Computers & Arithmetic

By: Fahad Hossaini

Binary

Using Computers

Computers have incredible computational ability. In the modern day, they can do millions of operations within seconds.

Using Computers

Computers have incredible computational ability. In the modern day, they can do millions of operations within seconds.

It's clear why we want computers to be able to do math. Many of the algorithms we use on a daily basis rely on being able to do math effectively. AI is all math.

Using Computers

Computers have incredible computational ability. In the modern day, they can do millions of operations within seconds.

It's clear why we want computers to be able to do math. Many of the algorithms we use on a daily basis rely on being able to do math effectively. AI is all math.

But someone must have come up with how computers should even do math, or more particularly, arithmetic.

We know that computers speak in 0s and 1s. But why?

We know that computers speak in 0s and 1s. But why?

It's because on (1) and off (0) is easy to encode!

We know that computers speak in 0s and 1s. But why?

It's because on (1) and off (0) is easy to encode!

So, we need to understand binary first!

We'll represent numbers in binary. We'll just rapid fire a few numbers.

Note: I'll subscript binary numbers so we know they are in binary right now.

1_2

We'll represent numbers in binary. We'll just rapid fire a few numbers.

Note: I'll subscript binary numbers so we know they are in binary right now.

1_2

This is 1. Ok, what is 10_2?

We'll represent numbers in binary. We'll just rapid fire a few numbers.

Note: I'll subscript binary numbers so we know they are in binary right now.

1_2

This is 1. Ok, what is 10_2?

This is 2. Now, what is 11_2?

We'll represent numbers in binary. We'll just rapid fire a few numbers.

Note: I'll subscript binary numbers so we know they are in binary right now.

1_2

This is 1. Ok, what is 10_2?

This is 2. Now, what is 11_2?

This is 3. Ok, what about 101_2?

We'll represent numbers in binary. We'll just rapid fire a few numbers.

Note: I'll subscript binary numbers so we know they are in binary right now.

1_2

This is 1. Ok, what is 10_2?

This is 2. Now, what is 11_2?

This is 3. Ok, what about 101_2?

This is 5.

What Is Actually Happening?

In base 10, each decimal represents a multiple of 10. We interpret 567 as 5 copies of 100 plus 6 copies of 10 plus 7 copies of 1.

In base 2, or binary, our multiples are powers of 2 rather than powers of 10. So, 11001_2 is:

$$1 \cdot 16 + 1 \cdot 8 + 0 \cdot 4 + 0 \cdot 2 + 1 \cdot 1 = 16 + 8 + 1 = 25$$

What Is Actually Happening?

In base 10, each decimal represents a multiple of 10. We interpret 567 as 5 copies of 100 plus 6 copies of 10 plus 7 copies of 1.

In base 2, or binary, our multiples are powers of 2 rather than powers of 10. So, 11001_2 is:

$$1 \cdot 16 + 1 \cdot 8 + 0 \cdot 4 + 0 \cdot 2 + 1 \cdot 1 = 16 + 8 + 1 = 25$$

Indeed, addition and multiplication is done as expected: We carry digits, but instead of overflowing when we encounter a 10, we overflow when we encounter a 2. Indeed, $1_2 + 1_2 = 2_2 = 10_2$ as 2 can't exist in binary.

How is this relevant?

How do we represent decimal numbers?

How is this relevant?

How do we represent decimal numbers?

The exact same, but in reverse. Let’s see how base 10 works.

How is this relevant?

How do we represent decimal numbers?

The exact same, but in reverse. Let's see how base 10 works.

0.47 is $4 \cdot \frac{1}{10} + 7 \cdot \frac{1}{100} = \frac{47}{100}$.

Likewise, if we do decimal in binary, we have:

$1.1101 = 1 \cdot 1 + 1 \cdot \frac{1}{2} + 1 \cdot \frac{1}{4} + 0 \cdot \frac{1}{8} + 1 \cdot \frac{1}{16} = \frac{29}{16}$.

Target Practice

101.01 and 11.11. Compute this out!

A Good Number System

Finite Space

We want to represent any “reasonable” number in a finite amount of size. There are two main concerns.

Finite Space

We want to represent any “reasonable” number in a finite amount of size. There are two main concerns.

1: Accuracy. We want our number to be as accurate as possible. Indeed, the real numbers can only be approximated by rational number, and we are working with only rational numbers.

Finite Space

We want to represent any "reasonable" number in a finite amount of size. There are two main concerns.

1: Accuracy. We want our number to be as accurate as possible. Indeed, the real numbers can only be approximated by rational number, and we are working with only rational numbers.

2: Size. We should be able to represent big scales as well.

Finite Space

We want to represent any "reasonable" number in a finite amount of size. There are two main concerns.

1: Accuracy. We want our number to be as accurate as possible. Indeed, the real numbers can only be approximated by rational number, and we are working with only rational numbers.

2: Size. We should be able to represent big scales as well.

Let's say we want to do this in 32 bits, or 32 digits. What's the best way to do so?

Key Insights

There's one really big insight we want to notice: Accuracy only matters in certain cases.

Indeed, if the number is 13582983, we can be imprecise by quite a bit, since in most applications, we just need to know the number is 'big.' Indeed, the difference between 13582983 and 13582983.1 is a small relative error.

Key Insights

There's one really big insight we want to notice: Accuracy only matters in certain cases.

Indeed, if the number is 13582983, we can be imprecise by quite a bit, since in most applications, we just need to know the number is 'big.' Indeed, the difference between 13582983 and 13582983.1 is a small relative error.

But indeed, 0.1 and 0.2 is a massive difference, since we double/halve the quantity!

In formal language, while the absolute error is the same, we want to reduce the relative error.

Key Insights

There's one really big insight we want to notice: Accuracy only matters in certain cases.

Indeed, if the number is 13582983, we can be imprecise by quite a bit, since in most applications, we just need to know the number is 'big.' Indeed, the difference between 13582983 and 13582983.1 is a small relative error.

But indeed, 0.1 and 0.2 is a massive difference, since we double/halve the quantity!

In formal language, while the absolute error is the same, we want to reduce the relative error.

So, how does this insight help us?

Scientific Notation

Scientific notation does exactly what we need!

It can both represent small numbers and big numbers. For small numbers, we can get high precision. For big numbers, we can explain how large it is via the exponent.

Scientific Notation

Scientific notation does exactly what we need!

It can both represent small numbers and big numbers. For small numbers, we can get high precision. For big numbers, we can explain how large it is via the exponent.

So, once again, we have to convert what we know from our base-10 world into the binary world.

Scientific Notation

Scientific notation does exactly what we need!

It can both represent small numbers and big numbers. For small numbers, we can get high precision. For big numbers, we can explain how large it is via the exponent.

So, once again, we have to convert what we know from our base-10 world into the binary world.

Scientific notation is as follows: $c \cdot 10^n$ for some $1 \leq c < 10$ and any $n \in \mathbb{Z}$.

Scientific Notation

Scientific notation does exactly what we need!

It can both represent small numbers and big numbers. For small numbers, we can get high precision. For big numbers, we can explain how large it is via the exponent.

So, once again, we have to convert what we know from our base-10 world into the binary world.

Scientific notation is as follows: $c \cdot 10^n$ for some $1 \leq c < 10$ and any $n \in \mathbb{Z}$.

For binary, this will be $c \cdot 2^n$ for some $1 \leq c < 2$ and any $n \in \mathbb{Z}$.

A Secret Advantage of Binary

Interestingly, since $1 \leq c < 2$, we always know that c will be: $1.x_1x_2...$. So, we only need to know what the digits after the decimal looks like.

Bringing it All Together

A floating-point number, or a float, is a number composed of three parts.

1: The sign bit. We dedicate a whole bit to whether the number is positive (0) or a negative (1).

2: The exponent. We dedicate 8 bits to the exponent, and represent it in two's complement. i.e: $-126 \leq n \leq 127$.

3: The mantissa. We dedicate 23 bits to the decimal expansion for the most pristine accuracy.

Let's Represent a Number Please!

Let's try to write $x = -13421.625$ as a float.

Let's Represent a Number Please!

Let's try to write $x = -13421.625$ as a float.

First, we notice that we set the sign bit to 1, and convert this number to binary.

Indeed, $13421.625 = 11010001101101.101_2$ (17 digits).

Let's Represent a Number Please!

Let's try to write $x = -13421.625$ as a float.

First, we notice that we set the sign bit to 1, and convert this number to binary.

Indeed, $13421.625 = 11010001101101.101_2$ (17 digits).

Now, move that exponent! So, we get: $1.10...01 \cdot 2^{13}$. Thus, our exponent is: 00001101. You'd think.

Let's try to write $x = -13421.625$ as a float.

First, we notice that we set the sign bit to 1, and convert this number to binary.

Indeed, $13421.625 = 11010001101101.101_2$ (17 digits).

Now, move that exponent! So, we get: $1.10...01 \cdot 2^{13}$. Thus, our exponent is: 00001101. You'd think.

Due to implementation (which you can ask me about now or later depending on time), we have to add 127, or, 01111111. So, we'll just add 10000000 and subtract 1 to get: 10001100.

So, our final number is:

$$1 \quad 10001100 \quad 11010001101101101000000$$

An Exercise at Home

Write 5 as a float!

