CS 598sm

Probabilistic &

Approximate

Computing

http://misailo.web.engr.Illinois.edu/courses/cs598

NUMBER REPRESENTATIONS

Numb3rs

Integers vs Machine Integers

- Precision
- Signed/unsigned

Reals vs Rationals vs Floats

- Precision
- Special values

Complex numbers etc.

Numb3rs

Dynamic range: the range of representable numbers.

Important to consider: number of values that can be represented within the dynamic range

Precision / resolution: the distance between two represented numbers

Warmup: First Approximation (everyone often neglects)

Dynamic range: [MIN_INT, MAX_INT]

Resolution: one

Trades off the (intuitive) rules of mathematics for finite representation

Case #1:

- Mathematical integers: Closure property says that *a+b* and *ab* are positive integers whenever *a* and *b* are positive integers
- Machine integers: How much is MAX_INT + 2 ?
  (cf. modular arithmetic)

Case #2:

- Mathematical integers: Division by zero is undefined
- Machine integers: the compiler may interpret 'undefined' as performing any action (e.g., simply return 0; or raise exception).

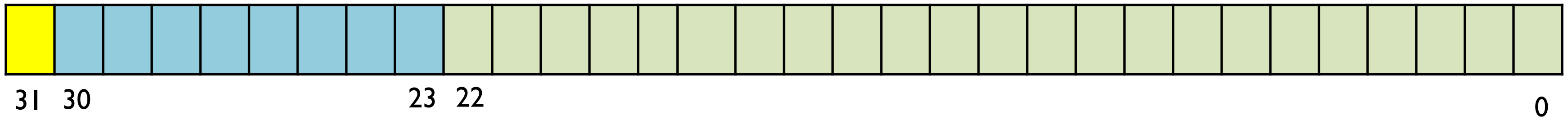
Floating Point Numbers

Trades off resolution for wider dynamic range

Standardized by IEEE 754

Example 32-bit (our old good friend ‘float’ in C):

Sign Exponent (8 bit) Mantissa(23 bits)



31 30 23 22 0

$$(-1)^{\text{sign}} \cdot 2^{\text{exponent}-127} \cdot \left(1 + \sum_{i=0}^{22} \text{mantissa}_{22-i} \cdot 2^{-i-1}\right)$$

*note, when exponent=0, we do special treatment, i.e. subnormal values

- **Dynamic range:** [-3.4028 * 10^38, -3.4028 * 10^38] (approx.)
- **Resolution:** 6 to 9 significant digits
- Min positive value 1.18*10^-38; min subnormal value 1.4*10^-45

Floating Point Numbers

Want more precision?

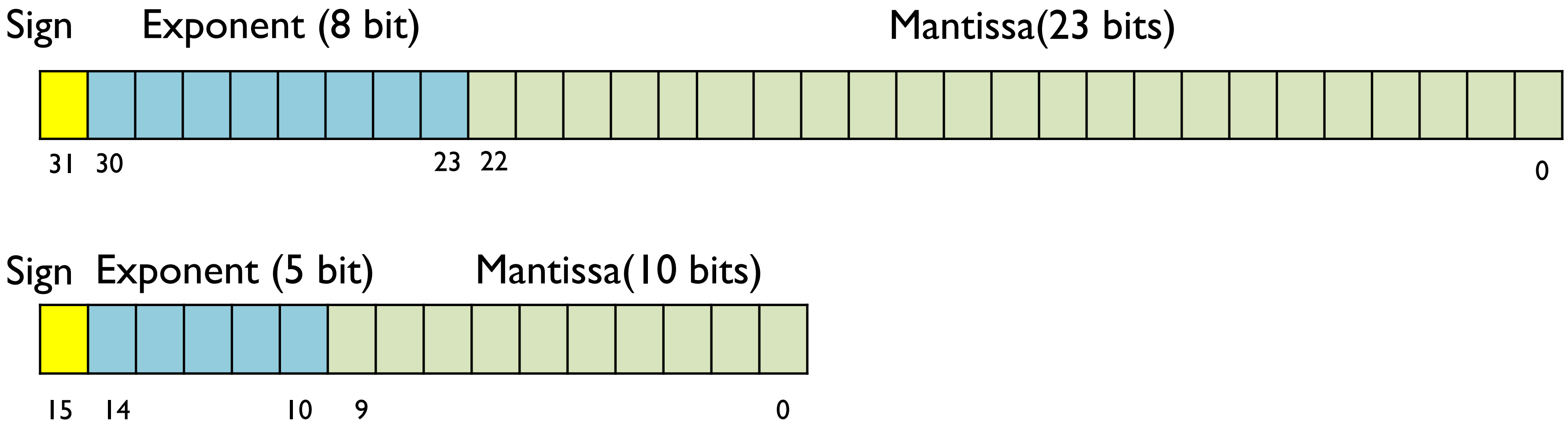
Double precision floating point (C ‘double’)

- 64 bits total: sign + exponent (11 bits) + mantissa (52 bits)
- Dynamic range: [10^-308, 10^308] (approx.)
- Resolution: between 2^n and 2^(n+1) it is 2^(n-52).

Extended precision (also part of IEEE 754; C ‘long double’):

- 80 bits total: sign + exponent (15 bits) + mantissa (63 bits)
- E.g., needed for exponentiation of doubles.
- Internally, x86 FPU computes on data in this format.

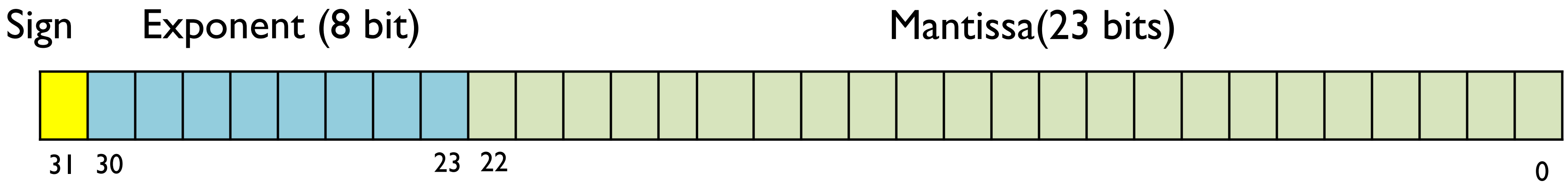
Floating Point Numbers

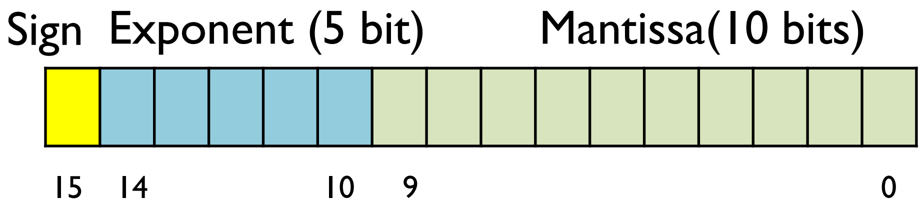


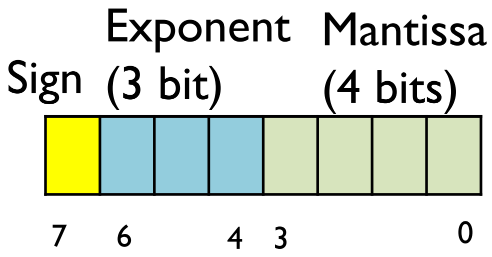
Half-float numbers:

- **Dynamic range:** [-65504,+65504]
- **Precision**: up to 0.00000006 (approx.)

