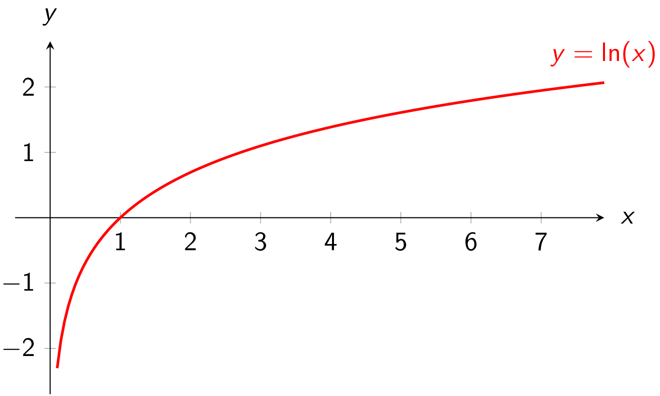
Computing correctly rounded logarithms with fixed-point operations

Julien Le Maire, Florent de Dinechin, Jean-Michel Muller and Nicolas Brunie

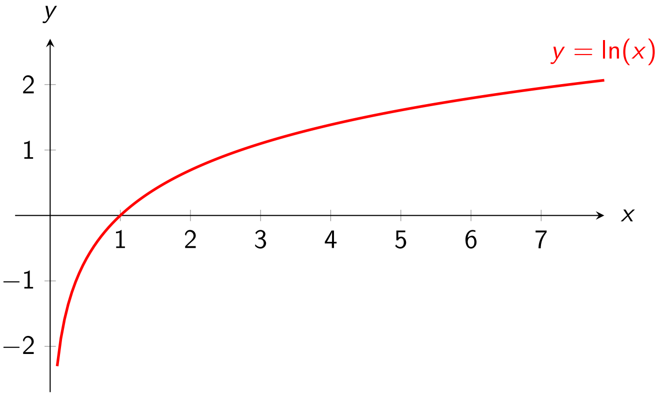
Outline

Logarithm, the mathematical version



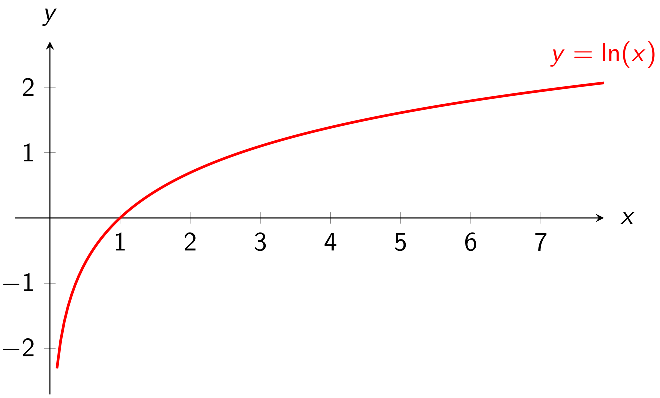
Logarithm, the mathematical version

- $\ln(a \times b) = \ln(a) + \ln(b)$



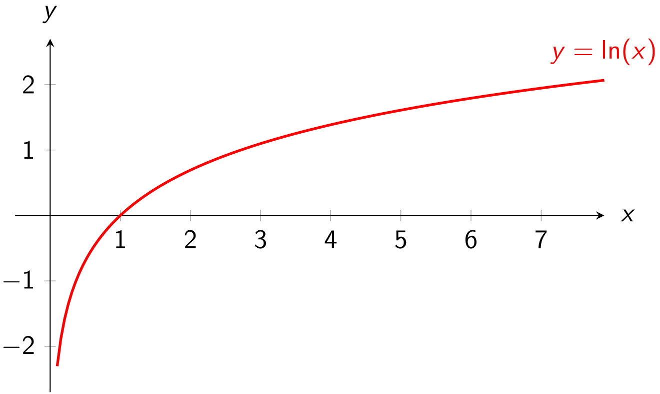
Logarithm, the mathematical version

- $\ln(a \times b) = \ln(a) + \ln(b)$
- $\ln(b^a) = a \times \ln(b)$



Logarithm, the mathematical version

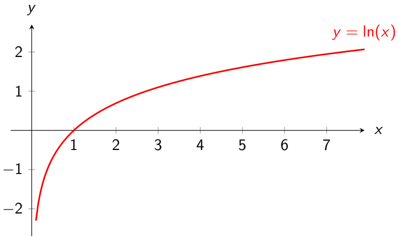
- $\ln(a \times b) = \ln(a) + \ln(b)$
- $\ln(b^a) = a \times \ln(b)$
- Taylor: for x small, $\ln(1+x) \approx x - x^2/2 + x^3/3...$



Logarithm, the floating-point version

The floating point version of the natural logarithm is called `log`
(you will also find `log2` and `log10` and a few others)

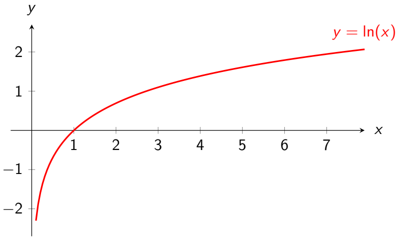
$$\forall x \in \mathbb{F}_{64} \quad \texttt{log(x)} = \circ\left(\ln(x)\right)$$



Logarithm, the floating-point version

The floating point version of the natural logarithm is called `log`
(you will also find `log2` and `log10` and a few others)

$$\forall x \in \mathbb{F}_{64} \quad \mathtt{log(x)} = \circ(\ln(x))$$



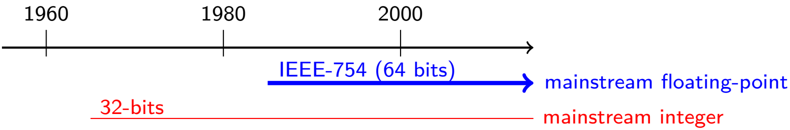
An experiment

Implementing the *floating-point* logarithm function

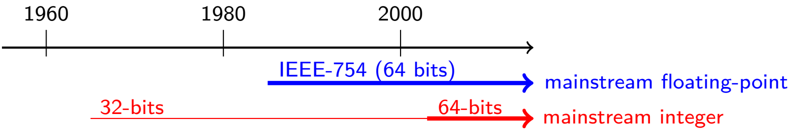
- using only *integer* arithmetic
- for *performance*

(previous work motivated by *lack of FP hardware*)

Why using fixed-point arithmetic?



Why using fixed-point arithmetic?



- 64-bit floating-point, but only 52-bit precision
  - if you can predict the value of the exponent, exponent bits are wasted bits.

