Using Floating Point Without Losing Your Sanity

Don Clugston

Sociomantic Labs GmbH

May 2016
Sanity Checks

- Sanity Checks
Sanity Checks

- Sanity Checks
- A Crisis of Confidence
Sanity Checks

- Sanity Checks
- A Crisis of Confidence
- The specialists, "Numerical Analysts", are rare -- yet ordinary programmers need to use floating point
Sanity Checks

- Sanity Checks
- A Crisis of Confidence
- The specialists, "Numerical Analysts", are rare -- yet ordinary programmers need to use floating point
- It's more fun if you view it as magic
A Child’s Magic Trick

Think of a number ...
Double it
Add 8
Halve it
Take away the number you first thought of
And your answer is ...
An Adult’s Magic Trick

Think of a floating point number...

```c
float magic ( float x )
{
    return x + 35 - x;
}
```

.magic( 1000 ) == 35
.magic( 1_000_000_000 ) == 64
.magic( 5_000_000_000 ) == 0

"Catastrophic Cancellation"
An Adult’s Magic Trick

- Think of a floating point number...

```c
float magic( float x )
{
    return x + 35 - x;
}
```

- magic( 1000 ) == 35
An Adult’s Magic Trick

- Think of a floating point number...

```c
float magic ( float x )
{
    return x + 35 - x;
}
```

- `magic( 1000 ) == 35`
- `magic( 1_000_000_000 ) == 64`
An Adult’s Magic Trick

- Think of a floating point number...

```c
float magic ( float x )
{
    return x + 35 - x;
}
```

- magic( 1000 ) == 35
- magic( 1_000_000_000 ) == 64
- magic( 5_000_000_000 ) == 0
An Adult’s Magic Trick

- Think of a floating point number...

```c
float magic ( float x )
{
    return x + 35 - x;
}
```

- `magic( 1000 ) == 35`
- `magic( 1_000_000_000 ) == 64`
- `magic( 5_000_000_000 ) == 0`

- "Catastrophic Cancellation"
Why does this happen?

- A float is 32 bits wide. It can only store 4 billion different numbers. 1000000035 is not one of them.
Why does this happen?

- A float is 32 bits wide. It can only store 4 billion different numbers. 1000000035 is not one of them.
- 1000000000 and 1000000064 are the closest available numbers.
Why does this happen?

- A float is 32 bits wide. It can only store 4 billion different numbers. 1000000035 is not one of them.
- 1000000000 and 1000000064 are the closest available numbers.
- In float land, 1000000035 == 1000000064
Why does this happen?

- A float is 32 bits wide. It can only store 4 billion different numbers. 1000000035 is not one of them.
- 1000000000 and 1000000064 are the closest available numbers.
- In float land, 1000000035 == 1000000064
- **Putting the uncountably infinite real number line...**
Why does this happen?

- A float is 32 bits wide. It can only store 4 billion different numbers. 1000000035 is not one of them.
- 1000000000 and 1000000064 are the closest available numbers.
- In float land, 1000000035 == 1000000064.
- Putting the uncountably infinite real number line...
- ... into a 32 bit float.
Why does this happen?

- A float is 32 bits wide. It can only store 4 billion different numbers. 1000000035 is not one of them.
- 1000000000 and 1000000064 are the closest available numbers.
- In float land, 1000000035 == 1000000064.
- Putting the uncountably infinite real number line...
- ... into a 32 bit float.

*We're pulling a trillion rabbits out of a 32-bit hat*
Floating point is a conjuring trick

- Cannot exactly represent
Floating point is a conjuring trick

- Cannot exactly represent
- PI
Floating point is a conjuring trick

- Cannot exactly represent
- PI
- sqrt(2)
Floating point is a conjuring trick

- Cannot exactly represent
- PI
- sqrt(2)
- 0.1
Floating point is a conjuring trick

- Cannot exactly represent
- \( \pi \)
- \( \sqrt{2} \)
- 0.1

- Addition isn’t even associative
Floating point is a conjuring trick

- Cannot exactly represent
  - PI
  - sqrt(2)
  - 0.1

- Addition isn’t even associative
  - \((35 + 1000000000) - 1000000000 == 64\)
Floating point is a conjuring trick

- Cannot exactly represent
  - $\pi$
  - $\sqrt{2}$
  - 0.1

- Addition isn’t even associative
  - $(35 + 1000000000) - 1000000000 == 64$
  - $35 + (1000000000 - 1000000000) == 35$
Floating point is a conjuring trick

- Cannot exactly represent
  - PI
  - \( \text{sqrt}(2) \)
  - 0.1

- Addition isn’t even associative
  - \((35 + 1000000000) - 1000000000 = 64\)
  - \(35 + (1000000000 - 1000000000) = 35\)

