Latin American Introductory School on Parallel Programming and Parallel Architecture for High-Performance Computing

Floating-Point Math and Accuracy

Dr. Richard Berger

High-Performance Computing Group
College of Science and Technology
Temple University
Philadelphia, USA

richard.berger@temple.edu

The Abdus Salam International Centre for Theoretical Physics

Temple University

Outline

Outline

Errors in Scientific Computing

Before a computation

- **modeling errors** due to neglecting properties or making assumptions
- **data errors**, due to imperfect empirical data
- **results from previous computations** from other (error-prone) numerical methods can introduce errors
- **programming errors**, sloppy programming and invalid data conversions
- **compilation errors**, buggy compiler, too aggressive optimizations

During a computation

- approximating a continuous solution with a discrete solution introduces a **discretization error**
- computers only offer a finite precision to represent real numbers. Any computation using these approximate numbers leads to **truncation and rounding errors**.

Example: Earth's surface area

Computing Earth's surface using

$$A = 4\pi r^2$$

introduces the following errors:

- Modelling Error: Earth is not a perfect sphere
- Empirical Data: Earth's radius is an empirical number
- Truncation: the value of π is truncated
- Rounding: all resulting numbers are rounded due to floating-point arithmetic

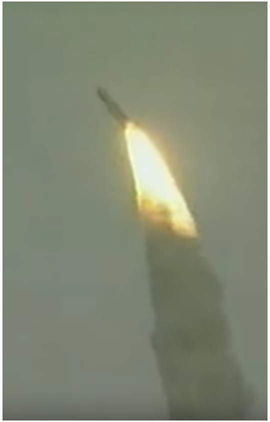
Importance of Floating-Point Math

- Understanding floating-point math and its limitations is essential for many HPC applications in physics, chemistry, applied math or engineering.
- real numbers have unlimited accuracy
- floating-point numbers in a computer are an approximation with a **limited precision**
- using them is always a **trade-off between speed and accuracy**
- not knowing about floating-point effects can have devastating results. . .

The Patriot Missile Failure

- February 25, 1991 in Dharan, Saudi Arabia (Gulf War)
- American Patriot Missile battery failure led to **28 deaths**, which is ultimately attributable to **poor handling of rounding errors of floating-point numbers.**
- System's time since boot was calculated by multipling an internal clock with 1/10 to get the number of seconds
- After over 100 hours of operation, the accumulated error had become large enough that an incoming SCUD missle could travel more than half a kilometer without being treated as threat.

The Explosion of the Adriane 5



- 4 June 1996, maiden flight of the Ariane 5 launcher
- 40 seconds after launch, at an altitude of about 3700m, the launcher veered off its flight path, broke up and exploded.
- An error in the software occured due to a data conversion of a 64-bit floating point to a 16-bit signed integer value. The value of the floating-point was larger than what could be represented in a 16-bit signed integer.
- After a decade of development costing **$7 billion**, the destroyed rocket and its cargo were valued at **$500 million**
- Video: `https://www.youtube.com/watch?v=gp_D8r-2hwk`

Outline

Fundamental Data Types

Processors have two different modes of doing calculations:

- integer arithmetic
- floating-point arithmetic

The operands of these calculations have to be stored in binary form. Because of this there are two groups of fundamental data types for numbers in a computer:

- integer data types
- floating-point data types

Recap: storing information in binary

with 1 bit, you can store 2 values

```
0, 1
```

Recap: storing information in binary

with 1 bit, you can store 2 values

```
0, 1
```

with 2 bit, you can store 4 values

```
00, 01, 10, 11
```

Recap: storing information in binary

with 1 bit, you can store 2 values

```
0, 1
```

with 2 bit, you can store 4 values

```
00, 01, 10, 11
```

with 3 bit, you can store 8 values

```
000, 001, 010, 011, 100, 101, 110, 111
```

Recap: storing information in binary

with 1 bit, you can store 2 values

```
0, 1
```

with 2 bit, you can store 4 values

```
00, 01, 10, 11
```

with 3 bit, you can store 8 values

```
000, 001, 010, 011, 100, 101, 110, 111
```

with 4 bit, you can store 16 values

```
0000, 0001, 0010, 0011, 0100, 0101, 0110, 0111,
1000, 1001, 1010, 1011, 1100, 1101, 1110, 1111
```

