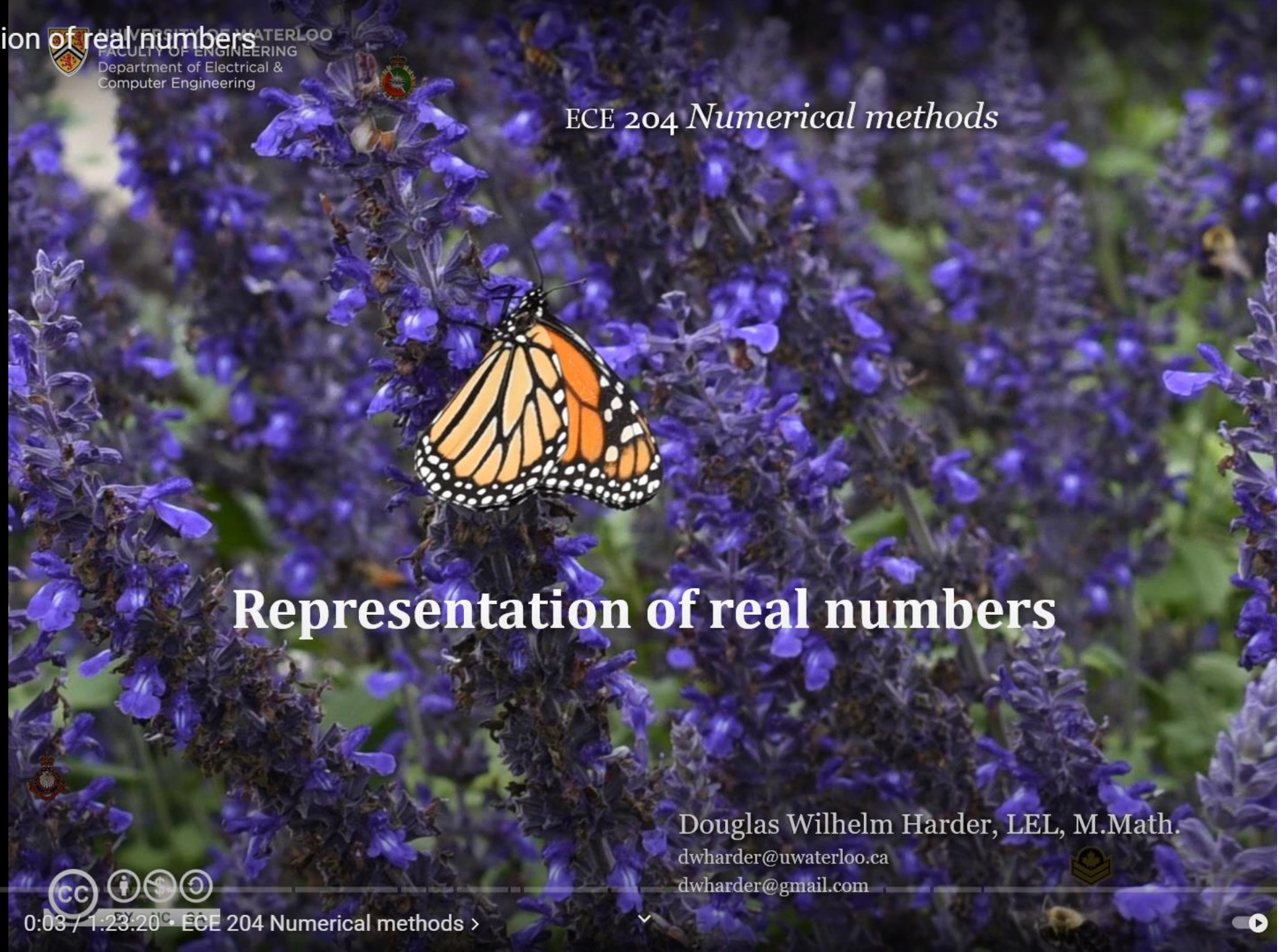
2 Representation of real numbers



Introduction

- In this topic, we will
  - Consider representations of real numbers
  - Describe our six-decimal-digit representation
  - Look at relative error, addition, multiplication and issues with floating-point numbers
  - Discuss special numbers such as representations of zero, denormalized, infinity and indeterminant
  - Describe the double-precision floating-point representation
  - Consider some of the properties
  - Perform addition and multiplication of doubles