- Why do we use such a grotesque, fraudulent type?
Floating point is a success story

- All modern engineering is based on floating point calculations
Floating point is a success story

- All modern engineering is based on floating point calculations
- Floating-point hardware is ubiquitous
Floating point is a success story

- All modern engineering is based on floating point calculations
- Floating-point hardware is ubiquitous
- Total GPU power exceeds CPU power
Floating point is a success story

- All modern engineering is based on floating point calculations
- Floating-point hardware is ubiquitous
- Total GPU power exceeds CPU power
- Despite being a horrendous approximation, 64 bit floating point is "good enough"
Two worlds

- The Mathematician’s World

- In reality we only have 4-10 bytes

- Sometimes we try too hard to stay in the Mathematician’s World

Using Floating Point Without Losing Your Sanity

Don Clugston
Two worlds

- The Mathematician’s World
- The uncountably infinite real number line
Two worlds

- The Mathematician’s World
- The uncountably infinite real number line
- The world where algebra works
Two worlds

- The Mathematician’s World
- The uncountably infinite real number line
- The world where algebra works

- The Magician’s World
Two worlds

- The Mathematician’s World
- The uncountably infinite real number line
- The world where algebra works

- The Magician’s World
- In reality we only have 4-10 bytes
Two worlds

- The Mathematician’s World
- The uncountably infinite real number line
- The world where algebra works

- The Magician’s World
- In reality we only have 4-10 bytes

**Sometimes we try too hard to stay in the Mathematician’s World**
3 Misconceptions

- BELIEF: Floating point arithmetic is "fuzzy", not deterministic
3 Misconceptions

- BELIEF: Floating point arithmetic is "fuzzy", not deterministic
- REALITY: Floats don’t obey normal algebra BUT they obey floating-point algebra
3 Misconceptions

- BELIEF: Floating point arithmetic is "fuzzy", not deterministic
- REALITY: Floats don’t obey normal algebra BUT they obey floating-point algebra
- Every exact int calculation is exact in double.
3 Misconceptions

- BELIEF: Floating point arithmetic is "fuzzy", not deterministic
- REALITY: Floats don’t obey normal algebra BUT they obey floating-point algebra
- Every exact int calculation is exact in double.
- In fact a double is a 54 bit int with a bonus
3 Misconceptions

- **BELIEF:** Floating point arithmetic is "fuzzy", not deterministic
- **REALITY:** Floats don’t obey normal algebra BUT they obey floating-point algebra
- Every exact int calculation is exact in double.
- In fact a double is a 54 bit int with a bonus

- **BELIEF:** "1000000064" means "every number between 1000000033 and 1000000095"
- **REALITY:** 1000000064 means 1000000064. 1000000033 is an alias for 1000000064
3 Misconceptions

- BELIEF: Floating point arithmetic is "fuzzy", not deterministic
- REALITY: Floats don’t obey normal algebra BUT they obey floating-point algebra
- Every exact int calculation is exact in double.
- In fact a double is a 54 bit int with a bonus

- BELIEF: "1000000064" means "every number between 1000000033 and 1000000095"
- REALITY: 1000000064 means 1000000064. 1000000033 is an alias for 1000000064
3 Misconceptions

- BELIEF: Floating point arithmetic is "fuzzy", not deterministic
- REALITY: Floats don’t obey normal algebra BUT they obey floating-point algebra

Every exact int calculation is exact in double.

In fact a double is a 54 bit int with a bonus

- BELIEF: "1000000064" means "every number between 1000000033 and 1000000095"
- REALITY: 1000000064 means 1000000064. 1000000033 is an alias for 1000000064

- BELIEF: Floating point is weird
3 Misconceptions

- BELIEF: Floating point arithmetic is "fuzzy", not deterministic
- REALITY: Floats don’t obey normal algebra BUT they obey floating-point algebra
- Every exact int calculation is exact in double.
- In fact a double is a 54 bit int with a bonus

- BELIEF: "1000000064" means "every number between 1000000033 and 1000000095"
- REALITY: 1000000064 means 1000000064. 1000000033 is an alias for 1000000064

- BELIEF: Floating point is weird
- REALITY: Most real-world measurements are similar
Floats are just ints with a scale

```c
struct float {
    bool sign;
    int mantissa;
    int exponent;
}

- mantissa * 2 ^^ exponent
```
Floats are just ints with a scale

```c
struct float {
    bool sign;
    int mantissa;
    int exponent;
}
```

- mantissa * 2 ^^ exponent
- If exponent is 0, it really is an integer
Floats are just ints with a scale

```c
struct float {
    bool sign;
    int mantissa;
    int exponent;
}
```

- mantissa \(*\ 2^{\text{exponent}}\)
- If exponent is 0, it really is an integer
- Most important property is the precision: the number of bits in the mantissa.
Floats are just ints with a scale

```c
struct float {
    bool sign;
    int mantissa;
    int exponent;
}
```

