

What Every Computer Programmer Should Know About Floating-Point Arithmetic

Joseph D. Darcy

joe.darcy@sun.com

<http://blogs.sun.com/darcy/>

Staff Engineer, Sun Microsystems, Inc.

Overview: Reduce Surprises, Increase Understanding

- Understand why
 - $0.1+0.1+0.1+0.1+0.1+0.1+0.1+0.1+0.1+0.1 \neq 1.0$
 - $0.1f \neq 0.1d$
- Outline
 - Floating-Point Fundamentals
 - Decimal \leftrightarrow Binary Conversion
 - Top 1.0e1 Floating-point FAQs, Mistakes, Surprises, and Misperceptions
 - Q & A

Floating-Point Fundamentals

Why Floating-point?

- Integers aren't convenient for all calculations
- Floating-point arithmetic is a *systematic approximation* of arithmetic on \mathcal{R}
 - Exponent and significand (mantissa) fields
 - “Decimal point” floats according to exponent value
- Exact multiplication can double the number of bits manipulated at each step — must approximate to keep computation tractable!
- Exactness rarely needed to get usable results

What are real numbers?

- Real numbers (\mathfrak{R}) include:
 - Integers (0, -1, 32768, ...)
 - Fractions (rational numbers) ($1/2$, $22/7$, $355/113$, ...)
 - Irrational numbers (π , e , $\sqrt{2}$, ...)
- Real numbers form a *field*; field axioms:
 - Addition and multiplication are commutative ($a \text{ op } b = b \text{ op } a$) and associative ($(a \text{ op } b) \text{ op } c = a \text{ op } (b \text{ op } c)$)
 - Closed under addition and multiplication
 - Also identity elements, distributivity, 13 total

How to Approximate?

- Not all approximations equally good!
- Would like approximation to be:
 - Deterministic
 - Reproducible
 - Predictable
 - Reliable
 - Accurate
 - Fast
 - ...

Floating-Point Approximation

- Represents a proper subset of \mathfrak{R}
- Obeys a proper subset of the field axioms
 - Addition and multiplication are commutative
 - Round-off precludes most other field axioms
- Floating-point is fundamentally *discrete*

Levels

Level 1	$-\infty \dots \text{---} 0 \text{---} \dots +\infty$	Extended real numbers
<i>many-to-one</i> ↓	<i>rounding</i>	↑ <i>one-to-many</i>
Level 2	$\{-\infty \cdot \cdot \cdot \cdot \cdot \cdot -0\}$ $\cup \{+0 \cdot \cdot \cdot \cdot \cdot \cdot +\infty\}$ $\cup \text{NaN}$	Floating-point data - an algebraically completed system
<i>one-to-many</i> ↓	<i>representation specification</i>	↑ <i>many-to-one</i>
Level 3	<i>(sign, exponent, significand)</i> $\cup \{-\infty, +\infty\}$ $\cup \text{qNaN} \cup \text{sNaN}$	Representations of floating-point data
<i>one-to-many</i> ↓	<i>encoding for floating-point data</i>	↑ <i>many-to-one</i>
Level 4	0111000...	Bit strings

Binary Floating-Point Numbers

- Infinite number of real numbers, only finite number of floating-point numbers

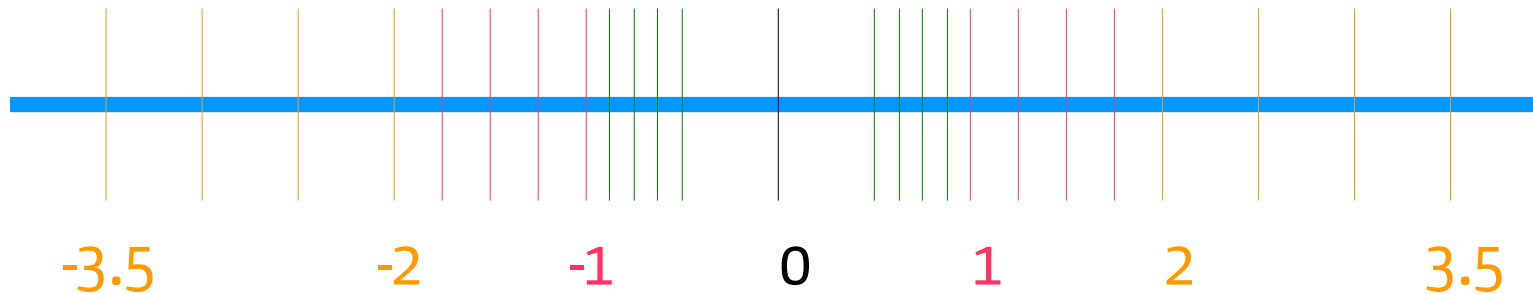
- Representable numbers:

$$\pm \text{binarySignificand} \cdot 2^{\text{exponent}} = \pm b_0 . b_1 b_2 \dots b_{p-1} \cdot 2^{\text{exponent}}$$

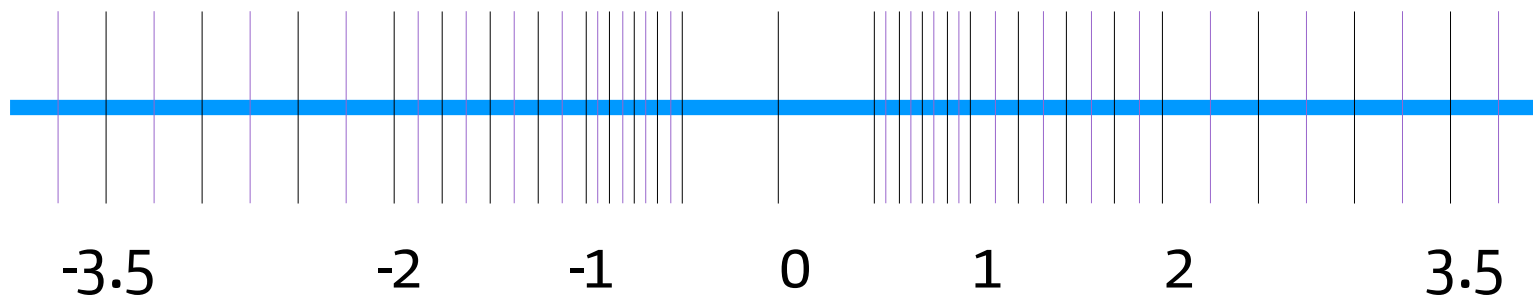
- *binarySignificand* limited in precision, only has p bits
 - *normalized*: (for now) assume $b_0 = 1$
- Floating-point numbers are sums of powers of two with a bounded ratio of components
- Fixed width \Rightarrow performance (roughly) independent of magnitude of operand(s) and result

