1. Monte Carlo method

The Monte Carlo (MC) methods are a class of computational algorithms that rely on repeated random sampling to approximate numerical results. Think of them like "computational gambling", instead of calculating an exact answer, we repeatedly take random guesses and average the results. A key strength of Monte Carlo is its ability to handle high-dimensional integrals that are intractable for deterministic methods due to the *curse of dimensionality*. Specifically, we will use a Monte Carlo algorithm to compute the ratio between the 'volume' of a hiper-sphere of unit radius and the 'volume' of the hiper-cube in which it is inscribed in any dimension.

1.1. Volume ratio of sphere-cube in N-dimensional space

We are going to compute the ratio between a N-dimensional hypersphere and an N-dimensional hypercube. It sounds very complicated, but as we are going to see, it is a simple idea.

Let's start thinking about it in 2 dimensions, here we consider a circle of unit radius and the square in which it is inscribed that has a side of 2.0. The area of a circle is πr^2 and the area of a square is $l^2 = (2r)^2$, so this ratio is:

$$\phi_2 = \frac{\pi r^2}{(2r)^2} = \frac{\pi}{4}$$

.

For 3D, we consider a sphere and a cube, in this case:

$$\phi_3 = \frac{4/3\pi r^3}{(2r)^3} = \frac{\pi}{6}$$

For any dimension, it can be proven that this ratio is:

$$\phi_d = \frac{\frac{\pi^{d/2} r^d}{\Gamma(\frac{d}{2}+1)}}{(2r)^d} = \frac{\pi^{d/2}}{2^d}\frac{1}{\Gamma(\frac{d}{2}+1)}$$

where $\Gamma(n)$ is Euler's Gamma function which, for integer numbers, can be defined as:

$$\Gamma(n) = (n-1)!$$

Also, in the C math header is defined `double tgamma(double arg)`.

In the right plot of figure 1 you can see that the ratio sphere/cube volume decreases greatly for moderate values of d. This is going to produce a challenge when we try to solve this value using Monte Carlo algorithms and it is a manifestation of the 'curse of dimensionality'.

1.2. Monte Carlo Estimation of ϕ_d

Using a Monte Carlo algorithm to stochastically estimate the ratio r_d is straightforward:

1. We sample points $\mathbf{x}_i$ uniformly inside the consider hypercube.
2. Count the fraction of points that are inside the hypersphere, meaning that they satisfy that the norm of their coordinate vector is less than one: $|\mathbf{x}_i| \leqslant 1$.

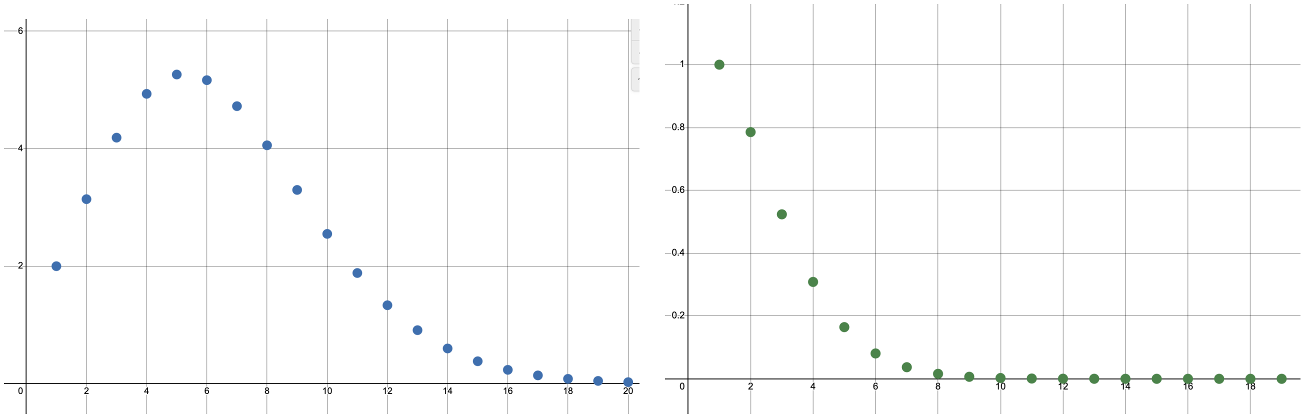

Figure 1: Left: Volume of a unit hiper-sphere as a function of d. Right: Ratio between the volume of the unit hiper-sphere and the hypercube as a function of d.

