RC Days 2023 Advanced topics in HPC:

Numerical accuracy, lies and statistics

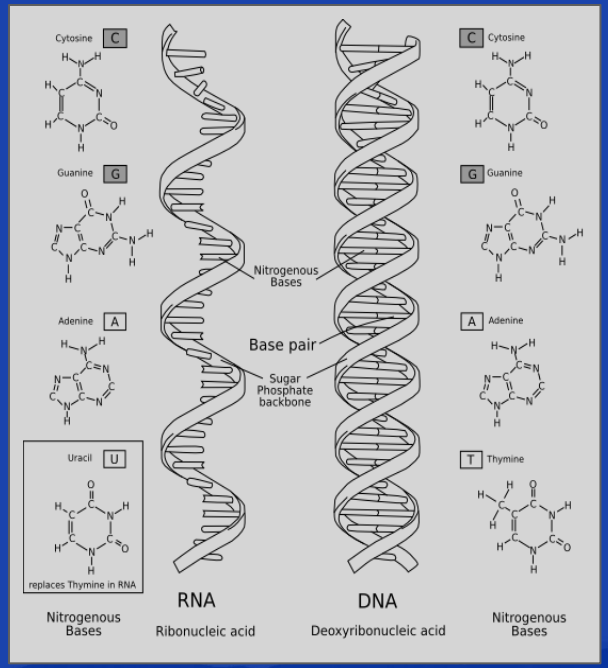
Dr. Frank T Willmore
Boise State University Research Computing
FrankWillmore@boisestate.edu

BOISE STATE UNIVERSITY

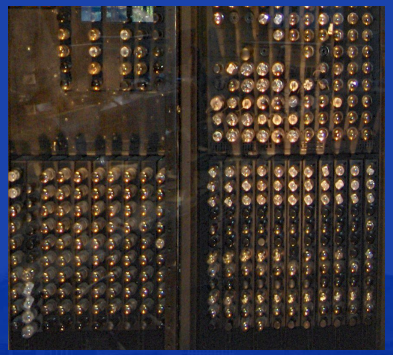
What is a number?

- Real
- Imaginary
- Complex
- Integer
- Countable
- Random
- Finite/Infinite
- Positive/Negative
- Rational/Irrational/Transcendental
- Binary/Decimal/Octal/Hexadecimal
- Exact/Approximate
- Floating point

What is a number? Storage and representation



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

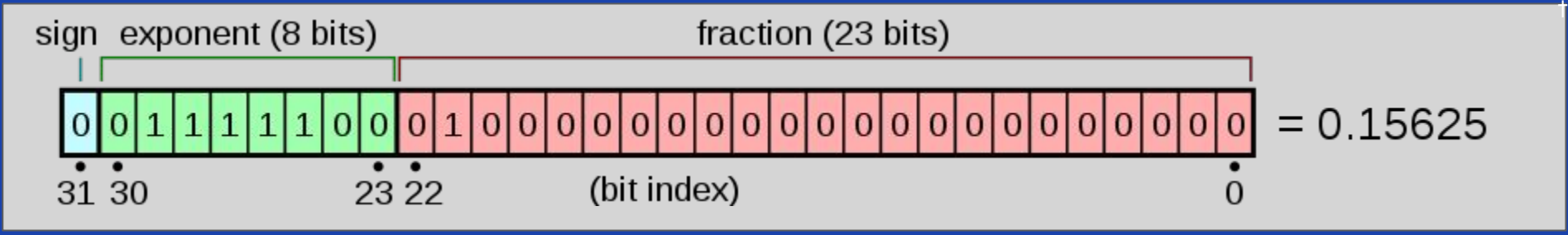


0x0	0	0	1	1	0	0	0	0
0x1	0	0	0	0	0	0	0	0
0x2	0	0	0	0	0	1	0	0
0x3	0	0	0	0	0	0	0	0
0x4	0	0	1	0	0	0	0	0
0x5	0	0	0	0	0	0	0	0
0x6	0	0	0	0	0	0	0	0
0x7	0	0	0	0	0	0	0	0
0x8	0	0	0	1	0	0	0	0
0x9	0	0	0	0	0	0	0	0
0xA	0	0	1	0	0	0	0	0
0xB	0	0	0	1	0	0	0	0
0xC	0	1	0	0	0	0	0	0
0xD	0	0	0	0	1	0	0	0
0xE	0	0	0	1	0	1	0	0
0xF	0	1	0	0	0	1	0	0