Toy Binary Floating-Point Numbers

- Toy format 1: $p = 3$, exponent $\in \{-1, 0, 1\}$



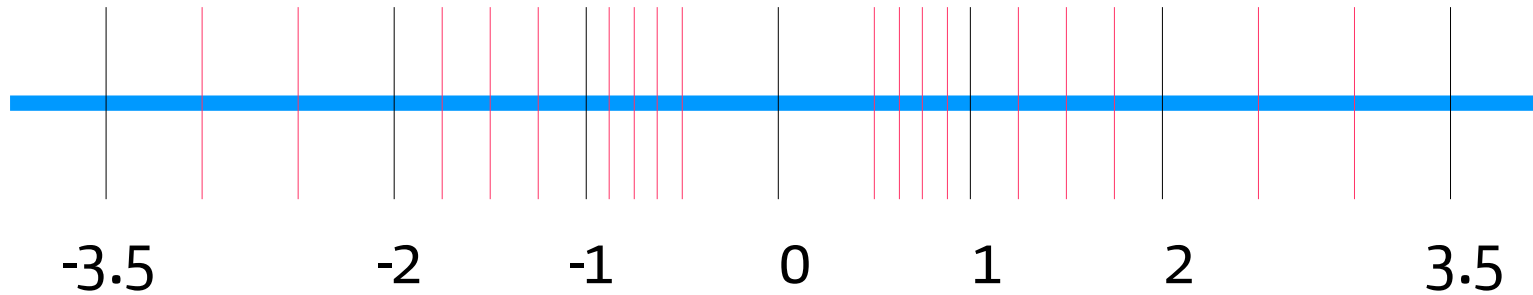
- Toy format 2: $p = 4$, exponent $\in \{-1, 0, 1\}$



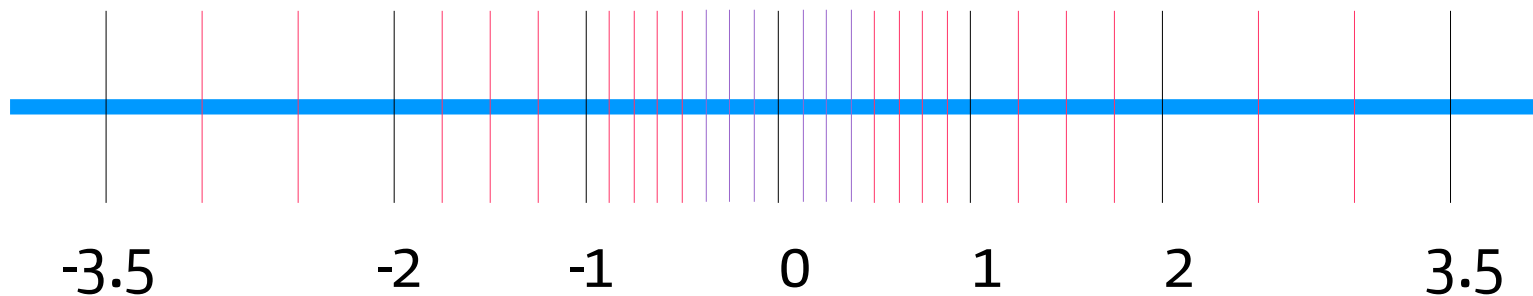
Toy Observations

- Cross exponent boundary
 - distance between adjacent numbers doubles
 - *relative precision is constant*
- One more precision bit, $2 \times$ representable numbers
 - Slight increase in largest representable number

Subnormals



- Fill-in comparatively large gap around zero
 - for smallest numbers, allow $b_0=0$; creates *subnormal* numbers
 - Eases numerical analysis



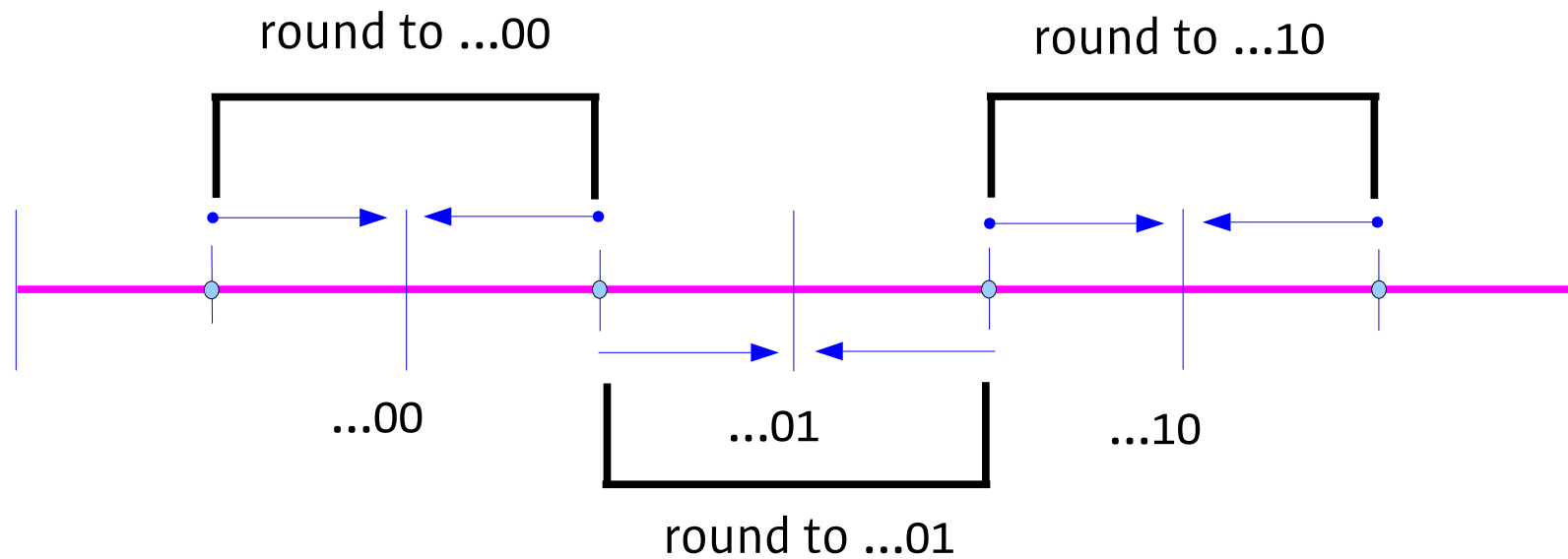
float and **double** formats

- **float** has $p = 24$, 8 exponent bits
- **double** has $p = 53$, 11 exponent bits
- All **float** numbers are exactly representable as **double** numbers
- Between adjacent **float** numbers,
 $2^{(53-24)} = 2^{29} \approx 500$ million **double** numbers