- $\text{mantissa} \times 2^{\text{exponent}}$
- If exponent is 0, it really is an integer
- Most important property is the precision: the number of bits in the mantissa.
- In D, `float.mant_dig` gives the precision
The Precision Budget

*The larger the precision, the more extravagant you can be*

<table>
<thead>
<tr>
<th>Float</th>
<th>22 bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>Double</td>
<td>54 bits</td>
</tr>
<tr>
<td>Real</td>
<td>64 bits</td>
</tr>
<tr>
<td>Quadruple</td>
<td>112 bits</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Operation</th>
<th>Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multiplication</td>
<td>1 bit</td>
</tr>
<tr>
<td>Division</td>
<td>1 bit</td>
</tr>
<tr>
<td>Addition</td>
<td>Many</td>
</tr>
<tr>
<td>Take away the number you first thought of</td>
<td>Bankrupt</td>
</tr>
</tbody>
</table>
The Funny Values

-0.0 exists, though it almost always means +0.0
The Funny Values

- -0.0 exists, though it almost always means +0.0
- It exists because tiny numbers get rounded to zero
The Funny Values

-0.0 exists, though it almost always means +0.0
It exists because tiny numbers get rounded to zero

If exponent is int.max, value is infinity or "Not a Number" (nan)
The Funny Values

-0.0 exists, though it almost always means +0.0
It exists because tiny numbers get rounded to zero
If exponent is int.max, value is infinity or "Not a Number" (nan)

Infinity happens when:
The Funny Values

- -0.0 exists, though it almost always means +0.0
- It exists because tiny numbers get rounded to zero
- If exponent is int.max, value is infinity or "Not a Number" (nan)
- Infinity happens when:
  - Overflow: double.max * 2 == double.infinity
The Funny Values

- -0.0 exists, though it almost always means +0.0
- It exists because tiny numbers get rounded to zero
- If exponent is int.max, value is infinity or "Not a Number" (nan)
- Infinity happens when:
  - Overflow: double.max * 2 == double.infinity
  - Division by zero: 1.0 / 0.0 == double.infinity
The Funny Values

-0.0 exists, though it almost always means +0.0
It exists because tiny numbers get rounded to zero

If exponent is int.max, value is infinity or "Not a Number" (nan)

Infinity happens when:
Overflow: double.max * 2 == double.infinity
Division by zero: 1.0 / 0.0 == double.infinity
NaN happens when something evil happens
The Funny Values

-0.0 exists, though it almost always means +0.0

It exists because tiny numbers get rounded to zero

If exponent is int.max, value is infinity or "Not a Number" (nan)

Infinity happens when:

Overflow: double.max * 2 == double.infinity

Division by zero: 1.0 / 0.0 == double.infinity

NaN happens when something evil happens

double.infinity - double.infinity is double.nan
The Funny Values

- 0.0 exists, though it almost always means +0.0
- It exists because tiny numbers get rounded to zero
- If exponent is int.max, value is infinity or "Not a Number" (nan)

Infinity happens when:
- Overflow: double.max * 2 == double.infinity
- Division by zero: 1.0 / 0.0 == double.infinity

NaN happens when something evil happens
- double.infinity - double.infinity is double.nan
- 0.0 / 0.0 is double.nan
Floating Point Exceptions

- The hardware can generate hardware traps when funny values are produced. Most programs should enable the severe traps inside main()

FloatingPointControl fpctrl;

// Enable hardware exceptions for division by zero,
// overflow to infinity, and invalid operations
fpctrl.enableExceptions(FloatingPointControl.severeExceptions);
Floating Point Exceptions

- The hardware can generate hardware traps when funny values are produced. Most programs should enable the severe traps inside main()

```cpp
FloatingPointControl fpctrl;
// Enable hardware exceptions for division by zero,
// overflow to infinity, and invalid operations
fpctrl.enableExceptions(FloatingPointControl.severeExceptions);
```

- Unfortunately there is no way to detect Catastrophic Cancellation
"Don’t use =="

- Why not? Because it destroys the illusion
"Don't use =="

- Why not? Because it destroys the illusion
- "x == y" really means:
  x and y are equal to as many significant figures as the CPU supports
"Don’t use =="

- Why not? Because it destroys the illusion
- "x == y" really means:
  - x and y are equal to as many significant figures as the CPU supports
- Exposes the implementation
"Don’t use =="

- Why not? Because it destroys the illusion
- "x == y" really means:
  x and y are equal to as many significant figures as the CPU supports
- Exposes the implementation
- But +0.0 == -0.0
"Don’t use =="

