UCLA 3D diffusion code
Description

Table of Contents

Introduction

The program is solving three dimensional Fokker-Plank equation <<equation>> on irregular grid with different types of boundary and initial conditions. Diffusion coefficients are calculating for different types of waves: day and night Chorus waves, plum EMIC waves, plum hiss waves, inside of the plasma pause hiss waves. The program output flux in <<units>> and phase space density in <<units>> on different radial distance from the Earth for different energies and pitch-angles.

All output files has format:

First line hold list of all variables present in file.
Other file divided into “zones” contains different data sets, for example different time slices (can be only one zone). First line of each zone hold description of sizes of arrays “K = xxx”, “J = xxx”, “I = xxx”, followed by column(s) of variables values and optional zone name.

For exemple:

VARIABLES = "L", "Energy", "Pitch-Angle", "Daa day side Chorus"
ZONE T = "Time is 1 day" K = 7, J = 5, I = 3

1	*1.43841*	*0.00453988*	*6.35503e-007*
1	*1.43841*	*1.02493*	*1.28952e-007*
1	*1.43841*	*1.56007*	*9.05082e-008*
1	*5.50889*	*0.00453988*	*5.09457e-008*

...

Constants

Enclosed structures, that hold all constants, loaded from .ini file.
Constants.h – header file.
Constants.cpp – source file.

Important functions:
FileToMap(string “file name”, map “map name) - read all variables from file to map.
ReadStructureFromMap(“map”) - fill all constants in structure from map;
ReadFromMap(“variable”, “map to read from”, “name in map”) - read specific variable value from map.
StrToVal(“variable value in text”, “variable”) - convert value written like text to value.

Constants are loading from .ini file with format:
VariableName1 = Value
VariableName2 = Value
VariableName3 = Value
...
into map (actually in multimap, read C++ manual for details) with format map(string “variable name”, string “variable value”) by **FileToMap**.

Then in **ReadStructureFromMap** each structure variable is searched in the map in “variable name” column by **ReadFromMap** and then corresponding “variable value” convert from “string value” to variable value by **StrToVar**.

Matrix

All work with 1D, 2D and 3D arrays: mathematical operations, memory care etc.

Phase Space Density

Hold phase space density array, know how to input and output it, make three diffusions:
radial diffusion,
energy diffusion,
alpha diffusion.

Important functions and methods:
fnpl1_nug("phase space dencity",
"one dimentional grid",
"lifetime",
"power of x",
"lower boundary conditions",
"upper boundary conditions",
"number of grid points",
"time step",
"diffusion coefficient",
"lifetime in loss cone",
"loss cone size",
"diffusion flag",
"type of lower boundary conditions: 0 - for value, 1 - for derivative",
"type of upper boundary conditions: 0 - for value, 1 - for derivative") - make 1-dimensional diffusion.

Output file format:
Two lines header that describe the sizes of the array: "I = xxx", "J = xxx", "K = xxx".
One column of data for each point of array in order I, J, K.

Grid

Creating 3D grids and boundary conditions. In case of 2D one dimension is equal to one. In case of 3D creation of all types of grid starts from orthogonal *Energy – Pitch-angle* grid on max *L*. Then creation continuous to lower *L* according special to each grid-type rules.

Grid.h – headers file.
Grid.cpp – source file.

Classes:

GridElement

One specific element of irregular grid (like radial distance, energy, pitch-angle or just 'x'). Hold 3D array of grid values. In 2D case 3-rd dimension equal 1. In case of regular grid values are varying only by one direction.

Important functions and methods:
Nothing important.

Grid

Hold 3 grid elements that compose *Grid*: *L*-radial distance from the Earth, *pc* – moment multiplied by speed of light, *alpha* – pitch-angle; and one additional element – *Energy* that directly depends of *pc*.

Important functions and methods:
MakeGrid(“grid type”, “second grid (optional)”) - create grid. For some specific grid types it needs second grid to be based on.
MakeBoundaryConditions(“phase space density”) - create boundary conditions for all grid elements.

BoundaryConditions

Hold one boundary condition (2D-array). Each grid element has two instances of this class – upper boundary condition and lower boundary condition.

Important functions and methods:
MakeBoundaryCondition(“2D phase space slice”) - create boundary condition.

Diffusion Coefficients

Make all work with diffusion coefficients.

DiffusionCoefficient.h – headers file.
DiffusionCoefficient.cpp – source file.

Classes:

DiffusionCoefficient

Diffusion coefficient class, hold the matrix with some diffusion coefficients.

Important functions and methods:
Get("grid", "diffusion coefficient parameters") - get diffusion coefficient matrix by calculating it or loading according to parameters.
LoadDiffusionCoefficient(...), **LoadDiffusionCoefficientFromFileWithGrid**(...), **LoadDiffusionCoefficientFromFileWithGrid**(...), **LoadDiffusionCoefficientFromPlaneFile**(...) - loading diffusion coefficient matrix.
Calculate(...) - calculating diffusion coefficient matrix.
MakeDLL(...) - make DLL.

DiffusionCoefficientGroup

Hold *group* of diffusion coefficients of one type and generate combined diffusion coefficient actual at specific time.

Important functions and methods:
Get("grid", "list of diffusion coefficients parameters", "type") - choose from list of diffusion coefficients all with specified "type", **Get** them and add to the *group*. Different type (like Dpp and Dpcpc) can be converted to one type and added together automatically.
Activate("time") - activate and deactivate diffusion coefficients from the list.
GetActual("time") - return time-actualized diffusion coefficient (the same as **Activate** diffusion coefficient group and then use it).

Main program

Connect everything together.
main.cpp – source file.

Simple algorithm looks like this:

1. Loading constants
2. Creating two PSDs: for radial diffusion and for local diffusions.
    1. Creating two Grids
    2. Creating all Boundary Conditions
    3. Initializing output streams.
3. Creating diffusion coefficients
4. Start main loop
    1. Interpolating to radial diffusion grid
    2. Making radial diffusion
    3. Interpolating back to local diffusions grid
    4. Making energy diffusion
    5. Making pitch-angle diffusion
    6. Output results of the step
5. End