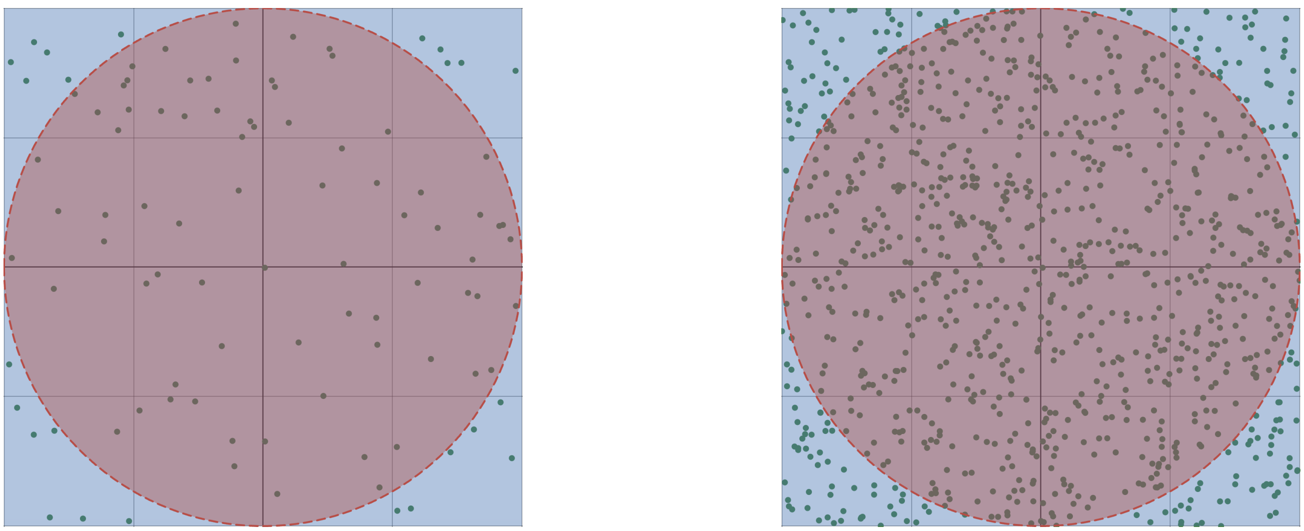

Figure 2: Comparison of Monte Carlo results: (left) 2D sampling visualization, (right) volume ratio by dimension

3. The ratio can be computed: $\phi_d = \frac{\text{points inside sphere}}{\text{total samples}}$
4. The estimation error should decrease *slowly* with the total number of samples.

You can see a graphical representation of the 2D version of this in Figure 2 for the sample points n = 100 (left) and n = 1000 (right).

The Monte Carlo approach leverages the law of large numbers and is trivial to parallelize, making it ideal for high-dimensional computation. However, as you can see 1, when d grows, the sampling efficiency decreases exponentially since ϕ_d very quickly, necessitating variance reduction techniques or alternative methods for large d.

1.3. Implementation

You can start the practice by implementing a simple hello world, as seen in the theory session. Take the `Makefile` and job.sh from Lab1 and adapt them to work with MPI.

- In the `Makefile`, remember to update the compiler, the source file, and the executable names.
- Check `job.sh`, you should adapt the SBATCH arguments to use MPI.

Once everything is running, you should see something like this:

```
Hello world from rank 6 / 8
Hello world from rank 3 / 8
Hello world from rank 0 / 8
Hello world from rank 4 / 8
Hello world from rank 5 / 8
Hello world from rank 7 / 8
Hello world from rank 2 / 8
Hello world from rank 1 / 8
```

Now, you can implement the Monte Carlo estimator. The goal is to reproduce this output:

```
$mpirun -n 12 ./montecarlo 4 100000000 10

Monte Carlo sphere/cube ratio estimation
N: 100000000 samples, d: 4, seed 10, size: 12
Ratio = 3.085e-01 (3.084e-01) Err: 1.05e-04
Elapsed time: 0.088 seconds
```

Hints:

- To compile and run the jobs, you will need mpicc and mpirun. To have them available, you need to load modules in the cluster:

  ```
  module load gcc/13.3.0
  module load openmpi/4.1.1
  ```

- You can take these defaults:

  ```
  int d = 3;
  long NUM_SAMPLES = 1000000;
  long SEED = time(NULL);
  ```

  and then, if the user introduces three command line arguments: N, NUM_SAMPLES, and SEED, override the default values.

