Algorithms

Loading parameters from .ini file

In the code we use organized output system. It outputs a text on the screen and the into log file the same time. It was organized by creating class *'Output'* (Output.h, Output.cpp) with static sub-class that hold file name for log-file and produce output.

First of all we initialize output to log file and open stream.

```
Output::setFileName("logfile.log", "clear");
Output::open();
```

Parameter *"clear"* means that file *"logfile.log"* will be erased before use.

Anywhere, where we include "Output.h" we can use function *'echo'* that produce output on the screen and into the log file and has *'printf'*-similar format.

```
Output::echo("Step %d of %d. Time = %d days, %.0f hours.\n", iteration, all_Iterations, time_days, time_hours);
```

Loading parameters from .ini file

Load parameters from file. If code was ran with argument, it used as parameters file name. Otherwise *'constant.ini' is used.*

```
Load_constants(constants, argv[1]);
```

Loads file *'filename'* into multimap *'iniFileMap'*. Map is C++ class, which holds keys and values. Multimap can hold more than one equal keys. Both, key and value have type string.

```
FileToMap(iniFileMap, filename);
```

To find keys and values inside parameters file we are looking for *'='* sign and put word from the left of the sign as *key* and word from the *right* as value.

```
while (input) {
        word2 = word1;
        input >> word1;
        if (word1 == "=") {
                input >> tmp;
                iniFileMap.insert(make_pair(word2, tmp));
        }
}
```

We have tree structure of calculation parameters.

Constants – **calculation parameters structure**
- *double nDays* – **number of days**
- *bool useRadialDifusion, useAlphaDifusion, useEnergyDifusion* – **witch diffusions**
- *DLLTypes DLLType* – **DLL type**
- *Matrix1D<double> Bf, Kp* – **arrays for boundary flux and Kp**
- *Matrix1D<double> Lpp, dBw2* – **arrays for Lpp and dBw^2**
- *string fileKp, fileBf* – **names of the files to load Kp and Bf**
- *double constKp, constBf* – **constant values for Kp and Bf, if not loaded from files**
- *bool useLpp, useBf, useBw2* – **use all this staff or not**
- *int totalIterationsNumber* – **total number of iterations**
- *double timeStep* – **time step (dt)**
- *double tau, tauLpp* – **tau (lifetime) inside and outside the plaspasphere**

PSD – **Phase Space Density**

Initial_PSD – **Initial PSD parameters**

InitialPSDTypes initialPSDType – **initial PSD type**

string fileName – **initial PSD file name**

double tauSteadyState – **tau for steady state**

double Kp0 – **Kp for steady state**

double value_psd – **minimal value of PSD**

Output – **output PSD parameters**

string folderName – **folder name**

string fileName4D – **file name**

double timeStep – **time step to output**

Grid – **grid for radial diffusion**

GridTypes type – **grid type**

GridElement – **grid element L**

string name - **name**

int size - **size**

bool useLogScale – **log scale or not log scale**

double min, max – **min and max values**

BoundaryCondition – **boundary condition**

BoundaryConditionTypes type – **type**

double constantValue – **constant value for bc (if used)**

ApproximationMethods approximationMethod – **approxim. method (for block method of solution)**

SolutionMethods solutionMethod – **solution method (for block method of solution)**

Output – **general output parameters (output for 1d variables like Kp etc)**

string logFileName – **log file name**

string folderName – **folder name**

string fileName1D – **file name for 1d variables**

double timeStep – **time step to output**

Interpolation - **interpolation**

InterpolationTypes type – **interpolation type**

bool useLog – **interpolate log of function or not**

double linearSplineCoef – **spline interpolation parameter**

double maxSecondDerivative – **spline interpolation parameter**

DiffusionCoefficientConstants – **diffusion coefficient parameters**

string filename – **file name to load or save data**

InputFileTypes filetype – **file type**

LoadOrCalculateChoise loadOrCalculate – **load or calculate flag**

string DxxName – **name**

DiffusionCoefficientTypes DxxType – **type (Daa, Dpcpc etc)**

string waveName – **wave name (chorus, hiss etc)**

NumberDencities numberDencity – **number density model**

double MLT_averaging – **MLS averaging value**

Omega_mTypes Omega_mType – **Omega_m type (part of Omega_e or absolute value)**

double Omega_m – **Omega_m**

double d_omega – **d_omega**

double omega_uc – **omega lower cut off**

double omega_lc – **omega upper cut off**

double eta1, eta2, eta3 – **calculation parameters**

double Bw – **Bw**

bool BwFromLambda – **if we need calculate Bw depends of Lambda**

int nint – **numbert of points in integral**

double lam_max – **max lambde**
double nu – **nu**
double s – **s**
double f – **f**
ParticleTypes particle – **type of something...**
double time_start – **time for diffusion coefficient to start**
double time_end – **time for diffusion coefficient to end**

