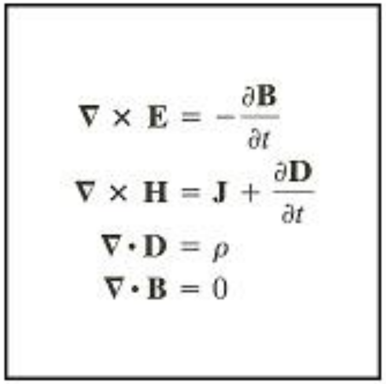
Numerical computing

How computers store real numbers

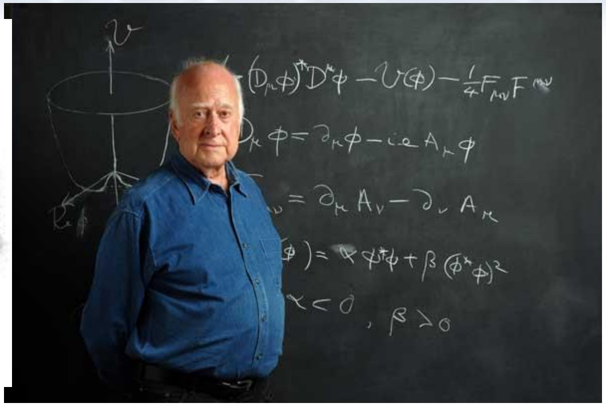
and the problems that result

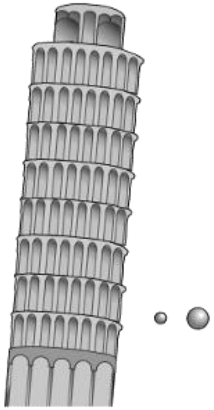
The scientific method

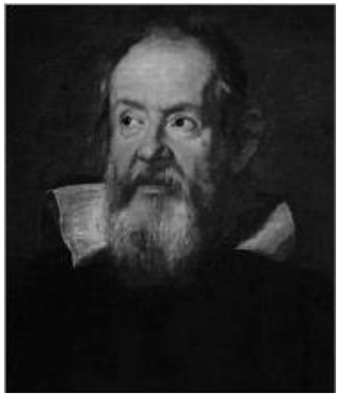
Theory: Mathematical equations provide a description or model



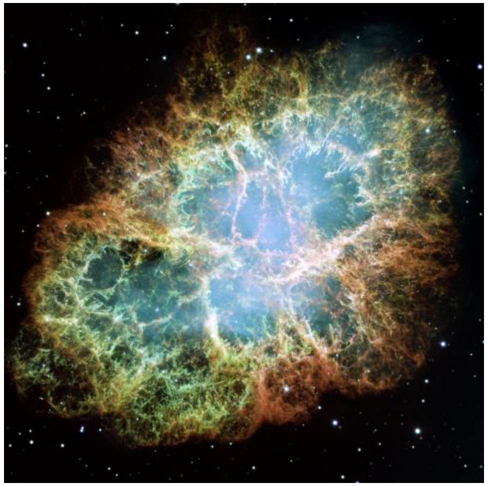
$$\nabla \times \mathbf{E} = -\frac{\partial \mathbf{B}}{\partial t}$$
$$\nabla \times \mathbf{H} = \mathbf{J} + \frac{\partial \mathbf{D}}{\partial t}$$
$$\nabla \cdot \mathbf{D} = \rho$$
$$\nabla \cdot \mathbf{B} = 0$$





