E-CLP High Precision Calculations

Ariah Klages Mundt Steffen Schuldenzucker

Nov 2022

Suppose $\bar{c}, \bar{s}$ be of magnitude < 1 with up to 18 digits of non-zero decimals. Such $\bar{c}, \bar{s}$ can be chosen such that

$$s := \sin(\varphi) = \bar{s}/d$$
$$c := \cos(\varphi) = \bar{c}/d$$

where $\bar{c}^2 + \bar{s}^2 = d^2$. In particular, we will choose $\bar{s}, \bar{c}$ to be the sine and cosine of φ truncated at the 18th decimal. This will allow us to work with $\bar{s}, \bar{c}$ in normal 18 decimals but correct for the error (by dividing by factors of d) at the end to regain precision.

Next, notice that the calculation of $\tau(\beta)$ (and similarly $\tau(\alpha)$) equates to

$$\tau(\beta) = \frac{1}{\sqrt{(c+\beta s)^2/\lambda^2 + (\beta c - s)^2}} \begin{bmatrix} c\beta - s \\ (c + s\beta)/\lambda \end{bmatrix} = d_\beta \begin{bmatrix} c\beta - s \\ (c + s\beta)/\lambda \end{bmatrix}$$

and similarly for $\tau(\alpha)$, defining the normalization factors as d_β, d_α respectively. Notice that the y-component of τ is always > 0. Notice also that $\tau(\beta)_x > \tau(\alpha)_x$ since $\beta > \alpha$. These will be useful facts in determining rounding directions.

Similar to s, c, define $\bar{\tau}$ (similarly both for $\tau(\alpha)$ and $\tau(\beta)$) to be

$$\bar{\tau}(\beta) = d_\beta \begin{bmatrix} \bar{c}\beta - \bar{s} \\ (\bar{c} + \bar{s}\beta)/\lambda \end{bmatrix}$$

and observe that $\tau(\beta) = \bar{\tau}(\beta)/d$. Note that the reason for defining $\bar{\tau}$ is to ensure an even number of d factors to correct for at the end of calculations so that a square root of d^2 is not required. It turns out that the extra factor of d coming from $\bar{\tau}$ in this way allows us to make this simplification. Observe in particular that d_β is defined with respect to the *actual* values s, c rather than the approximate values $\bar{s}, \bar{c}$ to receive this relationship. Note that we can compute d_β as

$$d_\beta = \frac{1}{\sqrt{(c+\beta s)^2/\lambda^2 + (\beta c - s)^2}} = \sqrt{\frac{d^2}{(\bar{c} + \beta\bar{s})^2/\lambda^2 + (\beta\bar{c} - \bar{s})^2}}$$

We will suppose that $d, \bar{\tau}(\alpha), \bar{\tau}(\beta)$ (and additionally $\bar{u}, \bar{v}, \bar{w}, \bar{z}$ defined below involving τ terms) are known to high precision (38 decimal places). In the current E-CLP iteration, these *derived parameters* are calculated off-chain and imported in the constructor as these variables are immutable in the current design. All calculations involving these terms will be done in such precision, and

operations involving s, c will be separated into operations of $\bar{c}, \bar{s}, d$, with d operations re-ordered to happen at the end alongside $\tau(\beta), \tau(\alpha)$ operations. The τ operations are similarly split into $\bar{\tau}$ and d operations, with the d operations combined at the end with other d operations. The reason is to ensure that parameter error (e.g., in the last decimal place) does not get amplified to significant decimal places in the multiplication with x^2 (which could cause the error to amplify 24 decimal places if balances are on the order of trillions). Note that these high precision calculations can be done without worry of overflow because all terms will be of magnitude < 1.

In principle, the derived parameters could be calculated on-chain, which would enable a later dynamic version of the E-CLP. The extra precision calculations would be more complex here, however, as we would need to handle realistic cases where $\beta, \alpha >> 1$, in which case the current extra precision calculations could overflow. By importing the derived parameters in the constructor, we ignore this complication for now.