Double-precision floating-point numbers

- The double-precision floating-point representation is as follows:
  bbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbb
- For 00000000001 to 11111111110, this represents

$$(-1)^{b}\ 1.b_1b_2b_3b_4b_5\cdots b_{52}\times 2^{bbbbbbbbbbb-01111111111}$$

- For 00000000000, this represents

$$(-1)^{b}\ 0.b_1b_2b_3b_4b_5\cdots b_{52}\times 2^{1-01111111111}$$

- We also have that
  b11111111110000000000000000000000000000000000000000000000000000

$$(-1)^{b}\ \infty$$

- We also have, a reserved representation for indeterminant results:
  011111111111000000000000000000000000000000000000000000000000000
  - This is called *not-a-number* or NaN

Double-precision floating-point numbers

- Approaching floating-point numbers in this format makes it difficult, at best, to understand the design
  - First, there are far too many bits to easily grasp
  - Second, binary numbers are already difficult to grasp
- Consequently, we will start, instead, with simpler decimal representations of real numbers
  - All the design issues we will see will also affect the binary representations of real numbers

Representation of integers

- While integers can be represented exactly
  - Suppose we restricted ourselves to storing integers using only six decimal digits:

    ±NNNNNN
  - This allows you to store numbers in the range

    –999999 to +999999
  - One problem: there are two representations of zero:

    –000000 and +000000
  - This is solved using 2s complement in binary and 10s complement in decimal
    - We won't go into this…

Representation of numbers on [0, 1)

- Suppose we were modelling processor usage:
  - Interpret the six digits

    NNNNNN

    as representing

    $0.NNNNNN$
  - This allows us to store any real number on the range [0, 1) with varying levels of precisions

Representing larger ranges

- Alternatively, we could interpret the six digits

  ±NNNNNN

  as representing

  ±NNN.NNN

  – This allows us to store any real number on the range

  [–1000, 1000],

  again, with varying precisions

Floating-point representations

- All such representations are called *fixed-point* representations
  - The decimal point is *fixed* in a specific location
  - This is ideal if you know that your real numbers must be bounded on some range
    - For example, [0, 1) or [0, 256)
  - This is less optimal if we want to represent a much more significant range of real numbers
  - For this, we will use the scientific notation approach described previously:

$$d_0.d_1d_2d_3d_4d_5\cdots d_n \times 10^e$$

$$b_0.b_1b_2b_3b_4b_5\cdots b_n \times 2^e$$

Floating-point representations

- IEEE 754 uses this approach in defining the double-precision floating-point representation (`double`)
  - You need to understand the reasoning behind the design, the choices, and the consequent weaknesses
  - It is more difficult when trying to describe 64-bit binary numbers!
- Thus, we will introduce a six-decimal-digit representation of real numbers that parallels the IEEE 754 standard
  - All the design features of this six-digit representation are faithful to parallel design decisions for
  - All the weaknesses of this six-decimal-digit representation parallel the weaknesses of the 64-bit binary representation

A simple floating-point representation

- Here is our representation:

±EENMMM

- For EE going from 00 to 98, this represents:

$$\pm N.MMM \times 10^{EE-49}$$

- For storing non-zero values, we will require that $N \neq 0$

Smallest and largest positive values

- The smallest positive number in this representation is

+0001000

- This represents

$$+1.000\times10^{00-49}=10^{-49}$$

- The largest positive number in this representation is

+9899999

- This represents

$$+9.999\times10^{98-49}=9.999\times10^{49}\approx10^{50}$$

The range and relative error

- Thus, this format allows us to represent real numbers on the range

$$\left[10^{-49}, 10^{50}\right]$$

with four digits of precision

  - Thus, we can represent essentially all such numbers with a relative error no greater than

$$5 \times 10^{-4} = 0.0005$$

or 0.05% relative error

- All things considered, that's not bad for six decimal digits
  - Essentially, we can actually represent $\sqrt{\text{googol}}$

Addition

- Suppose we want to add or multiply such numbers
  - Such operations take place as per normal,
    but the result must be stored in the given format
  - For example, suppose we want to add these two numbers:

+627235 +626323

$$7.235 \times 10^{62-49}$$

$$+\ \underline{6.323 \times 10^{62-49}}$$

$$13.558 \times 10^{62-49} = 1.3558 \times 10^{63-49}$$

  - We must, however, round this to four significant digits:

+631356

    - Note that we didn't need to calculate the exponent

Addition

- Even if the exponents are different, addition is straight-forward
  - For example, suppose we want to add these two numbers:

+2355486 +2566323

$$5.486\times10^{23-49} \qquad 0.05486\times10^{25-49}$$
$$+\ 6.323\times10^{25-49} \qquad +\ 6.323\times10^{25-49}$$
$$6.37786\times10^{25-49}$$

  - We must, however, round this to four significant digits:

+2566378

Addition

- Consider this example:
  - For example, suppose we want to add these two numbers:

```
+5499285      +5002911
```

$$9.285\times10^{54-49} \qquad 9.285 \times10^{54-49}$$

$$+\ \underline{2.911\times10^{50-49}} \qquad +\ \underline{0.0002911\times10^{54-49}}$$

$$9.2852911\times10^{54-49}$$

  - We must, however, round this to four significant digits:

```
+5499285
```

Weaknesses

- We note that:
  - In the first two examples, we had to round
    - This introduces an error into our result as a consequence of performing one arithmetic operation
      - How many arithmetic operations are required for Gaussian elimination and backward substitution?
  - In the third example, we had the following:

$$x + y = x$$

    - If you were shown this in secondary school, your first reaction would be that $y = 0$

Adding three numbers

- Consider this example:
  - For example, suppose we want to add these three numbers:

    +545234 +522348 +501593
  - Without explicitly calculating the result,
    the sum of the first two is $5.25748 \times 10^{54-49}$
    - This is stored as

      +545257
  - Without explicitly calculating the result,
    adding the third onto this is $5.2571593 \times 10^{54-49}$
    - This is stored as

      +545257

Adding three numbers

- Consider this example:
  - For example, suppose we want to add these three numbers:

    +545234 +522348 +501593
  - Instead, add the second two first,
    the sum of which is $2.36393 \times 10^{52-49}$
    - This is stored as

      +522364
  - Without explicitly calculating the result,
    adding this to the first is $5.25764 \times 10^{54-49}$
    - This is stored as

      +545258
  - Compare this with, and which is better?

    +545257

Adding three numbers

- If we were not to do intermediate rounding, the sum of

  +5455234 +5222348 +5001593

  has the exact value $5.2576393 \times 10^{54-49}$
- Rounded to four digits, this is $5.258 \times 10^{54-49}$
  - Consequently, the second is the result a *better* means of calculating this sum

    +5455257 +5455258
- What this means is that for floating-point numbers:

$$(x + y) + z \neq x + (y + z)$$

- When adding positive numbers,
  it is best to add them from smallest to largest

Kahan summation algorithm

- Issue: sorting a large list of numbers can be expensive
- Prof. Kahan, who led the IEEE 754 standards committee, came up with an algorithm for adding positive floating-point numbers without sorting them:

```
double kahan_summation( double const array[], std::size_t const capacity ) {
    double sum{ 0.0 };
    double compensation{ 0.0 };

    for ( std::size_t k{0}; k < capacity; ++k ) {
        double modified_array_k{ array[k] - compensation };
        double next_sum{ sum + modified_array_k };
        compensation = (next_sum - sum) - modified_array_k;
        sum = next_sum;
    }

    return sum;
}
```

  - See https://en.wikipedia.org/wiki/Kahan_summation_algorithm

Multiplication

- Consider this example:
  - For example, suppose we want to multiply these two numbers:

+503251 +512048

$$3.251\times10^{50-49} \qquad 3.251\times10^{1}$$
$$\times\ \underline{2.048\times10^{51-49}} \qquad \times\ \underline{2.048\times10^{2}}$$
$$6.658048\times10^{1+2} \quad = 6.658048\times10^{3}$$
$$= 6.658048\times10^{52-49}$$

  - We must, however, round this to four significant digits:

+526658

Reciprocals

- Suppose we are finding the reciprocal of a number:

+495492

  - Thus, we calculate:

$$\frac{1}{5.492\times10^{49-49}}=\frac{1}{5.492}\approx 0.1820830299$$

$$=1.820830299\times10^{-1}$$

$$=1.820830299\times10^{48-49}$$

  - We must, however, round this to four significant digits:

+481821

Reciprocals

- What is the reciprocal of

  +481821

  - Again, we calculate:

$$\frac{1}{1.821\times10^{48-49}}=\frac{1}{0.1821}\approx 5.491488193$$

$$=5.491488193\times10^{0}$$

$$=5.491488193\times10^{49-49}$$

  - We must, however, round this to four significant digits:

    +495491

  - Notice that the above number was the calculate reciprocal of:

    +495492

Further weaknesses

- Consequently, for floating-point numbers,
  it is also not true that

$$\frac{1}{1/x} \neq x$$

- You may additionally deduce that

$$\frac{xy}{y} \neq x$$

  if we calculate xy first

Subtractive cancellation

- Suppose we want to approximate the derivative of $\sin(x)$ at $x = 1$

$$\frac{\sin(1.001) - \sin(1)}{0.001} = 0.539881480$$

$$= \frac{+488420109 - +488414710}{+461000}$$

$$\frac{\sin(+491001) - \sin(+491000)}{+461000} = \frac{+488420 - +488415}{+461000}$$

$$= \frac{+455000}{+461000}$$

$$= +485000$$

$$= 0.5$$

- Even though each error is 0.05%, the relative error jumped to 7.4%
  - Rule: do not use subtraction of very similar numbers

Issue: Subtracting two numbers are relatively close to each other

Summary of floating-point arithmetic

- Consequently, each arithmetic operation:
  - May introduce additional rounding errors
  - May not even follow the commonly held rules of algebra
- We must therefore take steps to mitigate, as much as is reasonably possible, the effect of using floating-point numbers

Increasing precision

- One step is to store more digits:
  - The data type `float` is not very precise
    - Never use it for engineering computations, unless it deals with images to be viewed by humans
  - The name `double` comes from double-precision floating-point numbers, suggesting it has approximately twice as many significant bits
    - This is usually appropriate for most engineering computations with appropriately designed algorithms
  - The type `long double` occupies 10 bytes instead of 8 bytes, yielding even more bits of accuracy
    - This, however, is not necessary in most cases

Increasing precision

- The floating-point registers on some processors may actually store `long double` instead of `double`
  - Consequently, your calculations may be more accurate than you expect ☺
- Problem: as soon as they are stored back in main memory, they must be rounded to the precision of `double`

Floating point unit

Unique representations: normalization

- Recall our representation:

±EENMMM

- Here, EE went from 00 to 98, this represents:

$$\pm N.MMM \times 10^{EE-49}$$

- Why did we require that $N \neq 0$?
  - If we didn't make this restriction, there could be multiple representations of the same value:

+4910000

+5001000

+5100100

+5200001

Comparison operators

- Thus, our representation ensures that two floating-point numbers are equal if and only if their digits are equal
- Question: Of these two positive numbers, which is the larger?

+1555932

+2377184

- The hard way is to determine that these represent

$$5.932 \times 10^{-34} \text{ and } 7.184 \times 10^{-26}$$

  - Note, however, if we just compare the two numbers as integers, we immediately see the second is the largest integer
  - Thus, if one floating-point number interpreted as an integer is greater than another, then the floating-point number, too, is greater than the other
  - Thus, even comparisons can be done easily in hardware

Sorting floating-point numbers

- For example, sort these as floating-point numbers

```
+588431 +836273 +290481 +568189 +142515 +223636 +568866 +857600
```

- We don't need to know what they represent,
  just sort them as integers

```
+142515 +223636 +290481 +568189 +568866 +588431 +836273 +857600
```

Going outside our bounds

- Recall that with integer arithmetic:
  - Overflow and underflow cause values to wrap
  - Division by zero causes an exception that terminates execution
- For floating point numbers, it would be horrible if adding to the largest float results in the largest negative float…
  - It would also be sub-optimal if calculations resulted in the termination of the program…

Representation of zero

- Recall we that we required N to be non-zero:

  ±EENMMM

  - We required that EE goes from 00 to 98 to represent

    $$\pm N.MMM \times 10^{EE-49}$$

- We will use the following to represent plus or minus zero:

  ±0000000

  - Why two representations of zero?
    - The first, +0000000 represents small positive values
    - The second, -0000000 represents small negative values
  - Of course, $x + +0 = x$ and $x + -0 = x$ for all non-zero x