Floating Point Numbers



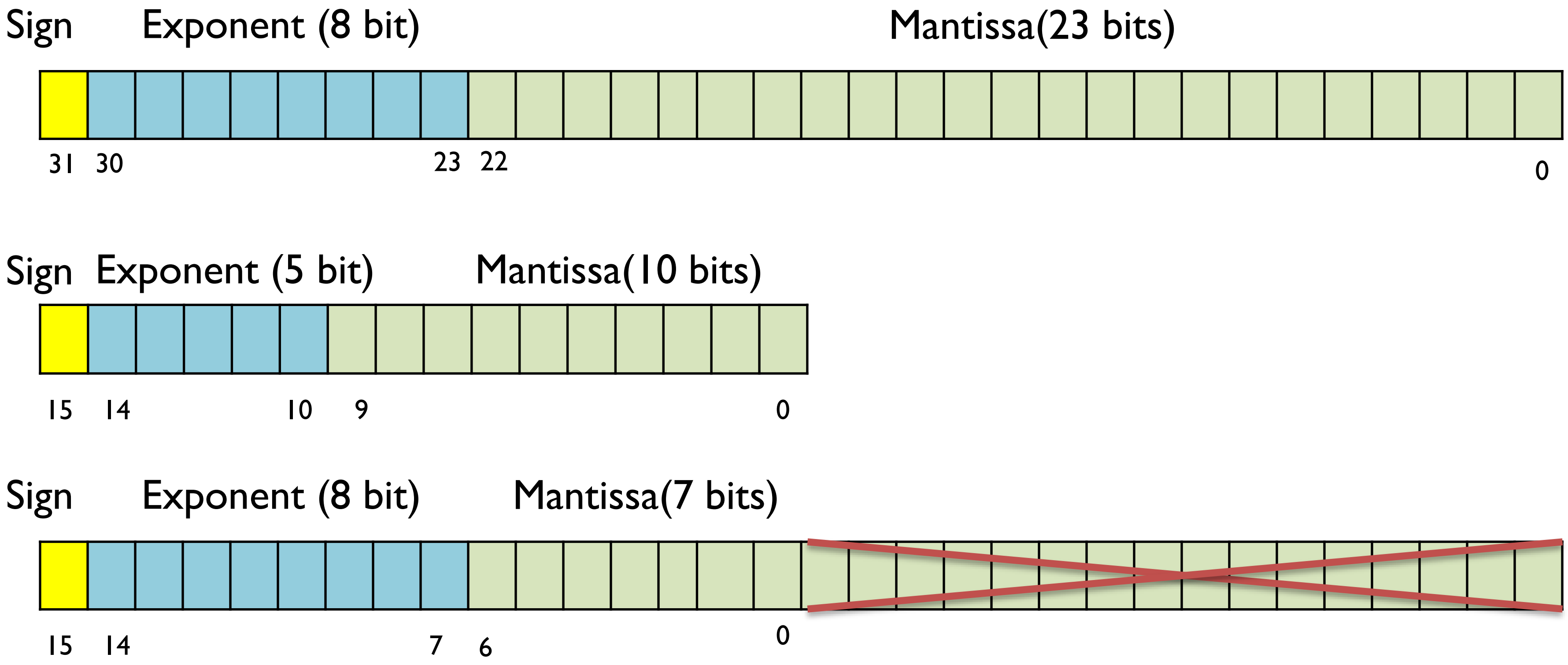




8bit-float numbers:

- **Dynamic range:** [-15.5,-0.25] U {0} U [0.25, 15.5]
- **Precision**: up to 0.1 (approx.)

Floating Point Numbers



Sign Exponent (8 bit) Mantissa(23 bits)

31 30 23 22 0

Sign Exponent (5 bit) Mantissa(10 bits)

15 14 10 9 0

Sign Exponent (8 bit) Mantissa(7 bits)

15 14 7 6 0

BFloat16 (Brain float) numbers, not a part of IEEE standard:

- **Dynamic range:** [-3.4 * 10^38, 3.4 * 10^38] (approx.)
- **Precision**: between 2 and 3 decimal digits

Easy conversion to/from FP32, reduced memory size

What if we need something different?

Simple fixed-point numbers:

- Integer scaled by a unit factor (common: binary or decimal)
- If we e.g. use 2 decimal digits as a scaling factor, we can interpret 673 as 6.73. We can similarly use scaling by powers of 2.
- When implemented well better control of rounding over floating point representation

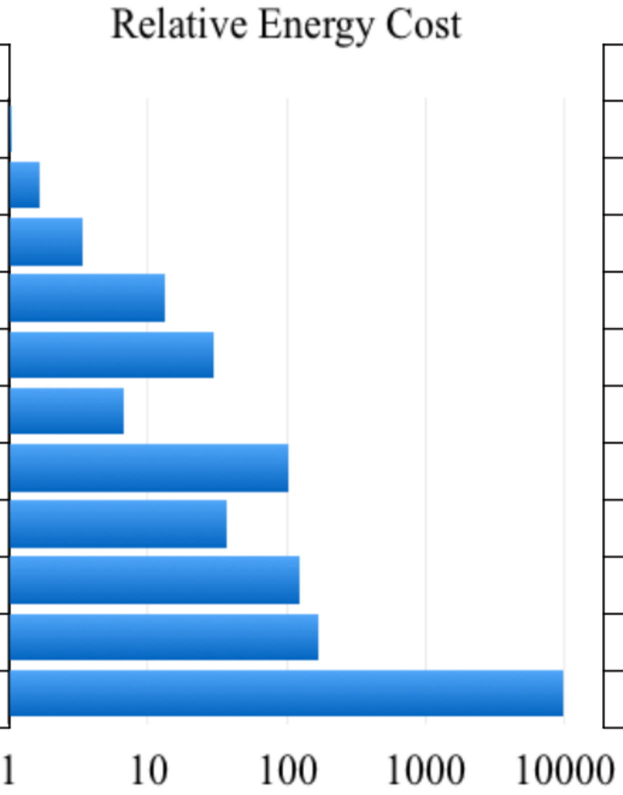
Practical concerns:

