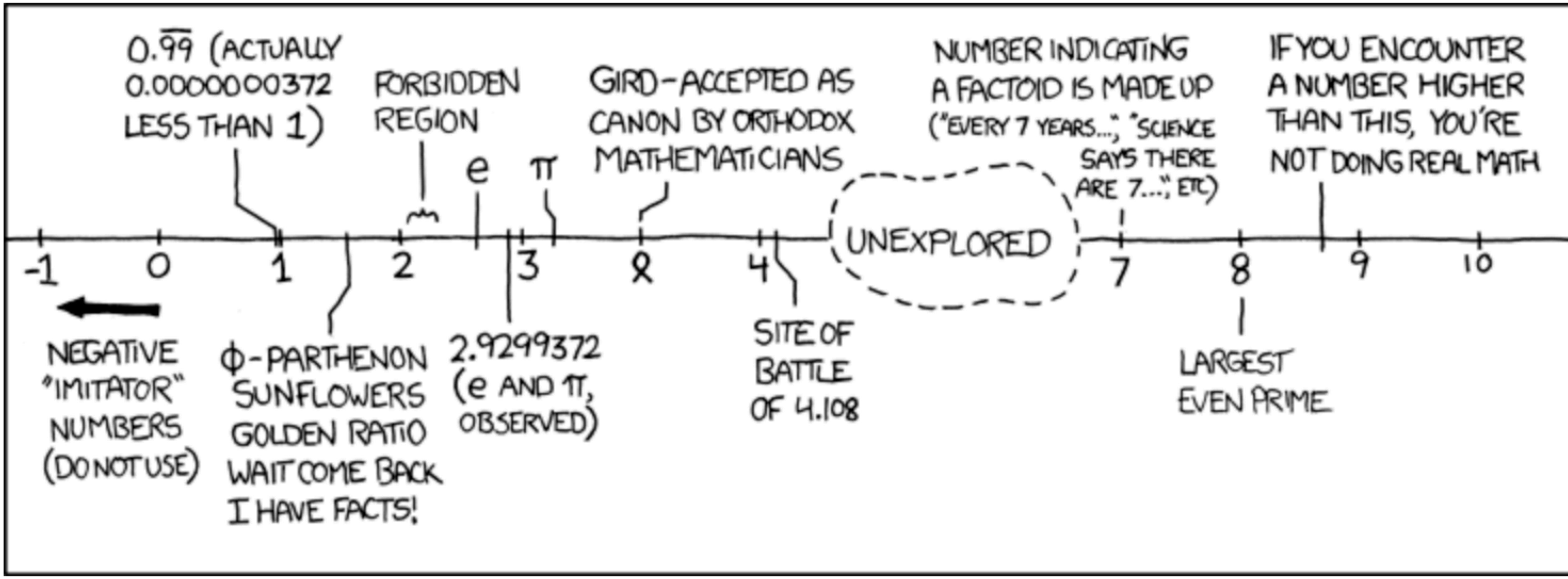
(unavoidable) Errors, Ints and Floats



http://xkcd.com/899/

Peter Beerli

```
clc
format long e
a=2.6+0.2
b=a+0.2
c=b+0.2
d=c-3.2
if d~=0
    e=1/d
end
```

a = 2.800000000000000e+00
b = 3.000000000000000e+00
c = 3.200000000000001e+00
d = 4.440892098500626e-16
e = 2.251799813685248e+15

```
a=2.6+0.6
b=a+0.6
c=b+0.6
e=c+0.6
e=d-5
```

a = 3.200000000000000e+00
b = 3.800000000000000e+00
c = 4.400000000000001e+00
d = 5
e = 0

