Constants

Constants.h

Describe structures for parameters, loaded from file and related functions.

structures:

DiffusionCoefficientConstants – diffusion coefficients parameters

Constants - general parameters

Constants::BoundaryCondition – boundary parameters

Constants::GridElement – grid element parameters

Constants::Grid – grid parameters

Constants::Output – output parameters

Constants::PSD – phase space density parameters

Constants::PSD::Initial_PSD – initial values of phase space density parameters

Constants::PSD::Output – phase space density output parameters

Constants::Interpolation – interpolation parameters

Functions:

Function Load_constants()

Function open file and run FileToMap.

bool Load_constants(Constants &constants, string filename);

Arguments:

Constants &constants – parameters structure.

string filename – file name.

Function FileToMap()

Function load parameters from file to map.

bool FileToMap(multimap <string, string> &iniFileMap, string filename);

Arguments:

multimap <string, string> &iniFileMap - multimap to load parameters in.

string filename – file name.

Function ReadFromMap()

Function read value from map into variable.

void ReadFromMap(T &place, multimap <string, string> iniFileMap, string varTextName)

void ReadFromMap(T &place, multimap <string, string> iniFileMap, string varTextName, T defValue, bool mandatory = false)

where 'T' is template typename.

Arguments:

T &place – variable to load value in.

multimap <string, string> iniFileMap – map file with loaded values of parameters.

string varTextName – name of the variable that we are loading from the map.

T defValue – default value of the variable, if exists.

bool mandatory = false – flag that show if the value should be in the map anyway or not.

The version of function to load diffusion coefficients parameters:

void ReadFromMap(DiffusionCoefficientConstantsList &place, multimap <string, string> iniFileMap, string varTextName, bool mandatory = false, DiffusionCoefficientConstantsList defValue = DiffusionCoefficientConstantsList());

Function StrToVal

Convert text to value (to the different types)

void StrToVal(string input, double &place);

void StrToVal(string input, int &place);

void StrToVal(string input, string &place);

void StrToVal(string input, bool &place);

void StrToVal(string input, InterpolationTypes &place);

void StrToVal(string input, NumberDencities &place);

void StrToVal(string input, Omega_mTypes &place);

void StrToVal(string input, WaveTypes &place);

void StrToVal(string input, DiffusionCoefficientTypes &place);

void StrToVal(string input, GridTypes &place);

void StrToVal(string input, BoundaryConditionTypes &place);

void StrToVal(string input, LoadOrCalculateChoise &place);

void StrToVal(string input, InputFileTypes &place);

void StrToVal(string input, InitialPSDTypes &place);

void StrToVal(string input, ParticleTypes &place);

void StrToVal(string input, DLLTypes &place);

void StrToVal(string input, ApproximationMethods &place);

void StrToVal(string input, SolutionMethods &place);

The version of function that get address of variable and type as input arguments.

bool StrToVal(string input, unsigned int address, AllTypes type);

Arguments:

string input – input text with value (like '123.45');

<type> &place – variable to store value in.

Function load_1d

Load one dimensional arrays from files with structure: <time> <value>. Interpolate (linear) to each time step.

void load_1d(Matrix1D<double> &var, string filename, double dt, double def_val, int var_size = 0);

Arguments:

Matrix1D<double> &var – one dimensional array.

string filename – file name.

double dt – time step to interpolate.

double def_val – default value.

int var_size = 0 – one dimensional array size. Zero means array already initialized (memory reserved).

Functions str2bool() and bool2str()

Convert from string to bool and from bool to string.

bool str2bool(string str);

string bool2str(bool b);

Constants.cpp

Body of functions from constants.h

Diffusion

1d_universal_solver.h

Solve one dimensional diffusion step.

Functions:

Function 1d_universal_solver()

Solve one dimensional diffusion step.

*bool fnpl1_nug(double *f, double *x, double tau, double n, double f_lower, double f_upper, int nx, double dt, double *Dxx, double taulc, double alc, int g_flag, int lower_border_condition_type, int upper_border_condition_type);*

Arguments:

*double *f* - phase space density function

*double *x* - one dimensional grid

double tau - lifetime

double n - power of x

double f_lower - lower boundary conditions

double f_upper - upper boundary conditions

int nx - number of grid points

double dt - time step

*double *Dxx* - diffusion coefficient

double taulc - lifetime in loss cone

double alc - loss cone size

int g_flag - diffusion flag

int lower_border_condition_type - type of lower boundary conditions: 0 - for value, 1 - for derivative

int upper_border_condition_type - type of upper boundary conditions: 0 - for value, 1 - for derivative

Function tridiag()

Solving equation AX=B from tridiagonal A matrix.

bool tridag(double a[], double b[], double c[], double r[], double u[], long n);

Arguments:

double a[] - first diagonal

double b[] - second diagonal

double c[] - third diagonal

double r[] - right hand side

double u[] - result

long n - size

1d_universal_solver.cpp

Body for functions from 1d_universal_solver.h

MatrixSolver.h

Make model matrix and solve equation AX=B.