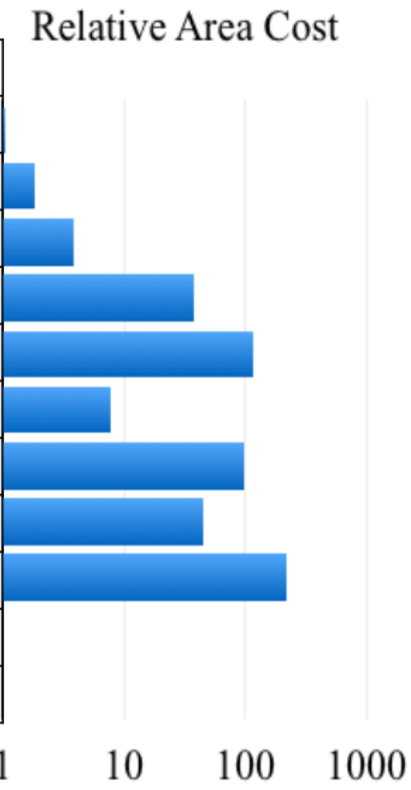
- In arithmetic operations fixed-point operations should be with the same scaling factor
- Beware of overflows (just as with integers)
- Division of fixed points is somewhat trickier
- Integer arithmetic spends less energy than FPU, which may be of relevance for low-end embeeded hardware.

System Impact of Numerical Operations

Cost of Operations

Operation:	Energy (pJ)	Area (μm^2)
8b Add	0.03	36
16b Add	0.05	67
32b Add	0.1	137
16b FP Add	0.4	1360
32b FP Add	0.9	4184
8b Mult	0.2	282
32b Mult	3.1	3495
16b FP Mult	1.1	1640
32b FP Mult	3.7	7700
32b SRAM Read (8KB)	5	N/A
32b DRAM Read	640	N/A





Energy numbers are from Mark Horowitz "Computing's Energy Problem (and what we can do about it)", ISSCC 2014
Area numbers are from synthesized result using Design Compiler under TSMC 45nm tech node. FP units used DesignWare Library.

Slide from William Dally's 2015 NIPS Tutorial

Numb3rs: it’s not so easy

Various tricky points when using floating point:

- Overflows
- Underflows
- Infinities
- NaN (not a number)
- No associativity (a+b)+c != a + (b+c)
- Catastrophic cancellation
- …

Rounding Error

Difference between results obtained between the exact solution (using the mathematical representation) and the finite-space representation of numbers

```
>>> 0.1
0.1000000000000000055511151231257827021181583404541015625
```

Machine epsilon: measure of roundoff error level

Other Tricky Points

NaN

Catastrophic cancellation

Infinities

…

What if we need something different?

https://docs.python.org/3/tutorial/floatingpoint.html

```
>>> from decimal import Decimal
>>> from fractions import Fraction

>>> Fraction.from_float(0.1) Fraction(3602879701896397, 36028797018963968)

>>> (0.1).as_integer_ratio() (3602879701896397, 36028797018963968)

>>> Decimal.from_float(0.1)
Decimal('0.1000000000000000055511151231257827021181583404541015625')

>>> (Decimal.from_float(0.1), '.17') '0.10000000000000001'
```

See also https://docs.oracle.com/javase/8/docs/api/java/math/BigDecimal.html

NUMERICAL APPROXIMATIONS

Common Error Metric

For idealized computation P (running on idealized input x) and approximate computation P' (running on the same input):

$$Err = \max_{x} |P(x) - P'(x)|$$

Algorithmic Approximation

How to compute sin(x) ?

Taylor Series (1715)

$$f(a) + \frac{f'(a)}{1!}(x-a)$$

$$+ \frac{f''(a)}{2!}(x-a)^2 + \frac{f'''(a)}{3!}(x-a)^3$$

$$+ \cdots,$$

Algorithmic Approximation

$$sin(x) \approx x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!}$$

What is the approximation error?

Algorithmic Approximation

$$\boldsymbol{sin(x) \approx x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!}}$$