What is a number? Storage and representation

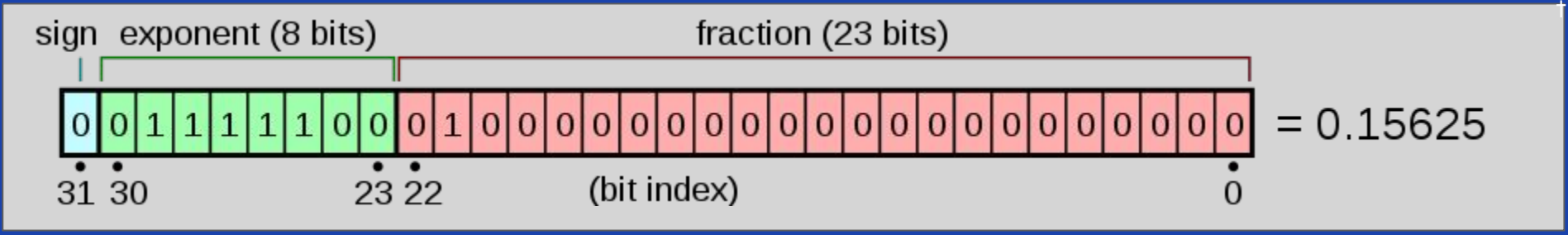
0x0	0	0	1	1	0	0	0	0
0x1	0	0	0	0	0	0	0	0
0x2	0	0	0	0	0	1	0	0
	0	0	0	0	0	0	0	0
	0	0	1	0	0	0	0	0
	0	0	0	0	0	0	0	0
	0	0	0	0	0	0	0	0
	0	0	0	0	0	0	0	0
0x8	0	0	0	1	0	0	0	0
0x9	0	0	0	0	0	0	0	0
0xA	0	0	1	0	0	0	0	0
0xB	0	0	0	1	0	0	0	0
0xC	0	1	0	0	0	0	0	0
0xD	0	0	0	0	1	0	0	0
0xE	0	0	0	1	0	1	0	0
0xF	0	1	0	0	0	1	0	0

What is a number? Machine representation of a floating point number



†https://en.wikipedia.org/wiki/Single-precision_floating-point_format

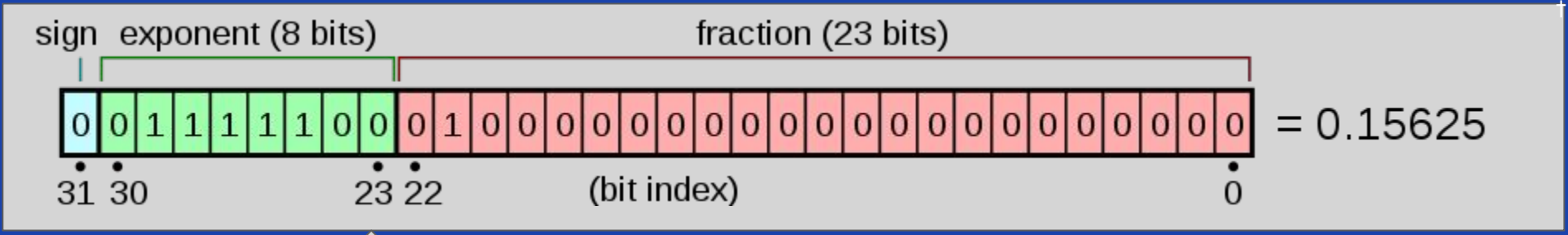
What is a number? Machine representation of a floating point number