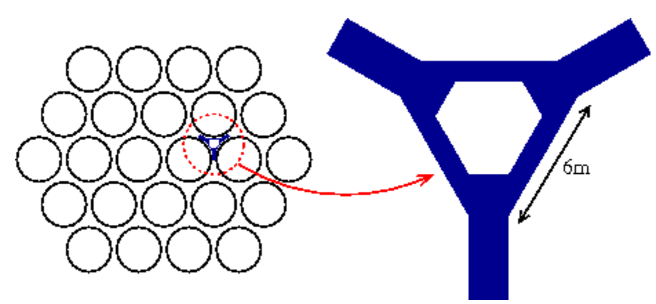
Numerical "Bugs"

- Obvious: Software has bugs.
  - A software bug causes deterministic errors in program execution. Given the same initial data, a specific sequence of actions results in the same erroneous outcome.
  - Software bugs may appear to be random in some situations, because the symptoms of the error may depend on the state of the computer, especially the data in memory, when the error occurs.
- Not-So-Obvious: Unavoidable numerical error
  - Roundoff error
  - Truncation error

Some Disasters

Some disasters attributable to bad numerical computing (Douglas Arnold)

- The Patriot Missile failure, in Dharan, Saudi Arabia, on February 25, 1991 which resulted in 28 deaths, is ultimately attributable to poor handling of rounding errors.
- The explosion of the Ariane 5 rocket just after lift-off on its maiden voyage off French Guiana, on June 4, 1996, was ultimately the consequence of a simple overflow.
- The sinking of the Sleipner A offshore platform in Gandsfjorden near Stavanger, Norway, on August 23, 1991, resulted in a loss of nearly one billion dollars. It was found to be the result of inaccurate finite element analysis.



The Pentium™ FDIV Bug

A-A/B*B=0

tium Chips

In some complex division problems, annoying errors.

corrected.

Some computer users said they believed that Intel had not acted quickly enough after discovering the error.

"Intel has known about this since the summer; why didn't they tell anyone?" said Andrew Schulman, the author of a series of technical books on PC's. "It's a hot issue, and I don't think they've handled this well.

The company said that after it discovered the problem this summer, it ran months of simulations of different applications, with the help of outside experts, to determine whether the problem was serious.

The Pentium error occurs in a portion of the chip known as the floating point unit, which is used for extremely precise computations. In rare cases, the error shows up in the result of a division operation.

Intel said the error occurred because of an omission in the translation of a formula into computer

Close, but Not Close Enough

The owners of computers that use Intel's Pentium microprocessors have found that the chips sometimes do not perform division calculations accurately enough.

The problems arise when the chip has to round a number in a preliminary calculation to get the final result, a task that all processors normally perform. In these cases, however, the Pentium's figures are exact to only 5 digits, not 16, as are those of other computer processors. The Pentium's error, while small, can be 10 billion times as large as those of most chips.

Here is an example of the way the imprecise rounding changes the results of a calculation and the way the deviation from the expected result is calculated.

PROBLEM

4,195,835 – [(**4,195,835 ÷ 3,145,727**) x 3,145,727]

CORRECT CALCULATION

= 4,195,835 – [(**1.3338204**) x 3,145,727] = **0**

PENTIUM'S CALCULATION

= 4,195,835 – [(**1.3337391**) x 3,145,727] = **256**

DEVIATION

256 ÷ 4,195,835 = 6.1 x 10^{-5}, or 61/100,000

Source: Cleve Moler, the Mathworks Inc.

Intel Timeline

June 1994	Intel engineers discover the division error. Managers decide the error will not impact many people. Keep the issue internal.
June 1994	Dr Nicely at Lynchburg College notices computation problems
Oct 19, 1994	After months of testing, Nicely confirms that other errors are not the cause. The problem is in the Intel Processor.
Oct 24, 1994	Nicely contacts Intel. Intel duplicates error.
Oct 30, 1994	After no action from Intel, Nicely sends an email

```
FROM:  Dr. Thomas R. Nicely
       Professor of Mathematics
       Lynchburg College
       1501 Lakeside Drive
       Lynchburg, Virginia 24501-3199

       Phone:     804-522-8374
       Fax:       804-522-8499
       Internet:  nicely@acavax.lynchburg.edu

TO:    Whom it may concern

RE:    Bug in the Pentium FPU

DATE:  30 October 1994

It appears that there is a bug in the floating point unit
(numeric coprocessor) of many, and perhaps all, Pentium
processors.

In short, the Pentium FPU is returning erroneous values
for certain division operations. For example,
     0001/824633702441.0
is calculated incorrectly (all digits beyond the eighth
significant digit are in error). This can be verified in
compiled code, an ordinary spreadsheet such as Quattro Pro
or Excel, or even the Windows calculator (use the
scientific mode), by computing
     00(824633702441.0)*(1/824633702441.0),
which should equal 1 exactly (within some extremely small
rounding error; in general, coprocessor results should
contain 19 significant decimal digits).  However, the
Pentiums tested return
     0000.999999996274709702
.
.
.
```