Analyzing Floating-Point Arithmetic

Python represents floating-point numbers in double-precision, or, 64 bits. In this case, we have 1 sign bit, 11 exponent bits and 52 mantissa bits (also called significand).

Python

Python represents floating-point numbers in double-precision, or, 64 bits. In this case, we have 1 sign bit, 11 exponent bits and 52 mantissa bits (also called significand).

From this point onwards, we will use Python and double precision to list out our numbers!

Differences in Base-2

There are a few weird quirks when working in base 2.

Differences in Base-2

There are a few weird quirks when working in base 2.

Firstly, notice that $\frac{3}{10} = 0.3$ in our decimal world.

Differences in Base-2

There are a few weird quirks when working in base 2.

Firstly, notice that $\frac{3}{10} = 0.3$ in our decimal world.

But in binary, it's $0.01001100110011\cdots$

Notice how this infinitely repeats?

Differences in Base-2

There are a few weird quirks when working in base 2.

Firstly, notice that $\frac{3}{10} = 0.3$ in our decimal world.

But in binary, it's $0.01001100110011 \cdots$

Notice how this infinitely repeats?

Just like how $\frac{1}{3} = 0.333 \cdots$, we can get numbers that are rationals but have an infinite decimal expansion.

Differences in Base-2

There are a few weird quirks when working in base 2.

Firstly, notice that $\frac{3}{10} = 0.3$ in our decimal world.

But in binary, it's $0.01001100110011\cdots$

Notice how this infinitely repeats?

Just like how $\frac{1}{3} = 0.333\cdots$, we can get numbers that are rationals but have an infinite decimal expansion.

This is actually 0.299999999999999988897769753748434595763683319091796875 in decimal.

Overflow

Here's an actually big problem in single point precision.

Let's say we want to square 2^{64}. Then, we should get 2^{128} right?

Overflow

Here's an actually big problem in single point precision.

Let's say we want to square 2^{64}. Then, we should get 2^{128} right?

Well, if your floats only support exponents up to 127, what happens when it reaches 128?

Overflow

Here's an actually big problem in single point precision.

Let's say we want to square 2^{64}. Then, we should get 2^{128} right?

Well, if your floats only support exponents up to 127, what happens when it reaches 128?

It gets set to -126...

Overflow

Here's an actually big problem in single point precision.

Let's say we want to square 2^{64}. Then, we should get 2^{128} right?

Well, if your floats only support exponents up to 127, what happens when it reaches 128?

It gets set to -126...

If we try to do this in Python, we actually get an overflow error! It does not allow us to perform the arithmetic!

Overflow

Here's an actually big problem in single point precision.

Let's say we want to square 2^{64}. Then, we should get 2^{128} right?

Well, if your floats only support exponents up to 127, what happens when it reaches 128?

It gets set to -126...

If we try to do this in Python, we actually get an overflow error! It does not allow us to perform the arithmetic!

In C#, they use 32-bit precision floats, and if it overflows, it gets set to infinity.

Integers vs. Floats

Python is actually really smart. If you try to overflow with an integer, it won't let you! It will keep increasing the size of the number until you get a memory error.

Integers vs. Floats

Python is actually really smart. If you try to overflow with an integer, it won't let you! It will keep increasing the size of the number until you get a memory error.

So, we'll try to use some floats, and cause some overflow errors.

The Largest Representable Number

We want to know the largest number that causes an error. So, what's the largest number that can be represented?

We want to know the largest number that causes an error. So, what's the largest number that can be represented?

We set the exponent AND the significand to the maximum value!

The Largest Representable Number

We want to know the largest number that causes an error. So, what's the largest number that can be represented?

We set the exponent AND the significand to the maximum value!

For single precision, our maximum exponent is 127, while our largest mantissa is: $2 - 2^{-23} \approx 1.99999988049$.

So, $1.9999998805 \cdot 2^{127}$ is about the largest number we can represent.

Notice that $2^{10} = 1024$, so we should expect an accuracy of $\frac{1}{2^{23}} \approx \frac{1}{10^6 \cdot 2^8} \approx \frac{1}{8000000}$.

The Largest Representable Number

We want to know the largest number that causes an error. So, what's the largest number that can be represented?

We set the exponent AND the significand to the maximum value!

For single precision, our maximum exponent is 127, while our largest mantissa is: $2 - 2^{-23} \approx 1.99999988049$.

So, $1.9999998805 \cdot 2^{127}$ is about the largest number we can represent.

Notice that $2^{10} = 1024$, so we should expect an accuracy of $\frac{1}{2^{23}} \approx \frac{1}{10^6 \cdot 2^8} \approx \frac{1}{8000000}$.

This turns out to be $3.4028235 \cdot 10^{34}$.

Imagine This is Python

In Python, this value is about: $1.8 \cdot 10^{308}$. Let's see!

In Python, this value is about: $1.8 \cdot 10^{308}$. Let's see!

```
>>> 10.018**308
1.7400287532476379e+308
>>> 10.019**308
1.794353452010299 4e+308
>>> 10.02**308
Traceback (most recent call last):
  File "<python-input-71>", line 1, in <module>
    10.02**308
    ~~~~~^^~~~
OverflowError: (34, 'Result too large')
```

Overflow In Real Life

Who has played COD zombies?

Overflow In Real Life

Who has played COD zombies?

In Black Ops 1, there are certain rounds that set the zombie health to a negative number! So, they'll die in one shot. Why does this happen?

Overflow In Real Life

Who has played COD zombies?

In Black Ops 1, there are certain rounds that set the zombie health to a negative number! So, they'll die in one shot. Why does this happen?

Overflow! In particular, BO1 uses 32-bit integers, so the largest number is $2^{31} \approx 10^{9}$.

Overflow In Real Life

Who has played COD zombies?

In Black Ops 1, there are certain rounds that set the zombie health to a negative number! So, they'll die in one shot. Why does this happen?

Overflow! In particular, BO1 uses 32-bit integers, so the largest number is $2^{31} \approx 10^9$.

Formula: On round 9, the zombies have 950HP, and every round after, it multiplies by 1.1.

Overflow In Real Life

Who has played COD zombies?

In Black Ops 1, there are certain rounds that set the zombie health to a negative number! So, they'll die in one shot. Why does this happen?

Overflow! In particular, BO1 uses 32-bit integers, so the largest number is $2^{31} \approx 10^9$.

Formula: On round 9, the zombies have 950HP, and every round after, it multiples by 1.1.

When is the first insta-kill round, or round when the health is rounded to 1?

```
>>> 2**31
2147483648
>>> 950 * 1.1**150
1536831943.987399
>>> 950 * 1.1**150 > 2**31
False
>>> 950 * 1.1**153
2045523317.4472284
>>> 950 * 1.1**153 > 2**31
False
>>> 950 * 1.1**154
2250075649.1919518
>>> 950 * 1.1**154 > 2**31
True
```

Indeed, since we started counting on round 9, we see that the first insta-kill round is: $9 + 154 = 163$

Indeed, since we started counting on round 9, we see that the first insta-kill round is: $9 + 154 = 163$

When working with integers, we can also overflow in the negative direction! So, $-2^{-31} - 1 = 2^{31}$.

Indeed, since we started counting on round 9, we see that the first insta-kill round is: $9 + 154 = 163$

When working with integers, we can also overflow in the negative direction! So, $-2^{-31} - 1 = 2^{31}$.

So, the rounds actually flip back and forth between insta-kill and regular rounds for about 20 rounds, before the numbers become a little more random.

Why Zero Matters

So, we looked at what happens when numbers get too big. But what about when they get small? Are there any issues that can occur?

Why Zero Matters

So, we looked at what happens when numbers get too big. But what about when they get small? Are there any issues that can occur?

We'll look at catastrophic cancellation in detail, and deal with underflow later.

Imagine we have two sticks, one with a length of 52.4 cm and another with 51.7 cm.

A reasonable thing to do would be to round these distances to the nearest whole number, and indeed, the relative error isn't too bad.

$\frac{0.7}{52.4} \approx 1.$something percent.

Relative and Absolute Error

Imagine we have two sticks, one with a length of 52.4 cm and another with 51.7 cm.

A reasonable thing to do would be to round these distances to the nearest whole number, and indeed, the relative error isn't too bad.

$\frac{0.7}{52.4} \approx$ 1.something percent.

But what if a guy takes your approximation as word. And now, this random guy wants to do math with the quantity between your two sticks.

Relative and Absolute Error

Imagine we have two sticks, one with a length of 52.4 cm and another with 51.7 cm.

A reasonable thing to do would be to round these distances to the nearest whole number, and indeed, the relative error isn't too bad.

$\frac{0.7}{52.4} \approx$ 1.something percent.

But what if a guy takes your approximation as word. And now, this random guy wants to do math with the quantity between your two sticks.

In the real world, the difference is 0.7cm, but after approximation, the difference is 0cm! That's technically infinite relative error!

What’s Actually Happening

Our sense of accuracy is dependant on our context. So, when we look at big things, small distances matter less.

What's Actually Happening

Our sense of accuracy is dependant on our context. So, when we look at big things, small distances matter less.

Indeed, it's only when we subtract two very similarly sized numbers do we end up losing accuracy!

What's Actually Happening

Our sense of accuracy is dependant on our context. So, when we look at big things, small distances matter less.

Indeed, it's only when we subtract two very similarly sized numbers do we end up losing accuracy!

