
CODE2 Documentation

Release 1.0

Albert Johansson

May 27, 2020

Contents

1	Classes	3
1.1	Sources	3
1.2	Operators	4
1.3	Implicit Operators	9
1.4	Explicit Operators	14
1.5	Quantities	16
1.6	Simulation	28
1.7	Simulation output	31
2	Help functions	35
2.1	cs = CalculateMollerCS(g,g1)	35
2.2	moment = CalculateRunawayMomentXiDependent(Nxi,Ny,f,xiIntMat,weightsMat)	35
2.3	sigma = CalculateSigmaIntegral(gamma,gamma_m)	36
2.4	[m,xiGrid] = GenerateCumIntMat(Nxi,nPoints)	36
2.5	f = interpolateDistribution(dist,MG)	36
2.6	outPls = LegendrePolynomials(l,x)	36
2.7	[x,w] = lgwt(N,a,b)	37
2.8	[x, w, D, DD] = m20121125_04_DifferentiationMatricesForUniformGrid(N, xMin, xMax, scheme)	37
2.9	m = MapToXiIntMat(cumintMat,xiGrid,EOverEc,y2,deltaRef)	38
2.10	[posF,negF] = SumLegModesAtXi1(f,Ny,Nxi)	38
2.11	i = CalculateCurrentAlongB(f,y,yWeights, Ny, nRef ,deltaRef)	38
2.12	n = CalculateDensity(f,Ny,Nxi,y,yWeights,nRef)	39
2.13	delta = getDeltaFromT(T)	39
2.14	[delta,lnLambda,collFreq,BHat] = getDerivedParameters(T,density,B)	39
2.15	[EOverEC,EOverED,EHat] = getNormalizedEFields(E,T,density)	39
2.16	sigma = GetSpitzerConductivity(T,n,Z)	40
3	Code internal structure	41
3.1	Code description	41
4	Indices and tables	43
	Index	45

Code documentation, its uses and internal structure

Below follows the different classes in CODE.

1.1 Sources

1.1.1 Sources

class Source

Abstract class defining a avalanche Source. Can for example be the avalanche source Rosenbluth and Putvinskii derived.

Properties

state

State object for knowing the physical params use for knowing the physical params usedd

eqSettings

EquationSettings object used for knowing what settings is used

sourceVector

sourceVector - vector with the source, calculated each step with getSourceVec.

Functions

getSourceVec (*this, f, iteration*)

Returns the source vector at iteration from f. Should recalc and update *sourceVector*.

1.1.2 RosenbluthSource

Extends *Source*. Describes and calculates the source vectors for Rosenbluth-Putvinski avalanche source.

class RosenbluthSource

Class describing a Rosenbluth Putvinskii source. Is *sourceMode* 1 and 2 in *EquationSettings*.

Properties (non-inherited)

tail_mask

Mask describing the tail of of the runaways. It is used in *sourceMode* 2 from *EquationSettings*.

nr_mask

Mask defining the runaways, i.e. when an electron is considered a runaway. This is done so all with energy bigger twice the critical energy is considered to be a runaway (if not *wMinForSource* is set by user).

Functions

getSourceVec (*this*, *f*, *iteration*)

Returns the source vector for this iteration and also updates the *sourceVector*.

DetermineFastParticleRegion ()

Internal function whihc determines the tail mask.

1.1.3 ChiuHarveySource

class ChiuHarveySource

extends *Source*. Class defining a Chiu Harvey source.

Functions

getSourceVec (*this*, *iteration*)

returns the source vector

1.2 Operators

1.2.1 Operator

(Abstract) Operator < handle & matlab.mixin.Copyable

Abstract superclass for operators. Both implicit and explicit.

Properties

state

eqSettings

operatorMatrix

Sparse properties

used for sparse matrix building in CODE, not required for implementation of the class, usage should be investigated in implementations of the class

predictedNNZ

sparseCreatorIndex

estimated_nnz

sparseCreator_i

sparseCreator_j

sparseCreator_s

Functions

this = Operator(state, eqSettings, varargin)

Construct a new instance of this class.

matrixHasChanged = generateOperatorMatrix(this,runIndex, inputArg)

generateOperatorMatrix implementation should modify operatorMatrix to match the parameters at time index runIndex in state object

M = getMatrix(this)

The functions below are all utilities for building sparse matrices:

resetSparseCreator (this)

Resets the sparse values.

clearSparseResidue (this)

Reduce the memory footprint by removing the vectors used for building the sparse matrix once they are no longer needed

addToSparse (this, i, j, s)

Adds values to the sparse matrix s at index (i,j) to matrix

addSparseBlock (this, rowIndices, colIndices, block)

Adds a block to the sparse matrix. rowIndices and colIndices should be vectors. numel(rowIndices) should equal the number of rows in 'block'. numel(colIndices) should equal the number of columns in 'block'.

sparseMatrix = createSparse(this)

After you are done adding elements to the sparse matrix using addToSparse() and addSparseBlock(), call this function to finalize the matrix.

1.2.2 EquationSettings

class EquationSettings

operators

Operators, in a cell array, necessary to fulfill the settings specified below.

useScreening

take screening into account

0. Ignore screening (complete screening, correct at low energy)
1. Fokker–Planck collision operator, TF model (works for all species)
2. Fokker–Planck collision operator, TF-DFT model (works for some ionization degrees of argon and Xe)
3. Fokker–Planck collision operator, full DFT model (only works for Ar+ so far)
4. Boltzmann collision operator, TF model
5. Boltzmann collision operator, TF-DFT model
6. Boltzmann collision operator, full DFT model
7. Full penetration (no screening, $Z0^2 \rightarrow Z^2$, correct in the limit $p \rightarrow \infty$)

useInelasticTake

inelastic collisions with bound electrons into account

0. Ignore inelastic collisions
1. Bethe formula
2. RP

useEnergyDependentLnLambdaScreening

Logarithmic enhancement of the Coulomb logarithm

0. No (standard formula)
1. Yes (change in the collision operator). Interpolated Landau form.
2. Enhance by an ad-hoc factor of 1.3.

Collision operators and conservational properties

collisionOperator

specifies the type of collision operator to use

0. Fully relativistic test-particle collision operator. NOT momentum-conserving. Relativistic Fokker-Planck coefficients from OJ Pike & SJ Rose, Phys Rev E 89 (2014).
1. Non-relativistic, linearized, momentum-conserving. Includes modified Rosenbluth potentials.
2. Use the non-relativistic field particle term from 1, together with the relativistic test particle term from 4. Generally works better than 1, as the tail quickly becomes relativistic. The field-particle term only affects the bulk, so the non-relativistic approximation is usually good there.
3. Use an ad-hoc relativistic generalization of the field-particle term from 1, together with the test-particle terms from 4. Generally appears to work better than 1 & 2, although the physics basis is shaky. !!!Experimental!!!

4. Relativistic collision operator for non-relativistic thermal speeds. Agrees with 0 when $T \ll 10$ keV. See the CODE-paper for details.

useNonRelativisticCollop

use the non-relativistic limit of the collision operator, for instance for benchmarking hot-tail generation against Smith et al. (2005), or Smith and Verwichte (2008).

Avalanche (secondary runaway) source term**sourceMode**

the type of avalanche source to use

0. do not include a source
1. include a Rosenbluth-Putvinskii-like source
2. the same source, but using the number of “fast” particles, rather than the number of runaways, to determine the source magnitude. Useful in runaway decay and/or when $E < E_c$, (when there are no runaways by definition). **DISCLAIMER:** This is not necessarily consistent, and is sensitive to the definition of a fast particle!
3. include a Chiu-Harvey-like source
4. include a Chiu-Harvey-like source which takes the pitch-dependence of the distribution into account (derived by Ola). Should be more correct than 3.
5. The same as 4, but including a sink term to conserve particle energy and momentum.

fastParticleDefinition

relevant when sourceMode = 2

0. Do not calculate the fast particle content, is not compatible with sourceMode 2
1. Based on where the “tail” “begins” (where $f/f_M >$ some threshold) Note that the shape of the source in momentum space is only recalculated if the electric field changes, and may at times not be consistent with the beginning of the tail.
2. Based on relative speed ($v/v_e >$ some threshold)
3. Based on absolute speed ($v/c >$ some threshold)
4. Selects the most generous of cases 2,3 and the critical speed for runaways. Useful when you want to follow n_r closely, but still have fast particle when E is close to (or below) E_c .

tailThreshold

for fastParticleDefinition = 1

relativeSpeedThreshold

for fastParticleDefinition = 2

absoluteSpeedThreshold

for fastParticleDefinition = 3

yCutSource

Arbitrary cutoff needed for source 5. Set to empty to use $y_{\text{cutSource}} = y_{\text{crit}}$ (currently only works with constant-plasma parameters) Can also be used for source 3 and 4 (cf. sourceMode 1 vs 2)

runawayRegionMode

Determines how to define the runaway region and how to calculate moments of f in that region. Is interpreted as 0 no matter what in sourceMode 2.