-1^{sign}
= 1

†https://en.wikipedia.org/wiki/Single-precision_floating-point_format

What is a number? Machine representation of a floating point number

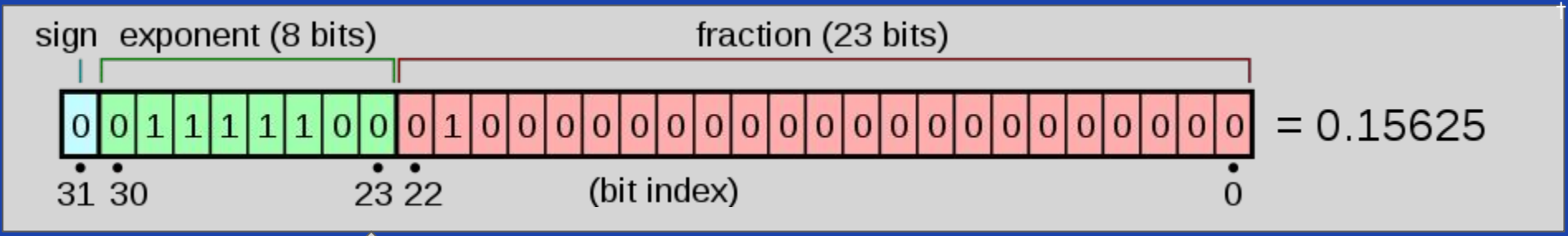


-1^{sign} = **1**

The exponent is 'offset binary' format with a shift of -127, i.e. the exponent is (0*128) + (1*64) + (1*32) + (1*16) + (1*8) + (1*4) + (0*2) + (0*1) **- 127**, or **-3**. This means a factor of $\mathbf{2^{-3}}$.

†https://en.wikipedia.org/wiki/Single-precision_floating-point_format

What is a number? Machine representation of a floating point number



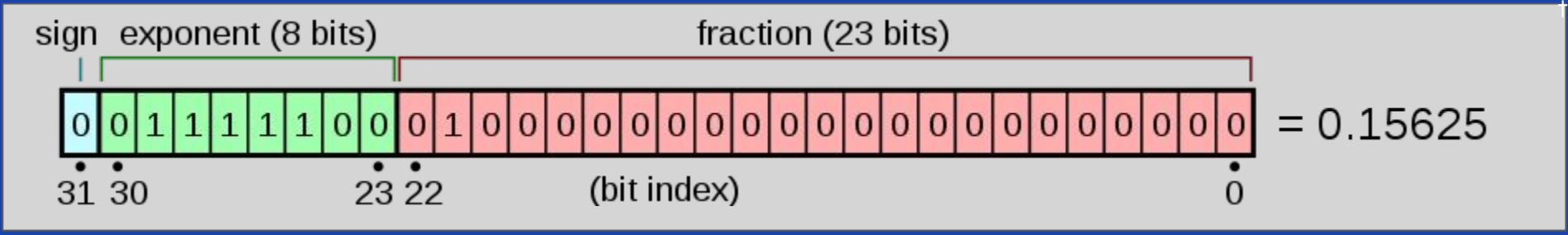
-1^{sign} = **1**

The exponent is 'offset binary' format with a shift of -127, i.e. the exponent is (0*128) + (1*64) + (1*32) + (1*16) + (1*8) + (1*4) + (0*2) + (0*1) **- 127**, or **-3**. This means a factor of $\mathbf{2^{-3}}$.

Just like base 10 scientific notation, the exponent is shifted so that the first non-zero digit in the fraction is to the left of the decimal point. In binary, this value (2^{-0}=1) is implicit and not recorded. Since only the 2^{-2} bit is set above, the fraction value here is $2^{-0} + 2^{-2} = 1 + ¼ =$ **1.25**.