1 Calculating the Invariant r

Given $t = (x, y)$, the invariant r is formulated as follows:

$$r = \frac{At \cdot A\chi + \sqrt{(At \cdot A\chi)^2 - (A\chi \cdot A\chi - 1)\, At \cdot At}}{A\chi \cdot A\chi - 1},$$

where $\chi := (e_x \cdot A^{-1}\tau(\beta),\, e_y \cdot A^{-1}\tau(\alpha))$. Note that this formula contains both (scalar) products of vectors and products of real numbers.

To achieve optimal fixed point precision, we compute these products at a lower level.

First note that

$$A\chi = A \begin{bmatrix} e_x A^{-1}\tau(\beta) \\ e_y A^{-1}\tau(\alpha) \end{bmatrix} = \begin{bmatrix} sc(\tau(\beta)_y - \tau(\alpha)_y)/\lambda + c^2\tau(\beta)_x + s^2\tau(\alpha)_x \\ \lambda sc(\tau(\beta)_x - \tau(\alpha)_x) + s^2\tau(\beta)_y + c^2\tau(\alpha)_y \end{bmatrix} = \begin{bmatrix} w/\lambda + z \\ \lambda u + v \end{bmatrix}$$

where w, z, u, v are defined in-line. Note that $|w|, |z|, |u|, |v| \leq 1$ as all individual terms are ≤ 1 in magnitude and sc is identical to a cosine function with magnitude 1/2. As noted above, the calculations of w, z, u, v will be done in very high precision so that multiplications by w, z, u, v can have limited error propagation. As with $\bar{\tau}$ above, $\bar{w}, \bar{z}, \bar{u}, \bar{v}$ will be calculated from $\bar{\tau}$ instead of τ and factors of d will be accounted for all at once at the end of implemented calculations.

To take stock, in the implemented calculations, a multiplication by $\bar{s}$, $\bar{c}$, or a component of $\bar{\tau}(\alpha)$ or $\bar{\tau}(\beta)$ will require a later division by a factor of d to correct for error. And a multiplication by $\bar{u}, \bar{v}, \bar{w}$, or $\bar{z}$ will require a later division by a factor of d^3. When the resulting factors of d are combined, they will always lead to an even power of d.

Notice that $v > 0$ since y-components of τ are > 0. We also have $u > 0$ since $\tau(\beta)_x > \tau(\alpha)_x$ (b/c the corresponding point on the circle is $-r\tau(p)$ for a given price). These will be useful in determining rounding direction.

$$At = \begin{bmatrix} cx/\lambda - sy/\lambda \\ sx + cy \end{bmatrix}$$

$$At \cdot A\chi = (cx - sy)(w/\lambda + z)/\lambda + (x\lambda s + y\lambda c)u + (sx + cy)v$$

After separating $\bar{c}, \bar{s}$, we will need to divide by d^4 as there are four factors of s, c in each term.

$$\begin{aligned} A\chi \cdot A\chi &= (w/\lambda + z)^2 + (\lambda u + v)^2 \\ &= (w/\lambda + z)^2 + \lambda^2 u^2 + \lambda 2uv + v^2 \end{aligned}$$

After separating $\bar{c}, \bar{s}$, we will need to divide by d^6 as there are six factors of s, c in each term.

The terms in the square root can be expanded, canceled, and reordered for better precision:

$$\begin{aligned} \sqrt{\dots}^2 &= \Big((At)_x(A\chi)_x + (At)_y(A\chi)_y\Big)^2 - \Big((A\chi)_x^2 + (A\chi)_y^2 - 1\Big)\Big((At)_x^2 + (At)_y^2\Big) \\ &= -(At)_x^2(A\chi)_y^2 + (At)_x^2 + 2(At)_x(At)_y(A\chi)_x(A\chi)_y - (At)_y^2(A\chi)_x^2 + (At)_y^2 \end{aligned}$$

$$\begin{aligned} (At)_x^2(A\chi)_y^2 &= (cx/\lambda - sy/\lambda)^2(\lambda u + v)^2 \\ &= (x^2c^2 - xy2sc + y^2s^2)(u^2 + 2uv/\lambda + v^2/\lambda^2) \end{aligned}$$