- Why not? Because it destroys the illusion
- "x == y" really means:
  x and y are equal to as many significant figures as the CPU supports
- Exposes the implementation
- But +0.0 == -0.0
- The horror: NaN != NaN
"Don’t use =="

- Why not? Because it destroys the illusion
- "x == y" really means:
  x and y are equal to as many significant figures as the CPU supports
- Exposes the implementation
- But +0.0 == -0.0
- The horror: NaN != NaN
- Some implementation details are hidden
"Don’t use =="

- Why not? Because it destroys the illusion
- "x == y" really means:
  - x and y are equal to as many significant figures as the CPU supports
- Exposes the implementation

- But +0.0 == -0.0
- The horror: NaN != NaN
- Some implementation details are hidden

- == is still useful for low-level code and unit tests.
Alternatives to ==

- In D, "x is y" compares implementation, no tricks
Alternatives to `==`

- In D, "x is y" compares implementation, no tricks
- Can we create a 'better =='?
Alternatives to `==`

- In D, "x is y" compares implementation, no tricks
- Can we create a 'better ==' ?
- No :(
Alternatives to `==`

- In D, "x is y" compares implementation, no tricks
- Can we create a 'better =='? 
- No :( 
- Reduce the number of bits that must be equal
Alternatives to ==

- In D, "x is y" compares implementation, no tricks
- Can we create a 'better =='?  
- No :(

- Reduce the number of bits that must be equal
- std.math.feqrel gives number of equal bits
Alternatives to \( == \)

- In D, "x is y" compares implementation, no tricks
- Can we create a 'better =='?
- No :(  
- Reduce the number of bits that must be equal
- std.math.feqrel gives number of equal bits
- How many must be equal? Arbitrary!
Alternatives to \( == \)

- In D, "x is y" compares implementation, no tricks
- Can we create a 'better =='?
- No :(  

Reduce the number of bits that must be equal

std.math.feqrel gives number of equal bits

How many must be equal? Arbitrary!

In Physics, there is no "exact equality" either
Alternatives to ==

- In D, "x is y" compares implementation, no tricks
- Can we create a 'better =='? No :( 
- Reduce the number of bits that must be equal
- std.math.feqrel gives number of equal bits
- How many must be equal? Arbitrary!
- In Physics, there is no "exact equality" either
- Always need to specify the precision
How D Makes It Better

- Standard IEEE arithmetic, bizarro implementations are forbidden
How D Makes It Better

- Standard IEEE arithmetic, bizarro implementations are forbidden
- Built-in floating point properties
How D Makes It Better

- Standard IEEE arithmetic, bizarro implementations are forbidden
- Built-in floating point properties
- `max`, `epsilon`, `mant_dig`, `infinity`, `nan`...
How D Makes It Better

- Standard IEEE arithmetic, bizarro implementations are forbidden
- Built-in floating point properties
- max, epsilon, mant_dig, infinity, nan ...
- Unit tests
How D Makes It Better

- Standard IEEE arithmetic, bizarro implementations are forbidden
- Built-in floating point properties
  - max, epsilon, mant_dig, infinity, nan ...
- Unit tests
- static if
How D Makes It Better

- Standard IEEE arithmetic, bizarro implementations are forbidden
- Built-in floating point properties
  - max, epsilon, mant_dig, infinity, nan ...
- Unit tests
- static if

- But sometimes we have Orwellian experiences...
Some Numerals Are More Equal Than Others

- float x = 1.30;
Some Numerals Are More Equal Than Others

- float x = 1.30;
- assert(x == 1.30); // FAILS!!
Some Numerals Are More Equal Than Others

- float x = 1.30;
- assert( x == 1.30 ); // FAILS!!
- assert( x == 1.30f ); // OK
Some Numerals Are More Equal Than Others

- float x = 1.30;
- assert( x == 1.30 ); // FAILS!!
- assert( x == 1.30f ); // OK

- double y = 1.30;
Some Numerals Are More Equal Than Others

- float x = 1.30;
- assert( x == 1.30 ); // FAILS!!
- assert( x == 1.30f ); // OK

- double y = 1.30;
- assert( y == 1.30 ); // OK
Some Numerals Are More Equal Than Others

- float x = 1.30;
- assert( x == 1.30 ); // FAILS!!
- assert( x == 1.30f ); // OK

- double y = 1.30;
- assert( y == 1.30 ); // OK
- assert( y == 1.30f ); // OK?!!!!
Some Numerals Are More Equal Than Others

- float x = 1.30;
- assert( x == 1.30 ); // FAILS!!
- assert( x == 1.30f ); // OK

- double y = 1.30;
- assert( y == 1.30 ); // OK
- assert( y == 1.30f ); // OK?!!!!

- assert( y == x ); // FAILS
Some Numerals Are More Equal Than Others

- float x = 1.30;
- assert( x == 1.30 ); // FAILS!!
- assert( x == 1.30f ); // OK

- double y = 1.30;
- assert( y == 1.30 ); // OK
- assert( y == 1.30f ); // OK?!!!!