Fill each structure we run it's own *' ReadStructureFromMap'* to fill its parameters from multimap.

```
constants.ReadStructureFromMap(iniFileMap);
```

We can use structure name as an argument, it will be prefix for all variables name in the parameter (.ini) file

```
output.ReadStructureFromMap(iniFileMap, "output.");
...

bool Constants::Output::ReadStructureFromMap(multimap <string, string> iniFileMap, string structureName)
{
	ReadFromMap(timeStep, iniFileMap, structureName + "outputEach");
```

Find each parameters structure variable inside multimap by key-name

```
ReadFromMap(nDays, iniFileMap, "nDays");
```

If we have default value for variable and/or variable is optional, we can call overloaded version of *'ReadFromMap'*. In next example *'string("logfile.log")'* is the default value.

```
ReadFromMap(logFileName, iniFileMap, structureName + "logFileName", string("logfile.log"));
```

If we passed default value into *'ReadFromMap'* function it search trough the parameters multimap for value and use default value if nothing found. If no default value passed then function call other overload version and pass *'mandatory = true'* with any default value (but with correct variable type) as a parameters

```
template <typename T>
void ReadFromMap(T &place, multimap <string, string> iniFileMap, string varTextName) {
	ReadFromMap(place, iniFileMap, varTextName, *(new T), true);
}
```

When we found key we convert corresponding text-value into variable-type value. *'StrToVal'* is overloaded function for each type of variables.

```
StrToVal(it->second, place);
```

At the end of initialization we load or calculate one dimensional variables like Kp and Bf from files interpolate and store them inside arrays.

```
Kp.AllocateMemory(totalIterationsNumber);
load_1d(Kp, fileKp, timeStep, constKp);
```

Initializing Phase Space Densities and Grids.

We have two grids (L, pc, alpha): one for radial diffusion with constant first and second adiabatic invariants along L-lines and second with perpendicular pc-alpha grid for each L. We make two instances of class PSD for each grid: '*psdRadialDiffusion*' and '*psdLocalDiffusions*'. Each instance

conclude its corresponding grid.

```
PSD psdRadialDiffusion(constants.psdRadialDiffusion);
PSD psdLocalDiffusions(constants.psdLocalDiffusions, psdRadialDiffusion, constants.interpolation);
```

We send corresponding parameters structures *'constants.psdRadialDiffusion'* and *' constants.psdLocalDiffusions'* to the constructors. To the second constructor we also send *'psdRadialDiffusion'* and interpolation parameters *'constants.interpolation' because initial 'psdLocalDiffusions'* will be create by interpolation from radial grid.

First line of constructor get PSD parameters as an input argument and run constructors of parent for PSD class Matrix3D and inhere class Grid with sending them related branches of parameters structure *'constants.grid'* and *'constants.gird.L.size', 'constants.gird.pc.size', 'constants.gird.alpha.size'*.

```
PSD::PSD(Constants::PSD constants) : grid(constants.grid), Matrix3D<double>(constants.grid.L.size,
constants.grid.pc.size, constants.grid.alpha.size)
```

Grid

In case of 3D-diffusion with two grids and interpolation between them, first we make radial grid with constant first and second adiabatic invariants along *L*-lines. We create regular grid for L and perpendicular by alpha and *pc* grid form *L=L_max*.

```
for (il = 0; il < L.size ; il++) {
        for (im = 0; im < pc.size; im++) {
                for (ia = 0; ia < alpha.size; ia++) {
                        L.SetRegularGridValue(il, im, ia, il);
                }
        }
}
il = L.size-1;
for (im = 0; im < pc.size; im++){
        for (ia = 0; ia < alpha.size; ia++) {
                pc.SetRegularGridValue(il, im, ia, im);
                alpha.SetRegularGridValue(il, im, ia, ia);
        }
}
```

