

# Assignment - Particles in a box

---

---

## Simulation of particles in a box

We are to construct a program that simulates a number of particles moving in a box. Such a system can be used to simulate e.g. gas in a closed container.

The particles are confined to a unit box with the lower left corner at (0,0) and the upper right corner at (1,1). Particle sizes are set to random values between 0.005 and 0.015 and given an initial velocity of  $v_0 = 0.01$ .

Each particle has a position,  $s$  a speed  $v$ , and a radius  $r$ .

Initially, all particles are randomly positioned in the box. Each particle has the same speed but moves in different directions. The speed vector can be computed as,  $v_0 \sin \alpha$ , where  $\alpha$  is a random number in  $[0, 2\pi]$ .

The particles move the distance  $\Delta t$  in each timestep according to the following criteria as.

1. Decide whether any particle collides with a wall in the box (this means that the distance from the border to the particle is less than the radius). If a collision occurs, the particle should bounce.
2. Decide whether two particles collide. If particles  $i$  och  $j$  collide, use the formula:

$$v_i = v_i + \frac{(v_j - v_i) \cdot (s_j - s_i)}{|s_j - s_i|^2} (s_j - s_i)$$

$$v_j = v_j + \frac{(v_i - v_j) \cdot (s_j - s_i)}{|s_j - s_i|^2} (s_j - s_i)$$

---

to update their speed. (The operation denotes dot product) Ignore the case where more than two particles collide.

3. When all particles have been tested, they move to a new position using the formula:

$$s = s + \Delta t \cdot v$$

---

The time step  $\Delta t$  should be selected so that the particles don't pass through the walls. A suitable value is  $\Delta t = r / (3 * v_0)$ . Input to the simulation is the number of particles, initial speed, and radius.

The skeleton code for this assignment can be downloaded here:

[skeleton code.tar.gz](#) (folder path: code/problem1)

[skeleton code.zip \(windows\)](#) ↓

Please look at the README.md in both of these examples.

## Task 1 - Fortran implementation

The implemented Fortran application should use modules for its implementation. Divide the application into logical modules:

- mf\_datatypes
  - Definition of constants that will be used throughout the application. Constants for `selected_real_kind()` and `selected_int_kind()` could be put in this module.
- mf\_utils
  - Utility routines for random numbers and printing matrices.
- app\_data
  - Could contain data structures and arrays used in the application. Routines for initializing, reading/writing, and destruction of data could also be located in this module.
- app\_sim
  - This module contains the actual routines that handle the actual simulation.
- app\_utils
  - Utility routines that are used throughout the application.
- vector\_operations
  - Module not specific to the application, but contains generic reusable code.

The app prefix is just a suggestion. Replace this with something that is more suitable for your application.

Use CMake to handle makefile generation. A sample CMakeLists.txt file is shown below:

```
cmake_minimum_required(VERSION 3.0)

project(particles)
enable_language(Fortran)
add_executable(particles main.f90
  mf_datatypes.f90 mf_utils.f90 app_utils.f90 app_data.f90
  app_sim.f90 vector_operations.f90)
```

To aid the debugging of the Fortran code, a special Python script is provided, `particle_player_vedo.py`, which can be downloaded from the course page. This script reads particle sizes and positions from a text file and displays a visualization of the movement of particles. The file format used is described below:

```
[number of particles n]
[particle size 1]
.
.
.
[particle size n]
[Number of particles, n, for timestep 1]
[x1] [y1] [z1]
.
.
.
[xn] [yn] [zn]
[Number of particles, n, for timestep 2]
[x1] [y1] [z1]
.
.
.
[xn] [yn] [zn]
[Number of particles, n, for timestep m]
[x1] [y1] [z1]
.
.
.
[xn] [yn] [zn]
```

## Task 2 - Create an F2PY interface for the Fortran code

In this task, the previous Fortran application will be given a Python interface using the F2PY tools. To make it easier to use the F2PY tool, a special Fortran module will be implemented that represents the interface to Python. This module should typically contain subroutines for driving the simulations (check\_collision, update\_positions, etc).

To make it easier to wrap the Fortran code, only the interface module will be converted by the F2PY converter. Other code will be compiled as a library separately and linked into the Python module from the F2PY command. A library can be compiled with CMake by adding the following to the existing `CMakeList.txt` file:

```
add_library(applib SHARED app_defs.f90 app_utils.f90
    app_data.f90 app_sim.f90 vector_operations.f90)
```


The python module can then be built using the following command:

```
f2py -m app -c app_interface.f90 -I./ build -L./ build -lapplib
```

When the extension module has been built, implement a python based main program that drives the simulation.

## Task 3 - Visualise particle movement using Vedo

Use the Python Vedo visualization library to visualize the particle movement for every simulation step.

[vedo \(embl.es\)](https://vedo.embl.es/)  (<https://vedo.embl.es/>)

To use vedo it is a good idea to create a virtual environment as it has specific python requirements for its dependencies. An environment for vedo can be created using the following commands (vedo-project can be changed to whatever you like):

```
conda create -n vedo-project python=3.7
```

To install vedo, activate the environment and use pip to install the package:

```
conda activate vedo-project  
pip install vedo
```

To verify that vedo works, the vedo\_demo.py application can be run in the skeleton\_code root directory.

```
python vedo_demo.py
```