The last term can be expanded out so that the last operation is dividing by lambda where applicable to minimize error. After separating $\bar{c}, \bar{s}$, we will need to divide by d^8 as there are eight factors of s, c in each term.

$$(At)_x^2 = (x^2c^2 - xy2sc + y^2s^2)/\lambda^2$$

After separating $\bar{c}, \bar{s}$, we will need to divide by d^2 as there are two factors of s, c in each term.

$$\begin{aligned} 2(At)_x(At)_y(A\chi)_x(A\chi)_y &= 2(cx/\lambda - sy/\lambda)(sx + cy)(w/\lambda + z)(\lambda u + v) \\ &= \Big((x^2 - y^2)sc + yxc^2 - yxs^2\Big)2\Big(zu + (wu + zv)/\lambda + wv/\lambda^2\Big) \end{aligned}$$

After separating $\bar{c}, \bar{s}$, we will need to divide by d^8 as there are eight factors of s, c in each term.

$$\begin{aligned} (At)_y^2(A\chi)_x^2 &= (sx + cy)^2(w/\lambda + z)^2 \\ &= (x^2s^2 + xy2sc + y^2c^2)(z^2 + 2zw/\lambda + w^2/\lambda^2) \end{aligned}$$

After separating $\bar{c}, \bar{s}$, we will need to divide by d^8 as there are eight factors of s, c in each term.

$$(At)_y^2 = x^2s^2 + xy2cs + y^2c^2$$

After separating $\bar{c}, \bar{s}$, we will need to divide by d^2 as there are two factors of s, c in each term.

In procuring the right rounding direction, we will use the fact that $A\chi \cdot A\chi > 1$. To see why this is the case, assume it is negative. Then the numerator of r would have to be negative also. But the numerator is of the form $b + \sqrt{b^2 + c}$ with $c > 0$. Even with $b < 0$, $\sqrt{b^2 + c} > |b|$, and so the numerator would be positive.

Error analysis Let $\varepsilon = \pm 1e - 18$ (error in the last digit of normal precision) and $\epsilon = \pm 1e - 38$ (error in the last digit of extra precision).

The error inside the square root is $O(\varepsilon)$ if balances are $O(1e10)$ as rounding errors in extra precision are not scaled above the extra precision decimal places. For balances $> O(1e10)$, extra precision rounding errors can be magnified into normal precision decimal places, as the error scales relative to the square of balances, which leads to an error term $O((x^2 + y^2)\epsilon)$ that is $> O(\varepsilon)$ in this case. To see this, consider that all multiplications of normal precision terms lead to $O(\varepsilon)$ error as all error terms that arise are only further multiplied by small numbers. This is similarly the case for the terms calculated in extra precision (the error term remains in the last decimal place in extra precision). Lastly, when we multiply the extra precision term by the normal precision term at the end, the error in the extra precision term is scaled by $O(x^2 + y^2)$. When balances are $O(1e10)$, the error in the extra precision term is magnified at most to the 18th decimal place (i.e., is magnified 20 decimal places contained in extra precision).

Let the square root evaluation be $\sqrt{b^2 - 4ac}$. The error after the square root is taken reduces with the square root to become $\frac{O((x^2+y^2)\epsilon+\varepsilon)}{2\sqrt{b^2-4ac}} + O(\varepsilon)$ if $\sqrt{b^2 - 4ac} >> 0$ (We will use the choice of $\geq 1e - 18$ although it's possible a different cutoff is better in some settings) and at most $O\Big(\sqrt{(x^2 + y^2)\epsilon + \varepsilon}\Big)$ otherwise The first case comes from $\Big(x + \frac{\varepsilon}{2\sqrt{x}}\Big)^2 \approx x + \varepsilon$ when $\sqrt{x} >> 0$. The second case comes from $\sqrt{x + \varepsilon} \leq \sqrt{x} + \sqrt{\varepsilon}$. The second case effectively moves the error term up by half its decimal places. Whether this error propagates into normal precision depends on the size of balances and the order of magnitude of the square root evaluation (i.e., it's possible that terms within the square root cancel and the square root itself is small, which would amplify the error as handled in the second case). If balances are small and we are in the latter case (which we might expect to usually coincide), then the error does not propagate into normal precision decimals even after the square root is taken.

