

HEWES : Heisenberg-Euler Weak-Field Expansion Simulator

Generated by Doxygen 1.9.4

1 HEWES: Heisenberg-Euler Weak-Field Expansion Simulator	1
1.1 Contents	1
1.2 Preparing the Makefile	1
1.3 Short User Manual	2
1.3.1 Note on Simulation Settings	3
1.3.2 Note on Resource Occupation	3
1.3.3 Note on Output Analysis	3
1.4 Authors	4
2 Hierarchical Index	5
2.1 Class Hierarchy	5
3 Data Structure Index	7
3.1 Data Structures	7
4 File Index	9
4.1 File List	9
5 Data Structure Documentation	11
5.1 Gauss1D Class Reference	11
5.1.1 Detailed Description	12
5.1.2 Constructor & Destructor Documentation	12
5.1.2.1 Gauss1D()	12
5.1.3 Member Function Documentation	13
5.1.3.1 addToSpace()	13
5.1.4 Field Documentation	14
5.1.4.1 kx	14
5.1.4.2 ky	14
5.1.4.3 kz	14
5.1.4.4 phig	14
5.1.4.5 phix	15
5.1.4.6 phiy	15
5.1.4.7 phiz	15
5.1.4.8 px	15
5.1.4.9 py	16
5.1.4.10 pz	16
5.1.4.11 x0x	16
5.1.4.12 x0y	16
5.1.4.13 x0z	17
5.2 Gauss2D Class Reference	17
5.2.1 Detailed Description	18
5.2.2 Constructor & Destructor Documentation	18
5.2.2.1 Gauss2D()	18
5.2.3 Member Function Documentation	19

5.2.3.1 addToSpace()	19
5.2.4 Field Documentation	20
5.2.4.1 A1	20
5.2.4.2 A2	20
5.2.4.3 Amp	20
5.2.4.4 axis	21
5.2.4.5 dis	21
5.2.4.6 lambda	21
5.2.4.7 Ph0	21
5.2.4.8 PhA	22
5.2.4.9 phip	22
5.2.4.10 w0	22
5.2.4.11 zr	22
5.3 Gauss3D Class Reference	23
5.3.1 Detailed Description	24
5.3.2 Constructor & Destructor Documentation	24
5.3.2.1 Gauss3D()	24
5.3.3 Member Function Documentation	25
5.3.3.1 addToSpace()	25
5.3.4 Field Documentation	25
5.3.4.1 A1	26
5.3.4.2 A2	26
5.3.4.3 Amp	26
5.3.4.4 axis	26
5.3.4.5 dis	27
5.3.4.6 lambda	27
5.3.4.7 Ph0	27
5.3.4.8 PhA	27
5.3.4.9 phip	28
5.3.4.10 w0	28
5.3.4.11 zr	28
5.4 gaussian1D Struct Reference	28
5.4.1 Detailed Description	29
5.4.2 Field Documentation	29
5.4.2.1 k	29
5.4.2.2 p	29
5.4.2.3 phi	29
5.4.2.4 phig	29
5.4.2.5 x0	30
5.5 gaussian2D Struct Reference	30
5.5.1 Detailed Description	30
5.5.2 Field Documentation	30

5.5.2.1 amp	30
5.5.2.2 axis	31
5.5.2.3 ph0	31
5.5.2.4 phA	31
5.5.2.5 phip	31
5.5.2.6 w0	31
5.5.2.7 x0	32
5.5.2.8 zr	32
5.6 gaussian3D Struct Reference	32
5.6.1 Detailed Description	32
5.6.2 Field Documentation	32
5.6.2.1 amp	33
5.6.2.2 axis	33
5.6.2.3 ph0	33
5.6.2.4 phA	33
5.6.2.5 phip	33
5.6.2.6 w0	34
5.6.2.7 x0	34
5.6.2.8 zr	34
5.7 ICSetter Class Reference	34
5.7.1 Detailed Description	35
5.7.2 Member Function Documentation	35
5.7.2.1 add()	36
5.7.2.2 addGauss1D()	36
5.7.2.3 addGauss2D()	37
5.7.2.4 addGauss3D()	38
5.7.2.5 addPlaneWave1D()	38
5.7.2.6 addPlaneWave2D()	39
5.7.2.7 addPlaneWave3D()	40
5.7.2.8 eval()	40
5.7.3 Field Documentation	41
5.7.3.1 gauss1Ds	41
5.7.3.2 gauss2Ds	41
5.7.3.3 gauss3Ds	42
5.7.3.4 planeWaves1D	42
5.7.3.5 planeWaves2D	42
5.7.3.6 planeWaves3D	42
5.8 Lattice Class Reference	43
5.8.1 Detailed Description	44
5.8.2 Constructor & Destructor Documentation	44
5.8.2.1 Lattice()	45
5.8.3 Member Function Documentation	45

5.8.3.1	get_dataPointDimension()	45
5.8.3.2	get_dx()	46
5.8.3.3	get_dy()	46
5.8.3.4	get_dz()	46
5.8.3.5	get_ghostLayerWidth()	47
5.8.3.6	get_stencilOrder()	47
5.8.3.7	get_tot_lx()	48
5.8.3.8	get_tot_ly()	49
5.8.3.9	get_tot_lz()	49
5.8.3.10	get_tot_noDP()	50
5.8.3.11	get_tot_noP()	50
5.8.3.12	get_tot_nx()	50
5.8.3.13	get_tot_ny()	50
5.8.3.14	get_tot_nz()	51
5.8.3.15	initializeCommunicator()	51
5.8.3.16	setDiscreteDimensions()	52
5.8.3.17	setPhysicalDimensions()	52
5.8.4	Field Documentation	53
5.8.4.1	comm	53
5.8.4.2	dataPointDimension	53
5.8.4.3	dx	54
5.8.4.4	dy	54
5.8.4.5	dz	54
5.8.4.6	ghostLayerWidth	54
5.8.4.7	my_prc	55
5.8.4.8	n_prc	55
5.8.4.9	statusFlags	55
5.8.4.10	stencilOrder	55
5.8.4.11	sunctx	56
5.8.4.12	tot_lx	56
5.8.4.13	tot_ly	56
5.8.4.14	tot_lz	56
5.8.4.15	tot_noDP	57
5.8.4.16	tot_noP	57
5.8.4.17	tot_nx	57
5.8.4.18	tot_ny	57
5.8.4.19	tot_nz	58
5.9	LatticePatch Class Reference	58
5.9.1	Detailed Description	61
5.9.2	Constructor & Destructor Documentation	61
5.9.2.1	LatticePatch()	62
5.9.2.2	~LatticePatch()	62

5.9.3 Member Function Documentation	62
5.9.3.1 checkFlag()	63
5.9.3.2 derive()	64
5.9.3.3 derotate()	69
5.9.3.4 discreteSize()	71
5.9.3.5 exchangeGhostCells()	72
5.9.3.6 generateTranslocationLookup()	74
5.9.3.7 getDelta()	76
5.9.3.8 initializeBuffers()	77
5.9.3.9 origin()	78
5.9.3.10 rotateIntoEigen()	79
5.9.3.11 rotateToX()	80
5.9.3.12 rotateToY()	82
5.9.3.13 rotateToZ()	83
5.9.4 Friends And Related Function Documentation	84
5.9.4.1 generatePatchwork	84
5.9.5 Field Documentation	85
5.9.5.1 buffData	85
5.9.5.2 buffX	85
5.9.5.3 buffY	85
5.9.5.4 buffZ	86
5.9.5.5 du	86
5.9.5.6 duData	86
5.9.5.7 dx	86
5.9.5.8 dy	87
5.9.5.9 dz	87
5.9.5.10 envelopeLattice	87
5.9.5.11 gCLData	87
5.9.5.12 gCRData	88
5.9.5.13 ghostCellLeft	88
5.9.5.14 ghostCellLeftToSend	88
5.9.5.15 ghostCellRight	88
5.9.5.16 ghostCellRightToSend	89
5.9.5.17 ghostCells	89
5.9.5.18 ghostCellsToSend	89
5.9.5.19 ID	89
5.9.5.20 lgcTox	89
5.9.5.21 lgcToy	90
5.9.5.22 lgcToz	90
5.9.5.23 Llx	90
5.9.5.24 Lly	90
5.9.5.25 Llz	91

5.9.5.26 lx	91
5.9.5.27 ly	91
5.9.5.28 lz	91
5.9.5.29 nx	92
5.9.5.30 ny	92
5.9.5.31 nz	92
5.9.5.32 rgcTox	92
5.9.5.33 rgcToy	93
5.9.5.34 rgcToz	93
5.9.5.35 statusFlags	93
5.9.5.36 u	93
5.9.5.37 uAux	94
5.9.5.38 uAuxData	94
5.9.5.39 uData	94
5.9.5.40 uTox	94
5.9.5.41 uToy	95
5.9.5.42 uToz	95
5.9.5.43 x0	95
5.9.5.44 xTou	95
5.9.5.45 y0	96
5.9.5.46 yTou	96
5.9.5.47 z0	96
5.9.5.48 zTou	96
5.10 OutputManager Class Reference	97
5.10.1 Detailed Description	97
5.10.2 Constructor & Destructor Documentation	97
5.10.2.1 OutputManager()	98
5.10.3 Member Function Documentation	98
5.10.3.1 generateOutputFolder()	98
5.10.3.2 getSimCode()	99
5.10.3.3 outUState()	100
5.10.3.4 set_outputStyle()	102
5.10.3.5 SimCodeGenerator()	102
5.10.4 Field Documentation	103
5.10.4.1 outputStyle	103
5.10.4.2 Path	103
5.10.4.3 simCode	104
5.11 PlaneWave Class Reference	104
5.11.1 Detailed Description	105
5.11.2 Field Documentation	105
5.11.2.1 kx	105
5.11.2.2 ky	105

5.11.2.3 kz	105
5.11.2.4 phix	106
5.11.2.5 phiy	106
5.11.2.6 phiz	106
5.11.2.7 px	106
5.11.2.8 py	107
5.11.2.9 pz	107
5.12 planewave Struct Reference	107
5.12.1 Detailed Description	107
5.12.2 Field Documentation	108
5.12.2.1 k	108
5.12.2.2 p	108
5.12.2.3 phi	108
5.13 PlaneWave1D Class Reference	109
5.13.1 Detailed Description	109
5.13.2 Constructor & Destructor Documentation	110
5.13.2.1 PlaneWave1D()	110
5.13.3 Member Function Documentation	110
5.13.3.1 addToSpace()	111
5.14 PlaneWave2D Class Reference	111
5.14.1 Detailed Description	112
5.14.2 Constructor & Destructor Documentation	112
5.14.2.1 PlaneWave2D()	112
5.14.3 Member Function Documentation	113
5.14.3.1 addToSpace()	113
5.15 PlaneWave3D Class Reference	114
5.15.1 Detailed Description	115
5.15.2 Constructor & Destructor Documentation	115
5.15.2.1 PlaneWave3D()	115
5.15.3 Member Function Documentation	116
5.15.3.1 addToSpace()	116
5.16 Simulation Class Reference	116
5.16.1 Detailed Description	118
5.16.2 Constructor & Destructor Documentation	118
5.16.2.1 Simulation()	119
5.16.2.2 ~Simulation()	119
5.16.3 Member Function Documentation	120
5.16.3.1 addInitialConditions()	120
5.16.3.2 addPeriodicCLayerInX()	121
5.16.3.3 addPeriodicCLayerInXY()	122
5.16.3.4 advanceToTime()	123
5.16.3.5 checkFlag()	124

5.16.3.6 checkNoFlag()	125
5.16.3.7 get_cart_comm()	126
5.16.3.8 initializeCVODEobject()	127
5.16.3.9 initializePatchwork()	128
5.16.3.10 outAllFieldData()	129
5.16.3.11 setDiscreteDimensionsOfLattice()	130
5.16.3.12 setInitialConditions()	131
5.16.3.13 setPhysicalDimensionsOfLattice()	132
5.16.3.14 start()	133
5.16.4 Field Documentation	134
5.16.4.1 ccode_mem	134
5.16.4.2 icsettings	135
5.16.4.3 lattice	135
5.16.4.4 latticePatch	135
5.16.4.5 NLS	135
5.16.4.6 outputManager	136
5.16.4.7 statusFlags	136
5.16.4.8 t	136
5.17 TimeEvolution Class Reference	136
5.17.1 Detailed Description	137
5.17.2 Member Function Documentation	137
5.17.2.1 f()	137
5.17.3 Field Documentation	138
5.17.3.1 c	138
5.17.3.2 TimeEvolver	138
6 File Documentation	139
6.1 README.md File Reference	139
6.2 src/DerivationStencils.cpp File Reference	139
6.2.1 Detailed Description	139
6.3 DerivationStencils.cpp	140
6.4 src/DerivationStencils.h File Reference	140
6.4.1 Detailed Description	142
6.4.2 Function Documentation	142
6.4.2.1 s10b() [1/2]	142
6.4.2.2 s10b() [2/2]	143
6.4.2.3 s10c() [1/2]	143
6.4.2.4 s10c() [2/2]	144
6.4.2.5 s10f() [1/2]	144
6.4.2.6 s10f() [2/2]	145
6.4.2.7 s11b() [1/2]	146
6.4.2.8 s11b() [2/2]	146

6.4.2.9 s11f() [1/2]	147
6.4.2.10 s11f() [2/2]	147
6.4.2.11 s12b() [1/2]	148
6.4.2.12 s12b() [2/2]	149
6.4.2.13 s12c() [1/2]	149
6.4.2.14 s12c() [2/2]	150
6.4.2.15 s12f() [1/2]	151
6.4.2.16 s12f() [2/2]	151
6.4.2.17 s13b() [1/2]	152
6.4.2.18 s13b() [2/2]	152
6.4.2.19 s13f() [1/2]	153
6.4.2.20 s13f() [2/2]	154
6.4.2.21 s1b() [1/2]	154
6.4.2.22 s1b() [2/2]	155
6.4.2.23 s1f() [1/2]	156
6.4.2.24 s1f() [2/2]	156
6.4.2.25 s2b() [1/2]	157
6.4.2.26 s2b() [2/2]	157
6.4.2.27 s2c() [1/2]	158
6.4.2.28 s2c() [2/2]	158
6.4.2.29 s2f() [1/2]	159
6.4.2.30 s2f() [2/2]	159
6.4.2.31 s3b() [1/2]	160
6.4.2.32 s3b() [2/2]	160
6.4.2.33 s3f() [1/2]	161
6.4.2.34 s3f() [2/2]	161
6.4.2.35 s4b() [1/2]	162
6.4.2.36 s4b() [2/2]	162
6.4.2.37 s4c() [1/2]	163
6.4.2.38 s4c() [2/2]	163
6.4.2.39 s4f() [1/2]	164
6.4.2.40 s4f() [2/2]	164
6.4.2.41 s5b() [1/2]	165
6.4.2.42 s5b() [2/2]	165
6.4.2.43 s5f() [1/2]	166
6.4.2.44 s5f() [2/2]	166
6.4.2.45 s6b() [1/2]	167
6.4.2.46 s6b() [2/2]	167
6.4.2.47 s6c() [1/2]	168
6.4.2.48 s6c() [2/2]	168
6.4.2.49 s6f() [1/2]	169
6.4.2.50 s6f() [2/2]	169

6.4.2.51 s7b() [1/2]	170
6.4.2.52 s7b() [2/2]	170
6.4.2.53 s7f() [1/2]	171
6.4.2.54 s7f() [2/2]	171
6.4.2.55 s8b() [1/2]	172
6.4.2.56 s8b() [2/2]	172
6.4.2.57 s8c() [1/2]	173
6.4.2.58 s8c() [2/2]	173
6.4.2.59 s8f() [1/2]	174
6.4.2.60 s8f() [2/2]	174
6.4.2.61 s9b() [1/2]	175
6.4.2.62 s9b() [2/2]	176
6.4.2.63 s9f() [1/2]	176
6.4.2.64 s9f() [2/2]	177
6.5 DerivationStencils.h	177
6.6 src/ICSetters.cpp File Reference	181
6.6.1 Detailed Description	181
6.7 ICSetters.cpp	181
6.8 src/ICSetters.h File Reference	186
6.8.1 Detailed Description	187
6.9 ICSetters.h	187
6.10 src/LatticePatch.cpp File Reference	191
6.10.1 Detailed Description	191
6.10.2 Function Documentation	191
6.10.2.1 check_error()	192
6.10.2.2 check_retval()	192
6.10.2.3 errorKill()	193
6.10.2.4 generatePatchwork()	194
6.11 LatticePatch.cpp	196
6.12 src/LatticePatch.h File Reference	207
6.12.1 Detailed Description	209
6.12.2 Function Documentation	209
6.12.2.1 check_error()	209
6.12.2.2 check_retval()	209
6.12.2.3 errorKill()	210
6.12.3 Variable Documentation	211
6.12.3.1 BuffersInitialized	211
6.12.3.2 FLatticeDimensionSet	212
6.12.3.3 FLatticePatchSetUp	212
6.12.3.4 GhostLayersInitialized	212
6.12.3.5 TranslocationLookupSetUp	212
6.13 LatticePatch.h	213

6.14 src/main.cpp File Reference	215
6.14.1 Detailed Description	216
6.14.2 Function Documentation	216
6.14.2.1 main()	216
6.15 main.cpp	220
6.16 src/Outputters.cpp File Reference	223
6.16.1 Detailed Description	223
6.17 Outputters.cpp	223
6.18 src/Outputters.h File Reference	225
6.18.1 Detailed Description	226
6.19 Outputters.h	226
6.20 src/SimulationClass.cpp File Reference	227
6.20.1 Detailed Description	227
6.21 SimulationClass.cpp	227
6.22 src/SimulationClass.h File Reference	231
6.22.1 Detailed Description	232
6.22.2 Variable Documentation	232
6.22.2.1 CcodeObjectSetUp	233
6.22.2.2 LatticeDiscreteSetUp	233
6.22.2.3 LatticePatchworkSetUp	233
6.22.2.4 LatticePhysicalSetUp	233
6.22.2.5 SimulationStarted	234
6.23 SimulationClass.h	234
6.24 src/SimulationFunctions.cpp File Reference	235
6.24.1 Detailed Description	236
6.24.2 Function Documentation	236
6.24.2.1 Sim1D()	236
6.24.2.2 Sim2D()	238
6.24.2.3 Sim3D()	241
6.24.2.4 timer()	243
6.25 SimulationFunctions.cpp	244
6.26 src/SimulationFunctions.h File Reference	247
6.26.1 Detailed Description	249
6.26.2 Function Documentation	249
6.26.2.1 Sim1D()	249
6.26.2.2 Sim2D()	251
6.26.2.3 Sim3D()	254
6.26.2.4 timer()	256
6.27 SimulationFunctions.h	257
6.28 src/TimeEvolutionFunctions.cpp File Reference	258
6.28.1 Detailed Description	259
6.28.2 Function Documentation	259

6.28.2.1 linear1DProp()	259
6.28.2.2 linear2DProp()	261
6.28.2.3 linear3DProp()	263
6.28.2.4 nonlinear1DProp()	265
6.28.2.5 nonlinear2DProp()	270
6.28.2.6 nonlinear3DProp()	274
6.29 TimeEvolutionFunctions.cpp	278
6.30 src/TimeEvolutionFunctions.h File Reference	286
6.30.1 Detailed Description	287
6.30.2 Function Documentation	287
6.30.2.1 linear1DProp()	288
6.30.2.2 linear2DProp()	289
6.30.2.3 linear3DProp()	291
6.30.2.4 nonlinear1DProp()	293
6.30.2.5 nonlinear2DProp()	298
6.30.2.6 nonlinear3DProp()	302
6.31 TimeEvolutionFunctions.h	306
Index	307

Chapter 1

HEWES: Heisenberg-Euler Weak-Field Expansion Simulator

The Heisenberg-Euler Weak-Field Expansion Simulator is a solver for the all-optical QED vacuum. It solves the equations of motion for electromagnetic waves in the Heisenberg-Euler effective QED theory in the weak-field expansion with up to six-photon processes.

There is a [paper](#) that introduces the algorithm and shows remarkable scientific results. Check that out before the code if you are interested in this project!

1.1 Contents

- Preparing the Makefile
- Short User Manual
 - Hints for Settings
 - Note on Resource Occupation
 - Note on Output Analysis
- Authors

1.2 Preparing the Makefile

The following descriptions assume you are using a Unix-like system.

The *make* utility is used for building and a recent compiler version. Features up to the C++20 standard are used. *OpenMP* is optional to enforce more vectorization and enable multi-threading. The latter is useful for performance only when a very large number of nodes is used.

Additionally required software:

- An MPI implementation such as [OpenMPI](#) or [MPICH](#).
- The [SUNDIALS](#) package with the [CVODE](#) solver.
Version 6 is required. The code is presumably compliant with the upcoming version 7.
For the installation of *SUNDIALS*, [CMake](#) is required. Enable MPI and specify the directory of the `mpicxx` wrapper for use of the MPI-based `NVECTOR_PARALLEL` module. 32-bit integer size is sufficient. Make sure to edit the *SUNDIALS* include and library directories in the provided minimal [Makefile](#).

1.3 Short User Manual

You have full control over all high-level simulation settings via the `main.cpp` file.

- First, specify the path you want the output data to go via the variable `outputDirectory`.
- Second, decide if you want to simulate in 1D, 2D, or 3D and uncomment only that full section. You can then specify
 - the relative and absolute integration tolerances of the *CVODE* solver. Recommended values are between 1e-12 and 1e-18.
 - the order of accuracy of the numerical scheme via the stencil order. You can choose an integer in the range 1-13.
 - the physical side lengths of the grid in meters.
 - the number of lattice points per dimension.
 - the slicing of the lattice into patches (only for 2D and 3D simulations, automatic in 1D) – this determines the number of patches and therefore the required distinct processing units for MPI. The total number of processes is given by the product of patches in any dimension. Note: In the 3D case patches are required to be cubic in terms of lattice points. This is decisive for computational efficiency and checked at compile-time.
 - whether to have periodic or vanishing boundary values (currently has to be chosen periodic).
 - whether you want to simulate on top of the linear vacuum only 4-photon processes (1), 6-photon processes (2), both (3), or none (0) – the linear Maxwell case.
 - the total time of the simulation in units $c=1$, i.e., the distance propagated by the light waves in meters.
 - the number of time steps that will be solved stepwise by *CVODE*. In order to keep interpolation errors small do not choose this number too small.
 - the multiple of steps at which you want the data to be written to disk.
 - the output format. It can be 'c' for comma separated values (csv), or 'b' for binary. For csv format the name of the files written to the output directory is of the form `{step_number}_{process_number}.csv`. For binary output all data per step is written into one file and the step number is the name of the file.
 - which electromagnetic waveform(s) you want to propagate. You can choose between a plane wave (not much physical content, but useful for checks) and implementations of Gaussians in 1D, 2D, and 3D. Their parameters can be tuned. A description of the wave implementations is given in [ref.pdf](#). Note that the 3D Gaussians, as they are implemented up to now, should be propagated in the xy-plane. More waveform implementations will follow in subsequent versions of the code.

A doxygen-generated complete code reference is provided with [ref.pdf](#).

- Third, in the `src` directory, build the executable `Simulation` via the `make` command.
- Forth, run the simulation. Make sure to use `src` as working directory as the code uses a relative path to log the configuration in `main.cpp`. You determine the number of processes via the MPI execution command. Note that in 2D and 3D simulations this number has to coincide with the actual number of patches, as described above. Here, the simulation would be executed distributed over four processes:


```
mpirun -np 4 ./Simulation
```


- Monitor stdout and stderr. The unique simulation identifier number (starting timestep = name of data directory), the process steps, and the used wall times per step are printed on stdout. Errors are printed on stderr.
Note: Convergence of the employed *CVODE* solver can not be guaranteed and issues of this kind can hardly be predicted. On top, they are even system dependent. Piece of advice: Only pass decimal numbers for the grid settings and initial conditions.
CVODE warnings and errors are reported on stdout and stderr.
A `config.txt` file containing the configuration part of `main.cpp` is written to the output directory in order to save the simulation settings of each particular run.

You can remove the object files and the executable via `make clean`.

1.3.1 Note on Simulation Settings

You may want to start with two Gaussian pulses in 1D colliding head-on in a pump-probe setup. For this event, specify a high-frequency probe pulse with a low amplitude and a low-frequency pump pulse with a high frequency. Both frequencies should be chosen to be below a forth of the Nyquist frequency to avoid nonphysical dispersion effects. The wavelengths should neither be chosen too large (bulky wave) on a fine patchwork of narrow patches. Their communication might be problematic with too small halo layer depths. You would observe a blurring over time. The amplitudes need be below 1 – the critical field strength – for the weak-field expansion to be valid. You can then investigate the arising of higher harmonics in frequency space via a Fourier analysis. The signals from the higher harmonics can be highlighted by subtracting the results of the same simulation in the linear Maxwell vacuum. You will be left with the nonlinear effects. Choosing the probe pulse to be polarized with an angle to the polarization of the pump you may observe a fractional polarization flip of the probe due to their nonlinear interaction. Decide beforehand which steps you need to be written to disk for your analysis.

Example scenarios of colliding Gaussians are preconfigured for any dimension.

1.3.2 Note on Resource Occupation

The computational load depends mostly on the grid size and resolution. The order of accuracy of the numerical scheme and *CVODE* are rather secondary except for simulations running on many processing units, as the communication load is dependent on the stencil order. Simulations in 1D are relatively cheap and can easily be run on a modern laptop within minutes. The output size per step is less than a megabyte. Simulations in 2D with about one million grid points are still feasible for a personal machine but might take about an hour of time to finish. The output size per step is in the range of some dozen megabytes. Sensible simulations in 3D require large memory resources and therefore need to be run on distributed systems. Even hundreds of cores can be kept busy for many hours or days. The output size quickly amounts to dozens of gigabytes for just a single state.

1.3.3 Note on Output Analysis

The field data are either written in csv format to one file per MPI process, the ending of which (after an underscore) corresponds to the process number, as described above. This is not an elegant solution, but a portable way that also works fast and is straightforward to analyze. Or, the option recommended for many larger write operations, in binary format with a single file per output step. Raw bytes are written to the files as they are in memory. This option is more performant and achieved with MPI IO. However, there is no guarantee of portability; postprocessing/conversion is required. The step number is the file name.

A `SimResults` folder is created in the chosen output directory if it does not exist and a folder named after the starting timestep of the simulation (in the form `yy-mm-dd_hh-MM-ss`) is created where the output files are written into. There are six columns in the csv files, corresponding to the six components of the electromagnetic

field: E_x , E_y , E_z , B_x , B_y , B_z . Each row corresponds to one lattice point.

Postprocessing is required to read-in the files in order. A [Python module](#) taking care of this is provided.

Likewise, another [Python module](#) is provided to read the binary data of a selected field component into a numpy array – its portability, however, cannot be guaranteed.

The process numbers first align along dimension 1 until the number of patches is that direction is reached, then continue on dimension two and finally fill dimension 3. For example, for a 3D simulation on $4 \times 4 \times 64$ cores, the field data is divided over the patches as follows:

z=1	z=2	z=3	z=4
x	x
^	^		
1 0 4 8 12	1 16 20 24 28		
2 1 5 9 13	2 17 21 25 29		
3 2 6 10 14	3 18 22 26 30		
4 3 7 11 15	4 19 23 27 31		
----->	----->		
1 2 3 4 y	1 2 3 4 y		

The axes denote the physical dimensions that are each divided into 4 sectors in this example. The numbers inside the 4×4 squares indicate the process number, which is the number of the patch and also the number at the end of the corresponding output csv file. The ordering of the array within a patch follows the standard C convention and can be reshaped in 2D and 3D to the actual size of the path.

More information describing settings and analysis procedures used for actual scientific results are given in an open-access [paper](#).

Some example Python analysis scripts can be found in the examples. The [first steps](#) demonstrate how the simulated data is accurately read-in from disk to numpy arrays using the provided [get field data module](#). [Harmonic generation](#) in various forms is sketched as one application showing nonlinear quantum vacuum effects. Analyses of 3D simulations are more involved due to large volumes of data. Visualization requires tools like Paraview; examples are shown [here](#). There is however *no simulation data provided* as it would make the repository size unnecessarily large.

1.4 Authors

- Arnau Pons Domenech
- Hartmut Ruhl (hartmut.ruhl@physik.uni-muenchen.de)
- Andreas Lindner (and.lindner@physik.uni-muenchen.de)
- Baris Ölmez (b.oelmez@physik.uni-muenchen.de)

Chapter 2

Hierarchical Index

2.1 Class Hierarchy

This inheritance list is sorted roughly, but not completely, alphabetically:

Gauss1D	11
Gauss2D	17
Gauss3D	23
gaussian1D	28
gaussian2D	30
gaussian3D	32
ICSetter	34
Lattice	43
LatticePatch	58
OutputManager	97
PlaneWave	104
PlaneWave1D	109
PlaneWave2D	111
PlaneWave3D	114
planewave	107
Simulation	116
TimeEvolution	136

Chapter 3

Data Structure Index

3.1 Data Structures

Here are the data structures with brief descriptions:

Gauss1D	Class for Gaussian pulses in 1D	11
Gauss2D	Class for Gaussian pulses in 2D	17
Gauss3D	Class for Gaussian pulses in 3D	23
gaussian1D	1D Gaussian wave structure	28
gaussian2D	2D Gaussian wave structure	30
gaussian3D	3D Gaussian wave structure	32
ICSetter	ICSetter class to initialize wave types with default parameters	34
Lattice	Lattice class for the construction of the enveloping discrete simulation space	43
LatticePatch	LatticePatch class for the construction of the patches in the enveloping lattice	58
OutputManager	Output Manager class to generate and coordinate output writing to disk	97
PlaneWave	Super-class for plane waves	104
planewave	Plane wave structure	107
PlaneWave1D	Class for plane waves in 1D	109
PlaneWave2D	Class for plane waves in 2D	111
PlaneWave3D	Class for plane waves in 3D	114
Simulation	Simulation class to instantiate the whole walkthrough of a Simulation	116
TimeEvolution	Monostate TimeEvolution class to propagate the field data in time in a given order of the HE weak-field expansion	136

Chapter 4

File Index

4.1 File List

Here is a list of all files with brief descriptions:

src/ DerivationStencils.cpp	
Empty. All definitions in the header	139
src/ DerivationStencils.h	
Definition of derivation stencils from order 1 to 13	140
src/ ICSetters.cpp	
Implementation of the plane wave and Gaussian wave packets	181
src/ ICSetters.h	
Declaration of the plane wave and Gaussian wave packets	186
src/ LatticePatch.cpp	
Costruction of the overall envelope lattice and the lattice patches	191
src/ LatticePatch.h	
Declaration of the lattice and lattice patches	207
src/ main.cpp	
Main function to configure the user's simulation settings	215
src/ Outputters.cpp	
Generation of output writing to disk	223
src/ Outputters.h	
OutputManager class to outstream simulation data	225
src/ SimulationClass.cpp	
Interface to the whole Simulation procedure: from wave settings over lattice construction, time evolution and outputs (also all relevant CVODE steps are performed here)	227
src/ SimulationClass.h	
Class for the Simulation object calling all functionality: from wave settings over lattice construction, time evolution and outputs initialization of the CNode object	231
src/ SimulationFunctions.cpp	
Implementation of the complete simulation functions for 1D, 2D, and 3D, as called in the main function	235
src/ SimulationFunctions.h	
Full simulation functions for 1D, 2D, and 3D used in main.cpp	247
src/ TimeEvolutionFunctions.cpp	
Implementation of functions to propagate data vectors in time according to Maxwell's equations, and various orders in the HE weak-field expansion	258
src/ TimeEvolutionFunctions.h	
Functions to propagate data vectors in time according to Maxwell's equations, and various orders in the HE weak-field expansion	286

Chapter 5

Data Structure Documentation

5.1 Gauss1D Class Reference

class for Gaussian pulses in 1D

```
#include <src/ICSetters.h>
```

Public Member Functions

- [Gauss1D](#) (std::array< sunrealtype, 3 > k={1, 0, 0}, std::array< sunrealtype, 3 > p={0, 0, 1}, std::array< sunrealtype, 3 > xo={0, 0, 0}, sunrealtype phig_=1.0, std::array< sunrealtype, 3 > phi={0, 0, 0})
construction with default parameters
- void [addToSpace](#) (sunrealtype x, sunrealtype y, sunrealtype z, sunrealtype *pTo6Space) const
function for the actual implementation in space

Private Attributes

- sunrealtype [kx](#)
wavenumber k_x
- sunrealtype [ky](#)
wavenumber k_y
- sunrealtype [kz](#)
wavenumber k_z
- sunrealtype [px](#)
polarization & amplitude in x-direction, p_x
- sunrealtype [py](#)
polarization & amplitude in y-direction, p_y
- sunrealtype [pz](#)
polarization & amplitude in z-direction, p_z
- sunrealtype [phix](#)
phase shift in x-direction, ϕ_x
- sunrealtype [phiy](#)
phase shift in y-direction, ϕ_y
- sunrealtype [phiz](#)

- *phase shift in z-direction, ϕ_z*
- sunrealtype [x0x](#)
center of pulse in x-direction, x_0
- sunrealtype [x0y](#)
center of pulse in y-direction, y_0
- sunrealtype [x0z](#)
center of pulse in z-direction, z_0
- sunrealtype [phig](#)
pulse width Φ_g

5.1.1 Detailed Description

class for Gaussian pulses in 1D

They are given in the form $\vec{E} = \vec{p} \exp\left(-(\vec{x} - \vec{x}_0)^2 / \Phi_g^2\right) \cos(\vec{k} \cdot \vec{x})$

Definition at line [83](#) of file [ICSetters.h](#).

5.1.2 Constructor & Destructor Documentation

5.1.2.1 Gauss1D()

```
Gauss1D::Gauss1D (
    std::array< sunrealtype, 3 > k = {1, 0, 0},
    std::array< sunrealtype, 3 > p = {0, 0, 1},
    std::array< sunrealtype, 3 > xo = {0, 0, 0},
    sunrealtype phig_ = 1.0,
    std::array< sunrealtype, 3 > phi = {0, 0, 0} )
```

construction with default parameters

[Gauss1D](#) construction with

- wavevectors k_x
- k_y
- k_z normalized to $1/\lambda$
- amplitude (polarization) in x-direction
- amplitude (polarization) in y-direction
- amplitude (polarization) in z-direction
- phase shift in x-direction
- phase shift in y-direction
- phase shift in z-direction
- width

- shift from origin in x-direction
- shift from origin in y-direction
- shift from origin in z-direction

Definition at line 125 of file ICSetters.cpp.

```
00127
00128     kx = k[0];          /** - wavevectors \f$ k_x \f$ */
00129     ky = k[1];          /** - \f$ k_y \f$ */
00130     kz = k[2];          /** - \f$ k_z \f$ normalized to \f$ 1/\lambda \f$ */
00131     px = p[0];          /** - amplitude (polarization) in x-direction */
00132     py = p[1];          /** - amplitude (polarization) in y-direction */
00133     pz = p[2];          /** - amplitude (polarization) in z-direction */
00134     phix = phi[0];       /** - phase shift in x-direction */
00135     phiy = phi[1];       /** - phase shift in y-direction */
00136     phiz = phi[2];       /** - phase shift in z-direction */
00137     phig = phig;         /** - width */
00138     x0x = xo[0];          /** - shift from origin in x-direction */
00139     x0y = xo[1];          /** - shift from origin in y-direction */
00140     x0z = xo[2];          /** - shift from origin in z-direction */
00141 }
```

References [kx](#), [ky](#), [kz](#), [phig](#), [phix](#), [phiy](#), [phiz](#), [px](#), [py](#), [pz](#), [x0x](#), [x0y](#), and [x0z](#).

5.1.3 Member Function Documentation

5.1.3.1 addToSpace()

```
void Gauss1D::addToSpace (
    sunrealtype x,
    sunrealtype y,
    sunrealtype z,
    sunrealtype * pTo6Space ) const
```

function for the actual implementation in space

[Gauss1D](#) implementation in space

Definition at line 144 of file ICSetters.cpp.

```
00145
00146     const sunrealtype wavelength =
00147         sqrt(kx * kx + ky * ky + kz * kz); /* \f$ 1/\lambda \f$ */
00148     x = x - x0x; /* x-coordinate minus shift from origin */
00149     y = y - x0y; /* y-coordinate minus shift from origin */
00150     z = z - x0z; /* z-coordinate minus shift from origin */
00151     const sunrealtype kScalarX = (kx * x + ky * y + kz * z) * 2 *
00152         std::numbers::pi; /* \f$ 2\pi \cdot \vec{k} \cdot \vec{x} \f$ */
00153     const sunrealtype envelopeAmp =
00154         exp(-(x * x + y * y + z * z) / phig / phig); /* enveloping Gauss shape */
00155     // Gaussian wave definition
00156     const std::array<sunrealtype, 3> E{
00157         {
00158             px * cos(kScalarX - phix) * envelopeAmp, /* \f$ E_x \f$ */
00159             py * cos(kScalarX - phiy) * envelopeAmp, /* \f$ E_y \f$ */
00160             pz * cos(kScalarX - phiz) * envelopeAmp}}; /* \f$ E_z \f$ */
00161     // Put E-field into space
00162     pTo6Space[0] += E[0];
00163     pTo6Space[1] += E[1];
00164     pTo6Space[2] += E[2];
00165     // and B-field
00166     pTo6Space[3] += (ky * E[2] - kz * E[1]) / wavelength;
00167     pTo6Space[4] += (kz * E[0] - kx * E[2]) / wavelength;
00168     pTo6Space[5] += (kx * E[1] - ky * E[0]) / wavelength;
00169 }
```

References [kx](#), [ky](#), [kz](#), [phig](#), [phix](#), [phiy](#), [phiz](#), [px](#), [py](#), [pz](#), [x0x](#), [x0y](#), and [x0z](#).

5.1.4 Field Documentation

5.1.4.1 k_x

```
sunrealtype Gauss1D::kx [private]
```

wavenumber k_x

Definition at line 86 of file [ICSetters.h](#).

Referenced by [addToSpace\(\)](#), and [Gauss1D\(\)](#).

5.1.4.2 k_y

```
sunrealtype Gauss1D::ky [private]
```

wavenumber k_y

Definition at line 88 of file [ICSetters.h](#).

Referenced by [addToSpace\(\)](#), and [Gauss1D\(\)](#).

5.1.4.3 k_z

```
sunrealtype Gauss1D::kz [private]
```

wavenumber k_z

Definition at line 90 of file [ICSetters.h](#).

Referenced by [addToSpace\(\)](#), and [Gauss1D\(\)](#).

5.1.4.4 Φ_g

```
sunrealtype Gauss1D::phig [private]
```

pulse width Φ_g

Definition at line 110 of file [ICSetters.h](#).

Referenced by [addToSpace\(\)](#), and [Gauss1D\(\)](#).

5.1.4.5 phix

```
sunrealtype Gauss1D::phix [private]
```

phase shift in x-direction, ϕ_x

Definition at line 98 of file [ICSetters.h](#).

Referenced by [addToSpace\(\)](#), and [Gauss1D\(\)](#).

5.1.4.6 phiy

```
sunrealtype Gauss1D::phiy [private]
```

phase shift in y-direction, ϕ_y

Definition at line 100 of file [ICSetters.h](#).

Referenced by [addToSpace\(\)](#), and [Gauss1D\(\)](#).

5.1.4.7 phiz

```
sunrealtype Gauss1D::phiz [private]
```

phase shift in z-direction, ϕ_z

Definition at line 102 of file [ICSetters.h](#).

Referenced by [addToSpace\(\)](#), and [Gauss1D\(\)](#).

5.1.4.8 px

```
sunrealtype Gauss1D::px [private]
```

polarization & amplitude in x-direction, p_x

Definition at line 92 of file [ICSetters.h](#).

Referenced by [addToSpace\(\)](#), and [Gauss1D\(\)](#).

5.1.4.9 py

```
sunrealtype Gauss1D::py [private]
```

polarization & amplitude in y-direction, p_y

Definition at line 94 of file [ICSetters.h](#).

Referenced by [addToSpace\(\)](#), and [Gauss1D\(\)](#).

5.1.4.10 pz

```
sunrealtype Gauss1D::pz [private]
```

polarization & amplitude in z-direction, p_z

Definition at line 96 of file [ICSetters.h](#).

Referenced by [addToSpace\(\)](#), and [Gauss1D\(\)](#).

5.1.4.11 x0x

```
sunrealtype Gauss1D::x0x [private]
```

center of pulse in x-direction, x_0

Definition at line 104 of file [ICSetters.h](#).

Referenced by [addToSpace\(\)](#), and [Gauss1D\(\)](#).

5.1.4.12 x0y

```
sunrealtype Gauss1D::x0y [private]
```

center of pulse in y-direction, y_0

Definition at line 106 of file [ICSetters.h](#).

Referenced by [addToSpace\(\)](#), and [Gauss1D\(\)](#).

5.1.4.13 x0z

sunrealtype Gauss1D::x0z [private]

center of pulse in z-direction, z_0

Definition at line 108 of file ICSetters.h.

Referenced by [addToSpace\(\)](#), and [Gauss1D\(\)](#).

The documentation for this class was generated from the following files:

- [src/ICSetters.h](#)
- [src/ICSetters.cpp](#)

5.2 Gauss2D Class Reference

class for Gaussian pulses in 2D

```
#include <src/ICSetters.h>
```

Public Member Functions

- [Gauss2D](#) (std::array< sunrealtype, 3 > dis_={0, 0, 0}, std::array< sunrealtype, 3 > axis_={1, 0, 0}, sunrealtype Amp_=1.0, sunrealtype phip_=0, sunrealtype w0_=1e-5, sunrealtype zr_=4e-5, sunrealtype Ph0_=2e-5, sunrealtype PhA_=0.45e-5)
construction with default parameters
- void [addToSpace](#) (sunrealtype x, sunrealtype y, sunrealtype z, sunrealtype *pTo6Space) const
function for the actual implementation in space

Private Attributes

- std::array< sunrealtype, 3 > [dis](#)
distance maximum to origin
- std::array< sunrealtype, 3 > [axis](#)
normalized propagation axis
- sunrealtype [Amp](#)
amplitude A
- sunrealtype [phip](#)
polarization rotation from TE-mode around propagation direction
- sunrealtype [w0](#)
taille ω_0
- sunrealtype [zr](#)
Rayleigh length $z_R = \pi \omega_0^2 / \lambda$.
- sunrealtype [Ph0](#)
center of beam Φ_0
- sunrealtype [PhA](#)
length of beam Φ_A
- sunrealtype [A1](#)
amplitude projection on TE-mode
- sunrealtype [A2](#)
amplitude projection on xy-plane
- sunrealtype [lambda](#)
wavelength λ

5.2.1 Detailed Description

class for Gaussian pulses in 2D

They are given in the form $\vec{E} = A \vec{\epsilon} \sqrt{\frac{\omega_0}{\omega(z)}} \exp(-r/\omega(z))^2 \exp(-(z_g - \Phi_0)/\Phi_A)^2) \cos\left(\frac{k r^2}{2R(z)} + g(z) - k z_g\right)$ with

- propagation direction (subtracted distance to origin) z_g
- radial distance to propagation axis $r = \sqrt{\vec{x}^2 - z_g^2}$
- $k = 2\pi/\lambda$
- waist at position z , $\omega(z) = w_0 \sqrt{1 + (z_g/z_R)^2}$
- Gouy phase $g(z) = \tan^{-1}(z_g/z_r)$
- beam curvature $R(z) = z_g (1 + (z_r/z_g)^2)$ obtained via the chosen parameters

Definition at line 139 of file [ICSetters.h](#).

5.2.2 Constructor & Destructor Documentation

5.2.2.1 Gauss2D()

```
Gauss2D::Gauss2D (
    std::array< sunrealtype, 3 > dis_ = {0, 0, 0},
    std::array< sunrealtype, 3 > axis_ = {1, 0, 0},
    sunrealtype Amp_ = 1.0,
    sunrealtype phip_ = 0,
    sunrealtype w0_ = 1e-5,
    sunrealtype zr_ = 4e-5,
    sunrealtype Ph0_ = 2e-5,
    sunrealtype PhA_ = 0.45e-5 )
```

construction with default parameters

[Gauss2D](#) construction with

- center it approaches
- direction form where it comes
- amplitude
- polarization rotation from TE-mode
- taille
- Rayleigh length
- beam center
- beam length

Definition at line 172 of file `ICSetters.cpp`.

```
00175 {
00176     dis = dis_;           /** - center it approaches */
00177     axis = axis_;         /** - direction form where it comes */
00178     Amp = Amp_;           /** - amplitude */
00179     phip = phip_;         /** - polarization rotation from TE-mode */
00180     w0 = w0_;             /** - taille */
00181     zr = zr_;             /** - Rayleigh length */
00182     Ph0 = Ph0_;           /** - beam center */
00183     PhA = PhA_;           /** - beam length */
00184     A1 = Amp * cos(phpip); // amplitude in z-direction
00185     A2 = Amp * sin(phpip); // amplitude on xy-plane
00186     lambda = std::numbers::pi * w0 * w0 / zr; // formula for wavelength
00187 }
```

References [A1](#), [A2](#), [Amp](#), [axis](#), [dis](#), [lambda](#), [Ph0](#), [PhA](#), [phpip](#), [w0](#), and [zr](#).

5.2.3 Member Function Documentation

5.2.3.1 addToSpace()

```
void Gauss2D::addToSpace (
    sunrealtype x,
    sunrealtype y,
    sunrealtype z,
    sunrealtype * pTo6Space ) const
```

function for the actual implementation in space

Definition at line 189 of file `ICSetters.cpp`.

```
00190 {
00191     //f$ \vec{x} = \vec{x}_0-\vec{dis} \f$ // coordinates minus distance to
00192     //origin
00193     x -= dis[0];
00194     y -= dis[1];
00195     // z=-dis[2];
00196     z = nan("0x12345"); // unused parameter
00197     // \f$ z_g = \vec{x}\cdot\vec{e}_g \f$ projection on propagation axis
00198     const sunrealtype zg =
00199         x * axis[0] + y * axis[1]; //+z*axis[2]; // =z-z0 -> propagation
00200         //direction, minus origin
00201     // \f$ r = \sqrt{\vec{x}^2 - z_g^2} \f$ -> pythagoras of radius minus
00202     // projection on prop axis
00203     const sunrealtype r = sqrt((x * x + y * y - zg * zg));
00204     zg * zg; // radial distance to propagation axis
00205     // \f$ w(z) = w0\sqrt{1+(z_g/z_R)^2} \f$
00206     // waist at position z
00207     const sunrealtype wz = w0 * sqrt(1 + (zg * zg / zr / zr));
00208     // \f$ g(z) = atan(z_g/z_r) \f$
00209     const sunrealtype gz = atan(zg / zr); // Gouy phase
00210     // \f$ R(z) = z_g*(1+(z_r/z_g)^2) \f$
00211     sunrealtype Rz = nan("0x12345"); // beam curvature
00212     if (abs(zg) > 1e-15)
00213         Rz = zg * (1 + (zr * zr / zg / zg));
00214     else
00215         Rz = 1e308;
00216     // wavenumber \f$ k = 2\pi/\lambda \f$
00217     const sunrealtype k = 2 * std::numbers::pi / lambda;
00218     // \f$ \Phi_F = kr^2/(2R(z))+g(z)-kz_g \f$
00219     const sunrealtype PhF =
00220         -k * r * r / (2 * Rz) + gz - k * zg; // to be inserted into cosine
00221     // \f$ G = \sqrt{w_0/w_z}\e^{-(r/w(z))^2}\e^{(zg-Ph0)^2/PhA^2}\cos(PhF) \f$
00222     // CNode is a diva, no chance to remove the square in the second exponential
00223     // -> h too small
00224     const sunrealtype G2D = sqrt(w0 / wz) * exp(-r * r / wz / wz) *
00225         exp(-(zg - Ph0) * (zg - Ph0) / PhA / PhA) *
00226         cos(PhF); // gauss shape
00227     // \f$ c_\alpha = \vec{e}_x\cdot\vec{axis} \f$
00228     // projection components; do like this for CNode convergence -> otherwise
00229     // results in machine error values for non-existent field components if
00230     // axis[0] and axis[1] are given
```

```

00231  const sunrealtype ca =
00232      axis[0]; // x-component of propagation axis which is given as parameter
00233  // no z-component for 2D propagation
00234  const sunrealtype sa = sqrt(1 - ca * ca);
00235  // E-field to space: polarization in xy-plane (A2) is projection of
00236  // z-polarization (A1) on x- and y-directions
00237  pTo6Space[0] += sa * (G2D * A2);
00238  pTo6Space[1] += -ca * (G2D * A2);
00239  pTo6Space[2] += G2D * A1;
00240  // B-field -> negative derivative wrt polarization shift of E-field
00241  pTo6Space[3] += -sa * (G2D * A1);
00242  pTo6Space[4] += ca * (G2D * A1);
00243  pTo6Space[5] += G2D * A2;
00244  }

```

References [A1](#), [A2](#), [axis](#), [dis](#), [lambda](#), [Ph0](#), [PhA](#), [w0](#), and [zr](#).

5.2.4 Field Documentation

5.2.4.1 A1

```
sunrealtype Gauss2D::A1 [private]
```

amplitude projection on TE-mode

Definition at line 159 of file [ICS setters.h](#).

Referenced by [addToSpace\(\)](#), and [Gauss2D\(\)](#).

5.2.4.2 A2

```
sunrealtype Gauss2D::A2 [private]
```

amplitude projection on xy-plane

Definition at line 161 of file [ICS setters.h](#).

Referenced by [addToSpace\(\)](#), and [Gauss2D\(\)](#).

5.2.4.3 Amp

```
sunrealtype Gauss2D::Amp [private]
```

amplitude A

Definition at line 146 of file [ICS setters.h](#).

Referenced by [Gauss2D\(\)](#).

5.2.4.4 axis

```
std::array<sunrealtype, 3> Gauss2D::axis [private]
```

normalized propagation axis

Definition at line 144 of file [ICSetters.h](#).

Referenced by [addToSpace\(\)](#), and [Gauss2D\(\)](#).

5.2.4.5 dis

```
std::array<sunrealtype, 3> Gauss2D::dis [private]
```

distance maximum to origin

Definition at line 142 of file [ICSetters.h](#).

Referenced by [addToSpace\(\)](#), and [Gauss2D\(\)](#).

5.2.4.6 lambda

```
sunrealtype Gauss2D::lambda [private]
```

wavelength λ

Definition at line 163 of file [ICSetters.h](#).

Referenced by [addToSpace\(\)](#), and [Gauss2D\(\)](#).

5.2.4.7 Ph0

```
sunrealtype Gauss2D::Ph0 [private]
```

center of beam Φ_0

Definition at line 155 of file [ICSetters.h](#).

Referenced by [addToSpace\(\)](#), and [Gauss2D\(\)](#).

5.2.4.8 PhA

```
sunrealtype Gauss2D::PhA [private]
```

length of beam Φ_A

Definition at line 157 of file [ICSetters.h](#).

Referenced by [addToSpace\(\)](#), and [Gauss2D\(\)](#).

5.2.4.9 phip

```
sunrealtype Gauss2D::phip [private]
```

polarization rotation from TE-mode around propagation direction

Definition at line 149 of file [ICSetters.h](#).

Referenced by [Gauss2D\(\)](#).

5.2.4.10 w0

```
sunrealtype Gauss2D::w0 [private]
```

taille ω_0

Definition at line 151 of file [ICSetters.h](#).

Referenced by [addToSpace\(\)](#), and [Gauss2D\(\)](#).

5.2.4.11 zr

```
sunrealtype Gauss2D::zr [private]
```

Rayleigh length $z_R = \pi\omega_0^2/\lambda$.

Definition at line 153 of file [ICSetters.h](#).

Referenced by [addToSpace\(\)](#), and [Gauss2D\(\)](#).

The documentation for this class was generated from the following files:

- [src/ICSetters.h](#)
- [src/ICSetters.cpp](#)

5.3 Gauss3D Class Reference

class for Gaussian pulses in 3D

```
#include <src/ICSetters.h>
```

Public Member Functions

- [Gauss3D](#) (std::array< sunrealtype, 3 > dis_={0, 0, 0}, std::array< sunrealtype, 3 > axis_={1, 0, 0}, sunrealtype Amp_=1.0, sunrealtype phip_=0, sunrealtype w0_=1e-5, sunrealtype zr_=4e-5, sunrealtype Ph0_=2e-5, sunrealtype PhA_=0.45e-5)
construction with default parameters
- void [addToSpace](#) (sunrealtype x, sunrealtype y, sunrealtype z, sunrealtype *pTo6Space) const
function for the actual implementation in space

Private Attributes

- std::array< sunrealtype, 3 > [dis](#)
distance maximum to origin
- std::array< sunrealtype, 3 > [axis](#)
normalized propagation axis
- sunrealtype [Amp](#)
amplitude A
- sunrealtype [phip](#)
polarization rotation from TE-mode around propagation direction
- sunrealtype [w0](#)
taille ω_0
- sunrealtype [zr](#)
Rayleigh length $z_R = \pi\omega_0^2/\lambda$.
- sunrealtype [Ph0](#)
center of beam Φ_0
- sunrealtype [PhA](#)
length of beam Φ_A
- sunrealtype [A1](#)
amplitude projection on TE-mode (z-axis)
- sunrealtype [A2](#)
amplitude projection on xy-plane
- sunrealtype [lambda](#)
wavelength λ

5.3.1 Detailed Description

class for Gaussian pulses in 3D

They are given in the form $\vec{E} = A \vec{e} \frac{\omega_0}{\omega(z)} \exp(-r/\omega(z))^2 \exp(-(z_g - \Phi_0)/\Phi_A)^2) \cos\left(\frac{k r^2}{2R(z)} + g(z) - k z_g\right)$ with

- propagation direction (subtracted distance to origin) z_g
- radial distance to propagation axis $r = \sqrt{\vec{x}^2 - z_g^2}$
- $k = 2\pi/\lambda$
- waist at position z , $\omega(z) = w_0 \sqrt{1 + (z_g/z_R)^2}$
- Gouy phase $g(z) = \tan^{-1}(z_g/z_r)$
- beam curvature $R(z) = z_g (1 + (z_r/z_g)^2)$ obtained via the chosen parameters

Definition at line 193 of file [ICSetters.h](#).

5.3.2 Constructor & Destructor Documentation

5.3.2.1 Gauss3D()

```
Gauss3D::Gauss3D (
    std::array< sunrealtype, 3 > dis_ = {0, 0, 0},
    std::array< sunrealtype, 3 > axis_ = {1, 0, 0},
    sunrealtype Amp_ = 1.0,
    sunrealtype phip_ = 0,
    sunrealtype w0_ = 1e-5,
    sunrealtype zr_ = 4e-5,
    sunrealtype Ph0_ = 2e-5,
    sunrealtype PhA_ = 0.45e-5 )
```

construction with default parameters

[Gauss3D](#) construction with

- center it approaches
- direction from where it comes
- amplitude
- polarization rotation form TE-mode
- taille
- Rayleigh length
- beam center
- beam length

Definition at line 247 of file ICSetters.cpp.

```
00252 {
00253     dis = dis_; /** - center it approaches */
00254     axis = axis_; /** - direction from where it comes */
00255     Amp = Amp_; /** - amplitude */
00256     // pol=pol_;
00257     phip = phip_; /** - polarization rotation form TE-mode */
00258     w0 = w0_; /** - taille */
00259     zr = zr_; /** - Rayleigh length */
00260     Ph0 = Ph0_; /** - beam center */
00261     PhA = PhA_; /** - beam length */
00262     lambda = std::numbers::pi * w0 * w0 / zr;
00263     A1 = Amp * cos(hip);
00264     A2 = Amp * sin(hip);
00265 }
```

References [A1](#), [A2](#), [Amp](#), [axis](#), [dis](#), [lambda](#), [Ph0](#), [PhA](#), [hip](#), [w0](#), and [zr](#).

5.3.3 Member Function Documentation

5.3.3.1 addToSpace()

```
void Gauss3D::addToSpace (
    sunrealtype x,
    sunrealtype y,
    sunrealtype z,
    sunrealtype * pTo6Space ) const
```

function for the actual implementation in space

[Gauss3D](#) implementation in space

Definition at line 268 of file ICSetters.cpp.

```
00269 {
00270     x -= dis[0];
00271     y -= dis[1];
00272     z -= dis[2];
00273     const sunrealtype zg = x * axis[0] + y * axis[1] + z * axis[2];
00274     const sunrealtype r = sqrt((x * x + y * y + z * z) - zg * zg);
00275     const sunrealtype wz = w0 * sqrt(1 + (zg * zg / zr / zr));
00276     const sunrealtype gz = atan(zg / zr);
00277     sunrealtype Rz = nan("0x12345");
00278     if (abs(zg) > 1e-15)
00279         Rz = zg * (1 + (zr * zr / zg / zg));
00280     else
00281         Rz = 1e308;
00282     const sunrealtype k = 2 * std::numbers::pi / lambda;
00283     const sunrealtype PhF = -k * r * r / (2 * Rz) + gz - k * zg;
00284     const sunrealtype G3D = (w0 / wz) * exp(-r * r / wz / wz) *
00285         exp(-(zg - Ph0) * (zg - Ph0) / PhA / PhA) * cos(PhF);
00286     const sunrealtype ca = axis[0];
00287     const sunrealtype sa = sqrt(1 - ca * ca);
00288     pTo6Space[0] += sa * (G3D * A2);
00289     pTo6Space[1] += -ca * (G3D * A2);
00290     pTo6Space[2] += G3D * A1;
00291     pTo6Space[3] += -sa * (G3D * A1);
00292     pTo6Space[4] += ca * (G3D * A1);
00293     pTo6Space[5] += G3D * A2;
00294 }
```

References [A1](#), [A2](#), [axis](#), [dis](#), [lambda](#), [Ph0](#), [PhA](#), [w0](#), and [zr](#).

5.3.4 Field Documentation

5.3.4.1 A1

```
sunrealtype Gauss3D::A1 [private]
```

amplitude projection on TE-mode (z-axis)

Definition at line 215 of file [ICSetters.h](#).

Referenced by [addToSpace\(\)](#), and [Gauss3D\(\)](#).

5.3.4.2 A2

```
sunrealtype Gauss3D::A2 [private]
```

amplitude projection on xy-plane

Definition at line 217 of file [ICSetters.h](#).

Referenced by [addToSpace\(\)](#), and [Gauss3D\(\)](#).

5.3.4.3 Amp

```
sunrealtype Gauss3D::Amp [private]
```

amplitude A

Definition at line 200 of file [ICSetters.h](#).

Referenced by [Gauss3D\(\)](#).

5.3.4.4 axis

```
std::array<sunrealtype, 3> Gauss3D::axis [private]
```

normalized propagation axis

Definition at line 198 of file [ICSetters.h](#).

Referenced by [addToSpace\(\)](#), and [Gauss3D\(\)](#).

5.3.4.5 dis

```
std::array<sunrealtype, 3> Gauss3D::dis [private]
```

distance maximum to origin

Definition at line 196 of file [ICSetters.h](#).

Referenced by [addToSpace\(\)](#), and [Gauss3D\(\)](#).

5.3.4.6 lambda

```
sunrealtype Gauss3D::lambda [private]
```

wavelength λ

Definition at line 219 of file [ICSetters.h](#).

Referenced by [addToSpace\(\)](#), and [Gauss3D\(\)](#).

5.3.4.7 Ph0

```
sunrealtype Gauss3D::Ph0 [private]
```

center of beam Φ_0

Definition at line 211 of file [ICSetters.h](#).

Referenced by [addToSpace\(\)](#), and [Gauss3D\(\)](#).

5.3.4.8 PhA

```
sunrealtype Gauss3D::PhA [private]
```

length of beam Φ_A

Definition at line 213 of file [ICSetters.h](#).

Referenced by [addToSpace\(\)](#), and [Gauss3D\(\)](#).

5.3.4.9 phip

```
sunrealtype Gauss3D::pkip [private]
```

polarization rotation from TE-mode around propagation direction

Definition at line 203 of file [ICSetters.h](#).

Referenced by [Gauss3D\(\)](#).

5.3.4.10 w0

```
sunrealtype Gauss3D::w0 [private]
```

taille ω_0

Definition at line 207 of file [ICSetters.h](#).

Referenced by [addToSpace\(\)](#), and [Gauss3D\(\)](#).

5.3.4.11 zr

```
sunrealtype Gauss3D::zr [private]
```

Rayleigh length $z_R = \pi\omega_0^2/\lambda$.