Infinity

- Recall we did not allow the exponent to be 99:

  ±EENMMM

  - We required that EE goes from 00 to 98 to represent

$$\pm N.MMM \times 10^{EE-49}$$

- We will use the following to represent plus or minus infinity:

  ±9900000

  - Thus, if we add two or multiply two large numbers that cannot b represented by a floating-point number, this results in +∞
- Note that we can still do arithmetic with infinity:

$+\infty + x = +\infty$ if $x > -\infty$ $\quad -\infty + x = -\infty$ if $x < +\infty$

$+\infty \times x = +\infty$ if $x > 0$ $\quad +\infty \times x = -\infty$ if $x < 0$

$1/+0 = +\infty$ $\quad 1/-0 = -\infty$

$1/+\infty = +0$ $\quad 1/-\infty = -0$

Not-a-number

- What happens if we calculate an indeterminant result:
  - Suppose we calculate +0/+0 or $+\infty/+\infty$ or $+\infty + -\infty$?

- All these are *indeterminant*, meaning they do not even evaluate evaluate to possibly a large or small number
  - In numerical algorithms, such results are called *not-a-number*
  - To represent a *non-a-number* or NaN, we will use

`+991000`

- Thus, we are currently using

`±990000` and `+991000`

  - All other such numbers are used reserved in the IEEE 754 standard

Denormalized numbers

- Finally, the smallest number in our representation is

  ±00 1000

  - This represents

$$\pm 1.000 \times 10^{-49}$$

- What if we divide this by two?
  - This would immediately go from four significant digits to zero…
- Instead, if the exponent is 00, we will allow the first digit to be zero:

$$\pm 0.MMM \times 10^{-49}$$

  - These are called *denormalized numbers*
  - These numbers have a larger relative error as there are only three, two or one significant digits

Denormalized numbers

- For example,

±00000MMM	$\pm 0.MMM \times 10^{-49} = \pm M.MM \times 10^{-50}$
±00000MM	$\pm 0.0MM \times 10^{-49} = \pm M.M \times 10^{-51}$
±00000M	$\pm 0.00M \times 10^{-49} = \pm M \times 10^{-52}$
±00000000	$\pm 0 \times 10^{-49} = 0$

- With each additional leading zero,
  the relative error is increased, but it is better than 0
- Also, this has one fringe benefit:

$$x - y = 0 \text{ if and only if } x = y$$

Six-digit summary

- Design:

±990000	$\pm\infty$
±991000	not-a-number or NaN
±EENMMM	$\pm N.MMM \times 10^{EE-49}$
±000MMM	$\pm 0.MMM \times 10^{-49}$
±000000	± 0

- Problems:

  Real numbers can only be represented with a relative error no greater than 0.05%

  Each calculation introduces additional error

  $x + y = x$ even if $y \neq 0$

  $(x + y) + z \neq x + (y + z)$

  $1/(1/x) \neq x$

Double-precision floating-point numbers

- The double-precision floating-point representation is as follows:
  `bbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbb`
- For `00000000001` to `11111111110`, this represents

$$(-1)^b \, 1.b_1 b_2 b_3 b_4 b_5 \cdots b_{52} \times 2^{bbbbbbbbbbb - 01111111111}$$

- For `00000000000`, this represents

$$(-1)^b \, 0.b_1 b_2 b_3 b_4 b_5 \cdots b_{52} \times 2^{1 - 01111111111}$$

- We also have that
  `b11111111111` `0000000000000000000000000000000000000000000000000000`

$$(-1)^b \, \infty$$

- We also have, a reserved representation for indeterminant results:
  `011111111111` `0000000000000000000000000000000000000000000000000001`
  - This is called *not-a-number* or NaN

Properties of double

- From this, we may deduce that the largest value in magnitude is
  `b11111111110`1111111111111111111111111111111111111111111111111
  - This represents

$$(-1)^b\, 1.11111\cdots1\times 2^{1111111111_2} = (-1)^b\, 1.11111\cdots1\times 2^{1023} \approx (-1)^b\, 2^{1024}$$