Nov 1, 1994	Software company Phar Lap Software receives Nicely's email. Sends to collegues at Microsoft, Borland, Watcom, etc. decide the error will not impact many people. Keep the issue internal.
Nov 2, 1994	Email with description goes global.
Nov 15, 1994	USC reverse-engineers the chip to expose the problem. Intell still denies a problem. Stock falls.
Nov 22, 1994	CNN *Moneyline* interviews Intel. Says the problem is minor.
Nov 23, 1994	The MathWorks develops a fix.
Nov 24, 1994	New York Times story. Intel still sending out flawed chips. Will replace chips only if it caused a problem in an important application.
Dec 12, 1994	IBM halts shipment of Pentium based PCs
Dec 16, 1994	Intel stock falls again.
Dec 19, 1994	More reports in the NYT: lawsuits, etc.
Dec 20, 1994	Intel admits. Sets aside $420 million to fix.

Numerical Errors

- Roundoff occurs in a computer calculation whenever digits to the right of decimal point are discarded.
- The digits in a decimal point (0.3333...) are lost (0.3333) because there is a limit on the memory available for storing one numerical value.
- Truncation error occurs whenever a numerical computation uses formulas involving discrete values as an approximate a continuous function.

Uncertainty: well or ill-conditioned?

Errors in input data can cause uncertain results.

- Input data can be from experimental measurements that subject to measurement error.
- Input data can be rounded when they are first stored in computer memory (log transformed). These lead to a certain variation in the results.
- well-conditioned: numerical results are insensitive to small variations in the input
- ill-conditioned: small variations lead to drastically different numerical calculations (a.k.a. poorly conditioned)

Exercise: Store a Integer

- What are bit, byte, and word?
- What is the decimal value for the binary number 1101? Check your result using the built-in function bin2dec.
- Express the decimal value 25 as a binary number. Check your result using the built-in function dec2bin.
- What is the absolute upper limit on the largest unsigned integer than can be stored as a 16-bit binary number?
- What is the range for signed integer for a 16-bit computer?

Bits, Bytes, and Words

base 10	conversion	base 2
1	$1 = 2^0$	0000 0001
2	$2 = 2^1$	0000 0010
4	$4 = 2^2$	0000 0100
8	$8 = 2^3$	0000 1000
9	$8 + 1 = 2^3 + 2^0$	0000 1001
10	$8 + 2 = 2^3 + 2^1$	0000 1010
27	$16 + 8 + 2 + 1 = 2^4 + 2^3 + 2^1 + 2^0$	$\underbrace{\text{0001 1011}}_{\text{one byte}}$

Digital Storage of Integers (1)

As a prelude to discussing the binary storage of floating point values, first consider the binary storage of integers.

- Integers can be exactly represented by base 2
- Typical size is 16 bits

Today it is more likely to see 32 bit or 64 bit

- $2^{16} = 65536$ is largest 16 bit integer
- $[-32768, 32767]$ is range of 16 bit integers in twos complement notation
- 32 bit and larger integers are available

Digital Storage of Floating Point Numbers (1)

Numeric values with non-zero fractional parts are stored as **floating point numbers**.

All floating point values are represented with a normalized scientific notation[1].