- assert( y == x ); // FAILS
- assert( 1.30 == 1.30f ); // OK!!
Sociomantic’s Nine Trillion Dollar Bug

- Losing your sanity, #1

```c
if ( price < 0 ) { error(); }
if ( price ) {
    bid( lround( price ) );
}
```

In an auction, we made a bid of $9,223,372,036,855 DMD.

Issue #13489 - never do an implicit cast from float to bool unless you can guarantee it is not NaN.
Sociomantic’s Nine Trillion Dollar Bug

- Losing your sanity, #1

```c
if ( price < 0 ) { error(); }
if ( price ) {
    bid( lround( price ) );
}
```

- price was NaN
Sociomantic’s Nine Trillion Dollar Bug

- Losing your sanity, #1

```c
if ( price < 0 ) { error(); }
if ( price ) {
    bid( lround( price ) );
}
```

- price was NaN
- In an auction, we made a bid of $9223372036855
Sociomantic’s Nine Trillion Dollar Bug

- Losing your sanity, #1

```java
if ( price < 0 ) { error(); }
if ( price ) {
    bid( lround( price ) );
}
```

- price was NaN
- In an auction, we made a bid of $9223372036855
- DMD Issue #13489 - never do an implicit cast from float to bool unless you can guarantee it is not NaN.
Mathematically, reals are an extension of integers
Generic Programming

- Mathematically, reals are an extension of integers
- `int` and `float` both have hardware support
Generic Programming

- Mathematically, reals are an extension of integers
- int and float both have hardware support
- Replace 'int' with 'double' and everything will compile
Generic Programming

- Mathematically, reals are an extension of integers
- int and float both have hardware support
- Replace 'int' with 'double' and everything will compile
- Test cases will still work
Generic Programming

- Mathematically, reals are an extension of integers
- Int and float both have hardware support
- Replace 'int' with 'double' and everything will compile
- Test cases will still work
- So let's make our code work with any numeric type!
"Any Numeric Type" is a Bad Idea

- The code will compile, but it will be wrong for floats
"Any Numeric Type" is a Bad Idea

- The code will compile, but it will be wrong for floats
- The problem: Floats are a conjuring trick
"Any Numeric Type" is a Bad Idea

- The code will compile, but it will be wrong for floats
- The problem: Floats are a conjuring trick
- Floats are not a subset of mathematical reals. Floats are not a superset of int.
"Any Numeric Type" is a Bad Idea

- The code will compile, but it will be wrong for floats
- The problem: Floats are a conjuring trick
- Floats are not a subset of mathematical reals. Floats are not a superset of int.

- The VALUES are a superset of int
"Any Numeric Type" is a Bad Idea

- The code will compile, but it will be wrong for floats
- The problem: Floats are a conjuring trick
- Floats are not a subset of mathematical reals. Floats are not a superset of int.

- The VALUES are a superset of int
- The SEMANTICS are not
"Any Numeric Type" is a Bad Idea

- The code will compile, but it will be wrong for floats
- The problem: Floats are a conjuring trick
- Floats are not a subset of mathematical reals. Floats are not a superset of int.

- The VALUES are a superset of int
- The SEMANTICS are not

- For generic code we need common semantics
Losing Your Sanity, #2

- A simple foreach range

```c
int doTen ( T )( T from )
{
    int howmany = 0;
    foreach (x; from .. from + 10)
        ++howmany;
    return howmany;
}
```

```c
// Examples
int doTen!float( 500 ) == 10
int doTen!float( 16777242 ) == 9
int doTen!float( 18000000 ) does not terminate
```
Losing Your Sanity, #2

- A simple foreach range

```c
int doTen ( T )( T from )
{
    int howmany = 0;
    foreach (x; from .. from + 10)
        ++howmany;
    return howmany;
}
```

doTen!float( 500 ) == 10
Losing Your Sanity, #2

- A simple foreach range

```c
int doTen ( T )( T from )
{
    int howmany = 0;
    foreach (x; from .. from + 10)
        ++howmany;
    return howmany;
}
```

- `doTen!float( 500 ) == 10`
- `doTen!float( 16777242 ) == 9`
Losing Your Sanity, #2

- A simple foreach range

```c
int doTen ( T ) ( T from )
{
    int howmany = 0;
    foreach (x; from .. from + 10)
        ++howmany;
    return howmany;
}
```

- `doTen(float( 500 ) == 10`
- `doTen(float( 16777242 ) == 9`
- `doTen(float( 18000000 ) does not terminate`

Using Floating Point Without Losing Your Sanity
Increment (or not)