Storing information in binary

$$\text{number of values represented by } b \text{ bits} = 2^b$$

- 1 byte is 8 bits $\Rightarrow 2^8 = 256$
- 2 bytes are 16 bits $\Rightarrow 2^{16} = 65,536$
- 4 bytes are 32 bits $\Rightarrow 2^{32} = 4,294,967,296$
- 8 bytes are 64 bits $\Rightarrow 2^{64} = 18,446,744,073,709,551,616$

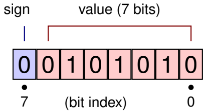
Integer Data Types in C/C++[1]

`char`	8 bit (1 byte)
`short int`	16 bit (2 byte)
`int`	32 bit (4 bytes)
`long int`	64 bit (8 bytes)

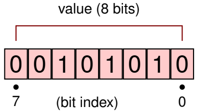
- **`short int`** can be abbreviated as **`short`**
- **`long int`** can be abbreviated as **`long`**
- integers are by default **`signed`**, and can be made **`unsigned`**

[1] sizes only valid on Linux x86_64

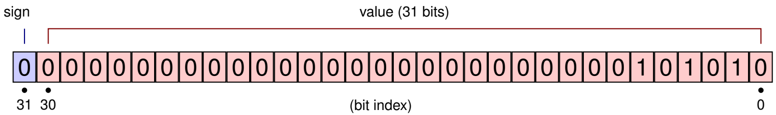
`char`



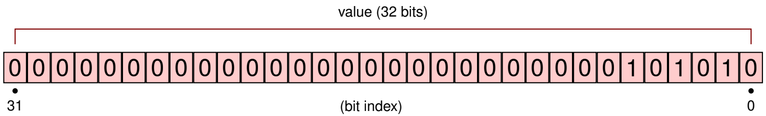
`unsigned char`



int



unsigned int



Integer Types - Value Ranges

unsigned integer (with b bits)

$$\left[0, 2^b - 1\right]$$

signed integer (with b bits, which means 1 sign bit and $b - 1$ value bits)

$$\left[-2^{b-1}, 2^{b-1} - 1\right]$$

Integer Types - Value Ranges

`char`	$[-128, 127]$
`short`	$[-32768, 32767]$
`int`	$[-2147483648, 2147483647]$
`long`	$[-9223372036854775808, 9223372036854775807]$
`signed char`	$[-128, 127]$
`signed short`	$[-32768, 32767]$
`signed int`	$[-2147483648, 2147483647]$
`signed long`	$[-9223372036854775808, 9223372036854775807]$
`unsigned char`	$[0, 255]$
`unsigned short`	$[0, 65535]$
`unsigned int`	$[0, 4294967295]$
`unsigned long`	$[0, 18446744073709551615]$

Scientific Notation

- Floating-point representation is similar to the concept of scientific notation
- Numbers in **scientific notation** are scaled by a power of 10, so that it lies with a range between 1 and 10.

$$123456789 = 1.23456789 \cdot 10^8$$

or more generally:

$$s \cdot 10^e$$

where s is the significand (or sometimes called mantissa), and e is the exponent.

Floating-point numbers

$$s \cdot \mathbf{2}^{e}$$

- a **floating-point** number consists of the following parts:
  - a signed **significand** (sometimes also called mantissa) in **base 2** of fixed length, which determines its precision
  - a signed integer **exponent** of fixed length which modifies its magnitude
- the *value* of a floating-point number is its *significand* multiplied by its *base* raised to the power of the *exponent*

Floating-point numbers

Example:

$$42_{10} = 101010_2 = 1.01010_2 \cdot 2^5$$

Floating-point formats

- in general, the radix point is assumed to be somewhere within the significand
- the name *floating-point* originates from the fact that the value is equivalent to shifting the radix point from its implied position by a number of places equal to the value of the exponent
- the amount of binary digits used for the significand and the exponent are defined by their **binary format**.
- over the years many different formats have been used, however, since the 1990s the most commonly used representations are the ones defined in the **IEEE 754 Standard**. The current version of this standard is IEEE 754-2008.

IEEE 754 Floating-point numbers

$$\pm 1.f \cdot \mathbf{2}^{\pm e}$$

The IEEE 754-1985 standard defines the following floating-point basic formats:

Single precision (binary32): 8-bit exponent, 23-bit fraction, 24-bit precision

Double precision (binary64) 11-bit exponent, 52-bit fraction, 53-bit precision

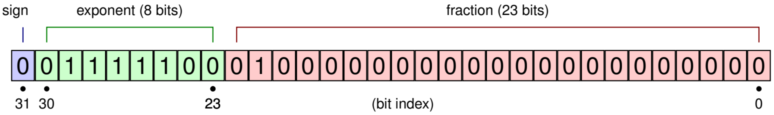
- Each format consists of a sign bit, exponent and fraction part
- one additional bit of precision in the significand is gained through normalization. Numbers are normalized to be in the form $1.f$, where f is the fractional portion of the singificand. Because of this, the leading 1 must not be stored.
- the exponent has an offset and is also used to encode special numbers like ± 0, $\pm\infty$ or NaN (not a number).