Example:

$$12.2792 = \underbrace{0.123792}_{\text{Mantissa}} \times 10^{2}$$

Exponent

[1]The IEEE Standard on Floating Point arithmetic defines a normalized *binary* format. Here we use a simplified *decimal* (base ten) format that, while abusing the standard notation, expresses the essential ideas behind the decimal to binary conversion.

Digital Storage of Floating Point Numbers (2)

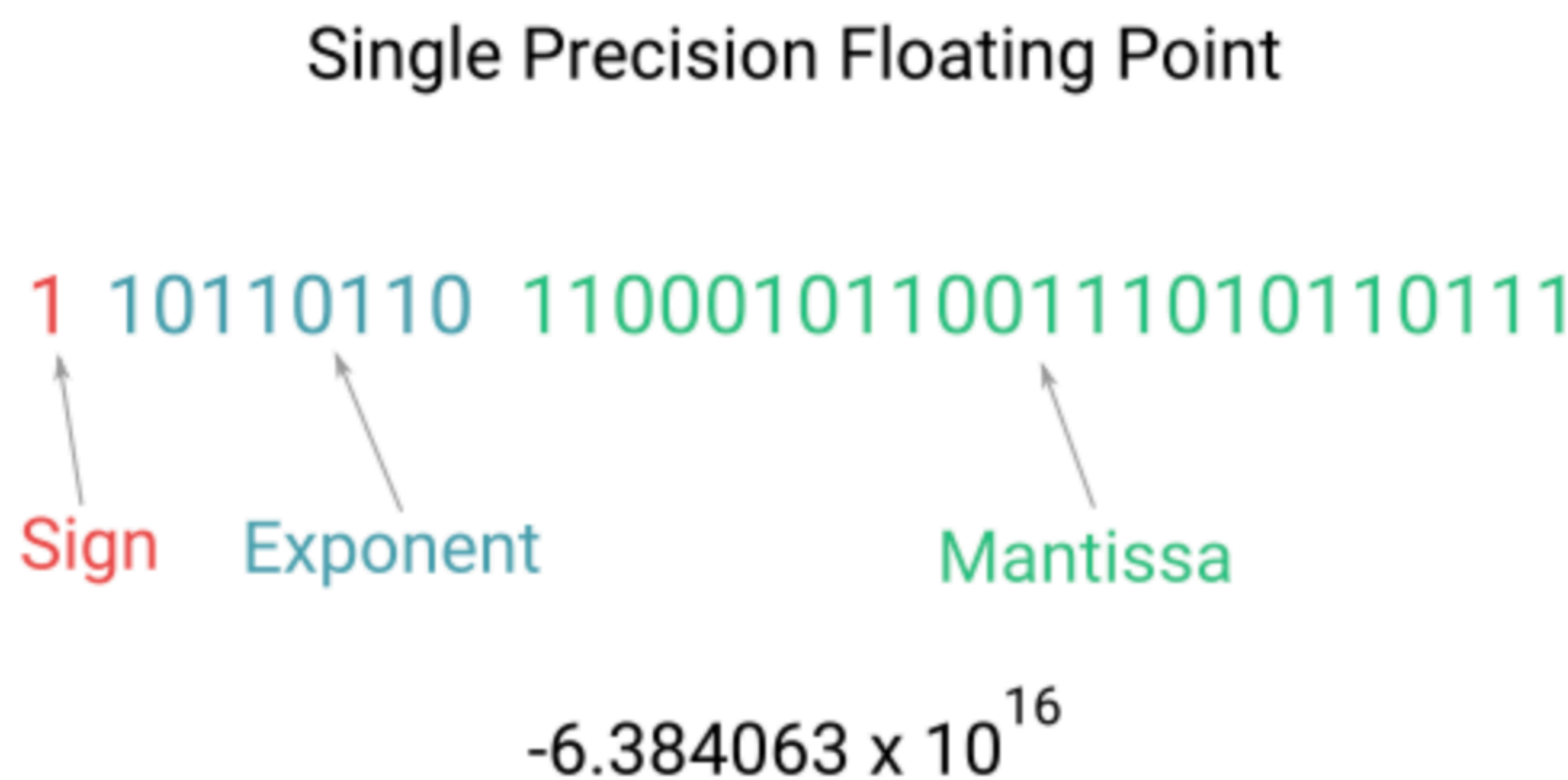
Floating point values have a fixed number of bits allocated for storage of the mantissa and a fixed number of bits allocated for storage of the exponent.