- As the random generator of C has statistical limitations for a large number of random numbers, use this one:

```c
typedef struct { uint64_t state;  uint64_t inc; } pcg32_random_t;
double pcg32_random(pcg32_random_t* rng)
{
    uint64_t oldstate = rng->state;
    rng->state = oldstate * 6364136223846793005ULL + (rng->inc|1);
    uint32_t xorshifted = ((oldstate >> 18u) ^ oldstate) >> 27u;
    uint32_t rot = oldstate >> 59u;
    uint32_t ran_int = (xorshifted >> rot) | (xorshifted << ((-rot) & 31));
    return (double)ran_int / (double)UINT32_MAX;
}
```

and then initialize like this:

```c
    pcg32_random_t rng;
    rng.state = SEED + rank;
    rng.inc = (rank << 16) | 0x3039;
```

and then obtain a double precision random number from zero to one with:

```c
double ran = pcg32_random(&rng);
```

Notice that every rank initializes the random generator with a different value based on its index. This assures that the random sequences are completely different for every rank.

- NUM_SAMPLES refers to the **total** number of samples.
- Use `MPI_Wtime()` to measure computational time.
- As the number of samples is going to be very large, be careful with overflow issues.
- The elapsed time shown must be the **maximum** of the processing times.

Report Questions 1 — Monte Carlo (25%)

1. Explain the modifications you made in the Makefile and job.sh to make it work for an MPI program.
2. Describe your approach to designing the program from a parallel computing perspective
3. Setting d=10 and starting in 100 million sample points, plot its strong and weak scaling from 1 to 192 processors. Include the job script used to generate this data in the code zip
4. What happens with the ratio computation error when you increase the number of samples?

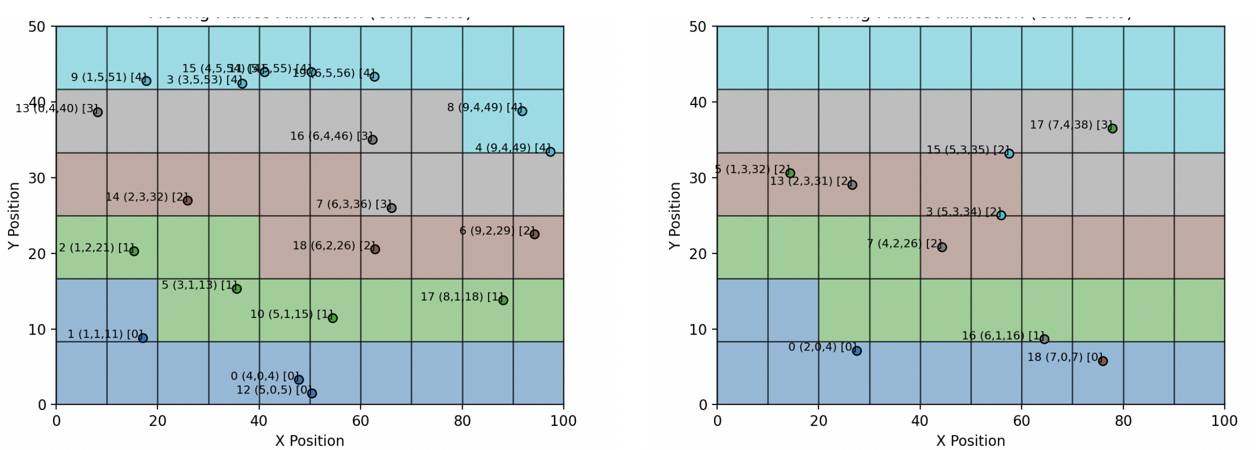

Figure 3: Example of simulation in a physical space of 100x50 and with a grid of 10x6. Background colour represents each rank. Each point represents a plane and shows its id, i, j, map_index, and rank.

2. Flight Controller

In this practice, we will work with a flight simulation that models the motion of multiple planes, ideally millions, over a 2D grid. Each plane has a position and a velocity, and it moves continuously across a domain that is divided into a uniform grid of cells of size N × M. For simplicity, we assume that all planes move in straight lines, following their constant velocity vectors.