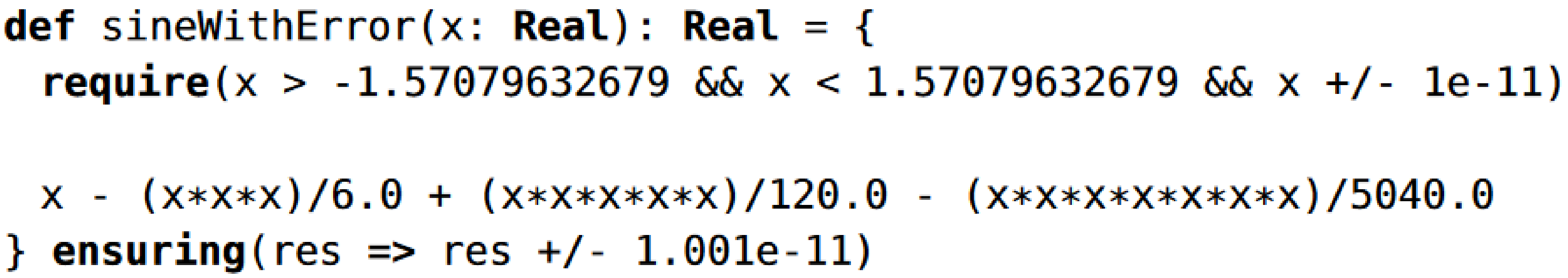
$$\boldsymbol{err < \frac{|x^9|}{9!}}$$

Algorithmic Approximation

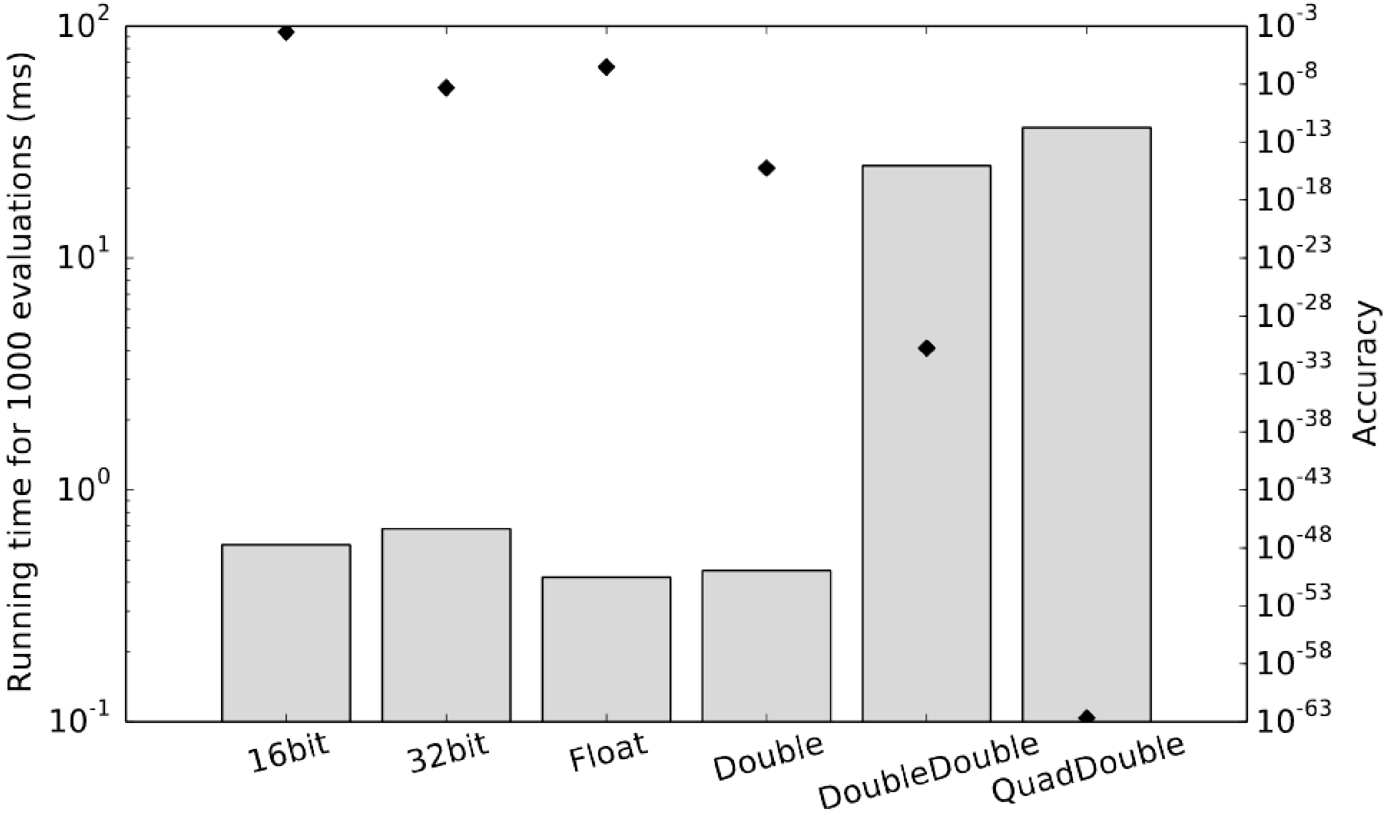
$$\boldsymbol{sin(x) \approx x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!}}$$

$$\boldsymbol{err < \frac{|x^9|}{9!}}$$

Where's the catch?



```
def sineWithError(x: Real): Real = {
  require(x > -1.57079632679 && x < 1.57079632679 && x +/- 1e-11)

  x - (x*x*x)/6.0 + (x*x*x*x*x)/120.0 - (x*x*x*x*x*x*x)/5040.0
} ensuring(res => res +/- 1.001e-11)
```



Towards a Compiler for Reals (TOPLAS 2017)

Other options

Orthogonal-basis polynomials: e.g., Chebyshev polynomials can approximate the function to the desired precision on the entire interval

Rational functions: functions that can be written as the ratio of two polynomials

Splines: piecewise functions, where each piece is a polynomial

Try out: https://www.chebfun.org/

What hides behind?

```
double x, y;
     …
y = sin(x);
```

Real Implementation

https://sourceware.org/git/?p=glibc.git;a=blob;f=sysdeps/ieee754/dbl-64/s_sin.c;hb=HEAD#l281

```
/*******************************************************************/
/* An ultimate sin routine. Given an IEEE double machine number x */
/* it computes the correctly rounded (to nearest) value of sin(x) */
/*******************************************************************/
#ifndef IN_SINCOS
double SECTION __sin (double x)
{
  double t, a, da; mynumber u; int4 k, m, n; ouble retval = 0;
  SET_RESTORE_ROUND_53BIT (FE_TONEAREST);

  u.x = x;
  m = u.i[HIGH_HALF];
  k = 0x7fffffff & m;     /* no sign           */
  if (k < 0x3e500000) {   /* if x->0 =>sin(x)=x */
      math_check_force_underflow (x);
      retval = x;
  }
/*------------------- 2^-26<|x|< 0.855469------------- */
  else if (k < 0x3feb6000) {
      /* Max ULP is 0.548.  */
      retval = do_sin (x, 0);
  } /*   else  if (k < 0x3feb6000)    */

/*---------------- 0.855469  <|x|<2.426265  -----------*/
  else if (k < 0x400368fd) {
      t = hp0 - fabs (x);
      /* Max ULP is 0.51.  */
      retval = copysign (do_cos (t, hp1), x);
  } /*   else  if (k < 0x400368fd)    */
```

```
/*--------------- 2.426265<|x|< 105414350 ---------*/
  else if (k < 0x419921FB) {
      n = reduce_sincos (x, &a, &da);
      retval = do_sincos (a, da, n);
  } /*   else  if (k <  0x419921FB )    */

/* ------ 105414350 <|x| <2^1024 ------------------*/
  else if (k < 0x7ff00000) {
      n = __branred (x, &a, &da);
      retval = do_sincos (a, da, n);
  }
/*------------ |x| > 2^1024 ----------------------*/
  else {
      if (k == 0x7ff00000 && u.i[LOW_HALF] == 0)
             __set_errno (EDOM);
      retval = x / x;
  }

  return retval;
}
```

Real Implementation (More!)