IEEE 754 Floating-point numbers

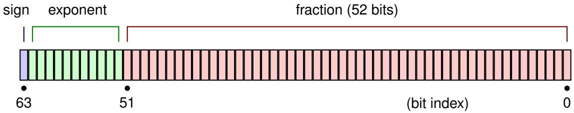
- exponents are stored with an offset
- single-precision: $e_{\text{stored}} = e + 127$
- double-precision: $e_{\text{stored}} = e + 1023$

Name	Value	Sign	(Stored) Exponent	Significand
positive zero	+0	0	0	0
negative zero	-0	1	0	0
positive subnormals	+0.f	0	0	non-zero
negative subnormals	-0.f	1	0	non-zero
positive normals	+1.f	0	$1 \ldots e_{max} - 1$	non-zero
negative normals	-1.f	1	$1 \ldots e_{max} - 1$	non-zero
positive infinity	$+\infty$	0	e_{max}	0
negative infinity	$-\infty$	1	e_{max}	0
not a number	NaN	any	e_{max}	non-zero

float (binary32)



double (binary64)



What the IEEE 754 Standard defines

- Arithmetic operations (add, subtract, multiply, divide, square root, fused multiply-add, remainder)
- Conversions between formats
- Encodings of special values
- This ensures portability of compute kernels

Floating-Point Types

`float`	32 bit (4 bytes)	single precision	$\approx$ 7 decimal digits
`double`	64 bit (8 bytes)	double precision	$\approx$ 15 decimal digits

- `float` numbers are between 10^{-38} to 10^{38}
- `double` numbers are between 10^{-308} to 10^{308}

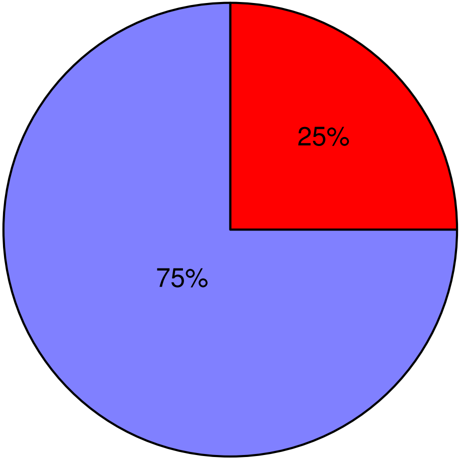
Outline

How many floating-point number are between 0 and 1?

float

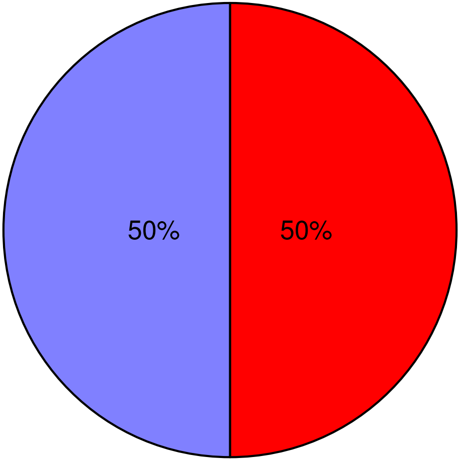
- 8 bits exponents
- 0 is represented with exponent -127
- 126 negative exponents, each has 2^{23} unique numbers
- 1,056,964,608
- +1 for one
- +1 for zero
- **1,056,964,610**
- total available numbers in 32bit: 2^{32} = 4,294,967,296

Floating-Point Numbers between 0.0 and 1.0



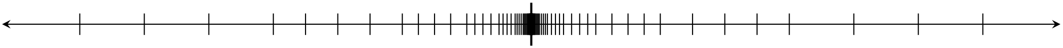
About **25%** of all numbers in a FP number are between 0.0 and 1.0

Floating-Point Numbers between -1.0 and 1.0



That also means, about **50%** of all numbers in a FP number are between -1.0 and 1.0

Density of Floating-Point numbers



- since the same number of bits is used for the fraction part of a FP number, the exponent determines the representable number density
- e.g. in a single-precision floating-point number there are 8,388,606 numbers between 1.0 and 2.0, but only 16,382 between 1023.0 and 1024.0
- $\Rightarrow$ accuracy depends on the magnitude
- $\Rightarrow$ all numbers beyond a threshold are even
  - **single-precision:** all numbers beyond 2^{24} are even
  - **double-precision:** all numbers beyond 2^{53} are even

Unit of Last Position (ulp)

The spacing between two neighboring floating-point numbers. i.e. the value the least significant digit of the significand represents if it is 1.

