

Sinking Point

Dynamic precision tracking
for floating-point

Bill Zorn

Dan Grossman

Zach Tatlock

W UNIVERSITY *of* WASHINGTON

billzorn@cs.washington.edu

IEEE 754 floating-point

- Fast, portable, implemented in hardware
- Every operation is completely specified
- Often behavior is similar enough to real numbers
 - But sometimes it isn't!
 - Can be difficult to reason about

A toy example

```
$ python
Python 3.6.5 |Anaconda, Inc.|
(default, Apr 29 2018, 16:14:56)
[GCC 7.2.0] on linux
>>> import math
>>> math.pi + 1e16 - 1e16
4.0
>>>
```

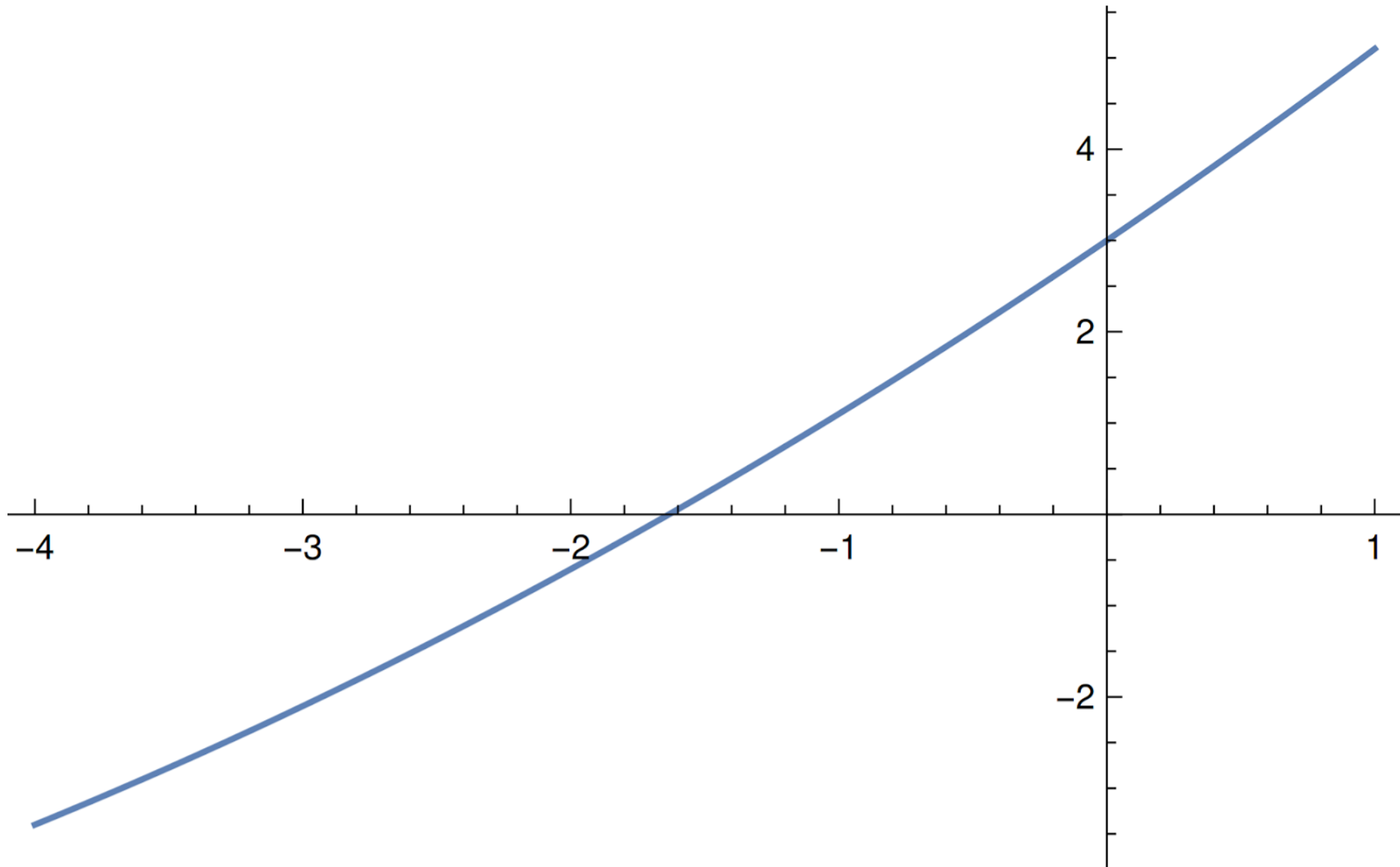
The quadratic formula

$$ax^2 + bx + c = 0$$

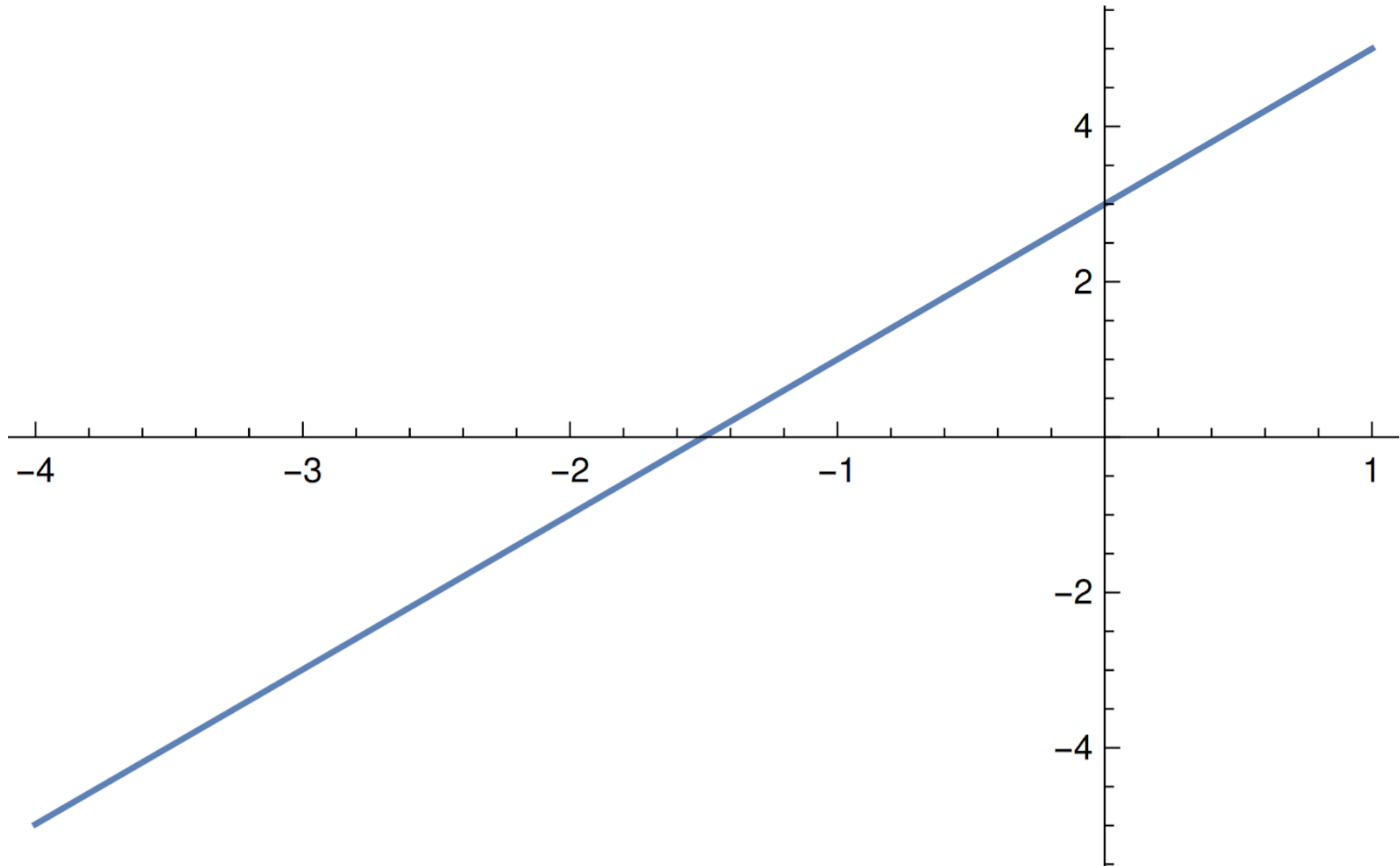
Suppose:

- $b = 2$
- $c = 3$
- a is very small

$$.1x^2 + 2x + 3$$



$$.001x^2 + 2x + 3$$



$$ax^2 + bx + c = 0 \quad x = \frac{-b + \sqrt{b^2 - 4ac}}{2a}$$

<i>a</i>	<i>b</i>	<i>c</i>	<i>x</i> (IEEE 754 double)
.1	2	3	-1.6333997346592444
.001	2	3	-1.5011266906707066
1e-9	2	3	-1.500000013088254
1e-15	2	3	-1.5543122344752189
1e-16	2	3	-2.2204460492503131
1e-17	2	3	0

$$ax^2 + bx + c = 0 \quad x = \frac{-b + \sqrt{b^2 - 4ac}}{2a}$$

<i>a</i>	<i>b</i>	<i>c</i>	<i>x</i> (posit64)
.1	2	3	-1.633399734659244512
.001	2	3	-1.501126690670720465
1e-9	2	3	-1.4999999999210466235
1e-15	2	3	-1.502270530195914978
1e-16	2	3	-1.491862189340054087
1e-17	2	3	-1.734723475976807101
1e-19	2	3	0

This behavior is “correct”

- IEEE 754 floating-point is fully specified
 - Just doing what the standard says
- Up to the programmer to determine which bits are meaningful
 - Write code carefully (like libm)
 - Code analysis tools (FPTaylor, Herbie)
 - Track accuracy dynamically throughout the computation

Sinking-point

- IEEE 754 has a fixed amount of precision
 - Based on available storage, not the properties of the computation
- Sinking-point allows precision to vary
 - Dynamically drop bits from the precision of the significand
 - Compact to store, fast to compute
 - No complex interval computations
 - Still an approximation
 - NOT a sound guarantee
 - NOT a replacement for interval analysis

Printing sinking-point numbers

- Unlike IEEE 754, sinking-point can represent numbers with the same value but different amounts of precision
 - They need to print differently and uniquely
 - 1.5 (how many bits is that?)
 - 1.[3-7] (1.5 with 2 bits of precision)
 - 1.[4991-5009] (1.5 with 10 bits of precision)
- Each “interval” *uniquely* identifies a number
 - NOT interval arithmetic intervals

$$ax^2 + bx + c = 0 \quad x = \frac{-b + \sqrt{b^2 - 4ac}}{2a}$$

<i>a</i>	<i>b</i>	<i>c</i>	<i>x</i> (IEEE 754 double)	<i>x</i> (exact)	<i>x</i> (sinking-point)
.1	2	3	-1.6333997346592444	-1.6333997346592446	-1.633399734659244[0-8]
.001	2	3	-1.5011266906707066	-1.5011266906707219	-1.501126690670[68-78]
1e-9	2	3	-1.500000013088254	-1.5000000011250001	-1.[49999995-50000005]
1e-15	2	3	-1.5543122344752189	-1.50000000000000011	-1.[44-56]
1e-16	2	3	-2.2204460492503131	-1.50000000000000002	-[1.8-2.5]
1e-17	2	3	0	-1.50000000000000000	[-1.-+1.]

$$ax^2 + bx + c = 0 \quad x = \frac{1}{(\sqrt{b^2 - 4ac} + b)(\frac{-1}{2c})}$$

<i>a</i>	<i>b</i>	<i>c</i>	<i>x</i> (IEEE 754 double)	<i>x</i> (exact)	<i>x</i> (sinking-point)
.1	2	3	-1.6333997346592446	-1.6333997346592446	-1.633399734659244[5-7]
.001	2	3	-1.5011266906707219	-1.5011266906707219	-1.50112669067072[18-20]
1e-9	2	3	-1.5000000011250001	-1.5000000011250001	-1.500000001125000[0-2]
1e-15	2	3	-1.50000000000000013	-1.50000000000000011	-1.5000000000000001[3-4]
1e-16	2	3	-1.50000000000000004	-1.50000000000000002	-1.5000000000000000[4-5]
1e-17	2	3	-1.5	-1.5000000000000000	-1.[4999999999999999 -50000000000000001]