Integer better than floating-point?

- modern 64-bit machines offer all sort of useful integer instructions
  - addition
  - multiplication $64x64 \rightarrow 128$ (*mulq*)
  - count leading zeroes, shifts (*lzcnt, bsr*)

Integer better than floating-point?

- modern 64-bit machines offer all sort of useful integer instructions
  - addition
  - multiplication $64x64 \rightarrow 128$ (*mulq*)
  - count leading zeroes, shifts   (*lzcnt, bsr*)
- most operations are faster on integers, especially addition
  (which more or less defines the processor cycle time)

Integer better than floating-point?

- modern 64-bit machines offer all sort of useful integer instructions
  - addition
  - multiplication 64*x*64 $\rightarrow$ 128 (*mulq*)
  - count leading zeroes, shifts (*lzcnt, bsr*)
- most operations are faster on integers, especially addition
  (which more or less defines the processor cycle time)
- multiprecision faster and simpler using integers

Integer better than floating-point?

- modern 64-bit machines offer all sort of useful integer instructions
  - addition
  - multiplication $64x64 \to 128$ (*mulq*)
  - count leading zeroes, shifts (*lzcnt, bsr*)
- most operations are faster on integers, especially addition (which more or less defines the processor cycle time)
- multiprecision faster and simpler using integers
- small multiprecision out of the box: mainstream compilers (*gcc, clang, icc*) support `__int_128`
  - addition $128x128 \to 128$ (*add, adc*)
  - shift on two registers (*shld, shrd*)

Integer better than floating-point?

- modern 64-bit machines offer all sort of useful integer instructions
  - addition
  - multiplication $64x64 \rightarrow 128$ (*mulq*)
  - count leading zeroes, shifts (*lzcnt, bsr*)
- most operations are faster on integers, especially addition
  (which more or less defines the processor cycle time)
- multiprecision faster and simpler using integers
- small multiprecision out of the box:
  mainstream compilers (*gcc, clang, icc*) support `__int_128`
  - addition $128x128 \rightarrow 128$ (*add, adc*)
  - shift on two registers (*shld, shrd*)

Caveat: integer SIMD/vector support still lagging behind FP
(no vector multiplication)

Outline

Introduction and context

Algorithm

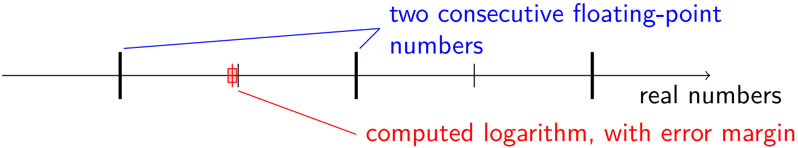
Results and comparisons

Conclusions

On-demand accuracy

Muller and Lefèvre solved the table maker dilema for ln

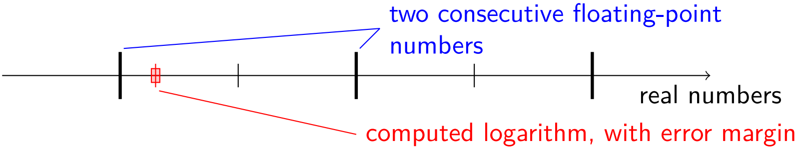
Computing the log with an error $\leq 2^{-113}$ enables correct rounding



On-demand accuracy

Muller and Lefèvre solved the table maker dilema for ln

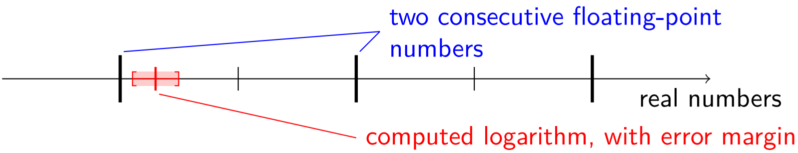
Computing the log with an error $\leq 2^{-113}$ enables correct rounding



On-demand accuracy

Muller and Lefèvre solved the table maker dilema for ln

Computing the log with an error $\leq 2^{-113}$ enables correct rounding



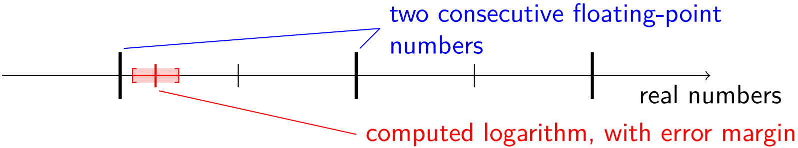
CRLibm refinement of Ziv's technique:

- First step: quick-and-dirty evaluation of $\ln(x)$
  (just accurate enough to ensure correct rounding in most cases)
- test if rounding can be decided
- if not (rarely), recompute $\ln(x)$ with the worst-case accuracy

On-demand accuracy

Muller and Lefèvre solved the table maker dilema for ln

Computing the log with an error $\leq 2^{-113}$ enables correct rounding



CRLibm refinement of Ziv's technique:

- First step: quick-and-dirty evaluation of $\ln(x)$
  (just accurate enough to ensure correct rounding in most cases)
- test if rounding can be decided
- if not (rarely), recompute $\ln(x)$ with the worst-case accuracy

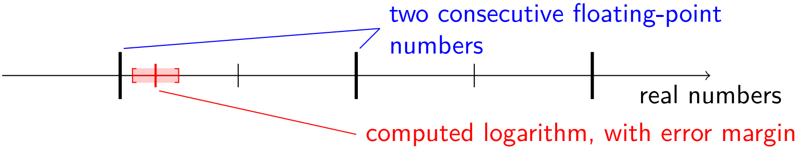
Trade-off between first and second steps:

$$MeanTime = \quad Time(1st\ step) \quad + \quad \Pr[need\ 2nd\ step] \cdot Time(2nd\ step)$$

On-demand accuracy

Muller and Lefèvre solved the table maker dilema for ln

Computing the log with an error $\leq 2^{-113}$ enables correct rounding



CRLibm refinement of Ziv's technique:

- First step: quick-and-dirty evaluation of $\ln(x)$
  (just accurate enough to ensure correct rounding in most cases)
- test if rounding can be decided
- if not (rarely), recompute $\ln(x)$ with the worst-case accuracy

Trade-off between first and second steps:

$$MeanTime = \quad Time(1st\ step) \quad + \quad \Pr[need\ 2nd\ step] \cdot Time(2nd\ step)$$

Best so far: $Time(2nd\ step) \approx 10 \times Time(1st\ step)$
This work: $Time(2nd\ step) \approx 2 \times Time(1st\ step)$

The big picture

1. Filter special cases (negative numbers, ∞, ...)
2. Argument range reduction
3. Polynomial approximation
4. Solution reconstruction
5. Error evaluation and rounding test
6. If more accuracy needed:
   Rerun the steps 3 and 4 with the worst-case accuracy.