IEEE 754 Floating-Point

- IEEE 754: universal binary floating-point standard
- Fundamentally *simple*
- For each of the defined operations $\{+, -, *, /, \vee\}$
 - First, calculate the infinitely precise result
 - Second, round this result to the nearest representable number in the target format
 - (On a tie, chose the one with the last bit zero — round to nearest even)

Rounding-to-Nearest-Even Illustrated



Round to Nearest Even

- Locally optimal, easy to understand and analyze
- *But*, still lose information
- Few useful field axioms

- Failure of associativity of addition:

$$\begin{aligned}(1.0f + 3.0e-8f) + 3.0e-8f &== 1.0f \\ 1.0f + (3.0e-8f + 3.0e-8f) &== 1.0000001f\end{aligned}$$

Creating Closure

- When an operation on a set of inputs doesn't have a defined result, define a new kind of result
- Creates a *closed* system
- New values might be unfamiliar but convenient, or sensible mathematically

Math through the ages

- Positive integers $\{1, 2, 3, \dots\}$ and subtraction \Rightarrow all integers $\{\dots, -1, 0, 1, \dots\}$
- Integers and division \Rightarrow rational numbers
 - division by zero not allowed
- Rational numbers and taking roots \Rightarrow
 - irrational numbers $(\sqrt{2})^\dagger$
 - imaginary and complex numbers $(\sqrt{-1})$

[†]Actually just *algebraic numbers*, all irrationals are more work

IEEE 754 Special Values

- Close the floating-point arithmetic operations
 - Can have sensible semantics for new values
 - Allows computation to continue to a point where it is convenient to detect the “error” (e.g., root finder)
- Besides representations for real numbers
 - Infinities ($\pm\infty$)
 - NaN (Not-a-Number)

Special Arithmetic

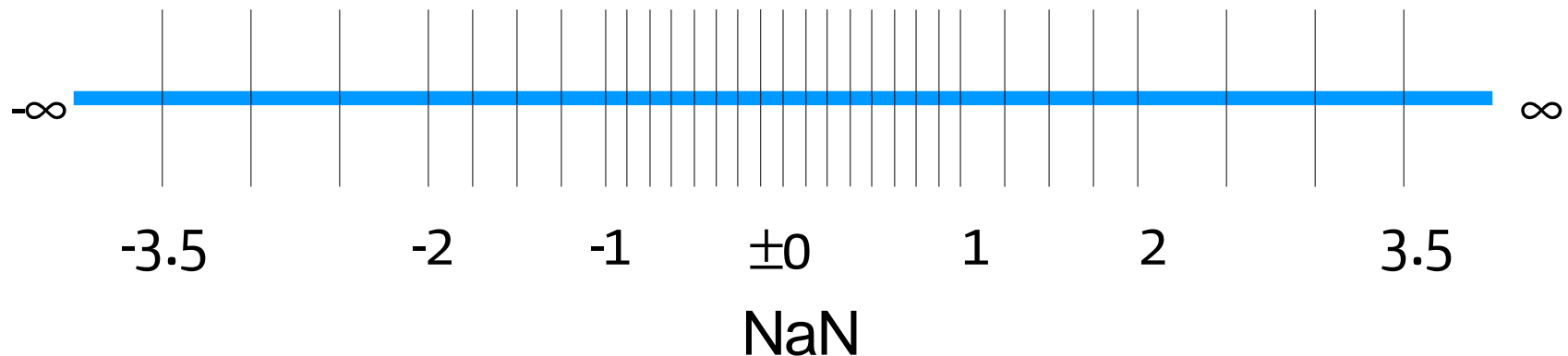
- Infinity results from
 - overflow, $\text{MAX_VALUE} * 2.0$
 - division by zero, $1.0/0.0$
- NaN represents *invalid* values
 - $0/0$
 - $\infty \cdot 0$
 - $\infty - \infty$
 - $\sqrt{-1}$, etc.
 - NaN shifts code complexity;
their creation doesn't have to be prevented

A Distinction with Zero Difference

- IEEE 754 has signed zeros as two distinct values
 - $+0.0 == -0.0$ *but*,
 - While, $\mathbb{F}(+0.0)$ is *usually* the same as $\mathbb{F}(-0.0)$ it is *not* always
 - $1.0/+0.0 == +\infty$
 - $1.0/-0.0 == -\infty$
- The sign of a zero input
 - Usually only affects of sign of a zero result (divide exception above)
 - Otherwise, can mostly be ignored

All the Toys

- All operations on toy floating-point values will result in some other toy floating-point value



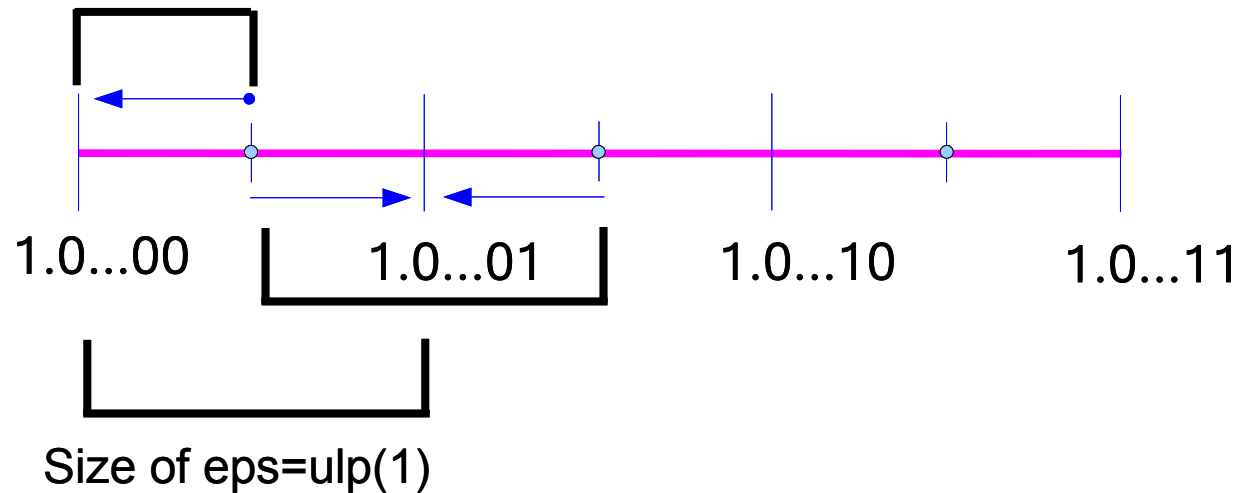
Defining an Order

- IEEE 754 defines, $<$, $>$, and $==$ relations
- NaN is *unordered* with respect to other values
 - NaN $<$ 5 is false; NaN $>$ 5 is false;
NaN $==$ 5 is false; NaN $!=$ 5 is true
 - NaN $==$ NaN is false too!
- Because of signed zeros and NaN, IEEE 754 $<$ does not define a total ordering
 - Use `{Float, Double}.compareTo` if you need a total ordering
- IEEE 754 $==$ is *not* an equivalence relation