Definition at line 209 of file [ICSetters.h](#).

Referenced by [addToSpace\(\)](#), and [Gauss3D\(\)](#).

The documentation for this class was generated from the following files:

- [src/ICSetters.h](#)
- [src/ICSetters.cpp](#)

5.4 gaussian1D Struct Reference

1D Gaussian wave structure

```
#include <src/SimulationFunctions.h>
```

Data Fields

- `std::array< sunrealtype, 3 > k`
- `std::array< sunrealtype, 3 > p`
- `std::array< sunrealtype, 3 > x0`
- `sunrealtype phig`
- `std::array< sunrealtype, 3 > phi`

5.4.1 Detailed Description

1D Gaussian wave structure

Definition at line 26 of file [SimulationFunctions.h](#).

5.4.2 Field Documentation

5.4.2.1 k

```
std::array<sunrealtype, 3> gaussian1D::k
```

wavevector (normalized to $1/\lambda$)

Definition at line 27 of file [SimulationFunctions.h](#).

5.4.2.2 p

```
std::array<sunrealtype, 3> gaussian1D::p
```

amplitude & polarization vector

Definition at line 28 of file [SimulationFunctions.h](#).

5.4.2.3 phi

```
std::array<sunrealtype, 3> gaussian1D::phi
```

phase shift

Definition at line 31 of file [SimulationFunctions.h](#).

5.4.2.4 phig

```
sunrealtype gaussian1D::phig
```

width

Definition at line 30 of file [SimulationFunctions.h](#).

5.4.2.5 x0

```
std::array<sunrealtype, 3> gaussian1D::x0
```

shift from origin

Definition at line 29 of file [SimulationFunctions.h](#).

The documentation for this struct was generated from the following file:

- [src/SimulationFunctions.h](#)

5.5 gaussian2D Struct Reference

2D Gaussian wave structure

```
#include <src/SimulationFunctions.h>
```

Data Fields

- `std::array< sunrealtype, 3 >` [x0](#)
- `std::array< sunrealtype, 3 >` [axis](#)
- `sunrealtype` [amp](#)
- `sunrealtype` [phip](#)
- `sunrealtype` [w0](#)
- `sunrealtype` [zr](#)
- `sunrealtype` [ph0](#)
- `sunrealtype` [phA](#)

5.5.1 Detailed Description

2D Gaussian wave structure

Definition at line 35 of file [SimulationFunctions.h](#).

5.5.2 Field Documentation

5.5.2.1 amp

```
sunrealtype gaussian2D::amp
```

amplitude

Definition at line 38 of file [SimulationFunctions.h](#).

5.5.2.2 axis

```
std::array<sunrealtype, 3> gaussian2D::axis
```

direction from where it comes

Definition at line 37 of file [SimulationFunctions.h](#).

5.5.2.3 ph0

```
sunrealtype gaussian2D::ph0
```

beam center

Definition at line 42 of file [SimulationFunctions.h](#).

5.5.2.4 phA

```
sunrealtype gaussian2D::phA
```

beam length

Definition at line 43 of file [SimulationFunctions.h](#).

5.5.2.5 phip

```
sunrealtype gaussian2D::phip
```

polarization rotation

Definition at line 39 of file [SimulationFunctions.h](#).

5.5.2.6 w0

```
sunrealtype gaussian2D::w0
```

taille

Definition at line 40 of file [SimulationFunctions.h](#).

5.5.2.7 x0

```
std::array<sunrealtype, 3> gaussian2D::x0
```

center

Definition at line 36 of file [SimulationFunctions.h](#).

5.5.2.8 zr

```
sunrealtype gaussian2D::zr
```

Rayleigh length

Definition at line 41 of file [SimulationFunctions.h](#).

The documentation for this struct was generated from the following file:

- [src/SimulationFunctions.h](#)

5.6 gaussian3D Struct Reference

3D Gaussian wave structure

```
#include <src/SimulationFunctions.h>
```

Data Fields

- `std::array< sunrealtype, 3 >` [x0](#)
- `std::array< sunrealtype, 3 >` [axis](#)
- `sunrealtype` [amp](#)
- `sunrealtype` [phip](#)
- `sunrealtype` [w0](#)
- `sunrealtype` [zr](#)
- `sunrealtype` [ph0](#)
- `sunrealtype` [phA](#)

5.6.1 Detailed Description

3D Gaussian wave structure

Definition at line 47 of file [SimulationFunctions.h](#).

5.6.2 Field Documentation

5.6.2.1 amp

```
sunrealtype gaussian3D::amp
```

amplitude

Definition at line 50 of file [SimulationFunctions.h](#).

5.6.2.2 axis

```
std::array<sunrealtype, 3> gaussian3D::axis
```

direction from where it comes

Definition at line 49 of file [SimulationFunctions.h](#).

5.6.2.3 ph0

```
sunrealtype gaussian3D::ph0
```

beam center

Definition at line 54 of file [SimulationFunctions.h](#).

5.6.2.4 phA

```
sunrealtype gaussian3D::phA
```

beam length

Definition at line 55 of file [SimulationFunctions.h](#).

5.6.2.5 phip

```
sunrealtype gaussian3D::phip
```

polarization rotation

Definition at line 51 of file [SimulationFunctions.h](#).

5.6.2.6 w0

```
sunrealtype gaussian3D::w0
```

taille

Definition at line 52 of file [SimulationFunctions.h](#).

5.6.2.7 x0

```
std::array<sunrealtype, 3> gaussian3D::x0
```

center

Definition at line 48 of file [SimulationFunctions.h](#).

5.6.2.8 zr

```
sunrealtype gaussian3D::zr
```

Rayleigh length

Definition at line 53 of file [SimulationFunctions.h](#).

The documentation for this struct was generated from the following file:

- [src/SimulationFunctions.h](#)

5.7 ICSetter Class Reference

[ICSetter](#) class to initialize wave types with default parameters.

```
#include <src/ICSetters.h>
```


Public Member Functions

- void [eval](#) (sunrealtype x, sunrealtype y, sunrealtype z, sunrealtype *pTo6Space)
function to set all coordinates to zero and then add the field values
- void [add](#) (sunrealtype x, sunrealtype y, sunrealtype z, sunrealtype *pTo6Space)
function to fill the lattice space with initial field values
- void [addPlaneWave1D](#) (std::array< sunrealtype, 3 > k={1, 0, 0}, std::array< sunrealtype, 3 > p={0, 0, 1}, std::array< sunrealtype, 3 > phi={0, 0, 0})
function to add plane waves in 1D to their container vector
- void [addPlaneWave2D](#) (std::array< sunrealtype, 3 > k={1, 0, 0}, std::array< sunrealtype, 3 > p={0, 0, 1}, std::array< sunrealtype, 3 > phi={0, 0, 0})
function to add plane waves in 2D to their container vector
- void [addPlaneWave3D](#) (std::array< sunrealtype, 3 > k={1, 0, 0}, std::array< sunrealtype, 3 > p={0, 0, 1}, std::array< sunrealtype, 3 > phi={0, 0, 0})
function to add plane waves in 3D to their container vector
- void [addGauss1D](#) (std::array< sunrealtype, 3 > k={1, 0, 0}, std::array< sunrealtype, 3 > p={0, 0, 1}, std::array< sunrealtype, 3 > xo={0, 0, 0}, sunrealtype phig_=1.0, std::array< sunrealtype, 3 > phi={0, 0, 0})
function to add Gaussian wave packets in 1D to their container vector
- void [addGauss2D](#) (std::array< sunrealtype, 3 > dis_={0, 0, 0}, std::array< sunrealtype, 3 > axis_={1, 0, 0}, sunrealtype Amp_=1.0, sunrealtype phip_=0, sunrealtype w0_=1e-5, sunrealtype zr_=4e-5, sunrealtype Ph0_=2e-5, sunrealtype PhA_=0.45e-5)
function to add Gaussian wave packets in 2D to their container vector
- void [addGauss3D](#) (std::array< sunrealtype, 3 > dis_={0, 0, 0}, std::array< sunrealtype, 3 > axis_={1, 0, 0}, sunrealtype Amp_=1.0, sunrealtype phip_=0, sunrealtype w0_=1e-5, sunrealtype zr_=4e-5, sunrealtype Ph0_=2e-5, sunrealtype PhA_=0.45e-5)
function to add Gaussian wave packets in 3D to their container vector

Private Attributes

- std::vector< [PlaneWave1D](#) > [planeWaves1D](#)
container vector for plane waves in 1D
- std::vector< [PlaneWave2D](#) > [planeWaves2D](#)
container vector for plane waves in 2D
- std::vector< [PlaneWave3D](#) > [planeWaves3D](#)
container vector for plane waves in 3D
- std::vector< [Gauss1D](#) > [gauss1Ds](#)
container vector for Gaussian wave packets in 1D
- std::vector< [Gauss2D](#) > [gauss2Ds](#)
container vector for Gaussian wave packets in 2D
- std::vector< [Gauss3D](#) > [gauss3Ds](#)
container vector for Gaussian wave packets in 3D

5.7.1 Detailed Description

[ICSetter](#) class to initialize wave types with default parameters.

Definition at line 238 of file [ICSetters.h](#).

5.7.2 Member Function Documentation

5.7.2.1 add()

```
void ICSetter::add (
    sunrealtype x,
    sunrealtype y,
    sunrealtype z,
    sunrealtype * pTo6Space )
```

function to fill the lattice space with initial field values

Add all initial field values to the lattice space

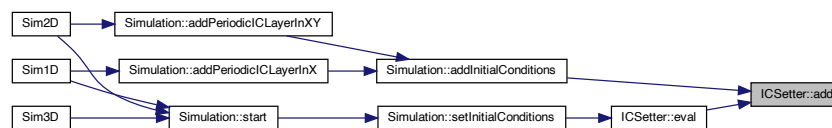
Definition at line 309 of file [ICSetters.cpp](#).

```
00310 {
00311     for (const auto &wave : planeWaves1D)
00312         wave.addToSpace(x, y, z, pTo6Space);
00313     for (const auto &wave : planeWaves2D)
00314         wave.addToSpace(x, y, z, pTo6Space);
00315     for (const auto &wave : planeWaves3D)
00316         wave.addToSpace(x, y, z, pTo6Space);
00317     for (const auto &wave : gauss1Ds)
00318         wave.addToSpace(x, y, z, pTo6Space);
00319     for (const auto &wave : gauss2Ds)
00320         wave.addToSpace(x, y, z, pTo6Space);
00321     for (const auto &wave : gauss3Ds)
00322         wave.addToSpace(x, y, z, pTo6Space);
00323 }
```

References [gauss1Ds](#), [gauss2Ds](#), [gauss3Ds](#), [planeWaves1D](#), [planeWaves2D](#), and [planeWaves3D](#).

Referenced by [Simulation::addInitialConditions\(\)](#), and [eval\(\)](#).

Here is the caller graph for this function:



5.7.2.2 addGauss1D()

```
void ICSetter::addGauss1D (
    std::array< sunrealtype, 3 > k = {1, 0, 0},
    std::array< sunrealtype, 3 > p = {0, 0, 1},
    std::array< sunrealtype, 3 > xo = {0, 0, 0},
    sunrealtype phig_ = 1.0,
    std::array< sunrealtype, 3 > phi = {0, 0, 0} )
```

function to add Gaussian wave packets in 1D to their container vector

Add Gaussian waves in 1D to their container vector

Definition at line 347 of file [ICSetters.cpp](#).

```
00350 {
00351     gauss1Ds.emplace_back(Gauss1D(k, p, xo, phig_, phi));
```

```
00352 }
```

References [gauss1Ds](#).

Referenced by [Sim1D\(\)](#).

Here is the caller graph for this function:



5.7.2.3 addGauss2D()

```

void ICSetter::addGauss2D (
    std::array< sunrealtype, 3 > dis_ = {0, 0, 0},
    std::array< sunrealtype, 3 > axis_ = {1, 0, 0},
    sunrealtype Amp_ = 1.0,
    sunrealtype phip_ = 0,
    sunrealtype w0_ = 1e-5,
    sunrealtype zr_ = 4e-5,
    sunrealtype Ph0_ = 2e-5,
    sunrealtype PhA_ = 0.45e-5 )
  
```

function to add Gaussian wave packets in 2D to their container vector

Add Gaussian waves in 2D to their container vector

Definition at line 355 of file [ICSetters.cpp](#).

```

00359 {
00360     gauss2Ds.emplace_back(
00361         Gauss2D(dis_, axis_, Amp_, phip_, w0_, zr_, Ph0_, PhA_));
00362 }
  
```

References [gauss2Ds](#).

Referenced by [Sim2D\(\)](#).

Here is the caller graph for this function:



5.7.2.4 addGauss3D()

```
void ICSetter::addGauss3D (
    std::array< sunrealtype, 3 > dis_ = {0, 0, 0},
    std::array< sunrealtype, 3 > axis_ = {1, 0, 0},
    sunrealtype Amp_ = 1.0,
    sunrealtype phip_ = 0,
    sunrealtype w0_ = 1e-5,
    sunrealtype zr_ = 4e-5,
    sunrealtype Ph0_ = 2e-5,
    sunrealtype PhA_ = 0.45e-5 )
```

function to add Gaussian wave packets in 3D to their container vector

Add Gaussian waves in 3D to their container vector

Definition at line 365 of file [ICSetters.cpp](#).

```
00369 {
00370     gauss3Ds.emplace_back(
00371         Gauss3D(dis_, axis_, Amp_, phip_, w0_, zr_, Ph0_, PhA_));
00372 }
```

References [gauss3Ds](#).

Referenced by [Sim3D\(\)](#).

Here is the caller graph for this function:



5.7.2.5 addPlaneWave1D()

```
void ICSetter::addPlaneWave1D (
    std::array< sunrealtype, 3 > k = {1, 0, 0},
    std::array< sunrealtype, 3 > p = {0, 0, 1},
    std::array< sunrealtype, 3 > phi = {0, 0, 0} )
```

function to add plane waves in 1D to their container vector

Add plane waves in 1D to their container vector

Definition at line 326 of file [ICSetters.cpp](#).

```
00328 {
00329     planeWaves1D.emplace_back(PlaneWave1D(k, p, phi));
00330 }
```

References [planeWaves1D](#).

Referenced by [Sim1D\(\)](#).

Here is the caller graph for this function:



5.7.2.6 addPlaneWave2D()

```
void ICSetter::addPlaneWave2D (
    std::array< sunrealtype, 3 > k = {1, 0, 0},
    std::array< sunrealtype, 3 > p = {0, 0, 1},
    std::array< sunrealtype, 3 > phi = {0, 0, 0} )
```

function to add plane waves in 2D to their container vector

Add plane waves in 2D to their container vector

Definition at line 333 of file [ICSetters.cpp](#).

```
00335     {
00336     planeWaves2D.emplace_back(PlaneWave2D(k, p, phi));
00337 }
```

References [planeWaves2D](#).

Referenced by [Sim2D\(\)](#).

Here is the caller graph for this function:



5.7.2.7 addPlaneWave3D()

```
void ICSetter::addPlaneWave3D (
    std::array< sunrealtype, 3 > k = {1, 0, 0},
    std::array< sunrealtype, 3 > p = {0, 0, 1},
    std::array< sunrealtype, 3 > phi = {0, 0, 0} )
```

function to add plane waves in 3D to their container vector

Add plane waves in 3D to their container vector

Definition at line 340 of file [ICSetters.cpp](#).

```
00342     {
00343     planeWaves3D.emplace_back(PlaneWave3D(k, p, phi));
00344 }
```

References [planeWaves3D](#).

Referenced by [Sim3D\(\)](#).

Here is the caller graph for this function:



5.7.2.8 eval()

```
void ICSetter::eval (
    sunrealtype x,
    sunrealtype y,
    sunrealtype z,
    sunrealtype * pTo6Space )
```

function to set all coordinates to zero and then add the field values

Evaluate lattice point values to zero and then add initial field values

Definition at line 297 of file [ICSetters.cpp](#).

```
00298     {
00299     pTo6Space[0] = 0;
00300     pTo6Space[1] = 0;
00301     pTo6Space[2] = 0;
00302     pTo6Space[3] = 0;
00303     pTo6Space[4] = 0;
00304     pTo6Space[5] = 0;
00305     add(x, y, z, pTo6Space);
00306 }
```

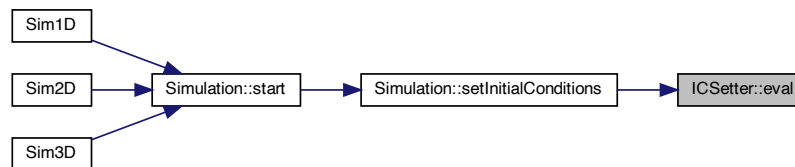
References [add\(\)](#).

Referenced by [Simulation::setInitialConditions\(\)](#).

Here is the call graph for this function:



Here is the caller graph for this function:



5.7.3 Field Documentation

5.7.3.1 gauss1Ds

```
std::vector<Gauss1D> ICSetter::gauss1Ds [private]
```

container vector for Gaussian wave packets in 1D

Definition at line 247 of file [ICSetters.h](#).

Referenced by [add\(\)](#), and [addGauss1D\(\)](#).

5.7.3.2 gauss2Ds

```
std::vector<Gauss2D> ICSetter::gauss2Ds [private]
```

container vector for Gaussian wave packets in 2D

Definition at line 249 of file [ICSetters.h](#).

Referenced by [add\(\)](#), and [addGauss2D\(\)](#).

5.7.3.3 gauss3Ds

```
std::vector<Gauss3D> ICSetter::gauss3Ds [private]
```

container vector for Gaussian wave packets in 3D

Definition at line 251 of file [ICSetters.h](#).

Referenced by [add\(\)](#), and [addGauss3D\(\)](#).

5.7.3.4 planeWaves1D

```
std::vector<PlaneWave1D> ICSetter::planeWaves1D [private]
```

container vector for plane waves in 1D

Definition at line 241 of file [ICSetters.h](#).

Referenced by [add\(\)](#), and [addPlaneWave1D\(\)](#).

5.7.3.5 planeWaves2D

```
std::vector<PlaneWave2D> ICSetter::planeWaves2D [private]
```

container vector for plane waves in 2D

Definition at line 243 of file [ICSetters.h](#).

Referenced by [add\(\)](#), and [addPlaneWave2D\(\)](#).

5.7.3.6 planeWaves3D

```
std::vector<PlaneWave3D> ICSetter::planeWaves3D [private]
```

container vector for plane waves in 3D

Definition at line 245 of file [ICSetters.h](#).

Referenced by [add\(\)](#), and [addPlaneWave3D\(\)](#).

The documentation for this class was generated from the following files:

- [src/ICSetters.h](#)
- [src/ICSetters.cpp](#)

5.8 Lattice Class Reference

[Lattice](#) class for the construction of the enveloping discrete simulation space.

```
#include <src/LatticePatch.h>
```

Public Member Functions

- void [initializeCommunicator](#) (const int Nx, const int Ny, const int Nz, const bool per)
function to create and deploy the cartesian communicator
- [Lattice](#) (const int StO)
default construction
- void [setDiscreteDimensions](#) (const sunindextype _nx, const sunindextype _ny, const sunindextype _nz)
component function for resizing the discrete dimensions of the lattice
- void [setPhysicalDimensions](#) (const sunrealtype _lx, const sunrealtype _ly, const sunrealtype _lz)
component function for resizing the physical size of the lattice
- const sunrealtype & [get_tot_lx](#) () const
- const sunrealtype & [get_tot_ly](#) () const
- const sunrealtype & [get_tot_lz](#) () const
- const sunindextype & [get_tot_nx](#) () const
- const sunindextype & [get_tot_ny](#) () const
- const sunindextype & [get_tot_nz](#) () const
- const sunindextype & [get_tot_noP](#) () const
- const sunindextype & [get_tot_noDP](#) () const
- const sunrealtype & [get_dx](#) () const
- const sunrealtype & [get_dy](#) () const
- const sunrealtype & [get_dz](#) () const
- constexpr int [get_dataPointDimension](#) () const
- const int & [get_stencilOrder](#) () const
- const int & [get_ghostLayerWidth](#) () const

Data Fields

- int [n_prc](#)
number of MPI processes
- int [my_prc](#)
number of MPI process
- MPI_Comm [comm](#)
personal communicator of the lattice
- SUNContext [sunctx](#)
SUNContext object.

Private Attributes

- sunrealtype [tot_lx](#)
physical size of the lattice in x-direction
- sunrealtype [tot_ly](#)
physical size of the lattice in y-direction
- sunrealtype [tot_lz](#)
physical size of the lattice in z-direction
- sunindextype [tot_nx](#)
number of points in x-direction
- sunindextype [tot_ny](#)
number of points in y-direction
- sunindextype [tot_nz](#)
number of points in z-direction
- sunindextype [tot_noP](#)
total number of lattice points
- sunindextype [tot_noDP](#)
number of lattice points times data dimension of each point
- sunrealtype [dx](#)
physical distance between lattice points in x-direction
- sunrealtype [dy](#)
physical distance between lattice points in y-direction
- sunrealtype [dz](#)
physical distance between lattice points in z-direction
- const int [stencilOrder](#)
stencil order
- const int [ghostLayerWidth](#)
required width of ghost layers (depends on the stencil order)
- unsigned int [statusFlags](#)
lattice status flags

Static Private Attributes

- static constexpr int [dataPointDimension](#) = 6
dimension of each data point set once and for all

5.8.1 Detailed Description

[Lattice](#) class for the construction of the enveloping discrete simulation space.

Definition at line 47 of file [LatticePatch.h](#).

5.8.2 Constructor & Destructor Documentation

5.8.2.1 Lattice()

```
Lattice::Lattice (
    const int StO )
```

default construction

Construct the lattice and set the stencil order.

Definition at line 39 of file [LatticePatch.cpp](#).

```
00039         : stencilOrder(StO),
00040     ghostLayerWidth(StO/2+1) {
00041     statusFlags = 0;
00042 }
```

References [statusFlags](#).

5.8.3 Member Function Documentation

5.8.3.1 get_dataPointDimension()

```
constexpr int Lattice::get_dataPointDimension ( ) const [inline], [constexpr]
```

getter function

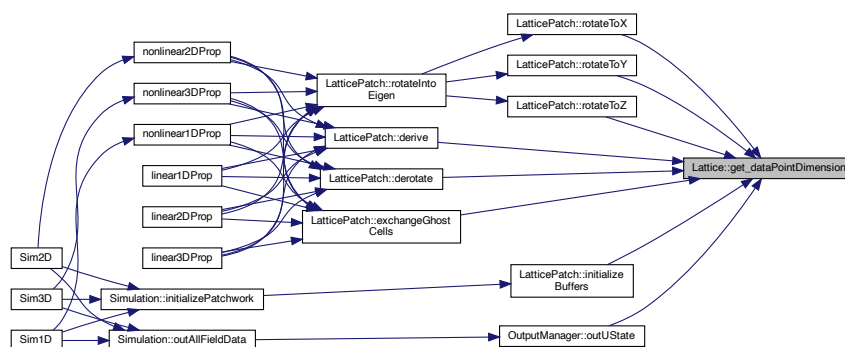
Definition at line 113 of file [LatticePatch.h](#).

```
00113                                     {
00114     return dataPointDimension;
00115 }
```

References [dataPointDimension](#).

Referenced by [LatticePatch::derive\(\)](#), [LatticePatch::derotate\(\)](#), [LatticePatch::exchangeGhostCells\(\)](#), [LatticePatch::initializeBuffers\(\)](#), [OutputManager::outUState\(\)](#), [LatticePatch::rotateToX\(\)](#), [LatticePatch::rotateToY\(\)](#), and [LatticePatch::rotateToZ\(\)](#).

Here is the caller graph for this function:



5.8.3.2 `get_dx()`

```
const sunrealtype & Lattice::get_dx ( ) const [inline]
```

getter function

Definition at line 110 of file [LatticePatch.h](#).

```
00110 { return dx; }
```

References [dx](#).

5.8.3.3 `get_dy()`

```
const sunrealtype & Lattice::get_dy ( ) const [inline]
```

getter function

Definition at line 111 of file [LatticePatch.h](#).

```
00111 { return dy; }
```

References [dy](#).

5.8.3.4 `get_dz()`

```
const sunrealtype & Lattice::get_dz ( ) const [inline]
```

getter function

Definition at line 112 of file [LatticePatch.h](#).

```
00112 { return dz; }
```

References [dz](#).

5.8.3.5 get_ghostLayerWidth()

```
const int & Lattice::get_ghostLayerWidth ( ) const [inline]
```

getter function

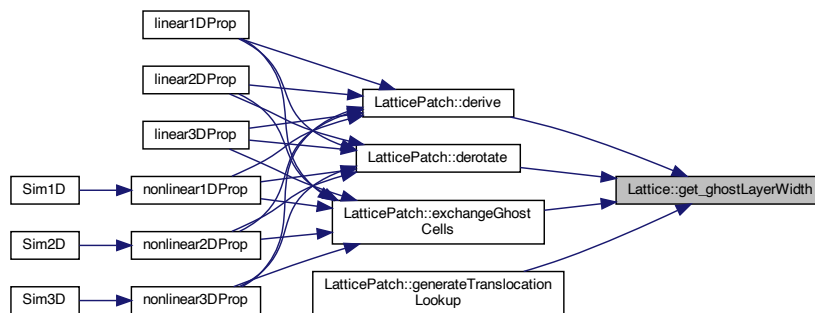
Definition at line 117 of file [LatticePatch.h](#).

```
00117 {
00118     return ghostLayerWidth;
00119 }
```

References [ghostLayerWidth](#).

Referenced by [LatticePatch::derive\(\)](#), [LatticePatch::derotate\(\)](#), [LatticePatch::exchangeGhostCells\(\)](#), and [LatticePatch::generateTranslocationLookup\(\)](#).

Here is the caller graph for this function:



5.8.3.6 get_stencilOrder()

```
const int & Lattice::get_stencilOrder ( ) const [inline]
```

getter function

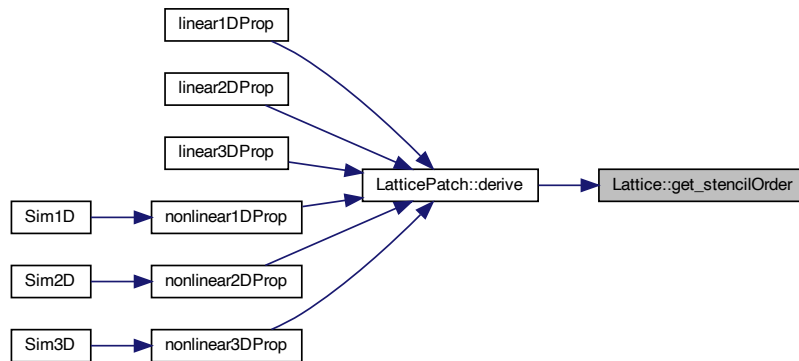
Definition at line 116 of file [LatticePatch.h](#).

```
00116 { return stencilOrder; }
```

References [stencilOrder](#).

Referenced by [LatticePatch::derive\(\)](#).

Here is the caller graph for this function:



5.8.3.7 `get_tot_lx()`

```
const sunrealtype & Lattice::get_tot_lx ( ) const [inline]
```

getter function

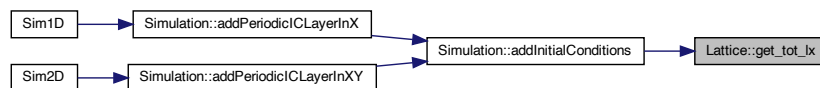
Definition at line 102 of file [LatticePatch.h](#).

```
00102 { return tot_lx; }
```

References [tot_lx](#).

Referenced by [Simulation::addInitialConditions\(\)](#).

Here is the caller graph for this function:



5.8.3.8 get_tot_ly()

```
const sunrealtype & Lattice::get_tot_ly ( ) const [inline]
```

getter function

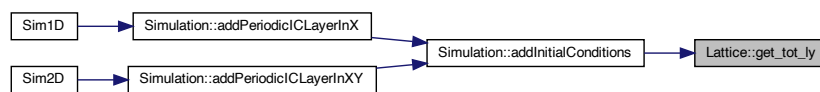
Definition at line 103 of file [LatticePatch.h](#).

```
00103 { return tot_ly; }
```

References [tot_ly](#).

Referenced by [Simulation::addInitialConditions\(\)](#).

Here is the caller graph for this function:



5.8.3.9 get_tot_lz()

```
const sunrealtype & Lattice::get_tot_lz ( ) const [inline]
```

getter function

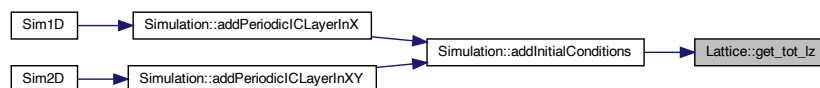
Definition at line 104 of file [LatticePatch.h](#).

```
00104 { return tot_lz; }
```

References [tot_lz](#).

Referenced by [Simulation::addInitialConditions\(\)](#).

Here is the caller graph for this function:



5.8.3.10 `get_tot_noDP()`

```
const sunindextype & Lattice::get_tot_noDP ( ) const [inline]
```

getter function

Definition at line 109 of file [LatticePatch.h](#).

```
00109 { return tot_noDP; }
```

References [tot_noDP](#).

5.8.3.11 `get_tot_noP()`

```
const sunindextype & Lattice::get_tot_noP ( ) const [inline]
```

getter function

Definition at line 108 of file [LatticePatch.h](#).

```
00108 { return tot_noP; }
```

References [tot_noP](#).

5.8.3.12 `get_tot_nx()`

```
const sunindextype & Lattice::get_tot_nx ( ) const [inline]
```

getter function

Definition at line 105 of file [LatticePatch.h](#).

```
00105 { return tot_nx; }
```

References [tot_nx](#).

5.8.3.13 `get_tot_ny()`

```
const sunindextype & Lattice::get_tot_ny ( ) const [inline]
```

getter function

Definition at line 106 of file [LatticePatch.h](#).

```
00106 { return tot_ny; }
```

References [tot_ny](#).

5.8.3.14 get_tot_nz()

```
const sunindextype & Lattice::get_tot_nz ( ) const [inline]
```

getter function

Definition at line 107 of file [LatticePatch.h](#).

```
00107 { return tot_nz; }
```

References [tot_nz](#).

5.8.3.15 initializeCommunicator()

```
void Lattice::initializeCommunicator (
    const int Nx,
    const int Ny,
    const int Nz,
    const bool per )
```

function to create and deploy the cartesian communicator

Initialize the cartesian communicator.

Definition at line 15 of file [LatticePatch.cpp](#).

```
00016 {
00017     const int dims[3] = {Nz, Ny, Nx};
00018     const int periods[3] = {static_cast<int>(per), static_cast<int>(per),
00019                             static_cast<int>(per)};
00020     // Create the cartesian communicator for MPI_COMM_WORLD
00021     MPI_Cart_create(MPI_COMM_WORLD, 3, dims, periods, 1, &comm);
00022     // Set MPI variables of the lattice
00023     MPI_Comm_size(comm, &(n_prc));
00024     MPI_Comm_rank(comm, &(my_prc));
00025     // Associate name to the communicator to identify it -> for debugging and
00026     // nicer error messages
00027     constexpr char lattice_comm_name[] = "Lattice";
00028     MPI_Comm_set_name(comm, lattice_comm_name);
00029
00030     // Test if process naming is the same for both communicators
00031     /*
00032     int myPrc;
00033     MPI_Comm_rank(MPI_COMM_WORLD, &myPrc);
00034     cout<<"\r"<<my_prc<<"\t"<<myPrc<<std::endl;
00035     */
00036 }
```

References [comm](#), [my_prc](#), and [n_prc](#).

Referenced by [Simulation::Simulation\(\)](#).

Here is the caller graph for this function:



5.8.3.16 setDiscreteDimensions()

```
void Lattice::setDiscreteDimensions (
    const sunindextype _nx,
    const sunindextype _ny,
    const sunindextype _nz )
```

component function for resizing the discrete dimensions of the lattice

Set the number of points in each dimension of the lattice.

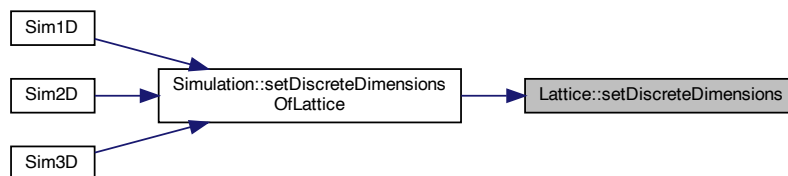
Definition at line 45 of file [LatticePatch.cpp](#).

```
00046 {
00047     // copy the given data for number of points
00048     tot_nx = _nx;
00049     tot_ny = _ny;
00050     tot_nz = _nz;
00051     // compute the resulting number of points and datapoints
00052     tot_noP = tot_nx * tot_ny * tot_nz;
00053     tot_noDP = dataPointDimension * tot_noP;
00054     // compute the new Delta, the physical resolution
00055     dx = tot_lx / tot_nx;
00056     dy = tot_ly / tot_ny;
00057     dz = tot_lz / tot_nz;
00058 }
```

References [dataPointDimension](#), [dx](#), [dy](#), [dz](#), [tot_lx](#), [tot_ly](#), [tot_lz](#), [tot_noDP](#), [tot_noP](#), [tot_nx](#), [tot_ny](#), and [tot_nz](#).

Referenced by [Simulation::setDiscreteDimensionsOfLattice\(\)](#).

Here is the caller graph for this function:



5.8.3.17 setPhysicalDimensions()

```
void Lattice::setPhysicalDimensions (
    const sunrealtype _lx,
    const sunrealtype _ly,
    const sunrealtype _lz )
```

component function for resizing the physical size of the lattice

Set the physical size of the lattice.

Definition at line 61 of file [LatticePatch.cpp](#).

```
00062 {
00063     tot_lx = _lx;
00064     tot_ly = _ly;
```

```

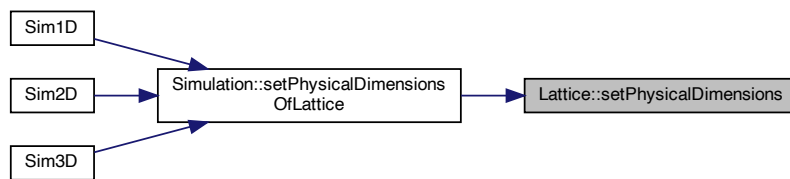
00065     tot_lz = _lz;
00066     // calculate physical distance between points
00067     dx = tot_lx / tot_nx;
00068     dy = tot_ly / tot_ny;
00069     dz = tot_lz / tot_nz;
00070     statusFlags |= FLatticeDimensionSet;
00071 }

```

References [dx](#), [dy](#), [dz](#), [FLatticeDimensionSet](#), [statusFlags](#), [tot_lx](#), [tot_ly](#), [tot_lz](#), [tot_nx](#), [tot_ny](#), and [tot_nz](#).

Referenced by [Simulation::setPhysicalDimensionsOfLattice\(\)](#).

Here is the caller graph for this function:



5.8.4 Field Documentation

5.8.4.1 comm

```
MPI_Comm Lattice::comm
```

personal communicator of the lattice

Definition at line 86 of file [LatticePatch.h](#).

Referenced by [Simulation::advanceToTime\(\)](#), [LatticePatch::exchangeGhostCells\(\)](#), [Simulation::get_cart_comm\(\)](#), [initializeCommunicator\(\)](#), [Simulation::initializeCVODEobject\(\)](#), [OutputManager::outUState\(\)](#), and [Simulation::Simulation\(\)](#).

5.8.4.2 dataPointDimension

```
constexpr int Lattice::dataPointDimension = 6 [static], [constexpr], [private]
```

dimension of each data point set once and for all

Definition at line 64 of file [LatticePatch.h](#).

Referenced by [get_dataPointDimension\(\)](#), and [setDiscreteDimensions\(\)](#).

5.8.4.3 dx

```
sunrealtype Lattice::dx [private]
```

physical distance between lattice points in x-direction

Definition at line 68 of file [LatticePatch.h](#).

Referenced by [get_dx\(\)](#), [setDiscreteDimensions\(\)](#), and [setPhysicalDimensions\(\)](#).

5.8.4.4 dy

```
sunrealtype Lattice::dy [private]
```

physical distance between lattice points in y-direction

Definition at line 70 of file [LatticePatch.h](#).

Referenced by [get_dy\(\)](#), [setDiscreteDimensions\(\)](#), and [setPhysicalDimensions\(\)](#).

5.8.4.5 dz

```
sunrealtype Lattice::dz [private]
```

physical distance between lattice points in z-direction

Definition at line 72 of file [LatticePatch.h](#).

Referenced by [get_dz\(\)](#), [setDiscreteDimensions\(\)](#), and [setPhysicalDimensions\(\)](#).

5.8.4.6 ghostLayerWidth

```
const int Lattice::ghostLayerWidth [private]
```

required width of ghost layers (depends on the stencil order)

Definition at line 76 of file [LatticePatch.h](#).

Referenced by [get_ghostLayerWidth\(\)](#).

5.8.4.7 my_prc

```
int Lattice::my_prc
```

number of MPI process

Definition at line 84 of file [LatticePatch.h](#).

Referenced by [Simulation::advanceToTime\(\)](#), [initializeCommunicator\(\)](#), [Simulation::initializeCVODEobject\(\)](#), [OutputManager::outUState\(\)](#), and [Simulation::Simulation\(\)](#).

5.8.4.8 n_prc

```
int Lattice::n_prc
```

number of MPI processes

Definition at line 82 of file [LatticePatch.h](#).

Referenced by [initializeCommunicator\(\)](#).

5.8.4.9 statusFlags

```
unsigned int Lattice::statusFlags [private]
```

lattice status flags

Definition at line 78 of file [LatticePatch.h](#).

Referenced by [Lattice\(\)](#), and [setPhysicalDimensions\(\)](#).

5.8.4.10 stencilOrder

```
const int Lattice::stencilOrder [private]
```

stencil order

Definition at line 74 of file [LatticePatch.h](#).

Referenced by [get_stencilOrder\(\)](#).

5.8.4.11 `sunctx`

```
SUNContext Lattice::sunctx
```

SUNContext object.

Definition at line 93 of file [LatticePatch.h](#).

Referenced by [Simulation::initializeCVODEobject\(\)](#), [Simulation::Simulation\(\)](#), and [Simulation::~~Simulation\(\)](#).

5.8.4.12 `tot_lx`

```
sunrealtype Lattice::tot_lx [private]
```

physical size of the lattice in x-direction

Definition at line 50 of file [LatticePatch.h](#).

Referenced by [get_tot_lx\(\)](#), [setDiscreteDimensions\(\)](#), and [setPhysicalDimensions\(\)](#).

5.8.4.13 `tot_ly`

```
sunrealtype Lattice::tot_ly [private]
```

physical size of the lattice in y-direction

Definition at line 52 of file [LatticePatch.h](#).

Referenced by [get_tot_ly\(\)](#), [setDiscreteDimensions\(\)](#), and [setPhysicalDimensions\(\)](#).

5.8.4.14 `tot_lz`

```
sunrealtype Lattice::tot_lz [private]
```

physical size of the lattice in z-direction

Definition at line 54 of file [LatticePatch.h](#).

Referenced by [get_tot_lz\(\)](#), [setDiscreteDimensions\(\)](#), and [setPhysicalDimensions\(\)](#).

5.8.4.15 tot_noDP

```
sunindextype Lattice::tot_noDP [private]
```

number of lattice points times data dimension of each point

Definition at line 66 of file [LatticePatch.h](#).

Referenced by [get_tot_noDP\(\)](#), and [setDiscreteDimensions\(\)](#).

5.8.4.16 tot_noP

```
sunindextype Lattice::tot_noP [private]
```

total number of lattice points

Definition at line 62 of file [LatticePatch.h](#).

Referenced by [get_tot_noP\(\)](#), and [setDiscreteDimensions\(\)](#).

5.8.4.17 tot_nx

```
sunindextype Lattice::tot_nx [private]
```

number of points in x-direction

Definition at line 56 of file [LatticePatch.h](#).

Referenced by [get_tot_nx\(\)](#), [setDiscreteDimensions\(\)](#), and [setPhysicalDimensions\(\)](#).

5.8.4.18 tot_ny

```
sunindextype Lattice::tot_ny [private]
```

number of points in y-direction

Definition at line 58 of file [LatticePatch.h](#).

Referenced by [get_tot_ny\(\)](#), [setDiscreteDimensions\(\)](#), and [setPhysicalDimensions\(\)](#).

5.8.4.19 tot_nz

```
sunindextype Lattice::tot_nz [private]
```

number of points in z-direction

Definition at line 60 of file [LatticePatch.h](#).

Referenced by [get_tot_nz\(\)](#), [setDiscreteDimensions\(\)](#), and [setPhysicalDimensions\(\)](#).

The documentation for this class was generated from the following files:

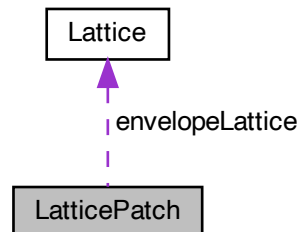
- [src/LatticePatch.h](#)
- [src/LatticePatch.cpp](#)

5.9 LatticePatch Class Reference

[LatticePatch](#) class for the construction of the patches in the enveloping lattice.

```
#include <src/LatticePatch.h>
```

Collaboration diagram for LatticePatch:



Public Member Functions

- [LatticePatch](#) ()
constructor setting up a default first lattice patch
- [~LatticePatch](#) ()
destructor freeing parallel vectors
- sunindextype [discreteSize](#) (int dir=0) const
function to get the discrete size of the [LatticePatch](#)
- sunrealtype [origin](#) (const int dir) const
function to get the origin of the patch
- sunrealtype [getDelta](#) (const int dir) const
function to get distance between points
- void [generateTranslocationLookup](#) ()

- function to fill out the lookup tables for cache efficiency*
- void [rotateIntoEigen](#) (const int dir)
function to rotate u into Z-matrix eigenraum
- void [derotate](#) (int dir, sunrealtype *buffOut)
function to derotate uAux into dudata lattice direction of x
- void [initializeBuffers](#) ()
initialize buffers to save derivatives
- void [exchangeGhostCells](#) (const int dir)
function to exchange ghost cells
- void [derive](#) (const int dir)
function to derive the centered values in uAux and save them noncentered
- void [checkFlag](#) (unsigned int flag) const
function to check if a flag has been set and if not abort

Data Fields

- int [ID](#)
ID of the [LatticePatch](#), corresponds to process number (for debugging)
- N_Vector [u](#)
N_Vector for saving field components $u=(E,B)$ in lattice points.
- N_Vector [du](#)
N_Vector for saving temporal derivatives of the field data.
- sunrealtype * [uData](#)
pointer to field data
- sunrealtype * [uAuxData](#)
pointer to auxiliary data vector
- sunrealtype * [duData](#)
pointer to time-derivative data
- std::array< sunrealtype *, 3 > [buffData](#)

- sunrealtype * [gCLData](#)
- sunrealtype * [gCRData](#)

Private Member Functions

- void [rotateToX](#) (sunrealtype *outArray, const sunrealtype *inArray, const std::vector< sunindextype > &lookup)
- void [rotateToY](#) (sunrealtype *outArray, const sunrealtype *inArray, const std::vector< sunindextype > &lookup)
- void [rotateToZ](#) (sunrealtype *outArray, const sunrealtype *inArray, const std::vector< sunindextype > &lookup)

Private Attributes

- sunrealtype [x0](#)
origin of the patch in physical space; x-coordinate
 - sunrealtype [y0](#)
origin of the patch in physical space; y-coordinate
 - sunrealtype [z0](#)
origin of the patch in physical space; z-coordinate
 - sunindextype [Llx](#)
inner position of lattice-patch in the lattice patchwork; x-points
 - sunindextype [Lly](#)
inner position of lattice-patch in the lattice patchwork; y-points
 - sunindextype [Llz](#)
inner position of lattice-patch in the lattice patchwork; z-points
 - sunrealtype [lx](#)
physical size of the lattice-patch in the x-dimension
 - sunrealtype [ly](#)
physical size of the lattice-patch in the y-dimension
 - sunrealtype [lz](#)
physical size of the lattice-patch in the z-dimension
 - sunindextype [nx](#)
number of points in the lattice patch in the x-dimension
 - sunindextype [ny](#)
number of points in the lattice patch in the y-dimension
 - sunindextype [nz](#)
number of points in the lattice patch in the z-dimension
 - sunrealtype [dx](#)
physical distance between lattice points in x-direction
 - sunrealtype [dy](#)
physical distance between lattice points in y-direction
 - sunrealtype [dz](#)
physical distance between lattice points in z-direction
 - unsigned int [statusFlags](#)
lattice patch status flags
 - const [Lattice](#) * [envelopeLattice](#)
pointer to the enveloping lattice
 - std::vector< sunrealtype > [uAux](#)
aid (auxiliary) vector including ghost cells to compute the derivatives
-
- std::vector< sunindextype > [uTox](#)
 - std::vector< sunindextype > [uToy](#)
 - std::vector< sunindextype > [uToz](#)
 - std::vector< sunindextype > [xTou](#)
 - std::vector< sunindextype > [yTou](#)
 - std::vector< sunindextype > [zTou](#)
-
- std::vector< sunrealtype > [buffX](#)

- `std::vector< sunrealtype > buffY`
- `std::vector< sunrealtype > buffZ`

- `std::vector< sunrealtype > ghostCellLeft`
- `std::vector< sunrealtype > ghostCellRight`
- `std::vector< sunrealtype > ghostCellLeftToSend`
- `std::vector< sunrealtype > ghostCellRightToSend`
- `std::vector< sunrealtype > ghostCellsToSend`
- `std::vector< sunrealtype > ghostCells`

- `std::vector< sunindextype > lgcTox`
- `std::vector< sunindextype > rgcTox`
- `std::vector< sunindextype > lgcToy`
- `std::vector< sunindextype > rgcToy`
- `std::vector< sunindextype > lgcToz`
- `std::vector< sunindextype > rgcToz`

Friends

- `int generatePatchwork` (const [Lattice](#) &envelopeLattice, [LatticePatch](#) &patchToMold, const int DLx, const int DLy, const int DLz)
friend function for creating the patchwork slicing of the overall lattice

5.9.1 Detailed Description

[LatticePatch](#) class for the construction of the patches in the enveloping lattice.

Definition at line 125 of file [LatticePatch.h](#).

5.9.2 Constructor & Destructor Documentation

5.9.2.1 LatticePatch()

```
LatticePatch::LatticePatch ( )
```

constructor setting up a default first lattice patch

Construct the lattice patch.

Definition at line 78 of file [LatticePatch.cpp](#).

```
00078     {
00079     // set default origin coordinates to (0,0,0)
00080     x0 = y0 = z0 = 0;
00081     // set default position in Lattice-Patchwork to (0,0,0)
00082     LIx = LIy = LIz = 0;
00083     // set default physical length for lattice patch to (0,0,0)
00084     lx = ly = lz = 0;
00085     // set default discrete length for lattice patch to (0,1,1)
00086     /* This is done in this manner as even in 1D simulations require a 1 point
00087      * width */
00088     nx = 0;
00089     ny = nz = 1;
00090
00091     // u is not initialized as it wouldn't make any sense before the dimensions
00092     // are set idem for the enveloping lattice
00093
00094     // set default statusFlags to non set
00095     statusFlags = 0;
00096 }
```

References [LIx](#), [LIy](#), [LIz](#), [lx](#), [ly](#), [lz](#), [nx](#), [ny](#), [nz](#), [statusFlags](#), [x0](#), [y0](#), and [z0](#).

5.9.2.2 ~LatticePatch()

```
LatticePatch::~LatticePatch ( )
```

destructor freeing parallel vectors

Destruct the patch and thereby destroy the NVectors.

Definition at line 99 of file [LatticePatch.cpp](#).

```
00099     {
00100     // Deallocate memory for solution vector
00101     if (statusFlags & FLatticePatchSetUp) {
00102         // Destroy data vectors
00103         N_VDestroy_Parallel(u);
00104         N_VDestroy_Parallel(du);
00105     }
00106 }
```

References [du](#), [FLatticePatchSetUp](#), [statusFlags](#), and [u](#).

5.9.3 Member Function Documentation

5.9.3.1 checkFlag()

```
void LatticePatch::checkFlag (
    unsigned int flag ) const
```

function to check if a flag has been set and if not abort

Check if all flags are set.

Definition at line 615 of file [LatticePatch.cpp](#).

```
00615                                     {
00616     if (!(statusFlags & flag)) {
00617         std::string errorMessage;
00618         switch (flag) {
00619             case FLatticePatchSetUp:
00620                 errorMessage = "The Lattice patch was not set up please make sure to "
00621                     "initilize a Lattice topology";
00622                 break;
00623             case TranslocationLookupSetUp:
00624                 errorMessage = "The translocation lookup tables have not been generated, "
00625                     "please be sure to run generateTranslocationLookup()";
00626                 break;
00627             case GhostLayersInitialized:
00628                 errorMessage = "The space for the ghost layers has not been allocated, "
00629                     "please be sure that the ghost cells are initialized ";
00630                 break;
00631             case BuffersInitialized:
00632                 errorMessage = "The space for the buffers has not been allocated, please "
00633                     "be sure to run initializeBuffers()";
00634                 break;
00635             default:
00636                 errorMessage = "Uppss, you've made a non-standard error, sadly I can't "
00637                     "help you there";
00638                 break;
00639         }
00640         errorKill(errorMessage);
00641     }
00642     return;
00643 }
```

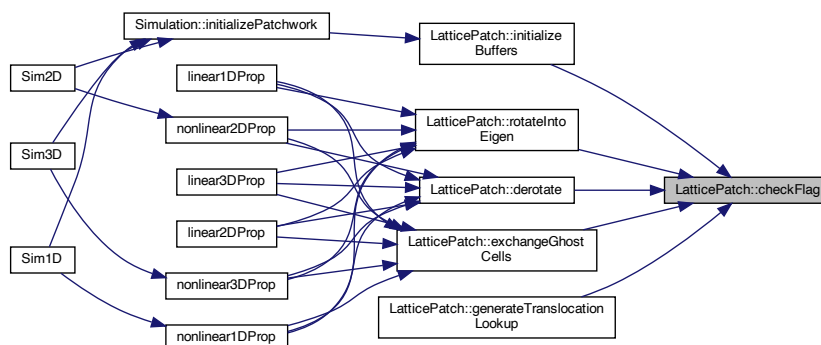
References [BuffersInitialized](#), [errorKill\(\)](#), [FLatticePatchSetUp](#), [GhostLayersInitialized](#), [statusFlags](#), and [TranslocationLookupSetUp](#).

Referenced by [derotate\(\)](#), [exchangeGhostCells\(\)](#), [generateTranslocationLookup\(\)](#), [initializeBuffers\(\)](#), and [rotateIntoEigen\(\)](#).

Here is the call graph for this function:



Here is the caller graph for this function:



5.9.3.2 derive()

```
void LatticePatch::derive (
    const int dir )
```

function to derive the centered values in uAux and save them noncentered

Calculate derivatives in the patch (uAux) in the specified direction.

Definition at line 646 of file [LatticePatch.cpp](#).

```

00646 {
00647     // ghost layer width adjusted to the chosen stencil order
00648     const int gLW = envelopeLattice->get_ghostLayerWidth();
00649     // dimensionality of data points -> 6
00650     const int dPD = envelopeLattice->get_dataPointDimension();
00651     // total width of patch in given direction including ghost layers at ends
00652     const sunindextype dirWidth = discreteSize(dir) + 2 * gLW;
00653     // width of patch only in given direction
00654     const sunindextype dirWidth0 = discreteSize(dir);
00655     // size of plane perpendicular to given dimension
00656     const sunindextype perpPlainSize = discreteSize() / discreteSize(dir);
00657     // physical distance between points in that direction
00658     sunrealtype dxi = nan("0x12345");
00659     switch (dir) {
00660     case 1:
00661         dxi = dx;
00662         break;
00663     case 2:
00664         dxi = dy;
00665         break;
00666     case 3:
00667         dxi = dz;
00668         break;
00669     default:
00670         dxi = 1;
00671         errorKill("Tried to derive in the wrong direction");
00672         break;
00673     }
00674     // Derive according to chosen stencil accuracy order
00675     const int order = envelopeLattice->get_stencilOrder();
00676     switch (order) {
00677     case 1: // gLW=1
00678         #pragma omp parallel for default(none) \
00679             shared(perpPlainSize, dxi, dirWidth, dirWidth0, gLW, dPD, uAux)
00680         for (sunindextype i = 0; i < perpPlainSize; i++) {
00681             #pragma omp simd
00682             for (sunindextype j = (i * dirWidth + gLW) * dPD;
00683                 j < (i * dirWidth + gLW + dirWidth0) * dPD; j += dPD) {
00684                 uAux[j + 0 - gLW * dPD] = slb(&uAux[j + 0]) / dxi;

```

```

00685         uAux[j + 1 - gLW * dPD] = s1b(&uAux[j + 1]) / dxi;
00686         uAux[j + 2 - gLW * dPD] = s1f(&uAux[j + 2]) / dxi;
00687         uAux[j + 3 - gLW * dPD] = s1f(&uAux[j + 3]) / dxi;
00688         uAux[j + 4 - gLW * dPD] = s1f(&uAux[j + 4]) / dxi;
00689         uAux[j + 5 - gLW * dPD] = s1f(&uAux[j + 5]) / dxi;
00690     }
00691 }
00692 break;
00693 case 2: // gLW=2
00694     #pragma omp parallel for default(none) \
00695     shared(perpPlainSize, dxi, dirWidth, dirWidth0, gLW, dPD, uAux)
00696     for (sunindextype i = 0; i < perpPlainSize; i++) {
00697         #pragma omp simd
00698         for (sunindextype j = (i * dirWidth + gLW) * dPD;
00699             j < (i * dirWidth + gLW + dirWidth0) * dPD; j += dPD) {
00700             uAux[j + 0 - gLW * dPD] = s2b(&uAux[j + 0]) / dxi;
00701             uAux[j + 1 - gLW * dPD] = s2b(&uAux[j + 1]) / dxi;
00702             uAux[j + 2 - gLW * dPD] = s2f(&uAux[j + 2]) / dxi;
00703             uAux[j + 3 - gLW * dPD] = s2f(&uAux[j + 3]) / dxi;
00704             uAux[j + 4 - gLW * dPD] = s2c(&uAux[j + 4]) / dxi;
00705             uAux[j + 5 - gLW * dPD] = s2c(&uAux[j + 5]) / dxi;
00706         }
00707     }
00708 break;
00709 case 3: // gLW=2
00710     #pragma omp parallel for default(none) \
00711     shared(perpPlainSize, dxi, dirWidth, dirWidth0, gLW, dPD, uAux)
00712     for (sunindextype i = 0; i < perpPlainSize; i++) {
00713         #pragma omp simd
00714         for (sunindextype j = (i * dirWidth + gLW) * dPD;
00715             j < (i * dirWidth + gLW + dirWidth0) * dPD; j += dPD) {
00716             uAux[j + 0 - gLW * dPD] = s3b(&uAux[j + 0]) / dxi;
00717             uAux[j + 1 - gLW * dPD] = s3b(&uAux[j + 1]) / dxi;
00718             uAux[j + 2 - gLW * dPD] = s3f(&uAux[j + 2]) / dxi;
00719             uAux[j + 3 - gLW * dPD] = s3f(&uAux[j + 3]) / dxi;
00720             uAux[j + 4 - gLW * dPD] = s3f(&uAux[j + 4]) / dxi;
00721             uAux[j + 5 - gLW * dPD] = s3f(&uAux[j + 5]) / dxi;
00722         }
00723     }
00724 break;
00725 case 4: // gLW=3
00726     #pragma omp parallel for default(none) \
00727     shared(perpPlainSize, dxi, dirWidth, dirWidth0, gLW, dPD, uAux)
00728     for (sunindextype i = 0; i < perpPlainSize; i++) {
00729         #pragma omp simd
00730         for (sunindextype j = (i * dirWidth + gLW) * dPD;
00731             j < (i * dirWidth + gLW + dirWidth0) * dPD; j += dPD) {
00732             uAux[j + 0 - gLW * dPD] = s4b(&uAux[j + 0]) / dxi;
00733             uAux[j + 1 - gLW * dPD] = s4b(&uAux[j + 1]) / dxi;
00734             uAux[j + 2 - gLW * dPD] = s4f(&uAux[j + 2]) / dxi;
00735             uAux[j + 3 - gLW * dPD] = s4f(&uAux[j + 3]) / dxi;
00736             uAux[j + 4 - gLW * dPD] = s4c(&uAux[j + 4]) / dxi;
00737             uAux[j + 5 - gLW * dPD] = s4c(&uAux[j + 5]) / dxi;
00738         }
00739     }
00740 break;
00741 case 5: // gLW=3
00742     #pragma omp parallel for default(none) \
00743     shared(perpPlainSize, dxi, dirWidth, dirWidth0, gLW, dPD, uAux)
00744     for (sunindextype i = 0; i < perpPlainSize; i++) {
00745         #pragma omp simd
00746         for (sunindextype j = (i * dirWidth + gLW) * dPD;
00747             j < (i * dirWidth + gLW + dirWidth0) * dPD; j += dPD) {
00748             uAux[j + 0 - gLW * dPD] = s5b(&uAux[j + 0]) / dxi;
00749             uAux[j + 1 - gLW * dPD] = s5b(&uAux[j + 1]) / dxi;
00750             uAux[j + 2 - gLW * dPD] = s5f(&uAux[j + 2]) / dxi;
00751             uAux[j + 3 - gLW * dPD] = s5f(&uAux[j + 3]) / dxi;
00752             uAux[j + 4 - gLW * dPD] = s5f(&uAux[j + 4]) / dxi;
00753             uAux[j + 5 - gLW * dPD] = s5f(&uAux[j + 5]) / dxi;
00754         }
00755     }
00756 break;
00757 case 6: // gLW=4
00758     #pragma omp parallel for default(none) \
00759     shared(perpPlainSize, dxi, dirWidth, dirWidth0, gLW, dPD, uAux)
00760     for (sunindextype i = 0; i < perpPlainSize; i++) {
00761         #pragma omp simd
00762         for (sunindextype j = (i * dirWidth + gLW) * dPD;
00763             j < (i * dirWidth + gLW + dirWidth0) * dPD; j += dPD) {
00764             uAux[j + 0 - gLW * dPD] = s6b(&uAux[j + 0]) / dxi;
00765             uAux[j + 1 - gLW * dPD] = s6b(&uAux[j + 1]) / dxi;
00766             uAux[j + 2 - gLW * dPD] = s6f(&uAux[j + 2]) / dxi;
00767             uAux[j + 3 - gLW * dPD] = s6f(&uAux[j + 3]) / dxi;
00768             uAux[j + 4 - gLW * dPD] = s6c(&uAux[j + 4]) / dxi;
00769             uAux[j + 5 - gLW * dPD] = s6c(&uAux[j + 5]) / dxi;
00770         }
00771     }