Example: exp(100)

$$\exp(100) = 2.68811714181613 56 \cdot 10^{43}$$

Example: exp(100)

$$\exp(100) = 2.6881171418161356 \cdot 10^{43}$$

number if we take this literally

`26,881,171,418,161,356,000,000,000,000,000,000,000,000,000`

Example: exp(100)

$$\exp(100) = 2.6881171418161356 \cdot 10^{43}$$

number if we take this literally

`26,881,171,418,161,356,000,000,000,000,000,000,000,000,000`

actual value stored

`26,881,171,418,161,356,`**`094,253,400,435,962,903,554,686,976`**

Example: exp(100)

$$\exp(100) = 2.6881171418161356 \cdot 10^{43}$$

number if we take this literally

`26,881,171,418,161,356,000,000,000,000,000,000,000,000,000`

actual value stored

`26,881,171,418,161,356,`**`094,253,400,435,962,903,554,686,976`**

correct value

`26,881,171,418,161,35`**`4,484,126,255,515,800,135,873,611,118`**

Example: exp(100) - What happened?

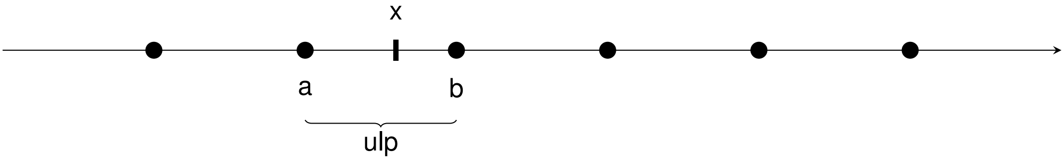
We want to store $x = e^{100}$

x=26,881,171,418,161,354,484,126,255,515,800,135,873,611,118

However, the closest (double-precision) floating-point numbers are:

a=26,881,171,418,161,**351,142,493,243,294,441,803,958,190,080**

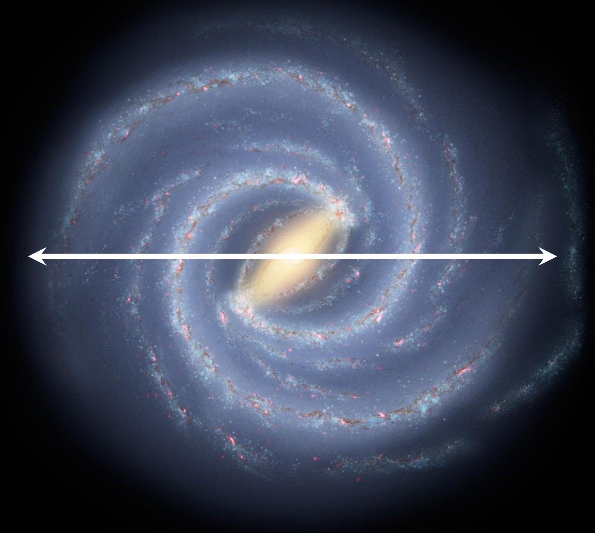
b=26,881,171,418,161,**356,094,253,400,435,962,903,554,686,976**



- x was rounded to the closest floating-point number, which is b
- this rounding introduces a relative error $|e| \leq \frac{1}{2}$ ulps

How bad could it be?

- 1.6x the diameter of our galaxy measured in micrometers
- the error is so high because in this range of floats 1 ulp is $2^{92} \approx 4.95 \cdot 10^{27}$



Milky Way Galaxy ($\varnothing \approx$ 100,000 light years)

Rounding

- Floating-point math can result in not representable numbers
- In this case the floating-point unit has to perform rounding, which follows the rules specified in IEEE 754
- in order to perform these rounding decisions, FPUs internally work by using a few bit more precision
- the resulting relative error $|e|$ should be $\leq \frac{1}{2}$ ulps

```
float f = 1.0f/3.0f;
```

```
double d = 1.0/100.0;
```

Warning:

- Be aware that while your code might compile, the numbers you write in your source code might not be representable!

```
float great_scientific_constant = 33554433.0f;

int main() {
    printf("The value is %f!", great_scientific_constant);
    // this will output "The value is 33554432.0!"
    return 0;
}
```

Floating-Point Arithmetic

- FP math is commutative, but **not associative**!