So if we have two approximations that are only 0.1% off, but the numbers are super close, this 0.1% accuracy in the big world can lead to a 10% accuracy difference in the small world.

This principle is called catastrophic cancellation.

We designed our arithmetic system to be accurate around 0. But not around large numbers...

We designed our arithmetic system to be accurate around 0. But not around large numbers...

So, what if we do something crazy like... $(2^{63} - 463.53623) - 2^{63}$? What should the answer be?

We designed our arithmetic system to be accurate around 0. But not around large numbers...

So, what if we do something crazy like... $(2^{63} - 463.53623) - 2^{63}$? What should the answer be?

```
>>> 2**63 - 463.53625 - 2**63
0.0
```

We designed our arithmetic system to be accurate around 0. But not around large numbers…

So, what if we do something crazy like… $(2^{63} - 463.53623) - 2^{63}$? What should the answer be?

```
>>> 2**63 - 463.53625 - 2**63
0.0
```

Specific Spacing Bounds

In particular, in double precision land, numbers between 2^{53} and 2^{54} (remember, the mantissa is 52 bits, but have 53 bits of precision) can only represent even numbers!

```
>>> 2**53
9007199254740992
>>> 2**53 - 0.1
9007199254740992.0
>>> 2**53 + 0.99
9007199254740992.0
```

Specific Spacing Bounds

In particular, in double precision land, numbers between 2^{53} and 2^{54} (remember, the mantissa is 52 bits, but have 53 bits of precision) can only represent even numbers!

```
>>> 2**53
9007199254740992
>>> 2**53 - 0.1
9007199254740992.0
>>> 2**53 + 0.99
9007199254740992.0
```

This gap just doubles every power of two!

For single precision floats, we can't tell the difference between even and odds for numbers that are about $16,800,000$.

Specific Spacing Bounds 2

So at 2^{100}, we can't tell the difference between numbers space $2^{47} \approx 10^{14}$ apart.

```
>>> from decimal import Decimal
>>> Decimal.from_float(2**100)
Decimal('1267650600228229401496703205376')
>>> Decimal.from_float(2**100 + 100000000000000)
Decimal('1267650600228229501496703205376')
```

Specific Spacing Bounds 2

So at 2^{100}, we can't tell the difference between numbers space $2^{47} \approx 10^{14}$ apart.

```
>>> from decimal import Decimal
>>> Decimal.from_float(2**100)
Decimal('1267650600228229401496703205376')
>>> Decimal.from_float(2**100 + 100000000000000)
Decimal('1267650600228229501496703205376')
```

Indeed, this principle still holds with small numbers! It's just that the scale is different.

Remember, this is a truth about ANY approximation, not just floating-point arithmetic!

So, we want to avoid subtracting two similarly sized numbers.

So, in an ideal world, we really don’t want to subtract two numbers that are really close together.

Weird Rules

So, in an ideal world, we really don’t want to subtract two numbers that are really close together.

But where would this show up in the real world?

So, in an ideal world, we really don't want to subtract two numbers that are really close together.

But where would this show up in the real world?

Example 1: Calulating Specific Series

We know that the Taylor series for $\frac{1}{x}$ at $a = 1$ is:

$$1 - (x - 1) + (x - 1)^2 - (x - 1)^3 + \cdots$$

Indeed, for values close to 1, we can get catastrophic cancellation, and absolutely ruin our arithmetic. Let's see it in action!

```
>>> 1 - (0.9999999999999-1)
1.0000000000001
>>> 1 - (0.9999999999999999-1)
1.0
>>> 1/0.9999999999999999
1.0000000000000002
```

While this is a tame example, what if $1 - x^2$ appears somewhere (and this is even worse if it appears in the denominator) for x values close to 1?

```
>>> 1 - (0.9999999999999-1)
1.0000000000001
>>> 1 - (0.9999999999999999-1)
1.0
>>> 1/0.9999999999999999
1.0000000000000002
```

While this is a tame example, what if $1 - x^2$ appears somewhere (and this is even worse if it appears in the denominator) for x values close to 1?

Luckily, here we can employ a difference of squares, but the point is: Be careful of catastrophic cancellation!

Example 2: Computing Differences Between Approximations

Let's say I want to approximate a real number, say π or φ.

The logical thing to do is take our approximation, subtract the 'real' value and see the accuracy.

But this stops working eventually because of catastrophic cancellation.

Another Example

Example 2: Computing Differences Between Approximations

Let's say I want to approximate a real number, say π or φ.

The logical thing to do is take our approximation, subtract the 'real' value and see the accuracy.

But this stops working eventually because of catastrophic cancellation.

There's no good fix to this! Indeed, the only thing we can do is give an error bound for how close those numbers really are!

The only 'fix' is higher precision.

ULP

We now define the notion of precision.

ULP, or Unit in Last Position, or, Unit of Least Precision is the distance from any floating-point number x to the next closest number.

ULP

We now define the notion of precision.

ULP, or Unit in Last Position, or, Unit of Least Precision is the distance from any floating-point number x to the next closest number.

Indeed, any value around x which is within 0.5 ULP is not differentiable from x. The only way to get a better approximation is by increasing the precision.

Underflow

One final thing before we can talk about IEEE 754: Underflow.

Underflow

One final thing before we can talk about IEEE 754: Underflow.

When any number rounds to 0, since it's the closest float, it's called underflow.

Underflow

One final thing before we can talk about IEEE 754: Underflow.

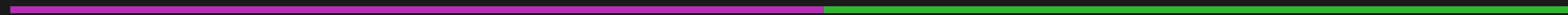
When any number rounds to 0, since it's the closest float, it's called underflow.

This is once again fixable by higher precision or smarter constructions.

For example, convergent infinite series will eventually have terms that approach zero, and eventually, these terms will underflow.

We can either increase precision or play around with the sum to avoid underflow.

IEEE 754



The Basics

This is a floating-point arithmetic standard, introduced in 1985, to ensure that every computer uses the same standard.

While there are many things specified, such as how to round, error handling, operations, all the details, and so on.

We'll focus on two main parts: Special values and subnormals.

Special Values

Turns out, IEEE754 allows for $+0$ and -0 to exist at the same time! But both are equal?

In particular, we set the exponent and mantissa to 0.

Special Values

Turns out, IEEE754 allows for $+0$ and -0 to exist at the same time! But both are equal?

In particular, we set the exponent and mantissa to 0.

Infinity allows comes in positive and negative!

We set the exponent to all 1 and the fraction to 0.

Special Values

Turns out, IEEE754 allows for $+0$ and -0 to exist at the same time! But both are equal?

In particular, we set the exponent and mantissa to 0.

Infinity allows comes in positive and negative!

We set the exponent to all 1 and the fraction to 0.

The worse is NaN, or, Not Available Number. This appear when we do illegal math, such as taking the square root of a negative number.

This is represented by any sign, exponent as all 1 and the mantissa to anything except 0

Lost Numbers

The real issue is this: In single precision, $2 \cdot (2^{23} - 1) = 16,777,214$ values are duplicated! (out of $4,294,967,296$, so we lose $\frac{1}{256}$ of all numbers)

In double precision, we lose $2 \cdot (2^{52} - 1) \approx 9 \cdot 10^{15}$ numbers!!! (out of $2^{64} \approx 1.844 \cdot 10^{19}$ numbers, we lose $\approx \frac{1}{2050}$)

Lost Numbers

The real issue is this: In single precision, $2 \cdot (2^{23} - 1) = 16,777,214$ values are duplicated! (out of $4,294,967,296$, so we lose $\frac{1}{256}$ of all numbers)

In double precision, we lose $2 \cdot (2^{52} - 1) \approx 9 \cdot 10^{15}$ numbers!!! (out of $2^{64} \approx 1.844 \cdot 10^{19}$ numbers, we lose $\approx \frac{1}{2050}$)

This is really annoying, especially since we want more precision… probably.

Subnormals

Now, we get into the most controversial part of IEEE 754, subnormals.

Subnormals

Now, we get into the most controversial part of IEEE 754, subnormals.

Instead of underflowing and directly setting any smaller values to 0, we'll somehow gradually underflow our way to 0.

Subnormals

Now, we get into the most controversial part of IEEE 754, subnormals.

Instead of underflowing and directly setting any smaller values to 0, we'll somehow gradually underflow our way to 0.

How do we do this?

Subnormals

Now, we get into the most controversial part of IEEE 754, subnormals.

Instead of underflowing and directly setting any smaller values to 0, we'll somehow gradually underflow our way to 0.

How do we do this?

When we (somehow) signal that our number is too small and about to underflow to zero, we switch how we calculate our numbers.

We now let number be represented by:

$$0.x_1 x_2 x_3 \cdots x_{23} \cdot 2^{-126}$$

How Subnormals Work

$$0.x_1x_2x_3 \cdots x_{23} \cdot 2^{-126}$$

Now, when $x_3 = 1$ and everything else is set to 0, then we represent the number 2^{129}, a smaller number than our exponent would allow.

How Subnormals Work

$$0.x_1 x_2 x_3 \cdots x_{23} \cdot 2^{-126}$$

Now, when $x_3 = 1$ and everything else is set to 0, then we represent the number 2^{129}, a smaller number than our exponent would allow.

So we can represent numbers from 2^{-126} all the way down to 2^{-149}.

In double precision, we go from −1022 to −1074 in the exponent.