```

```

00772     break;
00773 case 7: // gLW=4
00774     #pragma omp parallel for default(none) \
00775     shared(perpPlainSize, dxi, dirWidth, dirWidthO, gLW, dPD, uAux)
00776     for (sunindextype i = 0; i < perpPlainSize; i++) {
00777         #pragma omp simd
00778         for (sunindextype j = (i * dirWidth + gLW) * dPD;
00779             j < (i * dirWidth + gLW + dirWidthO) * dPD; j += dPD) {
00780             uAux[j + 0 - gLW * dPD] = s7b(&uAux[j + 0]) / dxi;
00781             uAux[j + 1 - gLW * dPD] = s7b(&uAux[j + 1]) / dxi;
00782             uAux[j + 2 - gLW * dPD] = s7f(&uAux[j + 2]) / dxi;
00783             uAux[j + 3 - gLW * dPD] = s7f(&uAux[j + 3]) / dxi;
00784             uAux[j + 4 - gLW * dPD] = s7f(&uAux[j + 4]) / dxi;
00785             uAux[j + 5 - gLW * dPD] = s7f(&uAux[j + 5]) / dxi;
00786         }
00787     }
00788     break;
00789 case 8: // gLW=5
00790     #pragma omp parallel for default(none) \
00791     shared(perpPlainSize, dxi, dirWidth, dirWidthO, gLW, dPD, uAux)
00792     for (sunindextype i = 0; i < perpPlainSize; i++) {
00793         #pragma omp simd
00794         for (sunindextype j = (i * dirWidth + gLW) * dPD;
00795             j < (i * dirWidth + gLW + dirWidthO) * dPD; j += dPD) {
00796             uAux[j + 0 - gLW * dPD] = s8b(&uAux[j + 0]) / dxi;
00797             uAux[j + 1 - gLW * dPD] = s8b(&uAux[j + 1]) / dxi;
00798             uAux[j + 2 - gLW * dPD] = s8f(&uAux[j + 2]) / dxi;
00799             uAux[j + 3 - gLW * dPD] = s8f(&uAux[j + 3]) / dxi;
00800             uAux[j + 4 - gLW * dPD] = s8c(&uAux[j + 4]) / dxi;
00801             uAux[j + 5 - gLW * dPD] = s8c(&uAux[j + 5]) / dxi;
00802         }
00803     }
00804     break;
00805 case 9: // gLW=5
00806     #pragma omp parallel for default(none) \
00807     shared(perpPlainSize, dxi, dirWidth, dirWidthO, gLW, dPD, uAux)
00808     for (sunindextype i = 0; i < perpPlainSize; i++) {
00809         #pragma omp simd
00810         for (sunindextype j = (i * dirWidth + gLW) * dPD;
00811             j < (i * dirWidth + gLW + dirWidthO) * dPD; j += dPD) {
00812             uAux[j + 0 - gLW * dPD] = s9b(&uAux[j + 0]) / dxi;
00813             uAux[j + 1 - gLW * dPD] = s9b(&uAux[j + 1]) / dxi;
00814             uAux[j + 2 - gLW * dPD] = s9f(&uAux[j + 2]) / dxi;
00815             uAux[j + 3 - gLW * dPD] = s9f(&uAux[j + 3]) / dxi;
00816             uAux[j + 4 - gLW * dPD] = s9f(&uAux[j + 4]) / dxi;
00817             uAux[j + 5 - gLW * dPD] = s9f(&uAux[j + 5]) / dxi;
00818         }
00819     }
00820     break;
00821 case 10: // gLW=6
00822     #pragma omp parallel for default(none) \
00823     shared(perpPlainSize, dxi, dirWidth, dirWidthO, gLW, dPD, uAux)
00824     for (sunindextype i = 0; i < perpPlainSize; i++) {
00825         #pragma omp simd
00826         for (sunindextype j = (i * dirWidth + gLW) * dPD;
00827             j < (i * dirWidth + gLW + dirWidthO) * dPD; j += dPD) {
00828             uAux[j + 0 - gLW * dPD] = s10b(&uAux[j + 0]) / dxi;
00829             uAux[j + 1 - gLW * dPD] = s10b(&uAux[j + 1]) / dxi;
00830             uAux[j + 2 - gLW * dPD] = s10f(&uAux[j + 2]) / dxi;
00831             uAux[j + 3 - gLW * dPD] = s10f(&uAux[j + 3]) / dxi;
00832             uAux[j + 4 - gLW * dPD] = s10c(&uAux[j + 4]) / dxi;
00833             uAux[j + 5 - gLW * dPD] = s10c(&uAux[j + 5]) / dxi;
00834         }
00835     }
00836     break;
00837 case 11: // gLW=6
00838     #pragma omp parallel for default(none) \
00839     shared(perpPlainSize, dxi, dirWidth, dirWidthO, gLW, dPD, uAux)
00840     for (sunindextype i = 0; i < perpPlainSize; i++) {
00841         #pragma omp simd
00842         for (sunindextype j = (i * dirWidth + gLW) * dPD;
00843             j < (i * dirWidth + gLW + dirWidthO) * dPD; j += dPD) {
00844             uAux[j + 0 - gLW * dPD] = s11b(&uAux[j + 0]) / dxi;
00845             uAux[j + 1 - gLW * dPD] = s11b(&uAux[j + 1]) / dxi;
00846             uAux[j + 2 - gLW * dPD] = s11f(&uAux[j + 2]) / dxi;
00847             uAux[j + 3 - gLW * dPD] = s11f(&uAux[j + 3]) / dxi;
00848             uAux[j + 4 - gLW * dPD] = s11f(&uAux[j + 4]) / dxi;
00849             uAux[j + 5 - gLW * dPD] = s11f(&uAux[j + 5]) / dxi;
00850         }
00851     }
00852     break;
00853 case 12: // gLW=7
00854     #pragma omp parallel for default(none) \
00855     shared(perpPlainSize, dxi, dirWidth, dirWidthO, gLW, dPD, uAux)
00856     for (sunindextype i = 0; i < perpPlainSize; i++) {
00857         #pragma omp simd
00858         for (sunindextype j = (i * dirWidth + gLW) * dPD;

```



```

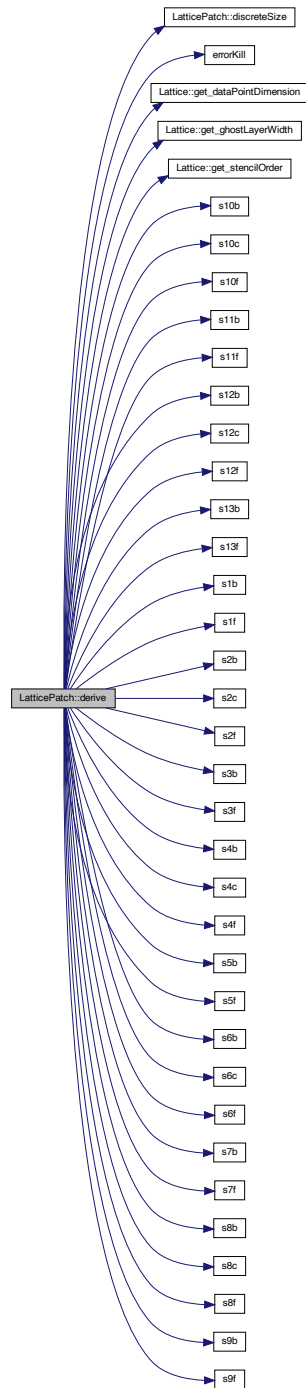
00859         j < (i * dirWidth + gLW + dirWidthO) * dPD; j += dPD) {
00860     uAux[j + 0 - gLW * dPD] = s12b(&uAux[j + 0]) / dxi;
00861     uAux[j + 1 - gLW * dPD] = s12b(&uAux[j + 1]) / dxi;
00862     uAux[j + 2 - gLW * dPD] = s12f(&uAux[j + 2]) / dxi;
00863     uAux[j + 3 - gLW * dPD] = s12f(&uAux[j + 3]) / dxi;
00864     uAux[j + 4 - gLW * dPD] = s12c(&uAux[j + 4]) / dxi;
00865     uAux[j + 5 - gLW * dPD] = s12c(&uAux[j + 5]) / dxi;
00866     }
00867     }
00868     break;
00869 case 13: // gLW=7
00870     // For all points in the plane perpendicular to the given direction
00871     #pragma omp parallel for default(none) \
00872     shared(perpPlainSize, dxi, dirWidth, dirWidthO, gLW, dPD, uAux)
00873     for (sunindextype i = 0; i < perpPlainSize; i++) {
00874         // iterate through the derivation direction
00875         #pragma omp simd
00876         for (sunindextype j = (i * dirWidth
00877             + gLW /*to shift left by gLW below */) * dPD;
00878             j < (i * dirWidth + gLW + dirWidthO) * dPD; j += dPD) {
00879             // Compute the stencil derivative for any of the six field components
00880             // and update position by ghost width shift
00881             uAux[j + 0 - gLW * dPD] = s13b(&uAux[j + 0]) / dxi;
00882             uAux[j + 1 - gLW * dPD] = s13b(&uAux[j + 1]) / dxi;
00883             uAux[j + 2 - gLW * dPD] = s13f(&uAux[j + 2]) / dxi;
00884             uAux[j + 3 - gLW * dPD] = s13f(&uAux[j + 3]) / dxi;
00885             uAux[j + 4 - gLW * dPD] = s13f(&uAux[j + 4]) / dxi;
00886             uAux[j + 5 - gLW * dPD] = s13f(&uAux[j + 5]) / dxi;
00887         }
00888     }
00889     break;
00890 default:
00891     errorKill("Please set an existing stencil order");
00892     break;
00893 }
00894 }
00895 }

```

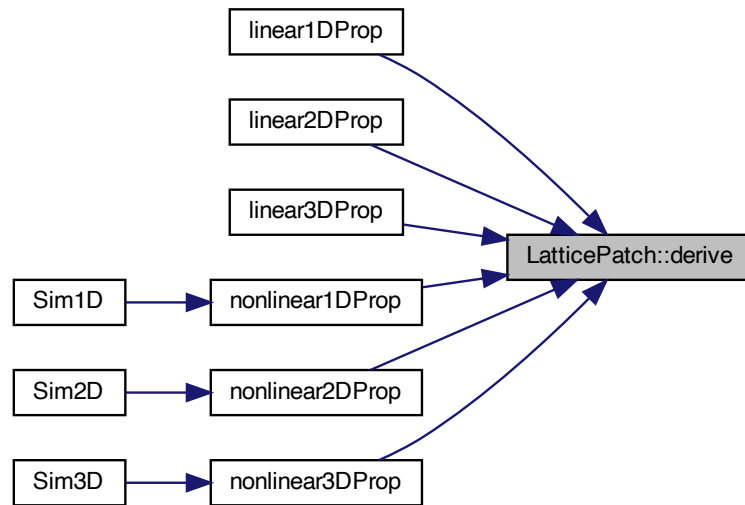
References [discreteSize\(\)](#), [dx](#), [dy](#), [dz](#), [envelopeLattice](#), [errorKill\(\)](#), [Lattice::get_dataPointDimension\(\)](#), [Lattice::get_ghostLayerWidth\(\)](#), [Lattice::get_stencilOrder\(\)](#), [s10b\(\)](#), [s10c\(\)](#), [s10f\(\)](#), [s11b\(\)](#), [s11f\(\)](#), [s12b\(\)](#), [s12c\(\)](#), [s12f\(\)](#), [s13b\(\)](#), [s13f\(\)](#), [s1b\(\)](#), [s1f\(\)](#), [s2b\(\)](#), [s2c\(\)](#), [s2f\(\)](#), [s3b\(\)](#), [s3f\(\)](#), [s4b\(\)](#), [s4c\(\)](#), [s4f\(\)](#), [s5b\(\)](#), [s5f\(\)](#), [s6b\(\)](#), [s6c\(\)](#), [s6f\(\)](#), [s7b\(\)](#), [s7f\(\)](#), [s8b\(\)](#), [s8c\(\)](#), [s8f\(\)](#), [s9b\(\)](#), [s9f\(\)](#), and [uAux](#).

Referenced by [linear1DProp\(\)](#), [linear2DProp\(\)](#), [linear3DProp\(\)](#), [nonlinear1DProp\(\)](#), [nonlinear2DProp\(\)](#), and [nonlinear3DProp\(\)](#).

Here is the call graph for this function:



Here is the caller graph for this function:



5.9.3.3 derotate()

```
void LatticePatch::derotate (
    int dir,
    sunrealtype * buffOut )
```

function to derotate uAux into dudata lattice direction of x

Derotate uAux with transposed rotation matrices and write to derivative buffer – normalization is done here by the factor 1/2

Definition at line 428 of file [LatticePatch.cpp](#).

```
00428                                     {
00429     // Check that the lattice as well as the translocation lookups have been set
00430     // up;
00431     checkFlag(FLatticePatchSetUp);
00432     checkFlag(TranslocationLookupSetUp);
00433     const int dPD = envelopeLattice->get_dataPointDimension();
00434     const int gLW = envelopeLattice->get_ghostLayerWidth();
00435     const sunindextype totalNP = discreteSize();
00436     sunindextype ii = 0, target = 0;
00437     switch (dir) {
00438     case 1:
00439         #pragma omp parallel for simd \
00440         private(ii, target) \
00441         shared(dPD, gLW, totalNP, uTox, uAux, buffOut) \
00442         schedule(static)
00443         for (sunindextype i = 0; i < totalNP; i++) {
00444             // get correct indices in u and rotation space
00445             target = dPD * i;
00446             ii = dPD * (uTox[i] - gLW);
00447             buffOut[target + 0] = uAux[5 + ii];
00448             buffOut[target + 1] = (-uAux[ii] + uAux[2 + ii]) / 2.;
00449             buffOut[target + 2] = (uAux[1 + ii] - uAux[3 + ii]) / 2.;
00450             buffOut[target + 3] = uAux[4 + ii];
00451             buffOut[target + 4] = (uAux[1 + ii] + uAux[3 + ii]) / 2.;
00452         }
```

```

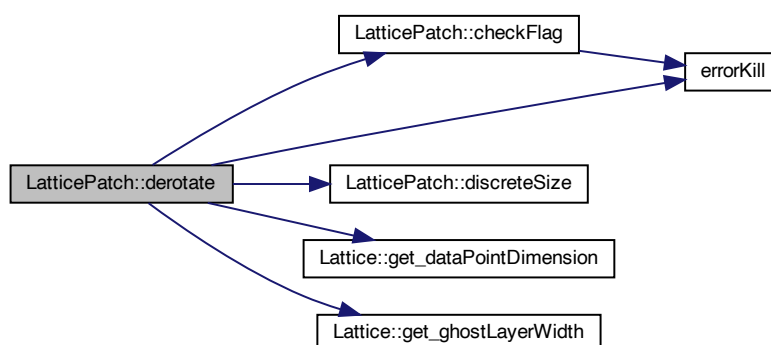
00452     buffOut[target + 5] = (uAux[ii] + uAux[2 + ii]) / 2.;
00453 }
00454 break;
00455 case 2:
00456 #pragma omp parallel for simd \
00457 private(ii, target) \
00458 shared(dPD, gLW, totalNP, uTox, uAux, buffOut) \
00459 schedule(static)
00460 for (sunindextype i = 0; i < totalNP; i++) {
00461     target = dPD * i;
00462     ii = dPD * (uToy[i] - gLW);
00463     buffOut[target + 0] = (uAux[ii] - uAux[2 + ii]) / 2.;
00464     buffOut[target + 1] = uAux[5 + ii];
00465     buffOut[target + 2] = (-uAux[1 + ii] + uAux[3 + ii]) / 2.;
00466     buffOut[target + 3] = (uAux[1 + ii] + uAux[3 + ii]) / 2.;
00467     buffOut[target + 4] = uAux[4 + ii];
00468     buffOut[target + 5] = (uAux[ii] + uAux[2 + ii]) / 2.;
00469 }
00470 break;
00471 case 3:
00472 #pragma omp parallel for simd \
00473 private(ii, target) \
00474 shared(dPD, gLW, totalNP, uTox, uAux, buffOut) \
00475 schedule(static)
00476 for (sunindextype i = 0; i < totalNP; i++) {
00477     target = dPD * i;
00478     ii = dPD * (uToz[i] - gLW);
00479     buffOut[target + 0] = (-uAux[ii] + uAux[2 + ii]) / 2.;
00480     buffOut[target + 1] = (uAux[1 + ii] - uAux[3 + ii]) / 2.;
00481     buffOut[target + 2] = uAux[5 + ii];
00482     buffOut[target + 3] = (uAux[1 + ii] + uAux[3 + ii]) / 2.;
00483     buffOut[target + 4] = (uAux[ii] + uAux[2 + ii]) / 2.;
00484     buffOut[target + 5] = uAux[4 + ii];
00485 }
00486 break;
00487 default:
00488     errorKill("Tried to derotate from the wrong direction");
00489 break;
00490 }
00491 }

```

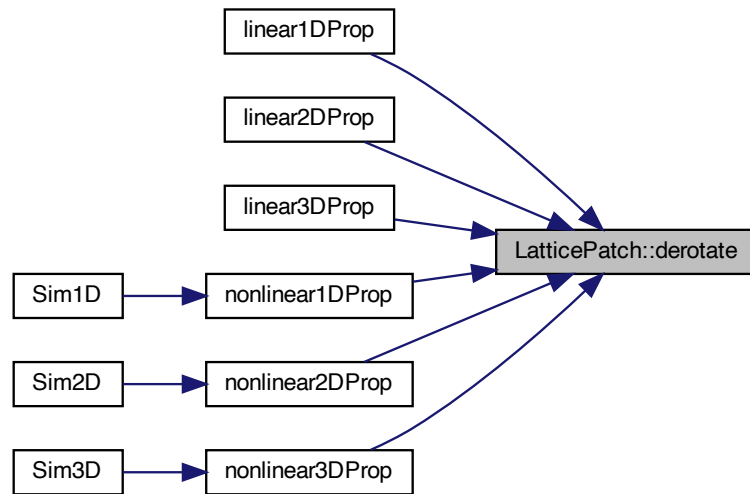
References [checkFlag\(\)](#), [discreteSize\(\)](#), [envelopeLattice](#), [errorKill\(\)](#), [FLatticePatchSetUp](#), [Lattice::get_dataPointDimension\(\)](#), [Lattice::get_ghostLayerWidth\(\)](#), [TranslocationLookupSetUp](#), [uAux](#), [uTox](#), [uToy](#), and [uToz](#).

Referenced by [linear1DProp\(\)](#), [linear2DProp\(\)](#), [linear3DProp\(\)](#), [nonlinear1DProp\(\)](#), [nonlinear2DProp\(\)](#), and [nonlinear3DProp\(\)](#).

Here is the call graph for this function:



Here is the caller graph for this function:



5.9.3.4 discreteSize()

```

sunindextype LatticePatch::discreteSize (
    int dir = 0 ) const

```

function to get the discrete size of the [LatticePatch](#)

Return the discrete size of the patch: number of lattice patch points in specified dimension

Definition at line 185 of file [LatticePatch.cpp](#).

```

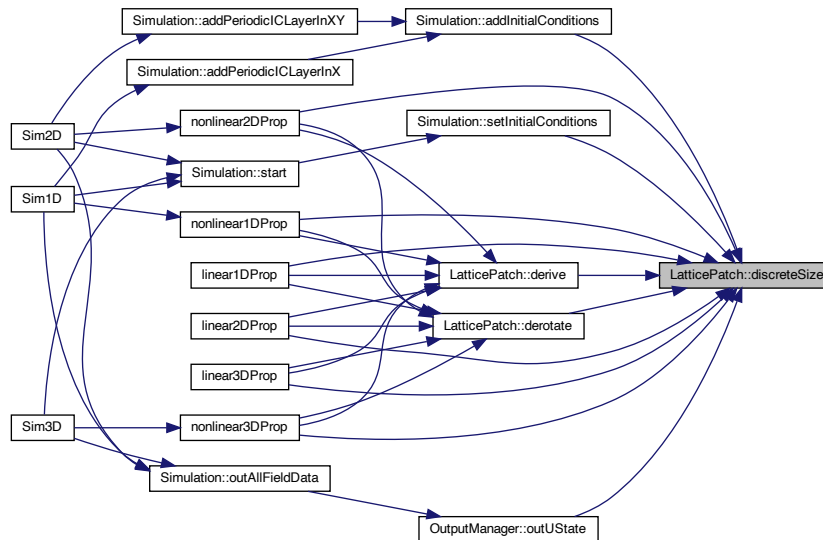
00185                                     {
00186     switch (dir) {
00187     case 0:
00188         return nx * ny * nz;
00189     case 1:
00190         return nx;
00191     case 2:
00192         return ny;
00193     case 3:
00194         return nz;
00195     // case 4: return uAux.size(); // for debugging
00196     default:
00197         return -1;
00198     }
00199 }

```

References [nx](#), [ny](#), and [nz](#).

Referenced by [Simulation::addInitialConditions\(\)](#), [derive\(\)](#), [derotate\(\)](#), [linear1DProp\(\)](#), [linear2DProp\(\)](#), [linear3DProp\(\)](#), [nonlinear1DProp\(\)](#), [nonlinear2DProp\(\)](#), [nonlinear3DProp\(\)](#), [OutputManager::outUState\(\)](#), and [Simulation::setInitialConditions\(\)](#).

Here is the caller graph for this function:



5.9.3.5 exchangeGhostCells()

```
void LatticePatch::exchangeGhostCells (
    const int dir )
```

function to exchange ghost cells

Perform the ghost cell exchange in a specified direction.

Definition at line 509 of file [LatticePatch.cpp](#).

```
00509 {
00510     // Check that the lattice has been set up
00511     checkFlag(FLatticeDimensionSet);
00512     checkFlag(FLatticePatchSetUp);
00513     // Variables to per dimension calculate the halo indices, and distance to
00514     // other side halo boundary
00515     int mx = 1, my = 1, mz = 1, distToRight = 1;
00516     const int gLW = envelopeLattice->get_ghostLayerWidth();
00517     // In the chosen direction m is set to ghost layer width while the others
00518     // remain to form the plane
00519     switch (dir) {
00520     case 1:
00521         mx = gLW;
00522         my = ny;
00523         mz = nz;
00524         distToRight = (nx - gLW);
00525         break;
00526     case 2:
00527         mx = nx;
00528         my = gLW;
00529         mz = nz;
00530         distToRight = nx * (ny - gLW);
00531         break;
00532     case 3:
00533         mx = nx;
00534         my = ny;
00535         mz = gLW;
00536         distToRight = nx * ny * (nz - gLW);
00537         break;
00538     }
```

```

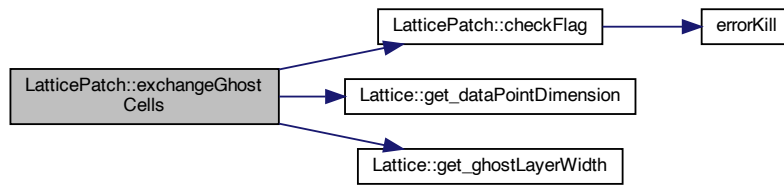
00539 // total number of exchanged points
00540 const int dPD = envelopeLattice->get_dataPointDimension();
00541 const sunindextype exchangeSize = mx * my * mz * dPD;
00542 // provide size of the halos for ghost cells
00543 ghostCellLeft.resize(exchangeSize);
00544 ghostCellRight.resize(ghostCellLeft.size());
00545 ghostCellLeftToSend.resize(ghostCellLeft.size());
00546 ghostCellRightToSend.resize(ghostCellLeft.size());
00547 gCLData = &ghostCellLeft[0];
00548 gCRData = &ghostCellRight[0];
00549 statusFlags |= GhostLayersInitialized;
00550
00551 // Initialize running index li for the halo buffers, and index ui of uData for
00552 // data transfer
00553 sunindextype li = 0, ui = 0;
00554 // Fill the halo buffers
00555 #pragma omp parallel for default(none) \
00556 private(ui) firstprivate(li) \
00557 shared(nx, ny, mx, my, mz, dPD, distToRight, uData, \
00558        ghostCellLeftToSend, ghostCellRightToSend)
00559 for (sunindextype iz = 0; iz < mz; iz++) {
00560     for (sunindextype iy = 0; iy < my; iy++) {
00561         // uData vector start index of halo data to be transferred
00562         // with each z-step add the whole xy-plane and with y-step the x-range ->
00563         // iterate all x-ranges
00564         ui = (iz * nx * ny + iy * nx) * dPD;
00565         // copy left halo data from uData to buffer, transfer size is given by
00566         // x-length (not x-range)
00567         std::copy(&uData[ui], &uData[ui + mx * dPD], &ghostCellLeftToSend[li]);
00568         ui += distToRight * dPD;
00569         std::copy(&uData[ui], &uData[ui + mx * dPD], &ghostCellRightToSend[li]);
00570
00571         // increase halo index by transferred items per y-iteration step
00572         // (x-length)
00573         li += mx * dPD;
00574     }
00575 }
00576
00577 /* Send and receive the data to and from neighboring latticePatches */
00578 // Adjust direction to cartesian communicator
00579 int dim = 2; // default for dir==1
00580 if (dir == 2) {
00581     dim = 1;
00582 } else if (dir == 3) {
00583     dim = 0;
00584 }
00585 int rank_source = 0, rank_dest = 0;
00586 MPI_Cart_shift(envelopeLattice->comm, dim, -1, &rank_source,
00587               &rank_dest); // s.t. rank_dest is left & v.v.
00588
00589 // nonblocking Irecv/Isend
00590
00591 MPI_Request requests[4];
00592 MPI_Irecv(&ghostCellRight[0], exchangeSize, MPI_SUNREALTYPE, rank_source, 1,
00593 envelopeLattice->comm, &requests[0]);
00594 MPI_Isend(&ghostCellLeftToSend[0], exchangeSize, MPI_SUNREALTYPE, rank_dest,
00595 1, envelopeLattice->comm, &requests[1]);
00596 MPI_Irecv(&ghostCellLeft[0], exchangeSize, MPI_SUNREALTYPE, rank_dest, 2,
00597 envelopeLattice->comm, &requests[2]);
00598 MPI_Isend(&ghostCellRightToSend[0], exchangeSize, MPI_SUNREALTYPE,
00599 rank_source, 2, envelopeLattice->comm, &requests[3]);
00600 MPI_Waitall(4, requests, MPI_STATUS_IGNORE);
00601
00602 // blocking Sendrecv:
00603 /*
00604 MPI_Sendrecv(&ghostCellLeftToSend[0], exchangeSize, MPI_SUNREALTYPE,
00605 rank_dest, 1, &ghostCellRight[0], exchangeSize, MPI_SUNREALTYPE,
00606 rank_source, 1, envelopeLattice->comm, MPI_STATUS_IGNORE);
00607 MPI_Sendrecv(&ghostCellRightToSend[0], exchangeSize, MPI_SUNREALTYPE,
00608 rank_source, 2, &ghostCellLeft[0], exchangeSize, MPI_SUNREALTYPE,
00609 rank_dest, 2, envelopeLattice->comm, MPI_STATUS_IGNORE);
00610 */
00611 */
00612 }

```

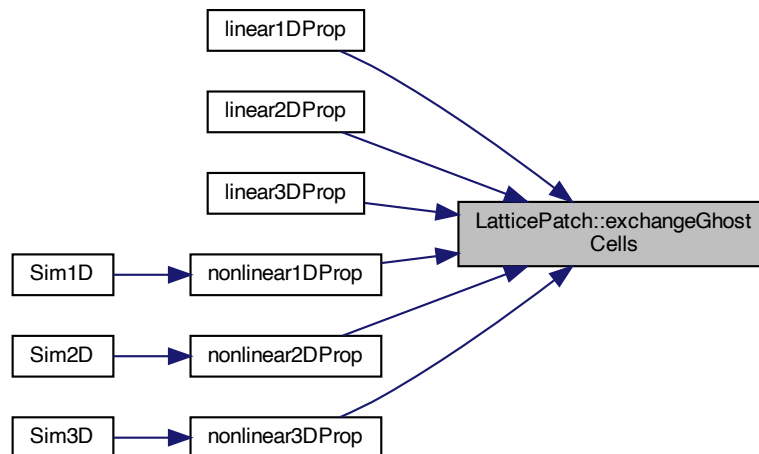
References [checkFlag\(\)](#), [Lattice::comm](#), [envelopeLattice](#), [FLatticeDimensionSet](#), [FLatticePatchSetUp](#), [gCLData](#), [gCRData](#), [Lattice::get_dataPointDimension\(\)](#), [Lattice::get_ghostLayerWidth\(\)](#), [ghostCellLeft](#), [ghostCellLeftToSend](#), [ghostCellRight](#), [ghostCellRightToSend](#), [GhostLayersInitialized](#), [nx](#), [ny](#), [nz](#), [statusFlags](#), and [uData](#).

Referenced by [linear1DProp\(\)](#), [linear2DProp\(\)](#), [linear3DProp\(\)](#), [nonlinear1DProp\(\)](#), [nonlinear2DProp\(\)](#), and [nonlinear3DProp\(\)](#).

Here is the call graph for this function:



Here is the caller graph for this function:



5.9.3.6 generateTranslocationLookup()

```
void LatticePatch::generateTranslocationLookup ( )
```

function to fill out the lookup tables for cache efficiency

In order to avoid cache misses: create vectors to translate u vector into space coordinates and vice versa and same for left and right ghost layers to space

Definition at line 235 of file [LatticePatch.cpp](#).

```

00235 {
00236     // Check that the lattice has been set up
00237     checkFlag(FLatticeDimensionSet);
00238     // lengths for auxilliary layers, including ghost layers
00239     const int gLW = envelopeLattice->get_ghostLayerWidth();
00240     const sunindextype mx = nx + 2 * gLW;
00241     const sunindextype my = ny + 2 * gLW;
00242     const sunindextype mz = nz + 2 * gLW;
  
```



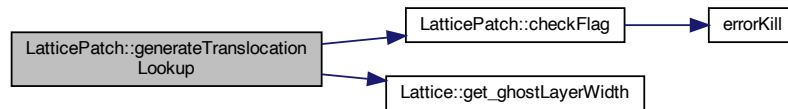
```

00243 // sizes for lookup vectors
00244 const sunindextype totalNP = nx * ny * nz;
00245 const sunindextype haloXSize = mx * ny * nz;
00246 const sunindextype haloYSize = nx * my * nz;
00247 const sunindextype haloZSize = nx * ny * mz;
00248 // generate u->uAux
00249 uTox.resize(totalNP);
00250 uToy.resize(totalNP);
00251 uToz.resize(totalNP);
00252 // generate uAux->u with length including halo
00253 xTou.resize(haloXSize);
00254 yTou.resize(haloYSize);
00255 zTou.resize(haloZSize);
00256 // same for ghost layer lookup tables
00257 const sunindextype ghostXSize = gLW * ny * nz;
00258 const sunindextype ghostYSize = gLW * nx * nz;
00259 const sunindextype ghostZSize = gLW * nx * ny;
00260 lgcTox.resize(ghostXSize);
00261 rgcTox.resize(ghostXSize);
00262 lgcToy.resize(ghostYSize);
00263 rgcToy.resize(ghostYSize);
00264 lgcToz.resize(ghostZSize);
00265 rgcToz.resize(ghostZSize);
00266 // variables for cartesian position in the 3D discrete lattice
00267 sunindextype px = 0, py = 0, pz = 0;
00268 // Fill the lookup tables
00269 #pragma omp parallel default(none) \
00270 private(px, py, pz) \
00271 shared(uTox, uToy, uToz, xTou, yTou, zTou, \
00272        nx, ny, mx, my, mz, gLW, totalNP, \
00273        lgcTox, rgcTox, lgcToy, rgcToy, lgcToz, rgcToz, \
00274        ghostXSize, ghostYSize, ghostZSize)
00275 {
00276 #pragma omp for simd schedule(static)
00277 for (sunindextype i = 0; i < totalNP; i++) { // loop over the patch
00278 // calculate cartesian coordinates
00279 px = i % nx;
00280 py = (i / nx) % ny;
00281 pz = (i / nx) / ny;
00282 // fill lookups extended by halos (useful for y and z direction)
00283 uTox[i] = (px + gLW) + py * mx +
00284           pz * mx * ny; // unroll (de-flatten) cartesian dimension
00285 xTou[px + py * mx + pz * mx * ny] =
00286 i; // match cartesian point to u location
00287 uToy[i] = (py + gLW) + pz * my + px * my * nz;
00288 yTou[py + pz * my + px * my * nz] = i;
00289 uToz[i] = (pz + gLW) + px * mz + py * mz * nx;
00290 zTou[pz + px * mz + py * mz * nx] = i;
00291 }
00292 #pragma omp for simd schedule(static)
00293 for (sunindextype i = 0; i < ghostXSize; i++) {
00294 px = i % gLW;
00295 py = (i / gLW) % ny;
00296 pz = (i / gLW) / ny;
00297 lgcTox[i] = px + py * mx + pz * mx * ny;
00298 rgcTox[i] = px + nx + gLW + py * mx + pz * mx * ny;
00299 }
00300 #pragma omp for simd schedule(static)
00301 for (sunindextype i = 0; i < ghostYSize; i++) {
00302 px = i % nx;
00303 py = (i / nx) % gLW;
00304 pz = (i / nx) / gLW;
00305 lgcToy[i] = py + pz * my + px * my * nz;
00306 rgcToy[i] = py + ny + gLW + pz * my + px * my * nz;
00307 }
00308 #pragma omp for simd schedule(static)
00309 for (sunindextype i = 0; i < ghostZSize; i++) {
00310 px = i % nx;
00311 py = (i / nx) % ny;
00312 pz = (i / nx) / ny;
00313 lgcToz[i] = pz + px * mz + py * mz * nx;
00314 rgcToz[i] = pz + nz + gLW + px * mz + py * mz * nx;
00315 }
00316 }
00317 statusFlags |= TranslocationLookupSetUp;
00318 }

```

References [checkFlag\(\)](#), [envelopeLattice](#), [FLatticeDimensionSet](#), [Lattice::get_ghostLayerWidth\(\)](#), [lgcTox](#), [lgcToy](#), [lgcToz](#), [nx](#), [ny](#), [nz](#), [rgcTox](#), [rgcToy](#), [rgcToz](#), [statusFlags](#), [TranslocationLookupSetUp](#), [uTox](#), [uToy](#), [uToz](#), [xTou](#), [yTou](#), and [zTou](#).

Here is the call graph for this function:



5.9.3.7 getDelta()

```

sunrealtype LatticePatch::getDelta (
    const int dir ) const
  
```

function to get distance between points

Return the distance between points in the patch in a dimension.

Definition at line 217 of file [LatticePatch.cpp](#).

```

00217                                     {
00218     switch (dir) {
00219     case 1:
00220         return dx;
00221     case 2:
00222         return dy;
00223     case 3:
00224         return dz;
00225     default:
00226         errorKill(
00227             "LatticePatch::getDelta function called with wrong dir parameter");
00228         return -1;
00229     }
00230 }
  
```

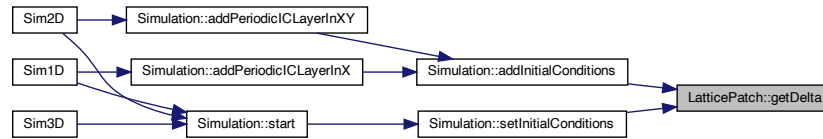
References [dx](#), [dy](#), [dz](#), and [errorKill\(\)](#).

Referenced by [Simulation::addInitialConditions\(\)](#), and [Simulation::setInitialConditions\(\)](#).

Here is the call graph for this function:



Here is the caller graph for this function:



5.9.3.8 initializeBuffers()

```
void LatticePatch::initializeBuffers ( )
```

initialize buffers to save derivatives

Create buffers to save derivative values, optimizing computational load.

Definition at line 494 of file [LatticePatch.cpp](#).

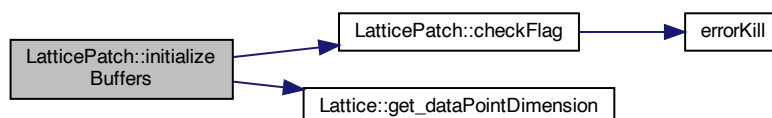
```

00494 {
00495     // Check that the lattice has been set up
00496     checkFlag(FLatticeDimensionSet);
00497     const int dPD = envelopeLattice->get_dataPointDimension();
00498     buffX.resize(nx * ny * nz * dPD);
00499     buffY.resize(nx * ny * nz * dPD);
00500     buffZ.resize(nx * ny * nz * dPD);
00501     // Set pointers used for propagation functions
00502     buffData[0] = &buffX[0];
00503     buffData[1] = &buffY[0];
00504     buffData[2] = &buffZ[0];
00505     statusFlags |= BuffersInitialized;
00506 }
```

References [buffData](#), [BuffersInitialized](#), [buffX](#), [buffY](#), [buffZ](#), [checkFlag\(\)](#), [envelopeLattice](#), [FLatticeDimensionSet](#), [Lattice::get_dataPointDimension\(\)](#), [nx](#), [ny](#), [nz](#), and [statusFlags](#).

Referenced by [Simulation::initializePatchwork\(\)](#).

Here is the call graph for this function:



Here is the caller graph for this function:



5.9.3.9 origin()

```
sunrealtype LatticePatch::origin (
    const int dir ) const
```

function to get the origin of the patch

Return the physical origin of the patch in a dimension.

Definition at line 202 of file [LatticePatch.cpp](#).

```
00202 {
00203     switch (dir) {
00204     case 1:
00205         return x0;
00206     case 2:
00207         return y0;
00208     case 3:
00209         return z0;
00210     default:
00211         errorKill("LatticePatch::origin function called with wrong dir parameter");
00212         return -1;
00213     }
00214 }
```

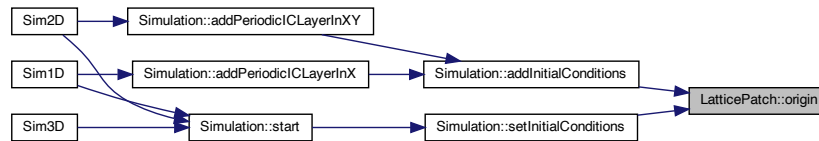
References [errorKill\(\)](#), [x0](#), [y0](#), and [z0](#).

Referenced by [Simulation::addInitialConditions\(\)](#), and [Simulation::setInitialConditions\(\)](#).

Here is the call graph for this function:



Here is the caller graph for this function:



5.9.3.10 rotateIntoEigen()

```
void LatticePatch::rotateIntoEigen (
    const int dir )
```

function to rotate u into Z-matrix eigenraum

Rotate into eigenraum along R matrices of paper using the rotation methods; uAuxData gets the rotated left-halo-, inner-patch-, right-halo-data

Definition at line 323 of file [LatticePatch.cpp](#).

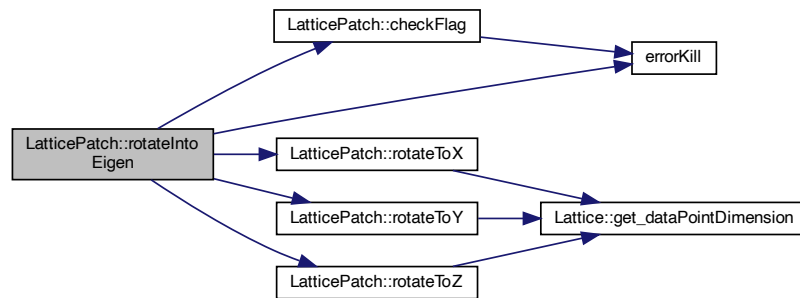
```

00323 {
00324     // Check that the lattice, ghost layers as well as the translocation lookups
00325     // have been set up;
00326     checkFlag(FLatticePatchSetUp);
00327     checkFlag(TranslocationLookupSetUp);
00328     checkFlag(GhostLayersInitialized); // this check is only after call to
00329                                         // exchange ghost cells
00330     switch (dir) {
00331     case 1:
00332         rotateToX(uAuxData, gCLData, lgcTox);
00333         rotateToX(uAuxData, uData, uTox);
00334         rotateToX(uAuxData, gCRData, rgcTox);
00335         break;
00336     case 2:
00337         rotateToY(uAuxData, gCLData, lgcToy);
00338         rotateToY(uAuxData, uData, uToy);
00339         rotateToY(uAuxData, gCRData, rgcToy);
00340         break;
00341     case 3:
00342         rotateToZ(uAuxData, gCLData, lgcToz);
00343         rotateToZ(uAuxData, uData, uToz);
00344         rotateToZ(uAuxData, gCRData, rgcToz);
00345         break;
00346     default:
00347         errorKill("Tried to rotate into the wrong direction");
00348         break;
00349     }
00350 }
```

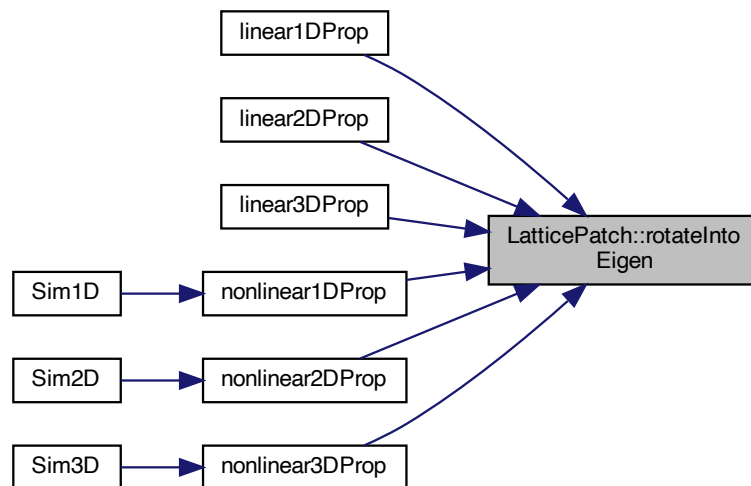
References [checkFlag\(\)](#), [errorKill\(\)](#), [FLatticePatchSetUp](#), [gCLData](#), [gCRData](#), [GhostLayersInitialized](#), [lgcTox](#), [lgcToy](#), [lgcToz](#), [rgcTox](#), [rgcToy](#), [rgcToz](#), [rotateToX\(\)](#), [rotateToY\(\)](#), [rotateToZ\(\)](#), [TranslocationLookupSetUp](#), [uAuxData](#), [uData](#), [uTox](#), [uToy](#), and [uToz](#).

Referenced by [linear1DProp\(\)](#), [linear2DProp\(\)](#), [linear3DProp\(\)](#), [nonlinear1DProp\(\)](#), [nonlinear2DProp\(\)](#), and [nonlinear3DProp\(\)](#).

Here is the call graph for this function:



Here is the caller graph for this function:



5.9.3.11 rotateToX()

```

void LatticePatch::rotateToX (
    sunrealtype * outArray,
    const sunrealtype * inArray,
    const std::vector< sunindextype > & lookup ) [inline], [private]
  
```

Rotate and translocate an input array according to a lookup into an output array

Rotate halo and inner-patch data vectors with rotation matrix Rx into eigenspace of Z matrix and write to auxiliary vector

Definition at line 354 of file [LatticePatch.cpp](#).

```

00356
00357     sunindextype ii = 0, target = 0;
00358     const sunindextype size = lookup.size();
00359     const int dPD = envelopeLattice->get_dataPointDimension();
00360     #pragma omp parallel for simd \
00361     private(target, ii) \
00362     shared(lookup, outArray, inArray, size, dPD) \
00363     schedule(static)
00364     for (sunindextype i = 0; i < size; i++) {
00365         // get correct u-vector and spatial indices along previously defined lookup
00366         // tables
00367         target = dPD * lookup[i];
00368         ii = dPD * i;
00369         outArray[target + 0] = -inArray[1 + ii] + inArray[5 + ii];
00370         outArray[target + 1] = inArray[2 + ii] + inArray[4 + ii];
00371         outArray[target + 2] = inArray[1 + ii] + inArray[5 + ii];
00372         outArray[target + 3] = -inArray[2 + ii] + inArray[4 + ii];
00373         outArray[target + 4] = inArray[3 + ii];
00374         outArray[target + 5] = inArray[ii];
00375     }
00376 }
```

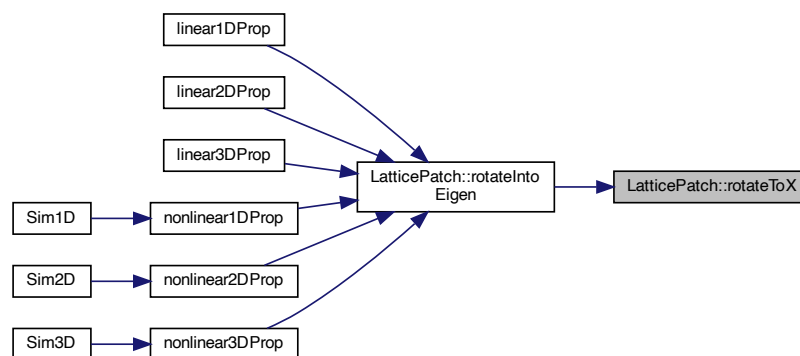
References [envelopeLattice](#), and [Lattice::get_dataPointDimension\(\)](#).

Referenced by [rotateIntoEigen\(\)](#).

Here is the call graph for this function:



Here is the caller graph for this function:



5.9.3.12 rotateToY()

```
void LatticePatch::rotateToY (
    sunrealtype * outArray,
    const sunrealtype * inArray,
    const std::vector< sunindextype > & lookup ) [inline], [private]
```

Rotate halo and inner-patch data vectors with rotation matrix Ry into eigenspace of Z matrix and write to auxiliary vector

Definition at line 380 of file [LatticePatch.cpp](#).

```
00382 {
00383     sunindextype ii = 0, target = 0;
00384     const int dPD = envelopeLattice->get_dataPointDimension();
00385     const sunindextype size = lookup.size();
00386     #pragma omp parallel for simd \
00387     private(target, ii) \
00388     shared(lookup, outArray, inArray, size, dPD) \
00389     schedule(static)
00390     for (sunindextype i = 0; i < size; i++) {
00391         target = dPD * lookup[i];
00392         ii = dPD * i;
00393         outArray[target + 0] = inArray[ii] + inArray[5 + ii];
00394         outArray[target + 1] = -inArray[2 + ii] + inArray[3 + ii];
00395         outArray[target + 2] = -inArray[ii] + inArray[5 + ii];
00396         outArray[target + 3] = inArray[2 + ii] + inArray[3 + ii];
00397         outArray[target + 4] = inArray[4 + ii];
00398         outArray[target + 5] = inArray[1 + ii];
00399     }
00400 }
```

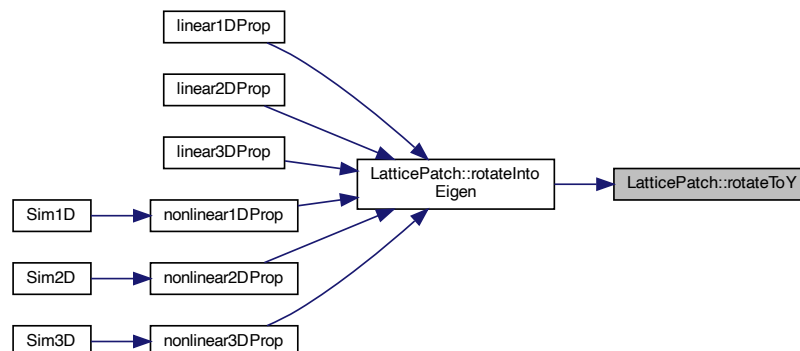
References [envelopeLattice](#), and [Lattice::get_dataPointDimension\(\)](#).

Referenced by [rotateIntoEigen\(\)](#).

Here is the call graph for this function:



Here is the caller graph for this function:



5.9.3.13 rotateToZ()

```
void LatticePatch::rotateToZ (
    sunrealtype * outArray,
    const sunrealtype * inArray,
    const std::vector< sunindextype > & lookup ) [inline], [private]
```

Rotate halo and inner-patch data vectors with rotation matrix R_z into eigenspace of Z matrix and write to auxiliary vector

Definition at line 404 of file [LatticePatch.cpp](#).

```
00406 {
00407     sunindextype ii = 0, target = 0;
00408     const sunindextype size = lookup.size();
00409     const int dPD = envelopeLattice->get_dataPointDimension();
00410     #pragma omp parallel for simd \
00411     private(target, ii) \
00412     shared(lookup, outArray, inArray, size, dPD) \
00413     schedule(static)
00414     for (sunindextype i = 0; i < size; i++) {
00415         target = dPD * lookup[i];
00416         ii = dPD * i;
00417         outArray[target + 0] = -inArray[ii] + inArray[4 + ii];
00418         outArray[target + 1] = inArray[1 + ii] + inArray[3 + ii];
00419         outArray[target + 2] = inArray[ii] + inArray[4 + ii];
00420         outArray[target + 3] = -inArray[1 + ii] + inArray[3 + ii];
00421         outArray[target + 4] = inArray[5 + ii];
00422         outArray[target + 5] = inArray[2 + ii];
00423     }
00424 }
```

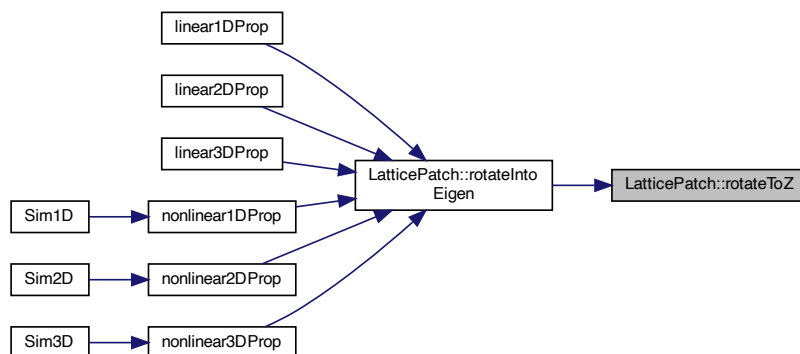
References [envelopeLattice](#), and [Lattice::get_dataPointDimension\(\)](#).

Referenced by [rotateIntoEigen\(\)](#).

Here is the call graph for this function:



Here is the caller graph for this function:



5.9.4 Friends And Related Function Documentation

5.9.4.1 generatePatchwork

```
int generatePatchwork (
    const Lattice & envelopeLattice,
    LatticePatch & patchToMold,
    const int DLx,
    const int DLy,
    const int DLz ) [friend]
```

friend function for creating the patchwork slicing of the overall lattice

Definition at line 109 of file [LatticePatch.cpp](#).

```
00111 {
00112 // Retrieve the ghost layer depth
00113 const int gLW = envelopeLattice.get_ghostLayerWidth();
00114 // Retrieve the data point dimension
00115 const int dPD = envelopeLattice.get_dataPointDimension();
00116 // MPI process/patch
00117 const int my_prc = envelopeLattice.my_prc;
00118 // Determine thicknes of the slice
00119 const sunindextype tot_NOXP = envelopeLattice.get_tot_nx();
00120 const sunindextype tot_NOYP = envelopeLattice.get_tot_ny();
00121 const sunindextype tot_NOZP = envelopeLattice.get_tot_nz();
00122 // position of the patch in the lattice of patches -> process associated to
00123 // position
00124 const sunindextype LIx = my_prc % DLx;
00125 const sunindextype LIy = (my_prc / DLx) % DLy;
00126 const sunindextype LIz = (my_prc / DLx) / DLy;
00127 // Determine the number of points in the patch and first absolute points in
00128 // each dimension
00129 const sunindextype local_NOXP = tot_NOXP / DLx;
00130 const sunindextype local_NOYP = tot_NOYP / DLy;
00131 const sunindextype local_NOZP = tot_NOZP / DLz;
00132 // absolute positions of the first point in each dimension
00133 const sunindextype firstXPoint = local_NOXP * LIx;
00134 const sunindextype firstYPoint = local_NOYP * LIy;
00135 const sunindextype firstZPoint = local_NOZP * LIz;
00136 // total number of points in the patch
00137 const sunindextype local_NODP = dPD * local_NOXP * local_NOYP * local_NOZP;
00138
00139 // Set patch up with above derived quantities
00140 patchToMold.dx = envelopeLattice.get_dx();
00141 patchToMold.dy = envelopeLattice.get_dy();
00142 patchToMold.dz = envelopeLattice.get_dz();
00143 patchToMold.x0 = firstXPoint * patchToMold.dx;
00144 patchToMold.y0 = firstYPoint * patchToMold.dy;
00145 patchToMold.z0 = firstZPoint * patchToMold.dz;
00146 patchToMold.LIx = LIx;
00147 patchToMold.LIy = LIy;
00148 patchToMold.LIz = LIz;
00149 patchToMold.nx = local_NOXP;
00150 patchToMold.ny = local_NOYP;
00151 patchToMold.nz = local_NOZP;
00152 patchToMold.lx = patchToMold.nx * patchToMold.dx;
00153 patchToMold.ly = patchToMold.ny * patchToMold.dy;
00154 patchToMold.lz = patchToMold.nz * patchToMold.dz;
00155 /* Create and allocate memory for parallel vectors with defined local and
00156 * global lenghts *
00157 * (-> CNode problem sizes Nlocal and N)
00158 * for field data and temporal derivatives and set extra pointers to them */
00159 patchToMold.u =
00160     N_VNew_Parallel(envelopeLattice.comm, local_NODP,
00161                     envelopeLattice.get_tot_noDP(), envelopeLattice.sunctx);
00162 patchToMold.uData = NV_DATA_P(patchToMold.u);
00163 patchToMold.du =
00164     N_VNew_Parallel(envelopeLattice.comm, local_NODP,
00165                     envelopeLattice.get_tot_noDP(), envelopeLattice.sunctx);
00166 patchToMold.duData = NV_DATA_P(patchToMold.du);
00167 // Allocate space for auxiliary uAux so that the lattice and all possible
00168 // directions of ghost layers fit
00169 const sunindextype s1 = patchToMold.nx, s2 = patchToMold.ny,
00170     s3 = patchToMold.nz;
```

```

00171     const sunindextype s_min = std::min(s1, std::min(s2, s3));
00172     patchToMold.uAux.resize(s1 * s2 * s3 / s_min * (s_min + 2 * gLW) * dPD);
00173     patchToMold.uAuxData = &patchToMold.uAux[0];
00174     patchToMold.envelopeLattice = &envelopeLattice;
00175     // Set patch "name" to process number -> only for debugging
00176     // patchToMold.ID=my_prc;
00177     // set flag
00178     patchToMold.statusFlags = FLatticePatchSetUp;
00179     patchToMold.generateTranslocationLookup();
00180     return 0;
00181 }

```

5.9.5 Field Documentation

5.9.5.1 buffData

`std::array<sunrealtype *, 3> LatticePatch::buffData`

pointer to spatial derivative data buffers

Definition at line 208 of file [LatticePatch.h](#).

Referenced by [initializeBuffers\(\)](#), [linear1DProp\(\)](#), [linear2DProp\(\)](#), [linear3DProp\(\)](#), [nonlinear1DProp\(\)](#), [nonlinear2DProp\(\)](#), and [nonlinear3DProp\(\)](#).

5.9.5.2 buffX

`std::vector<sunrealtype> LatticePatch::buffX [private]`

buffer to save spatial derivative values

Definition at line 169 of file [LatticePatch.h](#).

Referenced by [initializeBuffers\(\)](#).

5.9.5.3 buffY

`std::vector<sunrealtype> LatticePatch::buffY [private]`

buffer to save spatial derivative values

Definition at line 169 of file [LatticePatch.h](#).

Referenced by [initializeBuffers\(\)](#).

5.9.5.4 buffZ

```
std::vector<sunrealtype> LatticePatch::buffZ [private]
```

buffer to save spatial derivative values

Definition at line 169 of file [LatticePatch.h](#).

Referenced by [initializeBuffers\(\)](#).

5.9.5.5 du

```
N_Vector LatticePatch::du
```

N_Vector for saving temporal derivatives of the field data.

Definition at line 196 of file [LatticePatch.h](#).

Referenced by [~LatticePatch\(\)](#).

5.9.5.6 duData

```
sunrealtype* LatticePatch::duData
```

pointer to time-derivative data

Definition at line 202 of file [LatticePatch.h](#).

Referenced by [TimeEvolution::f\(\)](#), [linear1DProp\(\)](#), [linear2DProp\(\)](#), and [linear3DProp\(\)](#).

5.9.5.7 dx

```
sunrealtype LatticePatch::dx [private]
```

physical distance between lattice points in x-direction

Definition at line 152 of file [LatticePatch.h](#).

Referenced by [derive\(\)](#), and [getDelta\(\)](#).

5.9.5.8 dy

```
sunrealtype LatticePatch::dy [private]
```

physical distance between lattice points in y-direction

Definition at line 154 of file [LatticePatch.h](#).

Referenced by [derive\(\)](#), and [getDelta\(\)](#).

5.9.5.9 dz

```
sunrealtype LatticePatch::dz [private]
```

physical distance between lattice points in z-direction

Definition at line 156 of file [LatticePatch.h](#).

Referenced by [derive\(\)](#), and [getDelta\(\)](#).

5.9.5.10 envelopeLattice

```
const Lattice* LatticePatch::envelopeLattice [private]
```

pointer to the enveloping lattice

Definition at line 160 of file [LatticePatch.h](#).

Referenced by [derive\(\)](#), [derotate\(\)](#), [exchangeGhostCells\(\)](#), [generateTranslocationLookup\(\)](#), [initializeBuffers\(\)](#), [rotateToX\(\)](#), [rotateToY\(\)](#), and [rotateToZ\(\)](#).

5.9.5.11 gCLData

```
sunrealtype* LatticePatch::gCLData
```

pointer to halo data

Definition at line 205 of file [LatticePatch.h](#).

Referenced by [exchangeGhostCells\(\)](#), and [rotateIntoEigen\(\)](#).

5.9.5.12 gCRData

```
sunrealtype * LatticePatch::gCRData
```

pointer to halo data

Definition at line 205 of file [LatticePatch.h](#).

Referenced by [exchangeGhostCells\(\)](#), and [rotateIntoEigen\(\)](#).

5.9.5.13 ghostCellLeft

```
std::vector<sunrealtype> LatticePatch::ghostCellLeft [private]
```

buffer for passing ghost cell data

Definition at line 173 of file [LatticePatch.h](#).

Referenced by [exchangeGhostCells\(\)](#).

5.9.5.14 ghostCellLeftToSend

```
std::vector<sunrealtype> LatticePatch::ghostCellLeftToSend [private]
```

buffer for passing ghost cell data

Definition at line 173 of file [LatticePatch.h](#).

Referenced by [exchangeGhostCells\(\)](#).

5.9.5.15 ghostCellRight

```
std::vector<sunrealtype> LatticePatch::ghostCellRight [private]
```

buffer for passing ghost cell data

Definition at line 173 of file [LatticePatch.h](#).

Referenced by [exchangeGhostCells\(\)](#).

5.9.5.16 ghostCellRightToSend

```
std::vector<sunrealtype> LatticePatch::ghostCellRightToSend [private]
```

buffer for passing ghost cell data

Definition at line 174 of file [LatticePatch.h](#).

Referenced by [exchangeGhostCells\(\)](#).

5.9.5.17 ghostCells

```
std::vector<sunrealtype> LatticePatch::ghostCells [private]
```

buffer for passing ghost cell data

Definition at line 174 of file [LatticePatch.h](#).

5.9.5.18 ghostCellsToSend

```
std::vector<sunrealtype> LatticePatch::ghostCellsToSend [private]
```

buffer for passing ghost cell data

Definition at line 174 of file [LatticePatch.h](#).

5.9.5.19 ID

```
int LatticePatch::ID
```

ID of the [LatticePatch](#), corresponds to process number (for debugging)

Definition at line 192 of file [LatticePatch.h](#).

5.9.5.20 lgcTox

```
std::vector<sunindextype> LatticePatch::lgcTox [private]
```

ghost cell translocation lookup table

Definition at line 178 of file [LatticePatch.h](#).

Referenced by [generateTranslocationLookup\(\)](#), and [rotateIntoEigen\(\)](#).

5.9.5.21 lgcToy

```
std::vector<sunindextype> LatticePatch::lgcToy [private]
```

ghost cell translocation lookup table

Definition at line 178 of file [LatticePatch.h](#).

Referenced by [generateTranslocationLookup\(\)](#), and [rotateIntoEigen\(\)](#).

5.9.5.22 lgcToz

```
std::vector<sunindextype> LatticePatch::lgcToz [private]
```

ghost cell translocation lookup table

Definition at line 178 of file [LatticePatch.h](#).

Referenced by [generateTranslocationLookup\(\)](#), and [rotateIntoEigen\(\)](#).

5.9.5.23 LIx

```
sunindextype LatticePatch::LIx [private]
```

inner position of lattice-patch in the lattice patchwork; x-points

Definition at line 134 of file [LatticePatch.h](#).

Referenced by [LatticePatch\(\)](#).

5.9.5.24 LIy

```
sunindextype LatticePatch::LIy [private]
```

inner position of lattice-patch in the lattice patchwork; y-points

Definition at line 136 of file [LatticePatch.h](#).

Referenced by [LatticePatch\(\)](#).

5.9.5.25 LIz

```
sunindextype LatticePatch::LIz [private]
```

inner position of lattice-patch in the lattice patchwork; z-points

Definition at line 138 of file [LatticePatch.h](#).

Referenced by [LatticePatch\(\)](#).

5.9.5.26 lx

```
sunrealtype LatticePatch::lx [private]
```

physical size of the lattice-patch in the x-dimension

Definition at line 140 of file [LatticePatch.h](#).

Referenced by [LatticePatch\(\)](#).

5.9.5.27 ly

```
sunrealtype LatticePatch::ly [private]
```

physical size of the lattice-patch in the y-dimension

Definition at line 142 of file [LatticePatch.h](#).

Referenced by [LatticePatch\(\)](#).

5.9.5.28 lz

```
sunrealtype LatticePatch::lz [private]
```

physical size of the lattice-patch in the z-dimension

Definition at line 144 of file [LatticePatch.h](#).

Referenced by [LatticePatch\(\)](#).