Function *'SetRegularGridValue'* create regular grid value depends if we need to use logarithmic scale or not and depends 1D/2D/3D diffusion.

```
void GridElement::SetRegularGridValue(int iteratorL, int iteratorPc, int iteratorAlpha, int gridElementDirection)
{
        if (size > 1)
                if (useLogScale) (*this)[iteratorL][iteratorPc][iteratorAlpha]
                        = pow(10, log10(min) + gridElementDirection*(log10(max) - log10(min))/(size - 1));
                else (*this)[iteratorL][iteratorPc][iteratorAlpha]
                        = min + gridElementDirection*(max - min)/(size - 1);
        else (*this)[iteratorL][iteratorPc][iteratorAlpha] = max;
}
```

Then we calculate invariants on the top level of L and from that values calculate *pc* and alpha for each L.

```
mu = VF::mu_calc(L[L.size-1][im][ia], pc[L.size-1][im][ia], alpha[L.size-1][im][ia]);
Jc = VF::Jc_calc(L[L.size-1][im][ia], pc[L.size-1][im][ia], alpha[L.size-1][im][ia]);
for (il = 0; il < L.size-1 ; il++) {
        alpha[il][im][ia] = find_alpha(Jc * sqrt(L[il][im][ia]) / sqrt(8.0 * VF::B(1) * VC::mc2 * mu),
alpha.min, VC::pi/2);
        pc[il][im][ia] = sqrt(2.0 * mu * VC::mc2 * VF::B(L[il][im][ia])) / sin(alpha[il][im][ia]);
```

We need to make local grid. In radial grid *alpha* lines are parallel for each *L*. So, for each *L* we take *alpha* and create new *pc* grid, that is perpendicular to *alpha*. We take max and min pc values for each L and make regular grid between that values.

```
pc_local_max = SecondGrid.pc[il] [SecondGrid.pc.size-1] [alpha.size-1];
pc_local_min = SecondGrid.pc[il] [0] [alpha.size-1];
for (im = 0; im < pc.size; im++){
        for (ia = 0; ia < alpha.size; ia++){
                if (pc.size > 1) {
                        if (!pc.useLogScale) pc[il] [im] [ia] = pc_local_min + im*(pc_local_max - pc_local_min)/(pc.size - 1);
                        else pc[il] [im] [ia] = pow(10, log10(pc_local_min) + im*(log10(pc_local_max) - log10(pc_local_min))/(pc.size - 1));
                } else pc[il] [im] [ia] = pc.max;
```

We got two grids differ only by *pc*. That means we can use series of one dimensional diffusions to interpolate from one grid to another. And so we do.

Initial values of PSD

For 3D-case we create PSD on radial grid by solving steady state problem. Than we interpolate it to local grid.

Boundary conditions

For each grid direction we have two boundary conditions. It is convenient to keep boundary conditions in grid elements. Each boundary condition has a parent class Matrix2D for holding values at the boundary and type. Additionally it has initialization type and value.

class BoundaryCondition : public Matrix2D<double> **- class and parent class**
BoundaryConditionTypes calculationType **– calculation type**
BoundaryConditionTypes initialType **– initialization type**
double constantValue **– initialization value**

Boundary condition is a function on the boundary *'function(PSD) = boundary value'*. Calculation type determine the *'function'*, determine if the boundary condition is on PSD or PSD derivative. Initial type determine calculation type and the place to take *'boundary value'*: from initialization value or from initial PSD.

Output

We initialize output for PSD. It is a class that hold file name and other staff for output.

```
outputPSD.Init(this, constants.output.folderName + constants.output.fileName4D);
```

Diffusion coefficients

Waves produce diffusion and can be associated with diffusion coefficients. All diffusion coefficients differ into tree groups by diffusion direction (type): DLL, Dpcpc and Daa. Here we just introduced two new concepts (classes): *'DiffusionCoefficnent'* and *'DiffusionCoefficientGroup'*.

```
class DiffusionCoefficient : public Matrix3D<double>
class DiffusionCoefficientsGroup : public DiffusionCoefficient
```

The *'DiffusionCoefficient'* is a child of *'Matri3D'* and hold coefficients at each point of the grid. *'DiffusionCoefficientsGroup'* is a child of *'DiffusionCoefficient'*. It equal to the sum of the diffusion coefficients with the same type and hold a list of all diffusion coefficient with this type.

```
vector <DiffusionCoefficient> DxxList;
```

