Floating-Point Training
Module 5
Software Tools

Paul Zimmermann, Inria Nancy

Plan of the training

- Module 1: The IEEE 754 Standard
- Module 2: Math Fundamentals
- Module 3: Core Algorithms
- Module 4: Elementary Function Approximation
- **Module 5: Software Tools**

Module 5: Software Tools

Correct rounding with GNU MPFR

Manipulating floating-point numbers with SageMath

Computing minimax polynomial approximations with Sollya

Bounding rounding errors with Gappa

Computing hard-to-round cases with BaCSeL

Correct rounding with GNU MPFR

Correct rounding with GNU MPFR

GNU MPFR is a C library for multiple-precision arithmetic with correct rounding.

It performs correct rounding for basic arithmetic (addition, subtraction, multiplication, division, square root) as in IEEE 754, but also for mathematical functions, and base conversions (input/output).

It implements all C99 functions, and all new functions in C23 (except some taking integer arguments).

Usages of GNU MPFR

- reference implementation for correctly-rounded routines
- exhaustive testing of generic algorithms for small precision $p = 1, 2, 3, ...$
- through SageMath for example, debug floating-point programs
- used by GCC for constant folding

Constant folding

```
#include <stdio.h>
#include <math.h>

int main() {
  float x = 0x1.ffff7cp-1f, y = 21325791.0f;
  printf ("powf (x, y) = %a\n", powf (x, y));
}

$ gcc -O3 pow.c && ./a.out
powf (x, y) = 0x1.f45804p-122

$ gcc -fno-builtin pow.c -lm && ./a.out
powf (x, y) = 0x1.f45806p-122
```

Floating-point variables in GNU MPFR

Each floating-point variable has its own precision (in bits), which can vary from 1 to any 64-bit integer:

```
mpfr_t x, y, z;
mpfr_init2 (x, 4);
mpfr_init2 (y, 64);
mpfr_init2 (z, 2025);
```

Usually one does not change the precision of a variable, but this is possible (useful for Newton's iteration).

Initializing a variable

One can initialize a variable from the standard C types:

```
mpfr_set_ui (x, 17, MPFR_RNDN);
mpfr_set_d (y, 0.1, MPFR_RNDZ);
mpfr_set_str (z, "0.1", 10, MPFR_RNDU);
```

In the 1st example, the rounding mode is needed, because 17 has to be rounded if the precision of x is less than 5.

Warning: in the 2nd example, 0.1 is first converted by the compiler to the nearest binary64 number `0x1.999999999999ap-4`, thus y has only 53 bits of accuracy with respect to $1/10$.

In constrast, z is initialized to the closest 2025-bit number above $1/10$.

Rounding modes

GNU MPFR provides the following rounding modes:

- `MPFR_RNDN`: round to nearest, ties to even (roundTiesToEven in IEEE 754);
- `MPFR_RNDZ`: round toward zero (roundTowardZero in IEEE 754);
- `MPFR_RNDU`: round toward $+\infty$ (roundTowardPositive in IEEE 754);
- `MPFR_RNDD`: round toward $-\infty$ (roundTowardNegative in IEEE 754);
- `MPFR_RNDA`: round away from zero, symmetric of `MPFR_RNDZ` (not to be confused with round to nearest, ties away from zero);
- `MPFR_RNDF`: faithful rounding (experimental).

Mathematical functions

```
mpfr_t x, y;
mpfr_init2 (x, 17);
mpfr_init2 (y, 42);
mpfr_set_ui (x, 53, MPFR_RNDN);
int inex = mpfr_sin (y, x, MPFR_RNDN);
```

x is set (exactly) to 53, then $\sin(53)$ is rounded to nearest to 42 bits and stored into y.

The return value `inex` is the ternary value:

- `inex` < 0 iff $y < \sin(x)$;
- `inex` $= 0$ iff $y = \sin(x)$;
- `inex` > 0 iff $y > \sin(x)$.

Special case when the even rule is used to break ties: `inex` is set to -2 or $+2$. This is useful to detect double-rounding issues.

Exponent range

The default exponent range is huge (exponent stored in a long).

To reduce the exponent range:

```
mpfr_set_emax (42);
mpfr_set_emin (-17);
```

In MPFR, the significand is between 0.5 and 1 in absolute value. (This differs from IEEE 754 where it is between 1 and 2.) As a consequence, the largest positive number with the above example is just below 2^{42}, and the smallest positive one is 2^{-18}.

Contrary to the precision, the exponent range is shared by all numbers. You can change it during a computation, but you have to ensure all previous numbers have valid exponent in the new range.

