Fortran final project
3: Velocity filter simulation

By Ahti Katiska, 014257483

1. Introduction

The problem was to simulate the movement of charged particles with given mass, charge and start velocity in a homogeneus electric and magnetic fields. The particles' movement was restricted by the existence of a rectangular tube, through which the particles were moving. The aim of the simulation was to filter out such particles which got through this aforementioned tube without collision with the tube.
In short the the aim was to simulate a more general type of a Wien filter. The generality comes from the freedom of selecting the electric and magnecit field to be not perpendicular with each other.

Motivation to automate this simulation springs from the fact that analyzing by hand a massive amount of different particles, in a set of different environments, would be, to say the least, cumbersome.

2. Methods

To simulate motion of a particle, one must know the forces affecting the particle. In our case, we assume the only force affecting the particles to be the Lorenz force:

F = q * (**E** + **v** x **B**), where q is the charge of the particle
E is the electric field
v is the velocity of the particle, and
B is the magnetic field.
(*note all bolded letters are vectors)

And to calculate the posititon, acceleration and velocity with discrete time steps, one needs a numerical method to do so. In our case the velocity Verlet algorith was used.
Steps of the velocity verlet algorithm:

1. x(t + dt) = x(t) + v(t)*dt + ½ * a(t)dt^2
2. a(t + dt) = F(x(t + dt)) / m
3. v(t + dt) = v(t) + ½ * (a(t) + (a(t + dt)) dt

3. Implementation

The implementation of the program is separated between three main modules: the particledef module, the io module and the simulation module. Each of these modules will be discussed seperatetly.

The particleDef module:
The particleDef module does not contain any procedures, it is in fact only the container for some parameters and the definitions of the particle datatype and he resParticle datatype.
The particle datatype is the way the program represents particles; it has a mass and a charge, both of which are scalar real values, and a position, velocity and acceleration, which are three dimensional arrays of reals(vectors). The resParticle is a tuple with a particle and a logical value depicting of whether or not the particle succeeded in passing the simulation.

The io module:
The io module, is as it name suggests responsible for all the input-output of the program. It has many ugly subroutines which do ugly things. I'll spotlight only the public subroutines as they are the only ones that are imperative to the

init_all subroutine inits all the information used by the simulation, it reads the particles from file; timestep, magnetic field and electric field from command line, and saves them to the variables passed on to it.

write_file subroutine takes a list of particles and writes them down to a output file in the same format as the particle input file was read.

The simulation module:
The simulation model is the interesting part of the program, it handles the force calculation, verlet algorithm etc. The only public subroutine of this module is the run_sim subroutine. I'll now walk through what this subroutine does, which should explain everything.

First the run_sim calls init_all, which is a subroutine of the io module, that reads all the data provided, and in the case of incorrect input will return to label 10 which is the failure route. If initialization of the values was a success, then the program checks if the p_particles array, which is of type particle, is allocated, and if it is deallocates it. After all before any simulation is run how could the p_particles array have anything in it(p_particles comes from passed_particles).

Then comes the main simulation loop of the program, it runs the simulate_particle function on every particle of the particles list. The simulate_particle function is fairly simple, it just calls the verlet algorithm again and again as long as the particle does not collide with the tube, and has not passed through. The verlet subroutine and the lorenz_force function are quite trivial and are not discussed here.

The simulate_particle function returns a resParticle(tuple) which contains the data of the particle at the end of simulation and the logical value of success.

The return value of this function is then saved in a temporary variable. Then the temporary variables particle part is pushed to the p_particles list if the temporary variables logical part is true. Simply said if the particle had passed the simulation.

And at the end of the day the p_particles list is written to file using the io modules write_file subroutine.

Usage:

Compiling:
gfortran particleDef_module.f95 io_module.f95 simulation_module.f95 main.f95

Running:
./a.out <timestep> <b(x)> <b(y)> <b(z> <e(x)> <e(y)> <e(z)> <input filename> Or
./a.out -help

4. Results

The program manages to simulate particles, I won't take a stance whether or not it is memory efficient, or fast. The program has some sanity checks on the input, but it fails to notice "too large" timestep, so if the timestep is large enough the particles speed will rocket in one timestep, causing the program to fail printing them in the output file, because of the sheer size of the numbers in question.

On a test pattern of first 4 and then 8 particles, with electric and magnetic fields of those described in the topic of the assignment, and a timestep of 0.005 seconds, the program worked just as expected.

5. Conclusions

If one would have the motivation and/or reason to make this program more memory efficient, parallelizable etc etc it could be done with quite a few simple changes to the code. Memory efficiency for example would skyrocket if one would simply calculate which kind of reals would be enough to simulate this problem without catastrophic error, the default real kind is 300, which seems to be a bit overkill.

Parallelization could be achieved quite easily with the usage of forall construct in the main simulaiton loop. This is made easier by the fact that all functions/procedures used in simulation are pure.

All in all the project is of the standard what you'd expect from a lazy undergrad student.