†https://en.wikipedia.org/wiki/Single-precision_floating-point_format

What is a number? Machine representation of a floating point number



$$1 \times 2^{-3} \times 1.25 = 0.15625$$

†https://en.wikipedia.org/wiki/Single-precision_floating-point_format

So what does it all matter? How does this affect my research?

- Machine precision is inherently limited
- This means rounding errors
- Rounding errors are a source of experimental error (in addition to modeling shortfalls) for computational science and can affect results
- Good research accounts for measurement error

So what does it all matter? How does this affect my research?

- Machine precision is inherently limited
- This means rounding errors
- Rounding errors are a source of experimental error (in addition to modeling shortfalls) for computational science and can affect results
- Good research accounts for measurement error

What can I do?

- Recognize the limitations of machine precision
- Make responsible and efficient choices regarding precision
  - AI/ML applications often use very low precision
  - GROMACS uses single precision by default
- Document choices
- Do better research

EXAMPLE: SUMMATION AND ROUNDING

$$1 + x \rightarrow 1 + x;\ x > \delta$$
$$1 + x \rightarrow 1;\ x <= \delta$$

$$y + x \rightarrow \max(x,y);\ x/y > \delta$$
$$y + x \rightarrow x + y;\ x/y <= \delta$$

EXERCISE: summing floats

```c
#include <stdio.h>

int main()
{
  float ttf = 16777216;   // 2^24
  float delta = 1.0f/ttf;  // 0.000000059604645
  float sum = 0.0f;

  // check inputs
  printf("Starting with sum=%1.18f, delta=%1.18f:\n\n", sum, delta);

  // should add to one
  for (int i=0; i<ttf; i++) sum += delta;
  printf("sum=%1.18f\n", sum);

  // what happens if we keep adding?
  for (int i=0; i<ttf; i++) sum += delta;
  printf("sum=%1.18f\n", sum);
}
```

EXERCISE: summing floats

```
id@borah-login$ dev-session-bsu
…
id@cpu1xx$ module load borah-base openmpi/4.1.3/gcc/12.1.0
id@cpu1xx$ git clone https://github.com/bsurc/AdvancedHPC
id@cpu1xx$ cd AdvancedHPC/machine_numbers_and_MPI
id@cpu1xx$ vi add_float.c
id@cpu1xx$ make add_float
id@cpu1xx$ ./add_float
Starting with sum=0.000000000000000000, delta=0.000000059604644775:

sum=1.000000000000000000
sum=1.000000000000000000
```