Subnormal numbers

MPFR has no subnormal numbers. But `mpfr_subnormalize` can emulate them:

```
mpfr_t x;
mpfr_set_emin (-1073);
mpfr_set_emax (1024);
mpfr_init2 (x, 53);
mpfr_set_ui (x, ..., MPFR_RNDN);
int inex = mpfr_sin (x, x, MPFR_RNDN);
inex = mpfr_subnormalize (x, inex, MPFR_RNDN);
```

The final value of x is rounded to binary64 with gradual underflow, and `inex` is the corresponding ternary value.

The value of `emax` is the maximal binary64 exponent (with significand in $[0.5, 1)$), and `emin` is $-1074 + 1$, where 2^{-1074} is the smallest positive subnormal.

Input/output from/to strings

To convert the decimal number 0.1 to the variable z, with rounding towards $+\infty$:

```
mpfr_set_str (z, "0.1", 10, MPFR_RNDU);
```

To output the variable z to a decimal string of 17 digits and rounding towards zero:

```
mpfr_out_str (stdout, 10, 17, z, MPFR_RNDZ);
```

MPFR also guarantees correct rounding for input/output conversions.

Input/output from/to strings

Special value 0 for the number of output digits (for rounding to nearest):

```
mpfr_out_str (stdout, 10, 0, z, MPFR_RNDN);
```

When the output string is read again with rounding to nearest and the same precision as z, we recover the exact same value.

If you print a binary32 (resp. binary64) number with $m = 9$ digits (resp. $m = 17$ digits), and you read it back (all to nearest), you get the same number. General formula:

$$m = 1 + \left\lceil \frac{p \log(2)}{\log(10)} \right\rceil$$

Emulating a binary64 function

Extracted from CORE-MATH:

```
double ref_exp (double x) {
  mpfr_t y;
  mpfr_exp_t emin = mpfr_get_emin (); // save emin
  mpfr_set_emin (-1073);
  mpfr_init2 (y, 53);
  mpfr_set_d (y, x, MPFR_RNDN); // exact
  int inex = mpfr_exp (y, y, r); // r is the current rounding mode
  mpfr_subnormalize (y, inex, r);
  double ret = mpfr_get_d (y, r); // exact or overflow
  mpfr_clear (y);
  mpfr_set_emin (emin); // restore emin
  return ret;
}
```

Manipulating floating-point numbers with SageMath

SageMath is a nice way of playing interactively with MPFR (if efficiency is not an issue).

Create the "field" of floating-point numbers of 42 bits with rounding towards zero:

```
sage: Rz = RealField(42, rnd='RNDZ')
sage: a = Rz(pi)
sage: (a*a).hex()
'0x9.de9e64df1cp+0'
```

Same with towards $+\infty$:

```
sage: Ru = RealField(42, rnd='RNDU')
sage: a = Ru(pi)
sage: (a*a).hex()
'0x9.de9e64df28p+0'
```

Manipulating floating-point numbers with SageMath

`RR` is predefined as `RealField(53, rnd='RNDN')` or simply `RealField(53)`.

SageMath also has a domain for machine doubles:

```
sage: b = RDF(2^1024)
sage: b
+infinity
sage: a = RR(2^1024)
sage: a
1.79769313486232e308
```

(It is not possible to change the default exponent range for MPFR numbers.)

Toy floating-point numbers

```
sage: R2 = RealField(2)
sage: a = R2(1)
sage: for k in range(9):
         print (a)
         a = a.nextabove()
1.0
1.5
2.0
3.0
4.0
6.0
8.0
12.
16.
```

Useful methods

```
sage: a = RealField(17,rnd='RNDZ')(pi)
```

- `a.parent()`: the real field parent (gives the precision and rounding mode)

```
sage: a.parent()
Real Field with 17 bits of precision and rounding RNDZ
```

- `a.prec()`: precision of *a*

```
sage: a.prec()
17
```

- `a.ulp()`: unit in last place of *a* (depends on its precision)

```
sage: a.ulp()
0.00003051
sage: log(_)/log(2.)
-14.99
```

Useful methods

- `a.exact_rational()`: gives the rational (dyadic) number equal to a

```
sage: a.exact_rational()
102943/32768
```

- `a.nextabove()`: next number towards $+\infty$

```
sage: a.nextabove()
3.141
sage: a.nextabove().exact_rational()
3217/1024
```

- `a.nextbelow()`: next number towards $-\infty$

```
sage: a.nextbelow()
3.141
sage: a.nextbelow().exact_rational()
51471/16384
```

Useful methods

- `a.sign_mantissa_exponent()`: yields s, m, e such that $a = s \cdot m \cdot 2^e$

```
sage: a.sign_mantissa_exponent()
(1, 102943, -15)
```