We provide a working sequential version in the file `main_seq.c`. In this version, the information for all planes is stored and updated by a single process. Although this approach works well for small scenarios, it becomes increasingly inefficient as the number of planes or the size of the domain grows. To improve performance and scalability, we will need to parallelize the simulation. Moreover, our goal here is to mimic a decentralized architecture, where each region of the grid is managed independently, making MPI (Message Passing Interface) a perfect fit for this task.

2.1. Sequential version

The sequential version of the simulation models the motion of multiple planes over a 2D grid using a simple time-stepping approach. It runs entirely on a single process and follows this high-level structure:

1. **Argument parser:** The simulation program expects exactly `4` command-line arguments. The parser checks their validity and exits if incorrect values are given.

   ```
   ./fc_seq <filename> <max_steps> <mode> <check>
   ```

   - `<filename>`: Path to the input file containing initial plane data.
   - `<max_steps>`: Number of simulation steps (must be a positive integer).
   - `<mode>`: Communication mode (must be 0, 1, or 2). Only important for the parallel version.
   - `<check>`: Check option mode. 1: checks that the simulation is correct. 0: it does nothing.

2. **Input files:** The input file contains metadata about the simulation domain and a list of planes, one per line. An example file with four planes is this:

```
# Plane Data
# Map: 100.00, 50.00 : 10 6
# Number of Planes: 4
# i x y vx vy
0 47.83633 3.27652 -0.41423 0.07794
1 17.11402 8.78723 -0.36607 0.03621
2 15.37492 20.2739 0.01260  0.61355
3 36.68571 42.4225 0.39372  -0.3551
```

   In the header we can find:

   - `Map:  x_max, y_max :  N M`: Physical size and grid resolution.
   - `Number of Planes`: Total number of planes listed.

   The rest of the file is the planes, each in one row:

```
<ID> <x> <y> <vx> <vy>
```

   For this practice, we provide four files.

   - `input_planes_test.txt`: For testing, just 5 planes moving in vertical lines
   - `input_planes_100.txt`: For further testing, hundred planes moving randomly.
   - `input_planes_1k.txt`: For further testing, thounsand planes moving randomly.
   - `input_planes_10kk.txt`: Ten million planes, for scaling analysis.

3. **Reading input:** Plane data is read from a text file using the function:

```
read_planes_seq(const char* filename, PlaneList* planes, int* N, int* M,
        double* x_max, double* y_max)}
```

   which parses the grid size ($N \times M$), the physical domain limits (`x_max`, `y_max`), and the initial positions and velocities of all planes. The initial data is read. Each plane is stored in a dynamically managed double-linked list: `PlaneList`.

4. **Data representation:** Planes are stored using the `PlaneNode` structure, which holds their position (x, y), velocity (vx, vy), unique ID, and computed grid index. All nodes are linked in a `PlaneList`. We can manage this list as:

```
void insert_plane(struct PlaneList* list, int index_plane, int index_map,
        int rank, double x, double y, double vx, double vy);
void remove_plane(PlaneList* list, PlaneNode* node);
PlaneNode* seek_plane(PlaneList* list, int index_plane);
PlaneList copy_plane_list(PlaneList* list);
```

5. **Map representation:** The physical map is divided in tiles in both 2D directions, the number of tiles is set in the input file NxM. Each tile is represented by a i,j pair or by a global map_index. We can compute the grid indices corresponding with the physical coordinates with these functions:

```
int get_index_i(double x, double max_x, int N);
int get_index_j(double y, double max_y, int M);
int get_index(int i, int j, int N, int M);
```

6. **Simulation loop:** At each time step:
   - Every plane's position is updated based on its velocity.
   - The list is filtered using `filter_planes` to remove planes that exit the domain boundaries.
7. **Correctness check:** At the end of the simulation, `check_planes_seq` compares the initial and final states to verify that planes have moved as expected over the total number of steps.

2.2. Parallel version

The goal of this practice is to prepare the sequential version to be used in parallel using MPI for a very large number of planes. The main points that we need to do are the following:

1. Complete the Makefile and job.sh
2. Add any variables needed for the main program
3. Parallel read input file
4. Plane communication.

