CParticle

Nikos Tryfonidis

2015

Abstract

The current report summarizes the structure, use and results of *CParticle*, a charged particle simulation code written in C. *CParticle* simulates the motion of a charged particle (ion or electron) in electromagnetic fields, that are specified by the user in vector form (Cartesian coordinates). First, a small introduction to the code will be given in chapter 1. Afterwards, the structure of the code will be reviewed in chapter 2. Finally, a number of test cases will be shown, as examples on how to use the code while also verifying its correctness.

Contents

Chapter 1

Introduction

CParticle is a three-dimensional charged particle simulation code written in C. The code was written as part of my PhD project, in order to gain insight into the numerical simulation of charged particle motion and eventually to be integrated into a Particle-In-Cell code, as the particle "pusher".

In this chapter, a brief introduction to the code will be made, with instructions on how to compile, run and view the simulation results.

1.1 Compiling and Running

The code should compile without problems on any machine able to compile C code with "make". To compile, simply go to the main directory, where the makefile is located, and type "make". This will build the executable (named "*cparticle*") in the same directory. To run the code, simply run the executable from the command line, providing four (4) arguments:

```
./cparticle <particle type> <total time> <dt> <output interval>
```

particle type Choose *"i"* for ion (positive charge) or *"e"* for electron (negative charge).

total time The total time units for the particle motion to be simulated.

dt The timestep for the numerical solver.

output interval Number of steps between output.

For example, if we want to simulate the motion of an electron for 100 time units, with a timestep of 0.001, and want output every 10 timesteps, we will run as:

```
./cparticle e 100 0.001 10
```

Output is written in the "output" directory, in two ".txt" files:

output.txt Contains output in seven columns. The first column is the time for every output step of the simulation. The next 3 columns are the position of the particle (x, y, z). The final 3 columns are the velocity of the particle (v_x, v_y, v_z).

energy.txt The kinetic energy of the particle for every output step.

1.2 Visualization

Visualization of the results is handled with Python scripts that use NumPy and matplotlib, two well-known packages for scientific computing in Python. These can be easily installed in any Linux system, for example in Ubuntu:

```
sudo apt-get install python-numpy python-matplotlib
```

The following visualization scripts can be found in the "plot" directory:

plot_xyz.py Creates the plot of a single chosen coordinate (position, velocity) versus time, and also a 3D plot trajectory of the particle.

animate.py Creates a 3D animation of the particle trajectory.

plot.py This script is called by *plot_xyz.py*. The user does not have to edit or use this directly.

After running the program, to create the plots, the user can simply go to the *plot* directory and do the following:

```
python plot_xyz.py
```

This will produce the $x(t) - t$ and $v_x(t) - t$ plots in one figure, and also the phase-space plot $v_x(t) - x(t)$ in another. After the user closes these two figures, the 3D trajectory plot of the particle motion will be created.
To create the animation of the motion, the user can run the *animate.py* script:

```
python animate.py
```

This will show the animated motion of the particle. If the user wants to save the animation, he should uncomment the last lines in the script (the lines after the comment saying "save animation"), while also commenting out the *"plt.show()"* line. Please note that in order to save the animation, the script uses *ffmpeg*. This has to be installed and be reachable in */usr/bin/*.

Chapter 2

Code Structure

In this chapter, the structure of the code will be briefly reviewed. First, a description of the source files and code will be given, followed by a description of the data structures used. Afterwards, an outline of how to create a simulation (setting up the electromagnetic fields etc).

2.1 Directories

The project consists of the following directories:

src Contains the source files of the project. These will be described in more detail in the next section.

headers Contains headers needed for the source files. Every source files has its corresponding header file, with function headers for the functions that are callable *outside* their source file.

output Output files are written here by the program.

documentation Contains the current documentation.

plot Contains the Python plotting scripts.

obj Contains object files (.o).

Also, the *"input.txt"* file, in the main directory of the project, contains initial conditions for the particle (3 position coordinates and 3 velocity coordinates).