- `a.str(b)`: outputs a in radix b

```
sage: a.str(2)
'11.001001000011111'
sage: a.str(16)
'3.243e'
sage: a.hex()
'0x3.243ep+0'
```

Checking theorems exhaustively for small precision

For example, check FastTwoSum is exact for rounding to nearest:

```
def FastTwoSum(a,b):
   s = a + b
   z = s - a
   t = b - z
   return s, t

def CheckFastTwoSum(a,b):
   s, t = FastTwoSum(a,b)
   u = a.exact_rational() + b.exact_rational()
   v = s.exact_rational() + t.exact_rational()
   if u != v:
      print ("bug for", a.hex(), b.hex())
      raise ValueError
```

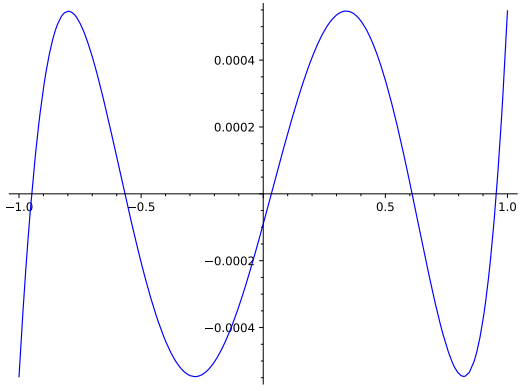
Checking theorems for small precisions

```
def CheckAll(p):
   R = RealField(p)
   for e in range(0,2*p):
      for A in range(2^(p-1),2^p):
         for B in range(2^(p-1),2^p):
            CheckFastTwoSum(R(A),R(B/2^e))

for p in [1..5]:
   CheckAll(p)
   print ("success for precision", p)
success for precision 1
success for precision 2
success for precision 3
success for precision 4
success for precision 5
```

Plot of functions

```
sage: p=1.00009+0.997309*x+0.498835*x^2+0.177345*x^3+0.0441555*x^4
sage: plot(exp(x)-p,(x,-1,1))
```




```
sage: plot(exp(x)-p,(x,-1,1)).save("err.pdf")
```

Computing worst cases of float-string conversion

Assume $m \cdot 2^e$ is near $M \cdot 10^E$ for $2^{52} \leq m < 2^{53}$ and $10^{16} \leq M < 10^{17}$. Then:

$$\frac{2^e}{10^E} \approx \frac{M}{m}$$

We can compute worst cases for the conversion from binary64 to string as follows:

- for a given binary exponent e, determine the possible decimal exponents E (there will be at most 2)
- for each E, compute the continued fraction expansion of $2^e/10^E$
- for all convergents M/m of $2^e/10^E$, check whether $10^{16} \leq M < 10^{17}$ and $2^{52} \leq m < 2^{53}$
- if this holds, output $m \cdot 2^e$ as a worst case

Computing worst cases of float-string conversion

Example for $e = 967$.

We have $2^{1019} \leq m \cdot 2^e < 2^{1020}$ for $2^{52} \leq m < 2^{53}$

Since $10^{E+16} \leq M \cdot 10^E < 10^{E+17}$, we should have:

$$2^{1019} \leq 10^{E+17} \qquad \text{and} \qquad 10^{E+16} \leq 2^{1020}$$

This yields $E \geq \log_{10}(2^{1019}) - 17$ and $E \leq \log_{10}(2^{1020}) - 16$.

```
sage: ceil(log(2^1019)/log(10))-17
290
sage: floor(log(2^1020)/log(10))-16
291
```

Computing worst cases of float-string conversion

```
sage: e=967; E=291
sage: l=continued_fraction(2^e/10^E)
sage: for r in l.convergents():
         if 10^16 <= r.numer() < 10^17 and 2^52 <= r.denom() < 2^53:
            print (r)
10503745438333595/8420509707638708
sage: n(8420509707638708*2^e, 130)
1.0503745438333595000000000000000026936e307
```

State of the art of decimal string conversion

The IEEE standard requires correctly rounded conversion up to 20 digits if binary64 is supported, 39 digits if binary128 is supported (section 5.12.2 of 754-2019).

For larger strings, a rounding error of less than 10^{-3} ulps is allowed.

In practice, current compiler/libraries (GCC/GNU libc, Intel, AMD, Clang/LLVM) seem to be correctly rounded.

For GNU libc, last bugs were fixed in 2013 (GNU libc 2.19), and in 2013 for GCC (version 4.9.0).