Notice how in single precision, we can only represent -126 to 127. But there are 256 possible combination. Where did the others go?

Implementation

Notice how in single precision, we can only represent -126 to 127. But there are 256 possible combination. Where did the others go?

When the exponent is all 1, we get the special values (zero, infinity, NaN) from above.

When the exponent is all 0, we get subnormals, or, denormalized numbers.

In all other cases, we say that the number is normalized.

Complexity

Turns out that this was the most controversial part of the IEEE 754 standard! What could possibly go wrong? Why was this so controversial?

Complexity

Turns out that this was the most controversial part of the IEEE 754 standard! What could possibly go wrong? Why was this so controversial?

Who here knows what music is?

Complexity

Turns out that this was the most controversial part of the IEEE 754 standard! What could possibly go wrong? Why was this so controversial?

Who here knows what music is?

Digital Audio Workstations (DAWs) in the early 2000′s were just living their life. But all of the sudden, Intel released their revolutionary Pentium 4 (released in 2000) chip. Any computer using these chips saw twice as much CPU usage being used. What could possibly be happening?

Complexity

Turns out that this was the most controversial part of the IEEE 754 standard! What could possibly go wrong? Why was this so controversial?

Who here knows what music is?

Digital Audio Workstations (DAWs) in the early 2000′s were just living their life. But all of the sudden, Intel released their revolutionary Pentium 4 (released in 2000) chip. Any computer using these chips saw twice as much CPU usage being used. What could possibly be happening?

Intel integrated the IEEE 754 standard into their CPU, and anytime volume faded to 0, well, instead of numbers actually being set to 0 via underflow, the computer calculated subnormal numbers! But at such a small level, no human could hear it!

Complexity

Turns out that this was the most controversial part of the IEEE 754 standard! What could possibly go wrong? Why was this so controversial?

Who here knows what music is?

Digital Audio Workstations (DAWs) in the early 2000′s were just living their life. But all of the sudden, Intel released their revolutionary Pentium 4 (released in 2000) chip. Any computer using these chips saw twice as much CPU usage being used. What could possibly be happening?

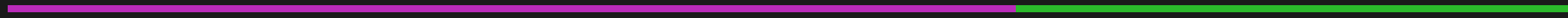
Intel integrated the IEEE 754 standard into their CPU, and anytime volume faded to 0, well, instead of numbers actually being set to 0 via underflow, the computer calculated subnormal numbers! But at such a small level, no human could hear it!

Generally, this resulted in about twice as more work being done by the computer!

An Improvement

The question we might want to ask is this: Can we develop a number system that eliminates some of these problems, while still doing a float's job well?

Posits



Formal Definition

The original paper for this idea comes from: Beating Floating Point at its Own Game: Posit Arithmetic. Wow, classy guys.

Formal Definition

The original paper for this idea comes from: Beating Floating Point at its Own Game: Posit Arithmetic. Wow, classy guys.

The paper is only about 17 pages, and goes into the technicalities, but I'm really trying to show you the structure and payoff.

Formal Definition

The original paper for this idea comes from: Beating Floating Point at its Own Game: Posit Arithmetic. Wow, classy guys.

The paper is only about 17 pages, and goes into the technicalities, but I'm really trying to show you the structure and payoff.

The idea? Make that scientific notation even more scientific notation-y, by introducing 2^{2^n}.

Formal Definition

The original paper for this idea comes from: Beating Floating Point at its Own Game: Posit Arithmetic. Wow, classy guys.

The paper is only about 17 pages, and goes into the technicalities, but I'm really trying to show you the structure and payoff.

The idea? Make that scientific notation even more scientific notation-y, by introducing 2^{2^n}.

The structure is as follows: Sign bit, the regime, the exponent, the fraction.

The Regime

First, define the length of the exponent, call it es. Say $es = 4$. We define the regime as per the following table:

Binary	0000	0001	001x	01xx	10xx	110x	1110	1111
k	-4	-3	-2	-1	0	1	2	3

Notice that the regime is not a fixed number of bits! Thus, depending on the circumstances, we get more or less precision since the exponent section is fixed.

Define a number called useed, and set it to $2^{2^{es}}$. Then, the number contributed by the regime is: useed^k.

es	0	1	2	3	4
useed	2	$2^2 = 4$	$4^2 = 16$	$16^2 = 256$	$256^2 = 65536$

What is a Posit?

A posit will take the following form:

$$\pm \text{useed}^{k} \cdot 2^{e} \cdot 1.f_1 f_2 f_3 \ldots$$

Where:

$\pm$ comes from our sign bit.

k comes from our regime.

e is our exponent value.

$f_1 f_2 \cdots$ is our mantissa/significand.

What is a Posit?

A posit will take the following form:

$$\pm \text{useed}^k \cdot 2^e \cdot 1.f_1 f_2 f_3 \ldots$$

Where:

$\pm$ comes from our sign bit.

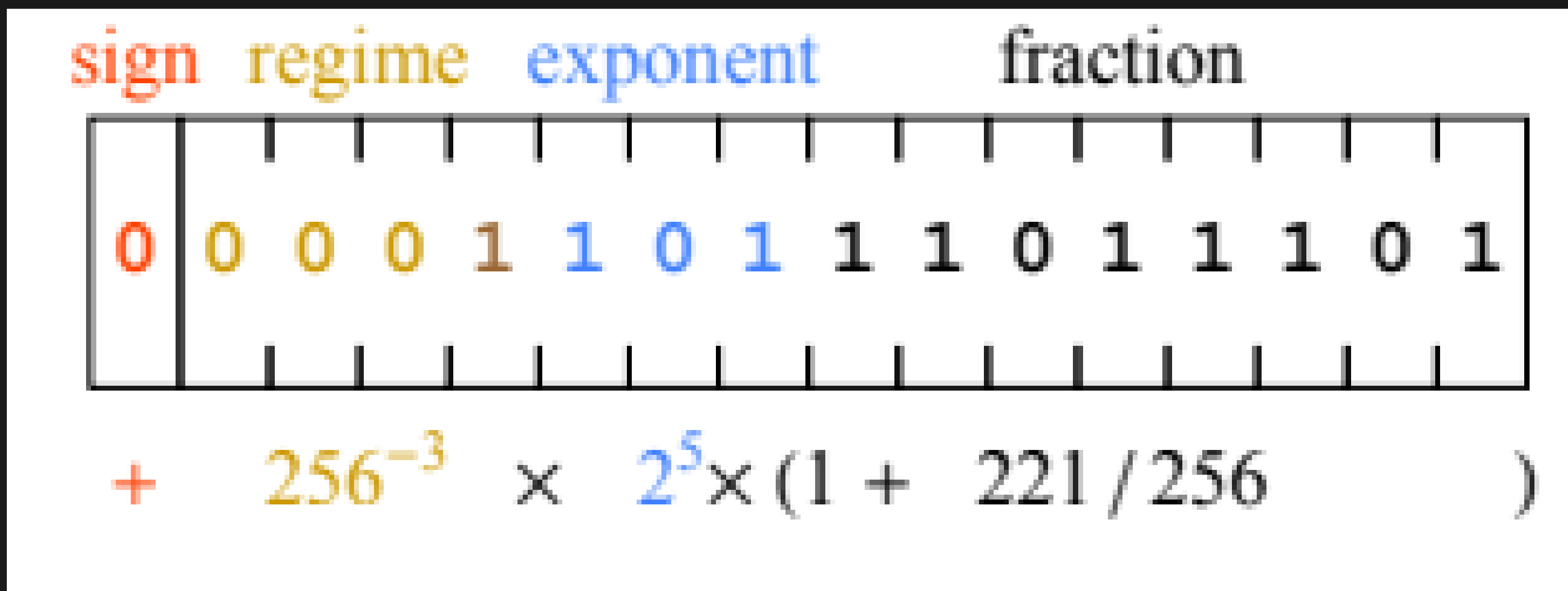
k comes from our regime.

e is our exponent value.

$f_1 f_2 \cdots$ is our mantissa/significand.

Notice that the regime being not a fixed size allows our mantissa to change sizes.

An Example



I ‘stole’ this from the paper. They color code things!

This number ends up being $\frac{477}{134217728} \approx 3.55393 \cdot 10^{-6}$

A Few Things to Notice

The first question is: Why is this such a weird definition? Why not just fix the number of bits and let k be much larger?

A Few Things to Notice

The first question is: Why is this such a weird definition? Why not just fix the number of bits and let k be much larger?

I don't really have the 'right' answer, only my interpretation.

In floating-point arithmetic, if we increase the largest representable number, we increase the number of exponents. But as we know, this is not the important thing, precision is.

But let's say we want to increase the number of exponent bits of a float by 2. Then, we increase the exponent by a factor of 4.

But increasing es by one allows our useed to scale exponentially!

Implications on the Exponent and Scaling

But here's something really cool. The exponent can be strictly positive as the regime is what allows our number to have positive or negative exponent!

Implications on the Exponent and Scaling

But here's something really cool. The exponent can be strictly positive as the regime is what allows our number to have positive or negative exponent!

So, let's take a look at a table to show how much better posits are at beating floating point with respect to largest/smallest representable number.

Implications on the Exponent and Scaling

But here's something really cool. The exponent can be strictly positive as the regime is what allows our number to have positive or negative exponent!

So, let's take a look at a table to show how much better posits are at beating floating point with respect to largest/smallest representable number.