First argument range reduction

$$\mathit{input} = 2^{E} \cdot (1 + x)$$
$$\ln(\mathit{input}) = E \cdot \ln(2) + \ln(1 + x)$$

First argument range reduction

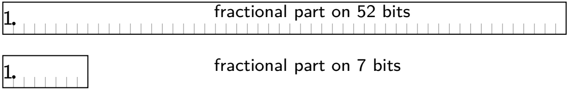
$$input = 2^E \cdot (1 + x)$$
$$\ln(input) = E \cdot \ln(2) + \ln(1 + x)$$

Evaluation algorithm:

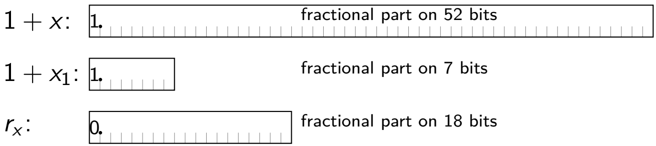
- approximate $\ln(1 + x)$ with a polynomial $p(x)$
  degree needed: at least 26
- evaluate $E \cdot \ln(2)$
- add both terms

Tang's range reduction

$1 + x$:
$1 + x_1$:



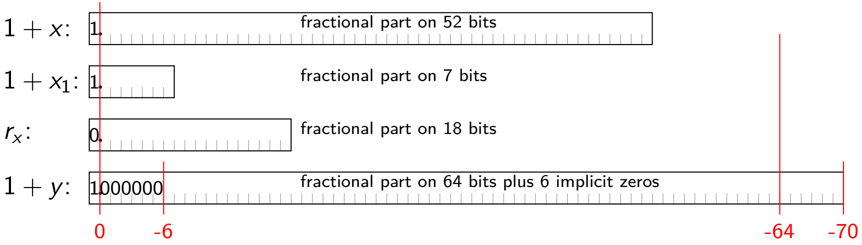
Tang's range reduction



- A table, addressed by x_1, the 7 most significand bits of x, stores

$$r_x \approx \frac{1}{1+x_1} \approx \frac{1}{1+x} \quad \text{and} \quad \ln\left(\frac{1}{r_x}\right)$$

Tang's range reduction

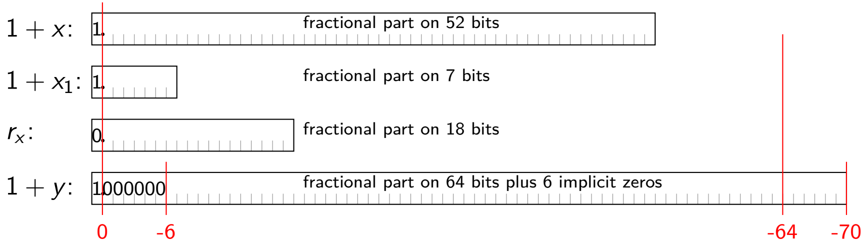


- A table, addressed by x_1, the 7 most significand bits of x, stores

$$r_x \approx \frac{1}{1+x_1} \approx \frac{1}{1+x} \quad \text{and} \quad \ln\left(\frac{1}{r_x}\right)$$

- Define $\quad 1 + y = r_x \cdot (1 + x) \approx 1$
- Then $\quad \ln(1 + x) = \ln(1 + y) + \ln(\frac{1}{r_x})$

Tang's range reduction algorithm



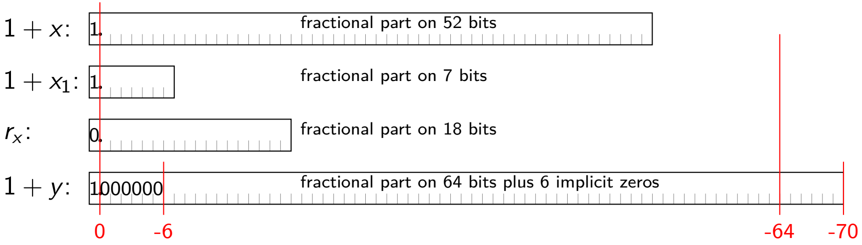
- Extract the index x_1
- Read, from a table addressed by x_1, both r_x and $\ln(\frac{1}{r_x})$
- compute $y = r_x \cdot (1 + x) - 1$ (exactly)
- approximate $\ln(1 + y)$ with a polynomial $p(y)$

  Degree needed: 8

- add it all:

$$\ln(input) \approx E \cdot \ln(2) + p(y) + \ln(\frac{1}{r_x})$$

Tang's range reduction

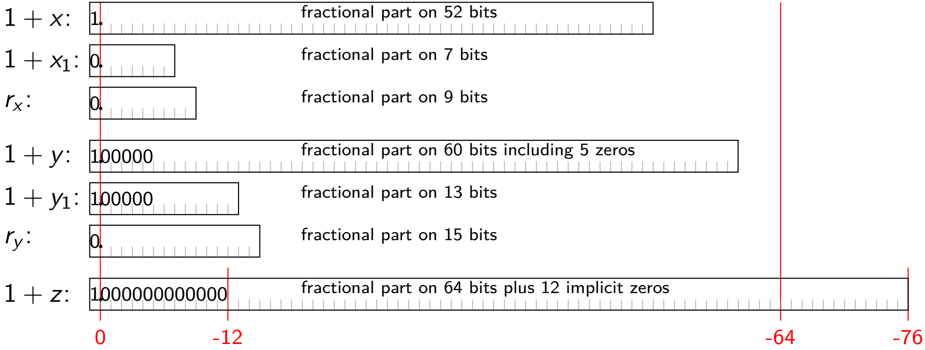


$$y = r_x \cdot (1 + x) - 1$$

With $1 + x$ on 53 bits we can tabulate r_x on 18 bits:

- the exact product would need 71 bits
- but we can predict the 7 leading bits
- ... so we can let them overflow quietly and use a $64 \times 64 \to 64$ multiplication.

Two levels of Tang reduction



$$x \in [0, 1)$$
$$y \in \left[0, 2^{-6.41504}\right)$$
$$z \in \left[0, 2^{-12.6747}\right)$$

x_1 *takes* 64 *different values*

y_1 *takes* 96 *different values*

the whole reduction of x to z is computed exactly in 64-bit int.