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

Example

```
d = 1.0 + (1.5e38 + (-1.5e38));
printf("%f", d); // prints 1.0

d = (1.0 + 1.5e38) + (-1.5e38);
printf("%f", d); // prints 0.0
```

Addition

$$
\begin{aligned}
a + b &= s_a \cdot 2^{e_a} \times s_b \cdot 2^{e_b} \\
&= (s_a \cdot 2^{e_a - e_b} + s_b) \cdot 2^{e_b} \qquad e_a \leq e_b
\end{aligned}
$$

- determine which operand has smaller exponent
- shift operand with smallest exponent until it matches the other exponent
- perform addition
- normalize number
- round and truncate

Notes on FP Subtraction

- Subtraction of two floating-point numbers of the same sign and similar magnitude (same exponent) will always be representable
- leading bits in fraction cancel
- $\Rightarrow$ results have less 'valid' digits
- $\Rightarrow$ (potential) loss of information
- catastrophic cancellation happens when operands are subject to rounding errors. subtraction may only leave parts of a FP number which are contaminated with previous errors.
- $\Rightarrow$ Careful when using result in multiplication, since result is *tainted* by low accuracy

Multiplication

$$\begin{aligned} a \times b &= s_a \cdot 2^{e_a} \times s_b \cdot 2^{e_b} \\ &= (s_a \times s_b) \cdot 2^{(e_a + e_b)} \end{aligned}$$

- after significands are multiplied, the result has to be normalized
- in general, the resulting number will have more bits than can be stored
- the resulting number is truncated and rounded to the closest representable value

Comparison

- since floating-point results are usually **inexact**, comparing for equality is **dangerous**
- E.g., don't use FP as loop index
- Exact comparison only really works if you know your results are bitwise identical. E.g. to avoid division by zero
- it's better to compare against expected error

Listing 1: Program to determine machine epsilon

```
float x = 1.0f;

while((1.0f + x) != 1.0f) {
        x = 0.5 * x;
}
```

Compiler Optimizations

- compilers will try to change your code to improve performance
- however, some of these transformations can affect floating-point math
- they could rearrange instructions, which changes the order of how numbers are computed (not commutative)
- be careful with `-ffast-math` and `-funsafe-math-optimizations`, as they may violate IEEE specs
- e.g., disabling subnormals

Floating-Point Exceptions

Exceptions

- invalid (0/0)
- division by zero
- underflow
- overflow
- inexact (most operations)

- can be detected at runtime
- by enabling FP exception traps, they can help you to intentionally crash your program in order to determine the source of a FP problem

Outline

Summation

- Avoid summing numbers of different magnitudes
- Contributions of numbers too small for a given magnitude are discarded due to truncation

Strategies

- sort numbers first and sum in ascending order
- sum in blocks (pairs) and then sum the sums
- Kahan summation, which uses a compensation variable to carry over errors
- Use (scaled) integers, if number range allows it
- Use higher precision numbers for sum (float $\rightarrow$ double, double $\rightarrow$ long double or binary128)

Note

summing numbers in parallel may give different results depending on parallelization

Data Conversions: integer → floating-point

Table: Exact conversion from integer to floating-point (x86_64) without truncation

	float (24bit significant)	double (53bit)	binary128 (113bit)
char	OK	OK	OK
short	OK	OK	OK
int	$\left[-(2^{24}), 2^{24}\right]$	OK	OK
long	$\left[-(2^{24}), 2^{24}\right]$	$\left[-(2^{53}), 2^{53}\right]$	OK
unsigned char	OK	OK	OK
unsigned short	OK	OK	OK
unsigned int	$\left[0, 2^{24}\right]$	OK	OK
unsigned long	$\left[0, 2^{24}\right]$	$\left[0, 2^{53}\right]$	OK

WARNING

These value ranges can be different on computer architectures other than x86_64!