ulps

- Ulp = unit in the last place
 - $\text{ulp}(r \in \mathfrak{R})$ = distance between floating-point numbers bracketing r
 - If r is exactly representable, $\text{ulp}(r)$ is the distance to the next larger floating-point value
 - Convenient measure of relative error
- Size of $\text{ulp}(r)$ function of:
 - Magnitude of r
 - Floating-point format's precision (and range)

Ulp, **eps**, and rounding threshold



- **eps** = $\text{ulp}(1) = \text{nextUp}(1) - 1 = 2^{-52} \approx 2.22\text{e-}16$
- Rounding threshold = **eps**/2 + $\varepsilon = 2^{-53} + 2^{-105}$

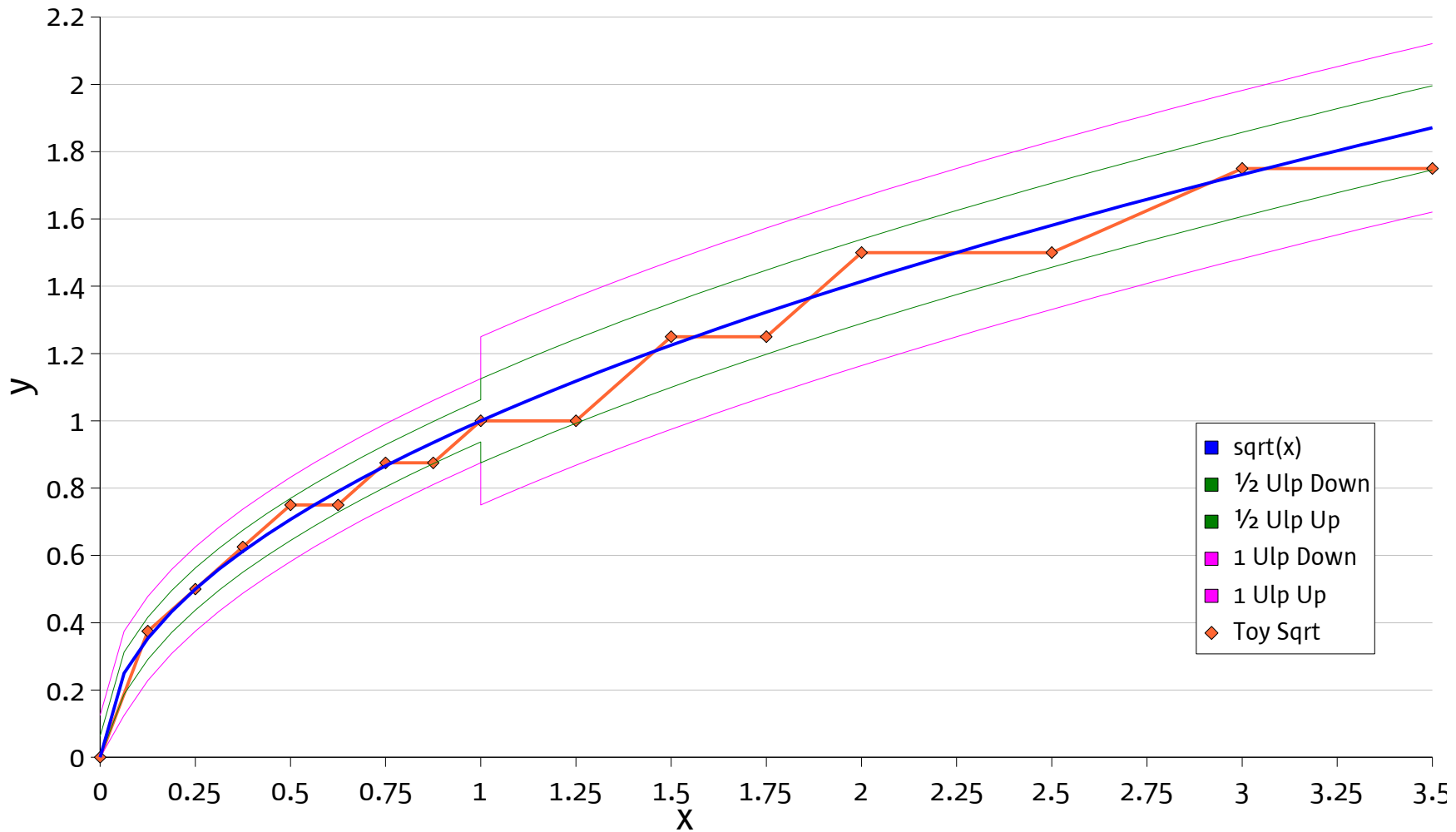
Correct Rounding

- Error $\leq \frac{1}{2}$ ulp \Rightarrow function is *correctly rounded*
 - Floating-point value closest to exact result is returned, best a floating-point function can do
 - Basic arithmetic and sqrt are correctly rounded
- Most single argument methods in **Math** and **StrictMath** have error < 1 ulp

Round, round, get around

- A correctly rounded approximation most likely to preserve other properties of interest
 - Cardinal values are correct;
 $\text{sqrt}(9.0) = 3.0$
 - Monotonicity;
 $x, y > 1; x \leq y \Rightarrow \text{sqrt}(x) \leq \text{sqrt}(y)$
- Other fairly accurate approximations don't necessarily preserve desired properties