A few Pareto points in the design space

Table size (bytes)	**degree 1st**	**degree 2nd**
39,936	3	5
12,288	3	6
4,032	**4**	**7**
2,240	4	8
2,016	4	9
900	5	10
594	6	12
298	7	14

Why stop at two levels of reduction?

Answer is: diminushing return.

For a target accuracy of 2^{-60}:

	interval of x	degree needed
No reduction	$[-1/2, 1/2]$	29
1 level	$[-2^{-7}, 2^{-7}]$	8
2 levels	$[-2^{-12}, 2^{-12}]$	4
3 levels	$[-2^{-18}, 2^{-18}]$	3

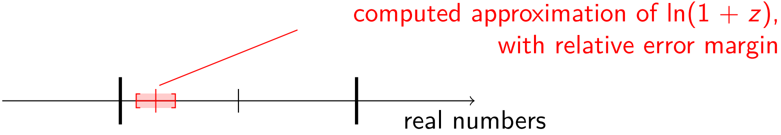
Adding more levels will cost more operations than it saves...

Polynomial approximation (advertisement)

We want to approximate $log(1+z)$ on an interval around 0.
Use the (now standard) tool set to obtain it.

- Sollya:
  - finds a machine-efficient polynomial $P(z)$
  - computes a safe bound on the approximation error $P(z) - \ln(1+z)$
- Gappa: bounds the accumulation of rounding errors when evaluating $P(z)$ in C

We obtain a Coq proof of the error:



Reconstructing the solution

$$input = 2^e \cdot (1 + x)$$

Reconstructing the solution

$$input = 2^e \cdot \frac{1}{r_x} \cdot (1+y)$$

Reconstructing the solution

$$input = 2^e \cdot \frac{1}{r_x} \cdot \frac{1}{r_y} \cdot (1 + z)$$

Reconstructing the solution

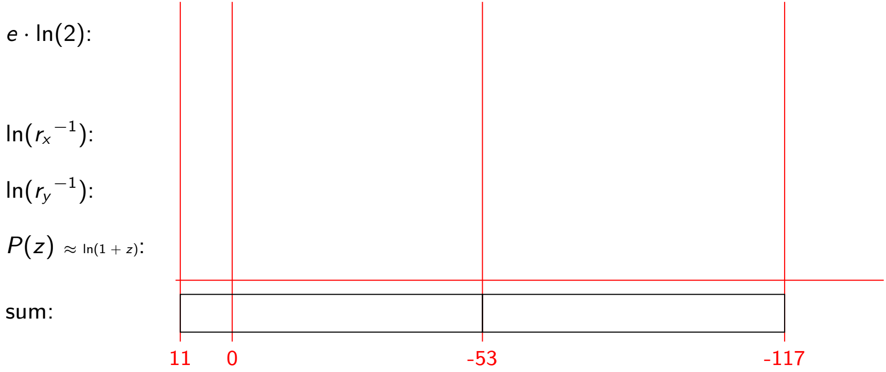
$$input = 2^e \cdot \frac{1}{r_x} \cdot \frac{1}{r_y} \cdot (1 + z)$$

$$\ln(input) = e \cdot \ln(2) \quad + \quad \ln({r_x}^{-1}) \quad + \quad \ln({r_y}^{-1}) \quad + \quad \ln(1 + z)$$

Reconstructing the solution

$$input = 2^e \cdot \frac{1}{r_x} \cdot \frac{1}{r_y} \cdot (1+z)$$

$$\ln(input) = e \cdot \ln(2) \quad + \quad \ln({r_x}^{-1}) \quad + \quad \ln({r_y}^{-1}) \quad + \quad \ln(1+z)$$

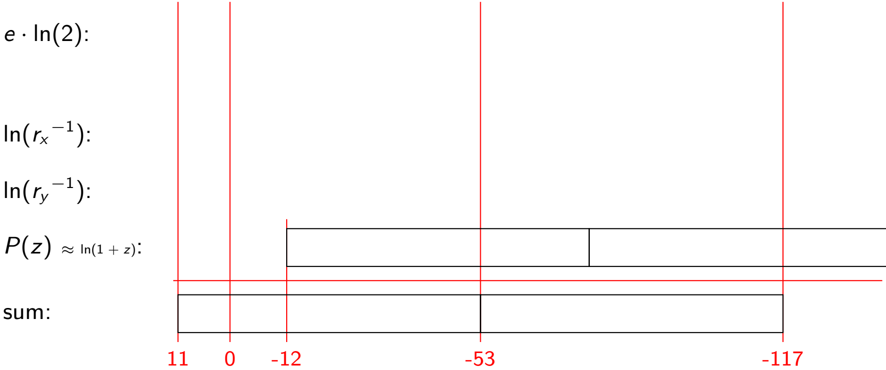


"If we can predict the exponents, exponent bits are wasted bits"

Reconstructing the solution

$$input = 2^e \cdot \frac{1}{r_x} \cdot \frac{1}{r_y} \cdot (1+z)$$

$$\ln(input) = e \cdot \ln(2) \quad + \quad \ln({r_x}^{-1}) \quad + \quad \ln({r_y}^{-1}) \quad + \quad \ln(1+z)$$

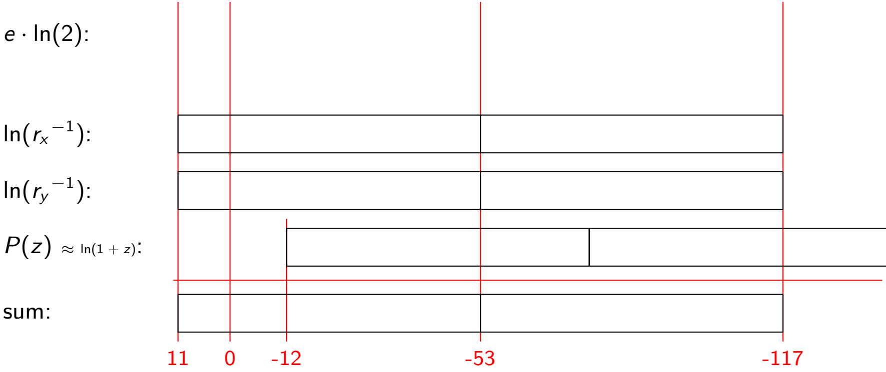


"If we can predict the exponents, exponent bits are wasted bits"

Reconstructing the solution

$$input = 2^e \cdot \frac{1}{r_x} \cdot \frac{1}{r_y} \cdot (1+z)$$

$$\ln(input) = e \cdot \ln(2) \quad + \quad \ln({r_x}^{-1}) \quad + \quad \ln({r_y}^{-1}) \quad + \quad \ln(1+z)$$