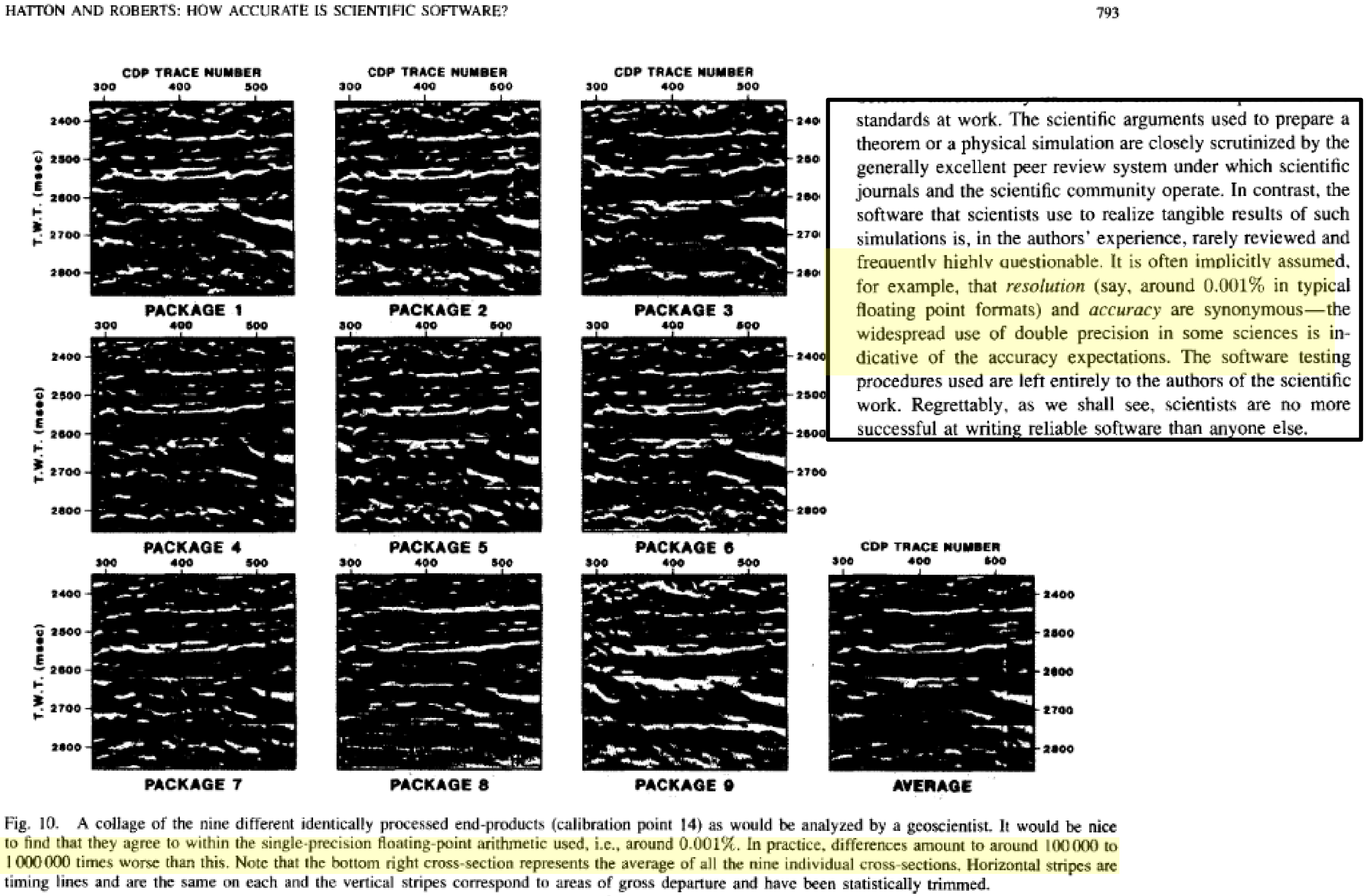
```
/* Given a number partitioned into X and DX, this function computes the sine of
   the number by combining the sin and cos of X (as computed by a variation of
   the Taylor series) with the values looked up from the sin/cos table to get
   the result.  */
static __always_inline double do_sin (double x, double dx)  {
  double xold = x;
  /* Max ULP is 0.501 if |x| < 0.126, otherwise ULP is 0.518.  */
  if (fabs (x) < 0.126)  return TAYLOR_SIN (x * x, x, dx);

  mynumber u;
  if (x <= 0) dx = -dx;
  u.x = big + fabs (x);
  x = fabs (x) - (u.x - big);

  double xx, s, sn, ssn, c, cs, ccs, cor;
  xx = x * x;
  s = x + (dx + x * xx * (sn3 + xx * sn5));
  c = x * dx + xx * (cs2 + xx * (cs4 + xx * cs6));
  SINCOS_TABLE_LOOKUP (u, sn, ssn, cs, ccs);
  cor = (ssn + s * ccs - sn * c) + cs * s;
  return copysign (sn + cor, xold);
}
```

…and this is not all!

Often, what we consider ‘exact’ is approximate to start with.

What matter is accuracy: the level of approximation and the ‘guarantees’ on the output quality

Fun Facts: End Results Can Differ



HATTON AND ROBERTS: HOW ACCURATE IS SCIENTIFIC SOFTWARE? 793

standards at work. The scientific arguments used to prepare a theorem or a physical simulation are closely scrutinized by the generally excellent peer review system under which scientific journals and the scientific community operate. In contrast, the software that scientists use to realize tangible results of such simulations is, in the authors' experience, rarely reviewed and frequently highly questionable. It is often implicitly assumed, for example, that *resolution* (say, around 0.001% in typical floating point formats) and *accuracy* are synonymous—the widespread use of double precision in some sciences is indicative of the accuracy expectations. The software testing procedures used are left entirely to the authors of the scientific work. Regrettably, as we shall see, scientists are no more successful at writing reliable software than anyone else.

Fig. 10. A collage of the nine different identically processed end-products (calibration point 14) as would be analyzed by a geoscientist. It would be nice to find that they agree to within the single-precision floating-point arithmetic used, i.e., around 0.001%. In practice, differences amount to around 100 000 to 1 000 000 times worse than this. Note that the bottom right cross-section represents the average of all the nine individual cross-sections. Horizontal stripes are timing lines and are the same on each and the vertical stripes correspond to areas of gross departure and have been statistically trimmed.

Sensitivity

So far we talked about the error that ***emerges inside*** the computation.

How does that error ***propagate through*** the subsequent computation?

Sensitivity

If the input x changes by δ,
by how much does the output of f(x) change?

$F_1(x) \;=\; x + 1$ $\qquad F_1(x + \delta) =$

$F_2(x) \;=\; x^2 \;+\; 1$ $\qquad F_2(x + \delta) =$

$F_3(x) \;=\; e^x$ $\qquad F_3(x + \delta) =$

Lipschitz Continuity

Sets a linear bound on error propagation:

$$\forall x_1, x_2 \;.\; |f(x_1) - f(x_2)| \leq K \cdot |x_1 - x_2|$$

Locally Lipschitz continuous in neighborhood U of x:

$$\forall x_1, \forall x_2 \in U(x_1) \;.\; |f(x_1) - f(x_2)| \leq K \cdot |x_1 - x_2|$$

Another Thought Experiment

Consider the function f(x) and its approximation f'(x).