2.2 The Code

The code is contained in the *src* directory, in the following source files:

main.c The main source file. Reads input, allocates memory, calls the solver wrapper and finally writes the output.

fields.c The electromagnetic field functions. Contains the Electromagnetic Force function calculator (*FLorentz*), a cross product evaluating function (*cross*), and functions for the user to set the Electric and Magnetic fields (*EField and BField*, respectively).

motion.c The solver wrapper function (*motion*) in this file calls the solver for the required number of steps and writes output in memory for every interval requested.

solver.c Contains the numerical solver function *RK4_ motion3D*. This function calculates the particle motion for one timestep and returns the particle with updated coordinates. The numerical method used is the *4th order Runge Kutta* method.

memory.c Contains the *array2D_ contiguous* function, that dynamically allocates a 2D array contiguously in memory and *free_ array2D_ contiguous*, which frees memory allocated in the previous way.

io.c Contains functions that read input from the input file and the command line, print the problem parameters to the screen and write output to memory and output files.

The following header files in *headers/struct.h* are also of interest:

definitions.h Contains values for particle (ion or electron) charge and mass. These are currently set to 1.0 or -1.0 respectively.

struct.h Contains the structure definitions for the program. These are described in the next section.

2.3 Data Structures

The program uses the following data structures, defined in *headers/struct.h*:

vector3D A 3D vector, consisting of 3 double variables, one for each coordinate. A very useful data structure for this program, vector structures are used in the *particle* structure and by the *field* functions in *field.c*.

```
struct vector3D {
    double x;
    double y;
    double z;
};
```

particle The main data structure of the program. Contains information about the particle whose motion is simulated. *Position* and *velocity* are *vector structures*. Also contains the particle *charge* and *mass*. These could have been defined as macros, but as the program aims to be part of a Particle-In-Cell code simulating many particles (both ions and electrons), this information was included in the particle structure.

```
struct particle {
    struct vector3D r;
    struct vector3D v;
    double q;
    double m;
};
```

time A structure containing the time parameters of the program, in order to pass them in a neat way to functions.

```
struct time {
  double totalTime;
  double dt;
  int output_interval;
  int totalTimeSteps;
  int nOutput;
};
```

2.4 The Numerical Method

The program essentially solves the charged particle equations of motion, following Newton's 2nd Law formalism:

$$\frac{d^2\overrightarrow{r}}{dt^2} = \frac{\overrightarrow{F_L}}{m} \tag{2.1}$$

where

$$\overrightarrow{F_L} = q\left(\overrightarrow{E} + \overrightarrow{v} \times \overrightarrow{B}\right). \tag{2.2}$$

Runge-Kutta (4th order) was chosen as the numerical method for the motion for the particle. Being a multistage method, Runge-Kutta requires the calculation of the right-hand-side of (2.1) for each of the four stages it calculates. A decision was made to write the solver function in the clearest possible way, avoiding switches and other methods that would make it more compact but much less clear.

A detailed analysis of Runge-Kutta is outside the scope of this report, but can be found in most Numerical Analysis textbooks dealing with the numerical solution of Ordinary Differential Equations, such as [1].

2.5 Setting Up a Simulation

To set up a particular problem, the user has to do the following actions:

- Set the desired Electric and Magnetic fields in *fields.c*.
- Set the definitions for particle mass and charge in in *headers/definitions.h*.
- Compile the program by typing "make".
- Set the desired initial conditions for the particle in *"input.txt"*.
- Run the program, as described in the introduction, specifying the time parameters.

After running the program, output is written in the *output* directory and can be visualized with the Python scripts in the *plot* directory, as shown in the introduction.

Chapter 3

Examples

In this chapter, a number of representative problems will be shown. A detailed analysis and study of charged particle motion is outside the scope of this report, but excellent references include [2], [3] and [4].

3.1 Simple Gyromotion

3.2 $\boldsymbol{E} \times \boldsymbol{B}$ Drift

3.3 Magnetic Dipole

Bibliography

[1] Randall J. LeVeque *Finite Difference Methods for Ordinary and Partial Differential Equations.* 2007.

[2] Robert J Goldston, Paul H Rutherford, *Introduction To Plasma Physics* 1995.

[3] Jeffrey Freidberg, *Plasma Physics and Fusion Energy* 2007.

[4] Paul M. Bellan *Fundamentals of Plasma Physics* 2006.