Some waves and corresponding diffusion coefficients can affect not always but only some period of time. Function *'Activate(time)'* of class *'DiffusionCoefficientGroup'* actualize it by sum all diffusion coefficients affecting at this time.

```
bool DiffusionCoefficientsGroup::Activate(double time) {
	unsigned int Dxx_it;
	for (Dxx_it = 0; Dxx_it < DxxList.size(); Dxx_it++) {
		if ((time >= DxxList[Dxx_it].time_start && time < DxxList[Dxx_it].time_end
			&& !DxxList[Dxx_it].is_active)
		|| (!(time >= DxxList[Dxx_it].time_start && time < DxxList[Dxx_it].time_end)
			&& DxxList[Dxx_it].is_active)) {
			(*this) = 0;
			for (Dxx_it = 0; Dxx_it < DxxList.size(); Dxx_it++) {
				if (time >= DxxList[Dxx_it].time_start && time < DxxList[Dxx_it].time_end) {
					(*this) = (*this) + DxxList[Dxx_it];
```

For convenient we also divided all diffusion coefficients groups by different groups for diffusion inside and outside of the plasmasphere: *'Dpcpc', 'DpcpcLpp', 'Daa', 'DaaLpp'*.

Before using diffusion coefficients we need to load or calculate them. Function *'Get'* of the class *'DiffusionCoefficient'* do this. Depends of parameter *'loadOrCalculate'* it calculate, load or try to load first and calculate in case of false.

```
	if (DxxConstants.loadOrCalculate == LOC_LOAD || DxxConstants.loadOrCalculate ==
LOC_LOADORCALCULATE) {
		try {
			// try to load
			LoadDiffusionCoefficient(grid.L, grid.epc, grid.alpha, DxxConstants.filename.c_str(),
DxxConstants.filetype);
		} catch (error_msg err) {
			if (DxxConstants.loadOrCalculate == LOC_LOADORCALCULATE) {
				Output::echo("%s", err.what().c_str());
				Calculate(grid.L, grid.epc, grid.alpha, DxxConstants);
				...
		}
	} else if (DxxConstants.loadOrCalculate == LOC_CALCULATE) {
		Calculate(grid.L, grid.epc, grid.alpha, DxxConstants);
		...
	}
```

Main loop

We calculate diffusion on the next time step inside the loop. In general, it is three diffusion:

radial, energy and pitch-angle. Radial diffusion goes on the “radial” grid with constant first and second adiabatic invariants along the L-lines, and energy and pitch-angle diffusions go on “local” grid, perpendicular by pitch-angle and energy for each L.

First we interpolate PSD to the “radial” grid.

```
psdRadialDiffusion.Interpolate(psdLocalDiffusions, constants.interpolation);
```

There are several methods of interpolation: linear, polynomial and spline. In polynomial and spline methods we pass boundary values for more accurate interpolation. In spline method are two more parameters: *linearSplineCoef*, *maxSecondDerivative*. *LinearSplineCoef* make splines method more linear (0 - pure spline, 1 - pure linear) and *maxSecondDerivative* define maximum second derivative of the function to do not use *linearSplineCoef* (look numerical recepies for more information).

Next step we calculate the DLL for current Kp index.

```
DLL.MakeDLL(psdRadialDiffusion.grid.L, psdRadialDiffusion.grid.pc, psdRadialDiffusion.grid.alpha,
constants.Kp[iteration], constants.DLLType);
```

We run radial diffusion.

```
psdRadialDiffusion.DiffusionL(constants.timeStep,
        constants.Lpp[iteration],  DLL,
        psdRadialDiffusion.grid.L.lowerBoundaryCondition,
        psdRadialDiffusion.grid.L.upperBoundaryCondition * constants.Bf[iteration],
        constants.tau, constants.tauLpp);
```