Size, Bits	IEEE Float Exp. Size	Approx. IEEE Float Dynamic Range	Posit es Value	Approx. Posit Dynamic Range
16	5	6×10^{-8} to 7×10^{4}	1	4×10^{-9} to 3×10^{8}
32	8	1×10^{-45} to 3×10^{38}	3	6×10^{-73} to 2×10^{72}
64	11	5×10^{-324} to 2×10^{308}	4	2×10^{-299} to 4×10^{298}
128	15	6×10^{-4966} to 1×10^{4932}	7	1×10^{-4855} to 1×10^{4855}
256	19	2×10^{-78984} to 2×10^{78913}	10	2×10^{-78297} to 5×10^{78296}

Pro and Contra

There are a few advantages:

1. No subnormals!
2. No NaN, so we use more of the representable numbers.
3. No weird anomalies like negative zero.

Disadvantages:

1. No parallelization since each section of bits aren't fixed size.

Pro and Contra

There are a few advantages:

1. No subnormals!
2. No NaN, so we use more of the representable numbers.
3. No weird anomalies like negative zero.

Disadvantages:

1. No parallelization since each section of bits aren't fixed size.

While there are many more technical details, we'll leave them in the paper.

Pro and Contra

There are a few advantages:

1. No subnormals!
2. No NaN, so we use more of the representable numbers.
3. No weird anomalies like negative zero.

Disadvantages:

1. No parallelization since each section of bits aren't fixed size.

While there are many more technical details, we'll leave them in the paper.

So now, we can look at graphs!

Graphs

Comparing Floats

The graphs take a mathematical function (like square root) or arithmetic function, and analyze the accuracy of posit arithmetic versus floating-point arithmetic.

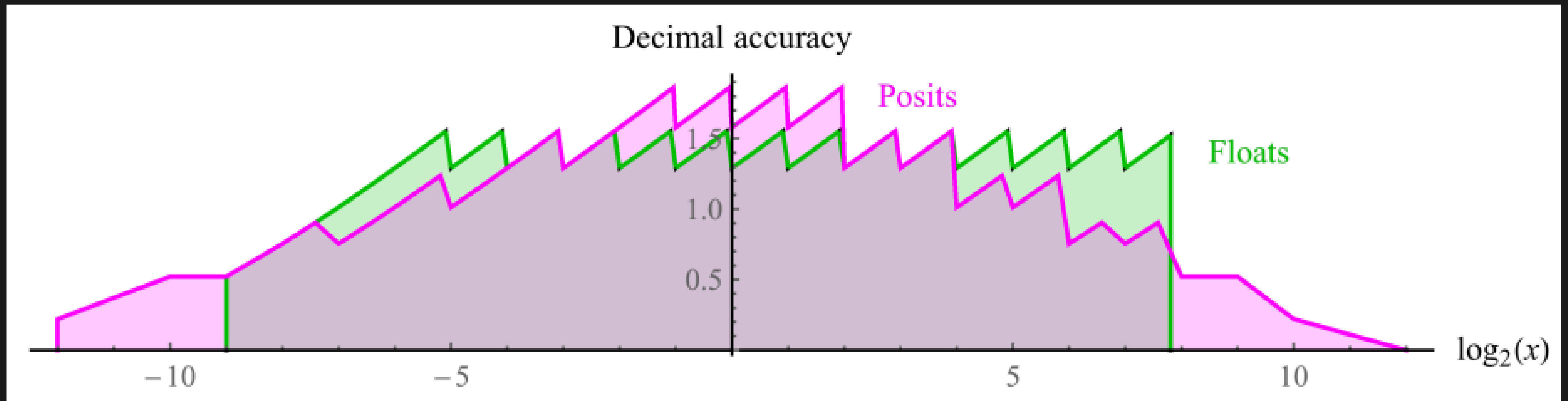
Note: We'll use 8-bit floats and posits.

For floats, we have 4 exponent bits and 3 fraction bits.

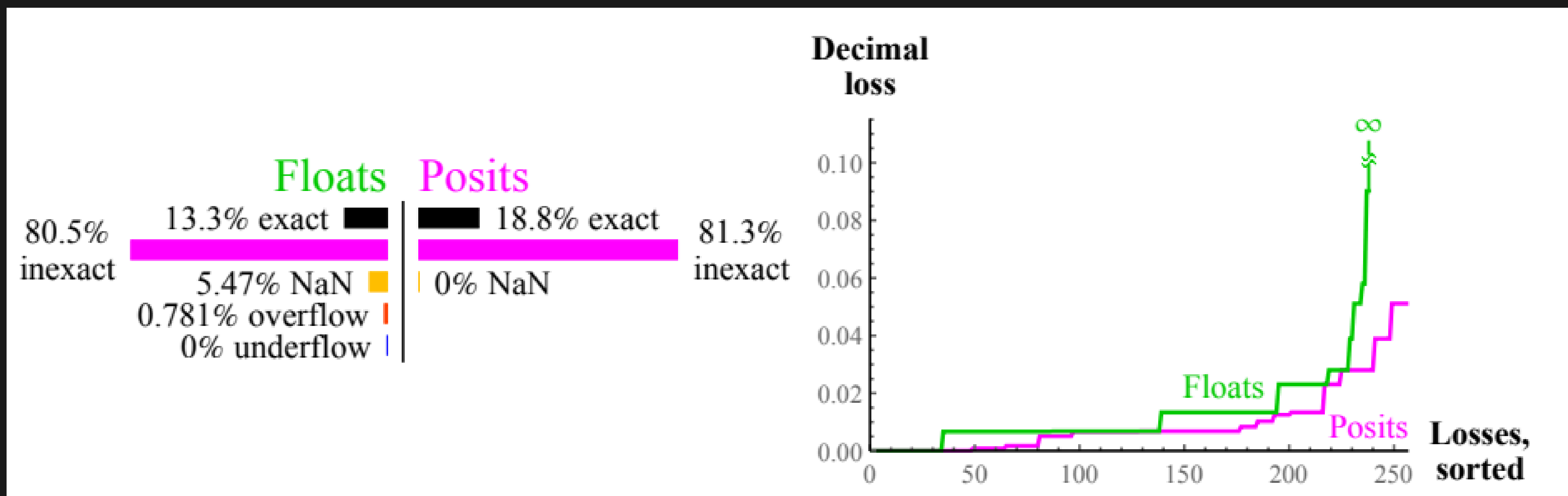
For posits, we have 1 exponent bit.

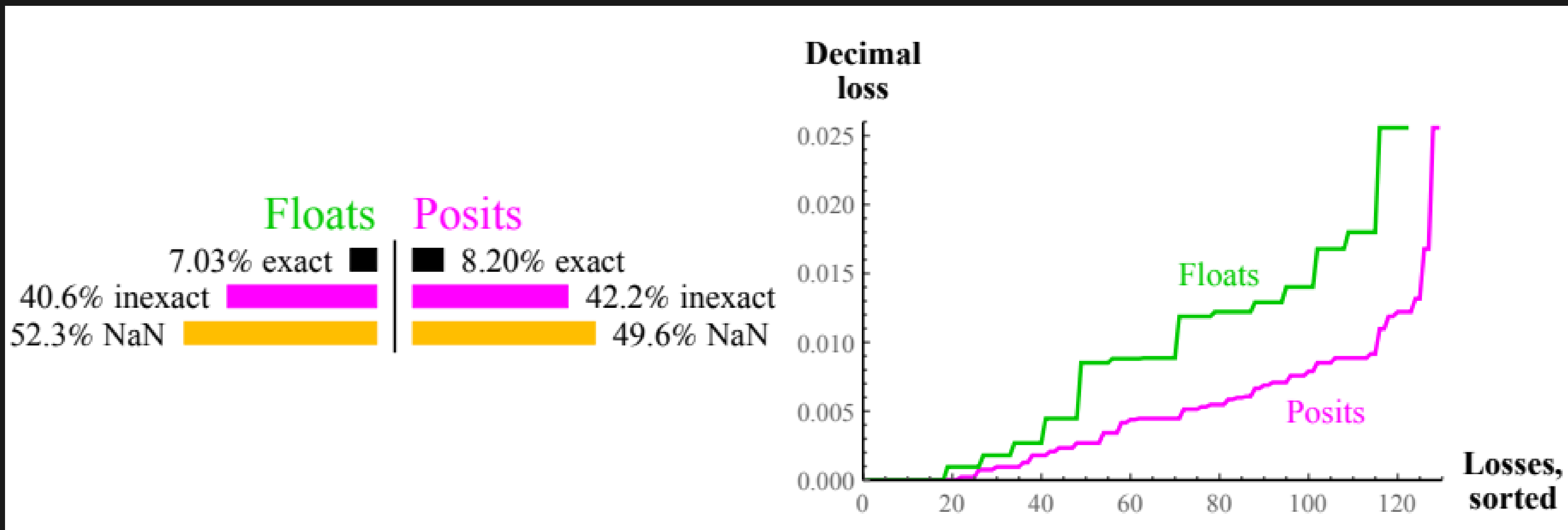
This will allow us to compute every combination of values for each operation.