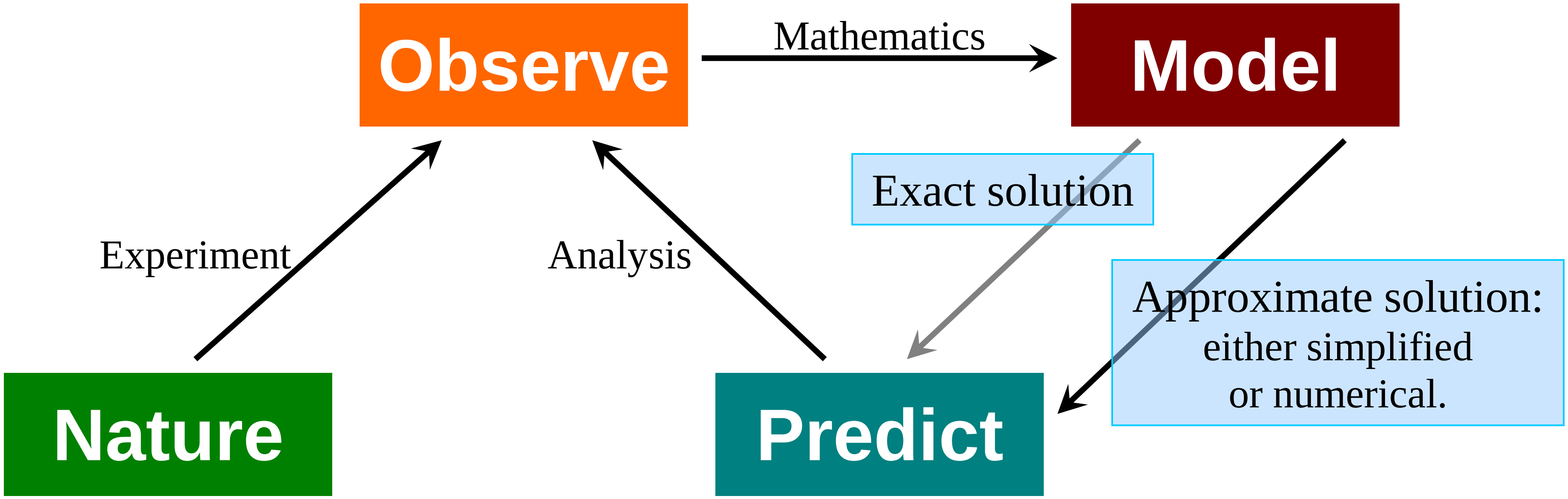


- Experiment
  - Inference from data
  - Test hypothesis



- Computer simulation
  - Too big, small, difficult or dangerous for standard methods
  - Explore the space of solutions