The other term in the numerator has error $O(\lambda(x + y)\epsilon) + O(\varepsilon)$. Whether this error propagates into normal precision depends on the size of balances and the size of λ. I.e., in some settings $(x+y)\lambda$ may be $> O(1e21)$ if balances are very large and $\lambda = O(1e8)$ or so.

The entire numerator has the following error

$$\text{Error in numerator} = O\Big(\lambda(x + y)\epsilon\Big) + \text{ error in square root } + O(\varepsilon).$$

If the denominator is small, this will additionally scale this error as well. In particular, after the above error propagation in extra precision, the error can propagate further with each decimal place that $A\chi \cdot A\chi - 1$ is below 1.

The error in the denominator (which is $O(\varepsilon)$) also causes a *relative* error in the invariant of $O(\varepsilon)$/denominator due to the division by a number that itself has some error.

2 Swap Execution

When r and one of x or y is given, we can compute the other reserve via

$$y = \frac{-sc\underline{\lambda}x' - \sqrt{s^2c^2\underline{\lambda}^2x'^2 - (1-\underline{\lambda}s^2)\left[(1-\underline{\lambda}c^2)x'^2 - r^2\right]}}{1-\underline{\lambda}s^2} + b$$
$$x = \frac{-sc\underline{\lambda}y' - \sqrt{s^2c^2\underline{\lambda}^2y'^2 - (1-\underline{\lambda}c^2)\left[(1-\underline{\lambda}s^2)y'^2 - r^2\right]}}{1-\underline{\lambda}c^2} + a,$$

where $\underline{\lambda} := 1 - 1/\lambda^2$, $x' := x - a$, and $y' := y - b$. We use this to execute swaps.

To achieve optimal precision, we reorder the calculation of these terms

$$\begin{aligned} y &= \frac{-x'\underline{\lambda}sc - \sqrt{x'x'\underline{\lambda}^2s^2c^2 + r^2(1-\underline{\lambda}s^2) - x'x'(1-\underline{\lambda}s^2)(1-\underline{\lambda}c^2)}}{1-\underline{\lambda}s^2} + b \\ &= \frac{-x'\underline{\lambda}sc - \sqrt{r^2(1-\underline{\lambda}s^2) - x'x'/\lambda^2}}{1-\underline{\lambda}s^2} + b \end{aligned}$$

The last line follows by noting that factors of $x'x'$ cancel:

$$\underline{\lambda}^2s^2c^2 - (1-\underline{\lambda}s^2)(1-\underline{\lambda}c^2) = \underline{\lambda}(c^2+s^2) - 1 = \underline{\lambda} - 1 = -1/\lambda^2$$

Considering that x' is a function of x and r and λ, its computation can now be reordered for better precision:

$$\begin{aligned} x'x'/\lambda^2 &= \Big(x - r\lambda\tau(\beta)_x c - rs\tau(\beta)_y\Big)^2/\lambda^2 \\ &= r^2\tau(\beta)_x^2c^2 + \Big(r^2 2cs\tau(\beta)_x\tau(\beta)_y - rx2c\tau(\beta)_x\Big)/\lambda + \Big(r^2s^2\tau(\beta)_y^2 - rx2s\tau(\beta)_y + x^2\Big)/\lambda^2 \end{aligned}$$

Lastly, notice that the calculation for x given y is directly parallel to that of y given x. In particular, we just need to switch $x \to y$, $s \to c$, $c \to s$, $\tau(\beta)_x \to -\tau(\alpha)_x$, $\tau(\beta)_y \to \tau(\alpha)_y$, $a \to b$, $b \to a$.

To minimize error propagation, we will perform $\underline{\lambda}$ multiplications with extra precision as we did τ factors above. We will also separate out $\bar{c}, \bar{s}, d$ terms and re-order the d operations to occur last. The d operations will involve $/d^2$ or $/d^4$ on each term depending on the number of factors of s, c (all terms have either 2 or 4 factors).