Data Conversions: floating-point $\rightarrow$ integer

Table: Valid conversion ranges from floating-point to integer (x86_64) without under/overflow

	float (24bit significant)	double (53bit)	binary128 (113bit)
char	$[-2^7, 2^7 - 1]$	$[-2^7, 2^7 - 1]$	$[-2^7, 2^7 - 1]$
short	$[-2^{15}, 2^{15} - 1]$	$[-2^{15}, 2^{15} - 1]$	$[-2^{15}, 2^{15} - 1]$
int	$\left[-(2^{31}), 2^{31} - 1\right]$	$\left[-(2^{31}), 2^{31} - 1\right]$	$\left[-(2^{31}), 2^{31} - 1\right]$
long	$\left[-(2^{63}), 2^{63} - 1\right]$	$\left[-(2^{63}), 2^{63} - 1\right]$	$\left[-(2^{63}), 2^{63} - 1\right]$
char	$[0, 2^8 - 1]$	$[0, 2^8 - 1]$	$[0, 2^8 - 1]$
short	$[0, 2^{16} - 1]$	$[0, 2^{16} - 1]$	$[-2^{16}, 2^{15} - 1]$
int	$\left[0, 2^{32} - 1\right]$	$\left[0, 2^{32} - 1\right]$	$\left[-(2^{31}), 2^{31} - 1\right]$
long	$\left[0, 2^{64} - 1\right]$	$\left[0, 2^{64} - 1\right]$	$\left[-(2^{63}), 2^{63} - 1\right]$

WARNING

These value ranges can be different on computer architectures other than x86_64!

Use library functions over own implementations

They are usually much better tested than your own code and have taken great care of special numerical cases. Look at newer language standards for better compliance with IEEE 754-2008 (e.g., C++11).

`log1pf(x), log1p(x), log1pl(x)`

A single-, double- and extended-precision variant of `log(1+x)`, which computes this function in a way which remains accurate even if x goes towards 0.

`expm1(x)`

Compute *e* raised to the power of *x* minus 1.0. This function is more accurate than exp(x) - 1.0 when *x* is close to zero.

Use higher precision numbers when accuracy matters

single-precision

If you are comfortable with single precision most of the time, use double-precision for critical functions and scale the final result back to single-precision.

double-precision

For double-precision you can take advantage of the 80-bit extended floating-point format (`long double`). If that isn't enough, you can go up to 128bit floats which are still supported by hardware.

If that all fails

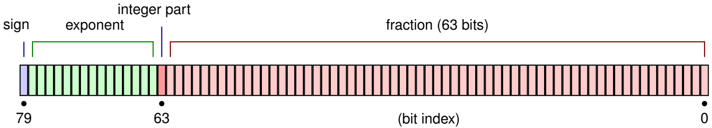
Use a higher-precision format such as the ones provided by MPFR.

Extended Precision, float128

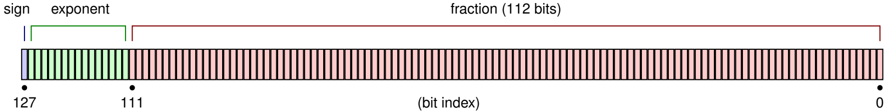
x86 extended precision format (80bit): 15-bit exponent, 63-bit fraction, 64-bit precision

quadruple precision (binary128) 15-bit exponent, 112-bit fraction, 113-bit precision

`long double` (x86 extended precision format)



`float128`



MPFR Library

- C-library for multiple-precision floating-point computations with correct rounding
- contains tuned and tested implementations of many mathematical functions and has well-defined semantics
- enables arbitrary precision with similar semantics as IEEE 754
- controllable accurary, at the expense of speed

Use special purpose formats

- Applications for financial and tax purposes need to emulate decimal rounding exactly. The IEEE 754-2008 standard also defines floating-point data types `decimal32`, `decimal64`, `decimal128` which exhibit this behaviour. There is experimental support for these in C++11.
- some research fields, such as machine learning, do not need a high degree of precision and only need values in the range of 0-1. For these applications **half-precision (binary16)** numbers, defined in IEEE 754-2008, have become popular. They are especially well supported on GPUs. However, extra care has to be taken to ensure the available range is good enough.

Outline

Computer Lab Instructions

1. Connect to ABACUS

```
ssh USERNAME@s0.edomex.cinvestav.mx
```

2. Copy `floating_point.tar.gz` and extract in your home directory

```
cp /lustre/scratch/user5/examples/floating_point.tar.gz $HOME
tar xvzf floating_point.tar.gz
```

3. Work through examples. Each of them showcases some of the properties we discussed. Follow the instruction in the README and source code.