Type definition:

typedef map <int, Matrix1D<double>> DiagMatrix;

Structure:

struct Tridiag – structure

```
bool MakeMatrix(double *A,
                double *B1,
                double *C,
                double *R,
          double    *x,                    // one
dimentional grid
          double    tau,                   // lifetime
          double    n,                     // power of x
          double    f_lower,               // lower boundary
conditions
          double    f_upper,               // upper boundary
conditions
          int       nx,                    // number of
grid points
          double    dt,                    // time step
```

```
                    double      *Dxx,                                   // diffusion coefficient
                    double      taulc,                                  // lifetime in loss cone
                    double      alc,                                    // loss cone size
                    int         g_flag,                                 // diffusion flag
                    int         lower_border_condition_type,// type of lower boundary
conditions: 0 - for value, 1 - for derivative
                    int         upper_border_condition_type,// type of upper boundary
conditions: 0 - for value, 1 - for derivative
                    ApproximationMethods approximationMethod = AM_Split_C);


// Solver for 1d general equation
bool SolveMatrix(double     *f,                                     // phase space density
function
                        double *A,
                        double *B1,
                        double *C,
                        double *R,
                        double      f_lower,                        // lower
boundary conditions
                        double      f_upper,                        // upper
boundary conditions
                        int     nx,                                 // number of
grid points
                        double      dt);                                    // time
step



void MakeModelMatrix_pc_alpha(Matrix1D<DiagMatrix> &diagMatrix,
Matrix1D<Matrix1D<double>> &rightHandSide,
                        GridElement &L, GridElement &pc, GridElement &alpha,
                        DiffusionCoefficient &Dpcpc, DiffusionCoefficient &DpcpcLpp,
                        DiffusionCoefficient &Daa, DiffusionCoefficient &DaaLpp,
                        Matrix3D<double> &Jacobian, double dt, double Lpp);


void SolveMatrix2D(Matrix1D<DiagMatrix> &pc_alpha_ModelMatrixA,
Matrix1D<Matrix1D<double>> &rightHandSide,
                         Matrix3D<double> &PSD,
```

```
                                GridElement &L, GridElement &pc, GridElement &alpha);


/*void SecondDerivativeApproximation(DiagMatrix diagMatrix, int lineNumber,
                                                int index_pc, int index_alpha,
                                                DerivativeTypes FirstDerivative,
DerivativeTypes SecondDerivative,
                                                GridElement &pc, GridElement &alpha,
                                                Matrix2D<double> &Dxx);*/

void SecondDerivativeApproximation(DiagMatrix &diagMatrix,
                                                int index_L, int index_pc, int index_alpha,
                                                DerivativeTypes FirstDerivative,
DerivativeTypes SecondDerivative,
                                                GridElement &L, GridElement &pc,
GridElement &alpha,
                                                DiffusionCoefficient &Dxx,
Matrix3D<double> &Jacobian,
                                                double multiplicator,
                                                ModelMatrixSizes M_size);

int getLineNumber2D(int ia, int im, int ia_size);
int getDiagonalNumber2D(int matrixLine, int ia, int im, int ia_size);

void SecondDerivativeApproximation(DiagMatrix &diagMatrix,
                                                int il, int im, int ia,
                                                        int dL1, int dPc1, int dAlpha1,
                                                        int dL2, int dPc2, int dAlpha2,
                                                GridElement &L, GridElement &pc,
GridElement &alpha,
                                                DiffusionCoefficient &Dxx,
Matrix3D<double> &Jacobian,
                                                double multiplicator,
                                                ModelMatrixSizes M_size);

void over_relaxation_diag(DiagMatrix &A, Matrix1D<double> &B, Matrix1D<double> &X);
```

```
void lin_solv(double *a, double *b, double *x, int n);


void SolveMatrix2D_gauss(Matrix1D<DiagMatrix> &pc_alpha_ModelMatrixA,
Matrix1D<Matrix1D<double>> &rightHandSide,
                                    Matrix3D<double> &PSD,
                                    GridElement &L, GridElement &pc, GridElement &alpha);
```

MatrixSolver.cpp

PSD.h

PSD.cpp

DiffusionCoefficient

bisection.h

bisection.cpp

rrouts.h

rrouts.cpp

diffusionCoefficient.h

diffusionCoefficient.cpp

Grid

grid.h

grid.cpp

Main

Main.cpp

Matrix

matrix.h

Interpolation

linear.h

polilinear.cpp

polilinear.h

polint.cpp

polint.h

ratint.cpp

ratint.h

spline.cpp

spline.h

Output

Output.h

Output.cpp

Various Functions

erf.cpp

erf.h

error.h

error.h.res

types.h

variousConstants.h

variousFunctions.cpp

variousFunctions.h