5.9.5.29 nx

```
sunindextype LatticePatch::nx [private]
```

number of points in the lattice patch in the x-dimension

Definition at line 146 of file [LatticePatch.h](#).

Referenced by [discreteSize\(\)](#), [exchangeGhostCells\(\)](#), [generateTranslocationLookup\(\)](#), [initializeBuffers\(\)](#), and [LatticePatch\(\)](#).

5.9.5.30 ny

```
sunindextype LatticePatch::ny [private]
```

number of points in the lattice patch in the y-dimension

Definition at line 148 of file [LatticePatch.h](#).

Referenced by [discreteSize\(\)](#), [exchangeGhostCells\(\)](#), [generateTranslocationLookup\(\)](#), [initializeBuffers\(\)](#), and [LatticePatch\(\)](#).

5.9.5.31 nz

```
sunindextype LatticePatch::nz [private]
```

number of points in the lattice patch in the z-dimension

Definition at line 150 of file [LatticePatch.h](#).

Referenced by [discreteSize\(\)](#), [exchangeGhostCells\(\)](#), [generateTranslocationLookup\(\)](#), [initializeBuffers\(\)](#), and [LatticePatch\(\)](#).

5.9.5.32 rgcTox

```
std::vector<sunindextype> LatticePatch::rgcTox [private]
```

ghost cell translocation lookup table

Definition at line 178 of file [LatticePatch.h](#).

Referenced by [generateTranslocationLookup\(\)](#), and [rotateIntoEigen\(\)](#).

5.9.5.33 rgcToy

```
std::vector<sunindextype> LatticePatch::rgcToy [private]
```

ghost cell translocation lookup table

Definition at line 178 of file [LatticePatch.h](#).

Referenced by [generateTranslocationLookup\(\)](#), and [rotateIntoEigen\(\)](#).

5.9.5.34 rgcToz

```
std::vector<sunindextype> LatticePatch::rgcToz [private]
```

ghost cell translocation lookup table

Definition at line 178 of file [LatticePatch.h](#).

Referenced by [generateTranslocationLookup\(\)](#), and [rotateIntoEigen\(\)](#).

5.9.5.35 statusFlags

```
unsigned int LatticePatch::statusFlags [private]
```

lattice patch status flags

Definition at line 158 of file [LatticePatch.h](#).

Referenced by [checkFlag\(\)](#), [exchangeGhostCells\(\)](#), [generateTranslocationLookup\(\)](#), [initializeBuffers\(\)](#), [LatticePatch\(\)](#), and [~LatticePatch\(\)](#).

5.9.5.36 u

```
N_Vector LatticePatch::u
```

N_Vector for saving field components $u=(E,B)$ in lattice points.

Definition at line 194 of file [LatticePatch.h](#).

Referenced by [Simulation::advanceToTime\(\)](#), [Simulation::initializeCVODEobject\(\)](#), and [~LatticePatch\(\)](#).

5.9.5.37 uAux

```
std::vector<sunrealtype> LatticePatch::uAux [private]
```

aid (auxilliary) vector including ghost cells to compute the derivatives

Definition at line 162 of file [LatticePatch.h](#).

Referenced by [derive\(\)](#), and [derotate\(\)](#).

5.9.5.38 uAuxData

```
sunrealtype* LatticePatch::uAuxData
```

pointer to auxiliary data vector

Definition at line 200 of file [LatticePatch.h](#).

Referenced by [rotateIntoEigen\(\)](#).

5.9.5.39 uData

```
sunrealtype* LatticePatch::uData
```

pointer to field data

Definition at line 198 of file [LatticePatch.h](#).

Referenced by [Simulation::addInitialConditions\(\)](#), [exchangeGhostCells\(\)](#), [TimeEvolution::f\(\)](#), [OutputManager::outUState\(\)](#), [rotateIntoEigen\(\)](#), and [Simulation::setInitialConditions\(\)](#).

5.9.5.40 uTox

```
std::vector<sunindextype> LatticePatch::uTox [private]
```

translocation lookup table

Definition at line 165 of file [LatticePatch.h](#).

Referenced by [derotate\(\)](#), [generateTranslocationLookup\(\)](#), and [rotateIntoEigen\(\)](#).

5.9.5.41 uToy

```
std::vector<sunindextype> LatticePatch::uToy [private]
```

translocation lookup table

Definition at line 165 of file [LatticePatch.h](#).

Referenced by [derotate\(\)](#), [generateTranslocationLookup\(\)](#), and [rotateIntoEigen\(\)](#).

5.9.5.42 uToz

```
std::vector<sunindextype> LatticePatch::uToz [private]
```

translocation lookup table

Definition at line 165 of file [LatticePatch.h](#).

Referenced by [derotate\(\)](#), [generateTranslocationLookup\(\)](#), and [rotateIntoEigen\(\)](#).

5.9.5.43 x0

```
sunrealtype LatticePatch::x0 [private]
```

origin of the patch in physical space; x-coordinate

Definition at line 128 of file [LatticePatch.h](#).

Referenced by [LatticePatch\(\)](#), and [origin\(\)](#).

5.9.5.44 xTou

```
std::vector<sunindextype> LatticePatch::xTou [private]
```

translocation lookup table

Definition at line 165 of file [LatticePatch.h](#).

Referenced by [generateTranslocationLookup\(\)](#).

5.9.5.45 y0

```
sunrealtype LatticePatch::y0 [private]
```

origin of the patch in physical space; y-coordinate

Definition at line 130 of file [LatticePatch.h](#).

Referenced by [LatticePatch\(\)](#), and [origin\(\)](#).

5.9.5.46 yTou

```
std::vector<sunindextype> LatticePatch::yTou [private]
```

translocation lookup table

Definition at line 165 of file [LatticePatch.h](#).

Referenced by [generateTranslocationLookup\(\)](#).

5.9.5.47 z0

```
sunrealtype LatticePatch::z0 [private]
```

origin of the patch in physical space; z-coordinate

Definition at line 132 of file [LatticePatch.h](#).

Referenced by [LatticePatch\(\)](#), and [origin\(\)](#).

5.9.5.48 zTou

```
std::vector<sunindextype> LatticePatch::zTou [private]
```

translocation lookup table

Definition at line 165 of file [LatticePatch.h](#).

Referenced by [generateTranslocationLookup\(\)](#).

The documentation for this class was generated from the following files:

- [src/LatticePatch.h](#)
- [src/LatticePatch.cpp](#)

5.10 OutputManager Class Reference

Output Manager class to generate and coordinate output writing to disk.

```
#include <src/Outputters.h>
```

Public Member Functions

- [OutputManager](#) ()
default constructor
- void [generateOutputFolder](#) (const std::string &dir)
function that creates folder to save simulation data
- void [set_outputStyle](#) (const char _outputStyle)
set the output style
- void [outUState](#) (const int &state, const [Lattice](#) &lattice, const [LatticePatch](#) &latticePatch)
function to write data to disk in specified way
- const std::string & [getSimCode](#) () const
simCode getter function

Static Private Member Functions

- static std::string [SimCodeGenerator](#) ()
function to create the Code of the Simulations

Private Attributes

- std::string [simCode](#)
variable to save the SimCode generated at execution
- std::string [Path](#)
variable for the path to the output folder
- char [outputStyle](#)
output style; csv or binary

5.10.1 Detailed Description

Output Manager class to generate and coordinate output writing to disk.

Definition at line 21 of file [Outputters.h](#).

5.10.2 Constructor & Destructor Documentation

5.10.2.1 OutputManager()

```
OutputManager::OutputManager ( )
```

default constructor

Directly generate the simCode at construction.

Definition at line 12 of file [Outputters.cpp](#).

```
00012     {
00013     simCode = SimCodeGenerator();
00014     outputStyle = 'c';
00015 }
```

References [outputStyle](#), [simCode](#), and [SimCodeGenerator\(\)](#).

Here is the call graph for this function:



5.10.3 Member Function Documentation

5.10.3.1 generateOutputFolder()

```
void OutputManager::generateOutputFolder (
    const std::string & dir )
```

function that creates folder to save simulation data

Generate the folder to save the data to by one process: In the given directory it creates a directory "SimResults" and a directory with the simCode. The relevant part of the main file is written to a "config.txt" file in that directory to log the settings.

Definition at line 47 of file [Outputters.cpp](#).

```
00047     {
00048     // Do this only once for the first process
00049     int myPrc;
00050     MPI_Comm_rank(MPI_COMM_WORLD, &myPrc);
00051     if (myPrc == 0) {
00052         if (!fs::is_directory(dir))
00053             fs::create_directory(dir);
00054         if (!fs::is_directory(dir + "/SimResults"))
00055             fs::create_directory(dir + "/SimResults");
00056         if (!fs::is_directory(dir + "/SimResults/" + simCode))
00057             fs::create_directory(dir + "/SimResults/" + simCode);
00058     }
00059     // path variable for the output generation
00060     Path = dir + "/SimResults/" + simCode + "/";
00061
00062     // Logging configurations from main.cpp
00063     std::ifstream fin("main.cpp");
00064     std::ofstream fout(Path + "config.txt");
00065     std::string line;
00066     int begin=1000;
```



```

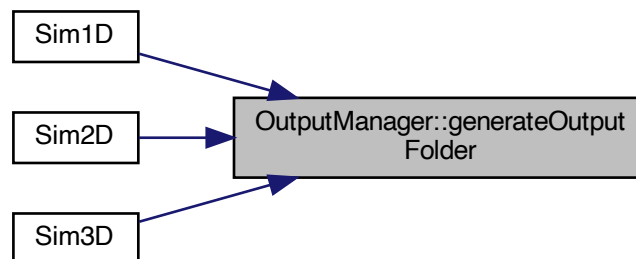
00067     for (int i = 1; !fin.eof(); i++) {
00068         getline(fin, line);
00069         if (line.starts_with("    //----- B")) {
00070             begin=i;
00071         }
00072         if (i < begin) {
00073             continue;
00074         }
00075         fout << line << std::endl;
00076         if (line.starts_with("    //----- E")) {
00077             break;
00078         }
00079     }
00080     return;
00081 }

```

References [Path](#), and [simCode](#).

Referenced by [Sim1D\(\)](#), [Sim2D\(\)](#), and [Sim3D\(\)](#).

Here is the caller graph for this function:



5.10.3.2 getSimCode()

```
const std::string & OutputManager::getSimCode ( ) const [inline]
```

simCode getter function

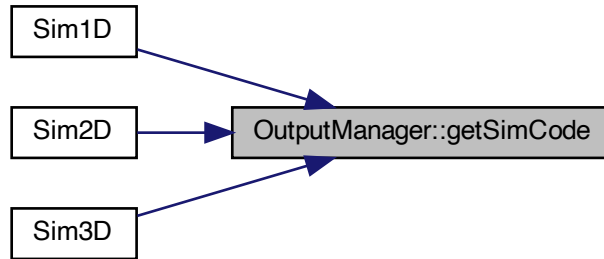
Definition at line 42 of file [Outputters.h](#).

```
00042 { return simCode; }
```

References [simCode](#).

Referenced by [Sim1D\(\)](#), [Sim2D\(\)](#), and [Sim3D\(\)](#).

Here is the caller graph for this function:



5.10.3.3 outUState()

```

void OutputManager::outUState (
    const int & state,
    const Lattice & lattice,
    const LatticePatch & latticePatch )
  
```

function to write data to disk in specified way

Write the field data either in csv format to one file per each process (patch) or in binary form to a single file. Files are stores inthe simCode directory. For csv files the state (simulation step) denotes the prefix and the suffix after an underscore is given by the process/patch number. Binary files are simply named after the step number.

Definition at line 92 of file [Outputters.cpp](#).

```

00093                                     {
00094     switch(outputStyle){
00095         case 'c': { // one csv file per process
00096             std::ofstream ofs;
00097             ofs.open(Path + std::to_string(state) + "_"
00098                 + std::to_string(lattice.my_prc) + ".csv");
00099             // Precision of sunrealtype in significant decimal digits; 15 for IEEE double
00100             ofs << std::setprecision(std::numeric_limits<sunrealtype>::digits10);
00101
00102             // Walk through each lattice point
00103             const sunindextype totalNP = latticePatch.discreteSize();
00104             for (sunindextype i = 0; i < totalNP * 6; i += 6) {
00105                 // Six columns to contain the field data: Ex,Ey,Ez,Bx,By,Bz
00106                 ofs << latticePatch.uData[i + 0] << "," << latticePatch.uData[i + 1] << ","
00107                     << latticePatch.uData[i + 2] << "," << latticePatch.uData[i + 3] << ","
00108                     << latticePatch.uData[i + 4] << "," << latticePatch.uData[i + 5]
00109                     << std::endl;
00110             }
00111             ofs.close();
00112             break;
00113         }
00114
00115         case 'b': { // a single binary file
00116             // Open the output file
00117             MPI_File fh;
00118             const std::string filename = Path+std::to_string(state);
00119             MPI_File_open(lattice.comm,&filename[0],MPI_MODE_WRONLY|MPI_MODE_CREATE,
00120                 MPI_INFO_NULL,&fh);
00121             // number of datapoints in the patch with process offset
00122             const sunindextype count = latticePatch.discreteSize()*
00123                 lattice.get_dataPointDimension();
00124             MPI_Offset offset = lattice.my_prc*count*sizeof(MPI_SUNREALTYPE);
  
```

```

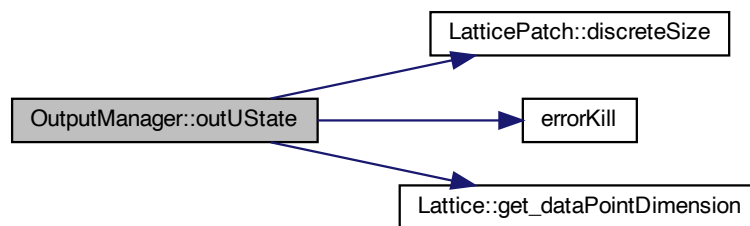
00125 // Go to offset in file and write data to it; maximal precision in
00126 // "native" representation
00127 MPI_File_set_view(fh,offset,MPI_SUNREALTYPE,MPI_SUNREALTYPE,"native",
00128 MPI_INFO_NULL);
00129 MPI_Request write_request;
00130 MPI_File_iwrite_all(fh,latticePatch.uData,count,MPI_SUNREALTYPE,
00131 &write_request);
00132 MPI_Wait(&write_request,MPI_STATUS_IGNORE);
00133 MPI_File_close(&fh);
00134 break;
00135 }
00136 default: {
00137 errorKill("No valid output style defined."
00138 " Choose between (c): one csv file per process,"
00139 " (b) one binary file");
00140 break;
00141 }
00142 }
00143 }

```

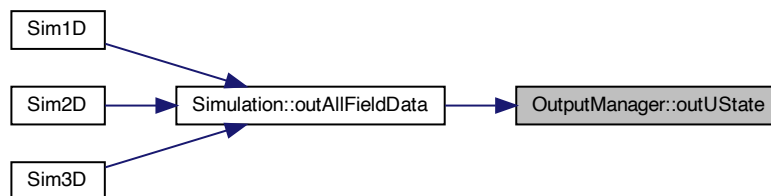
References [Lattice::comm](#), [LatticePatch::discreteSize\(\)](#), [errorKill\(\)](#), [Lattice::get_dataPointDimension\(\)](#), [Lattice::my_prc](#), [outputStyle](#), [Path](#), and [LatticePatch::uData](#).

Referenced by [Simulation::outAllFieldData\(\)](#).

Here is the call graph for this function:



Here is the caller graph for this function:



5.10.3.4 set_outputStyle()

```
void OutputManager::set_outputStyle (
    const char _outputStyle )
```

set the output style

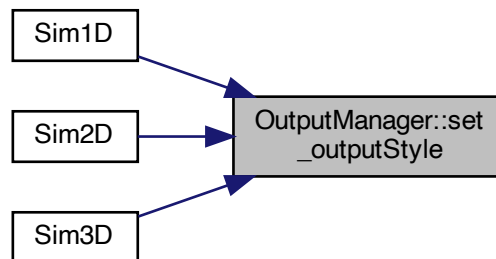
Definition at line 83 of file [Outputters.cpp](#).

```
00083                                     {
00084     outputStyle = _outputStyle;
00085 }
```

References [outputStyle](#).

Referenced by [Sim1D\(\)](#), [Sim2D\(\)](#), and [Sim3D\(\)](#).

Here is the caller graph for this function:



5.10.3.5 SimCodeGenerator()

```
std::string OutputManager::SimCodeGenerator ( ) [static], [private]
```

function to create the Code of the Simulations

Generate the identifier number reverse from year to minute in the format yy-mm-dd_hh-MM-ss

Definition at line 19 of file [Outputters.cpp](#).

```
00019                                     {
00020     const chrono::time_point<chrono::system_clock> now{
00021         chrono::system_clock::now() };
00022     const chrono::year_month_day ymd{chrono::floor<chrono::days>(now) };
00023     const auto tod = now - chrono::floor<chrono::days>(now);
00024     const chrono::hh_mm_ss hms{tod};
00025
00026     std::stringstream temp;
00027     temp << std::setfill('0') << std::setw(2)
00028         << static_cast<int>(ymd.year() - chrono::years(2000)) << "-"
00029         << std::setfill('0') << std::setw(2)
00030         << static_cast<unsigned>(ymd.month()) << "-"
00031         << std::setfill('0') << std::setw(2)
00032         << static_cast<unsigned>(ymd.day()) << "_"
00033         << std::setfill('0') << std::setw(2) << hms.hours().count()
00034         << "-" << std::setfill('0') << std::setw(2) << hms.minutes().count() << "-"
00035         << std::setfill('0') << std::setw(2) << hms.seconds().count();
```

```

00035         « std::setw(2) « hms.minutes().count() « "-"
00036         « std::setfill('0') « std::setw(2)
00037         « hms.seconds().count();
00038         //« "-" « hms.subseconds().count(); // subseconds render the filename
00039         // too large
00040         return temp.str();
00041     }

```

Referenced by [OutputManager\(\)](#).

Here is the caller graph for this function:



5.10.4 Field Documentation

5.10.4.1 outputStyle

```
char OutputManager::outputStyle [private]
```

output style; csv or binary

Definition at line 30 of file [Outputters.h](#).

Referenced by [OutputManager\(\)](#), [outUState\(\)](#), and [set_outputStyle\(\)](#).

5.10.4.2 Path

```
std::string OutputManager::Path [private]
```

variable for the path to the output folder

Definition at line 28 of file [Outputters.h](#).

Referenced by [generateOutputFolder\(\)](#), and [outUState\(\)](#).

5.10.4.3 simCode

```
std::string OutputManager::simCode [private]
```

variable to save the SimCode generated at execution

Definition at line 26 of file [Outputters.h](#).

Referenced by [generateOutputFolder\(\)](#), [getSimCode\(\)](#), and [OutputManager\(\)](#).

The documentation for this class was generated from the following files:

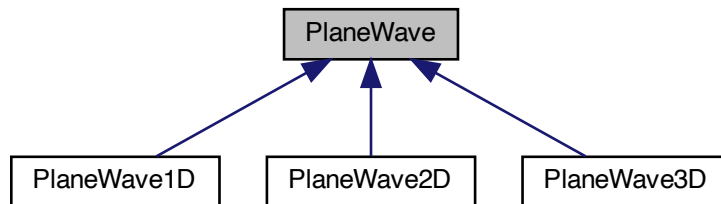
- [src/Outputters.h](#)
- [src/Outputters.cpp](#)

5.11 PlaneWave Class Reference

super-class for plane waves

```
#include <src/ICSetters.h>
```

Inheritance diagram for PlaneWave:



Protected Attributes

- sunrealtype [kx](#)
wavenumber k_x
- sunrealtype [ky](#)
wavenumber k_y
- sunrealtype [kz](#)
wavenumber k_z
- sunrealtype [px](#)
polarization & amplitude in x-direction, p_x
- sunrealtype [py](#)
polarization & amplitude in y-direction, p_y
- sunrealtype [pz](#)
polarization & amplitude in z-direction, p_z
- sunrealtype [phix](#)
phase shift in x-direction, ϕ_x
- sunrealtype [phiy](#)
phase shift in y-direction, ϕ_y
- sunrealtype [phiz](#)
phase shift in z-direction, ϕ_z

5.11.1 Detailed Description

super-class for plane waves

They are given in the form $\vec{E} = \vec{E}_0 \cos(\vec{k} \cdot \vec{x} - \phi)$

Definition at line 20 of file [ICSetters.h](#).

5.11.2 Field Documentation

5.11.2.1 kx

```
sunrealtype PlaneWave::kx [protected]
```

wavenumber k_x

Definition at line 23 of file [ICSetters.h](#).

Referenced by [PlaneWave1D::addToSpace\(\)](#), [PlaneWave2D::addToSpace\(\)](#), [PlaneWave3D::addToSpace\(\)](#), [PlaneWave1D::PlaneWave1D\(\)](#), [PlaneWave2D::PlaneWave2D\(\)](#), and [PlaneWave3D::PlaneWave3D\(\)](#).

5.11.2.2 ky

```
sunrealtype PlaneWave::ky [protected]
```

wavenumber k_y

Definition at line 25 of file [ICSetters.h](#).

Referenced by [PlaneWave1D::addToSpace\(\)](#), [PlaneWave2D::addToSpace\(\)](#), [PlaneWave3D::addToSpace\(\)](#), [PlaneWave1D::PlaneWave1D\(\)](#), [PlaneWave2D::PlaneWave2D\(\)](#), and [PlaneWave3D::PlaneWave3D\(\)](#).

5.11.2.3 kz

```
sunrealtype PlaneWave::kz [protected]
```

wavenumber k_z

Definition at line 27 of file [ICSetters.h](#).

Referenced by [PlaneWave1D::addToSpace\(\)](#), [PlaneWave2D::addToSpace\(\)](#), [PlaneWave3D::addToSpace\(\)](#), [PlaneWave1D::PlaneWave1D\(\)](#), [PlaneWave2D::PlaneWave2D\(\)](#), and [PlaneWave3D::PlaneWave3D\(\)](#).

5.11.2.4 phix

```
sunrealtype PlaneWave::phix [protected]
```

phase shift in x-direction, ϕ_x

Definition at line 35 of file [ICSetters.h](#).

Referenced by [PlaneWave1D::addToSpace\(\)](#), [PlaneWave2D::addToSpace\(\)](#), [PlaneWave3D::addToSpace\(\)](#), [PlaneWave1D::PlaneWave1D\(\)](#), [PlaneWave2D::PlaneWave2D\(\)](#), and [PlaneWave3D::PlaneWave3D\(\)](#).

5.11.2.5 phiy

```
sunrealtype PlaneWave::phiy [protected]
```

phase shift in y-direction, ϕ_y

Definition at line 37 of file [ICSetters.h](#).

Referenced by [PlaneWave1D::addToSpace\(\)](#), [PlaneWave2D::addToSpace\(\)](#), [PlaneWave3D::addToSpace\(\)](#), [PlaneWave1D::PlaneWave1D\(\)](#), [PlaneWave2D::PlaneWave2D\(\)](#), and [PlaneWave3D::PlaneWave3D\(\)](#).

5.11.2.6 phiz

```
sunrealtype PlaneWave::phiz [protected]
```

phase shift in z-direction, ϕ_z

Definition at line 39 of file [ICSetters.h](#).

Referenced by [PlaneWave1D::addToSpace\(\)](#), [PlaneWave2D::addToSpace\(\)](#), [PlaneWave3D::addToSpace\(\)](#), [PlaneWave1D::PlaneWave1D\(\)](#), [PlaneWave2D::PlaneWave2D\(\)](#), and [PlaneWave3D::PlaneWave3D\(\)](#).

5.11.2.7 px

```
sunrealtype PlaneWave::px [protected]
```

polarization & amplitude in x-direction, p_x

Definition at line 29 of file [ICSetters.h](#).

Referenced by [PlaneWave1D::addToSpace\(\)](#), [PlaneWave2D::addToSpace\(\)](#), [PlaneWave3D::addToSpace\(\)](#), [PlaneWave1D::PlaneWave1D\(\)](#), [PlaneWave2D::PlaneWave2D\(\)](#), and [PlaneWave3D::PlaneWave3D\(\)](#).

5.11.2.8 py

`sunrealtype PlaneWave::py` [protected]

polarization & amplitude in y-direction, p_y

Definition at line 31 of file [ICSetters.h](#).

Referenced by [PlaneWave1D::addToSpace\(\)](#), [PlaneWave2D::addToSpace\(\)](#), [PlaneWave3D::addToSpace\(\)](#), [PlaneWave1D::PlaneWave1D\(\)](#), [PlaneWave2D::PlaneWave2D\(\)](#), and [PlaneWave3D::PlaneWave3D\(\)](#).

5.11.2.9 pz

`sunrealtype PlaneWave::pz` [protected]

polarization & amplitude in z-direction, p_z

Definition at line 33 of file [ICSetters.h](#).

Referenced by [PlaneWave1D::addToSpace\(\)](#), [PlaneWave2D::addToSpace\(\)](#), [PlaneWave3D::addToSpace\(\)](#), [PlaneWave1D::PlaneWave1D\(\)](#), [PlaneWave2D::PlaneWave2D\(\)](#), and [PlaneWave3D::PlaneWave3D\(\)](#).

The documentation for this class was generated from the following file:

- [src/ICSetters.h](#)

5.12 planewave Struct Reference

plane wave structure

```
#include <src/SimulationFunctions.h>
```

Data Fields

- `std::array< sunrealtype, 3 >` [k](#)
- `std::array< sunrealtype, 3 >` [p](#)
- `std::array< sunrealtype, 3 >` [phi](#)

5.12.1 Detailed Description

plane wave structure

Definition at line 19 of file [SimulationFunctions.h](#).

5.12.2 Field Documentation

5.12.2.1 k

```
std::array<sunrealtype, 3> planewave::k
```

wavevector (normalized to $1/\lambda$)

Definition at line 20 of file [SimulationFunctions.h](#).

5.12.2.2 p

```
std::array<sunrealtype, 3> planewave::p
```

amplitde & polarization vector

Definition at line 21 of file [SimulationFunctions.h](#).

5.12.2.3 phi

```
std::array<sunrealtype, 3> planewave::phi
```

phase shift

Definition at line 22 of file [SimulationFunctions.h](#).

The documentation for this struct was generated from the following file:

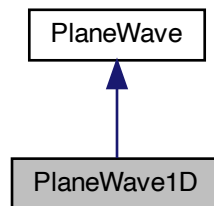
- [src/SimulationFunctions.h](#)

5.13 PlaneWave1D Class Reference

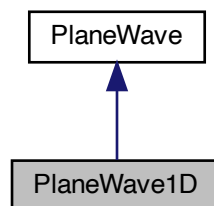
class for plane waves in 1D

```
#include <src/ICSetters.h>
```

Inheritance diagram for PlaneWave1D:



Collaboration diagram for PlaneWave1D:



Public Member Functions

- [PlaneWave1D](#) (std::array< sunrealtype, 3 > k={1, 0, 0}, std::array< sunrealtype, 3 > p={0, 0, 1}, std::array< sunrealtype, 3 > phi={0, 0, 0})
construction with default parameters
- void [addToSpace](#) (sunrealtype x, sunrealtype y, sunrealtype z, sunrealtype *pTo6Space) const
function for the actual implementation in the lattice

Additional Inherited Members

5.13.1 Detailed Description

class for plane waves in 1D

Definition at line 43 of file [ICSetters.h](#).

5.13.2 Constructor & Destructor Documentation

5.13.2.1 PlaneWave1D()

```
PlaneWave1D::PlaneWave1D (
    std::array< sunrealtype, 3 > k = {1, 0, 0},
    std::array< sunrealtype, 3 > p = {0, 0, 1},
    std::array< sunrealtype, 3 > phi = {0, 0, 0} )
```

construction with default parameters

[PlaneWave1D](#) construction with

- wavevectors k_x
- k_y
- k_z normalized to $1/\lambda$
- amplitude (polarization) in x-direction p_x
- amplitude (polarization) in y-direction p_y
- amplitude (polarization) in z-direction p_z
- phase shift in x-direction ϕ_x
- phase shift in y-direction ϕ_y
- phase shift in z-direction ϕ_z

Definition at line 11 of file [ICSetters.cpp](#).

```
00013 {
00014     kx = k[0]; /** - wavevectors \f$ k_x \f$ */
00015     ky = k[1]; /** - \f$ k_y \f$ */
00016     kz = k[2]; /** - \f$ k_z \f$ normalized to \f$ 1/\lambda \f$ */
00017     // Amplitude bug: lower by factor 3
00018     px = p[0] / 3; /** - amplitude (polarization) in x-direction \f$ p_x \f$ */
00019     py = p[1] / 3; /** - amplitude (polarization) in y-direction \f$ p_y \f$ */
00020     pz = p[2] / 3; /** - amplitude (polarization) in z-direction \f$ p_z \f$ */
00021     phix = phi[0]; /** - phase shift in x-direction \f$ \phi_x \f$ */
00022     phiy = phi[1]; /** - phase shift in y-direction \f$ \phi_y \f$ */
00023     phiz = phi[2]; /** - phase shift in z-direction \f$ \phi_z \f$ */
00024 }
```

References [PlaneWave::kx](#), [PlaneWave::ky](#), [PlaneWave::kz](#), [PlaneWave::phix](#), [PlaneWave::phiy](#), [PlaneWave::phiz](#), [PlaneWave::px](#), [PlaneWave::py](#), and [PlaneWave::pz](#).

5.13.3 Member Function Documentation

5.13.3.1 addToSpace()

```
void PlaneWave1D::addToSpace (
    sunrealtype x,
    sunrealtype y,
    sunrealtype z,
    sunrealtype * pTo6Space ) const
```

function for the actual implementation in the lattice

[PlaneWave1D](#) implementation in space

Definition at line 27 of file [ICSetters.cpp](#).

```
00029 {
00030     const sunrealtype wavelength =
00031         sqrt(kx * kx + ky * ky + kz * kz); /* \f$ 1/\lambda \f$ */
00032     const sunrealtype kScalarX = (kx * x + ky * y + kz * z) * 2 *
00033         std::numbers::pi; /* \f$ 2\pi \ \vec{k} \cdot \vec{x} \f$ */
00034     // Plane wave definition
00035     const std::array<sunrealtype, 3> E{{ /* E-field vector */
00036         px * cos(kScalarX - phix), /* \f$ E_x \f$ */
00037         py * cos(kScalarX - phiy), /* \f$ E_y \f$ */
00038         pz * cos(kScalarX - phiz)}}; /* \f$ E_z \f$ */
00039     // Put E-field into space
00040     pTo6Space[0] += E[0];
00041     pTo6Space[1] += E[1];
00042     pTo6Space[2] += E[2];
00043     // and B-field
00044     pTo6Space[3] += (ky * E[2] - kz * E[1]) / wavelength;
00045     pTo6Space[4] += (kz * E[0] - kx * E[2]) / wavelength;
00046     pTo6Space[5] += (kx * E[1] - ky * E[0]) / wavelength;
00047 }
```

References [PlaneWave::kx](#), [PlaneWave::ky](#), [PlaneWave::kz](#), [PlaneWave::phix](#), [PlaneWave::phiy](#), [PlaneWave::phiz](#), [PlaneWave::px](#), [PlaneWave::py](#), and [PlaneWave::pz](#).

The documentation for this class was generated from the following files:

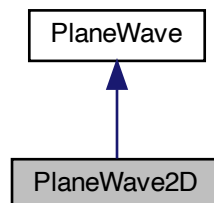
- [src/ICSetters.h](#)
- [src/ICSetters.cpp](#)

5.14 PlaneWave2D Class Reference

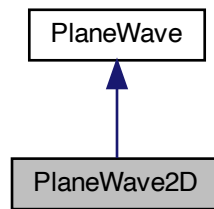
class for plane waves in 2D

```
#include <src/ICSetters.h>
```

Inheritance diagram for PlaneWave2D:



Collaboration diagram for PlaneWave2D:



Public Member Functions

- [PlaneWave2D](#) (`std::array< sunrealtype, 3 > k={1, 0, 0}`, `std::array< sunrealtype, 3 > p={0, 0, 1}`, `std::array< sunrealtype, 3 > phi={0, 0, 0}`)
construction with default parameters
- void [addToSpace](#) (`sunrealtype x`, `sunrealtype y`, `sunrealtype z`, `sunrealtype *pTo6Space`) const
function for the actual implementation in the lattice

Additional Inherited Members

5.14.1 Detailed Description

class for plane waves in 2D

Definition at line 55 of file [ICSetters.h](#).

5.14.2 Constructor & Destructor Documentation

5.14.2.1 PlaneWave2D()

```

PlaneWave2D::PlaneWave2D (
    std::array< sunrealtype, 3 > k = {1, 0, 0},
    std::array< sunrealtype, 3 > p = {0, 0, 1},
    std::array< sunrealtype, 3 > phi = {0, 0, 0} )

```

construction with default parameters

[PlaneWave2D](#) construction with

- wavevectors k_x

- k_y
- k_z normalized to $1/\lambda$
- amplitude (polarization) in x-direction p_x
- amplitude (polarization) in y-direction p_y
- amplitude (polarization) in z-direction p_z
- phase shift in x-direction ϕ_x
- phase shift in y-direction ϕ_y
- phase shift in z-direction ϕ_z

Definition at line 50 of file [ICSetters.cpp](#).

```
00052 {
00053     kx = k[0]; /** - wavevectors \f$ k_x \f$ */
00054     ky = k[1]; /** - \f$ k_y \f$ */
00055     kz = k[2]; /** - \f$ k_z \f$ normalized to \f$ 1/\lambda \f$ */
00056     // Amplitude bug: lower by factor 9
00057     px = p[0] / 9; /** - amplitude (polarization) in x-direction \f$ p_x \f$ */
00058     py = p[1] / 9; /** - amplitude (polarization) in y-direction \f$ p_y \f$ */
00059     pz = p[2] / 9; /** - amplitude (polarization) in z-direction \f$ p_z \f$ */
00060     phix = phi[0]; /** - phase shift in x-direction \f$ \phi_x \f$ */
00061     phiy = phi[1]; /** - phase shift in y-direction \f$ \phi_y \f$ */
00062     phiz = phi[2]; /** - phase shift in z-direction \f$ \phi_z \f$ */
00063 }
```

References [PlaneWave::kx](#), [PlaneWave::ky](#), [PlaneWave::kz](#), [PlaneWave::phix](#), [PlaneWave::phiy](#), [PlaneWave::phiz](#), [PlaneWave::px](#), [PlaneWave::py](#), and [PlaneWave::pz](#).

5.14.3 Member Function Documentation

5.14.3.1 addToSpace()

```
void PlaneWave2D::addToSpace (
    sunrealtype x,
    sunrealtype y,
    sunrealtype z,
    sunrealtype * pTo6Space ) const
```

function for the actual implementation in the lattice

[PlaneWave2D](#) implementation in space

Definition at line 66 of file [ICSetters.cpp](#).

```
00067 {
00068     const sunrealtype wavelength =
00069         sqrt(kx * kx + ky * ky + kz * kz); /** \f$ 1/\lambda \f$ */
00070     const sunrealtype kScalarX = (kx * x + ky * y + kz * z) * 2 *
00071         std::numbers::pi; /** \f$ 2\pi \cdot \vec{k} \cdot \vec{x} \f$ */
00072     // Plane wave definition
00073     const std::array<sunrealtype, 3> E{{          /** E-field vector */
00074         px * cos(kScalarX - phix),          /** \f$ E_x \f$ */
00075         py * cos(kScalarX - phiy),          /** \f$ E_y \f$ */
00076         pz * cos(kScalarX - phiz)}}; /** \f$ E_z \f$ */
00077     // Put E-field into space
00078     pTo6Space[0] += E[0];
00079     pTo6Space[1] += E[1];
00080     pTo6Space[2] += E[2];
00081     // and B-field
00082     pTo6Space[3] += (ky * E[2] - kz * E[1]) / wavelength;
```

```
00083   pTo6Space[4] += (kz * E[0] - kx * E[2]) / wavelength;  
00084   pTo6Space[5] += (kx * E[1] - ky * E[0]) / wavelength;  
00085 }
```

References [PlaneWave::kx](#), [PlaneWave::ky](#), [PlaneWave::kz](#), [PlaneWave::phix](#), [PlaneWave::phiy](#), [PlaneWave::phiz](#), [PlaneWave::px](#), [PlaneWave::py](#), and [PlaneWave::pz](#).

The documentation for this class was generated from the following files:

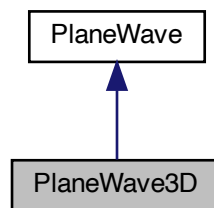
- [src/ICSetters.h](#)
- [src/ICSetters.cpp](#)

5.15 PlaneWave3D Class Reference

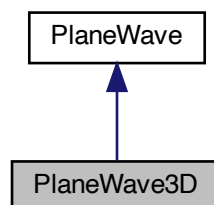
class for plane waves in 3D

```
#include <src/ICSetters.h>
```

Inheritance diagram for PlaneWave3D:



Collaboration diagram for PlaneWave3D:



Public Member Functions

- [PlaneWave3D](#) (std::array< sunrealtype, 3 > k={1, 0, 0}, std::array< sunrealtype, 3 > p={0, 0, 1}, std::array< sunrealtype, 3 > phi={0, 0, 0})
construction with default parameters
- void [addToSpace](#) (sunrealtype x, sunrealtype y, sunrealtype z, sunrealtype *pTo6Space) const
function for the actual implementation in space

Additional Inherited Members

5.15.1 Detailed Description

class for plane waves in 3D

Definition at line 67 of file [ICSetters.h](#).

5.15.2 Constructor & Destructor Documentation

5.15.2.1 PlaneWave3D()

```
PlaneWave3D::PlaneWave3D (
    std::array< sunrealtype, 3 > k = {1, 0, 0},
    std::array< sunrealtype, 3 > p = {0, 0, 1},
    std::array< sunrealtype, 3 > phi = {0, 0, 0} )
```

construction with default parameters

[PlaneWave3D](#) construction with

- wavevectors k_x
- k_y
- k_z normalized to $1/\lambda$
- amplitude (polarization) in x-direction p_x
- amplitude (polarization) in y-direction p_y
- amplitude (polarization) in z-direction p_z
- phase shift in x-direction ϕ_x
- phase shift in y-direction ϕ_y
- phase shift in z-direction ϕ_z

Definition at line 88 of file [ICSetters.cpp](#).

```
00090 {
00091     kx = k[0];    /** - wavevectors \f$ k_x \f$ */
00092     ky = k[1];    /** - \f$ k_y \f$ */
00093     kz = k[2];    /** - \f$ k_z \f$ normalized to \f$ 1/\lambda \f$ */
00094     px = p[0];    /** - amplitude (polarization) in x-direction \f$ p_x \f$ */
00095     py = p[1];    /** - amplitude (polarization) in y-direction \f$ p_y \f$ */
00096     pz = p[2];    /** - amplitude (polarization) in z-direction \f$ p_z \f$ */
00097     phix = phi[0]; /** - phase shift in x-direction \f$ \phi_x \f$ */
00098     phiy = phi[1]; /** - phase shift in y-direction \f$ \phi_y \f$ */
00099     phiz = phi[2]; /** - phase shift in z-direction \f$ \phi_z \f$ */
00100 }
```

References [PlaneWave::kx](#), [PlaneWave::ky](#), [PlaneWave::kz](#), [PlaneWave::phix](#), [PlaneWave::phiy](#), [PlaneWave::phiz](#), [PlaneWave::px](#), [PlaneWave::py](#), and [PlaneWave::pz](#).

5.15.3 Member Function Documentation

5.15.3.1 addToSpace()

```
void PlaneWave3D::addToSpace (
    sunrealtype x,
    sunrealtype y,
    sunrealtype z,
    sunrealtype * pTo6Space ) const
```

function for the actual implementation in space

[PlaneWave3D](#) implementation in space

Definition at line 103 of file [ICSetters.cpp](#).

```
00104 {
00105     const sunrealtype wavelength =
00106         sqrt(kx * kx + ky * ky + kz * kz); /* \f$ 1/\lambda \f$ */
00107     const sunrealtype kScalarX = (kx * x + ky * y + kz * z) * 2 *
00108         std::numbers::pi; /* \f$ 2\pi \vec{k} \cdot \vec{x} \f$ */
00109     // Plane wave definition
00110     const std::array<sunrealtype, 3> E{ /* E-field vector \f$ \vec{E} \f$ */
00111         px * cos(kScalarX - phix), /* \f$ E_x \f$ */
00112         py * cos(kScalarX - phiy), /* \f$ E_y \f$ */
00113         pz * cos(kScalarX - phiz) }; /* \f$ E_z \f$ */
00114     // Put E-field into space
00115     pTo6Space[0] += E[0];
00116     pTo6Space[1] += E[1];
00117     pTo6Space[2] += E[2];
00118     // and B-field
00119     pTo6Space[3] += (ky * E[2] - kz * E[1]) / wavelength;
00120     pTo6Space[4] += (kz * E[0] - kx * E[2]) / wavelength;
00121     pTo6Space[5] += (kx * E[1] - ky * E[0]) / wavelength;
00122 }
```

References [PlaneWave::kx](#), [PlaneWave::ky](#), [PlaneWave::kz](#), [PlaneWave::phix](#), [PlaneWave::phiy](#), [PlaneWave::phiz](#), [PlaneWave::px](#), [PlaneWave::py](#), and [PlaneWave::pz](#).

The documentation for this class was generated from the following files:

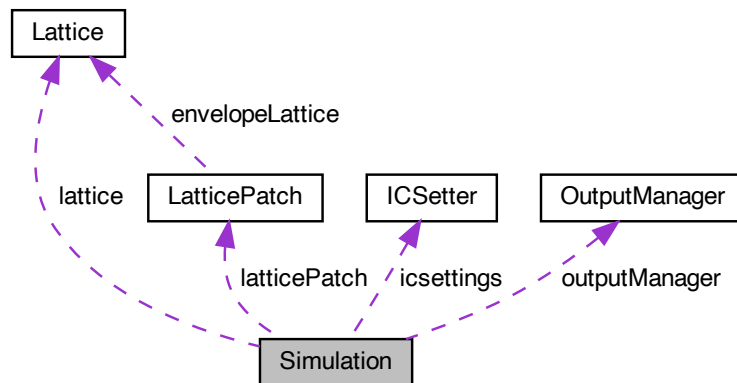
- [src/ICSetters.h](#)
- [src/ICSetters.cpp](#)

5.16 Simulation Class Reference

[Simulation](#) class to instantiate the whole walkthrough of a [Simulation](#).

```
#include <src/SimulationClass.h>
```

Collaboration diagram for Simulation:



Public Member Functions

- [Simulation](#) (const int Nx, const int Ny, const int Nz, const int StencilOrder, const bool periodicity)
constructor function for the creation of the cartesian communicator
- [~Simulation](#) ()
destructor function freeing C_Vode memory and Sundials context
- MPI_Comm * [get_cart_comm](#) ()
reference to the cartesian communicator of the lattice (for debugging)
- void [setDiscreteDimensionsOfLattice](#) (const sunindextype _tot_nx, const sunindextype _tot_ny, const sunindextype _tot_nz)
function to set discrete dimensions of the lattice
- void [setPhysicalDimensionsOfLattice](#) (const sunrealtype lx, const sunrealtype ly, const sunrealtype lz)
function to set physical dimensions of the lattice
- void [initializePatchwork](#) (const int nx, const int ny, const int nz)
function to initialize the Patchwork
- void [initializeC_VODEobject](#) (const sunrealtype reltol, const sunrealtype abstol)
function to initialize the C_VODE object with all requirements
- void [start](#) ()
function to start the simulation for time iteration
- void [setInitialConditions](#) ()
functions to set the initial field configuration onto the lattice
- void [addInitialConditions](#) (const sunindextype xm, const sunindextype ym, const sunindextype zm=0)
functions to add initial periodic field configurations
- void [addPeriodicICLayerInX](#) ()
function to add a periodic IC layer in one dimension
- void [addPeriodicICLayerInXY](#) ()
function to add periodic IC layers in two dimensions
- void [advanceToTime](#) (const sunrealtype &tEnd)
function to advance solution in time with C_VODE
- void [outAllFieldData](#) (const int &state)
function to write field data to disk

- void [checkFlag](#) (unsigned int flag) const
function to check if flag has been set
- void [checkNoFlag](#) (unsigned int flag) const
function to check if flag has not been set

Data Fields

- [ICSetter icsettings](#)
IC Setter object.
- [OutputManager outputManager](#)
Output Manager object.
- void * [cnode_mem](#)
pointer to CNode memory object
- SUNNonlinearSolver [NLS](#)
nonlinear solver object

Private Attributes

- [Lattice lattice](#)
Lattice object.
- [LatticePatch latticePatch](#)
LatticePatch object.
- sunrealtype [t](#)
current time of the simulation
- unsigned int [statusFlags](#)
simulation status flags

5.16.1 Detailed Description

[Simulation](#) class to instantiate the whole walkthrough of a [Simulation](#).

Definition at line 30 of file [SimulationClass.h](#).

5.16.2 Constructor & Destructor Documentation

5.16.2.1 Simulation()

```
Simulation::Simulation (
    const int Nx,
    const int Ny,
    const int Nz,
    const int StencilOrder,
    const bool periodicity )
```

constructor function for the creation of the cartesian communicator

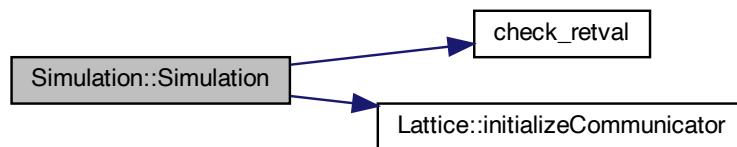
Along with the simulation object, create the cartesian communicator and SUNContext object

Definition at line 14 of file [SimulationClass.cpp](#).

```
00015                                     :
00016     lattice(StencilOrder){
00017     statusFlags = 0;
00018     t = 0;
00019     // Initialize the cartesian communicator
00020     lattice.initializeCommunicator(Nx, Ny, Nz, periodicity);
00021
00022     // Create the SUNContext object associated with the thread of execution
00023     int retval = 0;
00024     retval = SUNContext_Create(&lattice.comm, &lattice.sunctx);
00025     if (check_retval(&retval, "SUNContext_Create", 1, lattice.my_prc))
00026         MPI_Abort(lattice.comm, 1);
00027 }
```

References [check_retval\(\)](#), [Lattice::comm](#), [Lattice::initializeCommunicator\(\)](#), [lattice](#), [Lattice::my_prc](#), [statusFlags](#), [Lattice::sunctx](#), and [t](#).

Here is the call graph for this function:



5.16.2.2 ~Simulation()

```
Simulation::~Simulation ( )
```

destructor function freeing CNode memory and Sundials context

Free the CNode solver memory and Sundials context object with the finish of the simulation

Definition at line 31 of file [SimulationClass.cpp](#).

```
00031     {
00032     // Free solver memory
00033     if (statusFlags & CnodeObjectSetup) {
00034         CNodeFree(&cnode_mem);
00035         SUNNonlinSolFree(NLS);
00036         SUNContext_Free(&lattice.sunctx);
00037     }
00038 }
```

References [cnode_mem](#), [CnodeObjectSetup](#), [lattice](#), [NLS](#), [statusFlags](#), and [Lattice::sunctx](#).

5.16.3 Member Function Documentation

5.16.3.1 addInitialConditions()

```
void Simulation::addInitialConditions (
    const sunindextype xm,
    const sunindextype ym,
    const sunindextype zm = 0 )
```

functions to add initial periodic field configurations

Use parameters to add periodic IC layers.

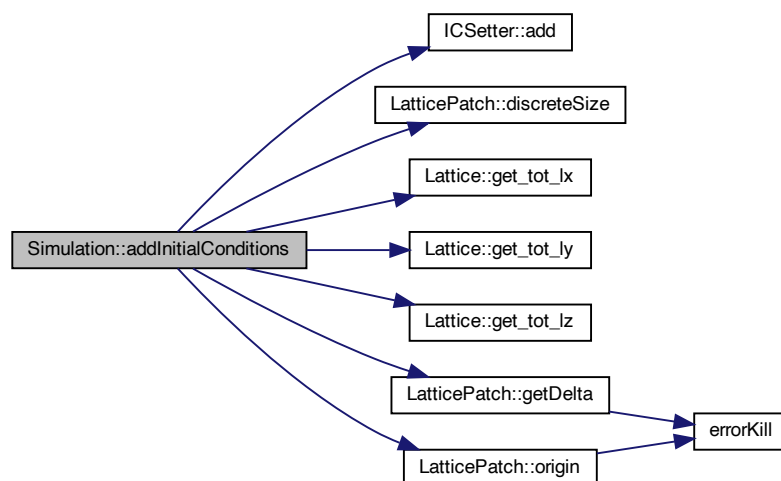
Definition at line 167 of file [SimulationClass.cpp](#).

```
00169 {
00170     const sunrealtype dx = latticePatch.getDelta(1);
00171     const sunrealtype dy = latticePatch.getDelta(2);
00172     const sunrealtype dz = latticePatch.getDelta(3);
00173     const sunindextype nx = latticePatch.discreteSize(1);
00174     const sunindextype ny = latticePatch.discreteSize(2);
00175     const sunindextype totalNP = latticePatch.discreteSize();
00176     // Correct for demanded displacement, rest as for setInitialConditions
00177     const sunrealtype x0 = latticePatch.origin(1) + xm*lattice.get_tot_lx();
00178     const sunrealtype y0 = latticePatch.origin(2) + ym*lattice.get_tot_ly();
00179     const sunrealtype z0 = latticePatch.origin(3) + zm*lattice.get_tot_lz();
00180     sunindextype px = 0, py = 0, pz = 0;
00181     for (sunindextype i = 0; i < totalNP * 6; i += 6) {
00182         px = (i / 6) % nx;
00183         py = ((i / 6) / nx) % ny;
00184         pz = ((i / 6) / nx) / ny;
00185         icsettings.add(static_cast<sunrealtype>(px) * dx + x0,
00186                     static_cast<sunrealtype>(py) * dy + y0,
00187                     static_cast<sunrealtype>(pz) * dz + z0, &latticePatch.uData[i]);
00188     }
00189     return;
00190 }
```

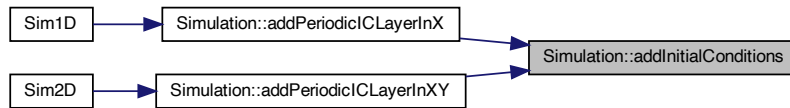
References [ICSetter::add\(\)](#), [LatticePatch::discreteSize\(\)](#), [Lattice::get_tot_lx\(\)](#), [Lattice::get_tot_ly\(\)](#), [Lattice::get_tot_lz\(\)](#), [LatticePatch::getDelta\(\)](#), [icsettings](#), [lattice](#), [latticePatch](#), [LatticePatch::origin\(\)](#), and [LatticePatch::uData](#).

Referenced by [addPeriodicICLayerInX\(\)](#), and [addPeriodicICLayerInXY\(\)](#).

Here is the call graph for this function:



Here is the caller graph for this function:



5.16.3.2 addPeriodicICLayerInX()

```
void Simulation::addPeriodicICLayerInX ( )
```

function to add a periodic IC layer in one dimension

Add initial conditions in one dimension.

Definition at line 193 of file [SimulationClass.cpp](#).

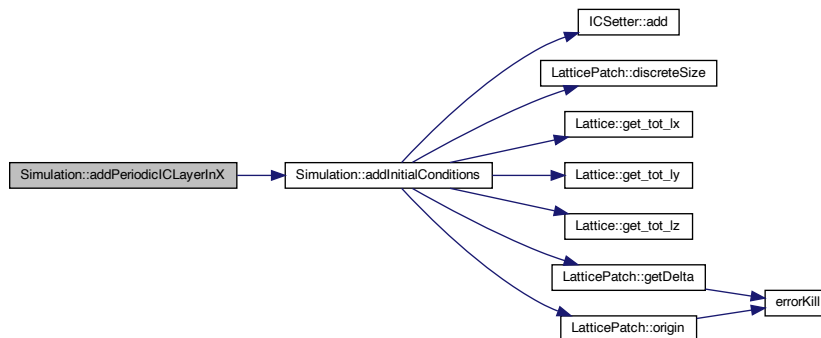
```

00193 {
00194     addInitialConditions(-1, 0, 0);
00195     addInitialConditions(1, 0, 0);
00196     return;
00197 }
```

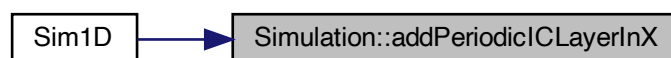
References [addInitialConditions\(\)](#).

Referenced by [Sim1D\(\)](#).

Here is the call graph for this function:



Here is the caller graph for this function:



5.16.3.3 addPeriodicICLayerInXY()

```
void Simulation::addPeriodicICLayerInXY ( )
```

function to add periodic IC layers in two dimensions

Add initial conditions in two dimensions.

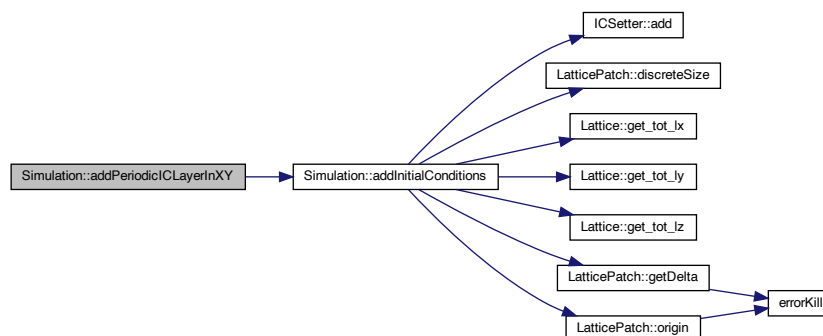
Definition at line 200 of file [SimulationClass.cpp](#).

```
00200 {
00201     addInitialConditions(-1, -1, 0);
00202     addInitialConditions(-1, 0, 0);
00203     addInitialConditions(-1, 1, 0);
00204     addInitialConditions(0, 1, 0);
00205     addInitialConditions(0, -1, 0);
00206     addInitialConditions(1, -1, 0);
00207     addInitialConditions(1, 0, 0);
00208     addInitialConditions(1, 1, 0);
00209     return;
00210 }
```

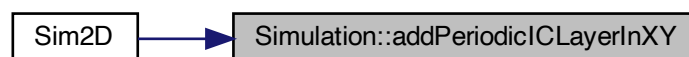
References [addInitialConditions\(\)](#).

Referenced by [Sim2D\(\)](#).

Here is the call graph for this function:



Here is the caller graph for this function:



5.16.3.4 advanceToTime()

```
void Simulation::advanceToTime (
    const sunrealtype & tEnd )
```

function to advance solution in time with CVODE

Advance the solution in time -> integrate the ODE over an interval t.

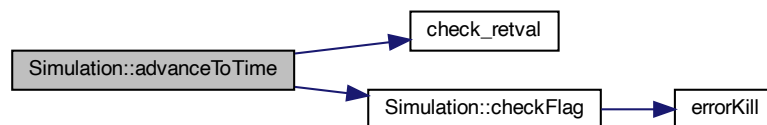
Definition at line 213 of file [SimulationClass.cpp](#).

```
00213 {
00214     checkFlag(SimulationStarted);
00215     int retval = 0;
00216     retval = CCode(cvode_mem, tEnd, latticePatch.u, &t,
00217                  CV_NORMAL); // CV_NORMAL: internal steps to reach tEnd, then
00218                             // interpolate to return latticePatch.u, return time
00219                             // reached by the solver as t
00220     if (check_retval(&retval, "Cvode", 1, lattice.my_prc))
00221         MPI_Abort(lattice.comm, 1);
00222 }
```

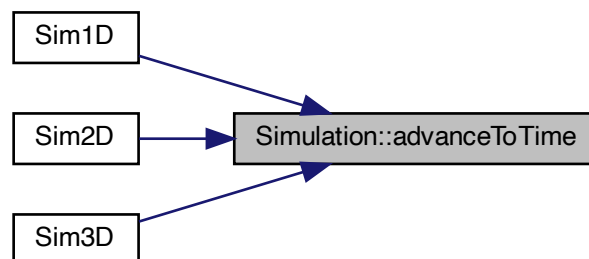
References [check_retval\(\)](#), [checkFlag\(\)](#), [Lattice::comm](#), [cvode_mem](#), [lattice](#), [latticePatch](#), [Lattice::my_prc](#), [SimulationStarted](#), [t](#), and [LatticePatch::u](#).

Referenced by [Sim1D\(\)](#), [Sim2D\(\)](#), and [Sim3D\(\)](#).

Here is the call graph for this function:



Here is the caller graph for this function:



5.16.3.5 checkFlag()

```
void Simulation::checkFlag (
    unsigned int flag ) const
```

function to check if flag has been set

Check presence of configuration flags.

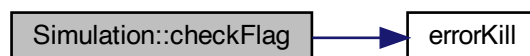
Definition at line 231 of file [SimulationClass.cpp](#).

```
00231 {
00232     if (!(statusFlags & flag)) {
00233         std::string errorMessage;
00234         switch (flag) {
00235             case LatticeDiscreteSetUp:
00236                 errorMessage = "The discrete size of the Simulation has not been set up";
00237                 break;
00238             case LatticePhysicalSetUp:
00239                 errorMessage = "The physical size of the Simulation has not been set up";
00240                 break;
00241             case LatticePatchworkSetUp:
00242                 errorMessage = "The patchwork for the Simulation has not been set up";
00243                 break;
00244             case CvodeObjectSetUp:
00245                 errorMessage = "The CVODE object has not been initialized";
00246                 break;
00247             case SimulationStarted:
00248                 errorMessage = "The Simulation has not been started";
00249                 break;
00250             default:
00251                 errorMessage = "Uppss, you've made a non-standard error, sadly I can't "
00252                               "help you there";
00253                 break;
00254         }
00255         errorKill(errorMessage);
00256     }
00257     return;
00258 }
```

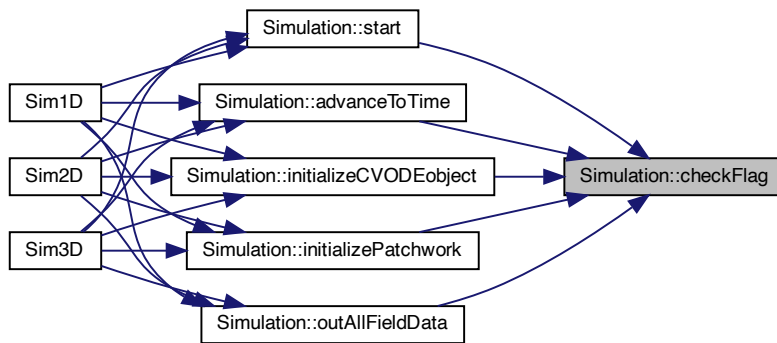
References [CvodeObjectSetUp](#), [errorKill\(\)](#), [LatticeDiscreteSetUp](#), [LatticePatchworkSetUp](#), [LatticePhysicalSetUp](#), [SimulationStarted](#), and [statusFlags](#).

Referenced by [advanceToTime\(\)](#), [initializeCVODEobject\(\)](#), [initializePatchwork\(\)](#), [outAllFieldData\(\)](#), and [start\(\)](#).

Here is the call graph for this function:



Here is the caller graph for this function:



5.16.3.6 checkNoFlag()

```
void Simulation::checkNoFlag (
    unsigned int flag ) const
```

function to check if flag has not been set

Check absence of configuration flags.

Definition at line 261 of file [SimulationClass.cpp](#).

```

00261 {
00262     if ((statusFlags & flag)) {
00263         std::string errorMessage;
00264         switch (flag) {
00265             case LatticeDiscreteSetUp:
00266                 errorMessage =
00267                     "The discrete size of the Simulation has already been set up";
00268                 break;
00269             case LatticePhysicalSetUp:
00270                 errorMessage =
00271                     "The physical size of the Simulation has already been set up";
00272                 break;
00273             case LatticePatchworkSetUp:
00274                 errorMessage = "The patchwork for the Simulation has already been set up";
00275                 break;
00276             case CvodeObjectSetUp:
00277                 errorMessage = "The CVODE object has already been initialized";
00278                 break;
00279             case SimulationStarted:
00280                 errorMessage = "The simulation has already started, some changes are no "
00281                     "longer possible";
00282                 break;
00283             default:
00284                 errorMessage = "Uppss, you've made a non-standard error, sadly I can't "
00285                     "help you there";
00286                 break;
00287         }
00288         errorKill(errorMessage);
00289     }
00290     return;
00291 }
```

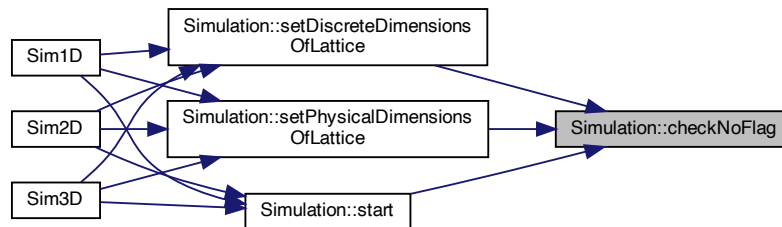
References [CvodeObjectSetUp](#), [errorKill\(\)](#), [LatticeDiscreteSetUp](#), [LatticePatchworkSetUp](#), [LatticePhysicalSetUp](#), [SimulationStarted](#), and [statusFlags](#).