Error analysis Suppose δ is the error in r. The error inside the swap square root has a term that is $O((r+x)\delta)$. Further, when $r + x > O(1e10)$ another error term additionally scales relative to $r(r + x/\lambda) + x^2/\lambda^2$ (as this is when extra precision rounding errors get magnified to normal digits of precision). To see this, consider that multiplying r by r leads to the error in r being scaled by r, and similarly with rx. When r is $O(1e10)$, the error in the extra precision term (in the 38th decimal place) isn't magnified in following multiplications further than the 18th decimal in normal precision, but otherwise this leads to an additional error term. For r^2 or $rx > O(1e20)$, the error will propagate an additional decimal for each magnitude increase. The total error inside the square root

is $O((r+x)\delta) + O(r(r+x/\lambda)\epsilon) + O(x^2/\lambda^2\epsilon)$.

Let the square root evaluation as $\surd_{\text{swap}}$. The error after the square root is taken reduces with the square root to become $\frac{O((r+x)\delta)+O(r(r+x/\lambda)\epsilon)+O(x^2/\lambda^2\epsilon)}{2\surd_{\text{swap}}}$ if $\surd_{\text{swap}} >> 0$ and

$$\sqrt{O((r+x)\delta) + O(r(r+x/\lambda)\epsilon) + O(x^2/\lambda^2\epsilon)}$$

otherwise.

The error in the other numerator term is $O(\lambda\delta)$ as $x' = x - a$, where a contains error $O(\lambda\delta)$, and the other multiplying terms are of small magnitude and so do not magnify this. The entire numerator then has error

$$\text{error in numerator} = O(\lambda\delta) + \text{ error in square root.}$$

This error can additionally scale with the denominator if it is small (but it can't be too small). For each decimal that the denominator is < 1, the error scales an additional decimal.

Rounding in the right direction is needed to account for these.

3 Offsets and Max Balances

We can compute the shifting vector (a, b) and intersection points (max balances (x^+, y^+)) as

$$\begin{aligned}(a, -y^{+\prime}) &:= rA^{-1}\tau(\beta)\\ (-x^{+\prime}, b) &:= rA^{-1}\tau(\alpha).\end{aligned}$$

and we have

$$\begin{aligned}x^+ &= x^{+\prime} + a\\ y^+ &= y^{+\prime} + b.\end{aligned}$$

Noting that $A^{-1}\begin{bmatrix}t_x\\ t_y\end{bmatrix} = \begin{bmatrix}ct_x\lambda + st_y\\ -st_x\lambda + ct_y\end{bmatrix}$, we have

$$\begin{aligned}a &= r\lambda c\tau(\beta)_x + rs\tau(\beta)_y\\ b &= -r\lambda s\tau(\alpha)_x + rc\tau(\alpha)_y\end{aligned}$$

ordered for optimal precision. Notice that after separating $\bar{c}, \bar{s}$, we need to divide by d^2 as there are two factors of s, c in each term.

For max balances, we have

$$\begin{aligned}x^+ &= -r(\lambda c\tau(\alpha)_x) - rs\tau(\alpha)_y + a\\ &= r\lambda c\Big(\tau(\beta)_x - \tau(\alpha)_x\Big) + rs\Big(\tau(\beta)_y - \tau(\alpha)_y\Big)\end{aligned}$$

and

$$\begin{aligned} y^+ &= r\lambda s\tau(\beta)_x - rc\tau(\beta)_y + b \\ &= r\lambda s\Big(\tau(\beta)_x - \tau(\alpha)_x\Big) + rc\Big(\tau(\alpha)_y - \tau(\beta)_y\Big) \end{aligned}$$

ordered for optimal precision. Similarly, after separating $\bar{c}, \bar{s}$, we need to divide by d^2 as there are two factors of s, c in each term.

4 Multiplying from Extra Precision to Normal Precision

We implement `mul{Up,Down}XpToNp` to multiply a normal precision number a by an extra precision number b (with 38 decimals) in a way to reduce error amplification. We do this by separating b into two 19 digit numbers and separately multiply a by these numbers and combine the results while converting to 18 digit precision.