Sinking-point addition

- Sinking-point stores extra information
 - $(\text{value}, \text{prec})$
- $(v_1, \text{prec}_1) + (v_2, \text{prec}_2) = (v_3, \text{prec}_3)$

~~5.25~~ + 4.015625 -3
~~0b101.01~~????
 + 0b100.000001



What information should *prec* hold?

- $\text{prec} = (\mathbf{p}, \mathbf{n})$
- $\mathbf{p}$ is the “bitwidth” of the number
- $\mathbf{n}$ is the “most significant unknown bit”

Sinking-point addition

- Sinking-point stores extra information
 - (value, p, n)
- $(v_1, p_1, n_1) + (v_2, p_2, n_2) = (v_3, p_3, n_3)$

$(v_1 = 5.25, p_1 = 5, n_1 = -3) + (v_2 = 4.015625, p_2 = 9, n_2 = -7)$

```
  0b101.01????
+ 0b100.000001
-----
  0b1001.010001
  0b1001.01
```

$n_3 = -3 = \max(n_1, n_2)$
 $v_3 = 9.25 = \text{round}(v_1 + v_2, n_3)$
 $p_3 = 6$

Subtraction

- Addition, but with opposite signs
- $(v_1, p_1, n_1) - (v_2, p_2, n_2) = (v_3, p_3, n_3)$

$$(v_1 = 5.25, p_1 = 5, n_1 = -3) - (v_2 = 4.015625, p_2 = 9, n_2 = -7)$$

```
  0b101.01????
- 0b100.000001
-----
  0b1.001111
  0b1.01
```

$$\begin{aligned} n_3 &= -3 &= \max(n_1, n_2) \\ v_3 &= 1.25 &= \text{round}(v_1 - v_2, n_3) \\ p_3 &= 3 \end{aligned}$$

Multiplication (and division)

- $(v_1, p_1, n_1) * (v_2, p_2, n_2) = (v_3, p_3, n_3)$
- “Grade school multiplication”

$$(v_1 = 5.25, p_1 = 5, n_1 = -3) * (v_2 = 5, p_2 = 3, n_2 = -1)$$

0b101.01	0b10101.???
* 0b101.??	0b0000.0??
-----	0b101.01?
0b10101.???	+ 0b???.???
0b0000.0??	-----
0b101.01?	0b11010.01
0b???.???	0b111???

$$p_3 = 3 = \min(p_1, p_2)$$

$$v_3 = 28 = \text{round}(v_1 * v_2, p_3)$$

$$n_3 = 1$$

Square root

- Only one operand, so no min or max

- $\text{sqrt}(v_1, p_1, n_1) = (v_2, p_2, n_2)$

$$(v_1 = 7.25, p_1 = 5, n_1 = -3)$$

$$p_2 = 6 = p_1 + 1$$

$$v_2 = 2.6837 = \text{round}(\text{sqrt}(v_1), p_3)$$

$$n_2 = -4$$

$$\text{sqrt}(111.001) = 10.1010\textcolor{brown}{1011}...$$

$$\text{sqrt}(111.01?) = 10.10110001...$$

$$\text{sqrt}(111.011) = 10.1011\textcolor{brown}{0111}...$$

Upshot

- Sinking-point gives confidence that the bits in results are meaningful
- About as fast as IEEE 754
 - Only a few extra bits ($\log_2(\mathbf{p}_{\max}) + 1$, see paper)
 - Bitwise compatible for exact inputs
- Like IEEE 754, still an approximation
 - Not a sound guarantee
 - Does not replace other analyses, but helps indicate when to use them

Titanic



titanic.uwplse.org

Guaranteed to
float correctly

FPBench



fpbench.org

Common standards
for the floating-point
research community