EXERCISE: summing floats

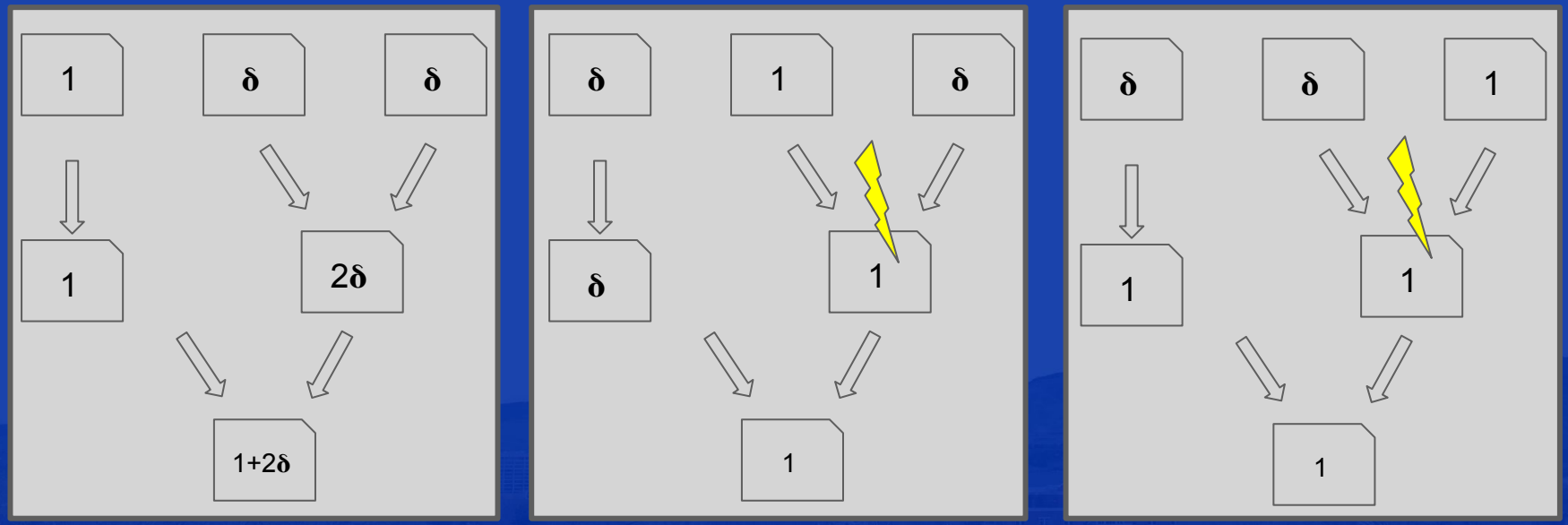
- What happens if you increase the size of ttf?
  - Edit add_float.c and recompile
  - What is the result?
- What happens if you decrease the size of ttf?
  - Edit add_float.c and attempt to recompile
  - What does the compiler say?
  - What does it mean?

Example of MPI Reduction

- MPI parallel processes run and complete in an indeterminate order
- Because of rounding, results of a reduction depend on the order in which operations are performed
- MPI operations can give results that vary between runs and depend on size of run

Example of MPI Reduction

- MPI parallel processes run and complete in an indeterminate order
- Because of rounding(), results of a reduction depend on the order in which operations are performed
- MPI operations can give results that vary between runs and depend on size of run



EXERCISE: MPI reduction

```c
#include <stdio.h>
#include <mpi.h>

int main(int argc, char* argv[])
{
  MPI_Init(NULL, NULL);

  float delta = 0.000000059604645;
  int mpi_rank, mpi_size;
  MPI_Comm_rank(MPI_COMM_WORLD, &mpi_rank);
  MPI_Comm_size(MPI_COMM_WORLD, &mpi_size);

  for (int i=0; i<mpi_size; i++)
  {
    float global_sum, local_val;
    if (mpi_rank == i) local_val = 1.0f;
    else local_val = delta;
    MPI_Reduce(&local_val, &global_sum, 1, MPI_FLOAT, MPI_SUM, 0, MPI_COMM_WORLD);
    if (!mpi_rank) printf("big val to rank %d, gives global sum = %1.10f\n", i, global_sum);
  }

  MPI_Finalize( );
}
```

EXERCISE: MPI reduction

```
id@cpu1xx$ vi reduce.c
id@cpu1xx$ make reduce
id@cpu1xx$ mpirun -n 3 ./reduce
rank 1/3 running.
rank 2/3 running.
rank 0/3 running.
big val to rank 0, gives global sum = 1.0000001192
big val to rank 1, gives global sum = 1.0000000000
big val to rank 2, gives global sum = 1.0000000000
```

EXERCISE: MPI reduction

- What happens if you run it again?
  - Do the processes still return in the same order?
  - What is the result?
- What happens if you increase the number of processes, e.g. to 4?
- What happens if you load a different MPI (e.g. mpich) and build/run?
  - Do OpenMPI and MPICH give the same results?
  - Why or why not?