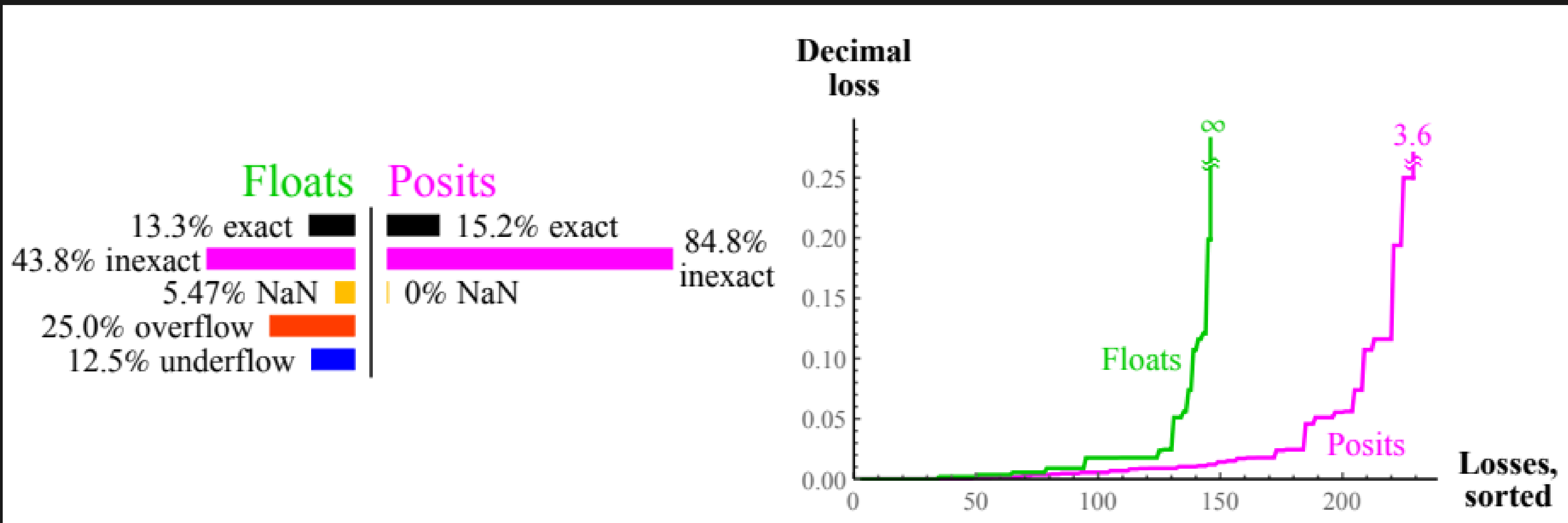
General Accuracy



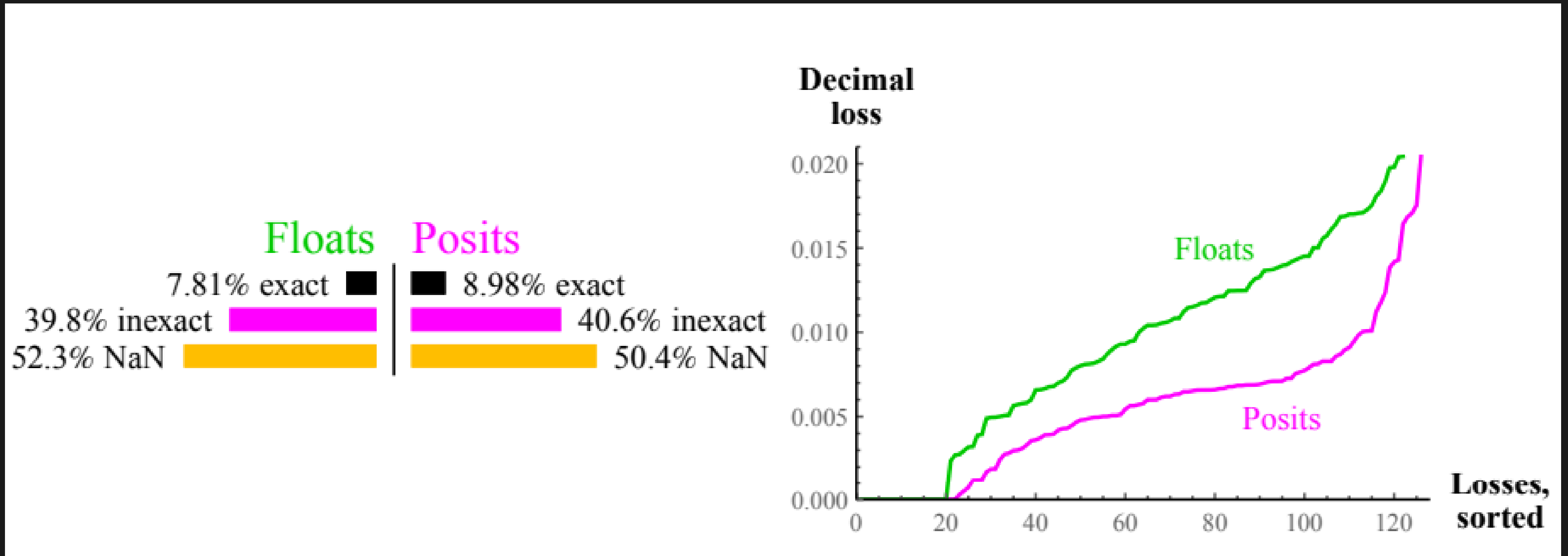
Reciprocal

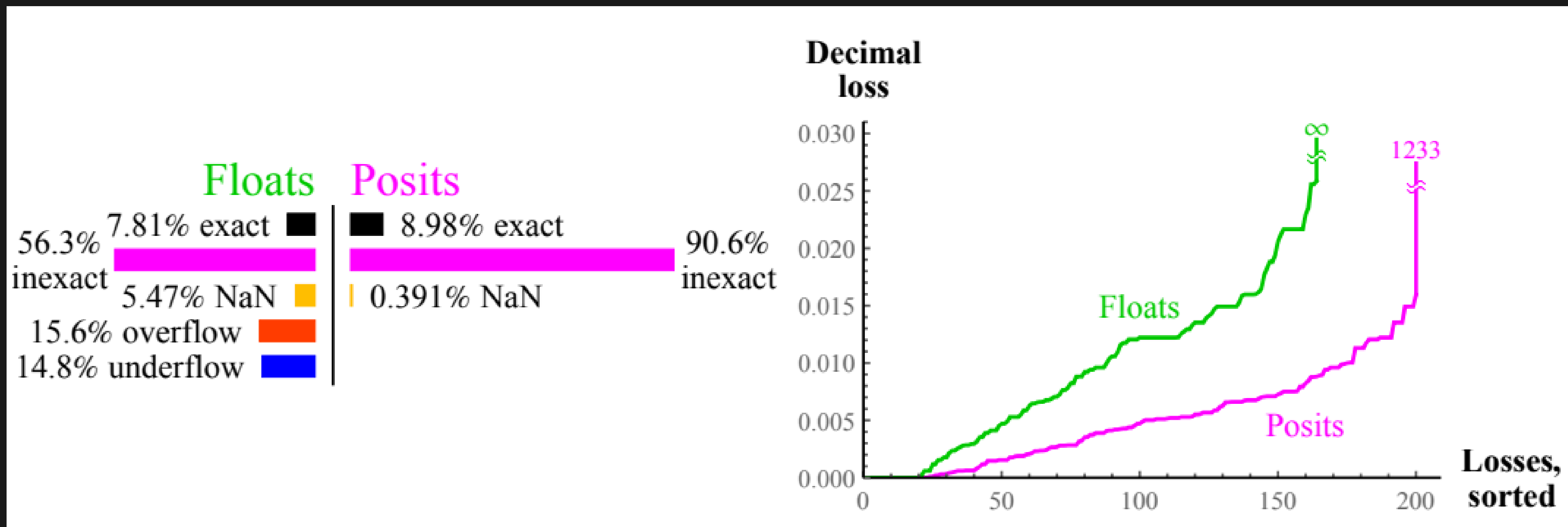



Floats
Posits
7.03% exact
8.20% exact
40.6% inexact
42.2% inexact
52.3% NaN
49.6% NaN
Decimal loss
0.025
0.020
0.015
0.010
0.005
0.000
0
20
40
60
80
100
120
Losses, sorted
Floats
Posits


Floats
Posits
13.3% exact
43.8% inexact
5.47% NaN
25.0% overflow
12.5% underflow
15.2% exact
84.8% inexact
0% NaN
Decimal loss
∞
3.6
0.25
0.20
0.15
0.10
0.05
0.00
0
50
100
150
200
Floats
Posits
Losses, sorted

Log (Base 2)