In particular, let $b_1 = b/1e19$ and $b_2 = b \,\%\, 1e19$ (i.e., modulus). Then $b = b_1 \cdot 1e19 + b_2$ and b_1 and b_2 have 19 digits less than b. The calculation now works as follows:

$$\begin{aligned} a \cdot b/1e38 &= a \cdot (b_1 \cdot 1e19 + b_2)/1e38 \\ &= \frac{a \cdot b_1 \cdot 1e19}{1e38} + \frac{a \cdot b_2}{1e38} \\ &= (a \cdot b_1 + a \cdot b_2/1e19)/1e19 \end{aligned}$$

with rounding direction modified appropriately.

5 Limits on Variables to Prevent Overflows

In this section, we discuss limits that we put on the different variables that ensure that no overflows occur when our calculations are implemented in 256-bit signed fixed point. We refer to the Solidity implementation to discuss some details.

5.1 Limits for Math Operations

A value is representable in 256-bit signed integer iff (ignoring an unimportant difference between signed and unsigned integers) its absolute value is $\leq 2^{255} - 1 \leq 5.78e76$. For the following analyses, we will ignore any signs, identifying a value with its absolute value. For any number x, we let $\overline{x} := x \cdot 1e18$ be the normal-precision encoding and $\overline{\overline{x}} := x \cdot 1e38$ the extra-precision encoding of x. These are the values that are being stored in memory. By regular variables like a and b we always denote the number that is being represented (i.e., the number *not* scaled by $1e18$ or $1e38$) and we denote the encoded values explicitly.

To prevent overflows in the numerical operations in `SignedFixedPoint`, the following constraints need to hold:

- `mul{Up,Down}Mag(a, b)` is ok if $a \cdot b \leq 5.78e40$, where a and b are normal-precision numbers.

This is because as an intermediate result, the function calculates $\overline{a} \cdot \overline{b} = a \cdot b \cdot 1e36$ and this should be $\leq 5.78e76$.

- `div{Up,Down}Mag(a, b)` is ok if $a \leq 5.78e + 40$, where a is a normal-precision number.

  This is because an intermediate result is $\overline{a} \cdot \mathrm{ONE} = a \cdot 1e36$.

- `mulXp(a, b)` is ok if $a \cdot b \leq 5.78$ when a and b are extra-precision numbers.

  This is because the function calculates $\overline{\overline{a}} \cdot \overline{\overline{b}} = a \cdot b \cdot 1e38 \cdot 1e38 = a \cdot b \cdot 1e76$.

- `divXp(a, b)` is ok if $a \leq 5.78$ when a is extra-precision.

  Note that larger extra-precision numbers than that are not problematic per se, but we cannot divide them by another extra-precision number using the default implementation.

- `mul{Up,Down}XpToNp(a, b)` is ok if $a \cdot \max(1, b) \leq 5.78e39$, where a is a normal-precision and b is an extra-precision number.

  This is because $\overline{a}$ is multiplied by $\overline{\overline{b}} \,\%\, 1e19 \leq 1e19$ and by $\overline{\overline{b}}/1e19$.

The validity of addition and subtraction is easy to check and turns out to be dominated by multiplication and division for our purposes.

Note that these functions can also be used with differently-scaled numbers. For example, we can use `divDown(a, b)` when a is an extra-precision and b is a normal-precision number to receive an extra-precision result. In these cases, the limits need to be adjusted.

5.2 Limits on Values

We use the following limits for the different values. These limits are enforced in the functions `validateParams()` and `validateDerivedParamsLimits()` and in `calculateInvariantWithErrors()`. The limits here refer to the represented values, not the encoded values. Note that some of these are normal-precision and some are extra-precision values.

Static Parameters The following limits on the static parameters are checked upon pool creation. Let $\|t\|^2 := t \cdot t$ be the squared l2 norm of a vector t.

- $1 \leq \lambda \leq 1e8$
- $s, c \leq 1$
- $\|(s, c)\|^2 = 1 \pm 1e{-}15$
- $\|\bar{\tau}(\alpha)\|^2 = 1 \pm 1e{-}15$ and $\|\bar{\tau}(\beta)\|^2 = 1 \pm 1e{-}15$
- $d^2 = 1 \pm 1e{-}15$
- $u, v, w, z \leq 1$
- $\frac{1}{A\chi \cdot A\chi - 1} \leq 1e5$. This expression is the denominator from the invariant calculation.

