤ 1.15e41`
where the bound on `mc` comes from the derived bounds above.

Since at this point (see 4), `deltaPlus ≤ 1.2e38`, addition does not overflow (by a margin) and
`deltaPlus.add(rootEst.mulDown(mc)) ≤ 1.31e38`. Therefore, the `divDown` call is safe.

L6

`md ≤ 1e33 ≤ 1.15e41`, so the `divDown` call is safe. The bound on `md` is from the derived bounds above. Again, as we have large margins for our overflows, addition is safe.

6.6 Case of large l

We now consider the case `2e13 < l ≤ 4.86e16`. (actually, we even have `l ≤ 5e15` or so)

L1

We have `l^3 ≤ 4.86e16^3 ≤ 1.148e50` and `l ≤ 4.86e16 ≤ 1.15e41` by a large margin.

L2

By the same argument and `root3Alpha ≤ 1`, the `mulDownLargeSmall` operations are safe.

L3

In the `divDownLarge()` call, we use `[d] = [e] = 1e9` and thus need `deltaMinus ≤ 1.15e50`. We have `deltaMinus = a l^3 ≤ l^3 ≤ 1.148e50` (see L1 above) and `dfRootEst ≥ 1` (this follows from the discussion below), so this is safe.
(Note: with the actual `~5e15` bound on l, we even have `l^3 ≤ 1.25e47`, i.e., there's some slack when we make the `_L_MAX` bound a bit tighter.)

We now bound the error due to the $\{-e\}$ error in the denominator in `divDownLarge`. The absolute error due to this is

$$\frac{al^3}{f'(l)^2}e$$

where e=1e-9. To bound this error, recall that the roots of f' are L_+ and L_- and thus we have (the constant factor being the coefficient of l^2)

$$f'(l) = 3a(l - L_+)(l - L_-).$$

from the formulas, it is clear that $L_- \leq 0$ (with 0 being a degenerate special case when the pool has only one positive asset) and therefore we can bound

$$\frac{al^3}{f'(l)^2}e = \frac{a}{(3a)^2}\frac{l^3}{(l-L_+)^2(l-L_-)^2}e \leq \frac{1}{9a}\frac{l}{(l-L_+)^2}e.$$

Let now γ be such that $l = \gamma L_+$. We know that, during the Newton iteration, we always have $\gamma \geq 1.5$ (see Derived Conditions above). We use $\gamma \geq 1.3$ (see bounds above) as a slightly weaker bound. Using $L_+ = l/\gamma$, the error is now

$$\begin{aligned}\frac{1}{9a}\frac{l}{(l-l/\gamma)^2}e &= \frac{1}{9a}\frac{1}{(1-1/\gamma)^2}\frac{1}{l}e \\ &\leq \frac{1}{(1-1/\gamma)^2}\cdot 5.55\text{e}{-11}\cdot e\end{aligned}$$

Since e=1e-9 and for γ ≥ 1.3, this error is bounded by 1.04e-18.

L4

We have `l^2 * mb ≤ 4.86e16^2 * 3e11 = 7.08e44` and (clearly) `mb ≤ 3e11 ≤ 1.15e41`, so the `mulDownSmallLarge()` is safe by a wide margin. Note that `mulDown()` is not necessarily safe.

L5

The `mulDown()` is safe because `mc * l ≤ 3e22 * 4.86e16 = 1.458e39 ≤ 1.15e41`.

L6

To see that the `divDownLarge()` is safe, use the bounds from the `mulDown()` in L5 and from L4 to see that the dividend is `l^2 * mb + mc * l ≤ 7.08e44 + 1.458e39 ≤ 7.09e44`. Thus, `divDownLarge()` is safe for `[d] ≤ 1.15e59 / 7.09e44 ≤ 1.63e14`. We use `[d] = 1e12`.

Towards the rounding error in the operation, we need to go into greater detail again. The rounding error is equal to (letting $\bar{b} := \mathrm{mb} = -b$ and likewise for c)

$$\begin{aligned}
\frac{l^2 * \bar{b} + l * \bar{c}}{f'(l)^2} e &= \frac{1}{9a^2} \frac{l^2 * \bar{b} + l * \bar{c}}{(l - L_+)^2 (l - L_-)^2} e \\
&\le \frac{1}{9a^2} \left(\frac{\bar{b}}{(l - L_+)^2} + \frac{\bar{c}}{(l - L_+)^2 l} \right) e \\
&= \frac{1}{9a^2} \frac{1}{(1 - 1/\gamma)^2} \left(\frac{\bar{b}}{l^2} + \frac{\bar{c}}{l^3} \right) e
\end{aligned}$$

We can bound

$$\begin{aligned}
\bar{b}/l^2 &\le 3\mathrm{e}11/(2\mathrm{e}13)^2 \le 7.5\mathrm{e}{-16} \\
\bar{c}/l^3 &\le 3\mathrm{e}22/(2\mathrm{e}13)^3 \le 3.75\mathrm{e}{-18} \\
1/a^2 &\le 1\mathrm{e}8
\end{aligned}$$

Therefore, the error is bounded by

$$\frac{1}{(1 - 1/\gamma)^2} \cdot 8.38\mathrm{e}{-9} \cdot e$$

Since we use `[e]=1e6` (so `e=1e-12`) and for $\gamma \ge 1.3$, this is bounded by `1.57e-19`.

Note: We can achieve a slightly better bound for the $\bar{b}$ term using a bit more work as follows:

- Use γ to express everything in terms of L+ instead of L.

- Replace the definition of L+ and note that the term $9a^2$ cancels out and the result is monotonically decreasing in $\bar{b}$.
- Now bound $\bar{b}$ from below by, again, considering the definition of L+ and the lower bound on L+. See Steffen's notebook p. 202.
  The bound we get is ca. one order of magnitude better than the one above because we're able to cancel out the $9a^2$ term.

L7

The `divDown` is safe because `md ≤ 1e33`.