For integers, ++x; --x; is a no-op
For floats it’s more fun

<table>
<thead>
<tr>
<th>x</th>
<th>After ++x; --x;</th>
</tr>
</thead>
<tbody>
<tr>
<td>31837</td>
<td>31837</td>
</tr>
<tr>
<td>1.25e-6</td>
<td>1.20e-6</td>
</tr>
<tr>
<td>-1e-20</td>
<td>0</td>
</tr>
<tr>
<td>16777250</td>
<td>16777252</td>
</tr>
</tbody>
</table>

If you use ++ on a float, someone will go insane.
isNumeric() in Phobos

- All uses of isNumeric() are trivial, except two
isNumeric() in Phobos

- All uses of isNumeric() are trivial, except two
- std.complex just casts integers to floating point
isNumeric() in Phobos

- All uses of isNumeric() are trivial, except two
- std.complex just casts integers to floating point
- std.random.dice() is incorrect for pathological cases
isNumeric() in Phobos

- All uses of isNumeric() are trivial, except two
- std.complex just casts integers to floating point
- std.random.dice() is incorrect for pathological cases
- There are probably no mathematical algorithms that work for both integers and floating point
"More Precision Is Always Better"

- More precision improves the illusion.

double magic ( double x )
{
    return x + 35 - x;
}

magic(1000000000) == 35
magic(5e17) == 64

Corner cases move but don't disappear
"More Precision Is Always Better"

- More precision improves the illusion.

```c
double magic ( double x )
{
    return x + 35 - x;
}
```

- `magic(1000000000) == 35`
- `magic(5e17) == 64`

Corner cases move but don't disappear
"More Precision Is Always Better"

- More precision improves the illusion.

```c
double magic ( double x )
{
    return x + 35 - x;
}
```

- `magic(1000000000) == 35`
- `magic(5e17) == 64`
"More Precision Is Always Better"

- More precision improves the illusion.

```
double magic ( double x )
{
    return x + 35 - x;
}
```

- magic(1000000000) == 35
- magic(5e17) == 64
- Corner cases move but don’t disappear
## Rounding Modes

<table>
<thead>
<tr>
<th>Rounding Mode</th>
<th>2.5</th>
<th>-5.5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Round to Nearest</td>
<td>2</td>
<td>-6</td>
</tr>
<tr>
<td>Round Up</td>
<td>3</td>
<td>-5</td>
</tr>
<tr>
<td>Round Down</td>
<td>2</td>
<td>-6</td>
</tr>
<tr>
<td>Round To Zero</td>
<td>2</td>
<td>-5</td>
</tr>
</tbody>
</table>
"More Precision Is Always Better"

Algorithms should be written to work based on the minimum precision of the calculation. They should not degrade or fail if the actual precision is greater. -- The D Spec

• Unfortunately this is not generally possible
"More Precision Is Always Better"

Algorithms should be written to work based on the minimum precision of the calculation. They should not degrade or fail if the actual precision is greater. -- The D Spec

- Unfortunately this is not generally possible

- Double rounding is a problem.
"More Precision Is Always Better"

Algorithms should be written to work based on the minimum precision of the calculation. They should not degrade or fail if the actual precision is greater. -- The D Spec

- Unfortunately this is not generally possible
- Double rounding is a problem.
- 3.49 rounds down to 3
"More Precision Is Always Better"

Algorithms should be written to work based on the minimum precision of the calculation. They should not degrade or fail if the actual precision is greater. -- The D Spec

- Unfortunately this is not generally possible
- Double rounding is a problem.
- 3.49 rounds down to 3
- 3.49 rounds up to 3.5, which rounds up to 4
Secret Precision

Extra hidden precision can happen when:

- The x87 FPU is used on x86 machines
- Processors support FMA (PPC, recent x86_64, Itanium...)
- If we do float calculations at double precision
  float (22 bits) * float == 44 bits precision
  double has 54 bits. So no rounding happens! We're OK.
- If we do double calculations at real precision:
  double (54 bits) * double == 118 bits precision
  real only has 64 bits. We'll round twice.
  One in 1024 calculations has an out-by-1 error
Secret Precision

- Extra hidden precision can happen when:
  - The x87 FPU is used on x86 machines
Secret Precision

- Extra hidden precision can happen when:
- The x87 FPU is used on x86 machines
- Processors support FMA (PPC, recent x86_64, Itanium...)