Referenced by [setDiscreteDimensionsOfLattice\(\)](#), [setPhysicalDimensionsOfLattice\(\)](#), and [start\(\)](#).

Here is the call graph for this function:



Here is the caller graph for this function:



5.16.3.7 get_cart_comm()

```
MPI_Comm * Simulation::get_cart_comm ( ) [inline]
```

reference to the cartesian communicator of the lattice (for debugging)

Definition at line 56 of file [SimulationClass.h](#).

```
00056 { return &lattice.comm; }
```

References [Lattice::comm](#), and [lattice](#).

5.16.3.8 initializeCVODeobject()

```
void Simulation::initializeCVODeobject (
    const sunrealtype reltol,
    const sunrealtype abstol )
```

function to initialize the CVODe object with all requirements

Configure CVODe.

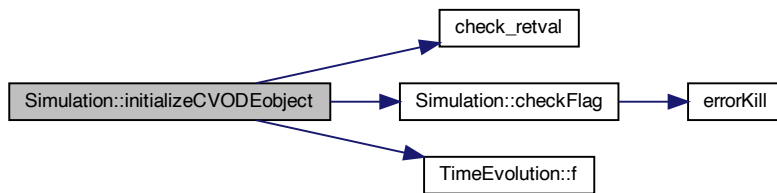
Definition at line 70 of file [SimulationClass.cpp](#).

```
00071     {
00072         checkFlag(SimulationStarted);
00073
00074         // CVODe settings return value
00075         int retval = 0;
00076
00077         // Create CVODe object -- returns a pointer to the cvoDe memory structure
00078         // with Adams method (Adams-Moulton formula) solver chosen for non-stiff ODE
00079         cvode_mem = CVODeCreate(CV_ADAMS, lattice.suncxtx);
00080
00081         // Specify user data and attach it to the main cvoDe memory block
00082         retval = CVODeSetUserData(
00083             cvode_mem,
00084             &latticePatch); // patch contains the user data as used in CVRhsFn
00085         if (check_retval(&retval, "CVODeSetUserData", 1, lattice.my_prc))
00086             MPI_Abort(lattice.comm, 1);
00087
00088         // Initialize CVODe solver
00089         retval = CVODeInit(cvode_mem, TimeEvolution::f, 0,
00090             latticePatch.u); // allocate memory, CVRhsFn f, t_i=0, u
00091                             // contains the initial values
00092         if (check_retval(&retval, "CVODeInit", 1, lattice.my_prc))
00093             MPI_Abort(lattice.comm, 1);
00094
00095         // Create fixed point nonlinear solver object (suitable for non-stiff ODE) and
00096         // attach it to CVODe
00097         NLS = SUNNonlinSol_FixedPoint(latticePatch.u, 0, lattice.suncxtx);
00098         retval = CVODeSetNonlinearSolver(cvode_mem, NLS);
00099         if (check_retval(&retval, "CVODeSetNonlinearSolver", 1, lattice.my_prc))
00100             MPI_Abort(lattice.comm, 1);
00101
00102         // Anderson damping factor
00103         retval = SUNNonlinSolSetDamping_FixedPoint(NLS, 1);
00104         if (check_retval(&retval, "SUNNonlinSolSetDamping_FixedPoint", 1,
00105             lattice.my_prc)) MPI_Abort(lattice.comm, 1);
00106
00107         // Specify integration tolerances -- a scalar relative tolerance and scalar
00108         // absolute tolerance
00109         retval = CVODeSStolerances(cvode_mem, reltol, abstol);
00110         if (check_retval(&retval, "CVODeSStolerances", 1, lattice.my_prc))
00111             MPI_Abort(lattice.comm, 1);
00112
00113         // Specify the maximum number of steps to be taken by the solver in its
00114         // attempt to reach the next tout
00115         retval = CVODeSetMaxNumSteps(cvode_mem, 10000);
00116         if (check_retval(&retval, "CVODeSetMaxNumSteps", 1, lattice.my_prc))
00117             MPI_Abort(lattice.comm, 1);
00118
00119         // maximum number of warnings for too small h
00120         retval = CVODeSetMaxHnilWarns(cvode_mem, 3);
00121         if (check_retval(&retval, "CVODeSetMaxHnilWarns", 1, lattice.my_prc))
00122             MPI_Abort(lattice.comm, 1);
00123
00124         statusFlags |= CVODeObjectSetUp;
00125     }
```

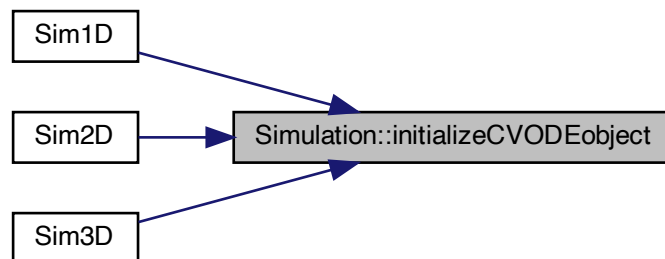
References [check_retval\(\)](#), [checkFlag\(\)](#), [Lattice::comm](#), [cvode_mem](#), [CVODeObjectSetUp](#), [TimeEvolution::f\(\)](#), [lattice](#), [latticePatch](#), [Lattice::my_prc](#), [NLS](#), [SimulationStarted](#), [statusFlags](#), [Lattice::suncxtx](#), and [LatticePatch::u](#).

Referenced by [Sim1D\(\)](#), [Sim2D\(\)](#), and [Sim3D\(\)](#).

Here is the call graph for this function:



Here is the caller graph for this function:



5.16.3.9 initializePatchwork()

```

void Simulation::initializePatchwork (
    const int nx,
    const int ny,
    const int nz )
  
```

function to initialize the Patchwork

Check that the lattice dimensions are set up and generate the patchwork.

Definition at line 57 of file [SimulationClass.cpp](#).

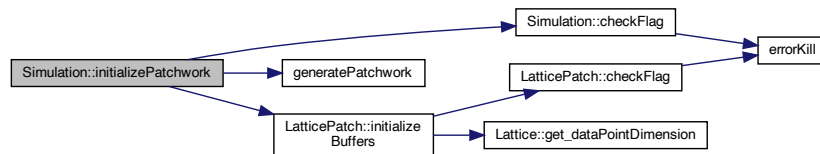
```

00058         {
00059     checkFlag(LatticeDiscreteSetUp);
00060     checkFlag(LatticePhysicalSetUp);
00061
00062     // Generate the patchwork
00063     generatePatchwork(lattice, latticePatch, nx, ny, nz);
00064     latticePatch.initializeBuffers();
00065
00066     statusFlags |= LatticePatchworkSetUp;
00067 }
  
```

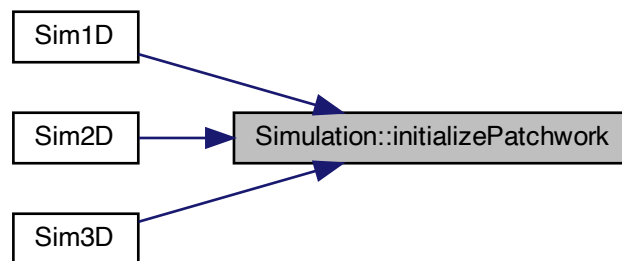
References [checkFlag\(\)](#), [generatePatchwork\(\)](#), [LatticePatch::initializeBuffers\(\)](#), [lattice](#), [LatticeDiscreteSetUp](#), [latticePatch](#), [LatticePatchworkSetUp](#), [LatticePhysicalSetUp](#), and [statusFlags](#).

Referenced by [Sim1D\(\)](#), [Sim2D\(\)](#), and [Sim3D\(\)](#).

Here is the call graph for this function:



Here is the caller graph for this function:



5.16.3.10 outAllFieldData()

```
void Simulation::outAllFieldData (
    const int & state )
```

function to write field data to disk

Write specified simulation steps to disk.

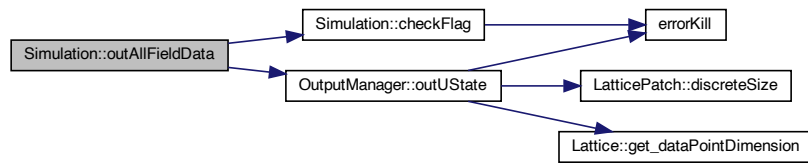
Definition at line 225 of file [SimulationClass.cpp](#).

```
00225 {
00226     checkFlag(SimulationStarted);
00227     outputManager.outUState(state, lattice, latticePatch);
00228 }
```

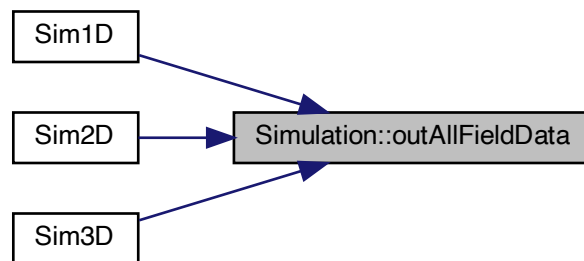
References [checkFlag\(\)](#), [lattice](#), [latticePatch](#), [outputManager](#), [OutputManager::outUState\(\)](#), and [SimulationStarted](#).

Referenced by [Sim1D\(\)](#), [Sim2D\(\)](#), and [Sim3D\(\)](#).

Here is the call graph for this function:



Here is the caller graph for this function:



5.16.3.11 setDiscreteDimensionsOfLattice()

```

void Simulation::setDiscreteDimensionsOfLattice (
    const sunindextype _tot_nx,
    const sunindextype _tot_ny,
    const sunindextype _tot_nz )
  
```

function to set discrete dimensions of the lattice

Set the discrete dimensions, the number of points per dimension.

Definition at line 41 of file [SimulationClass.cpp](#).

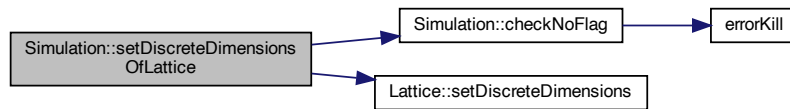
```

00042                                     {
00043     checkNoFlag(LatticePatchworkSetUp);
00044     lattice.setDiscreteDimensions(nx, ny, nz);
00045     statusFlags |= LatticeDiscreteSetUp;
00046 }
  
```

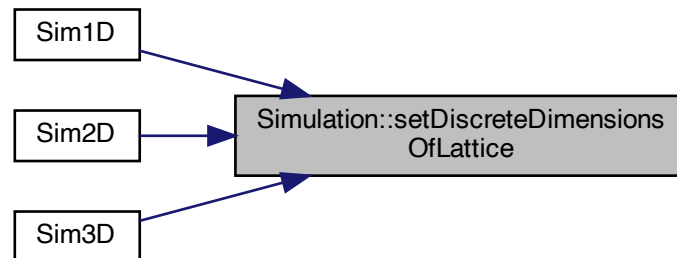
References [checkNoFlag\(\)](#), [lattice](#), [LatticeDiscreteSetUp](#), [LatticePatchworkSetUp](#), [Lattice::setDiscreteDimensions\(\)](#), and [statusFlags](#).

Referenced by [Sim1D\(\)](#), [Sim2D\(\)](#), and [Sim3D\(\)](#).

Here is the call graph for this function:



Here is the caller graph for this function:



5.16.3.12 setInitialConditions()

```
void Simulation::setInitialConditions ( )
```

functions to set the initial field configuration onto the lattice

Set initial conditions: Fill the lattice points with the initial field values

Definition at line 140 of file [SimulationClass.cpp](#).

```

00140     {
00141         const sunrealtype dx = latticePatch.getDelta(1);
00142         const sunrealtype dy = latticePatch.getDelta(2);
00143         const sunrealtype dz = latticePatch.getDelta(3);
00144         const sunindextype nx = latticePatch.discreteSize(1);
00145         const sunindextype ny = latticePatch.discreteSize(2);
00146         const sunindextype totalNP = latticePatch.discreteSize();
00147         const sunrealtype x0 = latticePatch.origin(1);
00148         const sunrealtype y0 = latticePatch.origin(2);
00149         const sunrealtype z0 = latticePatch.origin(3);
00150         sunindextype px = 0, py = 0, pz = 0;
00151         #pragma omp parallel for default(none) \
00152         shared(nx, ny, totalNP, dx, dy, dz, x0, y0, z0) \
00153         firstprivate(px, py, pz) schedule(static)
00154         for (sunindextype i = 0; i < totalNP * 6; i += 6) {
00155             px = (i / 6) % nx;
00156             py = ((i / 6) / nx) % ny;
00157             pz = ((i / 6) / nx) / ny;
00158             // Call the 'eval' function to fill the lattice points with the field data
00159             icsettings.eval(static_cast<sunrealtype>(px) * dx + x0,

```

```

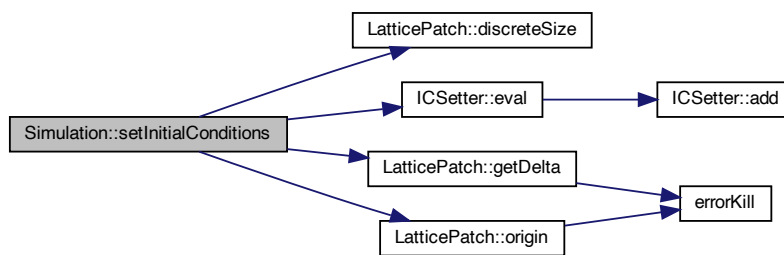
00160         static_cast<sunrealtype>(py) * dy + y0,
00161         static_cast<sunrealtype>(pz) * dz + z0, &latticePatch.uData[i]);
00162     }
00163     return;
00164 }

```

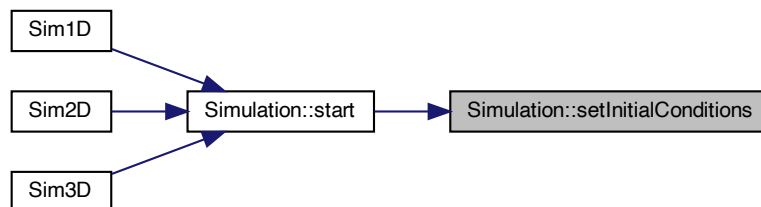
References [LatticePatch::discreteSize\(\)](#), [ICSetter::eval\(\)](#), [LatticePatch::getDelta\(\)](#), [icsettings](#), [latticePatch](#), [LatticePatch::origin\(\)](#), and [LatticePatch::uData](#).

Referenced by [start\(\)](#).

Here is the call graph for this function:



Here is the caller graph for this function:



5.16.3.13 setPhysicalDimensionsOfLattice()

```

void Simulation::setPhysicalDimensionsOfLattice (
    const sunrealtype lx,
    const sunrealtype ly,
    const sunrealtype lz )

```

function to set physical dimensions of the lattice

Set the physical dimensions with lengths in micro meters.

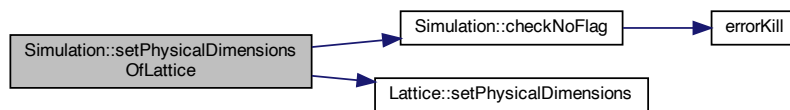
Definition at line 49 of file [SimulationClass.cpp](#).

```
00050 {
00051     checkNoFlag(LatticePatchworkSetUp);
00052     lattice.setPhysicalDimensions(lx, ly, lz);
00053     statusFlags |= LatticePhysicalSetUp;
00054 }
```

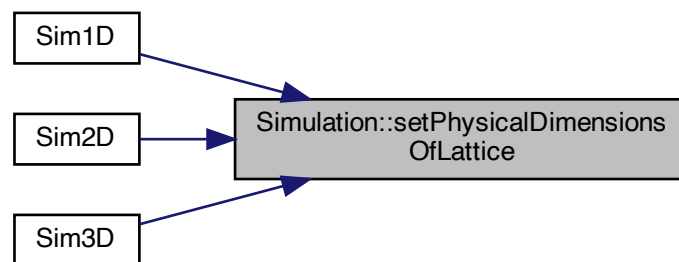
References [checkNoFlag\(\)](#), [lattice](#), [LatticePatchworkSetUp](#), [LatticePhysicalSetUp](#), [Lattice::setPhysicalDimensions\(\)](#), and [statusFlags](#).

Referenced by [Sim1D\(\)](#), [Sim2D\(\)](#), and [Sim3D\(\)](#).

Here is the call graph for this function:



Here is the caller graph for this function:



5.16.3.14 start()

```
void Simulation::start ( )
```

function to start the simulation for time iteration

Check if the lattice patchwork is set up and set the initial conditions.

Definition at line 128 of file [SimulationClass.cpp](#).

```
00128 {
00129     checkFlag(LatticeDiscreteSetUp);
00130     checkFlag(LatticePhysicalSetUp);
00131     checkFlag(LatticePatchworkSetUp);
00132     checkNoFlag(SimulationStarted);
00133 }
```

```

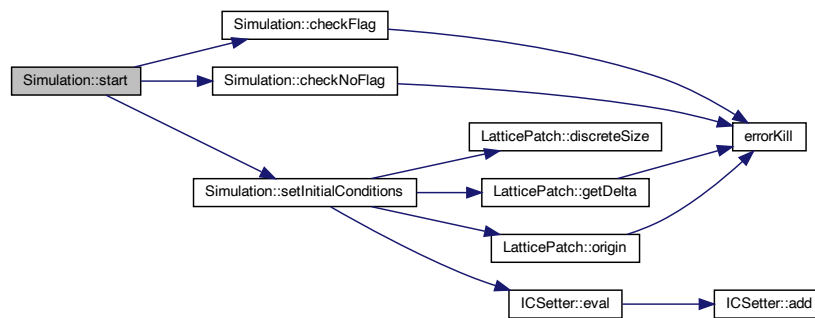
00133     checkNoFlag(CvodeObjectSetUp);
00134     setInitialConditions();
00135     statusFlags |= SimulationStarted;
00136 }

```

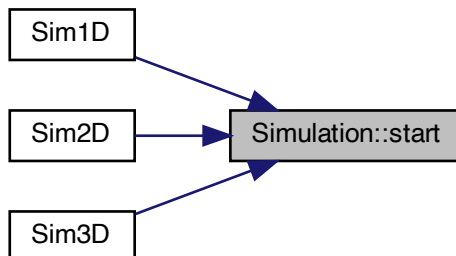
References [checkFlag\(\)](#), [checkNoFlag\(\)](#), [CvodeObjectSetUp](#), [LatticeDiscreteSetUp](#), [LatticePatchworkSetUp](#), [LatticePhysicalSetUp](#), [setInitialConditions\(\)](#), [SimulationStarted](#), and [statusFlags](#).

Referenced by [Sim1D\(\)](#), [Sim2D\(\)](#), and [Sim3D\(\)](#).

Here is the call graph for this function:



Here is the caller graph for this function:



5.16.4 Field Documentation

5.16.4.1 cvode_mem

```
void* Simulation::cvode_mem
```

pointer to CVode memory object

Definition at line 47 of file [SimulationClass.h](#).

Referenced by [advanceToTime\(\)](#), [initializeCVODEobject\(\)](#), and [~Simulation\(\)](#).

5.16.4.2 icsettings

```
ICSetter Simulation::icsettings
```

IC Setter object.

Definition at line 43 of file [SimulationClass.h](#).

Referenced by [addInitialConditions\(\)](#), [setInitialConditions\(\)](#), [Sim1D\(\)](#), [Sim2D\(\)](#), and [Sim3D\(\)](#).

5.16.4.3 lattice

```
Lattice Simulation::lattice [private]
```

[Lattice](#) object.

Definition at line 33 of file [SimulationClass.h](#).

Referenced by [addInitialConditions\(\)](#), [advanceToTime\(\)](#), [get_cart_comm\(\)](#), [initializeCVODEobject\(\)](#), [initializePatchwork\(\)](#), [outAllFieldData\(\)](#), [setDiscreteDimensionsOfLattice\(\)](#), [setPhysicalDimensionsOfLattice\(\)](#), [Simulation\(\)](#), and [~Simulation\(\)](#).

5.16.4.4 latticePatch

```
LatticePatch Simulation::latticePatch [private]
```

[LatticePatch](#) object.

Definition at line 35 of file [SimulationClass.h](#).

Referenced by [addInitialConditions\(\)](#), [advanceToTime\(\)](#), [initializeCVODEobject\(\)](#), [initializePatchwork\(\)](#), [outAllFieldData\(\)](#), and [setInitialConditions\(\)](#).

5.16.4.5 NLS

```
SUNNonlinearSolver Simulation::NLS
```

nonlinear solver object

Definition at line 49 of file [SimulationClass.h](#).

Referenced by [initializeCVODEobject\(\)](#), and [~Simulation\(\)](#).

5.16.4.6 outputManager

`OutputManager` `Simulation::outputManager`

Output Manager object.

Definition at line 45 of file [SimulationClass.h](#).

Referenced by [outAllFieldData\(\)](#), [Sim1D\(\)](#), [Sim2D\(\)](#), and [Sim3D\(\)](#).

5.16.4.7 statusFlags

`unsigned int` `Simulation::statusFlags` [private]

simulation status flags

Definition at line 39 of file [SimulationClass.h](#).

Referenced by [checkFlag\(\)](#), [checkNoFlag\(\)](#), [initializeCVODEobject\(\)](#), [initializePatchwork\(\)](#), [setDiscreteDimensionsOfLattice\(\)](#), [setPhysicalDimensionsOfLattice\(\)](#), [Simulation\(\)](#), [start\(\)](#), and [~Simulation\(\)](#).

5.16.4.8 t

`sunrealtype` `Simulation::t` [private]

current time of the simulation

Definition at line 37 of file [SimulationClass.h](#).

Referenced by [advanceToTime\(\)](#), and [Simulation\(\)](#).

The documentation for this class was generated from the following files:

- [src/SimulationClass.h](#)
- [src/SimulationClass.cpp](#)

5.17 TimeEvolution Class Reference

monostate [TimeEvolution](#) class to propagate the field data in time in a given order of the HE weak-field expansion

```
#include <src/TimeEvolutionFunctions.h>
```

Static Public Member Functions

- static int [f](#)(`sunrealtype` t, `N_Vector` u, `N_Vector` udot, void *data_loc)
CVODE right hand side function (CVRhsFn) to provide IVP of the ODE.

Static Public Attributes

- static int * [c](#) = nullptr
choice which processes of the weak field expansion are included
- static void(* [TimeEvolver](#))(LatticePatch *, N_Vector, N_Vector, int *) = [nonlinear1DProp](#)
Pointer to functions for differentiation and time evolution.

5.17.1 Detailed Description

monostate [TimeEvolution](#) class to propagate the field data in time in a given order of the HE weak-field expansion

Definition at line 15 of file [TimeEvolutionFunctions.h](#).

5.17.2 Member Function Documentation

5.17.2.1 f()

```
int TimeEvolution::f (
    sunrealtype t,
    N_Vector u,
    N_Vector udot,
    void * data_loc ) [static]
```

CVODE right hand side function (CVRhsFn) to provide IVP of the ODE.

Cvode right-hand-side function (CVRhsFn)

Definition at line 11 of file [TimeEvolutionFunctions.cpp](#).

```
00011 {
00012
00013 // Set recover pointer to provided lattice patch where the field data resides
00014 LatticePatch *data = static_cast<LatticePatch *>(data_loc);
00015
00016 // update circle
00017 // Access provided field values and temp. derivatives with NVector pointers
00018 sunrealtype *udata = NV_DATA_P(u),
00019             *dudata = NV_DATA_P(udot);
00020
00021 // Store original data location of the patch
00022 sunrealtype *originaluData = data->uData,
00023             *originalduData = data->duData;
00024
00025 // Point patch data to arguments of f
00026 data->uData = udata;
00027 data->duData = dudata;
00028
00029 // Time-evolve these arguments (the field data) with specific propagator below
00030 TimeEvolver(data, u, udot, c);
00031
00032 // Refer patch data back to original location
00033 data->uData = originaluData;
00034 data->duData = originalduData;
00035
00036 return (0);
00037 }
```

References [c](#), [LatticePatch::duData](#), [TimeEvolver](#), and [LatticePatch::uData](#).

Referenced by [Simulation::initializeCVODEobject\(\)](#).

Here is the caller graph for this function:



5.17.3 Field Documentation

5.17.3.1 c

```
int * TimeEvolution::c = nullptr [static]
```

choice which processes of the weak field expansion are included

Definition at line 18 of file [TimeEvolutionFunctions.h](#).

Referenced by [f\(\)](#), [Sim1D\(\)](#), [Sim2D\(\)](#), and [Sim3D\(\)](#).

5.17.3.2 TimeEvolver

```
void(* TimeEvolution::TimeEvolver)(LatticePatch *, N_Vector, N_Vector, int *) = nonlinear1DProp [static]
```

Pointer to functions for differentiation and time evolution.

Definition at line 21 of file [TimeEvolutionFunctions.h](#).

Referenced by [f\(\)](#), [Sim1D\(\)](#), [Sim2D\(\)](#), and [Sim3D\(\)](#).

The documentation for this class was generated from the following files:

- [src/TimeEvolutionFunctions.h](#)
- [src/SimulationFunctions.cpp](#)
- [src/TimeEvolutionFunctions.cpp](#)

Chapter 6

File Documentation

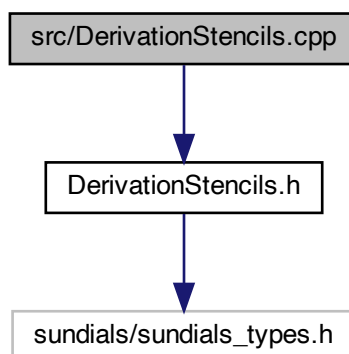
6.1 README.md File Reference

6.2 src/DerivationStencils.cpp File Reference

Empty. All definitions in the header.

```
#include "DerivationStencils.h"
```

Include dependency graph for DerivationStencils.cpp:



6.2.1 Detailed Description

Empty. All definitions in the header.

Definition in file [DerivationStencils.cpp](#).

6.3 DerivationStencils.cpp

[Go to the documentation of this file.](#)

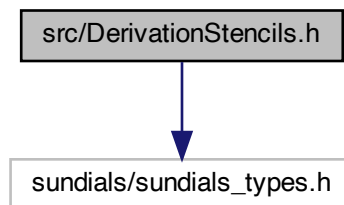
```
00001 ///////////////////////////////////////////////////////////////////
00002 /// @file DerivationStencils.cpp
00003 /// @brief Empty. All definitions in the header.
00004 ///////////////////////////////////////////////////////////////////
00005 #include "DerivationStencils.h"
```

6.4 src/DerivationStencils.h File Reference

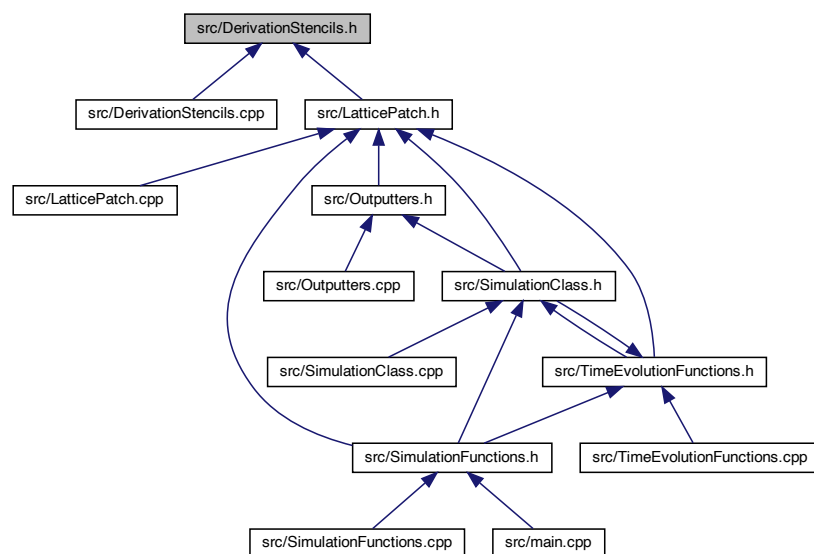
Definition of derivation stencils from order 1 to 13.

```
#include <sundials/sundials_types.h>
```

Include dependency graph for DerivationStencils.h:



This graph shows which files directly or indirectly include this file:



Functions

- sunrealtype [s1f](#) (sunrealtype const *udata, const int dPD)
- sunrealtype [s1b](#) (sunrealtype const *udata, const int dPD)
- sunrealtype [s2f](#) (sunrealtype const *udata, const int dPD)
- sunrealtype [s2c](#) (sunrealtype const *udata, const int dPD)
- sunrealtype [s2b](#) (sunrealtype const *udata, const int dPD)
- sunrealtype [s3f](#) (sunrealtype const *udata, const int dPD)
- sunrealtype [s3b](#) (sunrealtype const *udata, const int dPD)
- sunrealtype [s4f](#) (sunrealtype const *udata, const int dPD)
- sunrealtype [s4c](#) (sunrealtype const *udata, const int dPD)
- sunrealtype [s4b](#) (sunrealtype const *udata, const int dPD)
- sunrealtype [s5f](#) (sunrealtype const *udata, const int dPD)
- sunrealtype [s5b](#) (sunrealtype const *udata, const int dPD)
- sunrealtype [s6f](#) (sunrealtype const *udata, const int dPD)
- sunrealtype [s6c](#) (sunrealtype const *udata, const int dPD)
- sunrealtype [s6b](#) (sunrealtype const *udata, const int dPD)
- sunrealtype [s7f](#) (sunrealtype const *udata, const int dPD)
- sunrealtype [s7b](#) (sunrealtype const *udata, const int dPD)
- sunrealtype [s8f](#) (sunrealtype const *udata, const int dPD)
- sunrealtype [s8c](#) (sunrealtype const *udata, const int dPD)
- sunrealtype [s8b](#) (sunrealtype const *udata, const int dPD)
- sunrealtype [s9f](#) (sunrealtype const *udata, const int dPD)
- sunrealtype [s9b](#) (sunrealtype const *udata, const int dPD)
- sunrealtype [s10f](#) (sunrealtype const *udata, const int dPD)
- sunrealtype [s10c](#) (sunrealtype const *udata, const int dPD)
- sunrealtype [s10b](#) (sunrealtype const *udata, const int dPD)
- sunrealtype [s11f](#) (sunrealtype const *udata, const int dPD)
- sunrealtype [s11b](#) (sunrealtype const *udata, const int dPD)
- sunrealtype [s12f](#) (sunrealtype const *udata, const int dPD)
- sunrealtype [s12c](#) (sunrealtype const *udata, const int dPD)
- sunrealtype [s12b](#) (sunrealtype const *udata, const int dPD)
- sunrealtype [s13f](#) (sunrealtype const *udata, const int dPD)
- sunrealtype [s13b](#) (sunrealtype const *udata, const int dPD)
- sunrealtype [s1f](#) (sunrealtype const *udata)
- sunrealtype [s1b](#) (sunrealtype const *udata)
- sunrealtype [s2f](#) (sunrealtype const *udata)
- sunrealtype [s2c](#) (sunrealtype const *udata)
- sunrealtype [s2b](#) (sunrealtype const *udata)
- sunrealtype [s3f](#) (sunrealtype const *udata)
- sunrealtype [s3b](#) (sunrealtype const *udata)
- sunrealtype [s4f](#) (sunrealtype const *udata)
- sunrealtype [s4c](#) (sunrealtype const *udata)
- sunrealtype [s4b](#) (sunrealtype const *udata)
- sunrealtype [s5f](#) (sunrealtype const *udata)
- sunrealtype [s5b](#) (sunrealtype const *udata)
- sunrealtype [s6f](#) (sunrealtype const *udata)
- sunrealtype [s6c](#) (sunrealtype const *udata)
- sunrealtype [s6b](#) (sunrealtype const *udata)
- sunrealtype [s7f](#) (sunrealtype const *udata)
- sunrealtype [s7b](#) (sunrealtype const *udata)
- sunrealtype [s8f](#) (sunrealtype const *udata)
- sunrealtype [s8c](#) (sunrealtype const *udata)
- sunrealtype [s8b](#) (sunrealtype const *udata)
- sunrealtype [s9f](#) (sunrealtype const *udata)

- sunrealtype [s9b](#) (sunrealtype const *udata)
- sunrealtype [s10f](#) (sunrealtype const *udata)
- sunrealtype [s10c](#) (sunrealtype const *udata)
- sunrealtype [s10b](#) (sunrealtype const *udata)
- sunrealtype [s11f](#) (sunrealtype const *udata)
- sunrealtype [s11b](#) (sunrealtype const *udata)
- sunrealtype [s12f](#) (sunrealtype const *udata)
- sunrealtype [s12c](#) (sunrealtype const *udata)
- sunrealtype [s12b](#) (sunrealtype const *udata)
- sunrealtype [s13f](#) (sunrealtype const *udata)
- sunrealtype [s13b](#) (sunrealtype const *udata)

6.4.1 Detailed Description

Definition of derivation stencils from order 1 to 13.

Definition in file [DerivationStencils.h](#).

6.4.2 Function Documentation

6.4.2.1 [s10b\(\)](#) [1/2]

```
sunrealtype s10b (  
    sunrealtype const * udata ) [inline]
```

Definition at line [275](#) of file [DerivationStencils.h](#).

```
00276 { return s10b(udata, 6); }
```

References [s10b\(\)](#).

Here is the call graph for this function:



6.4.2.2 s10b() [2/2]

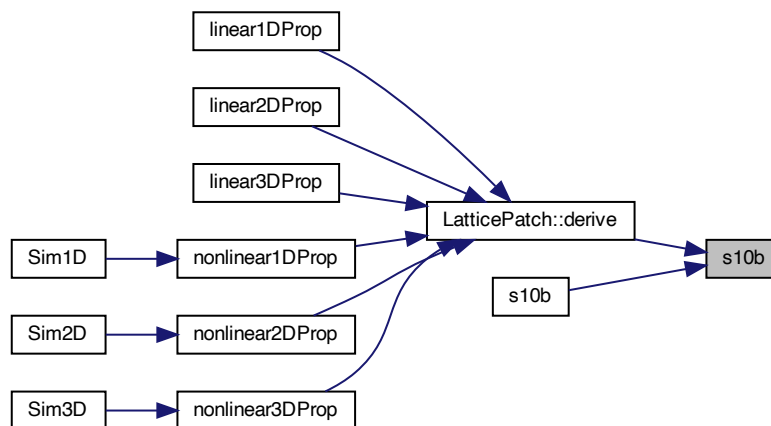
```
sunrealtype s10b (
    sunrealtype const * udata,
    const int dPD ) [inline]
```

Definition at line 169 of file [DerivationStencils.h](#).

```
00169 {
00170     return 1.0 / 840.0 * udata[-4 * dPD] - 1.0 / 63.0 * udata[-3 * dPD] +
00171           3.0 / 28.0 * udata[-2 * dPD] - 4.0 / 7.0 * udata[-1 * dPD] -
00172           11.0 / 30.0 * udata[0] + 6.0 / 5.0 * udata[1 * dPD] -
00173           1.0 / 2.0 * udata[2 * dPD] + 4.0 / 21.0 * udata[3 * dPD] -
00174           3.0 / 56.0 * udata[4 * dPD] + 1.0 / 105.0 * udata[5 * dPD] -
00175           1.0 / 1260.0 * udata[6 * dPD];
00176 }
```

Referenced by [LatticePatch::derive\(\)](#), and [s10b\(\)](#).

Here is the caller graph for this function:



6.4.2.3 s10c() [1/2]

```
sunrealtype s10c (
    sunrealtype const * udata ) [inline]
```

Definition at line 273 of file [DerivationStencils.h](#).

```
00274 { return s10c(udata, 6); }
```

References [s10c\(\)](#).

Here is the call graph for this function:



6.4.2.4 s10c() [2/2]

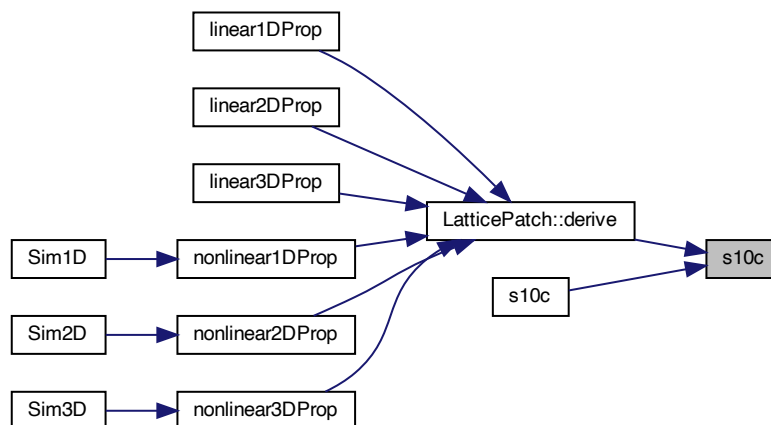
```
sunrealtype s10c (
    sunrealtype const * udata,
    const int dPD ) [inline]
```

Definition at line 161 of file [DerivationStencils.h](#).

```
00161 {
00162     return -1.0 / 1260.0 * udata[-5 * dPD] + 5.0 / 504.0 * udata[-4 * dPD] -
00163           5.0 / 84.0 * udata[-3 * dPD] + 5.0 / 21.0 * udata[-2 * dPD] -
00164           5.0 / 6.0 * udata[-1 * dPD] + 0 + 5.0 / 6.0 * udata[1 * dPD] -
00165           5.0 / 21.0 * udata[2 * dPD] + 5.0 / 84.0 * udata[3 * dPD] -
00166           5.0 / 504.0 * udata[4 * dPD] + 1.0 / 1260.0 * udata[5 * dPD];
00167 }
```

Referenced by [LatticePatch::derive\(\)](#), and [s10c\(\)](#).

Here is the caller graph for this function:



6.4.2.5 s10f() [1/2]

```
sunrealtype s10f (
    sunrealtype const * udata ) [inline]
```

Definition at line 271 of file [DerivationStencils.h](#).

```
00272 { return s10f(udata, 6); }
```

References [s10f\(\)](#).

Here is the call graph for this function:



6.4.2.6 s10f() [2/2]

```

sunrealtype s10f (
    sunrealtype const * udata,
    const int dPD ) [inline]
  
```

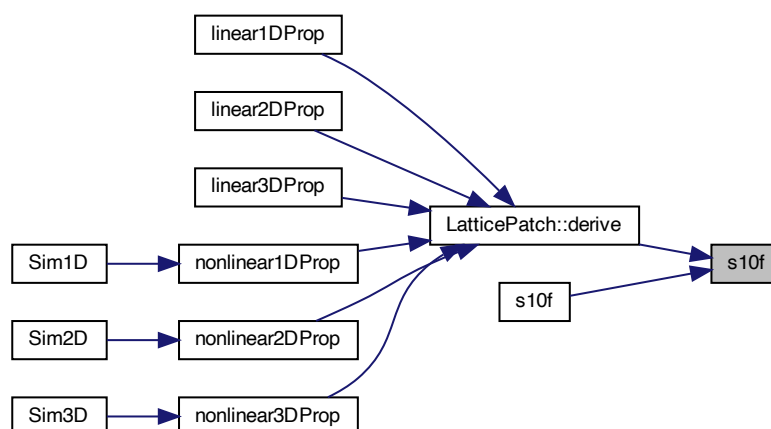
Definition at line 152 of file [DerivationStencils.h](#).

```

00152 {
00153     return 1.0 / 1260.0 * udata[-6 * dPD] - 1.0 / 105.0 * udata[-5 * dPD] +
00154           3.0 / 56.0 * udata[-4 * dPD] - 4.0 / 21.0 * udata[-3 * dPD] +
00155           1.0 / 2.0 * udata[-2 * dPD] - 6.0 / 5.0 * udata[-1 * dPD] +
00156           11.0 / 30.0 * udata[0] + 4.0 / 7.0 * udata[1 * dPD] -
00157           3.0 / 28.0 * udata[2 * dPD] + 1.0 / 63.0 * udata[3 * dPD] -
00158           1.0 / 840.0 * udata[4 * dPD];
00159 }
  
```

Referenced by [LatticePatch::derive\(\)](#), and [s10f\(\)](#).

Here is the caller graph for this function:



6.4.2.7 s11b() [1/2]

```
sunrealtype s11b (
    sunrealtype const * udata ) [inline]
```

Definition at line 279 of file [DerivationStencils.h](#).

```
00280 { return s11b(udata, 6); }
```

References [s11b\(\)](#).

Here is the call graph for this function:



6.4.2.8 s11b() [2/2]

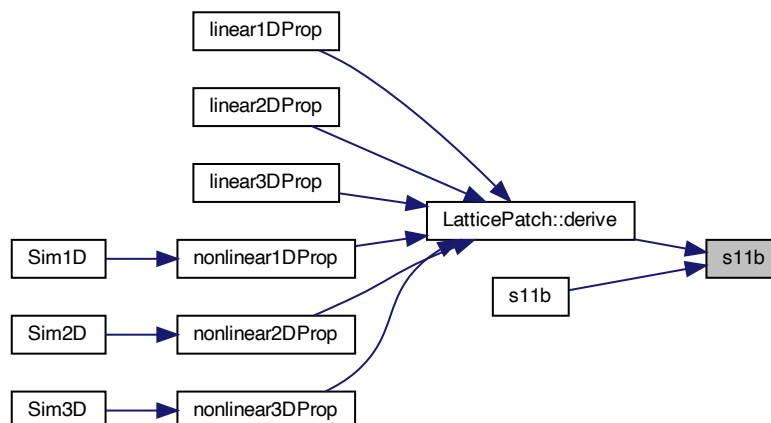
```
sunrealtype s11b (
    sunrealtype const * udata,
    const int dPD ) [inline]
```

Definition at line 187 of file [DerivationStencils.h](#).

```
00187 {
00188     return -1.0 / 2310.0 * udata[-5 * dPD] + 1.0 / 168.0 * udata[-4 * dPD] -
00189            5.0 / 126.0 * udata[-3 * dPD] + 5.0 / 28.0 * udata[-2 * dPD] -
00190            5.0 / 7.0 * udata[-1 * dPD] - 1.0 / 6.0 * udata[0] + udata[1 * dPD] -
00191            5.0 / 14.0 * udata[2 * dPD] + 5.0 / 42.0 * udata[3 * dPD] -
00192            5.0 / 168.0 * udata[4 * dPD] + 1.0 / 210.0 * udata[5 * dPD] -
00193            1.0 / 2772.0 * udata[6 * dPD];
00194 }
```

Referenced by [LatticePatch::derive\(\)](#), and [s11b\(\)](#).

Here is the caller graph for this function:



6.4.2.9 s11f() [1/2]

```
sunrealtype s11f (
    sunrealtype const * udata ) [inline]
```

Definition at line 277 of file [DerivationStencils.h](#).

```
00278 { return s11f(udata, 6); }
```

References [s11f\(\)](#).

Here is the call graph for this function:

**6.4.2.10 s11f()** [2/2]

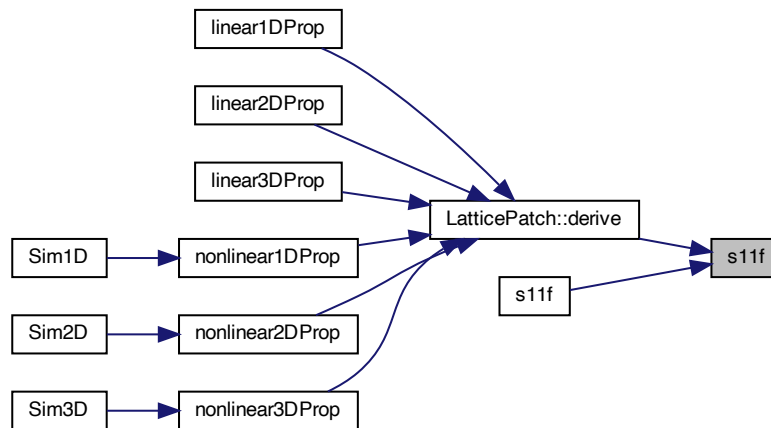
```
sunrealtype s11f (
    sunrealtype const * udata,
    const int dPD ) [inline]
```

Definition at line 178 of file [DerivationStencils.h](#).

```
00178 {
00179     return 1.0 / 2772.0 * udata[-6 * dPD] - 1.0 / 210.0 * udata[-5 * dPD] +
00180            5.0 / 168.0 * udata[-4 * dPD] - 5.0 / 42.0 * udata[-3 * dPD] +
00181            5.0 / 14.0 * udata[-2 * dPD] - 1.0 / 1.0 * udata[-1 * dPD] +
00182            1.0 / 6.0 * udata[0] + 5.0 / 7.0 * udata[1 * dPD] -
00183            5.0 / 28.0 * udata[2 * dPD] + 5.0 / 126.0 * udata[3 * dPD] -
00184            1.0 / 168.0 * udata[4 * dPD] + 1.0 / 2310.0 * udata[5 * dPD];
00185 }
```

Referenced by [LatticePatch::derive\(\)](#), and [s11f\(\)](#).

Here is the caller graph for this function:



6.4.2.11 `s12b()` [1/2]

```

sunrealtype s12b (
    sunrealtype const * udata ) [inline]

```

Definition at line 285 of file [DerivationStencils.h](#).

```

00286 { return s12b(udata, 6); }

```

References [s12b\(\)](#).

Here is the call graph for this function:



6.4.2.12 s12b() [2/2]

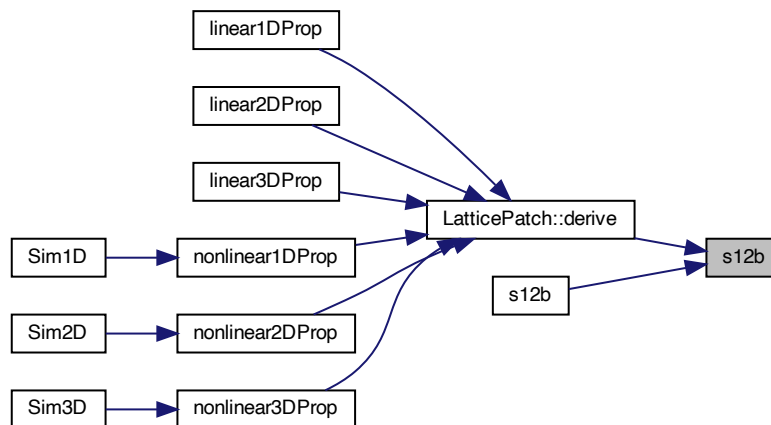
```
sunrealtype s12b (
    sunrealtype const * udata,
    const int dPD ) [inline]
```

Definition at line 215 of file [DerivationStencils.h](#).

```
00215 {
00216     return -1.0 / 3960.0 * udata[-5 * dPD] + 1.0 / 264.0 * udata[-4 * dPD] -
00217            1.0 / 36.0 * udata[-3 * dPD] + 5.0 / 36.0 * udata[-2 * dPD] -
00218            5.0 / 8.0 * udata[-1 * dPD] - 13.0 / 42.0 * udata[0] +
00219            7.0 / 6.0 * udata[1 * dPD] - 1.0 / 2.0 * udata[2 * dPD] +
00220            5.0 / 24.0 * udata[3 * dPD] - 5.0 / 72.0 * udata[4 * dPD] +
00221            1.0 / 60.0 * udata[5 * dPD] - 1.0 / 396.0 * udata[6 * dPD] +
00222            1.0 / 5544.0 * udata[7 * dPD];
00223 }
```

Referenced by [LatticePatch::derive\(\)](#), and [s12b\(\)](#).

Here is the caller graph for this function:

**6.4.2.13 s12c()** [1/2]

```
sunrealtype s12c (
    sunrealtype const * udata ) [inline]
```

Definition at line 283 of file [DerivationStencils.h](#).

```
00284 { return s12c(udata, 6); }
```

References [s12c\(\)](#).

Here is the call graph for this function:



6.4.2.14 s12c() [2/2]

```

sunrealtype s12c (
    sunrealtype const * udata,
    const int dPD ) [inline]
  
```

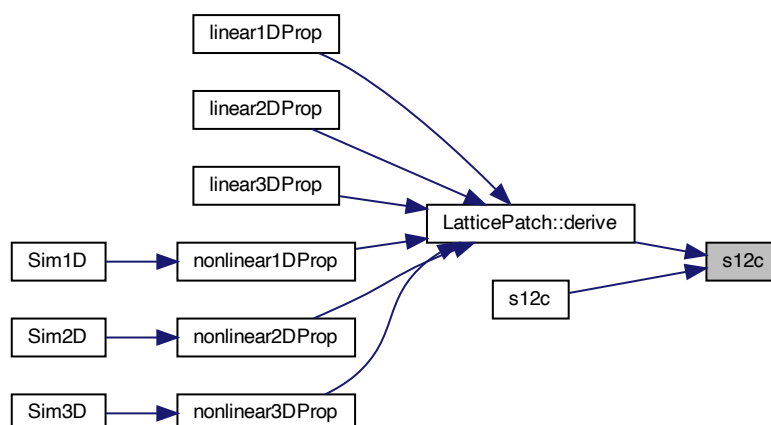
Definition at line 206 of file [DerivationStencils.h](#).

```

00206 {
00207     return 1.0 / 5544.0 * udata[-6 * dPD] - 1.0 / 385.0 * udata[-5 * dPD] +
00208           1.0 / 56.0 * udata[-4 * dPD] - 5.0 / 63.0 * udata[-3 * dPD] +
00209           15.0 / 56.0 * udata[-2 * dPD] - 6.0 / 7.0 * udata[-1 * dPD] + 0 +
00210           6.0 / 7.0 * udata[1 * dPD] - 15.0 / 56.0 * udata[2 * dPD] +
00211           5.0 / 63.0 * udata[3 * dPD] - 1.0 / 56.0 * udata[4 * dPD] +
00212           1.0 / 385.0 * udata[5 * dPD] - 1.0 / 5544.0 * udata[6 * dPD];
00213 }
  
```

Referenced by [LatticePatch::derive\(\)](#), and [s12c\(\)](#).

Here is the caller graph for this function:



6.4.2.15 s12f() [1/2]

```
sunrealtype s12f (
    sunrealtype const * udata ) [inline]
```

Definition at line 281 of file [DerivationStencils.h](#).

```
00282 { return s12f(udata, 6); }
```

References [s12f\(\)](#).

Here is the call graph for this function:

**6.4.2.16 s12f()** [2/2]

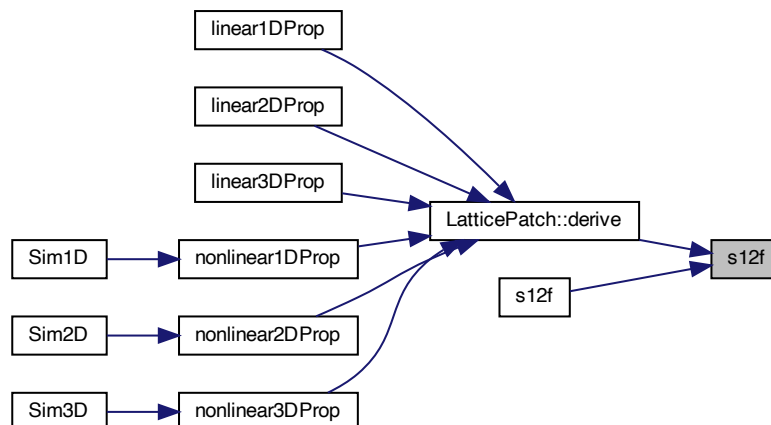
```
sunrealtype s12f (
    sunrealtype const * udata,
    const int dPD ) [inline]
```

Definition at line 196 of file [DerivationStencils.h](#).

```
00196 {
00197     return -1.0 / 5544.0 * udata[-7 * dPD] + 1.0 / 396.0 * udata[-6 * dPD] -
00198            1.0 / 60.0 * udata[-5 * dPD] + 5.0 / 72.0 * udata[-4 * dPD] -
00199            5.0 / 24.0 * udata[-3 * dPD] + 1.0 / 2.0 * udata[-2 * dPD] -
00200            7.0 / 6.0 * udata[-1 * dPD] + 13.0 / 42.0 * udata[0] +
00201            5.0 / 8.0 * udata[1 * dPD] - 5.0 / 36.0 * udata[2 * dPD] +
00202            1.0 / 36.0 * udata[3 * dPD] - 1.0 / 264.0 * udata[4 * dPD] +
00203            1.0 / 3960.0 * udata[5 * dPD];
00204 }
```

Referenced by [LatticePatch::derive\(\)](#), and [s12f\(\)](#).

Here is the caller graph for this function:



6.4.2.17 s13b() [1/2]

```
sunrealtype s13b (
    sunrealtype const * udata ) [inline]
```

Definition at line 289 of file [DerivationStencils.h](#).

```
00290 { return s13b(udata, 6); }
```

References [s13b\(\)](#).

Here is the call graph for this function:



6.4.2.18 s13b() [2/2]

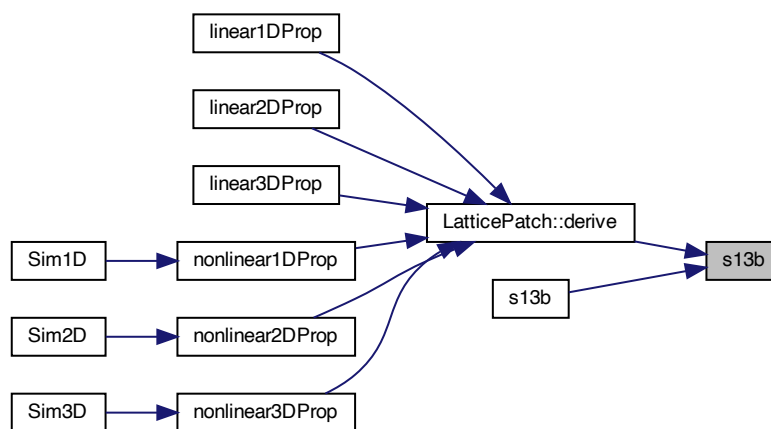
```
sunrealtype s13b (
    sunrealtype const * udata,
    const int dPD ) [inline]
```

Definition at line 235 of file [DerivationStencils.h](#).

```
00235 {
00236     return 1.0 / 10296.0 * udata[-6 * dPD] - 1.0 / 660.0 * udata[-5 * dPD] +
00237           1.0 / 88.0 * udata[-4 * dPD] - 1.0 / 18.0 * udata[-3 * dPD] +
00238           5.0 / 24.0 * udata[-2 * dPD] - 3.0 / 4.0 * udata[-1 * dPD] -
00239           1.0 / 7.0 * udata[0] + udata[1 * dPD] - 3.0 / 8.0 * udata[2 * dPD] +
00240           5.0 / 36.0 * udata[3 * dPD] - 1.0 / 24.0 * udata[4 * dPD] +
00241           1.0 / 110.0 * udata[5 * dPD] - 1.0 / 792.0 * udata[6 * dPD] +
00242           1.0 / 12012.0 * udata[7 * dPD];
00243 }
```

Referenced by [LatticePatch::derive\(\)](#), and [s13b\(\)](#).

Here is the caller graph for this function:



6.4.2.19 `s13f()` [1/2]

```

sunrealtype s13f (
    sunrealtype const * udata ) [inline]

```

Definition at line 287 of file [DerivationStencils.h](#).
 00288 { `return` `s13f`(udata, 6); }

References [s13f\(\)](#).

Here is the call graph for this function:



6.4.2.20 s13f() [2/2]

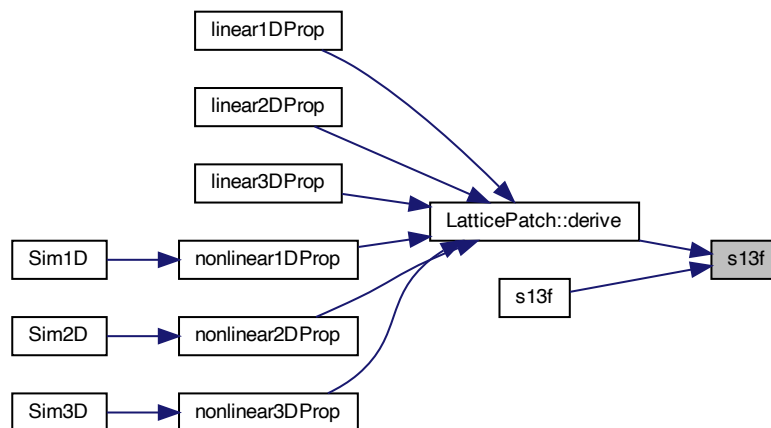
```
sunrealtype s13f (
    sunrealtype const * udata,
    const int dPD ) [inline]
```

Definition at line 225 of file [DerivationStencils.h](#).

```
00225 {
00226     return -1.0 / 12012.0 * udata[-7 * dPD] + 1.0 / 792.0 * udata[-6 * dPD] -
00227            1.0 / 110.0 * udata[-5 * dPD] + 1.0 / 24.0 * udata[-4 * dPD] -
00228            5.0 / 36.0 * udata[-3 * dPD] + 3.0 / 8.0 * udata[-2 * dPD] -
00229            1.0 / 1.0 * udata[-1 * dPD] + 1.0 / 7.0 * udata[0] +
00230            3.0 / 4.0 * udata[1 * dPD] - 5.0 / 24.0 * udata[2 * dPD] +
00231            1.0 / 18.0 * udata[3 * dPD] - 1.0 / 88.0 * udata[4 * dPD] +
00232            1.0 / 660.0 * udata[5 * dPD] - 1.0 / 10296.0 * udata[6 * dPD];
00233 }
```

Referenced by [LatticePatch::derive\(\)](#), and [s13f\(\)](#).

Here is the caller graph for this function:



6.4.2.21 s1b() [1/2]

```
sunrealtype s1b (
    sunrealtype const * udata ) [inline]
```

Definition at line 250 of file [DerivationStencils.h](#).

```
00250 { return s1b(udata, 6); }
```

References [s1b\(\)](#).

Here is the call graph for this function:



6.4.2.22 s1b() [2/2]

```

sunrealtype s1b (
    sunrealtype const * udata,
    const int dPD ) [inline]
  
```

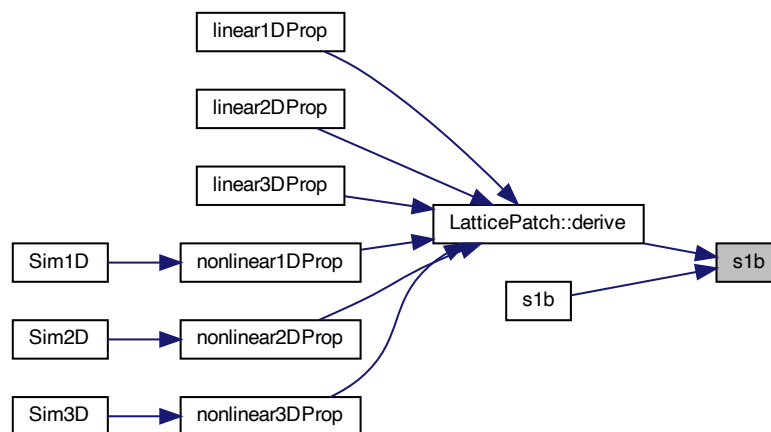
Definition at line 21 of file [DerivationStencils.h](#).

```

00021 {
00022     return -1.0 / 1.0 * udata[0] + udata[1 * dPD];
00023 }
  
```

Referenced by [LatticePatch::derive\(\)](#), and [s1b\(\)](#).

Here is the caller graph for this function:



6.4.2.23 s1f() [1/2]

```
sunrealtype s1f (
    sunrealtype const * udata ) [inline]
```

Definition at line 249 of file [DerivationStencils.h](#).

```
00249 { return s1f(udata, 6); }
```

References [s1f\(\)](#).