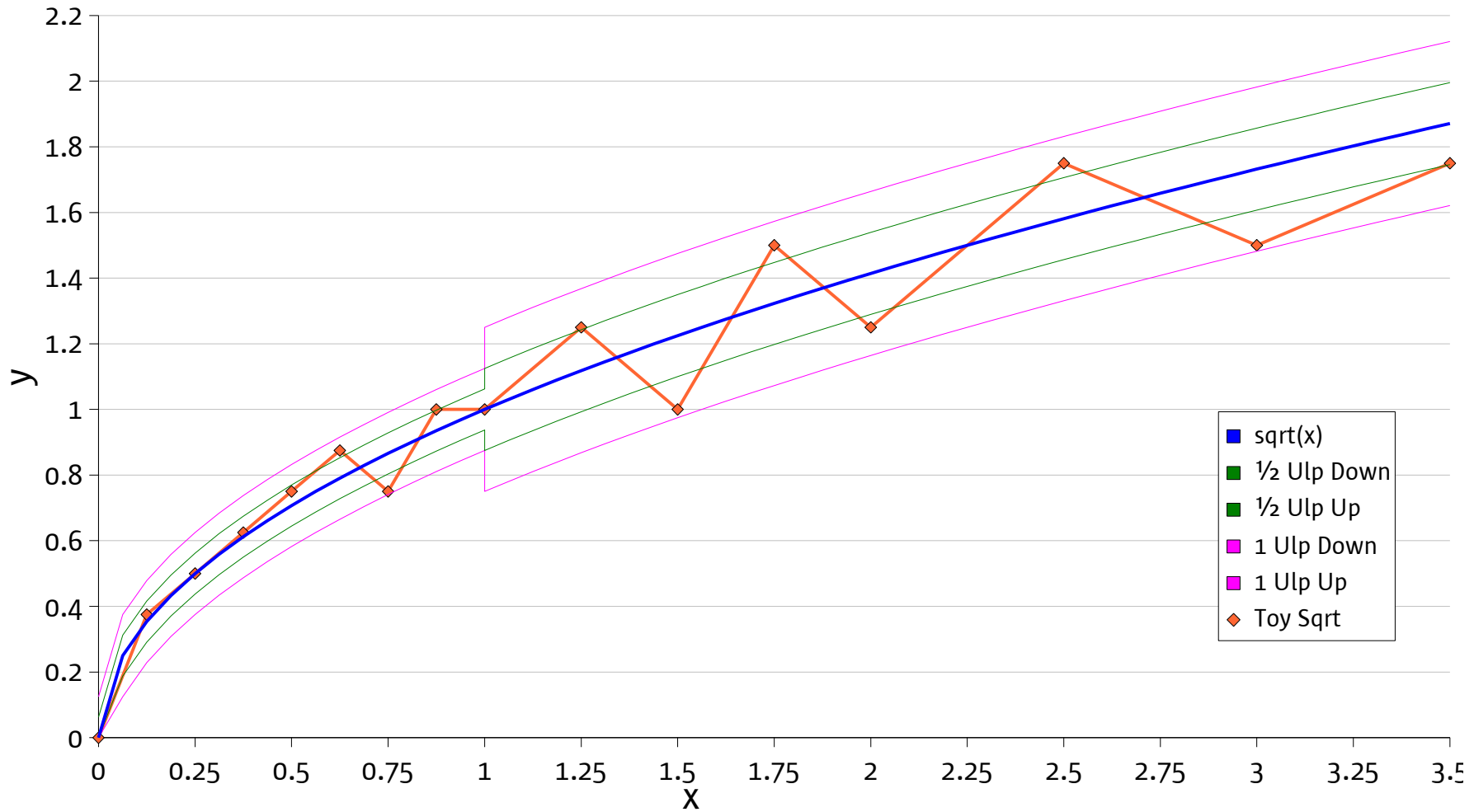
Correctly Rounded Toy Square Root



Tabulated Toy Square Root

x	sqrt(x)
0.0	0.0
0.125	0.375
0.250	0.500
0.375	0.625
0.500	0.750
0.625	0.750
0.750	0.875
0.875	0.875
1.000	1.000
1.250	1.000
1.500	1.250
1.750	1.250
2.000	1.500
2.500	1.500
3.000	1.750
3.500	1.750

Not Correctly Rounded Square Root



Getting More Precision

- Precision achieved from floating-point arithmetic
 - not limited to precision of a single value
 - depends on over/underflow threshold
- Several packages use floating-point operations to create higher-precision libraries
- Built on implementing exact individual floating-point operations

Higher precision operations

- Rounding errors are correlated, *not* independent
- Use knowledge of rounding to recover “lost” trailing bits
 - First calculate high-order bits
 - Then calculate low-order bits
- If you know exactly how big the error is, you can get rid of the error :-)

Addition

Given a and b as input, calculate non-overlapping x and y such that exact addition of $x + y$ is equal to the exact addition of $a + b$.

```
double a, b, x, y, a', b', a_round, b_round;
```

```
x = a + b; // calculate high order bits  
           // x == a + b + err(a+b)
```

```
b' = x - a;
```

```
a' = x - b;
```

```
a_round = a - a';
```

```
b_round = b - b';
```

```
y = a_round + b_round;
```

```
// y is rounding error in x = a + b
```

```
// ∴ x + y == exact value of a + b
```

Decimal \leftrightarrow Binary Conversion


Base Conversion Basics

- Integers can be represented exactly in any base
- In general, fractional quantities exactly representable as a finite string in one base *cannot* be exactly represented as a finite string in another base
 - In base 10, $1/3$ is the non-terminating expansion $0.33333333\dots$
 - In base 3, $1/3$ is $0.1_{(3)}$
- Many floating-point surprises are related to decimal \leftrightarrow binary conversion properties

Decimal → Binary

- Most terminating decimal fractions cannot be exactly represented as terminating binary fractions

- Try to convert 0.1 to a binary fraction

$0.1 \times 2 = \underline{0}.2$	0	
$0.2 \times 2 = \underline{0}.4$	0	
$0.4 \times 2 = \underline{0}.8$	0	
$0.8 \times 2 = \underline{1}.6$	1	
$0.6 \times 2 = \underline{1}.2$	1	Repeated state
$0.2 \times 2 = \underline{0}.4$	0	

- 0.1 is $0.0\underline{0011}\dots$ in binary

Binary \rightarrow Decimal

- All terminating binary fractions can be expressed exactly as terminating base 10 fractions
- Intuition: $10 = 2 \times 5$ so all fractions in base 2 or base 5 can also be expressed in base 10
- Proof: $\frac{1}{2^k} = \frac{5^k}{10^k}$
- 5^k is a representable integer; dividing by 10^k just shifts the decimal point; sums of 2^i still terminate
- Floating-point numbers are sums of powers of two

How to Convert

- Decimal → binary
(literals, `{Float, Double}.valueOf` and `parse{Float, Double}` methods)
 - Conversion must in general be inexact
 - Use standard floating-point rounding:
return binary floating-point value nearest exact decimal value of input
- Binary → decimal (`{Float, Double}.toString`)
 - Feasible to return exact decimal string...