If we do float calculations at double precision:
- `float (22 bits) * float == 44 bits precision`
- `double has 54 bits. So no rounding happens! We're OK.`

If we do double calculations at real precision:
- `double (54 bits) * double == 118 bits precision`
- `real only has 64 bits. We'll round twice.
  One in 1024 calculations has an out-by-1 error`
Secret Precision

- Extra hidden precision can happen when:
- The x87 FPU is used on x86 machines
- Processors support FMA (PPC, recent x86_64, Itanium...)
- If we do float calculations at double precision

```plaintext
float (22 bits) * float == 44 bits precision

double has 54 bits. So no rounding happens! We're OK.

If we do double calculations at real precision:

double (54 bits) * double == 118 bits precision
real only has 64 bits. We'll round twice.

One in 1024 calculations has an out-by-1 error
```
Secret Precision

- Extra hidden precision can happen when:
- The x87 FPU is used on x86 machines
- Processors support FMA (PPC, recent x86_64, Itanium...)
- If we do float calculations at double precision
- \( \text{float (22 bits)} \times \text{float} \equiv \text{44 bits precision} \)
Secret Precision

- Extra hidden precision can happen when:
  - The x87 FPU is used on x86 machines
  - Processors support FMA (PPC, recent x86_64, Itanium...)

- If we do float calculations at double precision
  - float (22 bits) * float == 44 bits precision
  - double has 54 bits. So no rounding happens! We're OK.
Secret Precision

- Extra hidden precision can happen when:
  - The x87 FPU is used on x86 machines
  - Processors support FMA (PPC, recent x86_64, Itanium...)
  - If we do float calculations at double precision
    - float (22 bits) * float == 44 bits precision
    - double has 54 bits. So no rounding happens! We’re OK.
  - If we do double calculations at real precision:
Secret Precision

- Extra hidden precision can happen when:
  - The x87 FPU is used on x86 machines
  - Processors support FMA (PPC, recent x86_64, Itanium...)

- If we do float calculations at double precision
  - float (22 bits) * float == 44 bits precision
  - double has 54 bits. So no rounding happens! We’re OK.

- If we do double calculations at real precision:
  - double (54 bits) * double == 118 bits precision
Secret Precision

- Extra hidden precision can happen when:
  - The x87 FPU is used on x86 machines
  - Processors support FMA (PPC, recent x86_64, Itanium...)

- If we do float calculations at double precision
  - float (22 bits) * float == 44 bits precision
  - double has 54 bits. So no rounding happens! We’re OK.

- If we do double calculations at real precision:
  - double (54 bits) * double == 118 bits precision
  - real only has 64 bits. We’ll round twice.
Secret Precision

- Extra hidden precision can happen when:
  - The x87 FPU is used on x86 machines
  - Processors support FMA (PPC, recent x86_64, Itanium...)

- If we do float calculations at double precision:
  float (22 bits) * float == 44 bits precision
  double has 54 bits. So no rounding happens! We’re OK.

- If we do double calculations at real precision:
  double (54 bits) * double == 118 bits precision
  real only has 64 bits. We’ll round twice.

- One in 1024 calculations has an out-by-1 error
Most library code splits the possible input values into smaller ranges, and then performs a different calculation for each range.
Root finding

Given a function

\[
double f( double x ), f( x0 ) > 0, f( x1 ) < 0
\]

find the point where \( f(x) = 0 \)

- State Of The Art: TOMS 748. Inverse cubic polynomial fitting.
Given a function

```c
double f( double x ), f( x0 ) > 0, f( x1 ) < 0
```

find the point where \( f(x) = 0 \)

- State Of The Art: TOMS 748. Inverse cubic polynomial fitting.
- Every iteration triples number of known bits. Best case 5 calls to \( f(x) \)
Root finding

Given a function

\[
\text{double } f(\text{ double } x), f( x0 ) > 0, f( x1 ) < 0
\]

find the point where \( f(x) == 0 \)

- State Of The Art: TOMS 748. Inverse cubic polynomial fitting.
- Every iteration triples number of known bits. Best case 5 calls to \( f(x) \)
- If this fails, use binary chop. Gives one bit per iteration in the worst case.
Root finding

Given a function

\[ \text{double } f( \text{double } x ), \ f( x_0 ) > 0, \ f( x_1 ) < 0 \]

find the point where \( f(x) == 0 \)

- State Of The Art: TOMS 748. Inverse cubic polynomial fitting.
- Every iteration triples number of known bits. Best case 5 calls to \( f(x) \)
- If this fails, use binary chop. Gives one bit per iteration in the worst case.
- But \( x \Rightarrow x^3 \); takes 1830 calls to converge!
Given a function

double f( double x ), f( x0 ) > 0, f( x1 ) < 0

find the point where f(x) == 0

- State Of The Art: TOMS 748. Inverse cubic polynomial fitting.
- Every iteration triples number of known bits. Best case 5 calls to f(x)
- If this fails, use binary chop. Gives one bit per iteration in the worst case.
- But x => x*x*x; takes 1830 calls to converge!
- With 80-bit reals, worst case is > 16000 calls
The Binary Chop That Isn’t

\[
\text{auto midpoint} = \frac{x_0 + x_1}{2};
\]