Here is the call graph for this function:

**6.4.2.24 s1f()** [2/2]

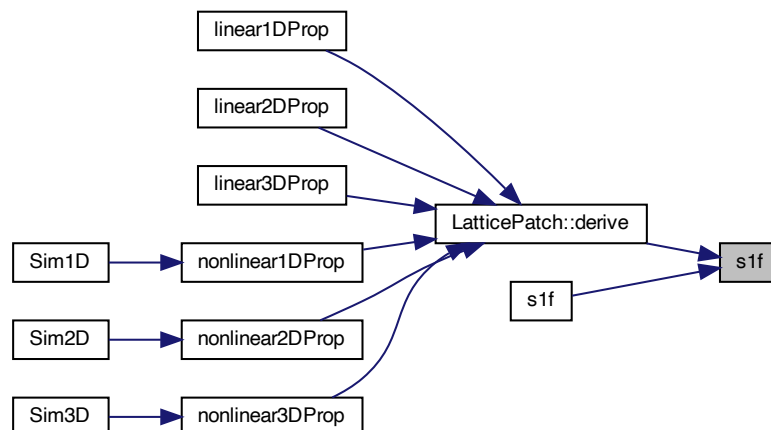
```
sunrealtype s1f (
    sunrealtype const * udata,
    const int dPD ) [inline]
```

Definition at line 16 of file [DerivationStencils.h](#).

```
00016 {
00017     return -1.0 / 1.0 * udata[-1 * dPD] + udata[0];
00018 }
```

Referenced by [LatticePatch::derive\(\)](#), and [s1f\(\)](#).

Here is the caller graph for this function:



6.4.2.25 s2b() [1/2]

```
sunrealtype s2b (
    sunrealtype const * udata ) [inline]
```

Definition at line 253 of file [DerivationStencils.h](#).

```
00253 { return s2b(udata, 6); }
```

References [s2b\(\)](#).

Here is the call graph for this function:

**6.4.2.26 s2b()** [2/2]

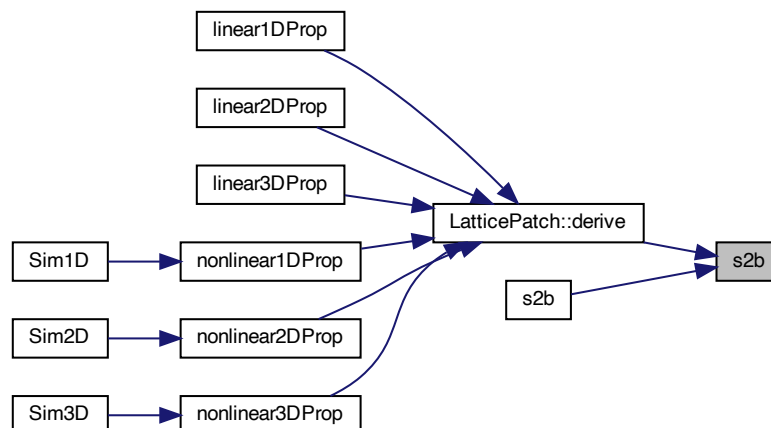
```
sunrealtype s2b (
    sunrealtype const * udata,
    const int dPD ) [inline]
```

Definition at line 35 of file [DerivationStencils.h](#).

```
00035 {
00036     return -3.0 / 2.0 * udata[0] + 2.0 / 1.0 * udata[1 * dPD] -
00037            1.0 / 2.0 * udata[2 * dPD];
00038 }
```

Referenced by [LatticePatch::derive\(\)](#), and [s2b\(\)](#).

Here is the caller graph for this function:



6.4.2.27 s2c() [1/2]

```
sunrealtype s2c (
    sunrealtype const * udata ) [inline]
```

Definition at line 252 of file [DerivationStencils.h](#).

```
00252 { return s2c(udata, 6); }
```

References [s2c\(\)](#).

Here is the call graph for this function:



6.4.2.28 s2c() [2/2]

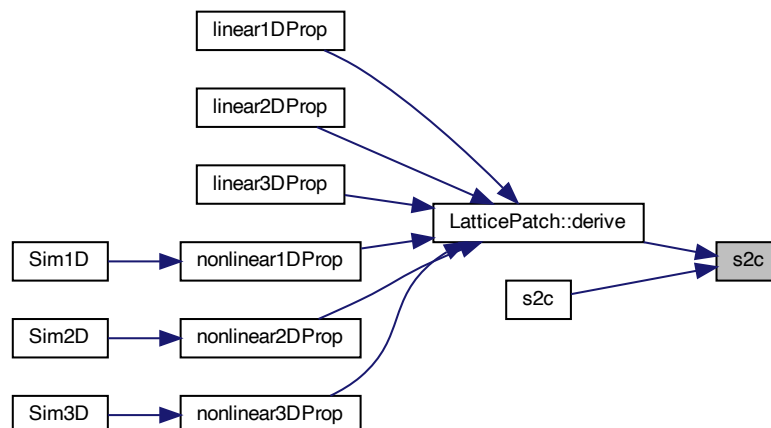
```
sunrealtype s2c (
    sunrealtype const * udata,
    const int dPD ) [inline]
```

Definition at line 31 of file [DerivationStencils.h](#).

```
00031 {
00032     return -1.0 / 2.0 * udata[-1 * dPD] + 0 + 1.0 / 2.0 * udata[1 * dPD];
00033 }
```

Referenced by [LatticePatch::derive\(\)](#), and [s2c\(\)](#).

Here is the caller graph for this function:



6.4.2.29 s2f() [1/2]

```
sunrealtype s2f (
    sunrealtype const * udata ) [inline]
```

Definition at line 251 of file [DerivationStencils.h](#).

```
00251 { return s2f(udata, 6); }
```

References [s2f\(\)](#).

Here is the call graph for this function:

**6.4.2.30 s2f()** [2/2]

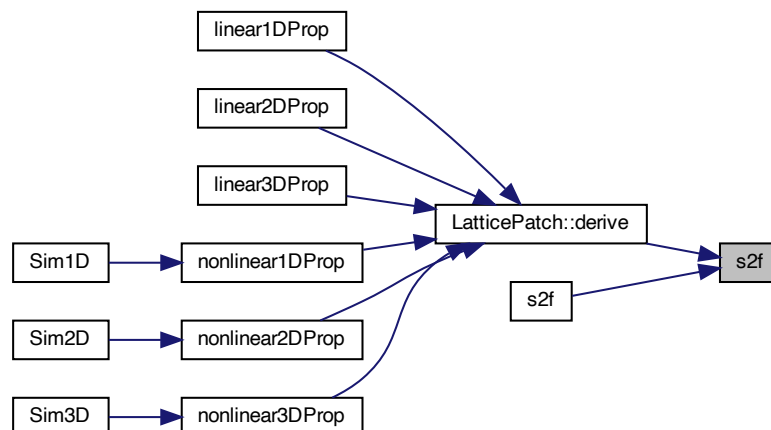
```
sunrealtype s2f (
    sunrealtype const * udata,
    const int dPD ) [inline]
```

Definition at line 26 of file [DerivationStencils.h](#).

```
00026 {
00027     return 1.0 / 2.0 * udata[-2 * dPD] - 2.0 / 1.0 * udata[-1 * dPD] +
00028           3.0 / 2.0 * udata[0];
00029 }
```

Referenced by [LatticePatch::derive\(\)](#), and [s2f\(\)](#).

Here is the caller graph for this function:



6.4.2.31 s3b() [1/2]

```
sunrealtype s3b (
    sunrealtype const * udata ) [inline]
```

Definition at line 255 of file [DerivationStencils.h](#).

```
00255 { return s3b(udata, 6); }
```

References [s3b\(\)](#).

Here is the call graph for this function:

**6.4.2.32 s3b()** [2/2]

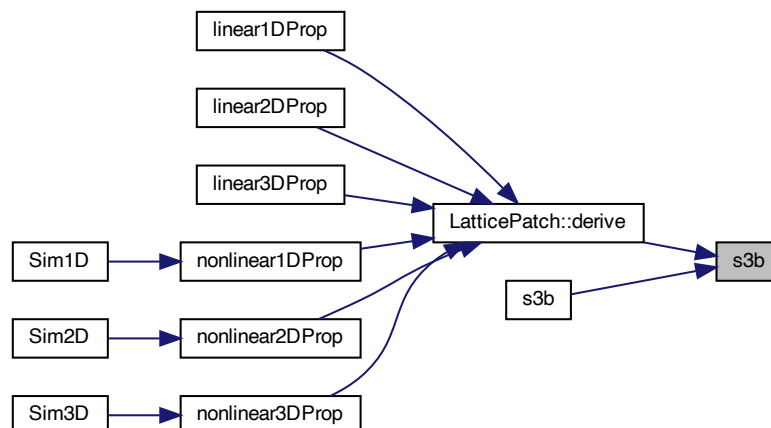
```
sunrealtype s3b (
    sunrealtype const * udata,
    const int dPD ) [inline]
```

Definition at line 45 of file [DerivationStencils.h](#).

```
00045 {
00046     return -1.0 / 3.0 * udata[-1 * dPD] - 1.0 / 2.0 * udata[0] + udata[1 * dPD] -
00047            1.0 / 6.0 * udata[2 * dPD];
00048 }
```

Referenced by [LatticePatch::derive\(\)](#), and [s3b\(\)](#).

Here is the caller graph for this function:



6.4.2.33 s3f() [1/2]

```
sunrealtype s3f (
    sunrealtype const * udata ) [inline]
```

Definition at line 254 of file [DerivationStencils.h](#).

```
00254 { return s3f(udata, 6); }
```

References [s3f\(\)](#).

Here is the call graph for this function:

**6.4.2.34 s3f()** [2/2]

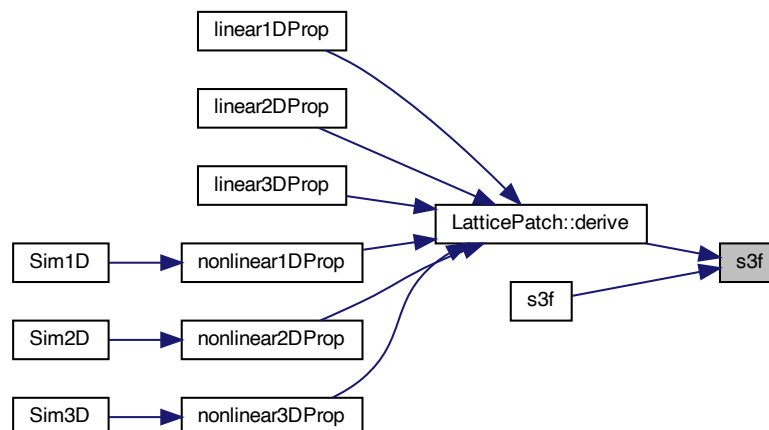
```
sunrealtype s3f (
    sunrealtype const * udata,
    const int dPD ) [inline]
```

Definition at line 40 of file [DerivationStencils.h](#).

```
00040 {
00041     return 1.0 / 6.0 * udata[-2 * dPD] - 1.0 / 1.0 * udata[-1 * dPD] +
00042           1.0 / 2.0 * udata[0] + 1.0 / 3.0 * udata[1 * dPD];
00043 }
```

Referenced by [LatticePatch::derive\(\)](#), and [s3f\(\)](#).

Here is the caller graph for this function:



6.4.2.35 s4b() [1/2]

```
sunrealtype s4b (
    sunrealtype const * udata ) [inline]
```

Definition at line 258 of file [DerivationStencils.h](#).

```
00258 { return s4b(udata, 6); }
```

References [s4b\(\)](#).

Here is the call graph for this function:

**6.4.2.36 s4b()** [2/2]

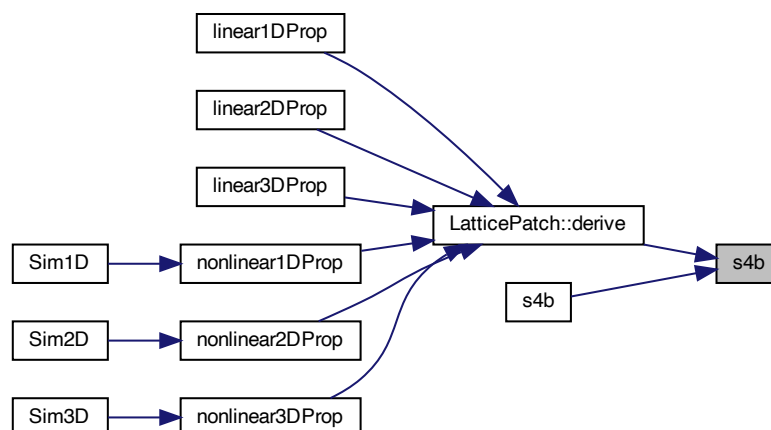
```
sunrealtype s4b (
    sunrealtype const * udata,
    const int dPD ) [inline]
```

Definition at line 61 of file [DerivationStencils.h](#).

```
00061 {
00062     return -1.0 / 4.0 * udata[-1 * dPD] - 5.0 / 6.0 * udata[0] +
00063           3.0 / 2.0 * udata[1 * dPD] - 1.0 / 2.0 * udata[2 * dPD] +
00064           1.0 / 12.0 * udata[3 * dPD];
00065 }
```

Referenced by [LatticePatch::derive\(\)](#), and [s4b\(\)](#).

Here is the caller graph for this function:



6.4.2.37 s4c() [1/2]

```
sunrealtype s4c (
    sunrealtype const * udata ) [inline]
```

Definition at line 257 of file [DerivationStencils.h](#).

```
00257 { return s4c(udata, 6); }
```

References [s4c\(\)](#).

Here is the call graph for this function:

**6.4.2.38 s4c()** [2/2]

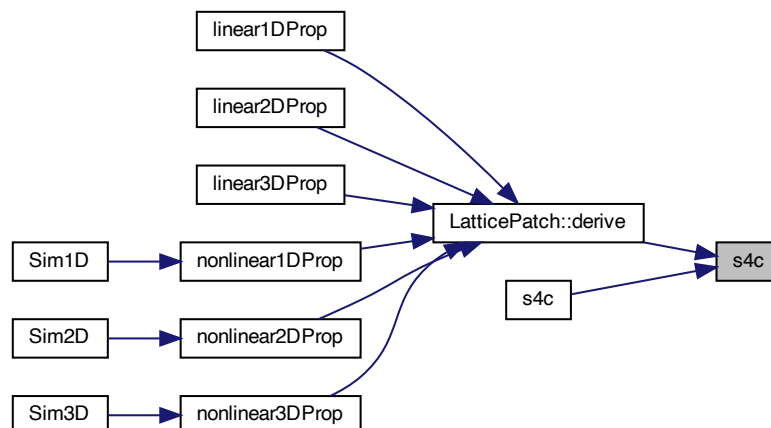
```
sunrealtype s4c (
    sunrealtype const * udata,
    const int dPD ) [inline]
```

Definition at line 56 of file [DerivationStencils.h](#).

```
00056 {
00057     return 1.0 / 12.0 * udata[-2 * dPD] - 2.0 / 3.0 * udata[-1 * dPD] + 0 +
00058           2.0 / 3.0 * udata[1 * dPD] - 1.0 / 12.0 * udata[2 * dPD];
00059 }
```

Referenced by [LatticePatch::derive\(\)](#), and [s4c\(\)](#).

Here is the caller graph for this function:



6.4.2.39 s4f() [1/2]

```
sunrealtype s4f (
    sunrealtype const * udata ) [inline]
```

Definition at line 256 of file [DerivationStencils.h](#).

```
00256 { return s4f(udata, 6); }
```

References [s4f\(\)](#).

Here is the call graph for this function:

**6.4.2.40 s4f()** [2/2]

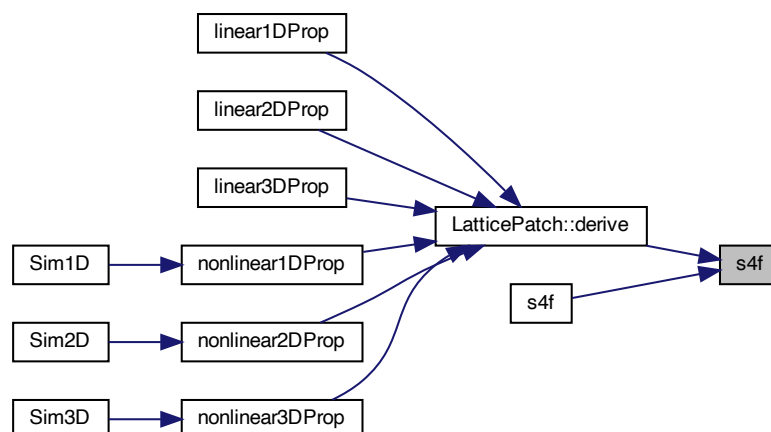
```
sunrealtype s4f (
    sunrealtype const * udata,
    const int dPD ) [inline]
```

Definition at line 50 of file [DerivationStencils.h](#).

```
00050
00051 return -1.0 / 12.0 * udata[-3 * dPD] + 1.0 / 2.0 * udata[-2 * dPD] -
00052         3.0 / 2.0 * udata[-1 * dPD] + 5.0 / 6.0 * udata[0] +
00053         1.0 / 4.0 * udata[1 * dPD];
00054 }
```

Referenced by [LatticePatch::derive\(\)](#), and [s4f\(\)](#).

Here is the caller graph for this function:



6.4.2.41 s5b() [1/2]

```
sunrealtype s5b (
    sunrealtype const * udata ) [inline]
```

Definition at line 260 of file [DerivationStencils.h](#).

```
00260 { return s5b(udata, 6); }
```

References [s5b\(\)](#).

Here is the call graph for this function:

**6.4.2.42 s5b()** [2/2]

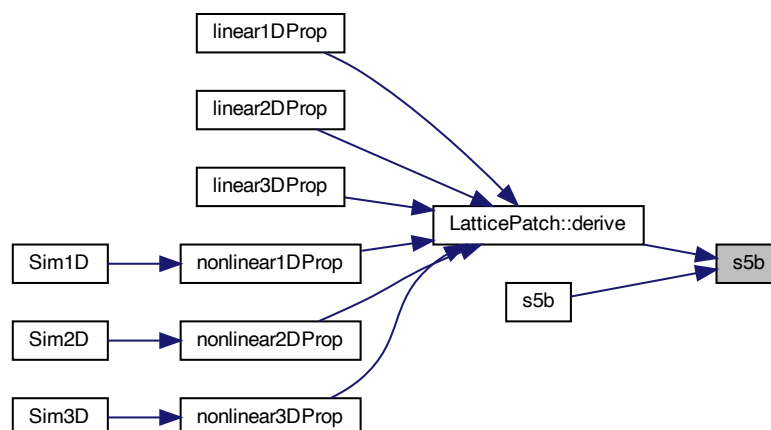
```
sunrealtype s5b (
    sunrealtype const * udata,
    const int dPD ) [inline]
```

Definition at line 73 of file [DerivationStencils.h](#).

```
00073 {
00074     return 1.0 / 20.0 * udata[-2 * dPD] - 1.0 / 2.0 * udata[-1 * dPD] -
00075           1.0 / 3.0 * udata[0] + udata[1 * dPD] - 1.0 / 4.0 * udata[2 * dPD] +
00076           1.0 / 30.0 * udata[3 * dPD];
00077 }
```

Referenced by [LatticePatch::derive\(\)](#), and [s5b\(\)](#).

Here is the caller graph for this function:



6.4.2.43 s5f() [1/2]

```
sunrealtype s5f (
    sunrealtype const * udata ) [inline]
```

Definition at line 259 of file [DerivationStencils.h](#).

```
00259 { return s5f(udata, 6); }
```

References [s5f\(\)](#).

Here is the call graph for this function:

**6.4.2.44 s5f()** [2/2]

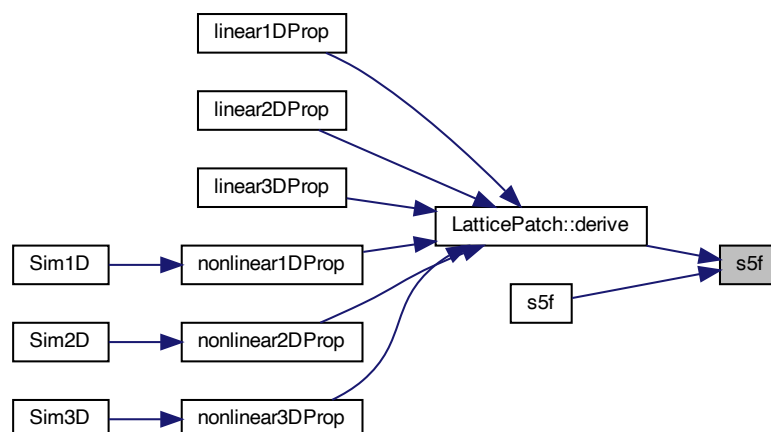
```
sunrealtype s5f (
    sunrealtype const * udata,
    const int dPD ) [inline]
```

Definition at line 67 of file [DerivationStencils.h](#).

```
00067 {
00068     return -1.0 / 30.0 * udata[-3 * dPD] + 1.0 / 4.0 * udata[-2 * dPD] -
00069           1.0 / 1.0 * udata[-1 * dPD] + 1.0 / 3.0 * udata[0] +
00070           1.0 / 2.0 * udata[1 * dPD] - 1.0 / 20.0 * udata[2 * dPD];
00071 }
```

Referenced by [LatticePatch::derive\(\)](#), and [s5f\(\)](#).

Here is the caller graph for this function:



6.4.2.45 s6b() [1/2]

```
sunrealtype s6b (
    sunrealtype const * udata ) [inline]
```

Definition at line 263 of file [DerivationStencils.h](#).

```
00263 { return s6b(udata, 6); }
```

References [s6b\(\)](#).

Here is the call graph for this function:

**6.4.2.46 s6b()** [2/2]

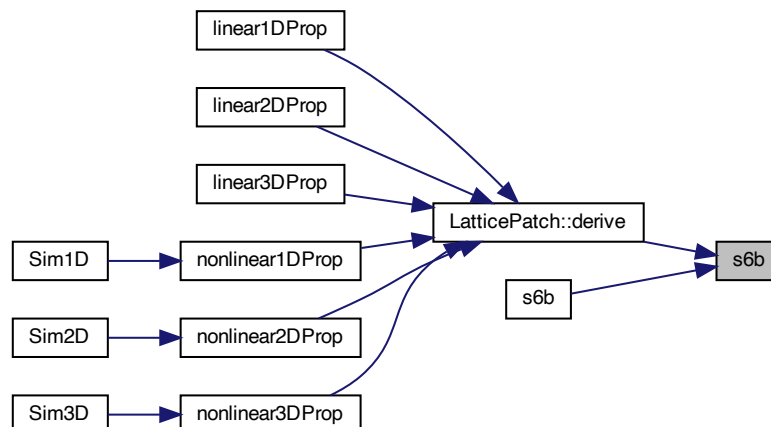
```
sunrealtype s6b (
    sunrealtype const * udata,
    const int dPD ) [inline]
```

Definition at line 92 of file [DerivationStencils.h](#).

```
00092 {
00093     return 1.0 / 30.0 * udata[-2 * dPD] - 2.0 / 5.0 * udata[-1 * dPD] -
00094            7.0 / 12.0 * udata[0] + 4.0 / 3.0 * udata[1 * dPD] -
00095            1.0 / 2.0 * udata[2 * dPD] + 2.0 / 15.0 * udata[3 * dPD] -
00096            1.0 / 60.0 * udata[4 * dPD];
00097 }
```

Referenced by [LatticePatch::derive\(\)](#), and [s6b\(\)](#).

Here is the caller graph for this function:



6.4.2.47 s6c() [1/2]

```
sunrealtype s6c (
    sunrealtype const * udata ) [inline]
```

Definition at line 262 of file [DerivationStencils.h](#).

```
00262 { return s6c(udata, 6); }
```

References [s6c\(\)](#).

Here is the call graph for this function:

**6.4.2.48 s6c()** [2/2]

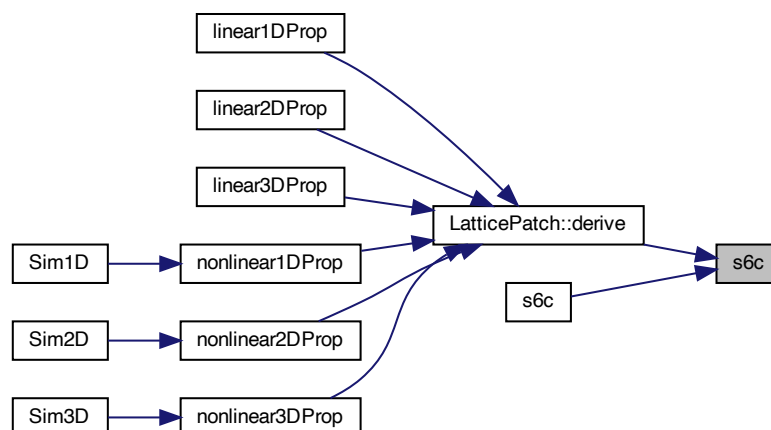
```
sunrealtype s6c (
    sunrealtype const * udata,
    const int dPD ) [inline]
```

Definition at line 86 of file [DerivationStencils.h](#).

```
00086 {
00087     return -1.0 / 60.0 * udata[-3 * dPD] + 3.0 / 20.0 * udata[-2 * dPD] -
00088            3.0 / 4.0 * udata[-1 * dPD] + 0 + 3.0 / 4.0 * udata[1 * dPD] -
00089            3.0 / 20.0 * udata[2 * dPD] + 1.0 / 60.0 * udata[3 * dPD];
00090 }
```

Referenced by [LatticePatch::derive\(\)](#), and [s6c\(\)](#).

Here is the caller graph for this function:



6.4.2.49 s6f() [1/2]

```
sunrealtype s6f (
    sunrealtype const * udata ) [inline]
```

Definition at line 261 of file [DerivationStencils.h](#).

```
00261 { return s6f(udata, 6); }
```

References [s6f\(\)](#).

Here is the call graph for this function:

**6.4.2.50 s6f()** [2/2]

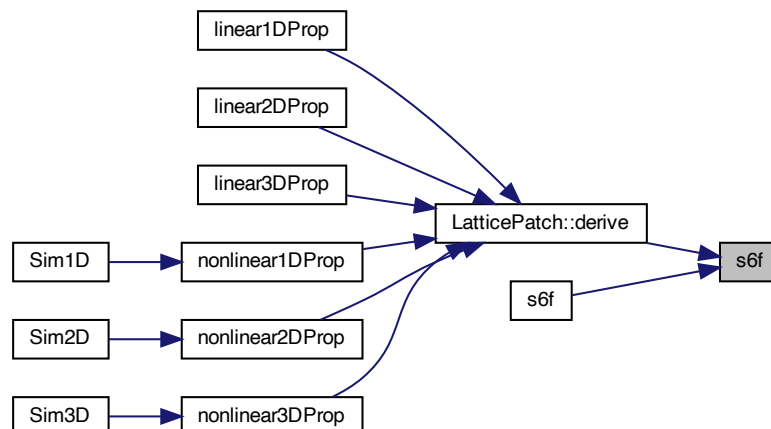
```
sunrealtype s6f (
    sunrealtype const * udata,
    const int dPD ) [inline]
```

Definition at line 79 of file [DerivationStencils.h](#).

```
00079
00080     return 1.0 / 60.0 * udata[-4 * dPD] - 2.0 / 15.0 * udata[-3 * dPD] + {
00081         1.0 / 2.0 * udata[-2 * dPD] - 4.0 / 3.0 * udata[-1 * dPD] +
00082         7.0 / 12.0 * udata[0] + 2.0 / 5.0 * udata[1 * dPD] -
00083         1.0 / 30.0 * udata[2 * dPD];
00084 }
```

Referenced by [LatticePatch::derive\(\)](#), and [s6f\(\)](#).

Here is the caller graph for this function:



6.4.2.51 s7b() [1/2]

```
sunrealtype s7b (
    sunrealtype const * udata ) [inline]
```

Definition at line 265 of file [DerivationStencils.h](#).

```
00265 { return s7b(udata, 6); }
```

References [s7b\(\)](#).

Here is the call graph for this function:

**6.4.2.52 s7b()** [2/2]

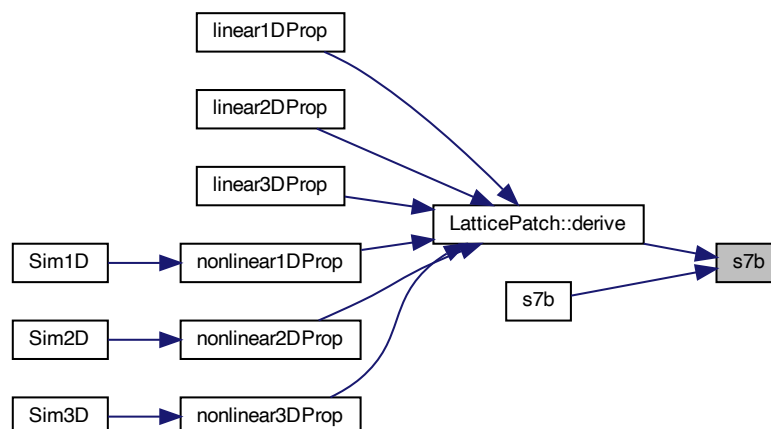
```
sunrealtype s7b (
    sunrealtype const * udata,
    const int dPD ) [inline]
```

Definition at line 106 of file [DerivationStencils.h](#).

```
00106
00107 return -1.0 / 105.0 * udata[-3 * dPD] + 1.0 / 10.0 * udata[-2 * dPD] -
00108         3.0 / 5.0 * udata[-1 * dPD] - 1.0 / 4.0 * udata[0] + udata[1 * dPD] -
00109         3.0 / 10.0 * udata[2 * dPD] + 1.0 / 15.0 * udata[3 * dPD] -
00110         1.0 / 140.0 * udata[4 * dPD];
00111 }
```

Referenced by [LatticePatch::derive\(\)](#), and [s7b\(\)](#).

Here is the caller graph for this function:



6.4.2.53 s7f() [1/2]

```
sunrealtype s7f (
    sunrealtype const * udata ) [inline]
```

Definition at line 264 of file [DerivationStencils.h](#).

```
00264 { return s7f(udata, 6); }
```

References [s7f\(\)](#).

Here is the call graph for this function:

**6.4.2.54 s7f()** [2/2]

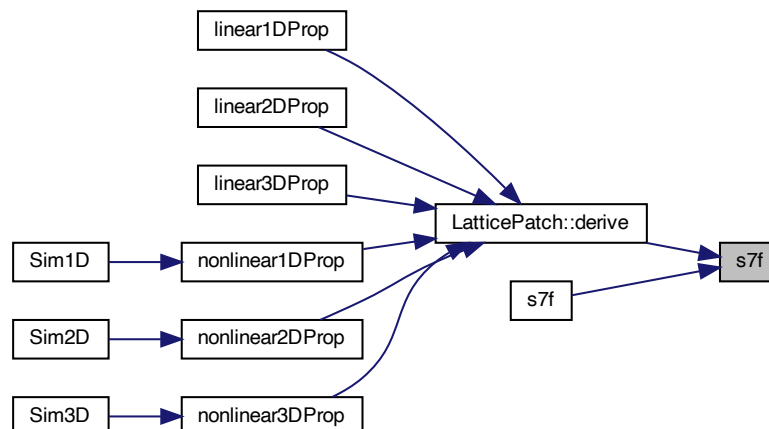
```
sunrealtype s7f (
    sunrealtype const * udata,
    const int dPD ) [inline]
```

Definition at line 99 of file [DerivationStencils.h](#).

```
00099 {
00100     return 1.0 / 140.0 * udata[-4 * dPD] - 1.0 / 15.0 * udata[-3 * dPD] +
00101            3.0 / 10.0 * udata[-2 * dPD] - 1.0 / 1.0 * udata[-1 * dPD] +
00102            1.0 / 4.0 * udata[0] + 3.0 / 5.0 * udata[1 * dPD] -
00103            1.0 / 10.0 * udata[2 * dPD] + 1.0 / 105.0 * udata[3 * dPD];
00104 }
```

Referenced by [LatticePatch::derive\(\)](#), and [s7f\(\)](#).

Here is the caller graph for this function:



6.4.2.55 s8b() [1/2]

```
sunrealtype s8b (
    sunrealtype const * udata ) [inline]
```

Definition at line 268 of file [DerivationStencils.h](#).

```
00268 { return s8b(udata, 6); }
```

References [s8b\(\)](#).

Here is the call graph for this function:



6.4.2.56 s8b() [2/2]

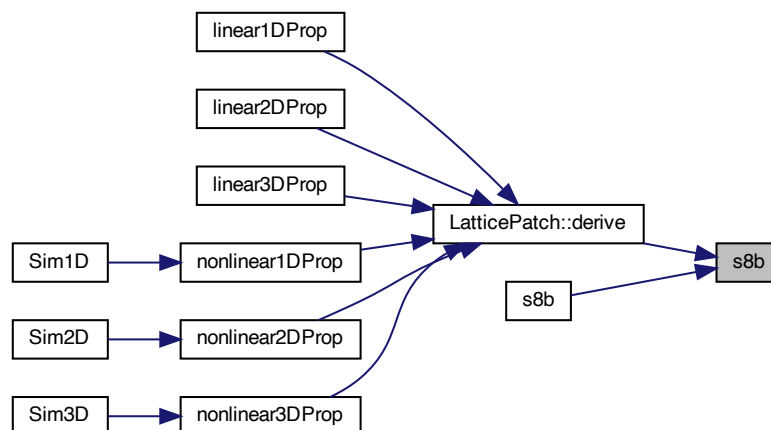
```
sunrealtype s8b (
    sunrealtype const * udata,
    const int dPD ) [inline]
```

Definition at line 128 of file [DerivationStencils.h](#).

```
00128
00129 return -1.0 / 168.0 * udata[-3 * dPD] + 1.0 / 14.0 * udata[-2 * dPD] -
00130         1.0 / 2.0 * udata[-1 * dPD] - 9.0 / 20.0 * udata[0] +
00131         5.0 / 4.0 * udata[1 * dPD] - 1.0 / 2.0 * udata[2 * dPD] +
00132         1.0 / 6.0 * udata[3 * dPD] - 1.0 / 28.0 * udata[4 * dPD] +
00133         1.0 / 280.0 * udata[5 * dPD];
00134 }
```

Referenced by [LatticePatch::derive\(\)](#), and [s8b\(\)](#).

Here is the caller graph for this function:



6.4.2.57 s8c() [1/2]

```
sunrealtype s8c (
    sunrealtype const * udata ) [inline]
```

Definition at line 267 of file [DerivationStencils.h](#).

```
00267 { return s8c(udata, 6); }
```

References [s8c\(\)](#).

Here is the call graph for this function:

**6.4.2.58 s8c()** [2/2]

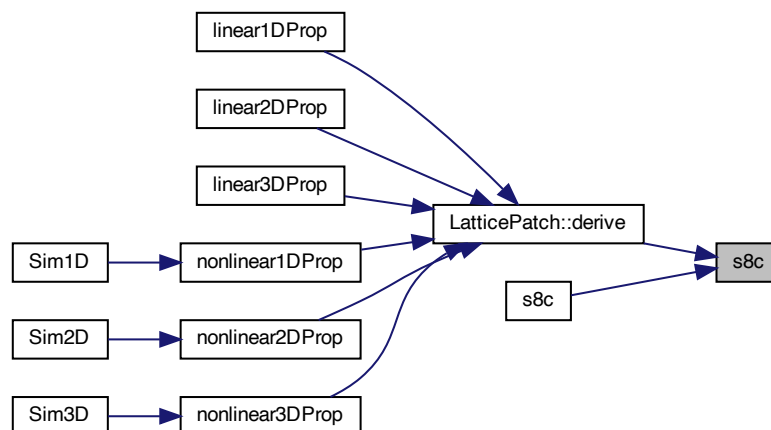
```
sunrealtype s8c (
    sunrealtype const * udata,
    const int dPD ) [inline]
```

Definition at line 121 of file [DerivationStencils.h](#).

```
00121 {
00122     return 1.0 / 280.0 * udata[-4 * dPD] - 4.0 / 105.0 * udata[-3 * dPD] +
00123           1.0 / 5.0 * udata[-2 * dPD] - 4.0 / 5.0 * udata[-1 * dPD] + 0 +
00124           4.0 / 5.0 * udata[1 * dPD] - 1.0 / 5.0 * udata[2 * dPD] +
00125           4.0 / 105.0 * udata[3 * dPD] - 1.0 / 280.0 * udata[4 * dPD];
00126 }
```

Referenced by [LatticePatch::derive\(\)](#), and [s8c\(\)](#).

Here is the caller graph for this function:



6.4.2.59 s8f() [1/2]

```
sunrealtype s8f (
    sunrealtype const * udata ) [inline]
```

Definition at line 266 of file [DerivationStencils.h](#).

```
00266 { return s8f(udata, 6); }
```

References [s8f\(\)](#).

Here is the call graph for this function:

**6.4.2.60 s8f()** [2/2]

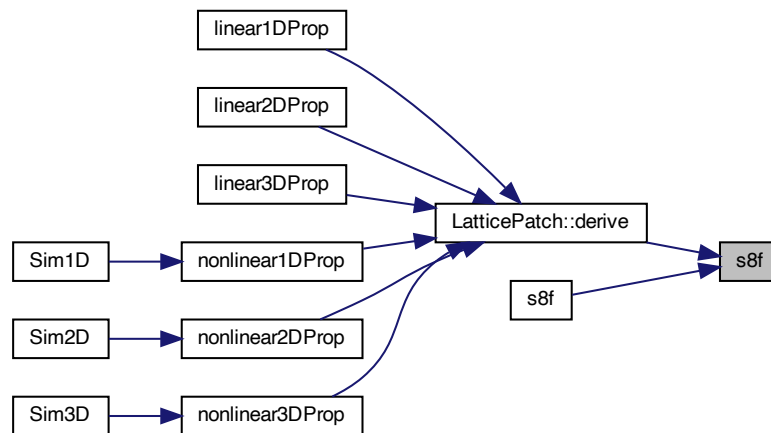
```
sunrealtype s8f (
    sunrealtype const * udata,
    const int dPD ) [inline]
```

Definition at line 113 of file [DerivationStencils.h](#).

```
00113 {
00114     return -1.0 / 280.0 * udata[-5 * dPD] + 1.0 / 28.0 * udata[-4 * dPD] -
00115            1.0 / 6.0 * udata[-3 * dPD] + 1.0 / 2.0 * udata[-2 * dPD] -
00116            5.0 / 4.0 * udata[-1 * dPD] + 9.0 / 20.0 * udata[0] +
00117            1.0 / 2.0 * udata[1 * dPD] - 1.0 / 14.0 * udata[2 * dPD] +
00118            1.0 / 168.0 * udata[3 * dPD];
00119 }
```

Referenced by [LatticePatch::derive\(\)](#), and [s8f\(\)](#).

Here is the caller graph for this function:



6.4.2.61 `s9b()` [1/2]

```

sunrealtype s9b (
    sunrealtype const * udata ) [inline]

```

Definition at line 270 of file [DerivationStencils.h](#).

```

00270 { return s9b(udata, 6); }

```

References [s9b\(\)](#).

Here is the call graph for this function:



6.4.2.62 s9b() [2/2]

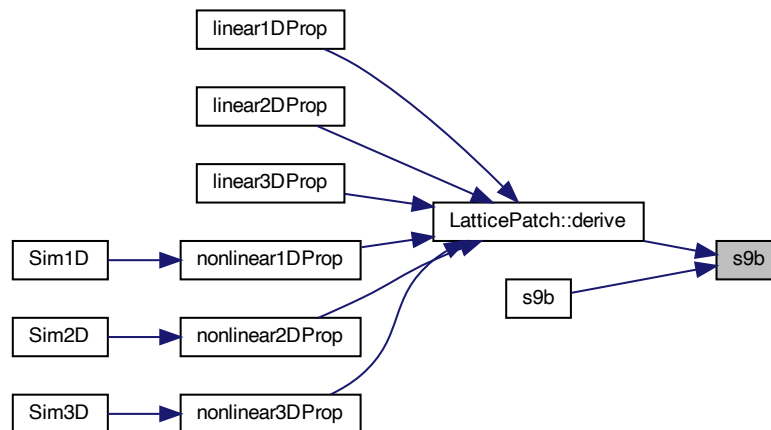
```
sunrealtype s9b (
    sunrealtype const * udata,
    const int dPD ) [inline]
```

Definition at line 144 of file [DerivationStencils.h](#).

```
00144 {
00145     return 1.0 / 504.0 * udata[-4 * dPD] - 1.0 / 42.0 * udata[-3 * dPD] +
00146           1.0 / 7.0 * udata[-2 * dPD] - 2.0 / 3.0 * udata[-1 * dPD] -
00147           1.0 / 5.0 * udata[0] + udata[1 * dPD] - 1.0 / 3.0 * udata[2 * dPD] +
00148           2.0 / 21.0 * udata[3 * dPD] - 1.0 / 56.0 * udata[4 * dPD] +
00149           1.0 / 630.0 * udata[5 * dPD];
00150 }
```

Referenced by [LatticePatch::derive\(\)](#), and [s9b\(\)](#).

Here is the caller graph for this function:



6.4.2.63 s9f() [1/2]

```
sunrealtype s9f (
    sunrealtype const * udata ) [inline]
```

Definition at line 269 of file [DerivationStencils.h](#).

```
00269 { return s9f(udata, 6); }
```

References [s9f\(\)](#).

Here is the call graph for this function:



6.4.2.64 s9f() [2/2]

```

sunrealtype s9f (
    sunrealtype const * udata,
    const int dPD ) [inline]

```

Definition at line 136 of file [DerivationStencils.h](#).

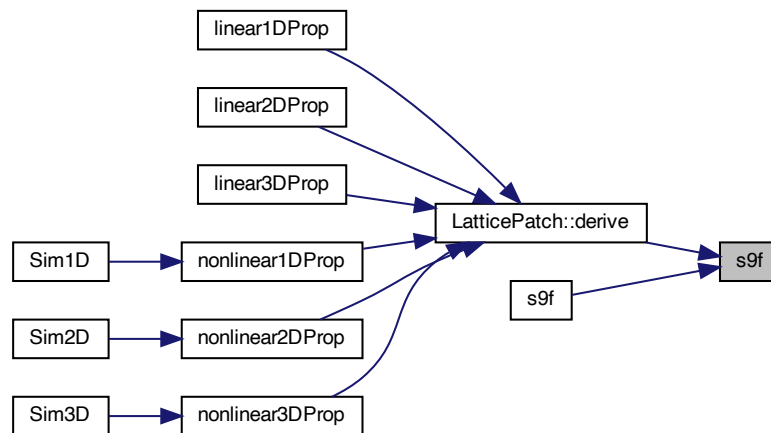
```

00136 {
00137     return -1.0 / 630.0 * udata[-5 * dPD] + 1.0 / 56.0 * udata[-4 * dPD] -
00138           2.0 / 21.0 * udata[-3 * dPD] + 1.0 / 3.0 * udata[-2 * dPD] -
00139           1.0 / 1.0 * udata[-1 * dPD] + 1.0 / 5.0 * udata[0] +
00140           2.0 / 3.0 * udata[1 * dPD] - 1.0 / 7.0 * udata[2 * dPD] +
00141           1.0 / 42.0 * udata[3 * dPD] - 1.0 / 504.0 * udata[4 * dPD];
00142 }

```

Referenced by [LatticePatch::derive\(\)](#), and [s9f\(\)](#).

Here is the caller graph for this function:



6.5 DerivationStencils.h

[Go to the documentation of this file.](#)

```

00001 ///////////////////////////////////////////////////////////////////
00002 /// @file DerivationStencils.h
00003 /// @brief Definition of derivation stencils from order 1 to 13
00004 ///////////////////////////////////////////////////////////////////
00005
00006 #pragma once
00007
00008 #include <sundials/sundials_types.h> /* definition of type sunrealtype */
00009
00010 ///////////////////////////////////////////////////////////////////
00011 // Stencils with variable dPD -- data point dimension //
00012 ///////////////////////////////////////////////////////////////////
00013
00014 // Downwind (forward) dfferentiating
00015 #pragma omp declare simd uniform(dPD) notinbranch
00016 inline sunrealtype s1f(sunrealtype const * udata, const int dPD) {
00017     return -1.0 / 1.0 * udata[-1 * dPD] + udata[0];
00018 }

```

```

00019 // Upwind (backward) differentiating
00020 #pragma omp declare simd uniform(dPD) notinbranch
00021 inline sunrealtype s1b(sunrealtype const *udata, const int dPD) {
00022     return -1.0 / 1.0 * udata[0] + udata[1 * dPD];
00023 }
00024
00025 #pragma omp declare simd uniform(dPD) notinbranch
00026 inline sunrealtype s2f(sunrealtype const *udata, const int dPD) {
00027     return 1.0 / 2.0 * udata[-2 * dPD] - 2.0 / 1.0 * udata[-1 * dPD] +
00028         3.0 / 2.0 * udata[0];
00029 }
00030 #pragma omp declare simd uniform(dPD) notinbranch
00031 inline sunrealtype s2c(sunrealtype const *udata, const int dPD) {
00032     return -1.0 / 2.0 * udata[-1 * dPD] + 0 + 1.0 / 2.0 * udata[1 * dPD];
00033 }
00034 #pragma omp declare simd uniform(dPD) notinbranch
00035 inline sunrealtype s2b(sunrealtype const *udata, const int dPD) {
00036     return -3.0 / 2.0 * udata[0] + 2.0 / 1.0 * udata[1 * dPD] -
00037         1.0 / 2.0 * udata[2 * dPD];
00038 }
00039 #pragma omp declare simd uniform(dPD) notinbranch
00040 inline sunrealtype s3f(sunrealtype const *udata, const int dPD) {
00041     return 1.0 / 6.0 * udata[-2 * dPD] - 1.0 / 1.0 * udata[-1 * dPD] +
00042         1.0 / 2.0 * udata[0] + 1.0 / 3.0 * udata[1 * dPD];
00043 }
00044 #pragma omp declare simd uniform(dPD) notinbranch
00045 inline sunrealtype s3b(sunrealtype const *udata, const int dPD) {
00046     return -1.0 / 3.0 * udata[-1 * dPD] - 1.0 / 2.0 * udata[0] + udata[1 * dPD] -
00047         1.0 / 6.0 * udata[2 * dPD];
00048 }
00049 #pragma omp declare simd uniform(dPD) notinbranch
00050 inline sunrealtype s4f(sunrealtype const *udata, const int dPD) {
00051     return -1.0 / 12.0 * udata[-3 * dPD] + 1.0 / 2.0 * udata[-2 * dPD] -
00052         3.0 / 2.0 * udata[-1 * dPD] + 5.0 / 6.0 * udata[0] +
00053         1.0 / 4.0 * udata[1 * dPD];
00054 }
00055 #pragma omp declare simd uniform(dPD) notinbranch
00056 inline sunrealtype s4c(sunrealtype const *udata, const int dPD) {
00057     return 1.0 / 12.0 * udata[-2 * dPD] - 2.0 / 3.0 * udata[-1 * dPD] + 0 +
00058         2.0 / 3.0 * udata[1 * dPD] - 1.0 / 12.0 * udata[2 * dPD];
00059 }
00060 #pragma omp declare simd uniform(dPD) notinbranch
00061 inline sunrealtype s4b(sunrealtype const *udata, const int dPD) {
00062     return -1.0 / 4.0 * udata[-1 * dPD] - 5.0 / 6.0 * udata[0] +
00063         3.0 / 2.0 * udata[1 * dPD] - 1.0 / 2.0 * udata[2 * dPD] +
00064         1.0 / 12.0 * udata[3 * dPD];
00065 }
00066 #pragma omp declare simd uniform(dPD) notinbranch
00067 inline sunrealtype s5f(sunrealtype const *udata, const int dPD) {
00068     return -1.0 / 30.0 * udata[-3 * dPD] + 1.0 / 4.0 * udata[-2 * dPD] -
00069         1.0 / 1.0 * udata[-1 * dPD] + 1.0 / 3.0 * udata[0] +
00070         1.0 / 2.0 * udata[1 * dPD] - 1.0 / 20.0 * udata[2 * dPD];
00071 }
00072 #pragma omp declare simd uniform(dPD) notinbranch
00073 inline sunrealtype s5b(sunrealtype const *udata, const int dPD) {
00074     return 1.0 / 20.0 * udata[-2 * dPD] - 1.0 / 2.0 * udata[-1 * dPD] -
00075         1.0 / 3.0 * udata[0] + udata[1 * dPD] - 1.0 / 4.0 * udata[2 * dPD] +
00076         1.0 / 30.0 * udata[3 * dPD];
00077 }
00078 #pragma omp declare simd uniform(dPD) notinbranch
00079 inline sunrealtype s6f(sunrealtype const *udata, const int dPD) {
00080     return 1.0 / 60.0 * udata[-4 * dPD] - 2.0 / 15.0 * udata[-3 * dPD] +
00081         1.0 / 2.0 * udata[-2 * dPD] - 4.0 / 3.0 * udata[-1 * dPD] +
00082         7.0 / 12.0 * udata[0] + 2.0 / 5.0 * udata[1 * dPD] -
00083         1.0 / 30.0 * udata[2 * dPD];
00084 }
00085 #pragma omp declare simd uniform(dPD) notinbranch
00086 inline sunrealtype s6c(sunrealtype const *udata, const int dPD) {
00087     return -1.0 / 60.0 * udata[-3 * dPD] + 3.0 / 20.0 * udata[-2 * dPD] -
00088         3.0 / 4.0 * udata[-1 * dPD] + 0 + 3.0 / 4.0 * udata[1 * dPD] -
00089         3.0 / 20.0 * udata[2 * dPD] + 1.0 / 60.0 * udata[3 * dPD];
00090 }
00091 #pragma omp declare simd uniform(dPD) notinbranch
00092 inline sunrealtype s6b(sunrealtype const *udata, const int dPD) {
00093     return 1.0 / 30.0 * udata[-2 * dPD] - 2.0 / 5.0 * udata[-1 * dPD] -
00094         7.0 / 12.0 * udata[0] + 4.0 / 3.0 * udata[1 * dPD] -
00095         1.0 / 2.0 * udata[2 * dPD] + 2.0 / 15.0 * udata[3 * dPD] -
00096         1.0 / 60.0 * udata[4 * dPD];
00097 }
00098 #pragma omp declare simd uniform(dPD) notinbranch
00099 inline sunrealtype s7f(sunrealtype const *udata, const int dPD) {
00100     return 1.0 / 140.0 * udata[-4 * dPD] - 1.0 / 15.0 * udata[-3 * dPD] +
00101         3.0 / 10.0 * udata[-2 * dPD] - 1.0 / 1.0 * udata[-1 * dPD] +
00102         1.0 / 4.0 * udata[0] + 3.0 / 5.0 * udata[1 * dPD] -
00103         1.0 / 10.0 * udata[2 * dPD] + 1.0 / 105.0 * udata[3 * dPD];
00104 }
00105 #pragma omp declare simd uniform(dPD) notinbranch

```



```

00106 inline sunrealtype s7b(sunrealtype const *udata, const int dPD) {
00107     return -1.0 / 105.0 * udata[-3 * dPD] + 1.0 / 10.0 * udata[-2 * dPD] -
00108           3.0 / 5.0 * udata[-1 * dPD] - 1.0 / 4.0 * udata[0] + udata[1 * dPD] -
00109           3.0 / 10.0 * udata[2 * dPD] + 1.0 / 15.0 * udata[3 * dPD] -
00110           1.0 / 140.0 * udata[4 * dPD];
00111 }
00112 #pragma omp declare simd uniform(dPD) notinbranch
00113 inline sunrealtype s8f(sunrealtype const *udata, const int dPD) {
00114     return -1.0 / 280.0 * udata[-5 * dPD] + 1.0 / 28.0 * udata[-4 * dPD] -
00115           1.0 / 6.0 * udata[-3 * dPD] + 1.0 / 2.0 * udata[-2 * dPD] -
00116           5.0 / 4.0 * udata[-1 * dPD] + 9.0 / 20.0 * udata[0] +
00117           1.0 / 2.0 * udata[1 * dPD] - 1.0 / 14.0 * udata[2 * dPD] +
00118           1.0 / 168.0 * udata[3 * dPD];
00119 }
00120 #pragma omp declare simd uniform(dPD) notinbranch
00121 inline sunrealtype s8c(sunrealtype const *udata, const int dPD) {
00122     return 1.0 / 280.0 * udata[-4 * dPD] - 4.0 / 105.0 * udata[-3 * dPD] +
00123           1.0 / 5.0 * udata[-2 * dPD] - 4.0 / 5.0 * udata[-1 * dPD] + 0 +
00124           4.0 / 5.0 * udata[1 * dPD] - 1.0 / 5.0 * udata[2 * dPD] +
00125           4.0 / 105.0 * udata[3 * dPD] - 1.0 / 280.0 * udata[4 * dPD];
00126 }
00127 #pragma omp declare simd uniform(dPD) notinbranch
00128 inline sunrealtype s8b(sunrealtype const *udata, const int dPD) {
00129     return -1.0 / 168.0 * udata[-3 * dPD] + 1.0 / 14.0 * udata[-2 * dPD] -
00130           1.0 / 2.0 * udata[-1 * dPD] - 9.0 / 20.0 * udata[0] +
00131           5.0 / 4.0 * udata[1 * dPD] - 1.0 / 2.0 * udata[2 * dPD] +
00132           1.0 / 6.0 * udata[3 * dPD] - 1.0 / 28.0 * udata[4 * dPD] +
00133           1.0 / 280.0 * udata[5 * dPD];
00134 }
00135 #pragma omp declare simd uniform(dPD) notinbranch
00136 inline sunrealtype s9f(sunrealtype const *udata, const int dPD) {
00137     return -1.0 / 630.0 * udata[-5 * dPD] + 1.0 / 56.0 * udata[-4 * dPD] -
00138           2.0 / 21.0 * udata[-3 * dPD] + 1.0 / 3.0 * udata[-2 * dPD] -
00139           1.0 / 1.0 * udata[-1 * dPD] + 1.0 / 5.0 * udata[0] +
00140           2.0 / 3.0 * udata[1 * dPD] - 1.0 / 7.0 * udata[2 * dPD] +
00141           1.0 / 42.0 * udata[3 * dPD] - 1.0 / 504.0 * udata[4 * dPD];
00142 }
00143 #pragma omp declare simd uniform(dPD) notinbranch
00144 inline sunrealtype s9b(sunrealtype const *udata, const int dPD) {
00145     return 1.0 / 504.0 * udata[-4 * dPD] - 1.0 / 42.0 * udata[-3 * dPD] +
00146           1.0 / 7.0 * udata[-2 * dPD] - 2.0 / 3.0 * udata[-1 * dPD] -
00147           1.0 / 5.0 * udata[0] + udata[1 * dPD] - 1.0 / 3.0 * udata[2 * dPD] +
00148           2.0 / 21.0 * udata[3 * dPD] - 1.0 / 56.0 * udata[4 * dPD] +
00149           1.0 / 630.0 * udata[5 * dPD];
00150 }
00151 #pragma omp declare simd uniform(dPD) notinbranch
00152 inline sunrealtype s10f(sunrealtype const *udata, const int dPD) {
00153     return 1.0 / 1260.0 * udata[-6 * dPD] - 1.0 / 105.0 * udata[-5 * dPD] +
00154           3.0 / 56.0 * udata[-4 * dPD] - 4.0 / 21.0 * udata[-3 * dPD] +
00155           1.0 / 2.0 * udata[-2 * dPD] - 6.0 / 5.0 * udata[-1 * dPD] +
00156           11.0 / 30.0 * udata[0] + 4.0 / 7.0 * udata[1 * dPD] -
00157           3.0 / 28.0 * udata[2 * dPD] + 1.0 / 63.0 * udata[3 * dPD] -
00158           1.0 / 840.0 * udata[4 * dPD];
00159 }
00160 #pragma omp declare simd uniform(dPD) notinbranch
00161 inline sunrealtype s10c(sunrealtype const *udata, const int dPD) {
00162     return -1.0 / 1260.0 * udata[-5 * dPD] + 5.0 / 504.0 * udata[-4 * dPD] -
00163           5.0 / 84.0 * udata[-3 * dPD] + 5.0 / 21.0 * udata[-2 * dPD] -
00164           5.0 / 6.0 * udata[-1 * dPD] + 0 + 5.0 / 6.0 * udata[1 * dPD] -
00165           5.0 / 21.0 * udata[2 * dPD] + 5.0 / 84.0 * udata[3 * dPD] -
00166           5.0 / 504.0 * udata[4 * dPD] + 1.0 / 1260.0 * udata[5 * dPD];
00167 }
00168 #pragma omp declare simd uniform(dPD) notinbranch
00169 inline sunrealtype s10b(sunrealtype const *udata, const int dPD) {
00170     return 1.0 / 840.0 * udata[-4 * dPD] - 1.0 / 63.0 * udata[-3 * dPD] +
00171           3.0 / 28.0 * udata[-2 * dPD] - 4.0 / 7.0 * udata[-1 * dPD] -
00172           11.0 / 30.0 * udata[0] + 6.0 / 5.0 * udata[1 * dPD] -
00173           1.0 / 2.0 * udata[2 * dPD] + 4.0 / 21.0 * udata[3 * dPD] -
00174           3.0 / 56.0 * udata[4 * dPD] + 1.0 / 105.0 * udata[5 * dPD] -
00175           1.0 / 1260.0 * udata[6 * dPD];
00176 }
00177 #pragma omp declare simd uniform(dPD) notinbranch
00178 inline sunrealtype s11f(sunrealtype const *udata, const int dPD) {
00179     return 1.0 / 2772.0 * udata[-6 * dPD] - 1.0 / 210.0 * udata[-5 * dPD] +
00180           5.0 / 168.0 * udata[-4 * dPD] - 5.0 / 42.0 * udata[-3 * dPD] +
00181           5.0 / 14.0 * udata[-2 * dPD] - 1.0 / 1.0 * udata[-1 * dPD] +
00182           1.0 / 6.0 * udata[0] + 5.0 / 7.0 * udata[1 * dPD] -
00183           5.0 / 28.0 * udata[2 * dPD] + 5.0 / 126.0 * udata[3 * dPD] -
00184           1.0 / 168.0 * udata[4 * dPD] + 1.0 / 2310.0 * udata[5 * dPD];
00185 }
00186 #pragma omp declare simd uniform(dPD) notinbranch
00187 inline sunrealtype s11b(sunrealtype const *udata, const int dPD) {
00188     return -1.0 / 2310.0 * udata[-5 * dPD] + 1.0 / 168.0 * udata[-4 * dPD] -
00189           5.0 / 126.0 * udata[-3 * dPD] + 5.0 / 28.0 * udata[-2 * dPD] -
00190           5.0 / 7.0 * udata[-1 * dPD] - 1.0 / 6.0 * udata[0] + udata[1 * dPD] -
00191           5.0 / 14.0 * udata[2 * dPD] + 5.0 / 42.0 * udata[3 * dPD] -
00192           5.0 / 168.0 * udata[4 * dPD] + 1.0 / 210.0 * udata[5 * dPD] -

```

```

00193         1.0 / 2772.0 * udata[6 * dPD];
00194     }
00195     #pragma omp declare simd uniform(dPD) notinbranch
00196     inline sunrealtype s12f(sunrealtype const *udata, const int dPD) {
00197         return -1.0 / 5544.0 * udata[-7 * dPD] + 1.0 / 396.0 * udata[-6 * dPD] -
00198             1.0 / 60.0 * udata[-5 * dPD] + 5.0 / 72.0 * udata[-4 * dPD] -
00199             5.0 / 24.0 * udata[-3 * dPD] + 1.0 / 2.0 * udata[-2 * dPD] -
00200             7.0 / 6.0 * udata[-1 * dPD] + 13.0 / 42.0 * udata[0] +
00201             5.0 / 8.0 * udata[1 * dPD] - 5.0 / 36.0 * udata[2 * dPD] +
00202             1.0 / 36.0 * udata[3 * dPD] - 1.0 / 264.0 * udata[4 * dPD] +
00203             1.0 / 3960.0 * udata[5 * dPD];
00204     }
00205     #pragma omp declare simd uniform(dPD) notinbranch
00206     inline sunrealtype s12c(sunrealtype const *udata, const int dPD) {
00207         return 1.0 / 5544.0 * udata[-6 * dPD] - 1.0 / 385.0 * udata[-5 * dPD] +
00208             1.0 / 56.0 * udata[-4 * dPD] - 5.0 / 63.0 * udata[-3 * dPD] +
00209             15.0 / 56.0 * udata[-2 * dPD] - 6.0 / 7.0 * udata[-1 * dPD] + 0 +
00210             6.0 / 7.0 * udata[1 * dPD] - 15.0 / 56.0 * udata[2 * dPD] +
00211             5.0 / 63.0 * udata[3 * dPD] - 1.0 / 56.0 * udata[4 * dPD] +
00212             1.0 / 385.0 * udata[5 * dPD] - 1.0 / 5544.0 * udata[6 * dPD];
00213     }
00214     #pragma omp declare simd uniform(dPD) notinbranch
00215     inline sunrealtype s12b(sunrealtype const *udata, const int dPD) {
00216         return -1.0 / 3960.0 * udata[-5 * dPD] + 1.0 / 264.0 * udata[-4 * dPD] -
00217             1.0 / 36.0 * udata[-3 * dPD] + 5.0 / 36.0 * udata[-2 * dPD] -
00218             5.0 / 8.0 * udata[-1 * dPD] - 13.0 / 42.0 * udata[0] +
00219             7.0 / 6.0 * udata[1 * dPD] - 1.0 / 2.0 * udata[2 * dPD] +
00220             5.0 / 24.0 * udata[3 * dPD] - 5.0 / 72.0 * udata[4 * dPD] +
00221             1.0 / 60.0 * udata[5 * dPD] - 1.0 / 396.0 * udata[6 * dPD] +
00222             1.0 / 5544.0 * udata[7 * dPD];
00223     }
00224     #pragma omp declare simd uniform(dPD) notinbranch
00225     inline sunrealtype s13f(sunrealtype const *udata, const int dPD) {
00226         return -1.0 / 12012.0 * udata[-7 * dPD] + 1.0 / 792.0 * udata[-6 * dPD] -
00227             1.0 / 110.0 * udata[-5 * dPD] + 1.0 / 24.0 * udata[-4 * dPD] -
00228             5.0 / 36.0 * udata[-3 * dPD] + 3.0 / 8.0 * udata[-2 * dPD] -
00229             1.0 / 1.0 * udata[-1 * dPD] + 1.0 / 7.0 * udata[0] +
00230             3.0 / 4.0 * udata[1 * dPD] - 5.0 / 24.0 * udata[2 * dPD] +
00231             1.0 / 18.0 * udata[3 * dPD] - 1.0 / 88.0 * udata[4 * dPD] +
00232             1.0 / 660.0 * udata[5 * dPD] - 1.0 / 10296.0 * udata[6 * dPD];
00233     }
00234     #pragma omp declare simd uniform(dPD) notinbranch
00235     inline sunrealtype s13b(sunrealtype const *udata, const int dPD) {
00236         return 1.0 / 10296.0 * udata[-6 * dPD] - 1.0 / 660.0 * udata[-5 * dPD] +
00237             1.0 / 88.0 * udata[-4 * dPD] - 1.0 / 18.0 * udata[-3 * dPD] +
00238             5.0 / 24.0 * udata[-2 * dPD] - 3.0 / 4.0 * udata[-1 * dPD] -
00239             1.0 / 7.0 * udata[0] + udata[1 * dPD] - 3.0 / 8.0 * udata[2 * dPD] +
00240             5.0 / 36.0 * udata[3 * dPD] - 1.0 / 24.0 * udata[4 * dPD] +
00241             1.0 / 110.0 * udata[5 * dPD] - 1.0 / 792.0 * udata[6 * dPD] +
00242             1.0 / 12012.0 * udata[7 * dPD];
00243     }
00244
00245     //////////////////////////////////////////
00246     // Stencils with dPD fixed to 6 //
00247     //////////////////////////////////////////
00248
00249     inline sunrealtype s1f(sunrealtype const *udata) { return s1f(udata, 6); }
00250     inline sunrealtype s1b(sunrealtype const *udata) { return s1b(udata, 6); }
00251     inline sunrealtype s2f(sunrealtype const *udata) { return s2f(udata, 6); }
00252     inline sunrealtype s2c(sunrealtype const *udata) { return s2c(udata, 6); }
00253     inline sunrealtype s2b(sunrealtype const *udata) { return s2b(udata, 6); }
00254     inline sunrealtype s3f(sunrealtype const *udata) { return s3f(udata, 6); }
00255     inline sunrealtype s3b(sunrealtype const *udata) { return s3b(udata, 6); }
00256     inline sunrealtype s4f(sunrealtype const *udata) { return s4f(udata, 6); }
00257     inline sunrealtype s4c(sunrealtype const *udata) { return s4c(udata, 6); }
00258     inline sunrealtype s4b(sunrealtype const *udata) { return s4b(udata, 6); }
00259     inline sunrealtype s5f(sunrealtype const *udata) { return s5f(udata, 6); }
00260     inline sunrealtype s5b(sunrealtype const *udata) { return s5b(udata, 6); }
00261     inline sunrealtype s6f(sunrealtype const *udata) { return s6f(udata, 6); }
00262     inline sunrealtype s6c(sunrealtype const *udata) { return s6c(udata, 6); }
00263     inline sunrealtype s6b(sunrealtype const *udata) { return s6b(udata, 6); }
00264     inline sunrealtype s7f(sunrealtype const *udata) { return s7f(udata, 6); }
00265     inline sunrealtype s7b(sunrealtype const *udata) { return s7b(udata, 6); }
00266     inline sunrealtype s8f(sunrealtype const *udata) { return s8f(udata, 6); }
00267     inline sunrealtype s8c(sunrealtype const *udata) { return s8c(udata, 6); }
00268     inline sunrealtype s8b(sunrealtype const *udata) { return s8b(udata, 6); }
00269     inline sunrealtype s9f(sunrealtype const *udata) { return s9f(udata, 6); }
00270     inline sunrealtype s9b(sunrealtype const *udata) { return s9b(udata, 6); }
00271     inline sunrealtype s10f(sunrealtype const *udata)
00272     { return s10f(udata, 6); }
00273     inline sunrealtype s10c(sunrealtype const *udata)
00274     { return s10c(udata, 6); }
00275     inline sunrealtype s10b(sunrealtype const *udata)
00276     { return s10b(udata, 6); }
00277     inline sunrealtype s11f(sunrealtype const *udata)
00278     { return s11f(udata, 6); }
00279     inline sunrealtype s11b(sunrealtype const *udata)

```

```

00280 { return s11b(udata, 6); }
00281 inline sunrealtype s12f(sunrealtype const *udata)
00282 { return s12f(udata, 6); }
00283 inline sunrealtype s12c(sunrealtype const *udata)
00284 { return s12c(udata, 6); }
00285 inline sunrealtype s12b(sunrealtype const *udata)
00286 { return s12b(udata, 6); }
00287 inline sunrealtype s13f(sunrealtype const *udata)
00288 { return s13f(udata, 6); }
00289 inline sunrealtype s13b(sunrealtype const *udata)
00290 { return s13b(udata, 6); }
00291

```

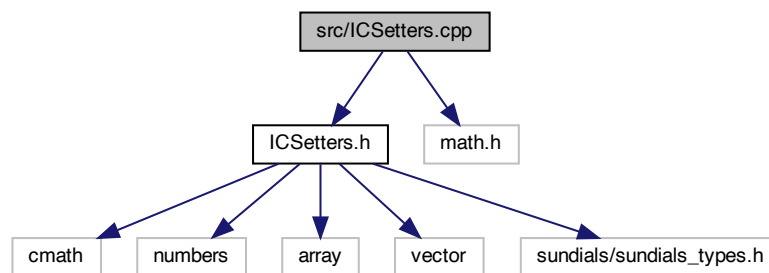
6.6 src/ICSetters.cpp File Reference

Implementation of the plane wave and Gaussian wave packets.

```
#include "ICSetters.h"
```

```
#include <math.h>
```

Include dependency graph for ICSetters.cpp:



6.6.1 Detailed Description

Implementation of the plane wave and Gaussian wave packets.