Two common precisions are provided in numeric computing languages

Precision	Bits for mantissa	Bits for exponent
Single	23	8
Double	53	11

Single precision, which uses 32 bits and has the following layout:

- **1 bit** for the sign of the number. 0 means positive and 1 means negative.
- **8 bits** for the exponent.
- **23 bits** for the mantissa.



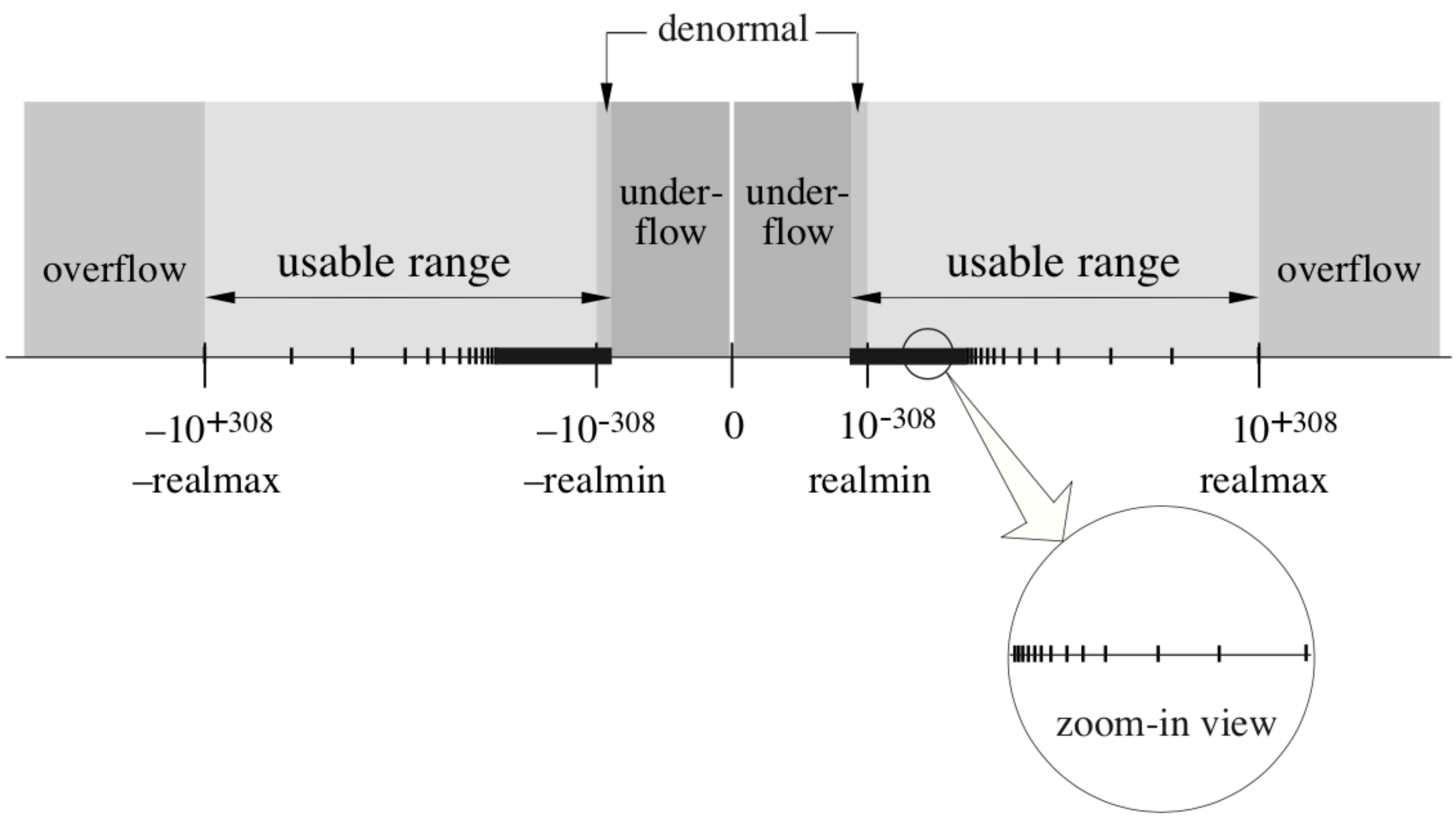
Double precision, which uses 64 bits and has the following layout.