  - This is approximately $\pm 1.80 \times 10^{308}$
- The smallest normalized number is
  `b00000000001`0000000000000000000000000000000000000000000000000
  - This represents

$$(-1)^b\, 1.000\cdots0\times 2^{1-1111111111_2} = (-1)^b\, 2^{-1022}$$

  - This is approximately $\pm 2.23 \times 10^{-308}$
- The smallest non-zero denormalized number is
  `b00000000000`0000000000000000000000000000000000000000000000001
  - This represents

$$(-1)^b\, 0.000\cdots01\times 2^{1-1111111111_2} = (-1)^b\, 2^{-52}\times 2^{-1022} = (-1)^b\, 2^{-1074}$$

  - This is approximately $\pm 4.94 \times 10^{-324}$

Properties of double

- You don't have to memorize these
  - Approximately 10^{-300} to 10^{300} is good enough
- The maximum relative error is $2^{-53} \approx 1.11 \times 10^{-16}$
  - This is also known as *machine epsilon*

Addition

- Let's try adding two such numbers:

```
00010000011101011000000000000000000000000000000000000000000000
00010000100011011000000000000000000000000000000000000000000000
```

- Note that the exponent is one higher for the second

$$\begin{array}{r} 0.101011 \\ +\,1.11011 \\ \hline 10.100001 \end{array}$$

- Thus, the sum will:
  - Have one higher exponent than the larger
  - The mantissa will be `0100001`

```
00010000100101000100000000000000000000000000000000000000000000
```

Addition

- Let's try adding two such numbers:

```
01110011011100110110000000000000000000000000000000000000000000
01110011010001010100000000000000000000000000000000000000000000
```

- Note that the exponent is three higher for the first

$$\begin{array}{r} 1.0011011 \\ +\ 0.001010101 \\ \hline 1.011000001 \end{array}$$

- Thus, the sum will:
  - The same exponent as the larger number
  - The mantissa will be `011000001`

```
01110011011101100000100000000000000000000000000000000000000000
```

Multiplication

- Let's try multiplying two such numbers:

```
0100000000010100000000000000000000000000000000000000000000000000
0011111111101011000000000000000000000000000000000000000000000000
```

- The first is 2 higher than the bias, the second is one lower

$$\begin{array}{rr} 101 & 0.11011 \\ \times\ \underline{0.11011} & \times\ \underline{101} \\ & 11011 \\ & +\ \underline{1101100} \\ & 100.00111 \end{array}$$

- Thus, the product will:
  - An exponent two higher than the bias
  - The mantissa will be `0000111`

```
0100000000010000111000000000000000000000000000000000000000000000
```

Multiplication

- We can check this in Matlab:

```
  0100000000010100000000000000000000000000000000000000000000000000
× 0011111111101011000000000000000000000000000000000000000000000000
  0100000000010000111000000000000000000000000000000000000000000000
```

- The first number is 5,

  the second is 0.5 + 0.25 + 0.0625 + 0.03125 = 0.84375
- The product of these is 4.21875
- In Matlab, we have:

```
>> format hex
>> 5
     ans = 4014000000000000
>> 0.84375
     ans = 3feb000000000000
>> 5*0.84375
     ans = 4010e00000000000
```

Summary

- Following this topic, you now
  - Understand the difference between fixed-point and floating-point representations
  - Understand the representation of numbers, including infinity, zero, and not-a-number
  - Are aware of the weaknesses of floating-point numbers
  - Understand the double-precision floating-point representation
  - Are aware of how to perform simple operations with such representations

References

[1] https://en.wikipedia.org/wiki/Double-precision_floating-point_format

[2] https://en.wikipedia.org/wiki/Floating-point_arithmetic

[3] https://en.wikipedia.org/wiki/IEEE_754

[4] https://en.wikipedia.org/wiki/William_Kahan

Acknowledgments

Whoever commented on my voice. Please let me know who you are.

Juliette Rocco for noting the error on Slide 14.

Colophon

These slides were prepared using the Cambria typeface. Mathematical equations use Times New Roman, and source code is presented using `Consolas`.
Mathematical equations are prepared in MathType by Design Science, Inc.

Examples may be formulated and checked using Maple by Maplesoft, Inc.

The photographs of flowers and a monarch butter appearing on the title slide and accenting the top of each other slide were taken at the Royal Botanical Gardens in October of 2017 by Douglas Wilhelm Harder. Please see

https://www.rbg.ca/

for more information.