0. Isotropic ($y_c = y_c(x_i=1)$ for all x_i) – default
1. x_i -dependent y_c – more correct. Most relevant for hot-tail scenarios

nPointsXiInt

resolution parameter for runawayRegionMode = 1

bremsMode

include an operator for bremsstrahlung radiation reaction

0. No bremsstrahlung losses
1. Simple, continuous slowing-down force (Bakhtiari, PRL, 2005)
2. Boltzmann op., full (both low and high k , angular-dependent cross section)
3. Boltzmann op., low and high k , no angular dependence
4. Boltzmann op., only high k , no angular dependence

1.2.3 DiagonalPlusBoundaryCondition

class DiagonalPlusBoundaryCondition

Builds the diagonal and boundary condition matrix in CODE version 1.03 CODEtimeDependent(). Basically does the BC and diag matrix. Extends ImplicitOperators but is not really a implicit operator. Should maybe be given a new class.

Properties

These properties are saved to determine if a new matrix is needed or not in next timestep.

NxiUsed

ddyUsed

RelLimitUsed

yMaxBC

Functions

DiagonalPlusBoundaryCondition (*state, eqSettings*)

Creates a new instance of this class

matrixHasChanged = **generateOperatorMatrix**(**this**, **iteration**)

Generates the diagonal and boundary condition from time derivative as if it was a ImplicitOperator.

1.3 Implicit Operators

1.3.1 ImplicitOperator

class ImplicitOperator

This is an abstract class handling implicit operators. These can ofcourse be used as explicit operators by adding a minus sign to the matrix after generation of the matrix.

Properties

state

State object for creating the matrices

eqSettings

EquationSettings object, containing what settings to use for the run.

operatorMatrix

the operatorMatrix. Is calculated with the function call *generateOperatorMatrix()*.

Sparse properties:

used for sparse matrix building in several implimations of this class. It is not required for implementation though.

predictedNNZ 0;

sparseCreatorIndex=1;

estimated_nnz = 0;

isparsedCreator_i=0;

sparseCreator_j=0;

sparseCreator_s=0;

Functions

generateOperatorMatrix (*this*, *runIndex*, *inputArg*)

generateOperatorMatrix implementation should modify operatorMatrix to be the operator matrix at runIndex. Implementation should also return 1 (true) if matrix changed from previously saved matrix and 0 otherwise.

1.3.2 EfieldOperator

class EfieldOperator

Extends *ImplicitOperator*. Describes the electric field term in the kinetic equation.

Functions

generateOperatorMatrix (*this*, *iteration*)

generates the operator matrix for the electric field at iteration from state object, with Legendre mode and number of momentum points defined in state object. Only recomputes if it relevant properties has changed.

1.3.3 CollisionOperator

class CollisionOperator

extends *ImplicitOperator*. Class controlling the collision operator Operator of the equation system. Right now gives support to 4 collision operators. One fully linearized, one fully relativistic (with relativistic bulk) and one with relativistic fast particles but with classical maxwellian bulk. insert info about all the collision operators.

Functions

generateOperatorMatrix (*this*, *iteration*)

Generates the Operator matrix for the iteration, physical quantities taken from the plasma class. Matrix is valid for the grid defined in grid class (number of Legendre Modes, number of radial grid points and exact spacing of the grid points).

CalculateEnergyDependentlnLambda (*this*, *p*, *y*, *lnLambda0*)

returns quantities relevant for Energy dependent lambda. This is an internal function and therefore returns are normalized to fit in with rest of normalization of the code. Calculates the enhancement of the Coulomb logarithm replacing the thermal speed with the relativistic momentum at high energy. Basically, $\nu_S \rightarrow \nu_S * (\ln\Lambda_{\text{hat}} + G_{\text{hat}})$. The formula is a simple interpolation between the energy dependent expression and the thermal speed expression at low energy. References: Wesson p. 792 and $(\ln\Lambda_0 = \ln\Lambda^{\text{ee}})$ and Solodev-Betti (2008) for energy dependence

OUTPUT:

- $\ln\Lambda_{\text{ae}}^{\text{ehat}}$ -
- $d\ln\Lambda_{\text{ae}}^{\text{ehat}}dp$ - derivative of $\ln\Lambda$ with respect to momentum
- $\ln\Lambda_{\text{ae}}^{\text{ihat}}$ -
- $\text{boltzCorrection}\ln\Lambda_{\text{ae}}^{\text{ehat}}$ -
- $\text{boltzCorrection}d\ln\Lambda_{\text{ae}}^{\text{ehat}}dp$ -

All have the same size as momentum grid.

CalculateInelasticEnhancement (*this*, *species*, *p*, *y*, *lnLambda0*, *useInelastic*, *iteration*)

calculates the enhancement of the electron-electron slowing down frequency (ν_S^{ee}) due to inelastic collisions with bound electrons around the ion. Basically, $\nu_S \rightarrow \nu_S * (\ln\Lambda_{\text{hat}} + G_{\text{hat}})$.

INPUTS:

- **species: struct with fields:**
 - $n_j(\text{species}, \text{times})$ - matrix containing number density of each species as a function of time (m^{-3})
 - $Z_{\text{AtomicNumber}}$ - atomic number of each species. row vector, not a function of time
 - $Z_{\text{NetCharge}}$ - net charge for each species. row vector, not a function of time
 - times - time steps. If constant, just set i to 1.

- p = normalized momentum $\gamma m v / (m c) = \Delta y$
- LnLambda the coulomb logarithm
- **useInelastic - which model to use** 0: Ignore inelastic collisions 1: Bethe formula with values of mean excitation energy from Sauer et al. 2: RP Rule of thumb (cutoff at $y=10$ to conserve particles)

OUTPUT:

- G_{hat} -
- dG_{hat}/dp - derivative of G_{hat}

Both have the same size as y .

CalculateSpeciesScreeningFactor (*this, useScreening, Z, Ne, Z0, p, Lmax*)

CalculateSpeciesScreeningFactor calculates the function $g(p)$: the $\nu = (1+g/\text{LnLambda})$, for each legendre mode. p must be a row vector. output: g , with size (L_{max}, N_y) .

INPUTS:

- **species - with the fields**
 - $n_j(\text{jspecies}, \text{times})$ - matrix containing number density of each
 - species as a function of time (m^{-3})
 - $Z_{\text{AtomicNumber}}$ atomic number of each species. row vector, not a function of time
 - $Z0_{\text{NetCharge}}$ net charge for each species. row vector, not a function of time
 - times time steps. If constant, just set i to 1.
- p = normalized momentum $\gamma m v / (m c) = \Delta y$
- LnLambda the coulomb logarithm. It is now a (N_x-1, N_y) matrix! (p -dependent)
- **useScreening: which model to use**
 0. Ignore screening (complete screening)
 1. Fokker–Planck collision operator, TF model (works for all species)
 2. Fokker–Planck collision operator, TF-DFT model (works for some ions)
 3. Fokker–Planck collision operator, full DFT model (works for some ions)
 4. Boltzmann collision operator, TF model
 5. Boltzmann collision operator, TF-DFT model
 6. Boltzmann collision operator, full DFT model
 7. no screening = full penetration

OUTPUT:

- g -

CalculateScreeningFactor (*this, species, p, Lmax, LnLambda, useScreening, iteration*)

CALCULATESCREENINGFACTOR calculates the enhancement of the electron-ion deflection frequency from screening effects

INPUTS:

- **species - struct with the fields**
 - $n_j(\text{jspecies}, \text{times})$ - matrix containing number density of each

- species as a function of time (m^{-3})
- ZAtomicNumber - atomic number of each species. row vector, not a function of time
- Z0NetCharge - net charge for each species. row vector, not a function of time
- times - time steps. If constant, just set i to 1.
- p = normalized momentum $\gamma m v / (m c) = \delta y$
- LnLambda the coulomb logarithm. It is now a (Nxi-1, Ny) matrix! (p-dependent)
- **useScreening: which model to use**
 0. Ignore screening (complete screening)
 1. Fokker–Planck collision operator, TF model (works for all species)
 2. Fokker–Planck collision operator, TF-DFT model (works for some ions)
 3. Fokker–Planck collision operator, full DFT model (works for some ions)
 4. Boltzmann collision operator, TF model
 5. Boltzmann collision operator, TF-DFT model
 6. Boltzmann collision operator, full DFT model
 7. no screening = full penetration

OUTPUT:

- screeningFactor - is a matrix of size (Nxi, Ny), that describes the enhancement of the deflection frequency for the each Legendre mode and all y values on the grid.

CalcgBoltzmann (~, F, Lmax, Z0, Ne)

calculates the Legendre-mode-dependent correction to the Fokker-Planck deflection frequency neglecting screening.