Computing minimax polynomial approximations with Sollya

Computing minimax polynomial approximations with Sollya

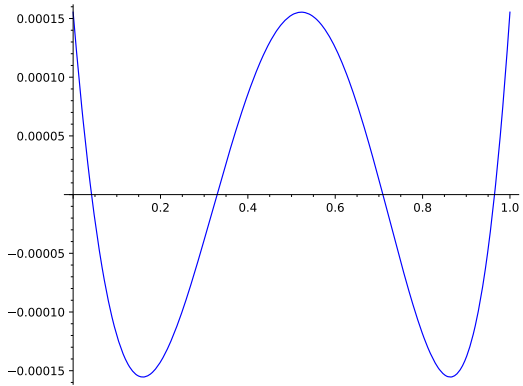
Sollya is a tool to compute (among other) minimax polynomial approximations, and to compute rigorous corresponding bounds.

We can give Sollya a range $[a, b]$, a degree (or a list of degrees), some constraints on the format of coefficients, one can fix some coefficients, ...

Two main commands: `fpminimax` and `infnorm` (here we will use `dirtyinfnorm` which has a simpler syntax).

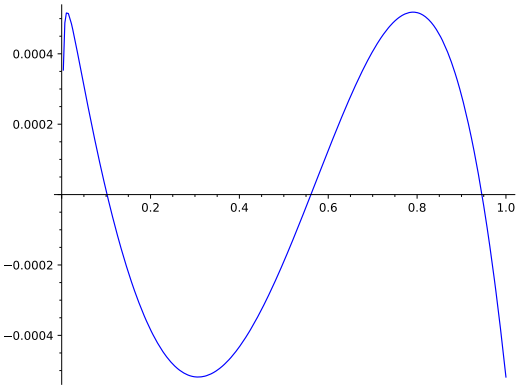
The fpminimax command

```
> fpminimax(sin(x), 3, [|24...|], [0,1], absolute);
-1.554026675876230001449584960937 5e-4 + x * (1.0044683218002319335937 5
+ x * (-1.945137232542037963867187 5e-2 + x * (-0.1435459703207015991210937 5)))
sage: plot(sin(x)-p,(x,0,1))
```



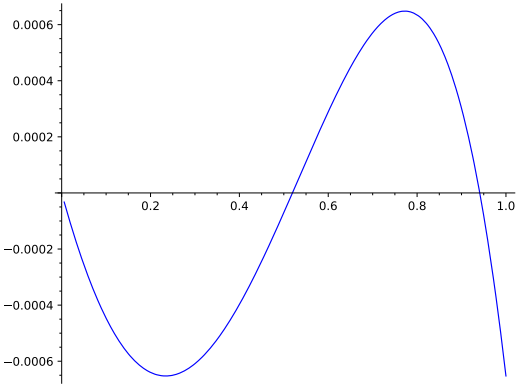
The fpminimax command

```
> fpminimax(sin(x), 3, [|24...|], [0.001,1], relative);
-1.230790530826197937130928039550781 25e-6 + x * (1.000720858573913574218 75
+ x * (-8.513902314007282257080078125e-3 + x * (-0.151170983910560607910156 25)))
sage: plot(p/sin(x)-1,(x,0,1))
```



The fpminimax command

```
> fpminimax(sin(x), [|1,2,3|], [|1,24,24|], [0,1], relative);
x * (1 + x * (-5.786848254501819610595703125e-3
+ x * (-0.153291270136833190917968 75)))
sage: plot(p/sin(x)-1,(x,0,1))
```



The fpminimax command

If I want only coefficients of odd degree:

```
> fpminimax(sin(x), [|1,3,5,7|], [|24...|], [0,1], relative);
x * (1 + x^2 * (-0.16666613519191741943359375
+ x^2 * (8.330207318067550659179687 5e-3
+ x^2 * (-1.9310935749672353267669677734375e-4))))
```

If I want the coefficient of degree 0 to have 40 bits, the coefficient of degree 1 to have 30 bits, that of degree 2 to have 20 bits, and that of degree 3 to have 10 bits:

```
> fpminimax(sin(x), 3, [|40,30,20,10|], [0,1], absolute);
-1.55122231870308269208180718123912811279296875e-4
+ x * (1.00446335785090923309326171875
+ x * (-1.9438236951828002929687 5e-2
+ x * (-0.1435546875)))
```

The fpminimax command

It is useful to print the coefficients in hexadecimal, to copy/paste them in your program:

```
> display = hexadecimal;
> fpminimax(sin(x), 3, [|40,30,20,10|], [0,1], absolute);
-0x1.45509d356p-13 + x * (0x1.012482b8p0
+ x * (-0x1.3e79ep-6 + x * (-0x1.26p-3)))
```

Unfortunately the exponents are then printed in hexadecimal too.