Dynamic Values The following limits apply to values that change across the evolution of the pool. They are checked while the invariant is calculated, which is conveniently the first operation from the math library used in any operation.

- $x + y \leq 1e16$
- $r \leq 3e19$. Whenever an under- and an overestimate is used, the limit applies to the overestimate.

5.3 Analysis of Some Individual Functions

To show that no overflows can occur, we consider each individual operation in the code. It matters which detailed implementation is chosen for our formulas and which values are stored in extra precision. We can simplify our analysis based on the above assumptions as follows:

- Since $s, c, u, v, w, z \leq 1$, multiplication by these values can be ignored in most places. However, the fact that a multiplication takes place cannot necessarily be ignored (this happens to be unproblematic in most cases though).
- Since $d^2 \approx 1$, we can furthermore disregard multiplication and division by d^2 to the same extent as above. While these operations can introduce some magnification, since d^2 is so close to 1 and we only apply division and multiplication by d^2 a limited number of times, the overall effect of this is insignificant.
- We also for the most part ignore divisions by λ to the same extent because they can only reduce the final value, but we cannot rely on that because we might have $\lambda = 1$.

virtualOffset0 As an example, we conduct an overflow analysis for the `virtualOffset0)()` function.

The first operation is `d.tauBeta.x.divXpU(d.dSq)`, or $\mathrm{divXp}(\tau(\beta)_x, d^2)$. This is safe if $\tau(\beta)_x \leq 5.78$, which must of course be satisfied because $\tau(\beta)$ is approximately normed. Note that $\mathrm{termXp} \leq 1$ up to an insignificant error.

The next line employs the operation $\mathrm{mulUpMagU}(\lambda, r)$, which is safe if $\lambda r \leq 5.78e40$, which is true because our constraints even guarantee $\lambda r \leq 1e8 \cdot 3e19 = 3e27$. The following multiplication is immediately seen to be safe as well.

The following operation in the same line is (essentially) $\mathrm{mulUpXpToNpU}(r\lambda, \mathrm{termXp})$ and thus we (essentially) require $r\lambda \leq 5.78e39$, which is well satisfied because of our constraints. Note that at this point $a \leq r\lambda$ essentially.

For the last line, it is sufficient to ensure that $r \leq 5.78e40$ and $r \leq 5.78e39$, both of which are easily guaranteed by our constraints.

calcMinAtxAChiySqPlusAtxSq The first two lines require (essentially) that all of x^2, y^2, and $2xy$ are $\leq 5.78e40$. A sufficient condition for this (which also takes care of additions and subtractions) is that the sum of these, $(x+y)^2 \leq 5.78e40$. This is equivalent to $x+y \leq \sqrt{5.78e40} \geq 2.4e20$. This is well covered by our limit of $x+y \leq 1e16$. The value of `termNp` is bounded above by $(x+y)^2 \leq 1e32$.

The first line for the extra-precision term is not problematic because $s, c, u, v, w, z \leq 1$, and the term itself is at most 4. In particular, since its value is below 5.78, we can apply `divXpU` to it. Since we divide by $d^2 \approx 1$, we do not significantly change the magnitude of the term.

We can compute the first expression for `val`, `(-termNp).mulDownXpToNpU(termXp)` because $\mathrm{termNp} \cdot \mathrm{termXp} \leq 1e32 \cdot 4 = 4e32 \leq 5.78e39$.

The final line is similar. The final operation is a division of a normal-precision value a that is at most $\mathrm{termNp} \leq 1e32$ by the extra-precision value d^2. We could have done this via $\mathrm{divXp}(a, d^2)$, but that would require (noting that a is a normal-precision value) $a \leq 5.78e20$, which we can however not guarantee. So we instead multiply by $1/d^2$ using $\mathrm{mulXpToNp}$. This special function uses additional memory to prevent these types of overflows.