Definition in file [ICSetters.cpp](#).

6.7 ICSetters.cpp

[Go to the documentation of this file.](#)

```

00001 //////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
00002 /// @file ICSetters.cpp
00003 /// @brief Implementation of the plane wave and Gaussian wave packets
00004 //////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
00005
00006 #include "ICSetters.h"
00007
00008 #include <math.h>
00009
00010 /** PlaneWave1D construction with */
00011 PlaneWave1D::PlaneWave1D(std::array<sunrealtype, 3> k,
00012     std::array<sunrealtype, 3> p,
00013     std::array<sunrealtype, 3> phi) {
00014     kx = k[0]; /** - wavevectors \f$ k_x \f$ */

```

```

00015 ky = k[1]; /** - \f$ k_y \f$ */
00016 kz = k[2]; /** - \f$ k_z \f$ normalized to \f$ 1/\lambda \f$ */
00017 // Amplitude bug: lower by factor 3
00018 px = p[0] / 3; /** - amplitude (polarization) in x-direction \f$ p_x \f$ */
00019 py = p[1] / 3; /** - amplitude (polarization) in y-direction \f$ p_y \f$ */
00020 pz = p[2] / 3; /** - amplitude (polarization) in z-direction \f$ p_z \f$ */
00021 phix = phi[0]; /** - phase shift in x-direction \f$ \phi_x \f$ */
00022 phiy = phi[1]; /** - phase shift in y-direction \f$ \phi_y \f$ */
00023 phiz = phi[2]; /** - phase shift in z-direction \f$ \phi_z \f$ */
00024 }
00025
00026 /** PlaneWave1D implementation in space */
00027 void PlaneWave1D::addToSpace(const sunrealtype x, const sunrealtype y,
00028     const sunrealtype z,
00029     sunrealtype *pTo6Space) const {
00030     const sunrealtype wavelength =
00031         sqrt(kx * kx + ky * ky + kz * kz); /** \f$ 1/\lambda \f$ */
00032     const sunrealtype kScalarX = (kx * x + ky * y + kz * z) * 2 *
00033         std::numbers::pi; /** \f$ 2\pi \ \vec{k} \cdot \vec{x} \f$ */
00034     // Plane wave definition
00035     const std::array<sunrealtype, 3> E{ /** E-field vector */
00036         px * cos(kScalarX - phix), /** \f$ E_x \f$ */
00037         py * cos(kScalarX - phiy), /** \f$ E_y \f$ */
00038         pz * cos(kScalarX - phiz)}; /** \f$ E_z \f$ */
00039     // Put E-field into space
00040     pTo6Space[0] += E[0];
00041     pTo6Space[1] += E[1];
00042     pTo6Space[2] += E[2];
00043     // and B-field
00044     pTo6Space[3] += (ky * E[2] - kz * E[1]) / wavelength;
00045     pTo6Space[4] += (kz * E[0] - kx * E[2]) / wavelength;
00046     pTo6Space[5] += (kx * E[1] - ky * E[0]) / wavelength;
00047 }
00048
00049 /** PlaneWave2D construction with */
00050 PlaneWave2D::PlaneWave2D(std::array<sunrealtype, 3> k,
00051     std::array<sunrealtype, 3> p,
00052     std::array<sunrealtype, 3> phi) {
00053     kx = k[0]; /** - wavevectors \f$ k_x \f$ */
00054     ky = k[1]; /** - \f$ k_y \f$ */
00055     kz = k[2]; /** - \f$ k_z \f$ normalized to \f$ 1/\lambda \f$ */
00056     // Amplitude bug: lower by factor 9
00057     px = p[0] / 9; /** - amplitude (polarization) in x-direction \f$ p_x \f$ */
00058     py = p[1] / 9; /** - amplitude (polarization) in y-direction \f$ p_y \f$ */
00059     pz = p[2] / 9; /** - amplitude (polarization) in z-direction \f$ p_z \f$ */
00060     phix = phi[0]; /** - phase shift in x-direction \f$ \phi_x \f$ */
00061     phiy = phi[1]; /** - phase shift in y-direction \f$ \phi_y \f$ */
00062     phiz = phi[2]; /** - phase shift in z-direction \f$ \phi_z \f$ */
00063 }
00064
00065 /** PlaneWave2D implementation in space */
00066 void PlaneWave2D::addToSpace(const sunrealtype x, const sunrealtype y,
00067     const sunrealtype z, sunrealtype *pTo6Space) const {
00068     const sunrealtype wavelength =
00069         sqrt(kx * kx + ky * ky + kz * kz); /** \f$ 1/\lambda \f$ */
00070     const sunrealtype kScalarX = (kx * x + ky * y + kz * z) * 2 *
00071         std::numbers::pi; /** \f$ 2\pi \ \vec{k} \cdot \vec{x} \f$ */
00072     // Plane wave definition
00073     const std::array<sunrealtype, 3> E{ /** E-field vector */
00074         px * cos(kScalarX - phix), /** \f$ E_x \f$ */
00075         py * cos(kScalarX - phiy), /** \f$ E_y \f$ */
00076         pz * cos(kScalarX - phiz)}; /** \f$ E_z \f$ */
00077     // Put E-field into space
00078     pTo6Space[0] += E[0];
00079     pTo6Space[1] += E[1];
00080     pTo6Space[2] += E[2];
00081     // and B-field
00082     pTo6Space[3] += (ky * E[2] - kz * E[1]) / wavelength;
00083     pTo6Space[4] += (kz * E[0] - kx * E[2]) / wavelength;
00084     pTo6Space[5] += (kx * E[1] - ky * E[0]) / wavelength;
00085 }
00086
00087 /** PlaneWave3D construction with */
00088 PlaneWave3D::PlaneWave3D(std::array<sunrealtype, 3> k,
00089     std::array<sunrealtype, 3> p,
00090     std::array<sunrealtype, 3> phi) {
00091     kx = k[0]; /** - wavevectors \f$ k_x \f$ */
00092     ky = k[1]; /** - \f$ k_y \f$ */
00093     kz = k[2]; /** - \f$ k_z \f$ normalized to \f$ 1/\lambda \f$ */
00094     px = p[0]; /** - amplitude (polarization) in x-direction \f$ p_x \f$ */
00095     py = p[1]; /** - amplitude (polarization) in y-direction \f$ p_y \f$ */
00096     pz = p[2]; /** - amplitude (polarization) in z-direction \f$ p_z \f$ */
00097     phix = phi[0]; /** - phase shift in x-direction \f$ \phi_x \f$ */
00098     phiy = phi[1]; /** - phase shift in y-direction \f$ \phi_y \f$ */
00099     phiz = phi[2]; /** - phase shift in z-direction \f$ \phi_z \f$ */
00100 }
00101

```

```

00102 /** PlaneWave3D implementation in space */
00103 void PlaneWave3D::addToSpace(sunrealtype x, unrealtype y, unrealtype z,
00104                             unrealtype *pTo6Space) const {
00105     const unrealtype wavelength =
00106         sqrt(kx * kx + ky * ky + kz * kz); /* \f$ 1/\lambda \f$ */
00107     const unrealtype kScalarX = (kx * x + ky * y + kz * z) * 2 *
00108         std::numbers::pi; /* \f$ 2\pi \cdot \vec{k} \cdot \vec{x} \f$ */
00109     // Plane wave definition
00110     const std::array<unrealtype, 3> E{ /* E-field vector \f$ \vec{E} \f$ */
00111         px * cos(kScalarX - phix), /* \f$ E_x \f$ */
00112         py * cos(kScalarX - phiy), /* \f$ E_y \f$ */
00113         pz * cos(kScalarX - phiz)}; /* \f$ E_z \f$ */
00114     // Put E-field into space
00115     pTo6Space[0] += E[0];
00116     pTo6Space[1] += E[1];
00117     pTo6Space[2] += E[2];
00118     // and B-field
00119     pTo6Space[3] += (ky * E[2] - kz * E[1]) / wavelength;
00120     pTo6Space[4] += (kz * E[0] - kx * E[2]) / wavelength;
00121     pTo6Space[5] += (kx * E[1] - ky * E[0]) / wavelength;
00122 }
00123
00124 /** Gauss1D construction with */
00125 Gauss1D::Gauss1D(std::array<unrealtype, 3> k, std::array<unrealtype, 3> p,
00126                  std::array<unrealtype, 3> xo, unrealtype phig_,
00127                  std::array<unrealtype, 3> phi) {
00128     kx = k[0]; /* - wavevectors \f$ k_x \f$ */
00129     ky = k[1]; /* - \f$ k_y \f$ */
00130     kz = k[2]; /* - \f$ k_z \f$ normalized to \f$ 1/\lambda \f$ */
00131     px = p[0]; /* - amplitude (polarization) in x-direction */
00132     py = p[1]; /* - amplitude (polarization) in y-direction */
00133     pz = p[2]; /* - amplitude (polarization) in z-direction */
00134     phix = phi[0]; /* - phase shift in x-direction */
00135     phiy = phi[1]; /* - phase shift in y-direction */
00136     phiz = phi[2]; /* - phase shift in z-direction */
00137     phig = phig_; /* - width */
00138     x0x = xo[0]; /* - shift from origin in x-direction */
00139     x0y = xo[1]; /* - shift from origin in y-direction */
00140     x0z = xo[2]; /* - shift from origin in z-direction */
00141 }
00142
00143 /** Gauss1D implementation in space */
00144 void Gauss1D::addToSpace(sunrealtype x, unrealtype y, unrealtype z,
00145                          unrealtype *pTo6Space) const {
00146     const unrealtype wavelength =
00147         sqrt(kx * kx + ky * ky + kz * kz); /* \f$ 1/\lambda \f$ */
00148     x = x - x0x; /* x-coordinate minus shift from origin */
00149     y = y - x0y; /* y-coordinate minus shift from origin */
00150     z = z - x0z; /* z-coordinate minus shift from origin */
00151     const unrealtype kScalarX = (kx * x + ky * y + kz * z) * 2 *
00152         std::numbers::pi; /* \f$ 2\pi \cdot \vec{k} \cdot \vec{x} \f$ */
00153     const unrealtype envelopeAmp =
00154         exp(-(x * x + y * y + z * z) / phig / phig); /* enveloping Gauss shape */
00155     // Gaussian wave definition
00156     const std::array<unrealtype, 3> E{
00157         {
00158             px * cos(kScalarX - phix) * envelopeAmp, /* \f$ E_x \f$ */
00159             py * cos(kScalarX - phiy) * envelopeAmp, /* \f$ E_y \f$ */
00160             pz * cos(kScalarX - phiz) * envelopeAmp}; /* \f$ E_z \f$ */
00161     // Put E-field into space
00162     pTo6Space[0] += E[0];
00163     pTo6Space[1] += E[1];
00164     pTo6Space[2] += E[2];
00165     // and B-field
00166     pTo6Space[3] += (ky * E[2] - kz * E[1]) / wavelength;
00167     pTo6Space[4] += (kz * E[0] - kx * E[2]) / wavelength;
00168     pTo6Space[5] += (kx * E[1] - ky * E[0]) / wavelength;
00169 }
00170
00171 /** Gauss2D construction with */
00172 Gauss2D::Gauss2D(std::array<unrealtype, 3> dis_,
00173                  std::array<unrealtype, 3> axis_,
00174                  unrealtype Amp_, unrealtype phip_, unrealtype w0_,
00175                  unrealtype zr_, unrealtype Ph0_, unrealtype PhA_) {
00176     dis = dis_; /* - center it approaches */
00177     axis = axis_; /* - direction form where it comes */
00178     Amp = Amp_; /* - amplitude */
00179     phip = phip_; /* - polarization rotation from TE-mode */
00180     w0 = w0_; /* - taille */
00181     zr = zr_; /* - Rayleigh length */
00182     Ph0 = Ph0_; /* - beam center */
00183     PhA = PhA_; /* - beam length */
00184     A1 = Amp * cos(phip); // amplitude in z-direction
00185     A2 = Amp * sin(phip); // amplitude on xy-plane
00186     lambda = std::numbers::pi * w0 * w0 / zr; // formula for wavelength
00187 }
00188

```

```

00189 void Gauss2D::addToSpace(sunrealtype x, unrealtype y, unrealtype z,
00190                          unrealtype *pTo6Space) const {
00191     // \f$ \vec{x} = \vec{x}_0 - \vec{dis} \f$ // coordinates minus distance to
00192     // origin
00193     x -= dis[0];
00194     y -= dis[1];
00195     // z -= dis[2];
00196     z = nan("0x12345"); // unused parameter
00197     // \f$ z_g = \vec{x} \cdot \vec{e}_g \f$ projection on propagation axis
00198     const unrealtype zg =
00199         x * axis[0] + y * axis[1]; // + z * axis[2]; // = z - z0 -> propagation
00200         // direction, minus origin
00201     // \f$ r = \sqrt{\vec{x}^2 - z_g^2} \f$ -> pythagoras of radius minus
00202     // projection on prop axis
00203     const unrealtype r = sqrt((x * x + y * y / (*+z*z/) -
00204                                zg * zg)); // radial distance to propagation axis
00205     // \f$ w(z) = w0 \sqrt{1 + (z_g / z_r)^2} \f$
00206     // waist at position z
00207     const unrealtype wz = w0 * sqrt(1 + (zg * zg / zr / zr));
00208     // \f$ g(z) = atan(z_g / z_r) \f$
00209     const unrealtype gz = atan(zg / zr); // Gouy phase
00210     // \f$ R(z) = z_g * (1 + (z_r / z_g)^2) \f$
00211     unrealtype Rz = nan("0x12345"); // beam curvature
00212     if (abs(zg) > 1e-15)
00213         Rz = zg * (1 + (zr * zr / zg / zg));
00214     else
00215         Rz = 1e308;
00216     // wavenumber \f$ k = 2\pi / \lambda \f$
00217     const unrealtype k = 2 * std::numbers::pi / lambda;
00218     // \f$ \Phi_F = kr^2 / (2R(z)) + g(z) - kz_g \f$
00219     const unrealtype PhF =
00220         -k * r * r / (2 * Rz) + gz - k * zg; // to be inserted into cosine
00221     // \f$ G = \sqrt{w0 / wz} \exp\{-(r/w(z))^2\} \exp\{(zg - Ph0)^2 / PhA^2\} \cos(PhF) \f$
00222     // CNode is a diva, no chance to remove the square in the second exponential
00223     // -> h too small
00224     const unrealtype G2D = sqrt(w0 / wz) * exp(-r * r / wz / wz) *
00225         exp(-(zg - Ph0) * (zg - Ph0) / PhA / PhA) *
00226         cos(PhF); // gauss shape
00227     // \f$ c_\alpha = \vec{e}_x \cdot \vec{axis} \f$
00228     // projection components; do like this for CNode convergence -> otherwise
00229     // results in machine error values for non-existent field components if
00230     // axis[0] and axis[1] are given
00231     const unrealtype ca =
00232         axis[0]; // x-component of propagation axis which is given as parameter
00233     // no z-component for 2D propagation
00234     const unrealtype sa = sqrt(1 - ca * ca);
00235     // E-field to space: polarization in xy-plane (A2) is projection of
00236     // z-polarization (A1) on x- and y-directions
00237     pTo6Space[0] += sa * (G2D * A2);
00238     pTo6Space[1] += -ca * (G2D * A2);
00239     pTo6Space[2] += G2D * A1;
00240     // B-field -> negative derivative wrt polarization shift of E-field
00241     pTo6Space[3] += -sa * (G2D * A1);
00242     pTo6Space[4] += ca * (G2D * A1);
00243     pTo6Space[5] += G2D * A2;
00244 }
00245
00246 /** Gauss3D construction with */
00247 Gauss3D::Gauss3D(std::array<unrealtype, 3> dis_,
00248                  std::array<unrealtype, 3> axis_,
00249                  unrealtype Amp_,
00250                  // std::array<unrealtype, 3> pol_,
00251                  unrealtype phip_, unrealtype w0_, unrealtype zr_,
00252                  unrealtype Ph0_, unrealtype PhA_) {
00253     dis = dis_; /** - center it approaches */
00254     axis = axis_; /** - direction from where it comes */
00255     Amp = Amp_; /** - amplitude */
00256     // pol = pol_;
00257     phip = phip_; /** - polarization rotation form TE-mode */
00258     w0 = w0_; /** - taille */
00259     zr = zr_; /** - Rayleigh length */
00260     Ph0 = Ph0_; /** - beam center */
00261     PhA = PhA_; /** - beam length */
00262     lambda = std::numbers::pi * w0 * w0 / zr;
00263     A1 = Amp * cos(phia);
00264     A2 = Amp * sin(phia);
00265 }
00266
00267 /** Gauss3D implementation in space */
00268 void Gauss3D::addToSpace(sunrealtype x, unrealtype y, unrealtype z,
00269                          unrealtype *pTo6Space) const {
00270     x -= dis[0];
00271     y -= dis[1];
00272     z -= dis[2];
00273     const unrealtype zg = x * axis[0] + y * axis[1] + z * axis[2];
00274     const unrealtype r = sqrt((x * x + y * y + z * z) - zg * zg);
00275     const unrealtype wz = w0 * sqrt(1 + (zg * zg / zr / zr));

```

```

00276     const sunrealtype gz = atan(zg / zr);
00277     sunrealtype Rz = nan("0x12345");
00278     if (abs(zg) > 1e-15)
00279         Rz = zg * (1 + (zr * zr / zg / zg));
00280     else
00281         Rz = 1e308;
00282     const sunrealtype k = 2 * std::numbers::pi / lambda;
00283     const sunrealtype PhF = -k * r * r / (2 * Rz) + gz - k * zg;
00284     const sunrealtype G3D = (w0 / wz) * exp(-r * r / wz / wz) *
00285         exp(-(zg - Ph0) * (zg - Ph0) / PhA / PhA) * cos(PhF);
00286     const sunrealtype ca = axis[0];
00287     const sunrealtype sa = sqrt(1 - ca * ca);
00288     pTo6Space[0] += sa * (G3D * A2);
00289     pTo6Space[1] += -ca * (G3D * A2);
00290     pTo6Space[2] += G3D * A1;
00291     pTo6Space[3] += -sa * (G3D * A1);
00292     pTo6Space[4] += ca * (G3D * A1);
00293     pTo6Space[5] += G3D * A2;
00294 }
00295
00296 /** Evaluate lattice point values to zero and then add initial field values */
00297 void ICSetter::eval(sunrealtype x, sunrealtype y, sunrealtype z,
00298     sunrealtype *pTo6Space) {
00299     pTo6Space[0] = 0;
00300     pTo6Space[1] = 0;
00301     pTo6Space[2] = 0;
00302     pTo6Space[3] = 0;
00303     pTo6Space[4] = 0;
00304     pTo6Space[5] = 0;
00305     add(x, y, z, pTo6Space);
00306 }
00307
00308 /** Add all initial field values to the lattice space */
00309 void ICSetter::add(sunrealtype x, sunrealtype y, sunrealtype z,
00310     sunrealtype *pTo6Space) {
00311     for (const auto &wave : planeWaves1D)
00312         wave.addToSpace(x, y, z, pTo6Space);
00313     for (const auto &wave : planeWaves2D)
00314         wave.addToSpace(x, y, z, pTo6Space);
00315     for (const auto &wave : planeWaves3D)
00316         wave.addToSpace(x, y, z, pTo6Space);
00317     for (const auto &wave : gauss1Ds)
00318         wave.addToSpace(x, y, z, pTo6Space);
00319     for (const auto &wave : gauss2Ds)
00320         wave.addToSpace(x, y, z, pTo6Space);
00321     for (const auto &wave : gauss3Ds)
00322         wave.addToSpace(x, y, z, pTo6Space);
00323 }
00324
00325 /** Add plane waves in 1D to their container vector */
00326 void ICSetter::addPlaneWave1D(std::array<sunrealtype, 3> k,
00327     std::array<sunrealtype, 3> p,
00328     std::array<sunrealtype, 3> phi) {
00329     planeWaves1D.emplace_back(PlaneWave1D(k, p, phi));
00330 }
00331
00332 /** Add plane waves in 2D to their container vector */
00333 void ICSetter::addPlaneWave2D(std::array<sunrealtype, 3> k,
00334     std::array<sunrealtype, 3> p,
00335     std::array<sunrealtype, 3> phi) {
00336     planeWaves2D.emplace_back(PlaneWave2D(k, p, phi));
00337 }
00338
00339 /** Add plane waves in 3D to their container vector */
00340 void ICSetter::addPlaneWave3D(std::array<sunrealtype, 3> k,
00341     std::array<sunrealtype, 3> p,
00342     std::array<sunrealtype, 3> phi) {
00343     planeWaves3D.emplace_back(PlaneWave3D(k, p, phi));
00344 }
00345
00346 /** Add Gaussian waves in 1D to their container vector */
00347 void ICSetter::addGauss1D(std::array<sunrealtype, 3> k,
00348     std::array<sunrealtype, 3> p,
00349     std::array<sunrealtype, 3> xo, sunrealtype phig_,
00350     std::array<sunrealtype, 3> phi) {
00351     gauss1Ds.emplace_back(Gauss1D(k, p, xo, phig_, phi));
00352 }
00353
00354 /** Add Gaussian waves in 2D to their container vector */
00355 void ICSetter::addGauss2D(std::array<sunrealtype, 3> dis_,
00356     std::array<sunrealtype, 3> axis_,
00357     sunrealtype Amp_, sunrealtype phip_, sunrealtype w0_,
00358     sunrealtype zr_, sunrealtype Ph0_, sunrealtype PhA_)
00359 {
00360     gauss2Ds.emplace_back(
00361         Gauss2D(dis_, axis_, Amp_, phip_, w0_, zr_, Ph0_, PhA_));
00362 }

```

```

00363
00364 /** Add Gaussian waves in 3D to their container vector */
00365 void ICSetter::addGauss3D(std::array<sunrealtype, 3> dis_,
00366     std::array<sunrealtype, 3> axis_,
00367     sunrealtype Amp_, sunrealtype phip_, sunrealtype w0_,
00368     sunrealtype zr_, sunrealtype Ph0_, sunrealtype PhA_)
00369 {
00370     gauss3Ds.emplace_back(
00371         Gauss3D(dis_, axis_, Amp_, phip_, w0_, zr_, Ph0_, PhA_));
00372 }

```

6.8 src/ICSetters.h File Reference

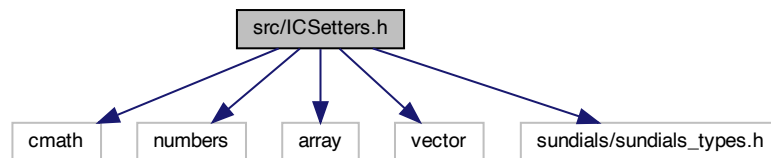
Declaration of the plane wave and Gaussian wave packets.

```

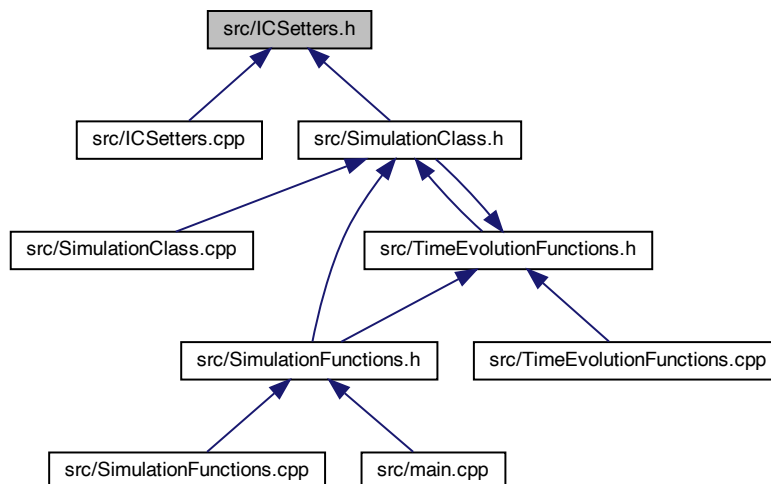
#include <cmath>
#include <numbers>
#include <array>
#include <vector>
#include <sundials/sundials_types.h>

```

Include dependency graph for ICSetters.h:



This graph shows which files directly or indirectly include this file:



Data Structures

- class [PlaneWave](#)
super-class for plane waves
- class [PlaneWave1D](#)
class for plane waves in 1D
- class [PlaneWave2D](#)
class for plane waves in 2D
- class [PlaneWave3D](#)
class for plane waves in 3D
- class [Gauss1D](#)
class for Gaussian pulses in 1D
- class [Gauss2D](#)
class for Gaussian pulses in 2D
- class [Gauss3D](#)
class for Gaussian pulses in 3D
- class [ICSetter](#)
[ICSetter](#) class to initialize wave types with default parameters.

6.8.1 Detailed Description

Declaration of the plane wave and Gaussian wave packets.

Definition in file [ICSetters.h](#).

6.9 ICSetters.h

[Go to the documentation of this file.](#)

```

00001 //////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
00002 /// @file ICSetters.h
00003 /// @brief Declaration of the plane wave and Gaussian wave packets
00004 //////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
00005
00006 #pragma once
00007
00008 // math, constants, vector, and array
00009 #include <cmath>
00010 #include <numbers>
00011 #include <array>
00012 #include <vector>
00013
00014 #include <sundials/sundials_types.h> /* definition of type sunrealtype */
00015
00016 /** @brief super-class for plane waves
00017  *
00018  * They are given in the form  $\vec{E} = \vec{E}_0 \cos(\vec{k} \cdot \vec{x} - \vec{\phi})$ 
00019  *  $\vec{x} = \vec{\phi}$ 
00020  */
00020 class PlaneWave {
00021 protected:
00022     /// wavenumber  $k_x$ 
00023     sunrealtype kx;
00024     /// wavenumber  $k_y$ 
00025     sunrealtype ky;
00026     /// wavenumber  $k_z$ 
00027     sunrealtype kz;
00028     /// polarization & amplitude in x-direction,  $p_x$ 
00029     sunrealtype px;
00030     /// polarization & amplitude in y-direction,  $p_y$ 
00031     sunrealtype py;
00032     /// polarization & amplitude in z-direction,  $p_z$ 
00033     sunrealtype pz;
00034     /// phase shift in x-direction,  $\phi_x$ 
00035     sunrealtype phix;

```

```

00036     /// phase shift in y-direction, \f$ \phi_y \f$
00037     sunrealtype phiy;
00038     /// phase shift in z-direction, \f$ \phi_z \f$
00039     sunrealtype phiz;
00040 };
00041
00042 /** @brief class for plane waves in 1D */
00043 class PlaneWave1D : public PlaneWave {
00044 public:
00045     /// construction with default parameters
00046     PlaneWave1D(std::array<sunrealtype, 3> k = {1, 0, 0},
00047                 std::array<sunrealtype, 3> p = {0, 0, 1},
00048                 std::array<sunrealtype, 3> phi = {0, 0, 0});
00049     /// function for the actual implementation in the lattice
00050     void addToSpace(sunrealtype x, sunrealtype y, sunrealtype z,
00051                     sunrealtype *pTo6Space) const;
00052 };
00053
00054 /** @brief class for plane waves in 2D */
00055 class PlaneWave2D : public PlaneWave {
00056 public:
00057     /// construction with default parameters
00058     PlaneWave2D(std::array<sunrealtype, 3> k = {1, 0, 0},
00059                 std::array<sunrealtype, 3> p = {0, 0, 1},
00060                 std::array<sunrealtype, 3> phi = {0, 0, 0});
00061     /// function for the actual implementation in the lattice
00062     void addToSpace(sunrealtype x, sunrealtype y, sunrealtype z,
00063                     sunrealtype *pTo6Space) const;
00064 };
00065
00066 /** @brief class for plane waves in 3D */
00067 class PlaneWave3D : public PlaneWave {
00068 public:
00069     /// construction with default parameters
00070     PlaneWave3D(std::array<sunrealtype, 3> k = {1, 0, 0},
00071                 std::array<sunrealtype, 3> p = {0, 0, 1},
00072                 std::array<sunrealtype, 3> phi = {0, 0, 0});
00073     /// function for the actual implementation in space
00074     void addToSpace(sunrealtype x, sunrealtype y, sunrealtype z,
00075                     sunrealtype *pTo6Space) const;
00076 };
00077
00078 /** @brief class for Gaussian pulses in 1D
00079  *
00080  * They are given in the form  $\vec{E} = \vec{p} \exp\left(-\frac{(\vec{x} - \vec{x}_0)^2}{\Phi_g^2}\right) \cos(\vec{k} \cdot \vec{x})$ 
00081  */
00082 */
00083 class Gauss1D {
00084 private:
00085     /// wavenumber \f$ k_x \f$
00086     sunrealtype kx;
00087     /// wavenumber \f$ k_y \f$
00088     sunrealtype ky;
00089     /// wavenumber \f$ k_z \f$
00090     sunrealtype kz;
00091     /// polarization & amplitude in x-direction, \f$ p_x \f$
00092     sunrealtype px;
00093     /// polarization & amplitude in y-direction, \f$ p_y \f$
00094     sunrealtype py;
00095     /// polarization & amplitude in z-direction, \f$ p_z \f$
00096     sunrealtype pz;
00097     /// phase shift in x-direction, \f$ \phi_x \f$
00098     sunrealtype phix;
00099     /// phase shift in y-direction, \f$ \phi_y \f$
00100     sunrealtype phiy;
00101     /// phase shift in z-direction, \f$ \phi_z \f$
00102     sunrealtype phiz;
00103     /// center of pulse in x-direction, \f$ x_0 \f$
00104     sunrealtype x0x;
00105     /// center of pulse in y-direction, \f$ y_0 \f$
00106     sunrealtype x0y;
00107     /// center of pulse in z-direction, \f$ z_0 \f$
00108     sunrealtype x0z;
00109     /// pulse width \f$ \Phi_g \f$
00110     sunrealtype phig;
00111
00112 public:
00113     /// construction with default parameters
00114     Gauss1D(std::array<sunrealtype, 3> k = {1, 0, 0},
00115             std::array<sunrealtype, 3> p = {0, 0, 1},
00116             std::array<sunrealtype, 3> xo = {0, 0, 0},
00117             sunrealtype phig_ = 1.0,
00118             std::array<sunrealtype, 3> phi = {0, 0, 0});
00119     /// function for the actual implementation in space
00120     void addToSpace(sunrealtype x, sunrealtype y, sunrealtype z,
00121                     sunrealtype *pTo6Space) const;
00122

```

```

00123 public:
00124 };
00125
00126 /** @brief class for Gaussian pulses in 2D
00127 *
00128 * They are given in the form
00129 *  $\vec{E} = A \vec{\epsilon} \sqrt{\frac{\omega_0}{\omega(z)}} \exp\left(-\frac{(z_g - \Phi_0)^2}{\Phi_A^2}\right)$ 
00130 *  $\exp\left(-\frac{(z_g - \Phi_0)^2}{\Phi_A^2}\right)$ 
00131 *  $\cos\left(\frac{k}{r^2} 2R(z) + g(z) - k z_g\right)$  with
00132 * - propagation direction (subtracted distance to origin)  $z_g$ 
00133 * - radial distance to propagation axis  $r = \sqrt{x^2 - z_g^2}$ 
00134 * -  $k = 2\pi / \lambda$ 
00135 * - waist at position  $z$ ,  $\omega(z) = w_0 \sqrt{1 + (z_g/z_R)^2}$ 
00136 * - Gouy phase  $g(z) = \tan^{-1}(z_g/z_R)$ 
00137 * - beam curvature  $R(z) = z_g / (1 + (z_r/z_g)^2)$ 
00138 * obtained via the chosen parameters */
00139 class Gauss2D {
00140 private:
00141     /// distance maximum to origin
00142     std::array<sunrealtype, 3> dis;
00143     /// normalized propagation axis
00144     std::array<sunrealtype, 3> axis;
00145     /// amplitude  $A$ 
00146     sunrealtype Amp;
00147     /// polarization rotation from TE-mode around propagation direction
00148     /// that determines  $\vec{\epsilon}$  above
00149     sunrealtype phip;
00150     /// taille  $\omega_0$ 
00151     sunrealtype w0;
00152     /// Rayleigh length  $z_R = \pi \omega_0^2 / \lambda$ 
00153     sunrealtype zr;
00154     /// center of beam  $\Phi_0$ 
00155     sunrealtype Ph0;
00156     /// length of beam  $\Phi_A$ 
00157     sunrealtype PhA;
00158     /// amplitude projection on TE-mode
00159     sunrealtype A1;
00160     /// amplitude projection on xy-plane
00161     sunrealtype A2;
00162     /// wavelength  $\lambda$ 
00163     sunrealtype lambda;
00164
00165 public:
00166     /// construction with default parameters
00167     Gauss2D(std::array<sunrealtype, 3> dis_ = {0, 0, 0},
00168             std::array<sunrealtype, 3> axis_ = {1, 0, 0},
00169             sunrealtype Amp_ = 1.0,
00170             sunrealtype phip_ = 0, sunrealtype w0_ = 1e-5,
00171             sunrealtype zr_ = 4e-5,
00172             sunrealtype Ph0_ = 2e-5, sunrealtype PhA_ = 0.45e-5);
00173     /// function for the actual implementation in space
00174     void addToSpace(sunrealtype x, sunrealtype y, sunrealtype z,
00175                    sunrealtype *pTo6Space) const;
00176
00177 public:
00178 };
00179
00180 /** @brief class for Gaussian pulses in 3D
00181 *
00182 * They are given in the form
00183 *  $\vec{E} = A \vec{\epsilon} \sqrt{\frac{\omega_0}{\omega(z)}} \exp\left(-\frac{(z_g - \Phi_0)^2}{\Phi_A^2}\right)$ 
00184 *  $\exp\left(-\frac{(z_g - \Phi_0)^2}{\Phi_A^2}\right)$ 
00185 *  $\cos\left(\frac{k}{r^2} 2R(z) + g(z) - k z_g\right)$  with
00186 * - propagation direction (subtracted distance to origin)  $z_g$ 
00187 * - radial distance to propagation axis  $r = \sqrt{x^2 - z_g^2}$ 
00188 * -  $k = 2\pi / \lambda$ 
00189 * - waist at position  $z$ ,  $\omega(z) = w_0 \sqrt{1 + (z_g/z_R)^2}$ 
00190 * - Gouy phase  $g(z) = \tan^{-1}(z_g/z_R)$ 
00191 * - beam curvature  $R(z) = z_g / (1 + (z_r/z_g)^2)$ 
00192 * obtained via the chosen parameters */
00193 class Gauss3D {
00194 private:
00195     /// distance maximum to origin
00196     std::array<sunrealtype, 3> dis;
00197     /// normalized propagation axis
00198     std::array<sunrealtype, 3> axis;
00199     /// amplitude  $A$ 
00200     sunrealtype Amp;
00201     /// polarization rotation from TE-mode around propagation direction
00202     /// that determines  $\vec{\epsilon}$  above
00203     sunrealtype phip;
00204     /// polarization
00205     std::array<sunrealtype, 3> pol;
00206     /// taille  $\omega_0$ 
00207     sunrealtype w0;
00208     /// Rayleigh length  $z_R = \pi \omega_0^2 / \lambda$ 
00209     sunrealtype zr;

```

```

00210    /// center of beam \f$ \Phi_0 \f$
00211    sunrealtype Ph0;
00212    /// length of beam \f$ \Phi_A \f$
00213    sunrealtype PhA;
00214    /// amplitude projection on TE-mode (z-axis)
00215    sunrealtype A1;
00216    /// amplitude projection on xy-plane
00217    sunrealtype A2;
00218    /// wavelength \f$ \lambda \f$
00219    sunrealtype lambda;
00220
00221 public:
00222    /// construction with default parameters
00223    Gauss3D(std::array<sunrealtype, 3> dis_ = {0, 0, 0},
00224            std::array<sunrealtype, 3> axis_ = {1, 0, 0},
00225            sunrealtype Amp_ = 1.0,
00226            sunrealtype phip_ = 0,
00227            // sunrealtype pol_={0,0,1},
00228            sunrealtype w0_ = 1e-5, sunrealtype zr_ = 4e-5,
00229            sunrealtype Ph0_ = 2e-5, sunrealtype PhA_ = 0.45e-5);
00230    /// function for the actual implementation in space
00231    void addToSpace(sunrealtype x, sunrealtype y, sunrealtype z,
00232                   sunrealtype *pTo6Space) const;
00233
00234 public:
00235 };
00236
00237 /** @brief ICSetter class to initialize wave types with default parameters */
00238 class ICSetter {
00239 private:
00240    /// container vector for plane waves in 1D
00241    std::vector<PlaneWave1D> planeWaves1D;
00242    /// container vector for plane waves in 2D
00243    std::vector<PlaneWave2D> planeWaves2D;
00244    /// container vector for plane waves in 3D
00245    std::vector<PlaneWave3D> planeWaves3D;
00246    /// container vector for Gaussian wave packets in 1D
00247    std::vector<Gauss1D> gauss1Ds;
00248    /// container vector for Gaussian wave packets in 2D
00249    std::vector<Gauss2D> gauss2Ds;
00250    /// container vector for Gaussian wave packets in 3D
00251    std::vector<Gauss3D> gauss3Ds;
00252
00253 public:
00254    /// function to set all coordinates to zero and then 'add' the field values
00255    void eval(sunrealtype x, sunrealtype y, sunrealtype z,
00256             sunrealtype *pTo6Space);
00257    /// function to fill the lattice space with initial field values
00258    // of all field vector containers
00259    void add(sunrealtype x, sunrealtype y, sunrealtype z,
00260            sunrealtype *pTo6Space);
00261    /// function to add plane waves in 1D to their container vector
00262    void addPlaneWave1D(std::array<sunrealtype, 3> k = {1, 0, 0},
00263                       std::array<sunrealtype, 3> p = {0, 0, 1},
00264                       std::array<sunrealtype, 3> phi = {0, 0, 0});
00265    /// function to add plane waves in 2D to their container vector
00266    void addPlaneWave2D(std::array<sunrealtype, 3> k = {1, 0, 0},
00267                       std::array<sunrealtype, 3> p = {0, 0, 1},
00268                       std::array<sunrealtype, 3> phi = {0, 0, 0});
00269    /// function to add plane waves in 3D to their container vector
00270    void addPlaneWave3D(std::array<sunrealtype, 3> k = {1, 0, 0},
00271                       std::array<sunrealtype, 3> p = {0, 0, 1},
00272                       std::array<sunrealtype, 3> phi = {0, 0, 0});
00273    /// function to add Gaussian wave packets in 1D to their container vector
00274    void addGauss1D(std::array<sunrealtype, 3> k = {1, 0, 0},
00275                   std::array<sunrealtype, 3> p = {0, 0, 1},
00276                   std::array<sunrealtype, 3> xo = {0, 0, 0},
00277                   sunrealtype phig_ = 1.0,
00278                   std::array<sunrealtype, 3> phi = {0, 0, 0});
00279    /// function to add Gaussian wave packets in 2D to their container vector
00280    void addGauss2D(std::array<sunrealtype, 3> dis_ = {0, 0, 0},
00281                   std::array<sunrealtype, 3> axis_ = {1, 0, 0},
00282                   sunrealtype Amp_ = 1.0, sunrealtype phip_ = 0,
00283                   sunrealtype w0_ = 1e-5, sunrealtype zr_ = 4e-5,
00284                   sunrealtype Ph0_ = 2e-5, sunrealtype PhA_ = 0.45e-5);
00285    /// function to add Gaussian wave packets in 3D to their container vector
00286    void addGauss3D(std::array<sunrealtype, 3> dis_ = {0, 0, 0},
00287                   std::array<sunrealtype, 3> axis_ = {1, 0, 0},
00288                   sunrealtype Amp_ = 1.0, sunrealtype phip_ = 0,
00289                   sunrealtype w0_ = 1e-5, sunrealtype zr_ = 4e-5,
00290                   sunrealtype Ph0_ = 2e-5, sunrealtype PhA_ = 0.45e-5);
00291 };
00292

```

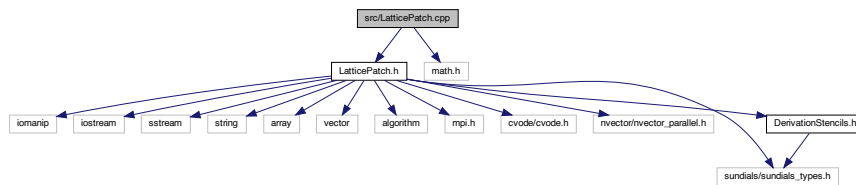
6.10 src/LatticePatch.cpp File Reference

Costruction of the overall envelope lattice and the lattice patches.

```
#include "LatticePatch.h"
```

```
#include <math.h>
```

Include dependency graph for LatticePatch.cpp:



Functions

- int [generatePatchwork](#) (const [Lattice](#) &envelopeLattice, [LatticePatch](#) &patchToMold, const int DLx, const int DLy, const int DLz)
Set up the patchwork.
- void [errorKill](#) (const std::string &errorMessage)
Print a specific error message to stderr.
- int [check_error](#) (int error, const char *funcname, int id)
helper function to check MPI errors
- int [check_retval](#) (void *returnvalue, const char *funcname, int opt, int id)
helper function to check CVec errors

6.10.1 Detailed Description

Costruction of the overall envelope lattice and the lattice patches.

Definition in file [LatticePatch.cpp](#).

6.10.2 Function Documentation

6.10.2.1 check_error()

```
int check_error (
    int error,
    const char * funcname,
    int id )
```

helper function to check MPI errors

Check MPI errors. Error handler must be set.

Definition at line 912 of file [LatticePatch.cpp](#).

```
00912                                     {
00913     int eclass, len;
00914     char errorstring[MPI_MAX_ERROR_STRING];
00915     if( error != MPI_SUCCESS ) {
00916         MPI_Error_class(error,&eclass);
00917         MPI_Error_string(error,errorstring,&len);
00918         std::cerr << "MPI Error(process " << id << ") in " << funcname << " : "
00919             << errorstring << ", from class " << eclass << std::endl;
00920         return 1;
00921     }
00922     return 0;
00923 }
```

6.10.2.2 check_retval()

```
int check_retval (
    void * returnvalue,
    const char * funcname,
    int opt,
    int id )
```

helper function to check CCode errors

Check function return value. Adapted from CCode examples. opt == 0 means SUNDIALS function allocates memory so check if returned NULL pointer opt == 1 means SUNDIALS function returns an integer value so check if retval < 0 opt == 2 means function allocates memory so check if returned NULL pointer

Definition at line 932 of file [LatticePatch.cpp](#).

```
00932                                     {
00933     int *retval = nullptr;
00934
00935     /* Check if SUNDIALS function returned NULL pointer - no memory allocated */
00936     if (opt == 0 && returnvalue == nullptr) {
00937         fprintf(stderr,
00938             "\nSUNDIALS_ERROR(%d): %s() failed - returned NULL pointer\n\n", id,
00939             funcname);
00940         return (1);
00941     }
00942
00943     /* Check if retval < 0 */
00944     else if (opt == 1) {
00945         retval = (int *)returnvalue;
00946         char *flagname = CCodeGetReturnFlagName(*retval);
00947         if (*retval < 0) {
00948             fprintf(stderr, "\nSUNDIALS_ERROR(%d): %s() failed with retval = %d: "
00949                 "%s\n\n",
00950                 id, funcname, *retval, flagname);
00951             return (1);
00952         }
00953     }
00954
00955     /* Check if function returned NULL pointer - no memory allocated */
00956     else if (opt == 2 && returnvalue == nullptr) {
00957         fprintf(stderr,
00958             "\nMEMORY_ERROR(%d): %s() failed - returned NULL pointer\n\n", id,
00959             funcname);
```

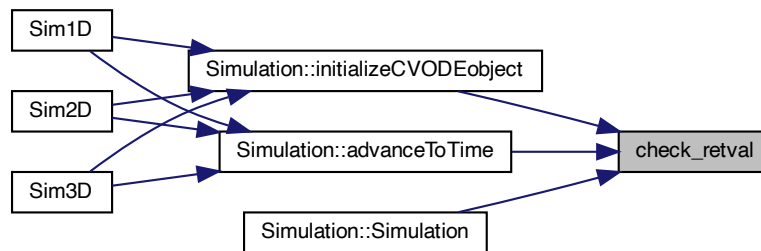
```

00960     return (1);
00961 }
00962
00963     return (0);
00964 }

```

Referenced by [Simulation::advanceToTime\(\)](#), [Simulation::initializeCVODEobject\(\)](#), and [Simulation::Simulation\(\)](#).

Here is the caller graph for this function:



6.10.2.3 errorKill()

```

void errorKill (
    const std::string & errorMessage )

```

Print a specific error message to stderr.

helper function for error messages

Definition at line 900 of file [LatticePatch.cpp](#).

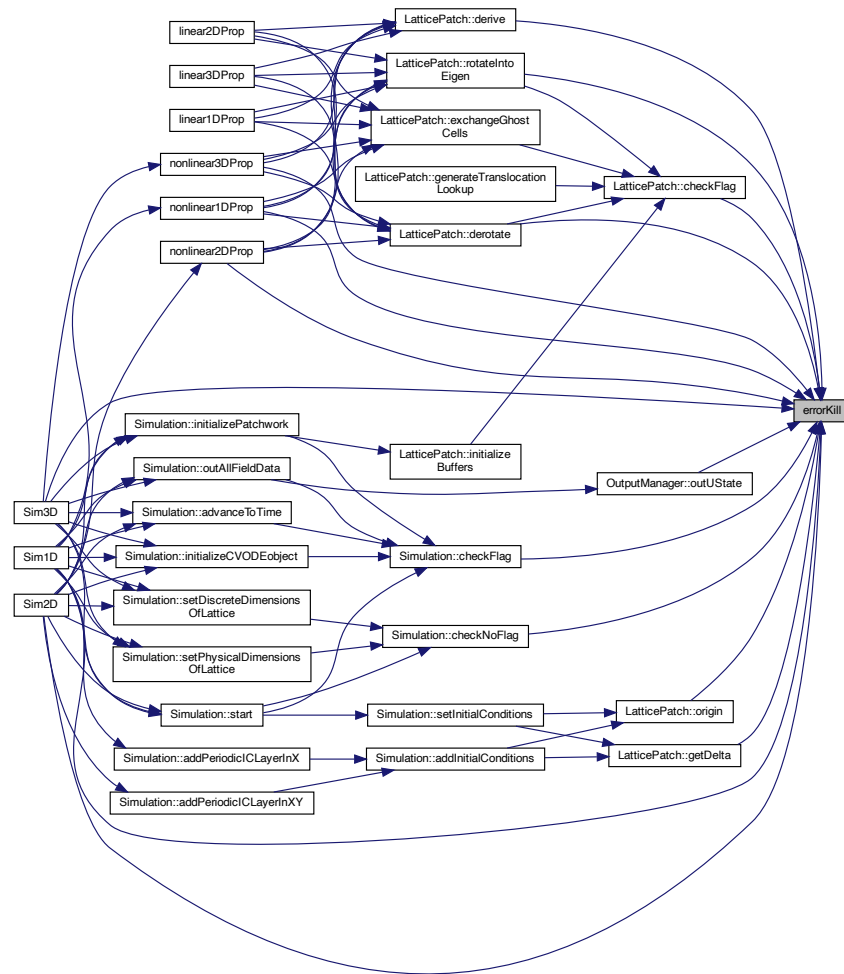
```

00900     {
00901     int my_prc=0;
00902     MPI_Comm_rank (MPI_COMM_WORLD, &my_prc);
00903     if (my_prc==0) {
00904         std::cerr << std::endl << "Error: " << errorMessage
00905         << "\nAborting..." << std::endl;
00906         MPI_Abort (MPI_COMM_WORLD, 1);
00907         return;
00908     }
00909 }

```

Referenced by [LatticePatch::checkFlag\(\)](#), [Simulation::checkFlag\(\)](#), [Simulation::checkNoFlag\(\)](#), [LatticePatch::derive\(\)](#), [LatticePatch::derotate\(\)](#), [LatticePatch::getDelta\(\)](#), [nonlinear1DProp\(\)](#), [nonlinear2DProp\(\)](#), [nonlinear3DProp\(\)](#), [LatticePatch::origin\(\)](#), [OutputManager::outUState\(\)](#), [LatticePatch::rotateIntoEigen\(\)](#), [Sim1D\(\)](#), [Sim2D\(\)](#), and [Sim3D\(\)](#).

Here is the caller graph for this function:



6.10.2.4 generatePatchwork()

```
int generatePatchwork (
    const Lattice & envelopeLattice,
    LatticePatch & patchToMold,
    const int DLx,
    const int DLy,
    const int DLz )
```

Set up the patchwork.

friend function for creating the patchwork slicing of the overall lattice

Definition at line 109 of file [LatticePatch.cpp](#).

```
00111 {
00112     // Retrieve the ghost layer depth
00113     const int gLW = envelopeLattice.get_ghostLayerWidth();
00114     // Retrieve the data point dimension
```



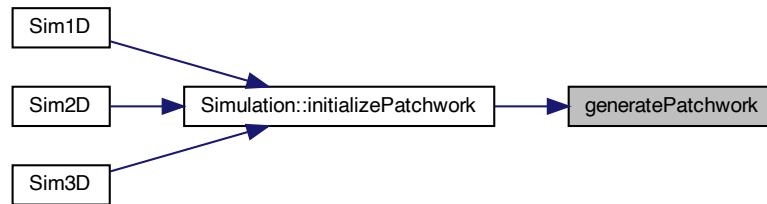
```

00115     const int dPD = envelopeLattice.get_dataPointDimension();
00116     // MPI process/patch
00117     const int my_prc = envelopeLattice.my_prc;
00118     // Determine thicknes of the slice
00119     const sunindextype tot_NOXP = envelopeLattice.get_tot_nx();
00120     const sunindextype tot_NOYP = envelopeLattice.get_tot_ny();
00121     const sunindextype tot_NOZP = envelopeLattice.get_tot_nz();
00122     // position of the patch in the lattice of patches -> process associated to
00123     // position
00124     const sunindextype LIx = my_prc % DLx;
00125     const sunindextype LIy = (my_prc / DLx) % DLy;
00126     const sunindextype LIz = (my_prc / DLx) / DLy;
00127     // Determine the number of points in the patch and first absolute points in
00128     // each dimension
00129     const sunindextype local_NOXP = tot_NOXP / DLx;
00130     const sunindextype local_NOYP = tot_NOYP / DLy;
00131     const sunindextype local_NOZP = tot_NOZP / DLz;
00132     // absolute positions of the first point in each dimension
00133     const sunindextype firstXPoint = local_NOXP * LIx;
00134     const sunindextype firstYPoint = local_NOYP * LIy;
00135     const sunindextype firstZPoint = local_NOZP * LIz;
00136     // total number of points in the patch
00137     const sunindextype local_NODP = dPD * local_NOXP * local_NOYP * local_NOZP;
00138
00139     // Set patch up with above derived quantities
00140     patchToMold.dx = envelopeLattice.get_dx();
00141     patchToMold.dy = envelopeLattice.get_dy();
00142     patchToMold.dz = envelopeLattice.get_dz();
00143     patchToMold.x0 = firstXPoint * patchToMold.dx;
00144     patchToMold.y0 = firstYPoint * patchToMold.dy;
00145     patchToMold.z0 = firstZPoint * patchToMold.dz;
00146     patchToMold.LIx = LIx;
00147     patchToMold.LIy = LIy;
00148     patchToMold.LIz = LIz;
00149     patchToMold.nx = local_NOXP;
00150     patchToMold.ny = local_NOYP;
00151     patchToMold.nz = local_NOZP;
00152     patchToMold.lx = patchToMold.nx * patchToMold.dx;
00153     patchToMold.ly = patchToMold.ny * patchToMold.dy;
00154     patchToMold.lz = patchToMold.nz * patchToMold.dz;
00155     /* Create and allocate memory for parallel vectors with defined local and
00156     * global lenghts *
00157     * (-> CNode problem sizes Nlocal and N)
00158     * for field data and temporal derivatives and set extra pointers to them */
00159     patchToMold.u =
00160         NVNew_Parallel(envelopeLattice.comm, local_NODP,
00161             envelopeLattice.get_tot_nodp(), envelopeLattice.sunctx);
00162     patchToMold.uData = NV_DATA_P(patchToMold.u);
00163     patchToMold.du =
00164         NVNew_Parallel(envelopeLattice.comm, local_NODP,
00165             envelopeLattice.get_tot_nodp(), envelopeLattice.sunctx);
00166     patchToMold.duData = NV_DATA_P(patchToMold.du);
00167     // Allocate space for auxiliary uAux so that the lattice and all possible
00168     // directions of ghost Layers fit
00169     const sunindextype s1 = patchToMold.nx, s2 = patchToMold.ny,
00170         s3 = patchToMold.nz;
00171     const sunindextype s_min = std::min(s1, std::min(s2, s3));
00172     patchToMold.uAux.resize(s1 * s2 * s3 / s_min * (s_min + 2 * gLW) * dPD);
00173     patchToMold.uAuxData = &patchToMold.uAux[0];
00174     patchToMold.envelopeLattice = &envelopeLattice;
00175     // Set patch "name" to process number -> only for debugging
00176     // patchToMold.ID=my_prc;
00177     // set flag
00178     patchToMold.statusFlags = FLatticePatchSetUp;
00179     patchToMold.generateTranslocationLookup();
00180     return 0;
00181 }

```

Referenced by [Simulation::initializePatchwork\(\)](#).