The `dirtyinfnorm` command

It gives you a bound on the absolute difference between a function and a (minimax) polynomial:

```
> p = fpminimax(sin(x), 3, [|24...|], [0,1], absolute);
> dirtyinfnorm(p-sin(x), [0,1]);
1.554083213741748186629051263927496225630607983710 65e-4
```

To get a bound for the relative error, simply give $p(x)/f(x) - 1$:

```
> p = fpminimax(sin(x), 3, [|24...|], [0.001,1], relative);
> dirtyinfnorm(p/sin(x)-1, [0,1]);
5.184305362996757754223272405809054168020027323564e-4
```

Forcing a coefficient

To force the degree-1 term to x:

```
> p = fpminimax(sin(x), [|2,3|], [|24...|], [0.001,1], relative, x);
> dirtyinfnorm(p/sin(x)-1, [0,1]);
6.5092225657772443300987653952771419092110783061518e-4
```

Another way is to force the precision to 1 bit:

```
> q = fpminimax(sin(x), [|1,2,3|], [|1,24...|], [0.001,1], relative);
> dirtyinfnorm(q/sin(x)-1, [0,1]);
6.5308834244078688012546845974512608870654650502189e-4
```

Back to the toy example from Module 4

```
> p = 1 - x^2/2 + x^4/24 - x^6/720;
> dirtyinfnorm(p-cos(x), [-0.125,0.125]);
1.47803318381913001047309176554952695882348569113 19e-12
```

This confirms the bound $2^{-39.2}$ we found for the mathematical error.

If we consider the actual coefficients c_2, c_4, c_6 we used:

```
> p = 1-x^2/2+0x1.5555555555555p-5*x^4-0x1.6c16c16c16c17p-10*x^6;
> dirtyinfnorm(p-cos(x), [-0.125,0.125]);
1.47803318438402084168124939836004385917908632341928e-12
```

The bound did not change significantly.

Bounding rounding errors with Gappa

Bounding rounding errors with Gappa

Gappa bounds the rounding error in a given format for a given rounding mode.

Remember our toy example for correct rounding of $\cos(x)$ in binary32 for $-0.125 \leq x \leq 0.125$, with internal computations in binary64:

```
double c2 = -0.5;                    // no error
double c4 = 0x1.5555555555555p-5;    // error < 2^-58.5
double c6 = -0x1.6c16c16c16c17p-10; // error < 2^-64.0
double x2 = x * x;
double t0 = c0 + x2 * c2;
double t4 = c4 + x2 * c6;
double x4 = x2 * x2;
double r = t0 + x4 * t4;
```

Bounding rounding errors with Gappa

Use Gappa with rounding towards zero (`zr`). Other modes: `ne` for RN, `up` for RU, `dn` for RD. Upper case for exact values, lower case for the corresponding approximations.

```
@rnd = float<ieee_64,zr>;
C2 = -1/2;
c2 = -0.5;
{ x in [-0.125,0.125] -> |c2-C2| in ? }
Results: |c2 - C2| in [0, 0]
```

This is fine so far.

```
C4 = 1/24;
c4 = 0x1.5555555555555p-5;
{ x in [-0.125,0.125] -> |c4-C4| in ? }
Results: |c4 - C4| in [..., 2^(-58.5737)]
```

This agrees with the bound $2^{-58.5}$ we had.

Bounding rounding errors with Gappa

```
C6 = -1/720;
c6 = -0x1.6c16c16c16c17p-10;
{ x in [-0.125,0.125] -> |c6-C6| in ? }
Results: |c6 - C6| in [..., 2^(-64)]
```

This agrees with the bound 2^{-64} we had.

To say $x_2 = \circ(x \cdot x)$, we use `x2 rnd= x * x` in Gappa:

```
X2 = x * x;
x2 rnd= x * x;
{ x in [-0.125,0.125] -> |x2-X2| in ? }
Results: |x2 - X2| in [0, 2^(-59)]
```

This agrees with the bound 2^{-59} we had.

Bounding rounding errors with Gappa

The next instruction is `t0 = c0 + x2 * c2`, which we decompose into `u0 = x2 * c2` and `t0 = c0 + u0`:

```
U0 = X2 * C2;
u0 rnd= x2 * c2;
{ x in [-0.125,0.125] -> |u0-U0| in ? }
Results: |u0 - U0| in [0, 2^(-59)]
```

We found only $2^{-58.9}$. Gappa finds 2^{-60} for RN, RU, RD.

With $C_0 = c_0 = 1$:

```
T0 = C0 + U0;
t0 rnd= c0 + u0;
{ x in [-0.125,0.125] -> |t0-T0| in ? }
Results: |t0 - T0| in [0, 2^(-53)]
```

We found $2^{-52.9}$. Gappa finds $2^{-52.9888}$ for RD and RU, and $2^{-53.977}$ for RN.