- Let \( x_0 = 1e100, \ x_1 = 1e-100 \), and ultimate solution is \( 2e-100 \)
The Binary Chop That Isn’t

\[ auto\ midpoint = (x_0 + x_1) / 2; \]

- Let \( x_0 = 1e100, x_1 = 1e-100 \), and ultimate solution is \( 2e-100 \)
- Midpoints are \( 5e99, 2.5e99, 1.2e99, 6e98, \ldots \)
The Binary Chop That Isn’t

\[ \text{auto \ midpoint} = (x_0 + x_1) / 2; \]

- Let \( x_0 = 1\times10^{100}, x_1 = 1\times10^{-100} \), and ultimate solution is \( 2\times10^{-100} \)
- Midpoints are \( 5\times10^{99}, 2.5\times10^{99}, 1.2\times10^{99}, 6\times10^{98}, \ldots \)
- We get to \( 2\times10^{-100} \) after 600 iterations
Binary Chop For Real

- Midpoint in implementation space

ulong x0_raw = reinterpret!ulong(x0);
ulong x1_raw = reinterpret!ulong(x1);
auto midpoint = reinterpret!double( x0_raw + x1_raw ) / 2;
Binary Chop For Real

- Midpoint in implementation space

ulong x0_raw = reinterpret!ulong(x0);
ulong x1_raw = reinterpret!ulong(x1);
auto midpoint = reinterpret!double( x0_raw + x1_raw ) / 2;

- Again let x0 == 1e100, x1 = 1e-100, and solution is 2e-100
Binary Chop For Real

- Midpoint in implementation space

ulong x0_raw = reinterpret!ulong(x0);
ulong x1_raw = reinterpret!ulong(x1);
auto midpoint = reinterpret!double( x0_raw + x1_raw ) / 2;

- Again let x0 == 1e100, x1 = 1e-100, and solution is 2e-100
- Midpoints are 5e0, 2.5e-50, 1.2e-75, 6e-88, 3e-94 ...
Binary Chop For Real

- Midpoint in implementation space

```c
ulong x0_raw = reinterpret!ulong(x0);
ulong x1_raw = reinterpret!ulong(x1);
auto midpoint = reinterpret!double( x0_raw + x1_raw ) / 2;
```

- Again let x0 == 1e100, x1 = 1e-100, and solution is 2e-100
- Midpoints are 5e0, 2.5e-50, 1.2e-75, 6e-88, 3e-94 ...
- We reach 2e-100 after 9 iterations
Performance Impact

- For 80 bit reals, worst case improves from 16000 calls, to about 150.
Performance Impact

- For 80 bit reals, worst case improves from 16000 calls, to about 150.
- TOMS 748 has a similar problem with linear interpolation.
Performance Impact

- For 80 bit reals, worst case improves from 16000 calls, to about 150.
- TOMS 748 has a similar problem with linear interpolation
- Fixing that improves the average case as well.
Performance Impact

- For 80 bit reals, worst case improves from 16000 calls, to about 150.
- TOMS 748 has a similar problem with linear interpolation.
- Fixing that improves the average case as well.
- Available in std.numeric.findRoot
Moral

*Even when floating point code compiles, and gives the mathematically correct answer, it can still be algorithmically wrong*
Summary

- Floating point is a trick created for engineers, not mathematicians.
Summary

- Floating point is a trick created for engineers, not mathematicians.
- "Take away the number you first thought of" destroys the illusion.
Summary

- Floating point is a trick created for engineers, not mathematicians.
- "Take away the number you first thought of" destroys the illusion
- More precision improves the illusion, but corner cases remain
Summary

- Floating point is a trick created for engineers, not mathematicians.
- "Take away the number you first thought of" destroys the illusion.
- More precision improves the illusion, but corner cases remain.
- float requires great care. Prefer double or real.
Summary

- Floating point is a trick created for engineers, not mathematicians.
- "Take away the number you first thought of" destroys the illusion.
- More precision improves the illusion, but corner cases remain.
- float requires great care. Prefer double or real.
- Use == only when you want to expose implementation details.
Summary

- Floating point is a trick created for engineers, not mathematicians.
- "Take away the number you first thought of" destroys the illusion.
- More precision improves the illusion, but corner cases remain.
- float requires great care. Prefer double or real.
- Use == only when you want to expose implementation details.
- Generic numeric code is almost certainly wrong in horrible, subtle ways.
Summary

- Floating point is a trick created for engineers, not mathematicians.
- "Take away the number you first thought of" destroys the illusion.
- More precision improves the illusion, but corner cases remain.
- `float` requires great care. Prefer `double` or `real`.
- Use `==` only when you want to expose implementation details.
- Generic numeric code is almost certainly wrong in horrible, subtle ways.
- D is (mostly) a pleasant language for floating point.