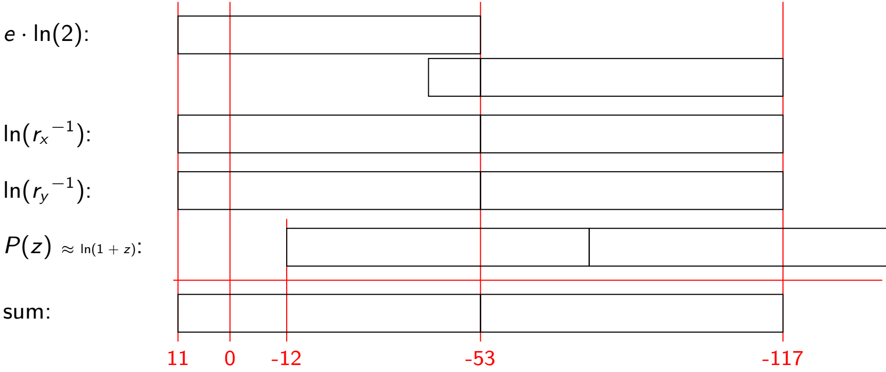


"If we can predict the exponents, exponent bits are wasted bits"

Reconstructing the solution

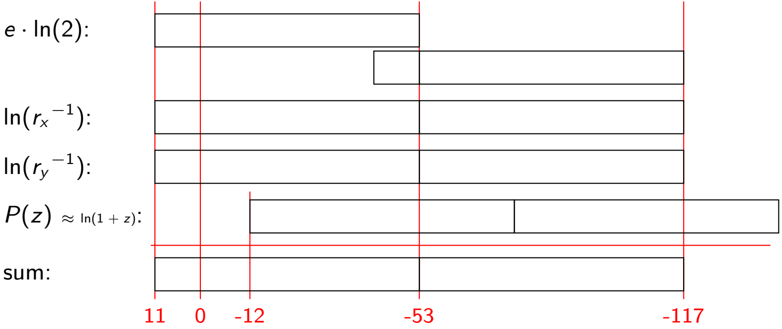
$$input = 2^e \cdot \frac{1}{r_x} \cdot \frac{1}{r_y} \cdot (1+z)$$

$$\ln(input) = e \cdot \ln(2) \quad + \quad \ln({r_x}^{-1}) \quad + \quad \ln({r_y}^{-1}) \quad + \quad \ln(1+z)$$

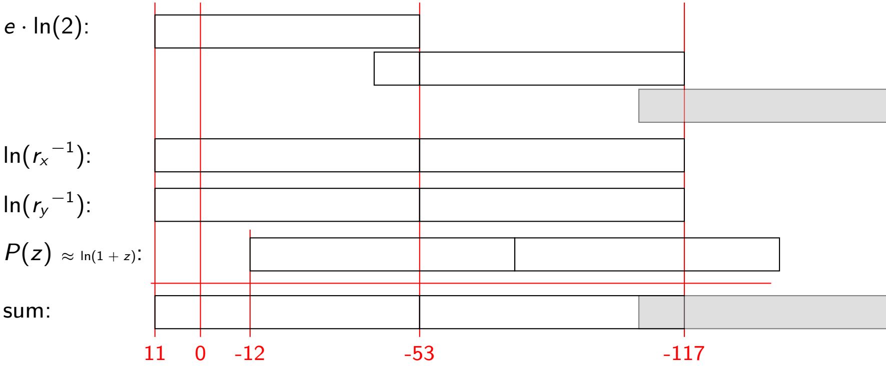


"If we can predict the exponents, exponent bits are wasted bits"

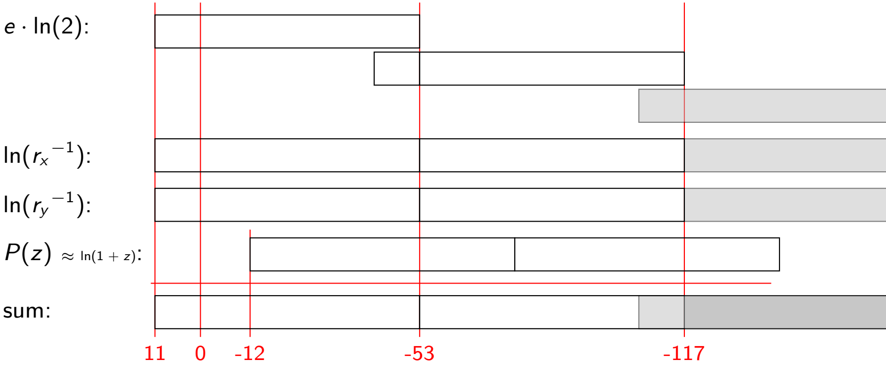
Error evaluation



Error evaluation

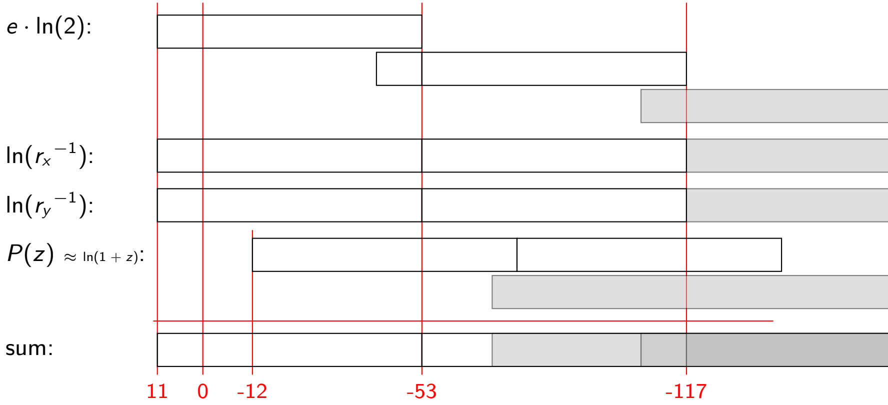


$$\epsilon < (|e|) \cdot 2^{-117}$$



$$\epsilon < (|e| + 1 + 1) \cdot 2^{-117}$$

Error evaluation

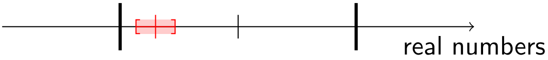


$$\epsilon < \left(|e| + 1 + 1 + P(z) \cdot 2^{-59}\right) \cdot 2^{-117}$$

Rounding test

Simple technique: compute the two bounds of the interval,
and see if they round to the same mantissa

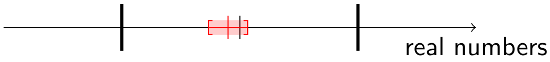
(two additions, a xor and a shift)



Rounding test

Simple technique: compute the two bounds of the interval, and see if they round to the same mantissa

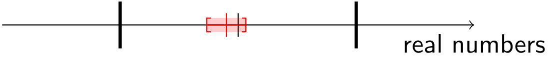
(two additions, a xor and a shift)



Rounding test

Simple technique: compute the two bounds of the interval,
and see if they round to the same mantissa

(two additions, a xor and a shift)



For comparison, the proof of the floating-point-based rounding test (invented by Ziv and used in CRLibm) is an 18-page paper that took 20 years to publish...