Bounding rounding errors with Gappa

The next instruction is `t4 = c4 + x2 * c6`, which we decompose into `u4 = x2 * c6` and `t4 = c4 + u4`:

```
U4 = X2 * C6;
u4 rnd= x2 * c6;
{ x in [-0.125,0.125] -> |u4-U4| in ? }
Results: |u4 - U4| in [0, 2^(-67.2251)]
```

We found $2^{-66.9}$. Gappa finds $2^{-67.6781}$ for RD, $2^{-67.8552}$ for RN, 2^{-68} for RU.

```
T4 = C4 + U4;
t4 rnd= c4 + u4;
{ x in [-0.125,0.125] -> |t4-T4| in ? }
Results: |t4 - T4| in [..., 2^(-56.582)]
```

We found $2^{-56.5}$. Gappa finds $2^{-56.5815}$ for RD, $2^{-57.2576}$ for RN, $2^{-57.5727}$ for RU.

Bounding rounding errors with Gappa

Next we approximate $x_4 = \circ(x_2 \cdot x_2)$:

```
X4 = X2 * X2;
x4 rnd= x2 * x2;
{ x in [-0.125,0.125] -> |x4-X4| in ? }
Results: |x4 - X4| in [0, 2^(-63.415)]
```

Gappa finds the same bound for RD and RU, and $2^{-64.415}$ for RN.

The last instruction is `r = t0 + x4 * t4`, which we decompose into `v4 = x4 * t4` and `r = t0 + v4`:

```
V4 = X4 * T4;
v4 rnd= x4 * t4;
{ x in [-0.125,0.125] -> |v4-V4| in ? }
Results: |v4 - V4| in [0, 2^(-66.8837)]
```

We found $2^{-66.8}$. Gappa finds $2^{-66.8834}$ for RD, $2^{-67.7756}$ for RN, $2^{-67.1233}$ for RU.

Bounding rounding errors with Gappa

With option `-Echange-threshold=0`, Gappa is able to prove that $r \leq 1$, which is not trivial.

```
R = T0 + V4;
r rnd= t0 + v4;
{ x in [-0.125,0.125] -> r <= 1 /\ r in ? }
Results: r in [127b-7 {0.992188, 2^(-0.0113153)}, 1]
```

```
R = T0 + V4;
r rnd= t0 + v4;
{ x in [-0.125,0.125] -> t0+v4 <= 1 /\ |r-R| in ? }
Results: |r - R| in [0, 2^(-51.9999)]
```

We found $2^{-51.9}$. Gappa finds $2^{-51.9943}$ for RD, $2^{-51.415}$ for RU, $2^{-52.9887}$ for RN.

Bounding rounding errors with Gappa

Instead of asking Gappa to find a bound for a variable with `x in ?`, one can ask him to prove a given bound with say `x <= 257b-60`, meaning $x \leq 257 \cdot 2^{-60}$.

```
{ x in [-0.125,0.125] -> |r-R| <= 129b-60 }
$ gappa cos.g
Error: some properties were not satisfied:
  BND(|r - R|), best: [1.1189e-16, 3.33074e-16]

{ x in [-0.125,0.125] -> |r-R| <= 257b-60 }
$ gappa cos.g
$ echo $?
0
```

Note: $257 \cdot 2^{-60} \approx 2^{-51.994}$.

Bounding rounding errors with Gappa

Gappa also enables one to perform dichotomy on x:

```
{ x in [-0.125,0.125] -> |r-R| in ? }
$ x;
```

We can also split x into a given number of subranges:

```
{ x in [-0.125,0.125] -> |r-R| in ? }
$ x in 10;
```

or split x at given points:

```
{ x in [-0.125,0.125] -> |r-R| in ? }
$ x in (-0.0625,0,0.0625);
```

Bounding rounding errors with Gappa

We have seen in this toy example that:

• Gappa is a more systematic way of analyzing the rounding errors, avoiding mistakes from manual analysis (I indeed did some mistakes that I found thanks to Gappa)

• Gappa finds better error bounds, and enables one to distinguish different rounding modes

• for more complex examples, you have to give hints to Gappa (out of the scope of this training)

Computing hard-to-round cases with BaCSeL

Computing hard-to-round cases with BaCSeL

BaCSeL is a tool to compute hard-to-round cases.

Given a mathematical function $f(x)$, a range $[x_0, x_1]$, a precision p, an integer m, it finds all cases with at least $m - 1$ identical bits after the round bit.

BaCSeL implement the SLZ algorithm, which requires the knowledge of the first derivatives of $f(x)$.