Computer Simulation in Science



Observe

Mathematics

Model

Exact solution

Experiment

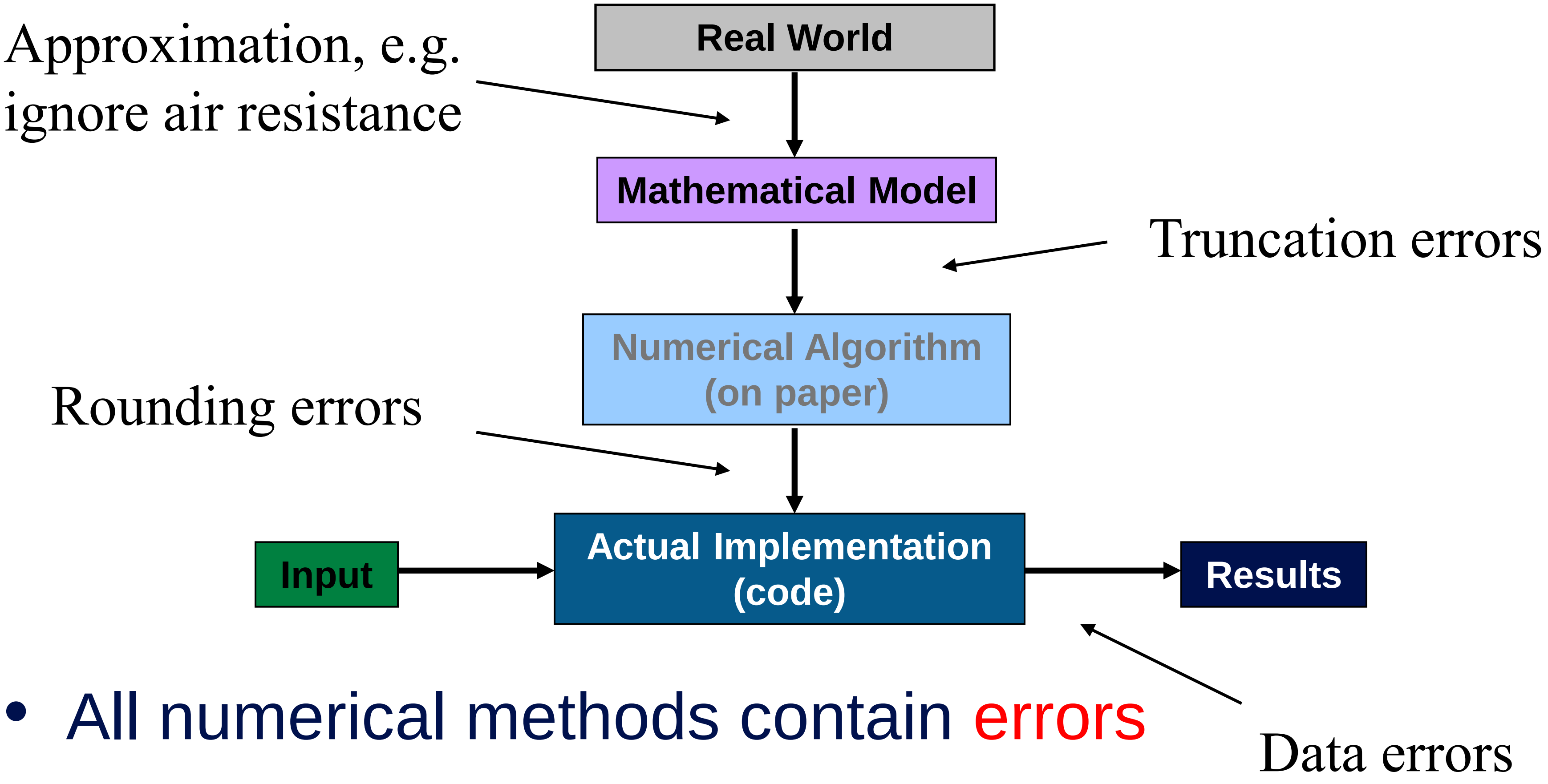
Analysis

Approximate solution:
either simplified
or numerical.

Nature

Predict

Numerical solution



Approximation, e.g.
ignore air resistance

Real World

Mathematical Model

Truncation errors

Numerical Algorithm (on paper)

Rounding errors

Input

Actual Implementation (code)

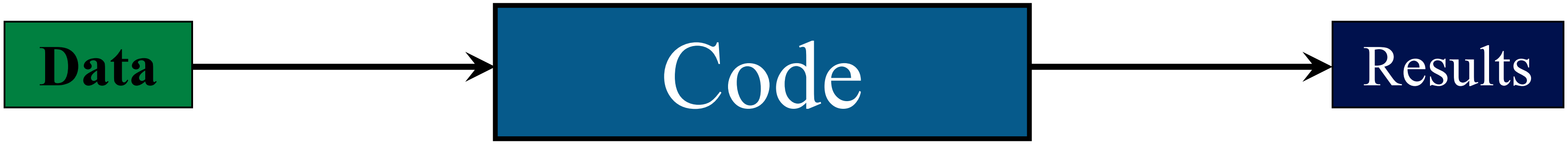
Results

Data errors

- All numerical methods contain errors
- It is *essential* to understand these errors
  - When is the solution obtained good enough

Data Errors

- Poor data.
  - It is very difficult to write code that will compensate for errors in your input data.
- Badly stored data:
  - The simplest failure can occur on the loading and saving of data.



Inputted and store as
integer or floating-point.

Output from internal
representation to
decimal (usually).