- **1 bit** for the sign of the number. 0 means positive and 1 means negative.
- **11 bits** for the exponent.
- **52 bits** for the mantissa.

Special numbers

Zero	0 00000000 00000000000000000000000
Negative Zero	1 00000000 00000000000000000000000
Infinity	0 11111111 00000000000000000000000
Negative Infinity	1 11111111 00000000000000000000000
Not a Number (NaN)	0 11111111 00001000000000100001000

Floating Point Number Line



Overflow

- The built-in MATLAB variable realmax corresponds to the overflow limits, the floating-point number having a binary representation with all of its usable bits turned on.
- Numbers with magnitudes greater than roughly 10^{+308} do not exist on the number line of 64-bit floating-point values.
- Any MATLAB calculation resulting in a magnitude greater than $\sim 10^{+308}$ causes an overflow error.
- A number greater than realmax is assigned the special value inf. Try 10*realmax.

Underflow

- Any MATLAB calculation that results in a magnitude smaller than, $\sim 10^{-308}$ and not exactly equal to zero, cannot be represented by a 64-bit number. There are no double-precision numbers between zero and roughly $\pm 10^{-308}$.
- This hole in the number line is the range where underflow errors occur.
- The built-in MATLAB variable realmin corresponds to the underflow limits, the smallest floating-point number than can be stored without a loss of prevision.

Denormal

- It is possible to store a floating-point number smaller than realmin if bits that are normally associated with the mantissa are used by the exponent. Try realmin/10.
- Such a number is called a denormal, because it is not stored in the normalized format used for other values on the floating-point number line.

- When a calculation results in a value smaller than realmin, there are two types of outcomes:
  - If the results is slightly less than realmin, the number is stored as a denormal, but you lose the precision (why?).
  - When the result is significantly smaller than realmin and cannot be sotred as a denormal, it is stored as exactly zero.

Floating Point Representation

Michael L. Overton

1 Computer Representation of Numbers

Computers which work with real arithmetic use a system called *floating point*. Suppose a real number x has the binary expansion

$$x = \pm m \times 2^E, \quad \text{where } 1 \leq m < 2$$

and

$$m = (b_0.b_1 b_2 b_3 \ldots)_2.$$

To store a number in floating point representation, a computer word is divided into 3 fields, representing the sign, the exponent E, and the significand m respectively. A 32-bit word could be divided into fields as follows: 1 bit for the sign, 8 bits for the exponent and 23 bits for the significand. Since the exponent field is 8 bits, it can be used to represent exponents between -128 and 127. The significand field can store the first 23 bits of the binary representation of m, namely

$$b_0.b_1 \ldots b_{22}.$$

If $b_{23}, b_{24}, \ldots$ are not all zero, this floating point representation of x is not exact but approximate. A number is called a *floating point number* if it can be stored *exactly* on the computer using the given floating point representation scheme, i.e. in this case, $b_{23}, b_{24}, \ldots$ are all zero. For example, the number

$$11/2 = (1.011)_2 \times 2^2$$

would be represented by

0	$E = 2$	1.0110000000000000000000

,

$+(1*2^0 + 0*2^{-1} + 1*2^{-2} + 1*2^{-3}) * 2^2 = (1+1/4+1/8) * 4 = 5.5$

and the number

$$71 = (1.000111)_2 \times 2^6$$

$+(1*2^0 + 0*2^{-1} + 0*2^{-2} + 0*2^{-3} + 1*2^{-4} + 1*2^{-5} + 1*2^{-6}) * 2^6 =$
$+ (1+1/16 + 1/32 + 1/64) * 64 =$
71

would be represented by

0	$E = 6$	1.00011100000000000000000

.