INPUT:

- F(p) - a function handle (if numeric, pchip(F,p) works)
- k - p/mc (vector)
- Z0 - net charge
- Ne: number of bound electrons
- Lmax - maximum legendre mode

The integrals are split at theta_cutoff because the legendres behave badly for too small arguments

OUTPUT:

- boltzmannng -
- p -

1.3.4 SynchrotronLoss

class SynchrotronLoss

Class used to calculate the synchrotron radiation from disitribution. Extends the implicit operator.

Properties

Properties used for the given operator

nueeBarUsed

What collision frequency is used for creating matrix.

BHatUsed

What normalized magnetic field is used for creating matrix.

NxiUsed

What number of legandre mode is used for creating matrix.

yUsed

What momentum grid is used for creating matrix.

deltaRefUsed

Which normalization factor is used for creating matrix.

gammaUsed

What gamma is used for creating matrix.

yMaxBCUsed

What boundary condition is used for creating matrix. (Relevant for where the matrix should start)

useFullSynchOpUsedA

If full synchrotron operator is used for creating matrix.

Functions

```
this = SynchrotronLoss (state, eqSettings)
```

Creates a new instance of this class.

```
matrixHasChanged = generateOperatorMatrix (this, iteration)
```

1.3.5 MagneticDiffusionOperator

```
MagneticDiffusionOperator < ImplicitOperator
```

Calculates local transport sink. Extends *ImplicitOperator*.

Properties

deltaBoverB

Size of radial magnetic field fluctuations

safetyFactor

Safety factor (q) needed for diffusion coefficient estimate

majorRadius

Major radius (in meters) for Rechester-Rosenbluth diffusion

radialScaleLength

Length scale for radial variation of runaway density (in meters)

ionMassNumber

Ion mass number (in proton masses)

1.3.6 Functions

function this = MagneticDiffusionOperator(state, eqSettings, varargin)

Creates a new instance of this class.

function matrixHasChanged = generateOperatorMatrix(this, iteration)

1.4 Explicit Operators

1.4.1 ExplicitOperator

(Abstract) ExplicitOperator < handle & Operator

EXPLICITOPERATOR an abstract superclass for explicit operators used in CODE. Defining abstract functions for updating its operation vector and non abstract utility functions for creating sparse matrices.

Properties

VERSION = 1.0

Functions

function this = ExplicitOperator(state, eqSettings, varargin)

Construct a new instance of this class

1.4.2 ImprovedChiuHarveySource

class ImprovedChiuHarveySource

Calculates pitch angle dependent Chiu Harvey like source and can be switched to have a particle conserving sink term to conserve number of particles and momentum (arissen from the source term). Extends ExplicitOperator. Is *sourceMode* 4,5 in *EquationSettings*.

Functions

generateOperatorMatrix (*this*, *iteration*)

Calculates the explicit source term at the iteration as the time index used. Uses paramters from the *State* object. Returns true if matrix has changed and 0 otherwise.

GenerateKnockOnMatrix (*this*, *iteration*)

Generates the KnockOn terms for the ChiuHarveySource with pitch angle dependencies.

BuildInterpolationMatrix (\sim, x, xp)

Help function used only in the class.

1.4.3 Bremsstrahlung

Bremsstrahlung < **ExplicitOperator**

BOLTZMANN generates the bremsstrahlung matrix basically does the things GenerateMatrices-ForBoltzmannOperators did minus the knock on matrix plus doing the line: `boltzmannMatrix = nBar*(1+this.plasma.Z(iteration))*bremsMatrix`

Properties

NxiUsed

What number of legandre modes used.

pUsed

What momentum vector used

NyInterpUsed

What number interpolation points of y used.

useScreeningUsed

What screening switch used in *EquationSettings*.

bremsModeUsed

What bremsmode is calculated

speciesUsed

Used species. Todo: smart implementation to check that only the current timestep is checked if same

nBarUsed

What density used

ZUsed

Charge used

deltaUsed

Velocity over speed of light used.

lnLambdaRefUsed

Reference coulomb logarithm used.

Functions

this = Bremsstrahlung(state, eqSettings)

Construct a new instance of this class

matrixHasChanged = generateOperatorMatrix(this, iteration)

Call this function to generate the matrix.

```
brMat = GenerateBremsMatrix(this,iteration)
[M,M_source,M_sink] = GenerateBremsstrahlungMatrix(this,mode,smallK)
```

Helpfunctions for building the matrices

```
dlsigma= dlsigma_BetheHeitler(~,p,k)
dlsigma_screened = dlsigma_BetheHeitler_screened(this,p0,k,species)
C = GeneratePitchOperator(~,p,Nxi,kMin,kMax)
[ ZMinusFSquared ] = calculateAveragedFormFactorBrems(this, species, iteration)
W = diffCrossSec(this,p,p1,theta)
outPl = LastLegendrePolynomial(this,l,x)
K = BuildInterpolationMatrix(this,x,xp)
```

1.5 Quantities

1.5.1 State

class State

Class gathering a collection of *PhysicalParams*, *MomentumGrid*, *TimeGrid* and *Reference* objects which point to each other according to their documentation. Can be seen as the total state and resolution at which the equation is solved for. Use this class to initiate the *PhysicalParams*, *MomentumGrid*, *TimeGrid* and *Reference*.

properties

VERSION

Version of class

Dependencies

physicalParams

PhysicalParams object shared with timeGrid and reference

reference

Reference object shared with all other objects in this class

momentumGrid

MomentumGrid object, containing a grid of momentum points

timeGrid

TimeGrid object containing timestep vector amongst other

Parameters for autoInitialGrid

NxiScalingFactor

uniformly rescales the predicted Nxi value from autoInitialGrid by a factor of NxiScalingFactor (default 1)

dyBulkScalingFactor

uniformly rescales the desired grid spacing dy at $y = 0$ used by autoInitialGrid (default 1, lower value yields higher resolution)

dyTailScalingFactor

uniformly rescales the desired grid spacing dy at $y = y_{\text{Max}}$ used by autoInitialGrid (default 1, lower value yields higher resolution)

Nxi_min

Nxi_max

pMax_ceiling

pMaxIncreaseFactor

minPMaxMarginFactor

pSwitch

percentBulk

r

Functions

Constructor

this = State(TRef,nRef)

Construct a new state with *MomentumGrid*, *TimeGrid*, *PhysicalParams* and *Reference* classes.

setInitialRunaway (*this*, *Distribution*)

Sets an initial runaway current to be used for autinitial grid. Distribution is *Distribution* object from which the current is calculated from. The autInitGrid then takes into account for already created runaways when estimating yMax and other parameters.

autoInitGrid (*this*, *useScreening*, *useInelastic*)

Automatically sets yMax, Nxi, Ny and gridWidth for gridMode 6 given the physical scenario. Unless Initial runaway is set, theory using a gaussian distribution as start distribution is used to estimate how far the tail will reach. Requires that all other physical and time parameters are set to their values to give meaningful results.

1.5.2 Reference

class Reference

Properties

Reference values

Design note: intention is that all objects containing an instance of this class also is contained in this object, think of it as pairs. Question is if more than one instance of each class should be able to share a *Reference* object. Right now it is possible to 'hack' the construction by creating a *Reference* object, and for example a *TimeGrid*. Then setting the *Reference* in the *TimeGrid*. Afterwards it is possible to create a new *TimeGrid* and set the already created *Reference* in the new *TimeGrid*. The firstly created *TimeGrid* now has a *Reference* object in it with a *TimeGrid* object in it which is not pointing to itself. The 'pair' structure is then broken. Twofixes are available: one separating the *Reference* to a copy of itself (new pointer) and using the copy for the old *TimeGrid* or save both *TimeGrid* objects in the same *Reference* object.

TRef

Reference temperature

nRef

Reference density

nueeRef

Reference collision frequency

deltaRef

Reference velocity over speed of light

lnLambdaRef

Reference coulumb logarithm

Paired Objects

physicalParams

PhysicalParams object which also contains a pointer to this class.

momentumGrid

MomentumGrid object which also contains a pointer to this class.

timeGrid

TimeGrid object which also contains a pointer to this class.

Functions

Reference (*TRef*, *nRef*)

Constructor

setPhysicalParams (*this*, *phP*)

Set the *PhysicalParams* to passed variable

setMomentumGrid (*this*, *mg*)

Set the *MomentumGrid* to passed variable

setTimeGrid (*this*, *tg*)

Set the *TimeGrid* to passed variable

updateReferenceVals (*this*, *TRef*, *nRef*)

Update *TRef*, *nRef* and all relevant attributes in *TimeGrid*, *MomentumGrid* and *PhysicalParams* objects in this class.