  - be careful to save as accurately as possible
  - writing as text may not be ideal

Truncation Errors

- Truncation errors
  - due to the differences between the mathematical model and the numerical algorithm.
    - independent of implementation or precision.

- Examples: Taylor Series expansion
  - the sin function may be calculated as:

$$\sin(x) = x - x^3/3! + x^5/5! - x^7/7! + \ldots\ldots$$

  - in practice we *truncate* the series at some finite order

- In real simulations, often comes from *discretising the* problem
  - eg a weather simulation takes place on a 10km grid
  - can reduce truncation errors by, eg, using a smaller grid
    - but this requires a lot more computer time!

Rounding Errors

- Due to the fact that real values are stored approximately
  - subject of the rest of this talk

Overview

- Integers
- Reals, floats, doubles, etc.
- Arithmetical operations and rounding errors

- We write:

```
x = sqrt(2.0)
```

  - but how is this stored?

Mathematics vs Computers

- Mathematics is an ideal world
  - integers can be as large as you want
  - real numbers can be as large or as small as you want
  - can represent every number exactly:

$$1, -3, 1/3, 10^{36237}, 10^{-232322}, \sqrt{2}, \pi, \ldots.$$

- Numbers range from - ∞ to +∞
  - there are also an infinite number in any interval (infinite precision)
- This not true on a computer
  - numbers have a limited range (integers and real numbers)
  - limited precision (real numbers)

Integers

- We like to use **base 10**
  - we only write the 10 characters 0,1,2,3,4,5,6,7,8,9
  - use *position* to represent each power of 10

$$\begin{aligned}\mathbf{125} &= \mathbf{1} * 10^2 + \mathbf{2} * 10^1 + \mathbf{5} * 10^0 \\ &= \mathbf{1}*100 + \mathbf{2}*10 + \mathbf{5}*1 = \mathbf{125}\end{aligned}$$

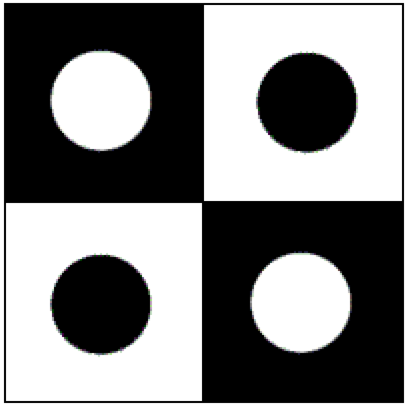
  - represent positive or negative using a leading “+” or “-”
- Computers are binary machines
  - can only store ones and zeros
  - minimum storage unit is 8 bits = 1 byte
- Use **base 2**

$$\begin{aligned}\mathbf{1111101} &= \mathbf{1}*2^6 + \mathbf{1}*2^5 + \mathbf{1}*2^4 + \mathbf{1}*2^3 + \mathbf{1}*2^2 + \mathbf{0}*2^1 + \mathbf{1}*2^0 \\ &= \mathbf{1}*64 + \mathbf{1}*32 + \mathbf{1}*16 + \mathbf{1}*8 + \mathbf{1}*4 + \mathbf{0}*2 + \mathbf{1}*1 \\ &= \mathbf{125}\end{aligned}$$