Let x be the original input and x' be the input with some noise, bounded by a constant ε.

We also know that the Lipschitz constant of f and f' is equal to K, and the error of f' is bounded by δ.

What is the upper bound on the total error between **f(x) and f'(x') ?**

Let x be the original input and x' be the input with some noise, bounded by a constant ε.

We also know that the Lipschitz constant of f and f' is equal to K, and the error of f' is bounded by δ.

What is the upper bound on the total error between **f(x) and f'(x') ?**

We know:

- $\forall x, x' \,.\, |x - x'| \leq \varepsilon$
- $\forall x, x' \,.\, |f(x) - f(x')| \leq K \cdot |x - x'|$
- $\forall x \,.\, |f(x) - f'(x)| \leq \delta$

Question: $\forall x, x' \,.\, |f(x) - f'(x')| \leq ??$

Let x be the original input and x' be the input with some noise, bounded by a constant ε.

We also know that the Lipschitz constant of f and f' is equal to K, and the error of f' is bounded by δ.

What is the upper bound on the total error between **f(x) and f'(x') ?**

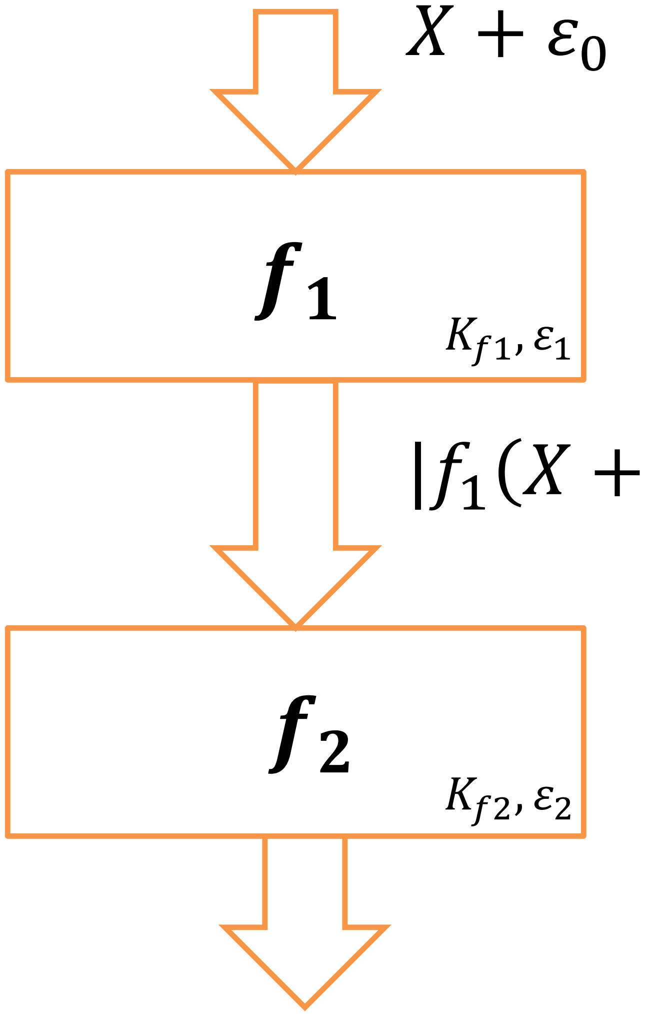
We know:

- $\forall x, x' . |x - x'| \leq \varepsilon$
- $\forall x, x' . |f(x) - f(x')| \leq K \cdot |x - x'|$
- $\forall x . |f(x) - f'(x)| \leq \delta$

Question: $\forall x, x' . |f(x) - f'(x')| \leq ??$

- $\forall x, x' . |f(x) - f'(x')|$
- $= |f(x) - f(x') + f(x') - f'(x')|$ Arithmetic, assuming no exceptions in execution
- $\leq |f(x) - f(x')| + |f(x') - f'(x')|$ Triangle inequality
- $\leq K \cdot \varepsilon + \delta$ Our initial knowledge

We can propagate error now



$$|f_1(X + \varepsilon_0) + \varepsilon_1 - f_1(X)| \leq K_{f1} \cdot \varepsilon_0 + \varepsilon_1$$

$$|f_2(f_1(X + \varepsilon_0) + \varepsilon_1) + \varepsilon_2 - f_2(f_1(X))|$$

$$\leq K_{f2} \cdot K_{f1} \cdot \varepsilon_0 + K_{f2} \cdot \varepsilon_1 + \varepsilon_2$$

What do we learn from $f'(x')$?

Total Error = Error of Local Approximation

+ Error of Propagation

+ Error caused by the interaction between these two errors*

Analysis Tradeoff: A precise analysis would need to (1) deal with non-linear interactions between propagated error and the error of approximation or (2) use inequalities that conservatively bound the total error. As downsides, the analysis 1 may not be computationally feasible; the analysis 2 may become too imprecise.

X *Y = ?

X / Y =?

Tuning Floating Point Programs: Precimonious

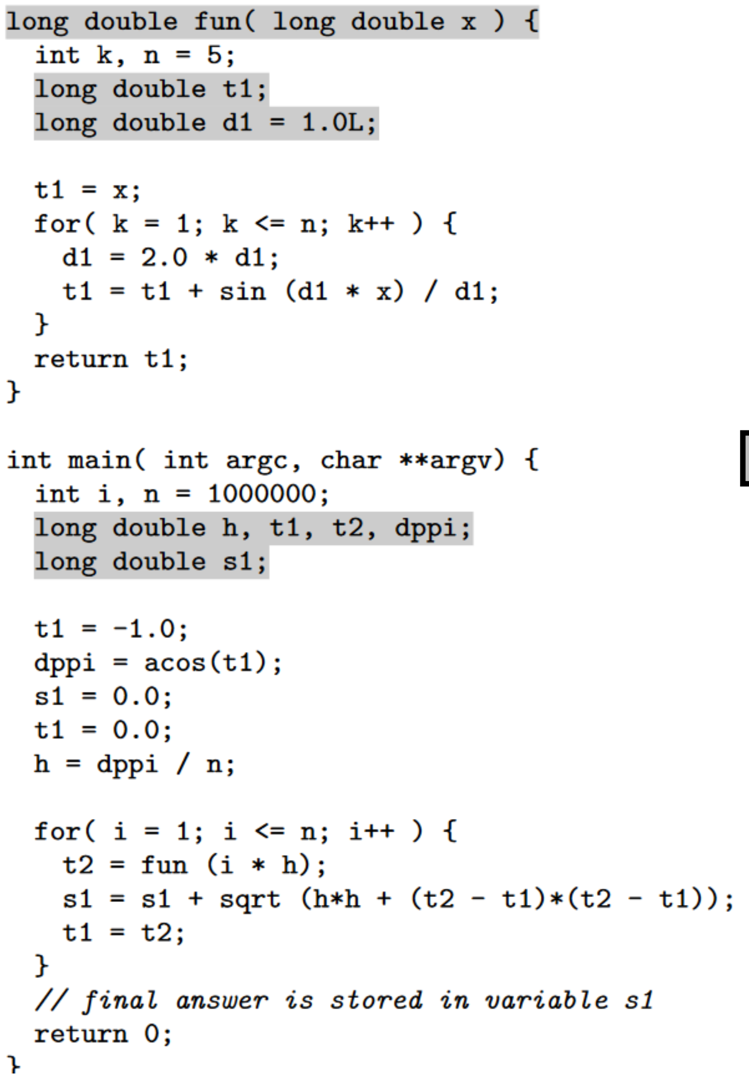
Key idea:

- Identify operations for which, when approximated the output is sensitive to change
- Do not reduce their precision, try other instructions

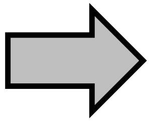
Delta debugging: make multiple changes, then reduce and split the sets if some variables cause low accuracy

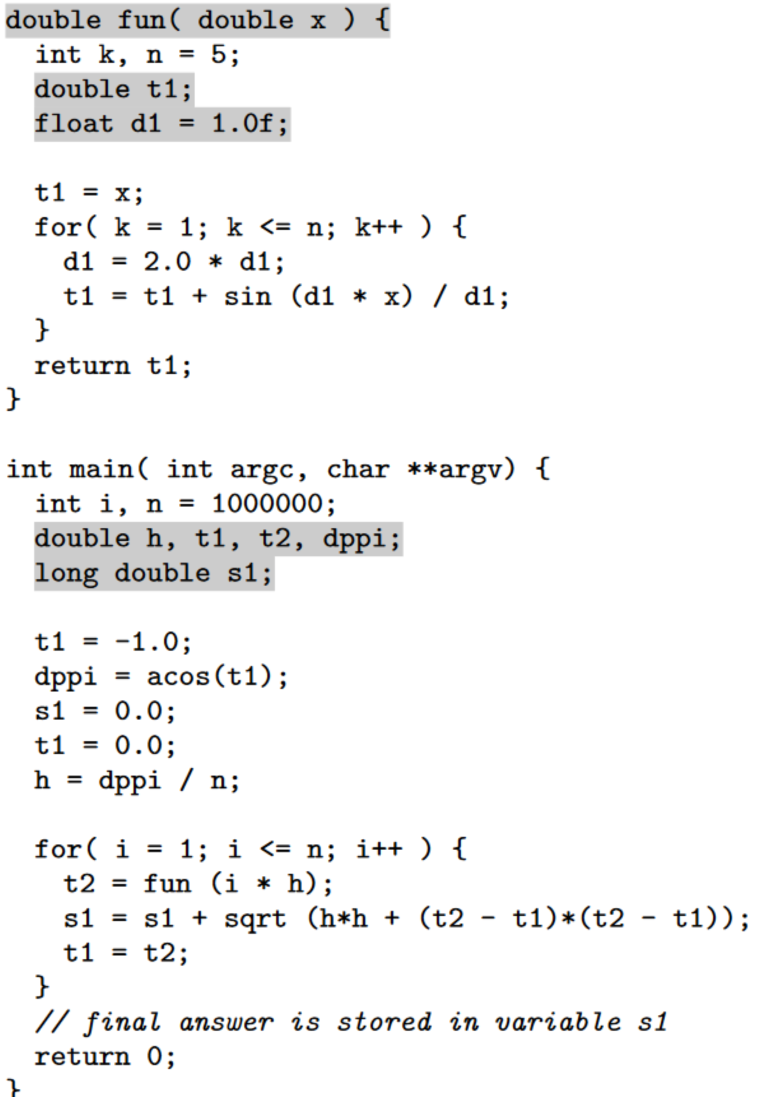
Precimonius: Tuning Assistant for Floating-Point Precision; Rubio-Gonzalez et al. SC 2013