Most usual functions are known to BaCSeL, and it is documented how to add a new function (out of the scope of this training).

A simple example

Compute hard-to-round cases of $\exp(x)$ in binary64 with at least 43 identical bits after the round bit in $[0.250, 0.251]$:

```
$ make DEFSAL="-DEXP -DBASIS=2 -DAUTOMATIC"
$ time ./bacsel -rnd_mode all -prec 128 -n 53 -nn 53 -m 44 -t 20
-t0 4503599627370496 -t1 4521614025879978 -d 2 -alpha 2 -e_in -1 -nthreads 64
*** x=0x4.01a1c5e78fd98p-4, distance is: 3.485390759e-16
*** x=0x4.02ca83221fd48p-4, distance is: 5.237832481e-14
real    0m21.915s. user    23m5.721s
```

```
sage: x = RR("0x4.01a1c5e78fd98p-4",16)
sage: n(exp(x.exact_rational()), 150).str(2)
'1.01001000110101110110110001100000000000010011000010000111
11111111111111111111111111111111111111111111001110'
```

Parameters of BaCSeL

- `-rnd_mode all`: compute hard-to-round cases for all rounding modes (can be `directed` or `nearest`)
- `-prec 128`: use 128 bits of precision internally (should be at least twice the target precision)
- `-n 53`: precision of the input number x
- `-nn 53`: precision of the output number y
- `-m 44`: output hard-to-round cases with $m-1$ bits after the round bit
- `-e_in -1`: search in binade $[2^{e-1}, 2^e]$
- `-t0 4503599627370496`: start at $x_0 = t_0 \cdot 2^{e-n}$
- `-t1 4521614025879978`: scan $[x_0, x_1)$ with $x_1 = t_1 \cdot 2^{e-n}$
- `-t 20`: scan intervals of length 2^t (automatically adjusted with `-DAUTOMATIC`)
- `-d 2 -alpha 2`: use degree-2 approximation and SLZ parameter α
- `-nthreads 64`: use 64 threads

Monotonic functions

BaCSeL assumes that $f(x)$ lies in the same binade for $x_0 \leq x < x_1$:

```
$ make DEFSAL="-DEXP -DBASIS=2 -DAUTOMATIC"
$ ./bacsel -rnd_mode all -prec 128 -n 53 -nn 53 -m 44 -t 20
-t0 4503599627370496 -t1 9007199254740992 -d 2 -alpha 2 -e_in 0
Error, f at t0 and near t1 have different exponents
```

Indeed, $\exp(0.5) \approx 1.65$ and $\exp(1.0) \approx 2.72$.

We can tell BaCSeL the function is monotonic. Then it automatically splits the interval in ranges $[x_0, x_1)$ where $f(x)$ stays in the same binade:

```
$ make DEFSAL="-DEXP -DBASIS=2 -DAUTOMATIC -DMONOTONIC"
$ ./bacsel -rnd_mode all -prec 128 -n 53 -nn 53 -m 44 -t 20
-t0 4503599627370496 -t1 9007199254740992 -d 2 -alpha 2 -e_in 0
*** x=0x8.01b338ffc3f78p-4, distance is: 1.294236867e-14
*** x=0x8.01b80b18fda6p-4, distance is: 4.424723552e-14
```

Output in the subnormal range

When the output is subnormal, we adapt the value of `-nn`. For example, for $x_0 \leq x < x_1$ with $x_0 =$ `-0x1.6a2b5deb4db9fp+9` and $x_1 =$ `-0x1.69d2a4df51d1p+9`, $\exp(x)$ lies in $[2^{29} \cdot 2^{-1074}, 2^{30} \cdot 2^{-1074})$, thus we can consider $\exp(x)$ has precision 30 bits.

```
$ make DEFSAL="-DEXP -DBASIS=2 -DAUTOMATIC"
$ ./bacsel -rnd_mode all -prec 128 -n 53 -nn 30 -m 42 -t 20
-t0 -6371351496809375 -t1 -6365254509731088 -d 2 -alpha 2 -e_in 10
*** x=-0x2.d4448d9ac443ep+8, distance is: 2.270579006e-13
*** x=-0x2.d40c9557b6dc4p+8, distance is: 1.583688278e-13
*** x=-0x2.d3fe43fb319fep+8, distance is: 4.887325005e-14
*** x=-0x2.d3c7902c1e8c6p+8, distance is: 1.468996004e-13
```

Input in the subnormal range

When the input is subnormal, we adapt the value of `-n`. For example, for $683565276 \cdot 2^{-1074} \leq x < 2^{30} \cdot 2^{-1074}$, we have $2^{31} \cdot 2^{-1074} \leq \mathrm{sinpi}(x) < 2^{32} \cdot 2^{-1074}$:

```
$ make DEFSAL="-DSINPI -DBASIS=2 -DAUTOMATIC"
$ ./bacsel -rnd_mode all -prec 128 -n 30 -nn 32 -m 30 -t 20
-t0 683565276 -t1 1073741824 -d 2 -alpha 2 -e_in -1044
*** x=0xb.1dddf18p-1048, distance is: 8.209085322e-10
*** x=0xc.17bc3d8p-1048, distance is: 4.474494938e-10
```

Which degree to choose?