1.5.3 PhysicalParams:

Class containing the physical parameters for the run. The idea is that the parameters you get out will always be interpolated to a *TimeGrid*. This is why you only are allowed to change parameters through function calls as this allows for consistent updates with *TimeGrid*. The reason for this is that if something changes, it is best if only you need to look at one place. So let us say that this would be done in the relevant implementation of *Solver* class instead, then if something needs special treatment in this class, you would need to update all implementations of *Solver*. However, as it is implemented now it is hard to change a variables name (as this is fetched by this class property) but you always now that it is consistent with the *TimeGrid*. Therefore, there are two sets for each supplied physical parameter, the raw supplied parameter from which the interpolation is done and the interpolated parameter. Example *rawT* which is the raw usersupplied temperature and *T* which is its interpolated value.

class PhysicalParams

Properties

Physical quantities:

T

Plasma temperature (eV), given in timesteps

n

Plasma density (m^{-3}), given in timesteps

Z

Plasma effective charge, given in timesteps

E

Electric field (V/m), given in timesteps

B

(optional) Magnetic field (T), only used for calculating synchrotron radiation reaction. If a magnetic field strength is provided, a term describing momentum loss due to synchrotron emission is included in the kinetic equation. If B is empty it is not included.

species

a struct used with the species properties so that species has the fields *nj*, *Z0NetCharge*, *ZAtomicNumber*, *times*:

- *nj*(*jspecies*, *times*) - matrix containing number density of each species as a function of time (m^{-3})
- *ZAtomicNumber* - atomic number of each species. row vector, not a function of time
- *Z0NetCharge* - net charge for each species. row vector, not a function of time
- *times* time steps, same as the *timessteps* vector in *timeGrid*.
- NB species cannot be set from *SetPhysicalParameters*. NB overrides specified *Z* and *n* since *nj* and *Z0NetCharge* specifies the effective charge. (*Z* is the fully screened value) NB only used if *useScreening* ≈ 0

rawspecies

Usersupplied species struct (non time interpolated)

a struct used with the species properties so that species has the fields nj, Z0NetCharge, ZAtomicNumber, times:

- nj(jspecies, times) - matrix containing number density of each species as a function of time (m^{-3})
- ZAtomicNumber - atomic number of each species. row vector, not a function of time
- Z0NetCharge - net charge for each species. row vector, not a function of time
- times time steps, where the ionization and densities are set
- NB species cannot be set from SetPhysicalParameters. NB overrides specified Z and n since nj and Z0NetCharge specifies the effective charge. (Z is the fully screened value) NB only used if useScreening ≈ 0

rawtT

times where rawT is defined, user supplied but to normalized time units (normalized)

rawtn

times where rawn is defined, user supplied but to normalized time units (normalized)

rawtZ

times where rawZ is defined, user supplied but to normalized time units (normalized)

rawtE

times where rawE is defined, user supplied but to normalized time units (normalized)

neTotalOverneFree

vector containing fraction density of total electrons compared to free electrons at all timesteps. Used when species is used (for impurities and boltzmann operator, and bound electrons are present). If no species are used or no source is used, this is set to 1.

nuees

Collisional frequency of electron electron collisions

deltas

thermal speed over speed of light

EOverEc

Electric field over the Critical Electrical field

EOverED

Electric field over the Dreicer Electrical field

lnLambdas

$$\ln \Lambda = 14.9 - 0.5 \ln\left(\frac{n}{10^{20} \text{ m}^{-3}}\right) + \ln\left(\frac{T}{10^3 \text{ eV}}\right)$$

Coloumb logarithm

Miscellaneous:**interpolationMethod**

Interpolation method to interpolate between given values and values at timestepping times. Used in `interp1` (default previous) NOTE: SINCE NEAREST IS DEFAULT, T WILL DROP AT HALF THE TIME YOU SPECIFY WHERE T SHOULD DROP

Time Grid:**timeGrid**

TimeGrid object. It is at these timesteps the parameters are set.

Reference values:**reference**

Reference object.

VERSION

Version of this object

Functions:

this = PhysicalParams (reference, timeGrid, varargin)

Set Physical properties:

setParams (*this*, *varargin*)

Set params in standard matlab syntax, *e.i.* 'name' followed by value. Does not support update of reference or TimeGrid.

setPhysicalParameters (*o*, *T*, *n*, *Z*, *E*, *varargin*)

Usage:

- `setPhysicalParameters(T,n,Z,E)`
- `setPhysicalParameters(T,n,Z,E,B)`
- `setPhysicalParameters(T,n,Z,E,B,tT,tn,tZ,tE)`

*t** should be in normalized units. If *t** is in seconds, simply multiply by `reference.nueeRef`.

setspecies (*this*, *species*)

Sets the species struct

setT (*this*, *T*, *tT*)

Sets the temperature. *tT* is the time at which the temperature is defined. *tT* is optional if *T* is scalar. *tT* should be in normalized units (simply if *tn* is in seconds, multiply by `reference.nueeRef`)

setn (*this*, *n*, *tn*)

Sets the density (m^{-3}). t_n is the time at which the density is defined. t_n is optional if n is scalar. t_n should be in normalized units (simply if t_n is in seconds, multiply by [reference.nueeRef](#))

setZ (*this*, Z , tZ)

Sets the charge of plasma. tZ is the time at which the charge is defined. tZ is optional if Z is scalar. tZ should be in normalized units (simply if t_n is in seconds, multiply by [reference.nueeRef](#))

setE (*this*, E , tE)

Sets the Electric field (V/m) tE is the time at which the electric field is defined. tE is optional if E is scalar. tE should be in normalized units (simply if t_n is in seconds, multiply by [reference.nueeRef](#))

setB (*this*, B)

Sets the magnetic field (Teslas).

Calculate Dependent Properties

calcDepParams (*this*)

Misc.

interpolatePhysicalParams (*this*)

updateRawTimes (*this*, *nueeRefNew*)

PHYSICALPARAMS DO NOT USE THIS FUNCTION, IT IS ONLY FOR REFERENCE CLASS. NO SOLUTION FOUND WHERE ONLY REFERENCE CLASS CAN GET THE RAW TIMES FOUND WHERE YOU ALSO CAN USE THE FUNCTION. DONT USE THIS FUNCTION UNLESS IN REFERENCE CLASS

1.5.4 MomentumGrid

class MomentumGrid

Class for the grid in momentum space.

Properties

Reference values

reference

[Reference](#) object

Resolution parameters:

Nxi

number of Legendre modes used to represent the pitch-angle coordinate (determines resolution in $\xi=\cos(\theta)$)

Ny

number of points in the grid used to represent momentum

yMax

maximum of the momentum grid, in units of $y = \gamma v/v_{th}$. Together, N_y and y_{Max} determine the resolution in momentum

y

momentum grid $p/p_{thermal}$, where $p_{thermal}$ is treated classically

y2

$y.^2$

y4

$y.^4$

x

$v/v_{thermal} \times x$. $x.^2$

gamma

Relativistic gamma function

Differential Calculation Params:

ddy

Differential operator, differentiating with respect to y , requires vector of length same as y

d2dy2

Same as **ddy** but second derivative

yWeights

Integration operator with respect to y , same limitation as **ddy** (row vector)

ddx

Differential operator with respect to x d^2dx^2 - second order differential operator with respect to x

Numerical grid parameters:

yGridMode

Specifies how the points on the momentum grid are chosen

0. uniform grid on $[0, dy \cdot (N_y - 1)]$
1. nonuniform grid, remapped using $y = \text{scaleFactor} \cdot \tan(\pi \cdot a \cdot s / 2)$
2. nonuniform grid, remapped using $y = -\text{scaleFactor} \cdot \ln(a - s)$
3. nonuniform grid, remapped using $y = \text{scaleFactor} \cdot s / (a - s)$
4. nonuniform grid, remapped using $y = s^2 + \text{gridParameter} \cdot s$, where $\text{gridParameter} > 0$ (default)
5. nonuniform grid, smooth tanh step between dense bulk and

sparse tail. gridParameter controls the spacing in the bulk (and is in units of y). gridStepWidth controls the width of the smooth step between bulk and tail, and gridStepPosition its position (in units of y_c). $\text{gridParameter} = 1/15$, $\text{gridStepWidth} = 1/50$ and $\text{gridStepPosition} = 2$ is a good starting point, not implemented yet Above, s is a uniform grid and a is a constant close to 1. When running a CODE+GO calculation, $yGridMode$ must be either 0 or 4. The reason is that only these two remappings are able to handle varying grids in the required way.

gridParameter

for use with yGridMode 4 and 5

gridStepWidth

for use with yGridMode 5

gridStepPosition

for use with yGridMode 5

yMaxBoundaryCondition

Boundary condition at yMax (does this more belong in the equation settings?)