Inside *'DiffusionL'* function we copy from 3-dimensional arrays to 1-dimensional for each pc and alpha and run *'MakeMatrix'* function.

```
for (im = 0; im < pc.size; im++) {
    for (ia = 0; ia < alpha.size; ia++) {
        for (il = 0; il < L.size; il++) {
            grid1D[il] = L[il][im][ia];
            Dxx1D[il] = DLL[il][im][ia];
        }
            MakeMatrix(&L_MatrixA[im][ia].A, - diagonals
                &L_MatrixA[im][ia].B, - diagonals
                &L_MatrixA[im][ia].C, - diagonals
                &L_MatrixA[im][ia].R, - right hand side
                &grid1D, - grid
                tau, - life time
                power_of_x - power of x
                lowerBoundaryCondition[im][ia], - Boundary conditions
                upperBoundaryCondition[im][ia],
                L.size,
                dt,
                &Dxx1D,
                tauLpp,
                Lpp,
                0, //g_flag
                lowerBoundaryCondition.calculationType,
```

```
                                upperBoundaryCondition.calculationType,
                                approximationMethod);
                }
        }
```

Then we run *'SolveMatrix'* and solve it by tridiagonal method.

```
SolveMatrix(&slicePSD1D,
                &L_MatrixA[im][ia].A,
                &L_MatrixA[im][ia].B,
                &L_MatrixA[im][ia].C,
                &L_MatrixA[im][ia].R,
                lowerBoundaryCondition[im][ia], - Boundary conditions
                upperBoundaryCondition[im][ia],
                L.size,
                dt);
...
```

```
tridag(A, B1, C, R1, f, nx);
```

After the end of radial diffusion we interpolate back to the “local” grid.

```
psdLocalDiffusions.Interpolate(psdRadialDiffusion, constants.interpolation);
```

Before energy and pitch-angle diffusion calculation we actualize diffusion coefficients.

```
Dpcpc.Activate(time);
DpcpcLpp.Activate(time);
Daa.Activate(time);
DaaLpp.Activate(time);
```

There are to methods to solve energy and pitch-angle diffusion: Block and Split methods. In block method we solve both diffusions together and in Split method we solve one then another. With small enough time step both methods give the same result.

```
        switch (constants.psdLocalDiffusions.approximationMethod) {
                case AM_Block_C: - block method with central derivatives approximation
                case AM_Block_LR: - block method with left-right der. approximation
                        ...
                        break;
                case AM_Split_C: - split method with central derivatives approximation
                case AM_Split_LR: - split method with left-right der. approx.
                        ...
                        break;
```

Split method

Split method is method is exactly the same as L-diffusion, except one moment. We use Daa or DaaLpp (Dpcpc or DpcpcLpp) depends on L and Lpp.

```
if (L[il][im][ia] >= Lpp) {
        Dxx1D[im] = Dpcpc[il][im][ia];
} else {
```

```
            Dxx1D[im] = DpcpcLpp[il][im][ia];
        }
```

Block method

In general algorithm is the same: we make ModelMatrix of the equation AX=B then solve it.

Output

Output generates into files. All output files has format. First line hold list of all variables present in file. The follow part divided into one or more “zones” contains different data sets, for example different time slices. 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.

```
        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
        ...
```

With specified time step program add next time slice of PSD to the output file.

```
if (check_time(time, constants.output.timeStep)) {
            Output::echo("Writing data files.\n");
            // write information about 1d wariables into the file with 1d variables
            output1D << time << "\t"
                        << constants.Kp[iteration] << "\t"
                        << constants.Bf[iteration] << "\t"
                        << constants.Lpp[iteration] << "\t"
                        << constants.dBw2[iteration] << "\t" << endl;
            psdLocalDiffusions.Output(time);
            psdRadialDiffusion.Output(time);
        }
```

Output function of PSD class make all PSD outputs (if there are more then one).

```
void PSD::Output(double time, GridElement &L, GridElement &epc, GridElement &alpha) {
        outputPSD.out(time);
        output4D.out(time, L, epc, alpha);
}
```

Currently useful is *'OutputPSD'* that write file with only PSD variable for different time zones.

```
void PSD::OutputPSD::out(double time) {
        if (this->useOutput) {
            int il, im, ia;
            (*output) << "ZONE T=\"" << time
                        << "\" I=" << psd->size_z
```

```
                                        << ", J=" << psd->size_y
                                        << ", K=" << psd->size_x << endl;
                        for (il = 0; il < psd->size_x; il++) {
                                for (im = 0; im < psd->size_y; im++) {
                                        for (ia = 0; ia < psd->size_z; ia++) {
                                                (*output) << (*psd)[il][im][ia] << endl;
                                        }
                                }
                        }
                }
}
```