Second step

- Use 3 words instead of 2 for the precomputed ln(2)
- Use a much more accurate polynomial:
  - with coefficients on 128 bits instead of 64
    (but z is still only a 64-bit number)
  - and using a higher degree polynomial

Outline

Introduction and context

Algorithm

Results and comparisons

Conclusions

Implementation parameters of correctly rounded implementations

	glibc	crlibm-td	crlibm-de	cr-FixP
degree pol. 1	3/8	6	7	4
degree pol. 2	20	12	14	7
tables size	13 Kb	8192 bytes	6144 bytes	4032 bytes
% accurate phase	N/A	1.5	0.4	4.4

Average and max runing time (in processor cycles)

Pentium timing (lower is better)

cycles	*MKL*	**glibc**	**crlibm**	**cr-de**	**cr-FixP**
avg time	*25*	90	69	46	49
max time	*25*	11,554	642	410	79

Timing breakdown on two processors

cycles	**Core i5**	**Bostan**
System	`glibc` 90	`newlib` *105*
quick phase alone	*42*	*94*
accurate phase alone	74	181
both phases (avg time)	49	121
both phases (max time)	79	225

Slanted means: no correct rounding

Average and max runing time (in processor cycles)

Pentium timing (lower is better)

cycles	*MKL*	**glibc**	**crlibm**	**cr-de**	**cr-FixP**
avg time	*25*	90	69	46	49
max time	*25*	11,554	642	410	79

Timing breakdown on two processors

cycles	**Core i5**	**Bostan**
System	`glibc` 90	`newlib` *105*
quick phase alone	*42*	*94*
accurate phase alone	74	181
both phases (avg time)	49	121
both phases (max time)	79	225

Slanted means: no correct rounding

Average and max runing time (in processor cycles)

Pentium timing (lower is better)

cycles	*MKL*	**glibc**	**crlibm**	**cr-de**	**cr-FixP**
avg time	*25*	90	69	46	49
max time	*25*	11,554	642	410	79

Timing breakdown on two processors

cycles	**Core i5**	**Bostan**
System	`glibc` 90	`newlib` *105*
quick phase alone	*42*	*94*
accurate phase alone	74	181
both phases (avg time)	49	121
both phases (max time)	79	225

Slanted means: no correct rounding

Average and max runing time (in processor cycles)

Pentium timing (lower is better)

cycles	*MKL*	**glibc**	**crlibm**	**cr-de**	**cr-FixP**
avg time	*25*	90	69	46	49
max time	*25*	11,554	642	410	79

Timing breakdown on two processors

cycles	**Core i5**	**Bostan**
System	`glibc` 90	`newlib` *105*
quick phase alone	*42*	*94*
accurate phase alone	74	181
both phases (avg time)	49	121
both phases (max time)	79	225

Slanted means: no correct rounding

Outline

Introduction and context

Algorithm

Results and comparisons

Conclusions

Conclusions

- Better range reduction thanks to a wider format
- ... leading to improvements in polynomial degree and table size
- Faster multiprecision due to higher precision
- Able to reuse some computations of the fast step in the accurate step
- Alternative rounding test for acurate step
- Probability to launch 2nd step is high,

  but this is acceptable since 2nd step is so cheap

Conclusions

- Better range reduction thanks to a wider format
- ... leading to improvements in polynomial degree and table size
- Faster multiprecision due to higher precision
- Able to reuse some computations of the fast step in the accurate step
- Alternative rounding test for acurate step
- Probability to launch 2nd step is high,
  but this is acceptable since 2nd step is so cheap

- Competitive against state-of-the-art
- Worst case improved compared to others implementations
- Second step-only version is a viable alternative

Conclusions

- Better range reduction thanks to a wider format
- ... leading to improvements in polynomial degree and table size
- Faster multiprecision due to higher precision
- Able to reuse some computations of the fast step in the accurate step
- Alternative rounding test for acurate step
- Probability to launch 2nd step is high,
  but this is acceptable since 2nd step is so cheap

- Competitive against state-of-the-art
- Worst case improved compared to others implementations
- Second step-only version is a viable alternative

Limitations:

- Require 64 bits integer support
- No support for vectorization

Thanks for your attention

Any question ?

Outline

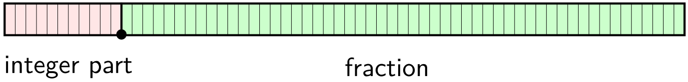
Bonus: a floating-point in, fixed-point out variant

Some code

details on the solution reconstruction

Floating-point in, fixed-point out

- output: fixed-point, 11 bits integer part, 53 bit fractional part



- target absolute accuracy 2^{-52}

output format	absolute accuracy	table size	Core i5 cycles	Bostan cycles
Fix64	2^{-52}	2304	24	66
Fix128	2^{-116}	4032	60	179
double (libm)	2^{-42}		90	105

- Fix64 is the code of the first step only,
  without the conversion to float.
  - tweak: poly degree 3 only for abs. accuracy 2^{-59}
- Fix128 is the code of the second step only, without the conversion to float.

Motivation

TKF91 : DNA sequence alignment algorithm

- dynamic programming algorithm:
  alignment as a path within a 2D array.
- borders of an array initialized with log-likelihoods
- then array filled using recurrence formulae
  that involve only max and $+$ operations.

All current implementations of this algorithm use a floating-point array, but

- int64 $+$ and max are 1-cycle, vectorizable, and exact operations;
- absolute accuracy of initialization logs: up to 2^{-42} with FP log, 2^{-52} with FixP log.