The Cost of Exactness

- Number of decimal digits for 2^{-n} grows with increasingly negative exponents

2^{-n}	Exact decimal string
2^{-1}	0.5
2^{-2}	0.25
2^{-3}	0.125
2^{-4}	0.0625
2^{-5}	0.03125
2^{-6}	0.015625
2^{-7}	0.0078125
2^{-8}	0.00390625
2^{-9}	0.001953125

Extreme Values

- **Double.MIN_VALUE** = 2^{-1074}

- Exact decimal value:

4.940656458412465441765687928682213723650598026143247644255856825006755
0727020875186529983636163599237979656469544571773092665671035593979639
8774796010781878126300713190311404527845817167848982103688718636056998
7307230500063874091535649843873124733972731696151400317153853980741262
3856559117102665855668676818703956031062493194527159149245532930545654
4401127480129709999541931989409080416563324524757147869014726780159355
2386115501348035264934720193790268107107491703332226844753335720832431
9360923828934583680601060115061698097530783422773183292479049825247307
7637592724787465608477820373446969953364701797267771758512566055119913
1504891101451037862738167250955837389733598993664809941164205702637090
279242767544565229087538682506419718265533447265625e-324

- Awkward and impractical, is this necessary?

Criteria for Conversions

- Want binary → decimal → binary conversion to reproduce the original value
 - **d2** in **d2=parseDouble (toString (d1))** is the same as **d1**
 - Allows text to be used for reliable data interchange
 - Exact decimal value is *not* necessary
- Decimal → binary conversion must already deal with imprecision and rounding
- Use an *inexact* decimal string with enough *precision* to recreate original value

How Long a String is Needed?

- **float** format has 6 to 9 digits of decimal precision
- **double** format has 15 to 17 digits of decimal precision
- Decimal precision varies since binary and decimal numbers have different relative densities in different ranges

Implications: *WYSI Not WYG*

- What you see is *not* what you get
 - "0.1f" \neq 0.1 after conversion; exact value:
0.100000001490116119384765625
 - "0.1d" \neq 0.1 after conversion; exact value:
0.10000000000000000055511151231...
- Correct digits
 - Leading 8 for **float**
 - Leading 17 for **double**
- When you see a decimal string, think of its nearest **binary** neighbor

You Are in a Twisty Maze of Little Passages, All Different...

- String representation of a floating-point value is format dependent
 - `Float.toString(0.1f) = "0.1"`
 - `Double.toString(0.1f) = "0.10000000149011612"`
 - `float` approximation has 24 significant bits;
`double` approximation has 53 significant bits
 - `Double.toString(0.1d) = "0.1"`
- To preserve values, must print out and read in floating-point numbers in the same format

Base Conversion Summary

- Both decimal to binary and binary to decimal conversions are inexact
 - Decimal \rightarrow binary: fundamentally inexact
 - Binary \rightarrow decimal: done inexactly for practical reasons
- Roundtrip binary \rightarrow decimal \rightarrow binary can be *exact* since the inexactness is correlated
- Can only exactly represent *binary* values

An Alternative: Hex floating-point literals

- Ways to specify a floating-point value
 - Decimal literal or string conversion
 - Bit pattern to **intBitsToFloat** or **longBitsToDouble**
 - What is `Float.intBitsToFloat(0x3dcccccd)` ?
 - What is `Double.longBitsToDouble(0x3fb999999999999aL)` ?
 - (Functionally analogous to an untyped union in C)
- Useful to have technique that is simultaneously
 - Human readable
 - Obviously unambiguous
 - Related to the floating-point representation

Hex floating-point literals

- Part of C99 and JDK 5
 - Unambiguous and (somewhat) readable
 - Have fields corresponding to sign, significand, and exponent; *sign0xsignificandpexponent*

- Examples:

1.0	0x1.0p0
-2.0	-0x1.0p1
0.25	0x1.0p-2
3.0	0x1.8p1
0.1	0x1.9999999999999999ap-4
Max value	0x1.fffffffffffffffffp1023
Min norm	0x1.0p-1022
Max sub	0x0.fffffffffffffffffp-1022
Min value	0x0.0000000000000001p-1022

Top 1.0e1 Floating-Point FAQs, Mistakes, Surprises, and Misperceptions

1 Precision is the same as Accuracy

- Precision \neq Accuracy
 - *Precision* is a measure of how fine a distinction you can make
 - *Accuracy* is a measure of error
- Using more precision for intermediate results usually gives a more accurate computed answer
- Need a notion of a correct answer to measure error

2 Expecting Exact Results

- $0.1+0.1+0.1+0.1+0.1+0.1+0.1+0.1+0.1+0.1 \neq 1.0$
 - The literal "0.1" doesn't equal 0.1
 - Limited precision of floating-point implies roundoff
- More generally, exact results also fail from
 - Limited range (overflow, underflow)
 - Special values

3 Expecting *Only* Inexact Results

- Exact floating-point computations
 - Operations on “small” integers
 - Representing in-range powers of 2 (e.g., 1.0/8.0)
 - Special algorithms
 - extending floating-point precision

3.5 Expecting *Only* Inexact Results

- A floating-point number is *not* a stand-in for nearby values
 - Floating-point arithmetic operations assume their inputs are exact
 - Use other techniques to estimate overall error
 - error analysis
 - interval arithmetic

4 Using Floating-point for Money

- Fractional \$, £, € can't be stored exactly
 - Decimal → binary conversion issue
- Operations on values won't be exact (bad for balancing a checkbook!)
- Recommendations
 - Use an integer type (**int** or **long**) operating on cents
 - Use **java.math.BigDecimal** for exact calculations on decimal fractions
 - If you must use floating-point, operate on cents
 - Problems with limited exact range

5 Cardinal sin/cos values on degrees

- Why doesn't
 - $\sin(\text{toRadians}(180)) == 0.0$
 - $\cos(\text{toRadians}(90)) == 0.0$
- $\text{toRadians} \equiv \text{angleInDegrees}/180.0 * \text{Double.PI}$
 - Conversion of degrees to radians is inexact
 - $\text{Double.PI} \neq \pi, \therefore \sin(\text{Double.PI}) \neq \sin(\pi)$
(in general case, roundoff in multiply, divide)
- Cope with small discrepancies or use degree-based transcendental functions