1. Dirichlet: $F=0$ (default)
2. Robin: $dF/dy + (2/y)*F=0$, which forces F to behave like $1/y^2$
3. Do not apply a boundary condition at yMax
4. Dirichlet: $F=0$, with a bit of extra d^2y^2 added at the last few grid points to eliminate ringing

artificialDissipationStrength

only used with yMaxBoundaryCondition=4 to control the amount of ringing

artificialDissipationWidth

in momentum (how close to yMax). Only used with yMaxBoundaryCondition=4 to control the amount of ringing

VERSION

Functions

Constructor

this = MomentumGrid(reference, varargin)

creates new momentum grid

Set functions

setResolution (*this*, *varargin*)

Sets resolution parameters, standard matlab syntax ('name',value). Also reinitializes the momentum grid

setNxi (*this*, *Nxi*)

Sets Nxi to passed value and also reinitializes the momentum grid

setNy (*this*, *Ny*)

Sets Ny to passed value and also reinitializes the momentum grid

setyMax (*this*, *yMax*)

Sets yMax to passed value and also reinitializes the momentum grid

setyGridMode (*this*, *yGridMode*)

Sets yGridMode to passed value and also reinitializes the momentum grid

setgridParameter (*this*, *gridParameter*)

Sets gridParameter to passed value and also reinitializes the momentum grid

setgridStepWidth (*this*, *gridStepWidth*)

Sets gridStepWidth to passed value and also reinitializes the momentum grid

setgridStepPosition (*this*, *gridStepPosition*)

Sets gridStepPosition to passed value and also reinitializes the momentum grid

setyMaxBoundaryCondition (*this*, *yMaxBoundaryCondition*)

Sets yMaxBoundaryCondition to passed value and also reinitializes the momentum grid

setartificialDissipationStrength (*this*, *artificialDissipationStrength*)

Sets artificialDissipationStrength to passed value and also reinitializes the momentum grid

setartificialDissipationWidth (*this*, *artificialDissipationWidth*)

Sets artificialDissipationWidth to passed value and also reinitializes the momentum grid

Misc.

initializeMomentumGrid (*this*)

Creates momentum vectors, derivatives and more

1.5.5 TimeGrid

Class describing and containing the discretization in time.

class TimeGrid

Properties

Time advance

dt

timestep size (normalized). Distance between different time steps in uniform grid. In case of nonuniform, determines the order of magnitude of time difference

tMax

maximum time (minimum is so far always 0) (normalized). Maximum time to simulate to. This will always be exact (to machine precision)

timeStepMode

specifies how the time step vector is generated

0. Constant time step (dt), in units of timeUnit
2. Logarithmic – use progressively longer time steps. Useful for convergence towards a steady state. dt is used for the first time step.
3. Stepwise logarithmic – like 2, except that several time steps are taken with each step length. This is to avoid rebuilding the matrix in every time step.

logGridScaling

step size scaling for the logarithmic grid

logGridSubSteps

how many steps to take for each time step length with timeStepMode 3

logGridMaxStep

maximum time step allowed in timeStepModes 2 and 3

timesteps

actual times the distribution is calculated in (normalized)

dts

vector of all small timechanges in (normalized)

dtsHasChanged

vector containing if the a element in dts vector has changed or not

nTimeSteps

number of times where the distribution is defined (actually one greater than the number of timesteps that should be taken)

Misc**PhysicalParams**

PhysicalParams object.

Design note: intention is that physicalParameters containing an instance of this class also is contained in this object, think of it as pairs. Right now it is possible to ‘hack’ the construction by creating a TimeGrid object, and a PhysicalParams. Then setting the TimeGrid in the PhysicalParams. Afterwards it is possible to create a new PhysicalParams and set the already created TimeGrid in the new PhysicalParams. The firstly created PhysicalParams now has a TimeGrid object in it with a PhysicalParams object in it which is not pointing to itself. The ‘pair’ structure is then broken. Two fixes are available: one separating the TimeGrid to a copy of itself (new pointer) and using the copy for the old TimeGrid or save both TimeGrid objects in the same reference object. properties (SetAccess = protected)

physicalParams

reference

Reference object shared amongst all state objects, containing reference values.

Functions**Constructor**

TimeGrid (*reference*, *varargin*)

Set functions

setPhysicalParams (*this*, *phP*)

Sets phP to passed value and also reinitializes the time grid

setResolution (*this*, *varargin*)

Sets varargin to passed value and also reinitializes the time grid

setdt (*this*, *dt*)

Sets dt to passed value and also reinitializes the time grid

settMax (*this*, *tMax*)

Sets tMax to passed value and also reinitializes the time grid

settimeStepMode (*this*, *timeStepMode*)

Sets timeStepMode to passed value and also reinitializes the time grid

setlogGridScaling (*this*, *logGridScaling*)

Sets logGridScaling to passed value and also reinitializes the time grid

setlogGridSubSteps (*this*, *logGridSubSteps*)

Sets logGridSubSteps to passed value and also reinitializes the time grid

setlogGridMaxStep (*this*, *logGridMaxStep*)

Sets logGridMaxStep to passed value and also reinitializes the time grid

initializeTimeGrid (*this*)

getdt (*this*, *timeUnit*)

Returns dt in specified time unit

Input:

timeUnit

s - seconds

ms - milliseconds

normalized - in nueeRef from

gettMax (*this*, *timeUnit*)

Returns tMax in specified time unit

Input:

timeUnit

s - seconds

ms - milliseconds

normalized - in nueeRef from

gettimeStepMode (*this*)

Returns timeStepMode

Returns timeStepMode

getlogGridScaling (*this*)

Returns logGridScaling

Returns logGridScaling

getlogGridSubSteps (*this*)

Returns logGridSubSteps

Returns logGridSubSteps

getLogGridMaxStep (*this*)

Returns logGridMaxStep

Returns logGridMaxStep

getTimesteps (*this*, *timeUnit*)

Returns timesteps in specified time unit

Input:

timeUnit

s - seconds

ms - milliseconds

normalized - in nueeRef from

getDts (*this*, *pts*, *timeUnit*)

Returns dts in specified timeUnit

Input:

timeUnit

s - seconds

ms - milliseconds

normalized - in nueeRef from

getDtsHasChanged (*this*)

Returns dtsHasChanged

getNTimeSteps (*this*)

Returns nTimeSteps

1.6 Simulation

1.6.1 Solver

class Solver

Abstract solver class structuring a solver class to be used in CODE

Properties

state

State object containing physical information about the plasma such as temperature, Efield, Bfield and about what momentum grid is used. This is a handle class shared by all operator objects

implicitOp

cell containing all implicit operators used in the run all objects in cell should extend ImplicitOperator class

explicitOp

cell containing all explicit operators used in the run all objects in cell should extend ExplicitOperator class

sources

cell containing all sources for the run, care these are not decided how they should be structured inside the code yet

eqSettings

EquationSettings containing what settings for the equation to use

Functions**this = Solver(state, eqSettings)**

Create a new instance of this class.

updateOperators (*this*)

Updates the operators in this class to match those of the *eqSettings* (*EquationSettings* object).

takeTimeSteps (*this*)

Abstract function, should step in time somehow.

1.6.2 SmartLUSolver**class SmartLUSolver**

Extends *Solver* class

Properties**timeIndex**

Time index where simulation will start and where *fattimeIndex* is defined.

fattimeIndex

Distribution function that will be used to simulate next timestep and is thus defined at *timeIndex*.

nSaves

How many saves to do when invoking *takeTimeSteps()*

saveIndices

Which indices will be saved. Initialized after nSaves have been set and is set when invoking *takeTimeSteps()* with *initSaveIndices()*.

factor_L

L factor in LU factorization for last taken step in time.

factor_U

U factor in LU factorization for last taken step in time.

factor_P

P factor in LU factorization for last taken step in time.

factor_Q

Q factor in LU factorization for last taken step in time.

Functions

this = SmartLUSolver(state, eqSettings)

Construct a new instance of this class

output = takeTimeSteps(this, output)

Takes the remaining timesteps, starts from *timeIndex* and ends at end of *TimeGrid* object in *State* object in this class.

If output is supplied, this will be modified by adding new output to it, usefull when extending the distributions. Remember that this is pointers, so you if you supply a output class you only need to write takeTimeSteps(output); (e.i. no assignment as it is a handle class).

Setters

setf (*this*, *distribution*, *momentumGrid*)

Sets the *fattimeIndex* to the distribution (is a *Distribution* object). If *momentumGrid* (*MomentumGrid* object) is supplied then it interpolates the distribution function from distribution to the supplied *momentumGrid* object.

setTime (*this*, *time*, *timeUnit*)

Sets the *timeIndex* to last index where time is greater than the *timesteps* in *TimeGrid* object in *state*.

setnSaves (*this*, *nSaves*)

Set how many steps will be returned. Must be greater than 2.

initializing functions

initsaveIndices (*this*)

Sets what indices to save. Is called by *takeTimeSteps*. Always saves first and last index

clearf (*this*)

Reset the distribution function *fattimeIndex* to be empty.

setftoZero (*this*)

Set the distribution function *fattimeIndex* to all zeroes.

resetttimeIndex (*this*)

Sets the *timeIndex* to be one, e.i. restart the simulation from first timestep.

setftoMaxmellian (*this*)

Sets the distribution function *fattimeIndex* to be Maxmellian.

rhs0 = enforceParticleAndHeat(this)

this adds a particle and heat source dependent on what settings are used in eqSetings. These are added to the 0th Legendre Mode.