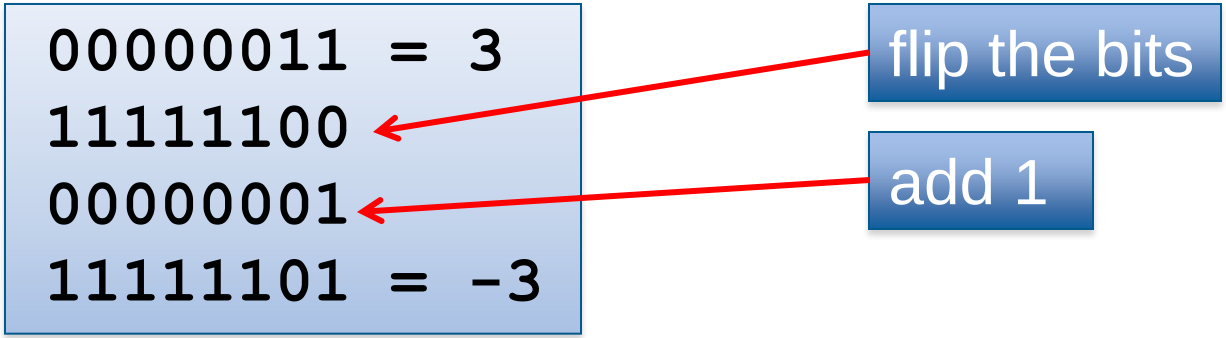
Storage and Range

- Assume we reserve 1 byte (8 bits) for integers
  - minimum value `0`
  - maximum value $2^8 - 1 = 255$
  - if result is out of range we will overflow and get wrong answer!
- Standard storage is 4 bytes = 32 bits
  - minimum value `0`
  - maximum value $2^{32} - 1 = 4294967291 = 4$ billion $= 4G$
- Is this a problem?
  - question: what is a 32-bit operating system?
- Can use 8 bytes (64 bit integers)

Aside: Negative Integers



- Use “two’s complement” representation
  - flip all ones to zeros and zeros to ones
  - then add one (ignoring overflow)
- Negative integers have the first bit set to “1”
  - for 8 bits, range is now: -128 to + 127
  - normal addition (ignoring overflow) gives the correct answer



```
00000011 = 3
11111100
00000001
11111101 = -3
```

flip the bits

add 1

```
125 + (-3) = 01111101 + 111111101 = 01111010  = 122
```

Integer Arithmetic

- Computers are brilliant at integer maths

- These can be added, subtracted and multiplied with complete accuracy…
  - …as long as the final result is not too large in magnitude

- But what about division?
  - 4/2 = 2, 27/3 = 9, but 7/3 = 2 (instead of 2.3333333333333…).
  - what do we do with numbers like that?
  - how do we store real numbers?

Fixed-point Arithmetic

- Can use an integer to represent a real number.
  - we have 8 bits stored in X 0-255.
  - represent real number *a* between 0.0 and 1.0 by dividing by 256
  - e.g. *a* = 5/9 = 0.55555 represented as X=142
    - 142/256 = 0.5546875
  - X = integer(*a* × 256), Y=integer(*b* × 256), Z=integer(*c* x 256) ....

- Operations now treat integers as fractions:
    - E.g. c = a × b becomes 256c = (256a × 256b)/256,

      i.e.Z = X×Y/256
  - Between the upper and lower limits (0.0 & 1.0), we have a uniform grid of possible ‘real’ numbers.