Rule of thumb:

- if f'' is of the same order as f, then use degree 2 with $\alpha = 2$: `-d 2 -alpha 2`
- if $|f''/f|$ is much larger than 1, then use degree 1 with $\alpha = 1$: `-d 1 -alpha 1`
- if $|f''/f|$ is huge, use degree 0: `-d 0 -alpha 1`. Then each value is checked independently with GNU MPFR.

Which degree to choose?

Option -v enables one to get an idea of the average interval size (with -DAUTOMATIC):

```
$ make DEFSAL="-DEXP -DBASIS=2 -DAUTOMATIC"
$ time ./bacsel -rnd_mode all -prec 128 -n 53 -nn 53 -m 44 -t 20
-t0 4503599627370496 -t1 4521614025879978 -d 2 -alpha 2 -e_in -1
-nthreads 48 -v
Final size of subintervals (log[2](T)): 2.098800E+01.
real 0m46.319s, user 36m38.186s

$ time ./bacsel -rnd_mode all -prec 128 -n 53 -nn 53 -m 44 -t 20
-t0 4503599627370496 -t1 4521614025879978 -d 1 -alpha 1 -e_in -1
-nthreads 48 -v
Final size of subintervals (log[2](T)): 1.751651E+01.
real 3m33.642s, user 169m24.830s
```

Status of the Table Maker's Dilemma

• binary32: fully solved (including bivariate functions). CORE-MATH contains 589,146 hard-to-round cases.

• binary64: fully solved for univariate functions ($\sin, \cos, \tan$ recently completed with Tue Ly from Google), still out of reach for bivariate functions. CORE-MATH contains 12,935,906 hard-to-round cases.

• double extended (`long double`): doable for some univariate functions (cbrt), would need state-level resources for other univariate functions, out of reach for bivariate functions. CORE-MATH contains 3,936,611 hard-to-round cases.

• binary128: still out of reach. CORE-MATH contains 1,036,802 hard-to-round cases.

Most hard-to-round cases computed with BaCSeL, some with ad-hoc algorithms.

Exercises

Exercise 1: using GNU MPFR, write a program that finds the value of x such that the AMD math library gives the largest ulp error for $\mathrm{acos}(x)$ in the binary32 format, for rounding to nearest.

Exercise 2: to emulate subnormals in binary32 with MPFR, which values should you give to `mpfr_set_emin` and `mpfr_set_emax`?

Exercise 3: write a program in SageMath to check Algorithm TwoSum is exact in precision $1 \leq p \leq 5$ for numbers of exponent difference less than $2p$, (for rounding to nearest).

Exercises

Exercise 4: if you want to approximate $\sin(x)$ on $[0.5, 1]$ with relative error less than 2^{-70} and double coefficients, what is the minimal degree to use? Compute the corresponding minimax polynomial and its maximal relative error.

Exercise 5: using Sollya, determine a minimax polynomial for $\sin(x)$ on $[0.5, 1]$, with coefficients of degree $1, 3, 5$ only, all in double precision. Still using Sollya, determine the corresponding maximal absolute error. Then use Gappa to bound the rounding error when approximating this polynomial using Horner's rule (for rounding to nearest), and deduce a bound for the total error.

Takeover message

- we now have nice software tools to play with floating-point numbers
- these tools are very useful (a) to implement mathematical functions
- (b) to determine inputs with large errors
- (c) or to check correct rounding (for univariate binary32 functions)

References

GNU MPFR, `mpfr.org`

SageMath, `sagemath.org`

Computational Mathematics with SageMath,
`https://www.sagemath.org/sagebook/english.html`

Sollya, `https://www.sollya.org/`

Gappa, `https://gappa.gitlabpages.inria.fr/`

BaCSeL, `https://gitlab.inria.fr/zimmerma/bacsel`

SLZ algorithm, Searching Worst Cases of a One-Variable Function Using Lattice Reduction, Stehlé, Lefèvre, Zimmermann, IEEE Transactions on Computers, 2005.

Questions, comments ?

Paul.Zimmermann@inria.fr