Only partial experiments

- Improvement in accuracy measured
- No noticeable improvement in performance

Outline

Bonus: a floating-point in, fixed-point out variant

Some code

details on the solution reconstruction

Special cases: businesss as usual

```
/* reinterpret x to manipulate its bits more easily */
uint64_t inputbits = ((union { double d; uint64_t u; }){input}).u;
int xe = inputbits >> 52;

/* filter the special cases: !(x is normalized and 0 < x < +Inf) *
if (0x7FEu <= (unsigned)xe - 1u) {
    /* x = +- 0:     raise a DivideByzero, return -Inf */
    if ((xbits & ~(1ull << 63)) == 0) return -1.0/0.0;
    /* x < 0.0:      raise a InvalidOperation, return a qNaN */
    if ((xbits &  (1ull << 63)) != 0) return (x-x)/0;
    /* x = qNaN:     return a qNaN
       x = sNaN:     raise a InvalidOperation, return a qNaN
       x = +Inf:     return +Inf */
    if (xe != 0) return x+x;
    /* x subnormal: change x to a normalized number */
    else {
        int u = clz64(xbits) - 12;
        xbits <<= u + 1;
        xe -= u;
    }
}
xe -= 1023;
```

Only interesting line: the subnormal management

Argument reduction

```
/*  input = 2^xe * (1 + X)  */
uint64_t x = inputbits & 0xFFFFFFFFFFFFFull;

/*  1 + X = (1/Ri) * Y  */
uint8_t x1 = x >> (52 - ARG_REDUC_1_PREC);
uint64_t y = argReduc1_inv[x1] * (x + (1ull << 52));
__builtin_prefetch(& argReduc1_log[x1], 0, 0);

/*  Y = (1/S) * (1 + Z)  */
/* with dZ = dz/2^(52 + ARG_REDUC_1_SIZE + ARG_REDUC_2_SIZE)
 and 1/S = argReduc2[si].val/2^ARG_REDUC_2_SIZE */
uint8_t y1 = (y >> (52 + ARG_REDUC_1_SIZE - ARG_REDUC_2_PREC))
           - (1u << ARG_REDUC_2_PREC);
uint64_t z = argReduc2_inv[y1] * y; // +1 part removed by overflow
__builtin_prefetch(& argReduc2_log[y1], 0, 0);
```

Polynomial evaluation

```
/* Polynomial approximation of log(1+Z)/Z ~= P(Z)  and  Z*P(Z) */
static const uint64_t a4 = UINT64_C(0x3ffc147cb4539237);
static const uint64_t a3 = UINT64_C(0x5555553dc70f0dfd);
static const uint64_t a2 = UINT64_C(0x7fffffffd574fd);
static const uint64_t a1 = UINT64_C(0xfffffffffffffa);
uint64_t pz = a1 - (highmul(z,
                a2 - (highmul(z,
                      a3 - (highmul(z, a4) >> 12)
                     ) >> 12)
               ) >> 12);
uint128_t zpz = fullmul(z, pz);
```

```
/* Polynomial approximation without shifts:
   replace highmul(z, a) >> 12  with  highmul(z, a >> 12) */
static const uint64_t a4 = UINT64_C(0x000000003ffc147);
static const uint64_t a3 = UINT64_C(0x000005555553dc6);
static const uint64_t a2 = UINT64_C(0x0007fffffffd57);
static const uint64_t a1 = UINT64_C(0xfffffffffffffa);
uint64_t pz = a1 - highmul(z,a2 - highmul(z,a3 - highmul(z,a4)));
uint128_t zpz = fullmul(z, pz);
```

Reconstructing the solution

```
/* Compute part of the result that don't depend on Z
   (xe*log(2) + log(1/Ri) + log(1/Si)) */
uint128_t cstpart = fullimul(xe, log2fw_mid)
     + UINT128((int64_t)xe * log2fw_high, 0) // fullmul not needed
     + UINT128(argReduc1[ri].log_hi, argReduc1[ri].log_lo)
     + UINT128(argReduc2[si].log_hi, argReduc2[si].log_lo);

/* Polynomial approximation of log(1+Z)/Z ~= P(Z)  and  Z*P(Z) */
...

/* Assemble the two parts, compute sign, mantissa and exponent */
uint128_t longres = cstpart + (zpz >> (11 + IMPLICIT_ZEROS));
uint64_t sign = - (HI(longres) >> 63);    // 0 or ~0
// if sign != 0, this is longres = ~ longres (-a = ~a + 1)
// to avoid the +1 approx, do:
//    longres = ((int64_t)sign + longres) ^ UINT128(sign, sign);
longres ^= UINT128(sign, sign);

int u = clz64(HI(longres)) + 1;
int exponent = 11 - u;
uint64_t mantissa = (HI(longres) << u) | (LO(longres) >> (64 - u))
```

Rounding test and conversion

```
/* Compute the maximal absolute error (aligned with longres):
     + 2 + abs(xe)      for xe*log(2), log(1/Ri) and log(1/Si)
     + 1 + zpz>>(POLYNOMIAL_PREC+IMPLICIT_ZEROS+11) for the polyno
 If result*(1 +- maxRelErr) are not rounded to the same number, we
uint64_t maxAbsErr = 3 + abs(xe) + (HI(zpz) >> (POLYNOMIAL_PREC +
uint64_t maxRelErr = (maxAbsErr >> (64 - u)) + 1;
if (((mantissa + maxRelErr) ^ (mantissa - maxRelErr)) >> 11) {
        return log_rn_accurate (cstpart, z, xe);
}

/* Assemble the computed result */
uint64_t resultbits = ((uint64_t)sign << 63)
        + ((uint64_t)(exponent+1023) << 52)
        + (mantissa >> 12)
        + ((mantissa >> 11) & 1); /* round to nearest */
return (union { uint64_t u; double d; }){ resultbits }.d;
```