To avoid confusion, the exponent E, which is actually stored in a binary representation, is shown in decimal for the moment.

The floating point representation of a nonzero number is unique as long as we require that $1 \leq m < 2$. If it were not for this requirement, the number $11/2$ could also be written

$$(0.01011)_2 \times 2^4$$

and could therefore be represented by

0	$E = 4$	0.01011000000000000000000

.

However, this is not allowed since $b_0 = 0$ and so $m < 1$.

$+(0*2^0 + 0*2^{-1} + 1*2^{-2} + 0*2^{-3} + 1*2^{-4} + 1*2^{-5}) * 2^4 =$
$=(0+1/4+1/16 + 1/32) * 16 = 5.5$

A more interesting example is

$$1/10 = (0.0001100110011\ldots)_2.$$

Since this binary expansion is infinite, we must *truncate* the expansion somewhere. (An alternative, namely *rounding*, is discussed later.) The simplest way to truncate the expansion to 23 bits would give the representation

0	$E = 0$	0.00011001100110011001100110

,

but this means $m < 1$ since $b_0 = 0$. An even worse choice of representation would be the following: since

$$1/10 = (0.00000001100110011\ldots)_2 \times 2^4,$$

the number could be represented by

0	$E = 4$	0.00000001100110011001100110

.

How to generate the binary representation

For example: 329.390625

329 → $256 + 64 + 8 + 1$

$2^8 + 2^6 + 2^3 + 2^0$

101001001

0.390625

0.390625	* 2 = 0.78125	0
0.78125	* 2 = 1.5625	1
0.5625	* 2 = 1.125	1
0.125	* 2 = 0.25	0
0.25	* 2 = 0.5	0
0.5	* 2 = 1	1
0		

101001001 . 011001

011001

329.390625 → 1 . 01001001011001 * 2^8

A more interesting example is

$$1/10 = (0.0001100110011\ldots)_2.$$

Since this binary expansion is infinite, we must *truncate* the expansion somewhere. (An alternative, namely *rounding*, is discussed later.) The simplest way to truncate the expansion to 23 bits would give the representation

$$\begin{array}{|c|c|c|}\hline 0 & \quad E = 0 \quad & 0.0001100110011001100110 \\ \hline \end{array},$$

but this means $m < 1$ since $b_0 = 0$. An even worse choice of representation would be the following: since

$$1/10 = (0.00000001100110011\ldots)_2 \times 2^4,$$

the number could be represented by

$$\begin{array}{|c|c|c|}\hline 0 & \quad E = 4 \quad & 0.0000000110011001100110 \\ \hline \end{array}.$$

0.1 * 2 = 0.2	0
0.2 * 2 = 0.4	0
0.4 * 2 = 0.8	0
0.8 * 2 = 1.6	1
0.6 * 2 = 1.2	1
0.2 * 2 = 0.4	0
0.4 * 2 = 0.8	0
0.8 * 2 = 1.6	1
0.6 * 2 = 1.2	1
0.2 * 2 = 0.4	0
0.4 * 2 = 0.8	0
…..	

This is clearly a bad choice since less of the binary expansion of $1/10$ is stored, due to the space wasted by the leading zeros in the significand field. *This is the reason why $m < 1$, i.e. $b_0 = 0$, is not allowed.* The only allowable representation for $1/10$ uses the fact that

$$1/10 = (1.100110011\ldots)_2 \times 2^{-4},$$

giving the representation

0	$E = -4$	1.10011001100110011001100110

.