Precimonious Example



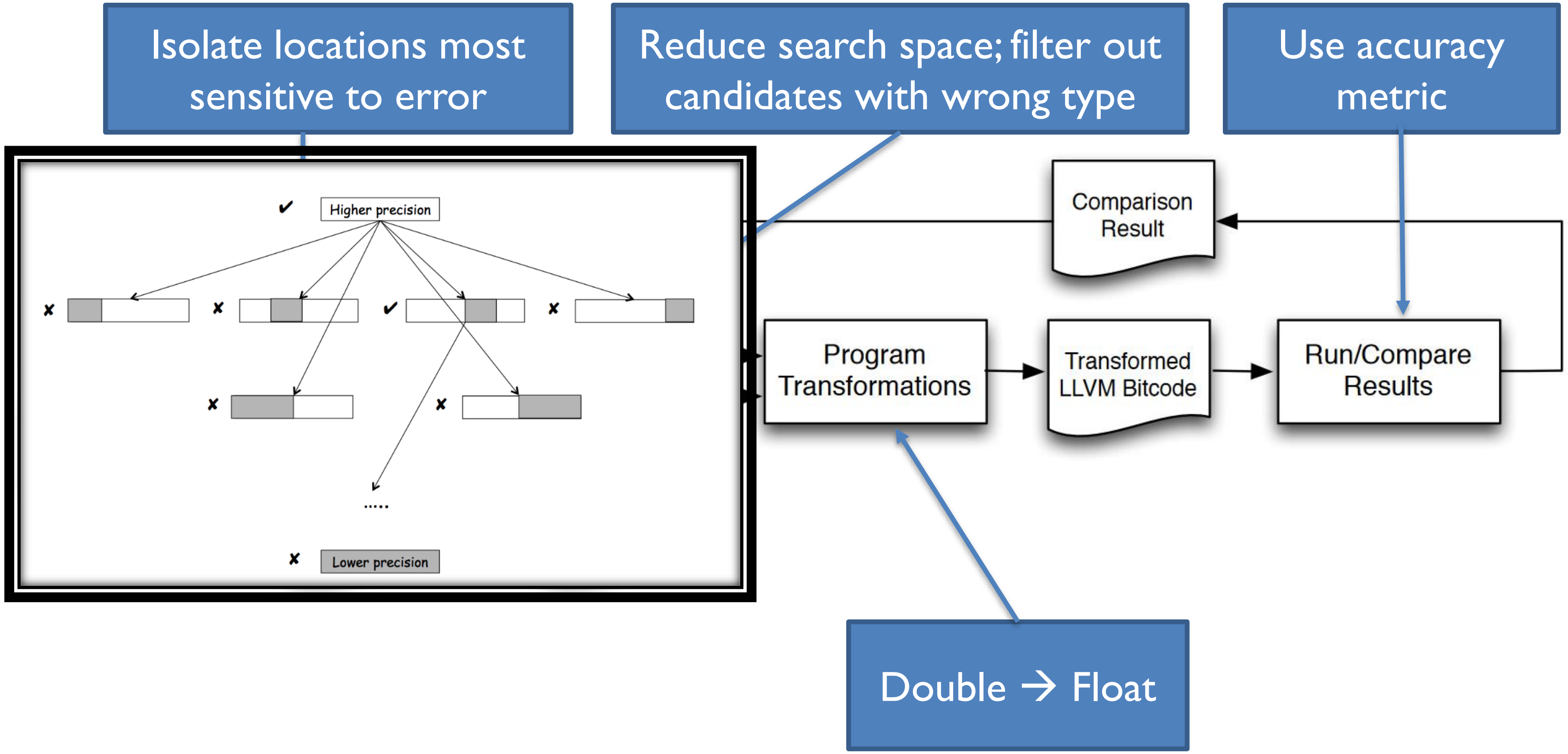
```
long double fun( long double x ) {
  int k, n = 5;
  long double t1;
  long double d1 = 1.0L;

  t1 = x;
  for( k = 1; k <= n; k++ ) {
    d1 = 2.0 * d1;
    t1 = t1 + sin (d1 * x) / d1;
  }
  return t1;
}

int main( int argc, char **argv) {
  int i, n = 1000000;
  long double h, t1, t2, dppi;
  long double s1;

  t1 = -1.0;
  dppi = acos(t1);
  s1 = 0.0;
  t1 = 0.0;
  h = dppi / n;

  for( i = 1; i <= n; i++ ) {
    t2 = fun (i * h);
    s1 = s1 + sqrt (h*h + (t2 - t1)*(t2 - t1));
    t1 = t2;
  }
  // final answer is stored in variable s1
  return 0;
}
```





```
double fun( double x ) {
  int k, n = 5;
  double t1;
  float d1 = 1.0f;

  t1 = x;
  for( k = 1; k <= n; k++ ) {
    d1 = 2.0 * d1;
    t1 = t1 + sin (d1 * x) / d1;
  }
  return t1;
}

int main( int argc, char **argv) {
  int i, n = 1000000;
  double h, t1, t2, dppi;
  long double s1;

  t1 = -1.0;
  dppi = acos(t1);
  s1 = 0.0;
  t1 = 0.0;
  h = dppi / n;

  for( i = 1; i <= n; i++ ) {
    t2 = fun (i * h);
    s1 = s1 + sqrt (h*h + (t2 - t1)*(t2 - t1));
    t1 = t2;
  }
  // final answer is stored in variable s1
  return 0;
}
```

Precimonious



Isolate locations most sensitive to error

Reduce search space; filter out candidates with wrong type

Use accuracy metric

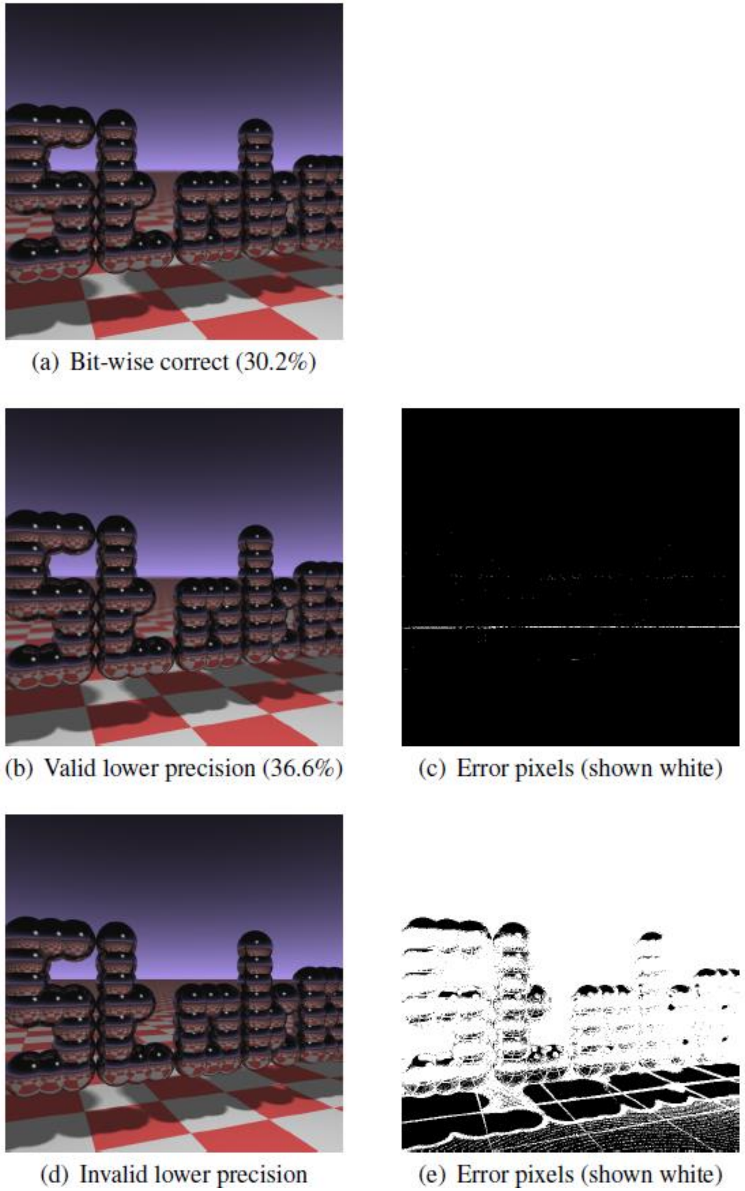
Double → Float

Precimonius: Tuning Assistant for Floating-Point Precision; Rubio-Gonzalez et al. SC 2013

Stoke

- Superoptimizer: tries various ordering of instructions
- Stochastic: searches for the regions of programs and instructions that may have better chance of giving high performance using MCMC

http://stoke.stanford.edu



(a) Bit-wise correct (30.2%)

(b) Valid lower precision (36.6%)

(c) Error pixels (shown white)

(d) Invalid lower precision

(e) Error pixels (shown white)

Stochastic Optimization of Floating-Point Programs with Tunable Precision (Schkufza et al. PLDI 2014)