Central Limit Theorem:

The average values of sets of samples of a randomly distributed variable will tend to be distributed normally:

$\mathcal{R} \in [0,1)$; for a uniformly distributed random variable $\mathcal{R}$

$R_j \triangleq (1/N)\, \Sigma_i \mathcal{R}_i$; with average values R_j for sets of N samples of $\mathcal{R}$

$p(R_j) \propto \exp\{-(R_j - \langle R_j \rangle)^2/2\}$; those average values R_j are distributed normally

Central Limit Theorem example:

$\mathfrak{R} \in [0,1)$; for a uniformly distributed random variable $\mathfrak{R}$
$R_j \triangleq (1/N)\, \Sigma_i \mathfrak{R}_i$; with average values for a sets of N samples of $\mathfrak{R}$
$p(R_j) \propto \exp\{-(R_j - \langle R_j \rangle)^2/2\}$; those average values are distributed normally

The experiment:

- Using an MPI code with different random number seeds scattered to different MPI processes, sample values of R_j are generated and gathered.
- MPI parallel processes then run and complete, although in an indeterminate order
- Because of rounding, results of a reduction depend on the order in which operations are performed
- MPI operations can give results that vary between runs and depend on size of run

EXERCISE: central limit theorem

```c
#include <stdio.h>
#include <mpi.h>
#include <math.h>
#include <stdlib.h>

int main(int argc, char* argv[])
{
  MPI_Init(NULL, NULL);

  int elements_per_proc = 8;
  int initial_seed = 123456;

  int mpi_rank, mpi_size;
  MPI_Comm_rank(MPI_COMM_WORLD, &mpi_rank);
  MPI_Comm_size(MPI_COMM_WORLD, &mpi_size);

  if (!mpi_rank) printf("running %d elements per task with %d tasks.\n", elements_per_proc, mpi_size);

  // create seeds
  int scatter_seeds[mpi_size];
  scatter_seeds[0] = initial_seed;
  if (!mpi_rank) for (int i=1; i<mpi_size; i++) scatter_seeds[i] = scatter_seeds[0] + i;

  // scatter the seeds
  int seed;
  MPI_Scatter(scatter_seeds, 1, MPI_INT, &seed, 1, MPI_INT, 0, MPI_COMM_WORLD);
```

EXERCISE: central limit theorem

```
// seed the RNG
srand(seed);

float sum = 0;
for (int i=0; i<elements_per_proc; i++)
{
  float sample = (float)rand() / (float)RAND_MAX;
  sum += sample;
}
sum /= elements_per_proc;

// gather and bin the results:
float results[mpi_size];
MPI_Gather(&sum, 1, MPI_FLOAT, results, 1, MPI_FLOAT, 0, MPI_COMM_WORLD);
```

EXERCISE: central limit theorem

```
  int bins[10];
  for (int i=0; i<10; i++) bins[i] = 0;

  if (!mpi_rank) for (int i=0; i<mpi_size; i++)
  {
    printf("binning %f\n", results[i]);
    bins[(int)(results[i]*10)]++;
  }

  if (!mpi_rank) for (int i=0; i<10; i++)
  {
    for (int j=0; j<bins[i]; j++) printf("X");
    printf("\n");

  }
  MPI_Finalize();
}
```

EXERCISE: central limit theorem

```
id@cpu1xx$ vi central_limit.c
id@cpu1xx$ make central_limit
id@cpu1xx$ mpirun -n 48 ./central_limit
running 8 elements per task with 48 tasks.
binning 0.488702
binning 0.511513
. . .


XX
XX
XXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXX
XXXXXXXX
XX
X
```

EXERCISE: central limit theorem

- What are the likely sources of error and uncertainty?
- What happens if you run it again for a different seed value?
- What happens if you increase the number of MPI processes?
- What happens if you use a different number of samples for each R_j?
- What happens if we have very small values of $\mathcal{R}$?

Takeaways:

- Greater precision isn’t always ‘better’
- Knowing when you can use a smaller type can often mean better performance
- Making good choices here depends on understanding the limits of machine numbers
- Operations can give results that vary between runs and depend on size of run
- ‘Randomness’ is not always an implicit source of error, it depends on how it’s used.
- There is a new floating point standard in play (posits) which looks to gain traction
- Limits of machine precision are inherent, no matter the standard