This representation includes *more of the binary expansion of* $1/10$ than the others, and is said to be *normalized*, since $b_0 = 1$, i.e. $m > 1$. Thus none of the available bits is wasted by storing leading zeros.

We can see from this example why the name *floating point* is used: the binary *point* of the number $1/10$ can be *floated* to any position in the bitstring we like by choosing the appropriate exponent: the normalized representation, with $b_0 = 1$, is the one which should be always be used when possible. It is clear that an irrational number such as π is also represented most accurately by a normalized representation: significand bits should not be wasted by storing leading zeros.

In a normal floating-point value, there are no leading zeros in the significand; instead leading zeros are moved to the exponent. So 0.0123 would be written as 1.23×10^{-2}. Denormal numbers are numbers where this representation would result in an exponent that is below the minimum exponent (the exponent usually having a limited range). Such numbers are represented using leading zeros in the significand.

The significand (or mantissa) of an IEEE floating point number is the part of a floating-point number that represents the significant digits. For a positive normalized number it can be represented as $m_0.m_1m_2m_3...m_{p-2}m_{p-1}$ (where m represents a significant digit and p is the precision, and m_0 is non-zero). Notice that for a binary radix, the leading binary digit is always 1. In a **denormal number**, since the exponent is the least that it can be, zero is the leading significand digit ($0.m_1m_2m_3...m_{p-2}m_{p-1}$), allowing the representation of numbers closer to zero than the smallest normal number. A floating point number may be recognized as denormal whenever its exponent is the least value possible.

By filling the underflow gap like this, significant digits are lost, but not as abruptly as when using the *flush to zero on underflow* approach (discarding all significant digits when underflow is reached). Hence the production of a denormal number is sometimes called **gradual underflow** because it allows a calculation to lose precision slowly when the result is small.

Rounding error

https://en.wikipedia.org/wiki/Rounding

Roundoff error occurs because of the computing device's inability to deal with certain numbers. Such numbers need to be rounded off to some near approximation which is dependent on the word size used to represent numbers of the device.

Truncation error

Truncation error refers to an error in a method, which occurs because some series (finite or infinite) is truncated to a fewer number of terms. Such errors are essentially algorithmic errors and we can predict the extent of the error that will occur in the method.

Truncation Errors

Truncation errors are the errors that result from using an approximation in place of an exact mathematical procedure.

$$e^x = \underbrace{\overbrace{\boxed{1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \ldots + \frac{x^n}{n!}}}^{\text{Approximation}} + \overbrace{\boxed{\frac{x^{n+1}}{(n+1)!} + \ldots}}^{\text{Truncation Errors}}}_{\text{Exact mathematical formulation}}$$

Relative error versus absolute error

Absolute Error is the magnitude of the difference between the true value x and the approximate value xa, Therefore absolute error=[x-xa] The error between two values is defined as

$$\epsilon_{abs} = \|x - xa\| \quad ,$$

where x denotes the exact value and xadenotes the approximation.

The relative error of $\tilde{x}$ is the absolute error relative to the exact value. Look at it this way: if your measurement has an error of ± 1 inch, this seems to be a huge error when you try to measure something which is 3 in. long. However, when measuring distances on the order of miles, this error is mostly negligible. The definition of the relative error is

$$\epsilon_{rel} = \frac{\|\tilde{x} - x\|}{\|x\|} \quad .$$

MathWorks Resources

- Academic resources
  http://www.mathworks.com/academia/
- Classroom resources -> Numerical and Symbolic Math

  https://www.mathworks.com/academia/courseware.html?s_tid=acb_cw

- Cleve Moler's textbooks
  http://www.mathworks.com/moler/exm/chapters.html
  http://www.mathworks.com/moler/index_ncm.html
- MATLAB Central
  http://www.mathworks.com/matlabcentral/
- MATLAB Plot Gallery
  http://www.mathworks.com/discovery/gallery.html
- MATLAB symbolic computing: MuPAD
  http://www.mathworks.com/discovery/mupad.html