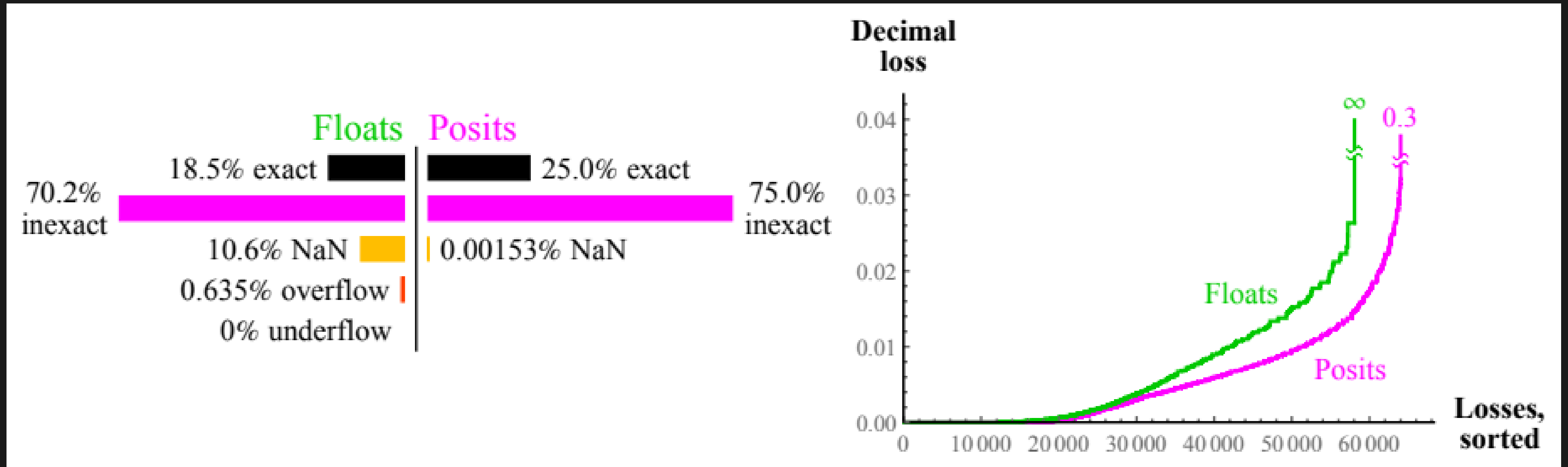

Floats
Posits
56.3% inexact
7.81% exact
5.47% NaN
15.6% overflow
14.8% underflow
8.98% exact
90.6% inexact
0.391% NaN
Decimal loss
0.030
0.025
0.020
0.015
0.010
0.005
0.000
0
50
100
150
200
∞
1233
Floats
Posits
Losses, sorted

Arithmetic Operation

For arithmetic operations, we'll compare $x + y$ and $x \cdot y$ for all possible combinations of x and y.

In particular, in the 2d graphs, each pixel represents a single operation.

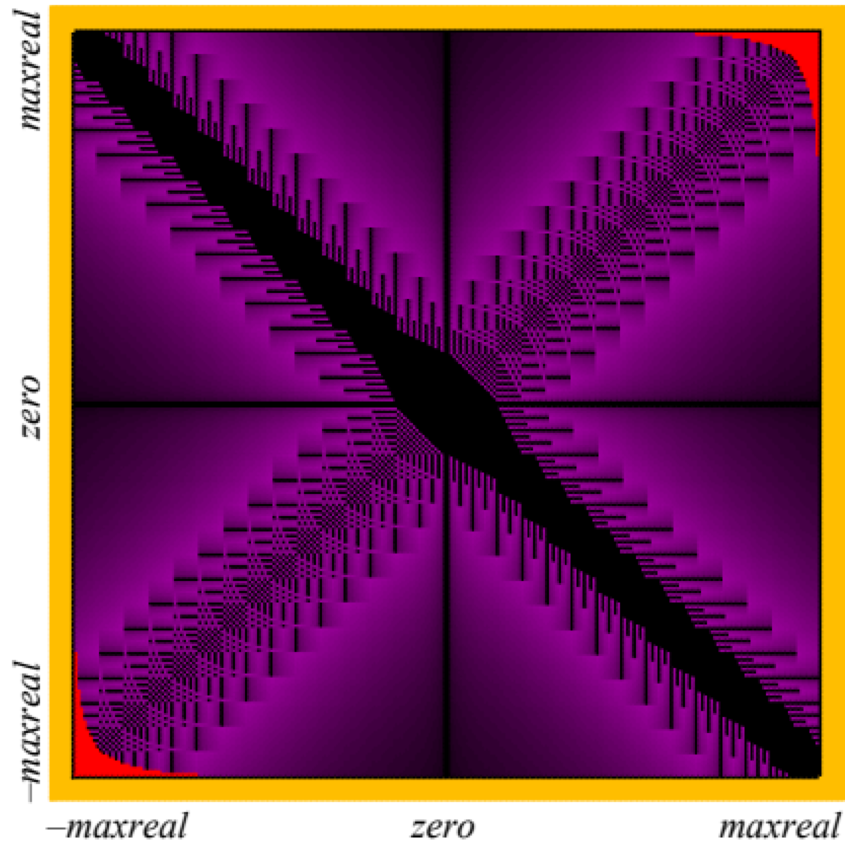
Addition



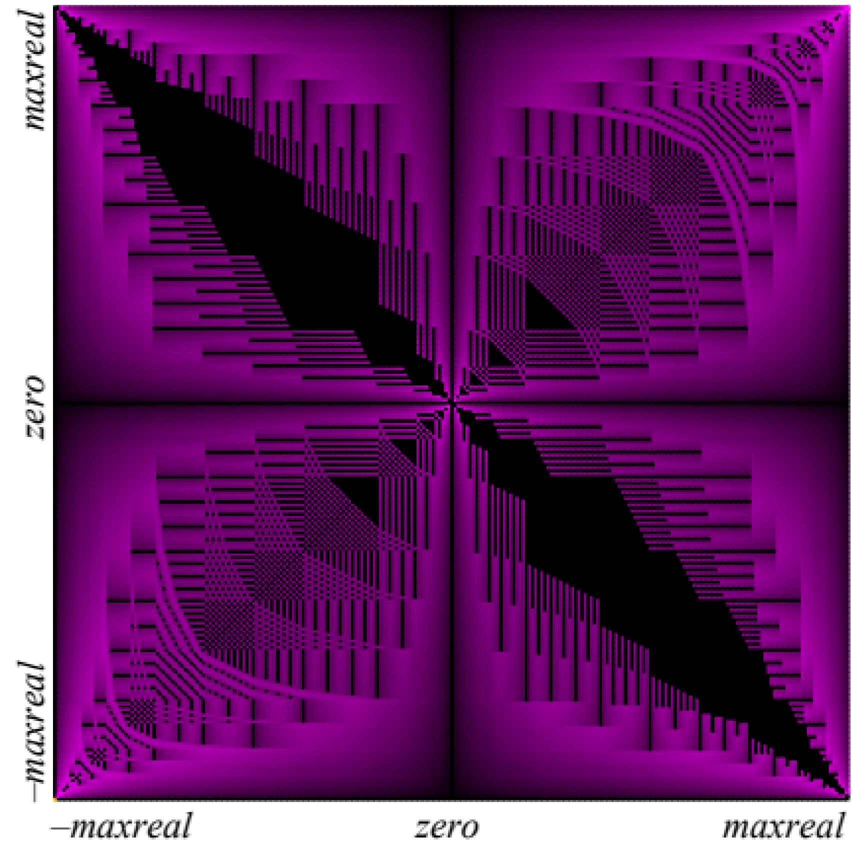
Addition Graph

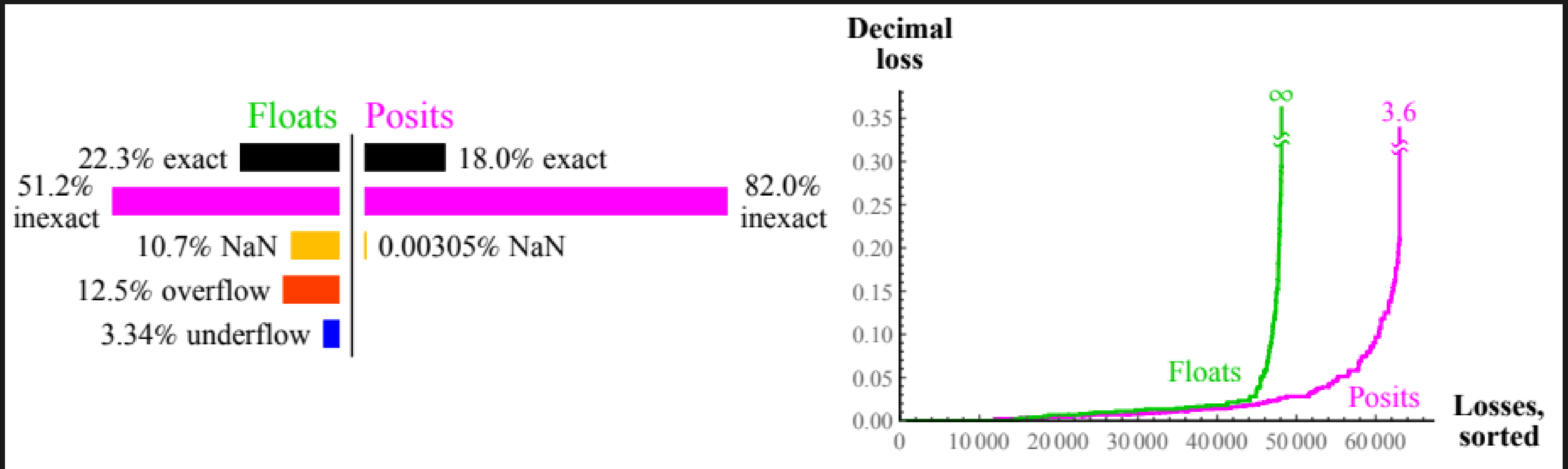
Addition Closure ■=Exact, ■=Inexact, ■=Overflow, ■=NaN