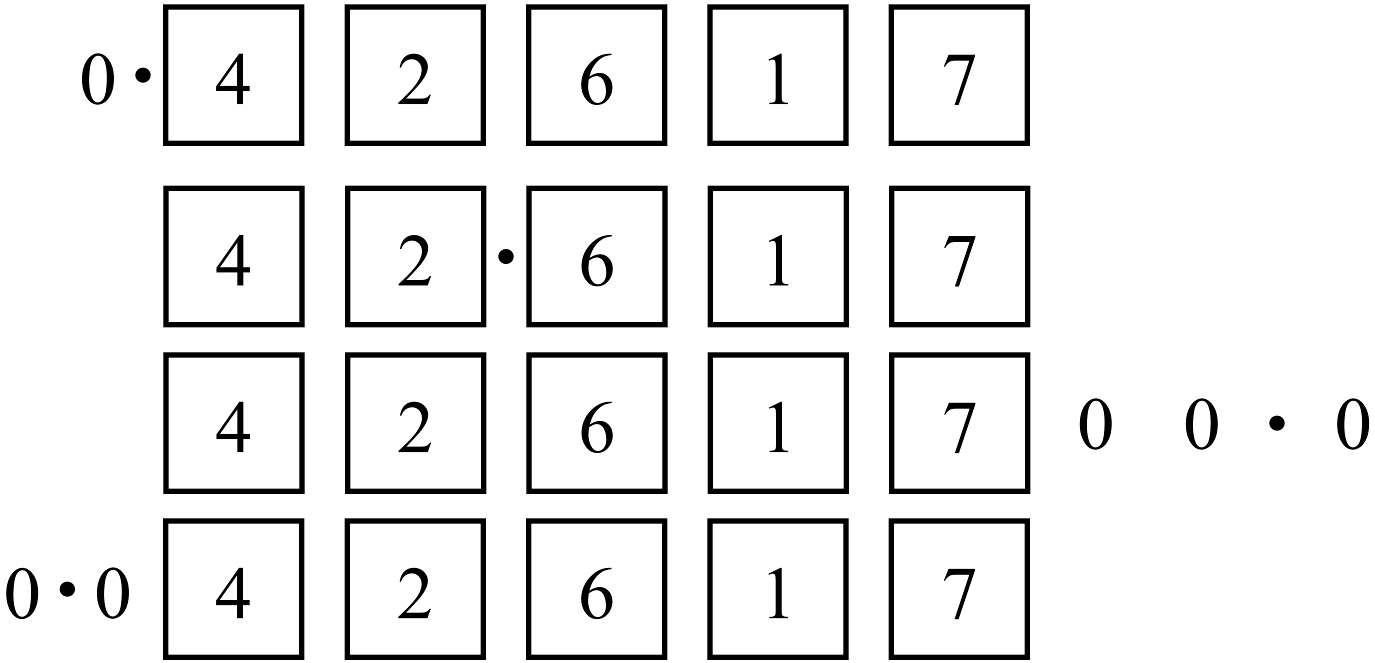
Problems with Fixed Point

- This arithmetic is very fast
  - but does not cope with large ranges
  - eg above, cannot represent numbers < 0 or numbers >= 1

- Can adjust the range
  - but at the cost of precision

Fixed-point analogy

- Decimal numbers
- Imagine we only have space for 5 numbers
  - put decimal point in a fixed location



0 • 4 2 6 1 7

4 2 • 6 1 7

4 2 6 1 7 0 0 • 0

0 • 0 4 2 6 1 7

Scientific Notation (in Decimal)

- How do we store 4261700.0 and 0.042617
  - in the same storage scheme?
- Decimal point was previously *fixed*
  - now let it *float* as appropriate
- Shift the decimal place so that it is at the start
  - ie 0.42617 (this is the mantissa *m*)
- Remember how many places we have to shift
  - ie +7 or -1 (the exponent *e*)
- Actual number is 0.mmmm x 10^e
  - ie 0.4262 * 10^{+7} or 0.4262 * 10^{-1}
    - always use all 5 numbers - don't waste space storing leading zero!
    - automatically adjusts to the magnitude of the number being stored
    - could have chosen to use 2 spaces for *e* to cope with very large numbers

Floating-Point Numbers

$$0\cdot\boxed{m}\,\boxed{m}\,\boxed{m}\,\boxed{m}\ \text{x}\ 10^{\boxed{e}}$$

- Decimal point “floats” left and right as required
  - fixed-point numbers have constant absolute error, eg +/- 0.00001
  - floating-point have a constant relative error, eg +/- 0.001%
- Computer storage of real numbers directly analogous to scientific notation
  - except using binary representation not decimal
  - ... with a few subtleties regarding sign of *m* and *e*
- All modern processors are designed to deal with floating-point numbers *directly in hardware*

The IEEE 754 Standard

- Mantissa made positive or negative:
  - the first bit indicates the sign: 0 = positive and 1 = negative.
- General binary format is:

Highest Bit	**1**	**1001010**	**10101000101000000101**	**Lowest Bit**
	Sign	Exponent	Mantissa	