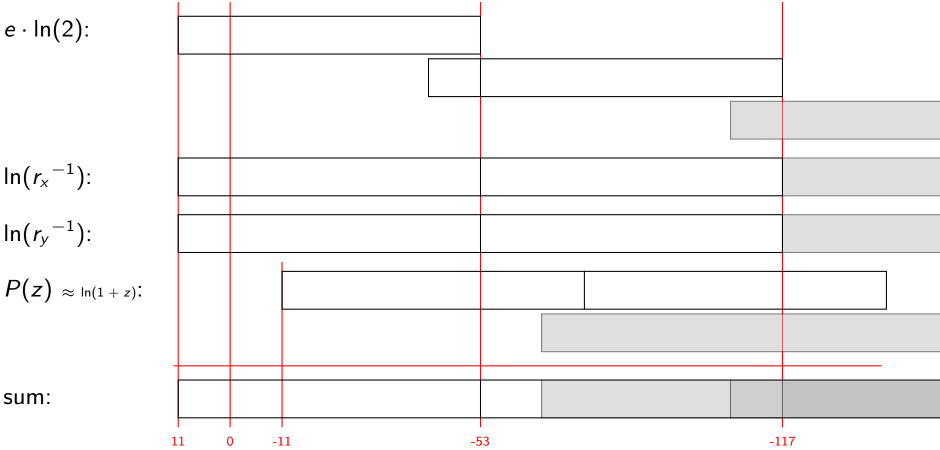
Outline

Bonus: a floating-point in, fixed-point out variant

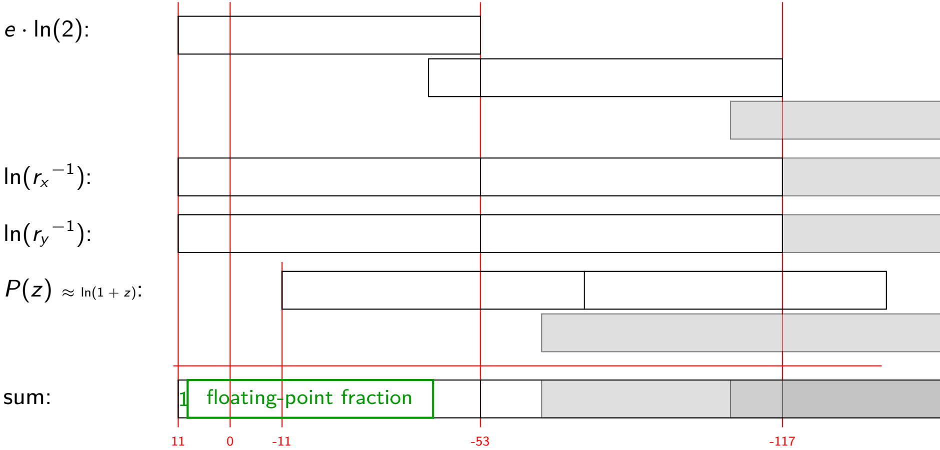
Some code

details on the solution reconstruction

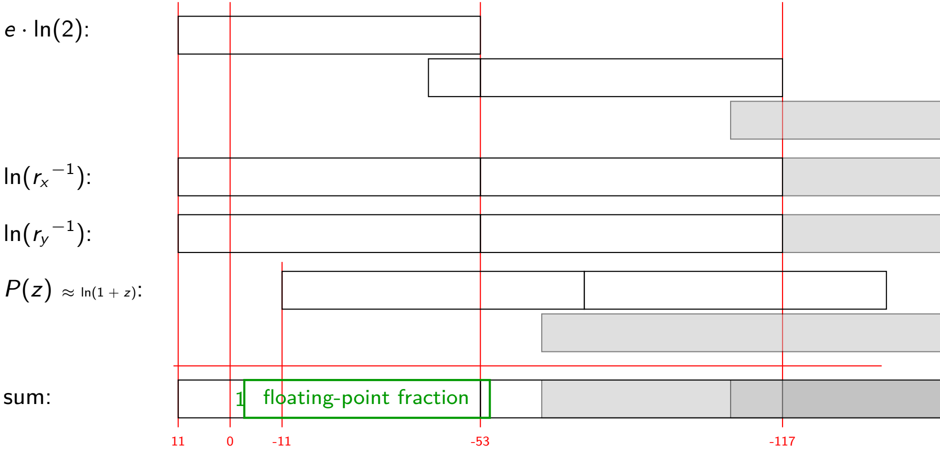
Reconstructing the solution



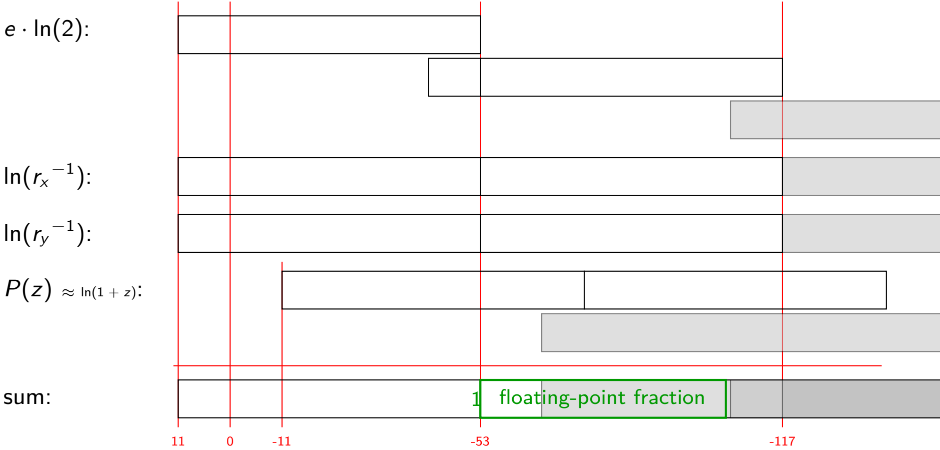
Reconstructing the solution



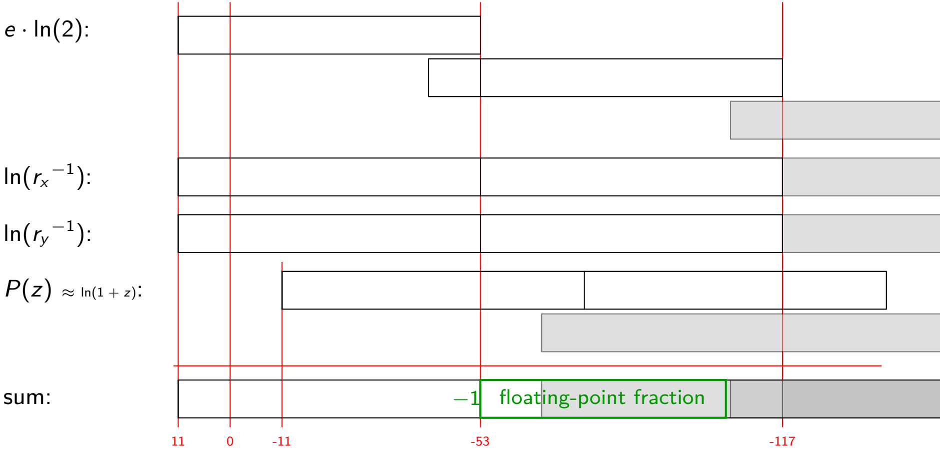
Reconstructing the solution



Reconstructing the solution



Reconstructing the solution



Reconstructing the solution