1.7 Simulation output

1.7.1 Output

class Output

Properties

Settings

saveDist

switch if distribution is saved, default true

Saved Values

distributions

Cell with Distributions at times.

yc

Cell with yc at times.

nr

Cell with nr at times.

averageEnergyMeV

Cell with averageEnergyMeV at times.

averageFastEnergyMeV

Cell with averageFastEnergyMeV at times.

averageREEnergyMeV

Cell with averageREEnergyMeV at times.

currentDensity

Cell with currentDensity at times.

currentDensityRE

Cell with currentDensityRE at times.

fracRE

Cell with fracRE at times.

fracRECurrent

Cell with fracRECurrent at times.

fracREEnergy

Cell with fracREEnergy at times.

growthRate

Cell with growthRate at times.

growthRatePerSecond

Cell with growthRatePerSecond at times.

times

Cell with times where other values are saved.

totalEnergyMeV

Cell with totalEnergyMeV at times.

totalEnergyJ

Cell with totalEnergyJ at times.

dJdtSI

Cell with dJdtSI at times.

dJ_rundtSI

Cell with dJ_rundtSI at times.

T

Cell with T at times.

n

Cell with n at times.

Z

Cell with Z at times.

E

Cell with E at times.

B

Cell with B at times.

species

Cell with species at times.

neTotalOverneFree

Cell with neTotalOverneFree at times.

nuees

Cell with nuees at times.

deltas

Cell with deltas at times.

EOverEc

Cell with EOverEc at times.

EOverED

Cell with EOverED at times.

EHats

Cell with EHats at times.

BHatRef

Cell with BHatRef at times.

nueeBars

Cell with nueeBars at times.

nBars

Cell with nBars at times.

veBars

Cell with veBars at times.

veBars

Cell with veBars at times.

veBars

Cell with veBars at times.

lnLambdas

Cell with lnLambdas at times.

Functions

this = Output()

save (*this*, *timeIndex*, *state*, *f*, *fbefore*)

Add another entry in all saved properties.

addDistribution (*this*, *f*, *state*, *saveIndex*)

Adds distrubtion function to saved cells, where values of distribution corresponds to saveIndex of state.physicalParams.[PARAM](saveIndex)

[f, times] = getDistributions(this)

Returns distributions and their corresponding times into matlab vector format. Each coloumn is a distribution function at a specific time.

[p, times] = getMomentumVectors(this)

Returns the momentum vectors used for the distributions functions returned in `getDistributions()` and their respective times

1.7.2 Distributions**Properties****Distribution function**

f

Disitrubition function f

$$f = \begin{bmatrix} f_1(y) \\ f_2(y) \\ \vdots \\ f_{N_\xi}(y) \end{bmatrix}$$

where $f_i(y)$ is the projection on the i _th Legendre mode at momentum y where y is defined in *momentumGrid*.

momentumGrid

MomentumGrid where the distribution is defined

Design note, momentum grid is not checked for changes when saved to this class. Therefore if things change in memory to which this points to, it will be unnoticed and wrong.

time

Time where distribution is defined

Reference Values

reference values as *Reference* object in *momentumGrid*. Just saved another time. This will prob dissappear in future version. Access with `[Disitribution].momentumGrid.[TRef,nRef,...]` instead

TRef

nRef

deltaRef

lnLambdaRef

nueeRef

Functions

Constructor

Distribution (f , *momentumGrid*, *time*)

Constructs a new Distribution

Below are different help functions located in CODElib folder documented.

2.1 **cs = CalculateMollerCS(g,g1)**

cs = CalculateMollerCS(g,g1)

Calculates the Møller cross-section (cs) for large-angle electron-electron collisions between an incoming electron with initial $\gamma = g1$ (relativistic gamma function) and an electron at rest, which acquires $\gamma = g$ in the collision. Input g and g1 can be arrays (of the same size).

2.2 **moment = CalculateRunawayMomentXiDependent(Nxi,Ny,f,xiIntMat,weightsMat)**

moment = CalculateRunawayMomentXiDependent(Nxi,Ny,f,xiIntMat,weightsMat)

Calculates the ξ dependent runaway moment given integration matrices.

Input:

Nxi - number of legandre modes

Ny - Number of grid points

f - distribution function

xiIntMat -

weightsMat -

2.3 sigma = CalculateSigmaIntegral(gamma,gamma_m)

sigma = CalculateSigmaIntegral(gamma,gamma_m)

Calculates the integral

$$\int_{\gamma_m}^{\gamma+1-\gamma_m} \Sigma(\gamma, \gamma_1) d\gamma_1$$

analytically. This calculation is needed for the full, conservative close-collision operator. Gamma can be a vector or array, γ_m must be a scalar. If $\gamma + 1 - \gamma_m < \gamma_m$, the integral is set to be zero.

2.4 [m,xiGrid] = GenerateCumIntMat(Nxi,nPoints)

[m,xiGrid] = GenerateCumIntMat(Nxi,nPoints)

Generates grids to cumulatively integrate legendre modes over a xi grid

2.5 f = interpolateDistribution(dist,MG)

f = interpolateDistribution(dist,MG)

Interpolates each legendre mode function from distribution for itself with the momentumGrid in query. The distribution in Distribution is interpreted to be zero above its maximum momentum, and in legendre modes higher than its Nxi. Input:

dist - *Distribution* object, from which the distribution should be interpolated.

MG - *MomentumGrid* object, to which grid the output distribution is interpolated.

2.6 outPls = LegendrePolynomials(l,x)

outPls = LegendrePolynomials(l,x)

LegendrePolynomials calculates $P_l(x)$.

Calculates the legendre polynomials $P_i(x)$ for $i = 0, 1, \dots, l$ using Bonnet's recursion formula:

$$(n+1) * P_{n+1}(x) = (2n+1) * x * P_n(x) - n * P_{n-1}(x),$$

where $P_0(x) = 1$ and $P_1(x) = x$.

Usage: pls = LegendrePolynomials(l,x)

l is the (highest) mode number and x is the coordinate, which must be a row vector (not a matrix).

pls has the structure

$$\begin{bmatrix} P_0(x) \\ P_1(x) \\ \dots \\ P_l(x) \end{bmatrix}$$

2.7 `[x,w] = lgwt(N,a,b)`

`[x,w] = lgwt(N,a,b)`

This script is for computing definite integrals using Legendre-Gauss Quadrature. Computes the Legendre-Gauss nodes and weights on an interval `[a,b]` with truncation order `N`

Suppose you have a continuous function `f(x)` which is defined on `[a,b]` which you can evaluate at any `x` in `[a,b]`. Simply evaluate it at all of the values contained in the `x` vector to obtain a vector `f`. Then compute the definite integral using `sum(f.*w)`.

Written by Greg von Winckel - 02/25/2004

2.8 `[x, w, D, DD] = m20121125_04_DifferentiationMatricesForUniformGrid(N, xMin, xMax, scheme)`

`[x, w, D, DD] = m20121125_04_DifferentiationMatricesForUniformGrid(N, xMin, xMax, scheme)`

Finite difference differentiation matrices and integration weights for a uniform grid.

Created by Matt Landreman, Massachusetts Institute of Technology, Plasma Science & Fusion Center, 2012.

Inputs:

`N` - number of grid points.

`xMin` - minimum value in the domain.

`xMax` - maximum value in the domain.

`scheme` - switch for controlling order of accuracy for differentiation and handling of endpoints.

Options for scheme:

0 - The domain `[xMin, xMax]` is assumed to be periodic. A 3-point stencil is used everywhere. A grid point will be placed at `xMin` but not `xMax`.

1 - Same as `scheme=0`, except a grid point will be placed at `xMax` but not `xMin`.

2 - The domain `[xMin, xMax]` is assumed to be non-periodic. A 3-point stencil is used everywhere. The first and last row of the differentiation matrices will use one-sided differences, so they will each have a non-tridiagonal element.

3 - The same as `scheme=2`, except that the first differentiation matrix will use a 2-point 1-sided stencil for the first and last elements so the matrix is strictly tri-diagonal. The 2nd derivative matrix is the same as for option 2, since it is not possible to compute the 2nd derivative with only a 2-point stencil.

10 - The domain `[xMin, xMax]` is assumed to be periodic. A 5-point stencil is used everywhere. A grid point will be placed at `xMin` but not `xMax`. This option is like `scheme=0` but more accurate.