- Exponent made positive or negative using a “biased” or “shifted” representation:
  - If the stored exponent, c, is X bits long, then the actual exponent is c – *bias* where the offset *bias* = $(2^X/2 - 1)$. e.g. X=3:

Stored (c,binary)	000	001	010	011	100	101	110	111
Stored (c,decimal)	0	1	2	3	4	5	6	7
Represents (c-3)	**-3**	**-2**	**-1**	**0**	**1**	**2**	**3**	**4**

IEEE – The Hidden Bit

- In base 10 exponent-mantissa notation:
  - we chose to standardise the mantissa so that it always lies in the binary range $0.0 \leq m < 1.0$
  - the first digit is always 0, so there is no need to write it.
- The FP mantissa is “normalised” to lie in the **binary** range:

  $1.0 \leq m < 10.0$ ie decimal range [1.0,2.0)

  - as the first bit is always one, there is no need to store it, We only store the variable part, called the significand (f).
  - the mantissa m = 1.f (in binary), and the 1 is called “The Hidden Bit”:
  - however, this means that zero requires special treatment.
    - having f and e as all zeros is defined to be (+/-) zero.

Binary Fractions: what does 1.f mean?

- Whole numbers are straightforward
  - base 10: 109 = $1*10^2 + 0*10^1 + 9*10^0 = 1*100 + 0*10 + 9*1 = 109$
  - base 2: 1101101 = $1*2^6+1*2^5+0*2^4+1*2^3+1*2^2+0*2^1+1*2^0$

    $= 1*64 + 1*32 + 0*16 + 1*8 + 1*4 + 0*2 + 1*1$

    $= 64 + 32 + 8 + 4 + 1 = 109$

- Simple extension to fractions

  $109.625 = 1*10^2 + 0*10^1 + 9*10^0 + 6*10^{-1} + 2*10^{-2} + 5*10^{-3}$

  $= 1*100 + 0*10 + 9*1 + 6*0.1 + 2*0.01 + 5*0.001$

  $1101101.101 = 109 + 1*2^{-1} + 0*2^{-2} + 1*2^{-3}$

  $= 109 + 1*(1/2) + 0*(1/4) + 1*(1/8)$

  $= 109 + 0.5 + 0.125$

  $= 109.625$

Relation to Fixed Point

- Like fixed point with divisor of 2^n
  - base 10: 109.625 = 109 + 625 / 10^3 = 109 + (625 / 1000)
  - base 2: 1101101.101 = 1101101 + (101 / 1000)
    = 109 + 5/8 = 109.625

- Or can think of shifting the decimal point

109.625 = 109625/10^3 = 109625 / 1000 (decimal)

1101101.101 = 1101101101 / 1000 (binary)

= 877/8 = 109.625

IEEE – Bitwise Storage Size

- The number of bits for the mantissa and exponent.
  - The normal floating-point types are defined as:

Type	Sign, a	Exponent, c	Mantissa, f	Representation
Single 32bit	1bit	8bits	23+1bits	$(-1)^s \times 1.f \times 2^{c-127}$ **Decimal: ~8s.f. $\times$ 10$^{\sim\pm 38}$**
Double 64bit	1bit	11bits	52+1bits	$(-1)^s \times 1.f \times 2^{c-1023}$ **Decimal: ~16s.f. $\times$ 10$^{\sim\pm 308}$**

  - there are also “Extended” versions of both the single and double types, allowing even more bits to be used.
  - the Extended types are not supported uniformly over a wide range of platforms; Single and Double are.