6 Checking Floating-Point Equality

- Sometimes okay to compare for equality
 - When calculations are known to be exact
 - To synthesize a comparison
 - Compare against 0.0 to avoid division by zero
- *But*, floating-point results are usually inexact
 - Comparing floating-point numbers for equality may have undesirable results

6.5 Checking Floating-Point Equality

- An infinite loop:

```
d = 0.0;  
while(d != 1.0)  
    d += 0.1;
```

- For counted loops, use an integer loop count:

```
d = 0.0;  
for(int i = 0; i < 10; i++)  
    d += 0.1;
```

- To test against a floating-point value, use ordered comparisons (<, <=, >, >=):

```
d = 0.0;  
while(d <= 1.0)  
    d += 0.1;
```

7 Using **float** for computations

- Storing low-precision data as **float** is fine, but
- Generally not recommended to use **float** for computations
 - **float** has less than half the precision of **double**
 - Using **double** intermediates greatly reduces the risk of roundoff problems polluting the answer
 - Round **double** value back to **float** to give a **float** result
- Extra internal precision is ablative armor against roundoff problems

8 Trusting Venerable Formulas

- Some formulas found in text books don't work very well with floating-point numbers
- Formulas may implicitly assume real arithmetic
- Don't adequately take floating-point rounding into account

8.25 Trusting Venerable Formulas

- Example: Heron's formula for the area of a triangle given the lengths of its sides:

$$s = ((a + b) + c) / 2,$$

$$\text{Area} = \text{sqrt}(s \cdot (s - a) \cdot (s - b) \cdot (s - c))$$

- Most often works just fine:
Area($\triangle(3, 4, 5)$), $s = 6$, Area = $\text{sqrt}(6 \cdot 3 \cdot 2 \cdot 1) = 6$
Area($\triangle(3, 4, 5)$) = $\frac{1}{2}b \cdot h = \frac{1}{2} \cdot 4 \cdot 3 = 6$
- But, formula can **fail** for needle like triangles
(no bits may be correct!)

8.5 Trusting Venerable Formulas

- Calculate the area of $\triangle(12,345,679, 12,345,679, 1.01234)$
 - $\frac{1}{2}b \cdot h$ should be $\approx 6,000,000$
 - Area from Heron's in **float** precision:
12,345,680.00, **2X too big!**
 - Use alternative formula
Area = $\text{sqrt}((a+(b+c))(c-(a-b))(c+(a-b))(a+(b-c)))/4$:
6,249,012.00, **Correct answer.**
 - Area from Heron's with **float** inputs **but double** intermediates:
6,249,012.00, **Correct answer.**

8.75 Trusting Venerable Formulas

- Calculate the area of $\triangle(12,345,679, 12,345,678, 1.01234)$
 - $\frac{1}{2}b \cdot h$ should be $\gg 0$
 - Area from Heron's in **float** precision:
0.0 !!!
 - Use alternative formula
Area = $\text{sqrt}((a+(b+c))(c-(a-b))(c+(a-b))(a+(b-c)))/4$:
972,730.06 **Correct answer.**
 - Area from Heron's with **float** inputs **but double** intermediates:
972,730.06, **Correct answer.**

8.875 Trusting Venerable Formulas

- Problem with Heron's formula hard to recognize and find since it doesn't occur all the time
 - Problem isn't inherently ill-conditioned
- Alternative formula is algebraically equivalent but more robust computationally
 - Where to find such formulas?
 - How to you know you need one?
- Be careful; have some way to check the answer

9 What is **strictfp**

- Java method and class qualifier
- Indicates floating-point computations must get *exactly* reproducible results
- Without **strictfp**, some variation is allowed
 - Intermediate results can have extended exponent range
 - Only makes a difference if an overflow or underflow would occur
- Only need to use **strictfp** if you want exactly reproducible results

10 Why multiple math libraries?

- **Math** and **StrictMath** classes
 - **StrictMath reproducibility**: methods must use a particular implementation algorithm, FDLIBM
 - **Math performance**: any algorithms meeting quality of implementation criteria
 - accuracy
 - monotonicity
- (Not directly related to **strictfp**)

C99

- Current C standard
 - Allows support for IEEE 754
 - low-level programming interface
 - Not yet widely implemented
 - gcc does *not* provide full floating-point support
- Introduced hexadecimal floating-point literals

754R

- Venerable 754 standard has undergone revision!
<http://grouper.ieee.org/groups/754/>
 - Decimal formats and arithmetic
 - Fused multiply add operation
 - More explicit conceptual model of levels of specification
 - Hexadecimal strings for binary floating-point values
 - Annexes on expression evaluation, alternate exception handling, and transcendental functions
- Balloting signup open until October 21, 2006
<http://754r.ucbtest.org/balloting.txt>

Decimal Floating-Point

- Not a panacea
 - Field axioms still fail to hold
- Worse rounding properties
 - distance between adjacent numbers goes up 10X on exponent increment
- Fixed point notions
 - not all values normalized (*cohorts*)
 - 1.0 *not* the same as 1.00
 - $10 \times 10^{-1} \not\equiv 100 \times 10^{-2}$
- Similar to **java.math.BigDecimal**

Summary

- Floating-point arithmetic only approximates real arithmetic
 - Floating-point approximation is predictable
- Avoid surprises from base conversion
- Understand when exact results should be expected

More information

- *Professor Kahan's webpages*,
<http://www.cs.berkeley.edu/~wkahan>
- *What Every Computer Scientist Should Know About Floating Point Arithmetic*, David Goldberg,
(with commentary by Doug Priest)
<http://www.validlab.com/goldberg/paper.pdf>
- *Numerical Computing with IEEE Floating Point Arithmetic*,
Michael L. Overton,
- *Floating-Point Fallacies*, Dan Zuras,
<http://www.zuras.net/FPFallacies.html>
- *The Art of Computer Programming, Vol. 2, Seminumerical Algorithms*, Donald Knuth

High-Precision References

- David H. Bailey, Multiprecision Software Directory, <http://www.nersc.gov/~dhbailey/mpdist/mpdist.html>
- T. J. Dekker, *A Floating-Point Technique for Extending the Available Precision*, Numerische Mathematik, vol. 18, 1971, pp. 224-242.
- Doug Priest, *On Properties of Floating-Point Arithmetics: Numerical Stability and the Cost of Accurate Computations*, Ph.D. thesis, UC Berkeley, November 1992
<ftp://ftp.icsi.berkeley.edu/pub/theory/priest-thesis.ps.Z>
- Jonathan Shewchuk, *Adaptive Precision Floating-Point Arithmetic and Fast Geometric Predicates*, CMU-CS-96-140, <http://www.cs.cmu.edu/~quake-papers/robust-arithmetic.ps>