11 - Same as `scheme=10`, except a grid point will be placed at `xMax` but not `xMin`. This option is like `scheme=1` but more accurate.

12 - The domain `[xMin, xMax]` is assumed to be non-periodic. A 5-point stencil is used everywhere. The first two and last two rows of the differentiation matrices will then each have non-pentadiagonal elements.

13 - The same as option 12, except that 3-point stencils are used for the first and last rows of the differentiation matrices, and 4-point stencils are used for the 2nd and penultimate rows of the differentiation matrices. With this option, both differentiation matrices are strictly penta-diagonal.

20 - The domain [xMin, xMax] is assumed to be periodic. Spectral differentiation matrices are returned. A grid point will be placed at xMin but not xMax.

21 - Same as scheme=20, except a grid point will be placed at xMax but not xMin.

Outputs:

x - column vector with the grid points.

w - column vector with the weights for integration using the trapezoid rule.

D - matrix for differentiation.

DD - matrix for the 2nd derivative.

2.9 **m = MapToXiIntMat(cumintMat,xiGrid,EOverEc,y2,deltaRef)**

m = MapToXiIntMat (cumintMat, xiGrid, EOverEc, y2, deltaRef)

TODO Documentation

2.10 **[posF,negF] = SumLegModesAtXi1(f,Ny,Nxi)**

[posF,negF] = SumLegModesAtXi1 (f, Ny, Nxi)

Calculates the total distribution at xi=1 (v_perp=0) by summing over all the Legendre modes of the distribution

Input:

f - distribution function with normalization as in CODE and Nxi legandre modes projections. first Ny points are for first legandre mode.

Nxi - Number of Legandre Modes

Ny - Number of radial grid points

2.11 **i = CalculateCurrentAlongB(f,y,yWeights, Ny, nRef ,deltaRef)**

i = CalculateCurrentAlongB(f,y,yWeights, Ny, nRef ,deltaRef)

Calculates current density in SI along magnetic axis specifically calculates

$$n \iiint \hat{f} e v d^3 v$$

which reduces to

$$4n_{Ref} e v_{Ref} / (3\sqrt{\pi} m_e^3) \int_0^\infty y^3 / \gamma dy$$

where e is electron charge, and v is the electron speed

Input:

f - distribution %function with normalization as in CODE and Nxi legandre modes projections. first Ny points are for first legandre mode.

y - is vector of momentum points as $y = \gamma v / v_{Ref}$ where γ is relativistic gamma function, v electron speed, and v_{Ref} reference thermal speed of electrons used in y

yWeights - contains dy weights for integrating over y
 nRef - referece density of electrons in SI
 deltaRef - vRef/c where c is speed of light

2.12 **n = CalculateDensity(f,Ny,Nxi,y,yWeights,nRef)**

n = CalculateDensity(f,Ny,Nxi,y,yWeights,nRef)

Calculates density n as integral over momentum space. The density is output in SI.

Input:

f - distribution %function normalized as in CODE
 y - momentum grid
 yWeights - weights for integrating y
 nRef - reference density
 Nxi - Legendre Modes
 Ny - points in y;

2.13 **delta = getDeltaFromT(T)**

delta = getDeltaFromT(T)

Computes the delta ($=v_{th}/c$) parameter used in CODE from the electron temperature (in eV).

2.14 **[delta,lnLambda,collFreq,BHat] = getDerivedParameters(T,density,B)**

[delta,lnLambda,collFreq,BHat] = getDerivedParameters(T,density,B)

Calculates the parameters delta, coloumb logarithm lnLambda and the collision frequency (and time) for given E (in V/m), T (in eV) and density (in m^{-3}). Collision frequency uses Matt's definition Collision time = 1/Collision Frequency

2.15 **[EOverEC,EOverED,EHat] = getNormalizedEFields(E,T,density)**

[EOverEC,EOverED,EHat] = getNormalizedEFields(E,T,density)

Normalizes an electric field in V/m to the critical field EOverEc, and to the dreicer field EOverED E is the field in V/m, T is the electron temperature in eV and density is the electron density in m^{-3} .

2.16 `sigma = GetSpitzerConductivity(T,n,Z)`

`sigma = GetSpitzerConductivity(T,n,Z)`

Calculates the plasma Spitzer conductivity in units of $(\text{Ohm m})^{-1}$, from the formula on p. 72 of Helander & Sigmar. Usage:

`sigma = GetSpitzerConductivity(T,n,Z)`

`n` must be in units of m^{-3} , `T` in eV. There is a prefactor that depends on `Z` which is only known numerically (table on p.74 in Helander & Sigmar and Table III in Spitzer & Härm). Let's interpolate to find the value for any `Z`. Since there is a data point at infinity, let's use $1/Z$ for the interpolation. Simplified expression in Chang (Eq. [5-76]): $\text{sigma} = 19.2307\text{e}3 * T^{(1.5)}/Z./\ln\Lambda$; Written by Adam Stahl

3.1 Code description

In this chapter we describe the overall structure of how indexing is done in various variables

3.1.1 Geneneral time concept

In variables in one dimension that depends on time, the indexing start on 1 which corresponds to the values at the initial distribution (whether the initial distribution is given or not). In variables with more than one dimension, one must check the specific variables documentation to know in which dimension (row/col) the time is defined. Sometimes there are also variables which have two dimensions which of none represent time advancement. Such can for example be when plotting the distribution function i 2D momentum space.

3.1.2 Structure of Common Variables

Here the some variables structure is define.

The distribution function is a 1d array (and is for example present in *SmartLUSolver* as variable `fatTimeIndex` or in *Distribution* as variable f). Each coloumn has the structure

$$f = \begin{bmatrix} f_1(y) \\ f_2(y) \\ \vdots \\ f_{N_\xi}(y) \end{bmatrix}$$

where $f_i(y)$ is the projection on the i _th Legendre mode at momentum y . The associated momentum vector is found in the *State* or *MomentumGrid* objects which are properties to the class.

Time dependent variables. In physical params class, E , T , n and Z can be time dependent. From the users perspective, these are always defined at the associated *TimeGrid* (unless half complicated manouvers are done, see later in this documentation). The raw input data is always saved and it is from this data the interpolation is done.

3.1.3 Equation

The equation beeing solved is the kinetic equation, which reads

$$\frac{df}{dt} + \vec{F} \cdot \frac{d\vec{f}}{d\vec{p}} + \vec{v} \cdot \frac{d\vec{f}}{d\vec{x}} = C\{f\}.$$

In this code, only momentum is considered and all spatial dependence is disregarded. The momentum is parameterized using spherical coordinates, where the polar angle is replaced with the pitch angle. The distribution function is assumed to be independent of the azimuthal angle due to gyro averaging. The pitch angle is described by its cosine as

$$\xi = \cos(\theta)$$

where theta is the pitch angle.

By only considering the average electric field along the magnetic field the equation

$$\frac{df}{dt} + eE(\xi \frac{df}{dp} + \frac{1 - \xi^2}{p} \frac{df}{d\xi}) = C\{f\}$$

is obtained; where p is the momentum, E the electric field strength and e the electron charge. This is then discretized as

$$f = \begin{bmatrix} f_1(y) \\ f_2(y) \\ \vdots \\ f_{N_\xi}(y) \end{bmatrix}$$

where $f_i(y)$ is the projection on the i -th Legendre mode at momentum y .

Numerically this is then rewritten as a matrix equation

$$\overleftrightarrow{O}_{\text{Implicit}} f^{i+1} + \frac{f^{i+1} - f^i}{\Delta t} = \overleftrightarrow{O}_{\text{Explicit}} f^i + S(f).$$

Operators extending `ImplicitOperator` is written to be a part of the operator indexed ‘‘Implicit’’ and operators extending `ExplicitOperator` is written to be part of the operator indexed Explicit. The classes extending the `Source` is written to be added to the $S(f)$ function and is done so since their matrix representation is large. Therefore it is more numerically feasible to handle them explicitly as functions of f . The difference between `Source` and `ExplicitOperator` is that `Source` uses the distribution function all the time and recalculates the source whilst the `ExplicitOperator` calculates a matrix representation and then uses the matrix product to calculate the term.

The equation is normalized to a reference temperature and a reference density such that the unit of time is in electron electron collision times using the most probable thermal speed in the collision frequency. The distribution function is normalized such that a 1D-maxwellian distribution with this normalization and density of reference density acquire a value of 1 at velocity zero. The full derivations and equations is described in ‘‘Reference To Adams Paper’’.