2.3. Tile displacement

The main variable you need to create is the `tile_displacement` that you must use in different parts of the code. This is a vector whose size is the number of ranks plus one, whose `i` component is the first map tile belonging to this rank. In this way, the rank `i` will contain the data from `tile_displacements[i]` to `tile_displacements[i+1]` (without including it). The last element is the total number of tiles. Once you have computed this vector, you can use these helper functions included in the auxiliary header:

```
int get_rank_from_indices(int i, int j, int N, int M, const int* tile_displacements, int
    size);
int get_rank_from_index(int index, const int* tile_displacements, int size);
```

where the first can be used with the indices (i,j) and the second with a global map index. You should compute this vector inside the read function, once you have read the NxM variables from the input file. The distribution of tiles in ranks should be done in the most balanced way possible, assuming that the total number of tiles does not need to be divisible by the number of ranks.

2.4. Parallel read input file

```
void read_planes_mpi(const char* filename, PlaneList* planes, int* N, int* M,
        double* x_max, double* y_max, int rank, int size, int* tile_displacements);
```

- This function reads the input file and creates the plane list at each rank.
- It needs to set the arguments `N, M, x_max y_max` from the file header.
- Once you have read N and M, you can already compute the `tile_displacements` vector. Do it inside this function.
- Each rank creates a PlaneList corresponding only to the read planes contained in the tiles belonging to that rank.
- You don't need to use the MPI I/O methods. All ranks can use the standard C methods independently.

2.5. Plane communication

```
void communicate_planes_send(PlaneList* list, int N, int M, double x_max,
        double y_max, int rank, int size, int* tile_displacements)
void communicate_planes_alltoall(PlaneList* list, int N, int M, double x_max,
        double y_max, int rank, int size, int* tile_displacements)
void communicate_planes_struct_mpi(PlaneList* list, int N, int M, double x_max,
        double y_max, int rank, int size, int* tile_displacements)
```

- At each step, the position of the planes are updated and then, we need to check if the planes that have gone outside of the tiles belonging to our rank and send its data.
- The key data to send is: its index_map, x, y, vx, vy.
- We will implement three different sending approaches
  - Send: implement a version that needs to be communicated using Send/Recv strategy for each plane. You can choose synchronous or asynchronous functions. Remember that they can be mixed, e.g. you can have an asynchronous Send and synchronous Recv. The Send/Recv is needed to be used for the communication of plane information, to communicate the number of planes, a collective like MPI_Alltoall is allowed. For simplicity, convert the index_map to double before send it.
  - Alltoall: implement a version that all to all communication. For simplicity, convert the index_map to double before send it.
  - Using the same all-to-all as the previous, but sending a custom datatype. Instead of using the PlaneNode as the base of the MPI datatype, you can use an intermediate struct with only the relevant data to send. Note: For computing the offsets, you can better used the function `offsetof` in the `stddef.h` header.

    ```
            offsets[0] = offsetof(Struct, var1);
            offsets[1] = offsetof(Struct, var2);
    ```

2.6. Debugging Tools

- Set `int debug = 1.0` to change the output of the check function. If debug is zero, only missing planes are showed, if set to one, also the correct ones are shown.

- You can also use the function print_planes_par_debug() to print all the planes. This function outputs sequentially the planes at all the ranks, so it is slow and verbose, but it can be used for debugging.

2.7. Summary

Once you have completed all the steps you should be able to see something like this:

```
mpirun -n 5 ./fc_mpi input_planes_1kk.txt 25 1 0
Total planes read: 1000000
Flight controller simulation: #input ../input_planes_1kk.txt mode: 1 size: 5
Time simulation:     0.44s
Time communication:  0.75s
Time total:          1.19s
```

If you enable the check option, this will check that your implementation is correct. Check only the small files, not the 1kk or 10kk files. Notice that this check only works if the reading is correct. So check this carefully before continuing. If you have any problem with any plane, you should see a message like this:

```
Missing Plane 0! The plane should be inside the map, but it could not be found.
   Expected: (10.00, 21.00) 21, 2
Missing Plane 1! The plane should be inside the map, but it could not be found.
   Expected: (30.00, 21.00) 23, 2
```

Report Questions 2 — Flight controller (75%)

1. Analyze the sequential version of the simulation. What are the main parts that you need to parallelize? What are the challenges?
2. Regarding the output, how have you managed to parallelize it? What could be the bottlenecks for a large number of ranks?
3. Discuss the different communication options. Check them `input_planes_10kk.txt` with a moderate number of ranks of 20. What are the key differences between them? Do you see a communication time difference? Why? Include the job script used to generate this data in the code zip.
4. Using the same file, use the best communication strategy and increase the number of ranks. Use 20, 40 60, and 80 ranks. Analyze what you observe. Include the job script used to generate this data in the code zip