Questions?

joe.darcy@sun.com
<http://blogs.sun.com/darcy>

Backup Slides

Java™ Programming Language

Floating-Point

- Required use of IEEE 754 numbers
 - Subnormals must be supported
 - Flush-to-zero not allowed
- Correctly rounded decimal \leftrightarrow binary conversion
- Well-defined expression evaluation rules
 - Yields predictable results
 - Code semantics depend on source, *not* on selection of compiler flags

C and Fortran Comparison

- C and FORTRAN compilers have been around longer
- The Java™ programming language has tighter semantics than C or FORTRAN
 - Can't "optimize" floating-point as much
 - Can't assume arrays aren't aliased
 - Can't make use of unspecified order of operations or undefined behavior
- Different languages have different goals

Multiplication — Background

- From algebra
 - FOIL – first, outer, inner last
 - $(a + b)(c + b) = ac + ab + bc + bd$
- If target format has at least twice the precision of starting format, multiplication is exact; e.g.
 - ```
float f1, f2;
double d1, d2;
d2 = (double)f1 * (double)f2; // exact
```
- Try to split two double numbers into two pieces and add up four resulting partial products

# How do to a split

```
c = (2s + 1) * a; // c = a*2s + a
a_big = c - a; // high-order bits of c
a_hi = c - a_big; // low-order bits of c, also
 // the high-order bits of a
a_lo = a - a_hi; // low order bits of a
// (a_hi + a_lo) equals a exactly
```

- Calculations of `a_hi` and `a_lo` are exact

# How to do an exact multiply

```
x = a * b;
a_hi = split(a, HIGH); a_lo = split(a, LOW);
b_hi = split(b, HIGH); b_lo = split(b, LOW);

err1 = x - (a_hi * b_hi); // First
err2 = err1 - (a_lo * b_hi); // inner
err3 = err2 - (a_hi * b_lo); // outer
y = (a_lo * b_lo) - err3; // last
// x + y == a * b exactly
```

- Calculation of `errn` and `y` are all done exactly

# Field Axioms

| Field Axiom over Reals                     | Example                                                   | Status of IEEE floating point                                                                                                                                                                                                                                                                                                                                        |
|--------------------------------------------|-----------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Quasiring Properties</b>                |                                                           |                                                                                                                                                                                                                                                                                                                                                                      |
| Closed under addition                      |                                                           | <i>true</i> , if $\infty$ and NaN are included                                                                                                                                                                                                                                                                                                                       |
| Associative addition                       | $(a+b) + c = a + (b+c)$                                   | <ul style="list-style-type: none"> <li><math>(\Omega + \Omega) + -\Omega = \infty \neq \Omega + (\Omega + -\Omega) = \Omega</math></li> <li><math>(1.0 + \delta) + \delta = 1.0 \neq 1.0 + (\delta + \delta) &gt; 1.0</math></li> <li><math>(\infty + -\infty) + \text{NaN}</math> signals invalid, <math>\infty + (-\infty + \text{NaN})</math> does not</li> </ul> |
| Identify element for addition              | $\forall a, a + 0 = 0 + a = a$                            | <i>false</i><br>if $a$ is $-0$ , $a + (+0) = +0$ , $+0$ is distinguishable from $-0$                                                                                                                                                                                                                                                                                 |
| Closed under multiplication                |                                                           | <i>true</i> , if $\infty$ and NaN are included                                                                                                                                                                                                                                                                                                                       |
| Associative multiplication                 | $(a*b)*c = a*(b*c)$                                       | <i>false</i> <ul style="list-style-type: none"> <li>roundoff and loss of precision on underflow</li> <li><math>(\Omega * 2.0) * (0.5) = \infty \neq \Omega * (2.0 * 0.5) = \Omega</math></li> <li><math>(\infty * 0.0) * \text{NaN}</math> signals invalid, <math>\infty * (0.0 * \text{NaN})</math> does not</li> </ul>                                             |
| <i>Identity element for multiplication</i> | $\forall a, a*1 = 1*a = a$                                | <i>true</i>                                                                                                                                                                                                                                                                                                                                                          |
| Zero annihilator                           | $\forall a, a*0 = 0*a = 0$                                | <i>false</i> <ul style="list-style-type: none"> <li><math>0 * \text{NaN}</math> and <math>\text{NaN} * 0</math> are NaN, NaN is not 0</li> <li><math>0 * \infty</math> and <math>\infty * 0</math> are NaN, NaN is not 0</li> </ul>                                                                                                                                  |
| <i>Commutative addition</i>                | $\forall a \forall b, a + b = b + a$                      | <i>true</i>                                                                                                                                                                                                                                                                                                                                                          |
| Distributivity                             | $a*(b + c) = a*b + a*c$<br>and<br>$(b + c)*a = b*c + c*a$ | <i>false</i> <ul style="list-style-type: none"> <li>roundoff</li> <li>overflow/underflow thresholds</li> <li>differences signaling invalid with <math>\infty</math>, NaN, and 0.0</li> </ul>                                                                                                                                                                         |

# More Field Axioms

| Ring Properties                   |                                                                    |                                                                                                                                                                                                                                                                    |
|-----------------------------------|--------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Additive inverse                  | $\forall a \exists b,$<br>$a + b = b + a = 0$                      | <i>false</i> <ul style="list-style-type: none"> <li><math>\infty + -\infty</math> is NaN, NaN is not 0; <math>\infty + a = \infty, \infty \neq 0</math></li> <li>NaN + <math>a</math> is NaN and NaN + <math>\infty</math> is NaN, NaN is not 0</li> </ul>         |
| Field Properties                  |                                                                    |                                                                                                                                                                                                                                                                    |
| <i>Commutative multiplication</i> | $\forall a \forall b, a * b = b * a$                               | <i>true</i>                                                                                                                                                                                                                                                        |
| Multiplication inverse            | except for $a=0,$<br>$\forall a \exists b,$<br>$a * b = b * a = 1$ | <i>false</i> <ul style="list-style-type: none"> <li>many ordinary floating point values lack exact inverses</li> <li><math>\infty * 0</math> is NaN, <math>\infty * \text{NaN}</math> is NaN and <math>\forall a \neq 0, \infty * a = \pm \infty</math></li> </ul> |