CHAPTER 4

Indices and tables

- `genindex`
- `modindex`
- `search`

A

absoluteSpeedThreshold, 7
 addDistribution() (*built-in function*), 33
 addSparseBlock() (*built-in function*), 5
 addToSparse() (*built-in function*), 5
 artificialDissipationStrength, 24
 artificialDissipationWidth, 24
 autoInitGrid() (*built-in function*), 17
 averageEnergyMeV, 31
 averageFastEnergyMeV, 31
 averageREEnergyMeV, 31

B

B, 19, 32
 BHatRef, 32
 BHatUsed, 13
 bremsMode, 8
 bremsModeUsed, 15
 BuildInterpolationMatrix() (*built-in function*), 14

C

calcDepParams() (*built-in function*), 22
 CalcgBoltzmann() (*built-in function*), 12
 CalculateEnergyDependentlnLambda() (*built-in function*), 10
 CalculateInelasticEnhancement() (*built-in function*), 10
 CalculateScreeningFactor() (*built-in function*), 11
 CalculateSpeciesScreeningFactor() (*built-in function*), 11
 ChiuHarveySource (*built-in class*), 4
 clearf() (*built-in function*), 30
 clearSparseResidue() (*built-in function*), 5
 collisionOperator, 6
 CollisionOperator (*built-in class*), 10
 currentDensity, 31
 currentDensityRE, 31

D

d2dy2, 23
 ddx, 23
 ddy, 23
 ddyUsed, 8
 deltaBoverB, 13
 deltaRef, 18, 34
 deltaRefUsed, 13
 deltas, 20, 32
 deltaUsed, 15
 DetermineFastParticleRegion() (*built-in function*), 4
 DiagonalPlusBoundaryCondition (*built-in class*), 8
 DiagonalPlusBoundaryCondition() (*built-in function*), 8
 Distribution() (*built-in function*), 34
 distributions, 31
 dJ_rundtSI, 32
 dJdtSI, 32
 dt, 25
 dts, 26
 dtsHasChanged, 26
 dyBulkScalingFactor, 17
 dyTailScalingFactor, 17

E

E, 19, 32
 EfieldOperator (*built-in class*), 9
 EHats, 32
 EOverEc, 20, 32
 EOverED, 20, 32
 eqSettings, 3, 5, 9, 29
 EquationSettings (*built-in class*), 6
 estimated_nnz, 5
 explicitOp, 28

F

f, 33

factor_L, 29
factor_P, 29
factor_Q, 29
factor_U, 29
fastParticleDefinition, 7
fattimeIndex, 29
fracRE, 31
fracRECurrent, 31
fracREEnergy, 31

G

gamma, 23
gammaUsed, 13
GenerateKnockOnMatrix() (built-in function), 14
generateOperatorMatrix() (built-in function), 9, 10, 14
getdt() (built-in function), 27
getdts() (built-in function), 28
getdtsHasChanged() (built-in function), 28
getlogGridMaxStep() (built-in function), 28
getlogGridScaling() (built-in function), 27
getlogGridSubSteps() (built-in function), 27
getnTimeSteps() (built-in function), 28
getSourceVec() (built-in function), 3, 4
gettimeStepMode() (built-in function), 27
gettimesteps() (built-in function), 28
gettMax() (built-in function), 27
gridParameter, 23
gridStepPosition, 24
gridStepWidth, 24
growthRate, 31
growthRatePerSecond, 31

I

implicitOp, 28
ImplicitOperator (built-in class), 9
ImprovedChiuHarveySource (built-in class), 14
initializeMomentumGrid() (built-in function), 25
initializeTimeGrid() (built-in function), 27
initsaveIndices() (built-in function), 30
interpolatePhysicalParams() (built-in function), 22
interpolationMethod, 21
ionMassNumber, 14

L

lnLambdaRef, 18, 34
lnLambdaRefUsed, 15
lnLambdas, 20, 33
logGridMaxStep, 26
logGridScaling, 25
logGridSubSteps, 26

M

majorRadius, 13
minPMaxMarginFactor, 17
momentumGrid, 16, 18, 34
MomentumGrid (built-in class), 22

N

n, 19, 32
nBars, 33
nBarUsed, 15
neTotalOverneFree, 20, 32
nPointsXiInt, 8
nr, 31
nr_mask, 4
nRef, 18, 34
nSaves, 29
nTimeSteps, 26
nueeBars, 33
nueeBarUsed, 13
nueeRef, 18, 34
nuees, 20, 32
Nxi, 22
Nxi_max, 17
Nxi_min, 17
NxiScalingFactor, 17
NxiUsed, 8, 13, 15
Ny, 22
NyInterpUsed, 15

O

operatorMatrix, 5, 9
operators, 6
Output (built-in class), 31

P

percentBulk, 17
PhysicalParams, 26
physicalParams, 16, 18
PhysicalParams (built-in class), 19
pMax_ceiling, 17
pMaxIncreaseFactor, 17
predictedNNZ, 5
pSwitch, 17
pUsed, 15

R

r, 17
radialScaleLength, 13
rawspecies, 19
rawtE, 20
rawtn, 20
rawtT, 20
rawtZ, 20

reference, 16, 21, 22, 26
 Reference (*built-in class*), 17
 Reference () (*built-in function*), 18
 relativeSpeedThreshold, 7
 RelLimitUsed, 8
 resetSparseCreator () (*built-in function*), 5
 resettimeIndex () (*built-in function*), 30
 RosenbluthSource (*built-in class*), 4
 runawayRegionMode, 7

S

safetyFactor, 13
 save () (*built-in function*), 33
 saveDist, 31
 saveIndices, 29
 setartificialDissipationStrength ()
 (*built-in function*), 25
 setartificialDissipationWidth () (*built-in function*), 25
 setB () (*built-in function*), 22
 setdt () (*built-in function*), 27
 setE () (*built-in function*), 22
 setf () (*built-in function*), 30
 setftoMaxmellian () (*built-in function*), 30
 setftoZero () (*built-in function*), 30
 setgridParameter () (*built-in function*), 24
 setgridStepPosition () (*built-in function*), 25
 setgridStepWidth () (*built-in function*), 25
 setInitialRunaway () (*built-in function*), 17
 setlogGridMaxStep () (*built-in function*), 27
 setlogGridScaling () (*built-in function*), 27
 setlogGridSubSteps () (*built-in function*), 27
 setMomentumGrid () (*built-in function*), 18
 setn () (*built-in function*), 21
 setnSaves () (*built-in function*), 30
 setNxi () (*built-in function*), 24
 setNy () (*built-in function*), 24
 setParams () (*built-in function*), 21
 setPhysicalParameters () (*built-in function*), 21
 setPhysicalParams () (*built-in function*), 18, 26
 setResolution () (*built-in function*), 24, 26
 setspecies () (*built-in function*), 21
 setT () (*built-in function*), 21
 setTime () (*built-in function*), 30
 setTimeGrid () (*built-in function*), 18
 settimeStepMode () (*built-in function*), 27
 settMax () (*built-in function*), 27
 setyGridMode () (*built-in function*), 24
 setyMax () (*built-in function*), 24
 setyMaxBoundaryCondition () (*built-in function*), 25
 setZ () (*built-in function*), 22
 SmartLUSolver (*built-in class*), 29
 Solver (*built-in class*), 28

Source (*built-in class*), 3
 sourceMode, 7
 sources, 29
 sourceVector, 3
 sparseCreator_i, 5
 sparseCreator_j, 5
 sparseCreator_s, 5
 sparseCreatorIndex, 5
 species, 19, 32
 speciesUsed, 15
 state, 3, 5, 9, 28
 State (*built-in class*), 16
 SynchrotronLoss (*built-in class*), 12

T

T, 19, 32
 tail_mask, 4
 tailThreshold, 7
 takeTimeSteps () (*built-in function*), 29
 time, 34
 timeGrid, 16, 18, 21
 TimeGrid (*built-in class*), 25
 TimeGrid () (*built-in function*), 26
 timeIndex, 29
 times, 32
 timeStepMode, 25
 timesteps, 26
 tMax, 25
 totalEnergyJ, 32
 totalEnergyMeV, 32
 TRef, 18, 34

U

updateOperators () (*built-in function*), 29
 updateRawTimes () (*built-in function*), 22
 updateReferenceVals () (*built-in function*), 19
 useEnergyDependentLnLambdaScreening, 6
 useFullSynchOpUsedA, 13
 useInelastictake, 6
 useNonRelativisticCollOp, 7
 useScreening, 6
 useScreeningUsed, 15

V

veBars, 33
 VERSION, 16, 21, 24

X

x, 23

Y

y, 23
 y2, 23
 y4, 23

[yc](#), [31](#)
[yCutSource](#), [7](#)
[yGridMode](#), [23](#)
[yMax](#), [22](#)
[yMaxBC](#), [8](#)
[yMaxBCUsed](#), [13](#)
[yMaxBoundaryCondition](#), [24](#)
[yUsed](#), [13](#)
[yWeights](#), [23](#)

Z

[Z](#), [19](#), [32](#)
[ZUsed](#), [15](#)