32-bit and 64-bit floating point

- Conventionally called single and double precision
  - C, C++ and Java: **`float`** (32-bit), **`double`** (64-bit)
  - Fortran: **`REAL`** (32-bit), **`DOUBLE PRECISION`** (64-bit)
    - or **`REAL(KIND(1.0e0))`**, **`REAL(KIND(1.0d0))`**
    - or **`REAL (Kind=4)`**, **`REAL (Kind=8)`**
  - **NOTHING TO DO with 32-bit / 64-bit operating systems!!!**
- Single precision accurate to 8 significant figures
  - eg 3.2037743 E+03
- Double precision to 16
  - eg 3.203774283170437 E+03
- Fortran usually knows this when printing default format
  - C and Java often don't
  - depends on compiler

Limitations

- Numbers cannot be stored exactly
  - gives problems when they have very different magnitudes
- Eg 1.0E-6 and 1.0E+6
  - no problem storing each number separately, but when adding:

0.000001 + 1000000.0 = 1000000.000001 = 1.000000000001E6

  - in 32-bit will be rounded to 1.0E6
- So

(0.000001 + 1000000.0) - 1000000.0 = 0.0

0.000001 + (1000000.0 - 1000000.0) = 0.000001

  - FP arithmetic is commutative but not associative!

Example – order matters!

```
#include <iostream>

template <typename T>
void order(const char* name) {
  T a, b, c, x, y;

  a = -1.0e10;
  b = 1.0e10;
  c = 1.0;
  x = (a + b) + c;
  y = a + (b + c);

  std::cout << name << ": x = " << x << ", y = " << y << std::endl;
}
int main()
{
  order<float>(" float");
  order<double>("double");
  return 0;
}
```

This code adds three numbers together in a different order.
Single and double precision.

$$x = \left(-1.0 \times 10^{10} + 1.0 \times 10^{10}\right) + 1.0$$

$$y = -1.0 \times 10^{10} + \left(1.0 \times 10^{10} + 1.0\right)$$

What is the answer?

The result. One

```
$ clang++ -O0 order.cpp -o order
$ ./order
 float: x = 1, y = 0
double: x = 1, y = 1
```

Special Values

- We have seen that zero is treated specially
    - corresponds to all bits being zero (except the sign bit)
- There are other special numbers
    - infinity: which is usually printed as “Inf”
    - Not a Number: which is usually printed as “NaN”
- These also have special bit patterns

Infinity and Not a Number

- Infinity is usually generated by dividing any finite number by 0.
  - although can also be due to numbers being too large to store
  - some operations using infinity are well defined, e.g. $-3/\infty = -0$
- NaN is generated under a number of conditions:

$$\infty + (-\infty), \quad 0 \times \infty, \quad 0/0, \quad \infty/\infty, \quad \sqrt{(X)} \text{ where } X < 0.0$$

  - most common is the last one, eg x = sqrt(-1.0)
- Any computation involving NaN's returns NaN.
  - there is actually a whole set of NaN binary patterns, which can be used to indicate why the NaN occurred.

IEEE Special Values

Exponent, e (unshifted)	Mantissa, f	Represents
000000…	0	±0
000000…	≠0	$0.f \times 2^{(1-bias)}$ [denormal]
000… < e < 111…	Any	$1.f \times 2^{(e-bias)}$
111111…	0	±∞
111111…	≠0	NaN

- Most numbers are in standard form (middle row)
  - have already covered zero, infinity and NaN
  - denormal numbers are a special case – not covered here

Implementations: C & FORTRAN

- Most C and FORTRAN compilers are fully IEEE 754 compliant.
  - compiler switches are used to switch on exception handlers.
  - these may be very expensive if dealt with in software.
  - you may wish to switch them on for testing (except inexact), and switch them off for production runs.

- But there are more subtle differences.
  - FORTRAN always preserves the order of calculations:
    - A + B + C = (A + B) +C, always.
  - C compilers are free to modify the order during optimisation.
    - A + B + C may become (A + B) + C or A + (B + C).
    - Usually, switching off optimisations retains the order of operations.

Summary

- Real numbers stored in floating-point format
- Conform to IEEE 754 standard
  - defines storage format
  - can be single (32-bit) and double (64-bit) precison
  - and the result of all arithmetical operations
- Lots of issues to be aware of…