Floats



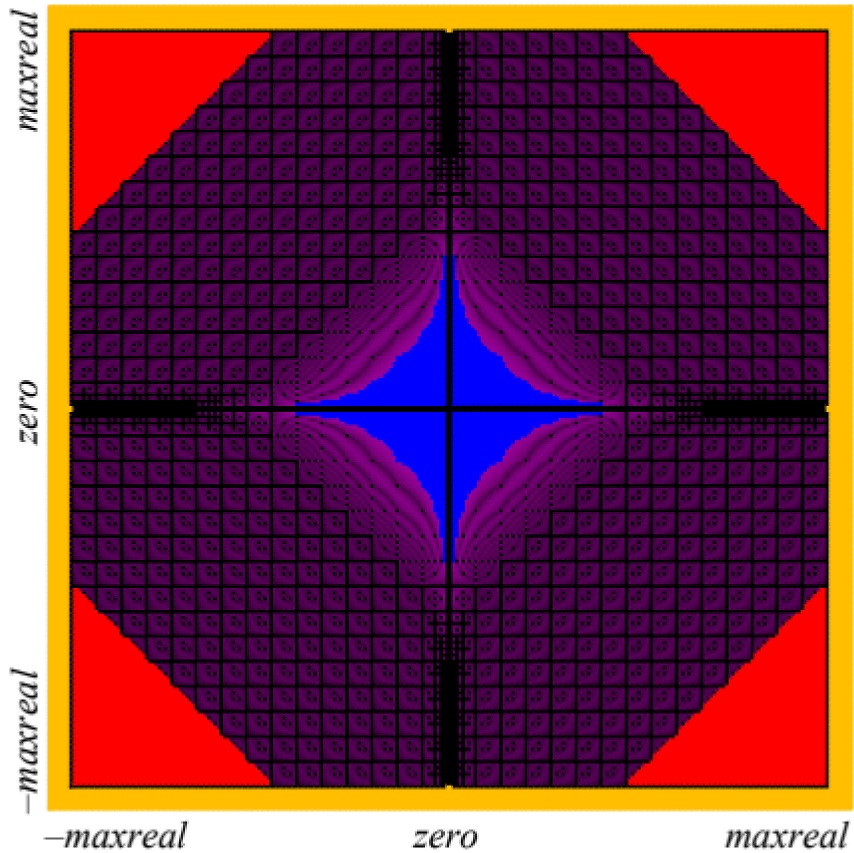
Posits

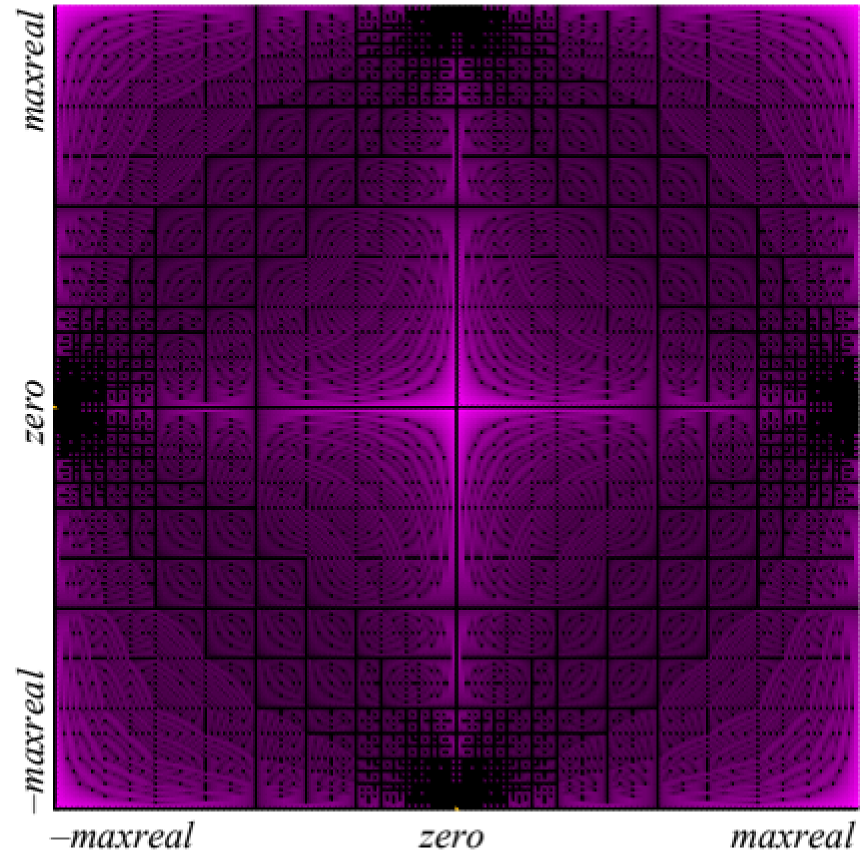



Floats
Posits
22.3% exact
18.0% exact
51.2% inexact
82.0% inexact
10.7% NaN
0.00305% NaN
12.5% overflow
3.34% underflow
Decimal loss
0.35
0.30
0.25
0.20
0.15
0.10
0.05
0.00
∞
3.6
Floats
Posits
0
10 000
20 000
30 000
40 000
50 000
60 000
Losses, sorted

Multiplication Graph

Multiplication Closure ■=Exact, ■=Inexact, ■=Overflow, ■=Underflow, ■=NaN





Numerical Analysis

What is Numerical Analysis

Numerical Analysis is the field of doing math with computers. And hopefully you see the complexity that can arise from just doing basic arithmetic.

What is Numerical Analysis

Numerical Analysis is the field of doing math with computers. And hopefully you see the complexity that can arise from just doing basic arithmetic.

But it's not just arithmetic! Any time that we need to calculate something in math, we need to understand numerical analysis to do this job.

What is Numerical Analysis

Numerical Analysis is the field of doing math with computers. And hopefully you see the complexity that can arise from just doing basic arithmetic.

But it’s not just arithmetic! Any time that we need to calculate something in math, we need to understand numerical analysis to do this job.

For example, finding roots of functions, or minimums or maximums of functions. When we can’t do it by analysis, we have no choice but to rely on computers.

What is Numerical Analysis

Numerical Analysis is the field of doing math with computers. And hopefully you see the complexity that can arise from just doing basic arithmetic.

But it's not just arithmetic! Any time that we need to calculate something in math, we need to understand numerical analysis to do this job.

For example, finding roots of functions, or minimums or maximums of functions. When we can't do it by analysis, we have no choice but to rely on computers.

Ever calculated a derivative?

What is Numerical Analysis

Numerical Analysis is the field of doing math with computers. And hopefully you see the complexity that can arise from just doing basic arithmetic.

But it's not just arithmetic! Any time that we need to calculate something in math, we need to understand numerical analysis to do this job.

For example, finding roots of functions, or minimums or maximums of functions. When we can't do it by analysis, we have no choice but to rely on computers.

Ever calculated a derivative?

The definition literally requires us to take limits with increasingly small numbers! That's a fundamental issue we talked about here!

What is Numerical Analysis 2

Differential equations are a huge component of numerical analysis! To compute well, we need to understand how computers can do math!

What is Numerical Analysis 2

Differential equations are a huge component of numerical analysis! To compute well, we need to understand how computers can do math!

Here's something interesting. What if instead of floating-point arithmetic, we instead use intervals and just say that the real value we want to calculate is contained in an interval?

What is Numerical Analysis 2

Differential equations are a huge component of numerical analysis! To compute well, we need to understand how computers can do math!

Here's something interesting. What if instead of floating-point arithmetic, we instead use intervals and just say that the real value we want to calculate is contained in an interval?

Then, we can define arithmetic on this number system. What are the advantages? Disadvantages?

What is Numerical Analysis 2

Differential equations are a huge component of numerical analysis! To compute well, we need to understand how computers can do math!

Here's something interesting. What if instead of floating-point arithmetic, we instead use intervals and just say that the real value we want to calculate is contained in an interval?

Then, we can define arithmetic on this number system. What are the advantages? Disadvantages?

Take it one step further, and do math of complex numbers using intervals. Are there any problems that appear? Well, for one, a rectangle in $\mathbb{C}$ multiplied by another rectangle in $\mathbb{C}$ doesn't produce a rectangle!

Last But Not Least, Number Theory

What about number theory? Are there algorithms or computations that we want to do? In simple cases, we need to be able to at least understand the basic of this. Yes, we might not use floating-point arithmetic, but overflow is still a problem!

Last But Not Least, Number Theory

What about number theory? Are there algorithms or computations that we want to do? In simple cases, we need to be able to at least understand the basic of this. Yes, we might not use floating-point arithmetic, but overflow is still a problem!

And what about analytic number theory? The 'exercise' about $\mathbb{C}$ in the previous slide is something someone actually dealt with when trying to compute values for the Generalized Riemann Hypothesis (David J. Platt) in their Ph.D Thesis.

Thank You!

Sources

https://www.reddit.com/r/CODZombies/comments/5tuy65/an_indepth_look_into_black_ops_1_zombie_mechanics/

https://zombiesarchive.fandom.com/wiki/Insta_Kill_Rounds

https://www.youtube.com/watch?v=y-NOz94ZEOA

Wikipedia. All of Wikipedia.

https://posithub.org/docs/BeatingFloatingPoint.pdf