Here is the caller graph for this function:



6.11 LatticePatch.cpp

[Go to the documentation of this file.](#)

```

00001 ///////////////////////////////////////////////////////////////////
00002 /// @file LatticePatch.cpp
00003 /// @brief Construction of the overall envelope lattice and the lattice patches
00004 ///////////////////////////////////////////////////////////////////
00005
00006 #include "LatticePatch.h"
00007
00008 #include <math.h>
00009
00010 ///////////////////////////////////////////////////////////////////
00011 /// Implementation of Lattice component functions ///
00012 ///////////////////////////////////////////////////////////////////
00013
00014 /// Initialize the cartesian communicator
00015 void Lattice::initializeCommunicator(const int Nx, const int Ny,
00016     const int Nz, const bool per) {
00017     const int dims[3] = {Nz, Ny, Nx};
00018     const int periods[3] = {static_cast<int>(per), static_cast<int>(per),
00019         static_cast<int>(per)};
00020     // Create the cartesian communicator for MPI_COMM_WORLD
00021     MPI_Cart_create(MPI_COMM_WORLD, 3, dims, periods, 1, &comm);
00022     // Set MPI variables of the lattice
00023     MPI_Comm_size(comm, &(n_prc));
00024     MPI_Comm_rank(comm, &(my_prc));
00025     // Associate name to the communicator to identify it -> for debugging and
00026     // nicer error messages
00027     constexpr char lattice_comm_name[] = "Lattice";
00028     MPI_Comm_set_name(comm, lattice_comm_name);
00029
00030     // Test if process naming is the same for both communicators
00031     /*
00032     int myPrc;
00033     MPI_Comm_rank(MPI_COMM_WORLD, &myPrc);
00034     cout<<"\r"<<my_prc<<"\t"<<myPrc<<std::endl;
00035     */
00036 }
00037
00038 /// Construct the lattice and set the stencil order
00039 Lattice::Lattice(const int StO) : stencilOrder(StO),
00040     ghostLayerWidth(StO/2+1) {
00041     statusFlags = 0;
00042 }
00043
00044 /// Set the number of points in each dimension of the lattice
00045 void Lattice::setDiscreteDimensions(const sunindextype _nx,
00046     const sunindextype _ny, const sunindextype _nz) {
00047     // copy the given data for number of points
00048     tot_nx = _nx;
00049     tot_ny = _ny;
00050     tot_nz = _nz;
00051     // compute the resulting number of points and datapoints
00052     tot_noP = tot_nx * tot_ny * tot_nz;
00053     tot_noDP = dataPointDimension * tot_noP;
00054     // compute the new Delta, the physical resolution
00055     dx = tot_lx / tot_nx;
00056     dy = tot_ly / tot_ny;

```

```

00057     dz = tot_lz / tot_nz;
00058 }
00059
00060 /// Set the physical size of the lattice
00061 void Lattice::setPhysicalDimensions(const sunrealtype _lx,
00062                                     const sunrealtype _ly, const sunrealtype _lz) {
00063     tot_lx = _lx;
00064     tot_ly = _ly;
00065     tot_lz = _lz;
00066     // calculate physical distance between points
00067     dx = tot_lx / tot_nx;
00068     dy = tot_ly / tot_ny;
00069     dz = tot_lz / tot_nz;
00070     statusFlags |= FLatticeDimensionSet;
00071 }
00072
00073 ///////////////////////////////////////////////////////////////////
00074 /// Implementation of LatticePatch component functions ///
00075 ///////////////////////////////////////////////////////////////////
00076
00077 /// Construct the lattice patch
00078 LatticePatch::LatticePatch() {
00079     // set default origin coordinates to (0,0,0)
00080     x0 = y0 = z0 = 0;
00081     // set default position in Lattice-Patchwork to (0,0,0)
00082     Llx = Lly = Llz = 0;
00083     // set default physical length for lattice patch to (0,0,0)
00084     lx = ly = lz = 0;
00085     // set default discrete length for lattice patch to (0,1,1)
00086     /* This is done in this manner as even in 1D simulations require a 1 point
00087      * width */
00088     nx = 0;
00089     ny = nz = 1;
00090
00091     // u is not initialized as it wouldn't make any sense before the dimensions
00092     // are set idem for the enveloping lattice
00093
00094     // set default statusFlags to non set
00095     statusFlags = 0;
00096 }
00097
00098 /// Destruct the patch and thereby destroy the NVectors
00099 LatticePatch::~LatticePatch() {
00100     // Deallocate memory for solution vector
00101     if (statusFlags & FLatticePatchSetUp) {
00102         // Destroy data vectors
00103         N_VDestroy_Parallel(u);
00104         N_VDestroy_Parallel(du);
00105     }
00106 }
00107
00108 /// Set up the patchwork
00109 int generatePatchwork(const Lattice &envelopeLattice,
00110                     LatticePatch &patchToMold,
00111                     const int DLx, const int DLy, const int DLz) {
00112     // Retrieve the ghost layer depth
00113     const int gLW = envelopeLattice.get_ghostLayerWidth();
00114     // Retrieve the data point dimension
00115     const int dPD = envelopeLattice.get_dataPointDimension();
00116     // MPI process/patch
00117     const int my_prc = envelopeLattice.my_prc;
00118     // Determine thicknes of the slice
00119     const sunindextype tot_NOXP = envelopeLattice.get_tot_nx();
00120     const sunindextype tot_NOYP = envelopeLattice.get_tot_ny();
00121     const sunindextype tot_NOZP = envelopeLattice.get_tot_nz();
00122     // position of the patch in the lattice of patches -> process associated to
00123     // position
00124     const sunindextype Llx = my_prc % DLx;
00125     const sunindextype Lly = (my_prc / DLx) % DLy;
00126     const sunindextype Llz = (my_prc / DLx) / DLy;
00127     // Determine the number of points in the patch and first absolute points in
00128     // each dimension
00129     const sunindextype local_NOXP = tot_NOXP / DLx;
00130     const sunindextype local_NOYP = tot_NOYP / DLy;
00131     const sunindextype local_NOZP = tot_NOZP / DLz;
00132     // absolute positions of the first point in each dimension
00133     const sunindextype firstXPoint = local_NOXP * Llx;
00134     const sunindextype firstYPoint = local_NOYP * Lly;
00135     const sunindextype firstZPoint = local_NOZP * Llz;
00136     // total number of points in the patch
00137     const sunindextype local_NODP = dPD * local_NOXP * local_NOYP * local_NOZP;
00138
00139     // Set patch up with above derived quantities
00140     patchToMold.dx = envelopeLattice.get_dx();
00141     patchToMold.dy = envelopeLattice.get_dy();
00142     patchToMold.dz = envelopeLattice.get_dz();
00143     patchToMold.x0 = firstXPoint * patchToMold.dx;

```

```

00144 patchToMold.y0 = firstYPoint * patchToMold.dy;
00145 patchToMold.z0 = firstZPoint * patchToMold.dz;
00146 patchToMold.LIx = LIx;
00147 patchToMold.LIy = LIy;
00148 patchToMold.LIz = LIz;
00149 patchToMold.nx = local_NOXP;
00150 patchToMold.ny = local_NOYP;
00151 patchToMold.nz = local_NOZP;
00152 patchToMold.lx = patchToMold.nx * patchToMold.dx;
00153 patchToMold.ly = patchToMold.ny * patchToMold.dy;
00154 patchToMold.lz = patchToMold.nz * patchToMold.dz;
00155 /* Create and allocate memory for parallel vectors with defined local and
00156    * global lenghts *
00157    * (-> CNode problem sizes Nlocal and N)
00158    * for field data and temporal derivatives and set extra pointers to them */
00159 patchToMold.u =
00160     NVNew_Parallel(envelopeLattice.comm, local_NODP,
00161                   envelopeLattice.get_tot_noDP(), envelopeLattice.sunctx);
00162 patchToMold.uData = NV_DATA_P(patchToMold.u);
00163 patchToMold.du =
00164     NVNew_Parallel(envelopeLattice.comm, local_NODP,
00165                   envelopeLattice.get_tot_noDP(), envelopeLattice.sunctx);
00166 patchToMold.duData = NV_DATA_P(patchToMold.du);
00167 // Allocate space for auxiliary uAux so that the lattice and all possible
00168 // directions of ghost Layers fit
00169 const sunindextype s1 = patchToMold.nx, s2 = patchToMold.ny,
00170 s3 = patchToMold.nz;
00171 const sunindextype s_min = std::min(s1, std::min(s2, s3));
00172 patchToMold.uAux.resize(s1 * s2 * s3 / s_min * (s_min + 2 * gLW) * dPD);
00173 patchToMold.uAuxData = &patchToMold.uAux[0];
00174 patchToMold.envelopeLattice = &envelopeLattice;
00175 // Set patch "name" to process number -> only for debugging
00176 // patchToMold.ID=my_prc;
00177 // set flag
00178 patchToMold.statusFlags = FLatticePatchSetUp;
00179 patchToMold.generateTranslocationLookup();
00180 return 0;
00181 }
00182
00183 /// Return the discrete size of the patch: number of lattice patch points in
00184 /// specified dimension
00185 sunindextype LatticePatch::discreteSize(int dir) const {
00186     switch (dir) {
00187         case 0:
00188             return nx * ny * nz;
00189         case 1:
00190             return nx;
00191         case 2:
00192             return ny;
00193         case 3:
00194             return nz;
00195         // case 4: return uAux.size(); // for debugging
00196         default:
00197             return -1;
00198     }
00199 }
00200
00201 /// Return the physical origin of the patch in a dimension
00202 sunrealtype LatticePatch::origin(const int dir) const {
00203     switch (dir) {
00204         case 1:
00205             return x0;
00206         case 2:
00207             return y0;
00208         case 3:
00209             return z0;
00210         default:
00211             errorKill("LatticePatch::origin function called with wrong dir parameter");
00212             return -1;
00213     }
00214 }
00215
00216 /// Return the distance between points in the patch in a dimension
00217 sunrealtype LatticePatch::getDelta(const int dir) const {
00218     switch (dir) {
00219         case 1:
00220             return dx;
00221         case 2:
00222             return dy;
00223         case 3:
00224             return dz;
00225         default:
00226             errorKill(
00227                 "LatticePatch::getDelta function called with wrong dir parameter");
00228             return -1;
00229     }
00230 }

```

```

00231
00232 /** In order to avoid cache misses:
00233  * create vectors to translate u vector into space coordinates and vice versa
00234  * and same for left and right ghost layers to space */
00235 void LatticePatch::generateTranslocationLookup() {
00236     // Check that the lattice has been set up
00237     checkFlag(FLatticeDimensionSet);
00238     // lenghts for auxilliary layers, including ghost layers
00239     const int gLW = envelopeLattice->get_ghostLayerWidth();
00240     const sunindextype mx = nx + 2 * gLW;
00241     const sunindextype my = ny + 2 * gLW;
00242     const sunindextype mz = nz + 2 * gLW;
00243     // sizes for lookup vectors
00244     const sunindextype totalNP = nx * ny * nz;
00245     const sunindextype haloXSize = mx * ny * nz;
00246     const sunindextype haloYSize = nx * my * nz;
00247     const sunindextype haloZSize = nx * ny * mz;
00248     // generate u->uAux
00249     uTox.resize(totalNP);
00250     uToy.resize(totalNP);
00251     uToz.resize(totalNP);
00252     // generate uAux->u with length including halo
00253     xTou.resize(haloXSize);
00254     yTou.resize(haloYSize);
00255     zTou.resize(haloZSize);
00256     // same for ghost layer lookup tables
00257     const sunindextype ghostXSize = gLW * ny * nz;
00258     const sunindextype ghostYSize = gLW * nx * nz;
00259     const sunindextype ghostZSize = gLW * nx * ny;
00260     lgcTox.resize(ghostXSize);
00261     rgcTox.resize(ghostXSize);
00262     lgcToy.resize(ghostYSize);
00263     rgcToy.resize(ghostYSize);
00264     lgcToz.resize(ghostZSize);
00265     rgcToz.resize(ghostZSize);
00266     // variables for cartesian position in the 3D discrete lattice
00267     sunindextype px = 0, py = 0, pz = 0;
00268     // Fill the lookup tables
00269     #pragma omp parallel default(none) \
00270     private(px, py, pz) \
00271     shared(uTox, uToy, uToz, xTou, yTou, zTou, \
00272           nx, ny, mx, my, mz, gLW, totalNP, \
00273           lgcTox, rgcTox, lgcToy, rgcToy, lgcToz, rgcToz, \
00274           ghostXSize, ghostYSize, ghostZSize)
00275     {
00276     #pragma omp for simd schedule(static)
00277     for (sunindextype i = 0; i < totalNP; i++) { // loop over the patch
00278         // calculate cartesian coordinates
00279         px = i % nx;
00280         py = (i / nx) % ny;
00281         pz = (i / nx) / ny;
00282         // fill lookups extended by halos (useful for y and z direction)
00283         uTox[i] = (px + gLW) + py * mx +
00284                 pz * mx * ny; // unroll (de-flatten) cartesian dimension
00285         xTou[px + py * mx + pz * mx * ny] =
00286             i; // match cartesian point to u location
00287         uToy[i] = (py + gLW) + pz * my + px * my * nz;
00288         yTou[py + pz * my + px * my * nz] = i;
00289         uToz[i] = (pz + gLW) + px * mz + py * mz * nx;
00290         zTou[pz + px * mz + py * mz * nx] = i;
00291     }
00292     #pragma omp for simd schedule(static)
00293     for (sunindextype i = 0; i < ghostXSize; i++) {
00294         px = i % gLW;
00295         py = (i / gLW) % ny;
00296         pz = (i / gLW) / ny;
00297         lgcTox[i] = px + py * mx + pz * mx * ny;
00298         rgcTox[i] = px + nx + gLW + py * mx + pz * mx * ny;
00299     }
00300     #pragma omp for simd schedule(static)
00301     for (sunindextype i = 0; i < ghostYSize; i++) {
00302         px = i % nx;
00303         py = (i / nx) % gLW;
00304         pz = (i / nx) / gLW;
00305         lgcToy[i] = py + pz * my + px * my * nz;
00306         rgcToy[i] = py + ny + gLW + pz * my + px * my * nz;
00307     }
00308     #pragma omp for simd schedule(static)
00309     for (sunindextype i = 0; i < ghostZSize; i++) {
00310         px = i % nx;
00311         py = (i / nx) % ny;
00312         pz = (i / nx) / ny;
00313         lgcToz[i] = pz + px * mz + py * mz * nx;
00314         rgcToz[i] = pz + nz + gLW + px * mz + py * mz * nx;
00315     }
00316     }
00317     statusFlags |= TranslocationLookupSetUp;

```

```

00318 }
00319
00320 /** Rotate into eigenraum along R matrices of paper using the rotation
00321 * methods;
00322 * uAuxData gets the rotated left-halo-, inner-patch-, right-halo-data */
00323 void LatticePatch::rotateIntoEigen(const int dir) {
00324     // Check that the lattice, ghost layers as well as the translocation lookups
00325     // have been set up;
00326     checkFlag(FLatticePatchSetUp);
00327     checkFlag(TranslocationLookupSetUp);
00328     checkFlag(GhostLayersInitialized); // this check is only after call to
00329                                         // exchange ghost cells
00330     switch (dir) {
00331     case 1:
00332         rotateToX(uAuxData, gCLData, lgcTox);
00333         rotateToX(uAuxData, uData, uTox);
00334         rotateToX(uAuxData, gCRData, rgcTox);
00335         break;
00336     case 2:
00337         rotateToY(uAuxData, gCLData, lgcToy);
00338         rotateToY(uAuxData, uData, uToy);
00339         rotateToY(uAuxData, gCRData, rgcToy);
00340         break;
00341     case 3:
00342         rotateToZ(uAuxData, gCLData, lgcToz);
00343         rotateToZ(uAuxData, uData, uToz);
00344         rotateToZ(uAuxData, gCRData, rgcToz);
00345         break;
00346     default:
00347         errorKill("Tried to rotate into the wrong direction");
00348         break;
00349     }
00350 }
00351
00352 /// Rotate halo and inner-patch data vectors with rotation matrix Rx into
00353 /// eigenspace of Z matrix and write to auxiliary vector
00354 inline void LatticePatch::rotateToX(sunrealtype *outArray,
00355                                     const sunrealtype *inArray,
00356                                     const std::vector<sunindextype> &lookup) {
00357     sunindextype ii = 0, target = 0;
00358     const sunindextype size = lookup.size();
00359     const int dPD = envelopeLattice->get_dataPointDimension();
00360     #pragma omp parallel for simd \
00361     private(target, ii) \
00362     shared(lookup, outArray, inArray, size, dPD) \
00363     schedule(static)
00364     for (sunindextype i = 0; i < size; i++) {
00365         // get correct u-vector and spatial indices along previously defined lookup
00366         // tables
00367         target = dPD * lookup[i];
00368         ii = dPD * i;
00369         outArray[target + 0] = -inArray[1 + ii] + inArray[5 + ii];
00370         outArray[target + 1] = inArray[2 + ii] + inArray[4 + ii];
00371         outArray[target + 2] = inArray[1 + ii] + inArray[5 + ii];
00372         outArray[target + 3] = -inArray[2 + ii] + inArray[4 + ii];
00373         outArray[target + 4] = inArray[3 + ii];
00374         outArray[target + 5] = inArray[ii];
00375     }
00376 }
00377
00378 /// Rotate halo and inner-patch data vectors with rotation matrix Ry into
00379 /// eigenspace of Z matrix and write to auxiliary vector
00380 inline void LatticePatch::rotateToY(sunrealtype *outArray,
00381                                     const sunrealtype *inArray,
00382                                     const std::vector<sunindextype> &lookup) {
00383     sunindextype ii = 0, target = 0;
00384     const int dPD = envelopeLattice->get_dataPointDimension();
00385     const sunindextype size = lookup.size();
00386     #pragma omp parallel for simd \
00387     private(target, ii) \
00388     shared(lookup, outArray, inArray, size, dPD) \
00389     schedule(static)
00390     for (sunindextype i = 0; i < size; i++) {
00391         target = dPD * lookup[i];
00392         ii = dPD * i;
00393         outArray[target + 0] = inArray[ii] + inArray[5 + ii];
00394         outArray[target + 1] = -inArray[2 + ii] + inArray[3 + ii];
00395         outArray[target + 2] = -inArray[ii] + inArray[5 + ii];
00396         outArray[target + 3] = inArray[2 + ii] + inArray[3 + ii];
00397         outArray[target + 4] = inArray[4 + ii];
00398         outArray[target + 5] = inArray[1 + ii];
00399     }
00400 }
00401
00402 /// Rotate halo and inner-patch data vectors with rotation matrix Rz into
00403 /// eigenspace of Z matrix and write to auxiliary vector
00404 inline void LatticePatch::rotateToZ(sunrealtype *outArray,

```

```

00405                                     const sunrealttype *inArray,
00406                                     const std::vector<sunindextype> &lookup) {
00407     sunindextype ii = 0, target = 0;
00408     const sunindextype size = lookup.size();
00409     const int dPD = envelopeLattice->get_dataPointDimension();
00410     #pragma omp parallel for simd \
00411     private(target, ii) \
00412     shared(lookup, outArray, inArray, size, dPD) \
00413     schedule(static)
00414     for (sunindextype i = 0; i < size; i++) {
00415         target = dPD * lookup[i];
00416         ii = dPD * i;
00417         outArray[target + 0] = -inArray[ii] + inArray[4 + ii];
00418         outArray[target + 1] = inArray[1 + ii] + inArray[3 + ii];
00419         outArray[target + 2] = inArray[ii] + inArray[4 + ii];
00420         outArray[target + 3] = -inArray[1 + ii] + inArray[3 + ii];
00421         outArray[target + 4] = inArray[5 + ii];
00422         outArray[target + 5] = inArray[2 + ii];
00423     }
00424 }
00425
00426 /// Derotate uAux with transposed rotation matrices and write to derivative
00427 /// buffer -- normalization is done here by the factor 1/2
00428 void LatticePatch::derotate(int dir, sunrealttype *buffOut) {
00429     // Check that the lattice as well as the translocation lookups have been set
00430     // up;
00431     checkFlag(FLatticePatchSetUp);
00432     checkFlag(TranslocationLookupSetUp);
00433     const int dPD = envelopeLattice->get_dataPointDimension();
00434     const int gLW = envelopeLattice->get_ghostLayerWidth();
00435     const sunindextype totalNP = discreteSize();
00436     sunindextype ii = 0, target = 0;
00437     switch (dir) {
00438     case 1:
00439         #pragma omp parallel for simd \
00440         private(ii, target) \
00441         shared(dPD, gLW, totalNP, uTox, uAux, buffOut) \
00442         schedule(static)
00443         for (sunindextype i = 0; i < totalNP; i++) {
00444             // get correct indices in u and rotation space
00445             target = dPD * i;
00446             ii = dPD * (uTox[i] - gLW);
00447             buffOut[target + 0] = uAux[5 + ii];
00448             buffOut[target + 1] = (-uAux[ii] + uAux[2 + ii]) / 2.;
00449             buffOut[target + 2] = (uAux[1 + ii] - uAux[3 + ii]) / 2.;
00450             buffOut[target + 3] = uAux[4 + ii];
00451             buffOut[target + 4] = (uAux[1 + ii] + uAux[3 + ii]) / 2.;
00452             buffOut[target + 5] = (uAux[ii] + uAux[2 + ii]) / 2.;
00453         }
00454         break;
00455     case 2:
00456         #pragma omp parallel for simd \
00457         private(ii, target) \
00458         shared(dPD, gLW, totalNP, uTox, uAux, buffOut) \
00459         schedule(static)
00460         for (sunindextype i = 0; i < totalNP; i++) {
00461             target = dPD * i;
00462             ii = dPD * (uToy[i] - gLW);
00463             buffOut[target + 0] = (uAux[ii] - uAux[2 + ii]) / 2.;
00464             buffOut[target + 1] = uAux[5 + ii];
00465             buffOut[target + 2] = (-uAux[1 + ii] + uAux[3 + ii]) / 2.;
00466             buffOut[target + 3] = (uAux[1 + ii] + uAux[3 + ii]) / 2.;
00467             buffOut[target + 4] = uAux[4 + ii];
00468             buffOut[target + 5] = (uAux[ii] + uAux[2 + ii]) / 2.;
00469         }
00470         break;
00471     case 3:
00472         #pragma omp parallel for simd \
00473         private(ii, target) \
00474         shared(dPD, gLW, totalNP, uTox, uAux, buffOut) \
00475         schedule(static)
00476         for (sunindextype i = 0; i < totalNP; i++) {
00477             target = dPD * i;
00478             ii = dPD * (uToz[i] - gLW);
00479             buffOut[target + 0] = (-uAux[ii] + uAux[2 + ii]) / 2.;
00480             buffOut[target + 1] = (uAux[1 + ii] - uAux[3 + ii]) / 2.;
00481             buffOut[target + 2] = uAux[5 + ii];
00482             buffOut[target + 3] = (uAux[1 + ii] + uAux[3 + ii]) / 2.;
00483             buffOut[target + 4] = (uAux[ii] + uAux[2 + ii]) / 2.;
00484             buffOut[target + 5] = uAux[4 + ii];
00485         }
00486         break;
00487     default:
00488         errorKill("Tried to derotate from the wrong direction");
00489         break;
00490     }
00491 }

```

```

00492
00493 /// Create buffers to save derivative values, optimizing computational load
00494 void LatticePatch::initializeBuffers() {
00495     // Check that the lattice has been set up
00496     checkFlag(FLatticeDimensionSet);
00497     const int dPD = envelopeLattice->get_dataPointDimension();
00498     buffX.resize(nx * ny * nz * dPD);
00499     buffY.resize(nx * ny * nz * dPD);
00500     buffZ.resize(nx * ny * nz * dPD);
00501     // Set pointers used for propagation functions
00502     buffData[0] = &buffX[0];
00503     buffData[1] = &buffY[0];
00504     buffData[2] = &buffZ[0];
00505     statusFlags |= BuffersInitialized;
00506 }
00507
00508 /// Perform the ghost cell exchange in a specified direction
00509 void LatticePatch::exchangeGhostCells(const int dir) {
00510     // Check that the lattice has been set up
00511     checkFlag(FLatticeDimensionSet);
00512     checkFlag(FLatticePatchSetUp);
00513     // Variables to per dimension calculate the halo indices, and distance to
00514     // other side halo boundary
00515     int mx = 1, my = 1, mz = 1, distToRight = 1;
00516     const int gLW = envelopeLattice->get_ghostLayerWidth();
00517     // In the chosen direction m is set to ghost layer width while the others
00518     // remain to form the plane
00519     switch (dir) {
00520     case 1:
00521         mx = gLW;
00522         my = ny;
00523         mz = nz;
00524         distToRight = (nx - gLW);
00525         break;
00526     case 2:
00527         mx = nx;
00528         my = gLW;
00529         mz = nz;
00530         distToRight = nx * (ny - gLW);
00531         break;
00532     case 3:
00533         mx = nx;
00534         my = ny;
00535         mz = gLW;
00536         distToRight = nx * ny * (nz - gLW);
00537         break;
00538     }
00539     // total number of exchanged points
00540     const int dPD = envelopeLattice->get_dataPointDimension();
00541     const sunindextype exchangeSize = mx * my * mz * dPD;
00542     // provide size of the halos for ghost cells
00543     ghostCellLeft.resize(exchangeSize);
00544     ghostCellRight.resize(ghostCellLeft.size());
00545     ghostCellLeftToSend.resize(ghostCellLeft.size());
00546     ghostCellRightToSend.resize(ghostCellLeft.size());
00547     gCLData = &ghostCellLeft[0];
00548     gCRData = &ghostCellRight[0];
00549     statusFlags |= GhostLayersInitialized;
00550
00551     // Initialize running index li for the halo buffers, and index ui of uData for
00552     // data transfer
00553     sunindextype li = 0, ui = 0;
00554     // Fill the halo buffers
00555     #pragma omp parallel for default(none) \
00556     private(ui) firstprivate(li) \
00557     shared(nx, ny, mx, my, mz, dPD, distToRight, uData, \
00558     ghostCellLeftToSend, ghostCellRightToSend)
00559     for (sunindextype iz = 0; iz < mz; iz++) {
00560         for (sunindextype iy = 0; iy < my; iy++) {
00561             // uData vector start index of halo data to be transferred
00562             // with each z-step add the whole xy-plane and with y-step the x-range ->
00563             // iterate all x-ranges
00564             ui = (iz * nx * ny + iy * nx) * dPD;
00565             // copy left halo data from uData to buffer, transfer size is given by
00566             // x-length (not x-range)
00567             std::copy(&uData[ui], &uData[ui + mx * dPD], &ghostCellLeftToSend[li]);
00568             ui += distToRight * dPD;
00569             std::copy(&uData[ui], &uData[ui + mx * dPD], &ghostCellRightToSend[li]);
00570
00571             // increase halo index by transferred items per y-iteration step
00572             // (x-length)
00573             li += mx * dPD;
00574         }
00575     }
00576
00577     /* Send and receive the data to and from neighboring latticePatches */
00578     // Adjust direction to cartesian communicator

```



```

00579     int dim = 2; // default for dir==1
00580     if (dir == 2) {
00581         dim = 1;
00582     } else if (dir == 3) {
00583         dim = 0;
00584     }
00585     int rank_source = 0, rank_dest = 0;
00586     MPI_Cart_shift(envelopeLattice->comm, dim, -1, &rank_source,
00587                   &rank_dest); // s.t. rank_dest is left & v.v.
00588
00589     // nonblocking Irecv/Isend
00590
00591     MPI_Request requests[4];
00592     MPI_Irecv(&ghostCellRight[0], exchangeSize, MPI_SUNREALTYPE, rank_source, 1,
00593              envelopeLattice->comm, &requests[0]);
00594     MPI_Isend(&ghostCellLeftToSend[0], exchangeSize, MPI_SUNREALTYPE, rank_dest,
00595              1, envelopeLattice->comm, &requests[1]);
00596     MPI_Irecv(&ghostCellLeft[0], exchangeSize, MPI_SUNREALTYPE, rank_dest, 2,
00597              envelopeLattice->comm, &requests[2]);
00598     MPI_Isend(&ghostCellRightToSend[0], exchangeSize, MPI_SUNREALTYPE,
00599              rank_source, 2, envelopeLattice->comm, &requests[3]);
00600     MPI_Waitall(4, requests, MPI_STATUS_IGNORE);
00601
00602
00603     // blocking Sendrecv:
00604     /*
00605     MPI_Sendrecv(&ghostCellLeftToSend[0], exchangeSize, MPI_SUNREALTYPE,
00606                 rank_dest, 1, &ghostCellRight[0], exchangeSize, MPI_SUNREALTYPE,
00607                 rank_source, 1, envelopeLattice->comm, MPI_STATUS_IGNORE);
00608     MPI_Sendrecv(&ghostCellRightToSend[0], exchangeSize, MPI_SUNREALTYPE,
00609                 rank_source, 2, &ghostCellLeft[0], exchangeSize, MPI_SUNREALTYPE,
00610                 rank_dest, 2, envelopeLattice->comm, MPI_STATUS_IGNORE);
00611     */
00612 }
00613
00614 /// Check if all flags are set
00615 void LatticePatch::checkFlag(unsigned int flag) const {
00616     if (!(statusFlags & flag)) {
00617         std::string errorMessage;
00618         switch (flag) {
00619             case FLatticePatchSetUp:
00620                 errorMessage = "The Lattice patch was not set up please make sure to "
00621                                "initilize a Lattice topology";
00622                 break;
00623             case TranslocationLookupSetUp:
00624                 errorMessage = "The translocation lookup tables have not been generated, "
00625                                "please be sure to run generateTranslocationLookup()";
00626                 break;
00627             case GhostLayersInitialized:
00628                 errorMessage = "The space for the ghost layers has not been allocated, "
00629                                "please be sure that the ghost cells are initialized ";
00630                 break;
00631             case BuffersInitialized:
00632                 errorMessage = "The space for the buffers has not been allocated, please "
00633                                "be sure to run initializeBuffers()";
00634                 break;
00635             default:
00636                 errorMessage = "Uppss, you've made a non-standard error, sadly I can't "
00637                                "help you there";
00638                 break;
00639         }
00640         errorKill(errorMessage);
00641     }
00642     return;
00643 }
00644
00645 /// Calculate derivatives in the patch (uAux) in the specified direction
00646 void LatticePatch::derive(const int dir) {
00647     // ghost layer width adjusted to the chosen stencil order
00648     const int gLW = envelopeLattice->get_ghostLayerWidth();
00649     // dimensionality of data points -> 6
00650     const int dPD = envelopeLattice->get_dataPointDimension();
00651     // total width of patch in given direction including ghost layers at ends
00652     const sunindextype dirWidth = discreteSize(dir) + 2 * gLW;
00653     // width of patch only in given direction
00654     const sunindextype dirWidth0 = discreteSize(dir);
00655     // size of plane perpendicular to given dimension
00656     const sunindextype perpPlainSize = discreteSize() / discreteSize(dir);
00657     // physical distance between points in that direction
00658     sunrealtype dxi = nan("0x12345");
00659     switch (dir) {
00660     case 1:
00661         dxi = dx;
00662         break;
00663     case 2:
00664         dxi = dy;
00665         break;

```

```

00666     case 3:
00667         dxi = dz;
00668         break;
00669     default:
00670         dxi = 1;
00671         errorKill("Tried to derive in the wrong direction");
00672         break;
00673     }
00674     // Derive according to chosen stencil accuracy order
00675     const int order = envelopeLattice->get_stencilOrder();
00676     switch (order) {
00677     case 1: // gLW=1
00678         #pragma omp parallel for default(none) \
00679         shared(perpPlainSize, dxi, dirWidth, dirWidthO, gLW, dPD, uAux)
00680         for (sunindextype i = 0; i < perpPlainSize; i++) {
00681             #pragma omp simd
00682             for (sunindextype j = (i * dirWidth + gLW) * dPD;
00683                 j < (i * dirWidth + gLW + dirWidthO) * dPD; j += dPD) {
00684                 uAux[j + 0 - gLW * dPD] = s1b(&uAux[j + 0]) / dxi;
00685                 uAux[j + 1 - gLW * dPD] = s1b(&uAux[j + 1]) / dxi;
00686                 uAux[j + 2 - gLW * dPD] = s1f(&uAux[j + 2]) / dxi;
00687                 uAux[j + 3 - gLW * dPD] = s1f(&uAux[j + 3]) / dxi;
00688                 uAux[j + 4 - gLW * dPD] = s1f(&uAux[j + 4]) / dxi;
00689                 uAux[j + 5 - gLW * dPD] = s1f(&uAux[j + 5]) / dxi;
00690             }
00691         }
00692         break;
00693     case 2: // gLW=2
00694         #pragma omp parallel for default(none) \
00695         shared(perpPlainSize, dxi, dirWidth, dirWidthO, gLW, dPD, uAux)
00696         for (sunindextype i = 0; i < perpPlainSize; i++) {
00697             #pragma omp simd
00698             for (sunindextype j = (i * dirWidth + gLW) * dPD;
00699                 j < (i * dirWidth + gLW + dirWidthO) * dPD; j += dPD) {
00700                 uAux[j + 0 - gLW * dPD] = s2b(&uAux[j + 0]) / dxi;
00701                 uAux[j + 1 - gLW * dPD] = s2b(&uAux[j + 1]) / dxi;
00702                 uAux[j + 2 - gLW * dPD] = s2f(&uAux[j + 2]) / dxi;
00703                 uAux[j + 3 - gLW * dPD] = s2f(&uAux[j + 3]) / dxi;
00704                 uAux[j + 4 - gLW * dPD] = s2c(&uAux[j + 4]) / dxi;
00705                 uAux[j + 5 - gLW * dPD] = s2c(&uAux[j + 5]) / dxi;
00706             }
00707         }
00708         break;
00709     case 3: // gLW=2
00710         #pragma omp parallel for default(none) \
00711         shared(perpPlainSize, dxi, dirWidth, dirWidthO, gLW, dPD, uAux)
00712         for (sunindextype i = 0; i < perpPlainSize; i++) {
00713             #pragma omp simd
00714             for (sunindextype j = (i * dirWidth + gLW) * dPD;
00715                 j < (i * dirWidth + gLW + dirWidthO) * dPD; j += dPD) {
00716                 uAux[j + 0 - gLW * dPD] = s3b(&uAux[j + 0]) / dxi;
00717                 uAux[j + 1 - gLW * dPD] = s3b(&uAux[j + 1]) / dxi;
00718                 uAux[j + 2 - gLW * dPD] = s3f(&uAux[j + 2]) / dxi;
00719                 uAux[j + 3 - gLW * dPD] = s3f(&uAux[j + 3]) / dxi;
00720                 uAux[j + 4 - gLW * dPD] = s3f(&uAux[j + 4]) / dxi;
00721                 uAux[j + 5 - gLW * dPD] = s3f(&uAux[j + 5]) / dxi;
00722             }
00723         }
00724         break;
00725     case 4: // gLW=3
00726         #pragma omp parallel for default(none) \
00727         shared(perpPlainSize, dxi, dirWidth, dirWidthO, gLW, dPD, uAux)
00728         for (sunindextype i = 0; i < perpPlainSize; i++) {
00729             #pragma omp simd
00730             for (sunindextype j = (i * dirWidth + gLW) * dPD;
00731                 j < (i * dirWidth + gLW + dirWidthO) * dPD; j += dPD) {
00732                 uAux[j + 0 - gLW * dPD] = s4b(&uAux[j + 0]) / dxi;
00733                 uAux[j + 1 - gLW * dPD] = s4b(&uAux[j + 1]) / dxi;
00734                 uAux[j + 2 - gLW * dPD] = s4f(&uAux[j + 2]) / dxi;
00735                 uAux[j + 3 - gLW * dPD] = s4f(&uAux[j + 3]) / dxi;
00736                 uAux[j + 4 - gLW * dPD] = s4c(&uAux[j + 4]) / dxi;
00737                 uAux[j + 5 - gLW * dPD] = s4c(&uAux[j + 5]) / dxi;
00738             }
00739         }
00740         break;
00741     case 5: // gLW=3
00742         #pragma omp parallel for default(none) \
00743         shared(perpPlainSize, dxi, dirWidth, dirWidthO, gLW, dPD, uAux)
00744         for (sunindextype i = 0; i < perpPlainSize; i++) {
00745             #pragma omp simd
00746             for (sunindextype j = (i * dirWidth + gLW) * dPD;
00747                 j < (i * dirWidth + gLW + dirWidthO) * dPD; j += dPD) {
00748                 uAux[j + 0 - gLW * dPD] = s5b(&uAux[j + 0]) / dxi;
00749                 uAux[j + 1 - gLW * dPD] = s5b(&uAux[j + 1]) / dxi;
00750                 uAux[j + 2 - gLW * dPD] = s5f(&uAux[j + 2]) / dxi;
00751                 uAux[j + 3 - gLW * dPD] = s5f(&uAux[j + 3]) / dxi;
00752                 uAux[j + 4 - gLW * dPD] = s5f(&uAux[j + 4]) / dxi;

```

```

00753         uAux[j + 5 - gLW * dPD] = s5f(&uAux[j + 5]) / dxi;
00754     }
00755 }
00756 break;
00757 case 6: // gLW=4
00758     #pragma omp parallel for default(none) \
00759     shared(perpPlainSize, dxi, dirWidth, dirWidthO, gLW, dPD, uAux)
00760     for (sunindextype i = 0; i < perpPlainSize; i++) {
00761         #pragma omp simd
00762         for (sunindextype j = (i * dirWidth + gLW) * dPD;
00763             j < (i * dirWidth + gLW + dirWidthO) * dPD; j += dPD) {
00764             uAux[j + 0 - gLW * dPD] = s6b(&uAux[j + 0]) / dxi;
00765             uAux[j + 1 - gLW * dPD] = s6b(&uAux[j + 1]) / dxi;
00766             uAux[j + 2 - gLW * dPD] = s6f(&uAux[j + 2]) / dxi;
00767             uAux[j + 3 - gLW * dPD] = s6f(&uAux[j + 3]) / dxi;
00768             uAux[j + 4 - gLW * dPD] = s6c(&uAux[j + 4]) / dxi;
00769             uAux[j + 5 - gLW * dPD] = s6c(&uAux[j + 5]) / dxi;
00770         }
00771     }
00772     break;
00773 case 7: // gLW=4
00774     #pragma omp parallel for default(none) \
00775     shared(perpPlainSize, dxi, dirWidth, dirWidthO, gLW, dPD, uAux)
00776     for (sunindextype i = 0; i < perpPlainSize; i++) {
00777         #pragma omp simd
00778         for (sunindextype j = (i * dirWidth + gLW) * dPD;
00779             j < (i * dirWidth + gLW + dirWidthO) * dPD; j += dPD) {
00780             uAux[j + 0 - gLW * dPD] = s7b(&uAux[j + 0]) / dxi;
00781             uAux[j + 1 - gLW * dPD] = s7b(&uAux[j + 1]) / dxi;
00782             uAux[j + 2 - gLW * dPD] = s7f(&uAux[j + 2]) / dxi;
00783             uAux[j + 3 - gLW * dPD] = s7f(&uAux[j + 3]) / dxi;
00784             uAux[j + 4 - gLW * dPD] = s7f(&uAux[j + 4]) / dxi;
00785             uAux[j + 5 - gLW * dPD] = s7f(&uAux[j + 5]) / dxi;
00786         }
00787     }
00788     break;
00789 case 8: // gLW=5
00790     #pragma omp parallel for default(none) \
00791     shared(perpPlainSize, dxi, dirWidth, dirWidthO, gLW, dPD, uAux)
00792     for (sunindextype i = 0; i < perpPlainSize; i++) {
00793         #pragma omp simd
00794         for (sunindextype j = (i * dirWidth + gLW) * dPD;
00795             j < (i * dirWidth + gLW + dirWidthO) * dPD; j += dPD) {
00796             uAux[j + 0 - gLW * dPD] = s8b(&uAux[j + 0]) / dxi;
00797             uAux[j + 1 - gLW * dPD] = s8b(&uAux[j + 1]) / dxi;
00798             uAux[j + 2 - gLW * dPD] = s8f(&uAux[j + 2]) / dxi;
00799             uAux[j + 3 - gLW * dPD] = s8f(&uAux[j + 3]) / dxi;
00800             uAux[j + 4 - gLW * dPD] = s8c(&uAux[j + 4]) / dxi;
00801             uAux[j + 5 - gLW * dPD] = s8c(&uAux[j + 5]) / dxi;
00802         }
00803     }
00804     break;
00805 case 9: // gLW=5
00806     #pragma omp parallel for default(none) \
00807     shared(perpPlainSize, dxi, dirWidth, dirWidthO, gLW, dPD, uAux)
00808     for (sunindextype i = 0; i < perpPlainSize; i++) {
00809         #pragma omp simd
00810         for (sunindextype j = (i * dirWidth + gLW) * dPD;
00811             j < (i * dirWidth + gLW + dirWidthO) * dPD; j += dPD) {
00812             uAux[j + 0 - gLW * dPD] = s9b(&uAux[j + 0]) / dxi;
00813             uAux[j + 1 - gLW * dPD] = s9b(&uAux[j + 1]) / dxi;
00814             uAux[j + 2 - gLW * dPD] = s9f(&uAux[j + 2]) / dxi;
00815             uAux[j + 3 - gLW * dPD] = s9f(&uAux[j + 3]) / dxi;
00816             uAux[j + 4 - gLW * dPD] = s9f(&uAux[j + 4]) / dxi;
00817             uAux[j + 5 - gLW * dPD] = s9f(&uAux[j + 5]) / dxi;
00818         }
00819     }
00820     break;
00821 case 10: // gLW=6
00822     #pragma omp parallel for default(none) \
00823     shared(perpPlainSize, dxi, dirWidth, dirWidthO, gLW, dPD, uAux)
00824     for (sunindextype i = 0; i < perpPlainSize; i++) {
00825         #pragma omp simd
00826         for (sunindextype j = (i * dirWidth + gLW) * dPD;
00827             j < (i * dirWidth + gLW + dirWidthO) * dPD; j += dPD) {
00828             uAux[j + 0 - gLW * dPD] = s10b(&uAux[j + 0]) / dxi;
00829             uAux[j + 1 - gLW * dPD] = s10b(&uAux[j + 1]) / dxi;
00830             uAux[j + 2 - gLW * dPD] = s10f(&uAux[j + 2]) / dxi;
00831             uAux[j + 3 - gLW * dPD] = s10f(&uAux[j + 3]) / dxi;
00832             uAux[j + 4 - gLW * dPD] = s10c(&uAux[j + 4]) / dxi;
00833             uAux[j + 5 - gLW * dPD] = s10c(&uAux[j + 5]) / dxi;
00834         }
00835     }
00836     break;
00837 case 11: // gLW=6
00838     #pragma omp parallel for default(none) \
00839     shared(perpPlainSize, dxi, dirWidth, dirWidthO, gLW, dPD, uAux)

```

```

00840     for (sunindextype i = 0; i < perpPlainSize; i++) {
00841         #pragma omp simd
00842         for (sunindextype j = (i * dirWidth + gLW) * dPD;
00843             j < (i * dirWidth + gLW + dirWidthO) * dPD; j += dPD) {
00844             uAux[j + 0 - gLW * dPD] = s11b(&uAux[j + 0]) / dxi;
00845             uAux[j + 1 - gLW * dPD] = s11b(&uAux[j + 1]) / dxi;
00846             uAux[j + 2 - gLW * dPD] = s11f(&uAux[j + 2]) / dxi;
00847             uAux[j + 3 - gLW * dPD] = s11f(&uAux[j + 3]) / dxi;
00848             uAux[j + 4 - gLW * dPD] = s11f(&uAux[j + 4]) / dxi;
00849             uAux[j + 5 - gLW * dPD] = s11f(&uAux[j + 5]) / dxi;
00850         }
00851     }
00852     break;
00853 case 12: // gLW=7
00854     #pragma omp parallel for default(none) \
00855     shared(perpPlainSize, dxi, dirWidth, dirWidthO, gLW, dPD, uAux)
00856     for (sunindextype i = 0; i < perpPlainSize; i++) {
00857         #pragma omp simd
00858         for (sunindextype j = (i * dirWidth + gLW) * dPD;
00859             j < (i * dirWidth + gLW + dirWidthO) * dPD; j += dPD) {
00860             uAux[j + 0 - gLW * dPD] = s12b(&uAux[j + 0]) / dxi;
00861             uAux[j + 1 - gLW * dPD] = s12b(&uAux[j + 1]) / dxi;
00862             uAux[j + 2 - gLW * dPD] = s12f(&uAux[j + 2]) / dxi;
00863             uAux[j + 3 - gLW * dPD] = s12f(&uAux[j + 3]) / dxi;
00864             uAux[j + 4 - gLW * dPD] = s12c(&uAux[j + 4]) / dxi;
00865             uAux[j + 5 - gLW * dPD] = s12c(&uAux[j + 5]) / dxi;
00866         }
00867     }
00868     break;
00869 case 13: // gLW=7
00870     // For all points in the plane perpendicular to the given direction
00871     #pragma omp parallel for default(none) \
00872     shared(perpPlainSize, dxi, dirWidth, dirWidthO, gLW, dPD, uAux)
00873     for (sunindextype i = 0; i < perpPlainSize; i++) {
00874         // iterate through the derivation direction
00875         #pragma omp simd
00876         for (sunindextype j = (i * dirWidth
00877             + gLW /*to shift left by gLW below */) * dPD;
00878             j < (i * dirWidth + gLW + dirWidthO) * dPD; j += dPD) {
00879             // Compute the stencil derivative for any of the six field components
00880             // and update position by ghost width shift
00881             uAux[j + 0 - gLW * dPD] = s13b(&uAux[j + 0]) / dxi;
00882             uAux[j + 1 - gLW * dPD] = s13b(&uAux[j + 1]) / dxi;
00883             uAux[j + 2 - gLW * dPD] = s13f(&uAux[j + 2]) / dxi;
00884             uAux[j + 3 - gLW * dPD] = s13f(&uAux[j + 3]) / dxi;
00885             uAux[j + 4 - gLW * dPD] = s13f(&uAux[j + 4]) / dxi;
00886             uAux[j + 5 - gLW * dPD] = s13f(&uAux[j + 5]) / dxi;
00887         }
00888     }
00889     break;
00890 default:
00891     errorKill("Please set an existing stencil order");
00892     break;
00893 }
00894 }
00895 }
00896
00897 ////////// Helper functions //////////
00898
00899 /// Print a specific error message to stderr
00900 void errorKill(const std::string & errorMessage) {
00901     int my_prc=0;
00902     MPI_Comm_rank(MPI_COMM_WORLD, &my_prc);
00903     if (my_prc==0) {
00904         std::cerr << std::endl << "Error: " << errorMessage
00905         << "\nAborting..." << std::endl;
00906         MPI_Abort(MPI_COMM_WORLD, 1);
00907         return;
00908     }
00909 }
00910
00911 /** Check MPI errors. Error handler must be set. */
00912 int check_error(int error, const char *funcname, int id) {
00913     int eclass, len;
00914     char errorstring[MPI_MAX_ERROR_STRING];
00915     if( error != MPI_SUCCESS ) {
00916         MPI_Error_class(error, &eclass);
00917         MPI_Error_string(error, errorstring, &len);
00918         std::cerr << "MPI Error(process " << id << ") in " << funcname << " : "
00919         << errorstring << ", from class " << eclass << std::endl;
00920         return 1;
00921     }
00922     return 0;
00923 }
00924
00925 /** Check function return value. Adapted from CCode examples.
00926     opt == 0 means SUNDIALS function allocates memory so check if

```

```

00927         returned NULL pointer
00928     opt == 1 means SUNDIALS function returns an integer value so check if
00929         retval < 0
00930     opt == 2 means function allocates memory so check if returned
00931         NULL pointer */
00932 int check_retval(void *returnvalue, const char *funcname, int opt, int id) {
00933     int *retval = nullptr;
00934
00935     /* Check if SUNDIALS function returned NULL pointer - no memory allocated */
00936     if (opt == 0 && returnvalue == nullptr) {
00937         fprintf(stderr,
00938             "\nSUNDIALS_ERROR(%d): %s() failed - returned NULL pointer\n\n", id,
00939             funcname);
00940         return (1);
00941     }
00942
00943     /* Check if retval < 0 */
00944     else if (opt == 1) {
00945         retval = (int *)returnvalue;
00946         char *flagname = CNodeGetReturnFlagName(*retval);
00947         if (*retval < 0) {
00948             fprintf(stderr, "\nSUNDIALS_ERROR(%d): %s() failed with retval = %d: "
00949                 "%s\n\n",
00950                 id, funcname, *retval, flagname);
00951             return (1);
00952         }
00953     }
00954
00955     /* Check if function returned NULL pointer - no memory allocated */
00956     else if (opt == 2 && returnvalue == nullptr) {
00957         fprintf(stderr,
00958             "\nMEMORY_ERROR(%d): %s() failed - returned NULL pointer\n\n", id,
00959             funcname);
00960         return (1);
00961     }
00962
00963     return (0);
00964 }

```

6.12 src/LatticePatch.h File Reference

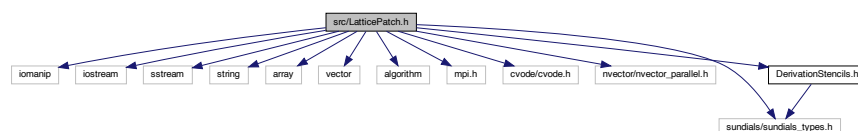
Declaration of the lattice and lattice patches.

```

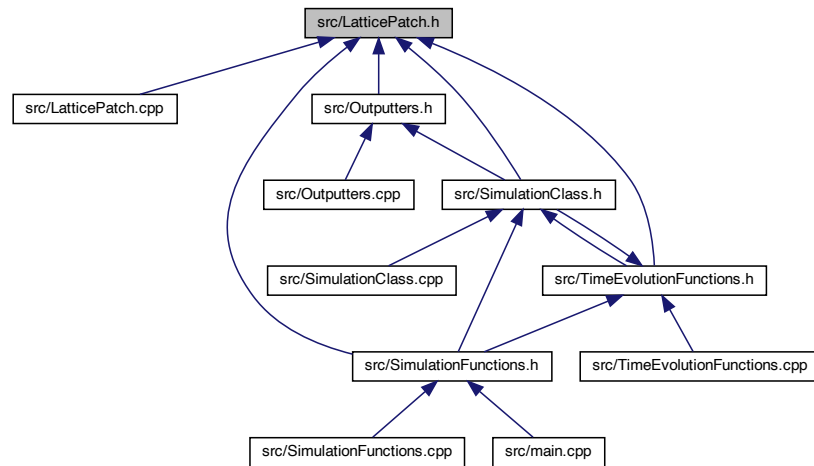
#include <iomanip>
#include <iostream>
#include <sstream>
#include <string>
#include <array>
#include <vector>
#include <algorithm>
#include <mpi.h>
#include <cvode/cvode.h>
#include <nvector/nvector_parallel.h>
#include <sundials/sundials_types.h>
#include "DerivationStencils.h"

```

Include dependency graph for LatticePatch.h:



This graph shows which files directly or indirectly include this file:



Data Structures

- class [Lattice](#)
Lattice class for the construction of the enveloping discrete simulation space.
- class [LatticePatch](#)
LatticePatch class for the construction of the patches in the enveloping lattice.

Functions

- void [errorKill](#) (const std::string &errorMessage)
helper function for error messages
- int [check_error](#) (int error, const char *funcname, int id)
helper function to check MPI errors
- int [check_retval](#) (void *returnvalue, const char *funcname, int opt, int id)
helper function to check CCode errors

Variables

- constexpr unsigned int [FLatticeDimensionSet](#) = 0x01
lattice construction checking flag
- constexpr unsigned int [FLatticePatchSetUp](#) = 0x01
- constexpr unsigned int [TranslocationLookupSetUp](#) = 0x02
- constexpr unsigned int [GhostLayersInitialized](#) = 0x04
- constexpr unsigned int [BuffersInitialized](#) = 0x08

6.12.1 Detailed Description

Declaration of the lattice and lattice patches.

Definition in file [LatticePatch.h](#).

6.12.2 Function Documentation

6.12.2.1 check_error()

```
int check_error (
    int error,
    const char * funcname,
    int id )
```

helper function to check MPI errors

Check MPI errors. Error handler must be set.

Definition at line 912 of file [LatticePatch.cpp](#).

```
00912                                     {
00913     int eclass, len;
00914     char errorstring[MPI_MAX_ERROR_STRING];
00915     if( error != MPI_SUCCESS ) {
00916         MPI_Error_class(error,&eclass);
00917         MPI_Error_string(error,errorstring,&len);
00918         std::cerr << "MPI Error(process " << id << ") in " << funcname << " : "
00919             << errorstring << ", from class " << eclass << std::endl;
00920         return 1;
00921     }
00922     return 0;
00923 }
```

6.12.2.2 check_retval()

```
int check_retval (
    void * returnvalue,
    const char * funcname,
    int opt,
    int id )
```

helper function to check CNode errors

Check function return value. Adapted from CNode examples. opt == 0 means SUNDIALS function allocates memory so check if returned NULL pointer opt == 1 means SUNDIALS function returns an integer value so check if retval < 0 opt == 2 means function allocates memory so check if returned NULL pointer

Definition at line 932 of file [LatticePatch.cpp](#).

```
00932                                     {
00933     int *retval = nullptr;
00934
00935     /* Check if SUNDIALS function returned NULL pointer - no memory allocated */
00936     if (opt == 0 && returnvalue == nullptr) {
00937         fprintf(stderr,
00938             "\nSUNDIALS_ERROR(%d): %s() failed - returned NULL pointer\n\n", id,
```

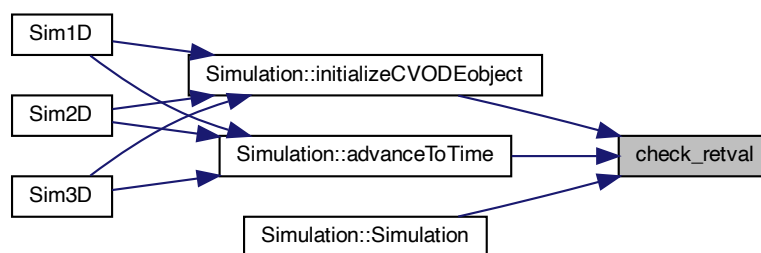
```

00939         funcname);
00940     return (1);
00941 }
00942
00943 /* Check if retval < 0 */
00944 else if (opt == 1) {
00945     retval = (int *)returnvalue;
00946     char *flagname = CNodeGetReturnFlagName(*retval);
00947     if (*retval < 0) {
00948         fprintf(stderr, "\nSUNDIALS_ERROR(%d): %s() failed with retval = %d: "
00949             "%s\n\n",
00950             id, funcname, *retval, flagname);
00951         return (1);
00952     }
00953 }
00954
00955 /* Check if function returned NULL pointer - no memory allocated */
00956 else if (opt == 2 && returnvalue == nullptr) {
00957     fprintf(stderr,
00958         "\nMEMORY_ERROR(%d): %s() failed - returned NULL pointer\n\n", id,
00959         funcname);
00960     return (1);
00961 }
00962
00963 return (0);
00964 }

```

Referenced by [Simulation::advanceToTime\(\)](#), [Simulation::initializeCVODEobject\(\)](#), and [Simulation::Simulation\(\)](#).

Here is the caller graph for this function:



6.12.2.3 errorKill()

```

void errorKill (
    const std::string & errorMessage )

```

helper function for error messages

helper function for error messages

Definition at line 900 of file [LatticePatch.cpp](#).

```

00900     {
00901     int my_prc=0;
00902     MPI_Comm_rank(MPI_COMM_WORLD, &my_prc);
00903     if (my_prc==0) {
00904         std::cerr << std::endl << "Error: " << errorMessage
00905             << "\nAborting..." << std::endl;
00906         MPI_Abort(MPI_COMM_WORLD, 1);
00907         return;
00908     }

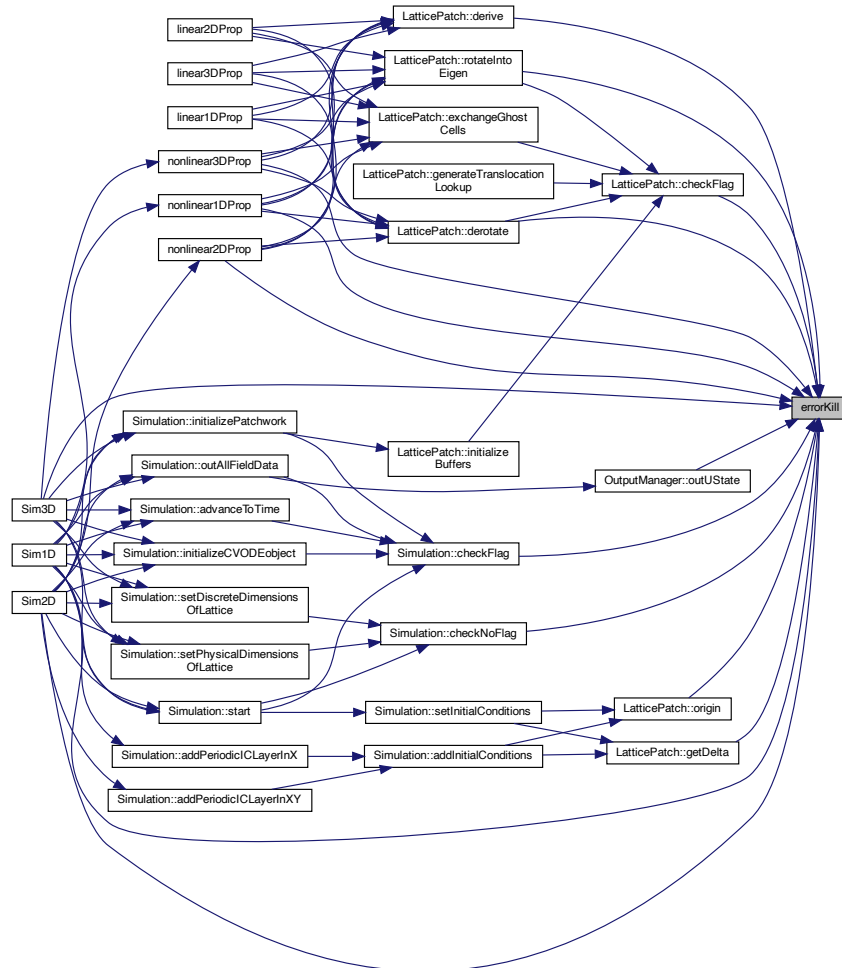
```



```
00909 }
```

Referenced by [LatticePatch::checkFlag\(\)](#), [Simulation::checkFlag\(\)](#), [Simulation::checkNoFlag\(\)](#), [LatticePatch::derive\(\)](#), [LatticePatch::derotate\(\)](#), [LatticePatch::getDelta\(\)](#), [nonlinear1DProp\(\)](#), [nonlinear2DProp\(\)](#), [nonlinear3DProp\(\)](#), [LatticePatch::origin\(\)](#), [OutputManager::outUState\(\)](#), [LatticePatch::rotateIntoEigen\(\)](#), [Sim1D\(\)](#), [Sim2D\(\)](#), and [Sim3D\(\)](#).

Here is the caller graph for this function:



6.12.3 Variable Documentation

6.12.3.1 BuffersInitialized

```
constexpr unsigned int BuffersInitialized = 0x08 [constexpr]
```

lattice patch construction checking flag

Definition at line 42 of file [LatticePatch.h](#).

Referenced by [LatticePatch::checkFlag\(\)](#), and [LatticePatch::initializeBuffers\(\)](#).

6.12.3.2 FLatticeDimensionSet

```
constexpr unsigned int FLatticeDimensionSet = 0x01 [constexpr]
```

lattice construction checking flag

Definition at line 35 of file [LatticePatch.h](#).

Referenced by [LatticePatch::exchangeGhostCells\(\)](#), [LatticePatch::generateTranslocationLookup\(\)](#), [LatticePatch::initializeBuffers\(\)](#), and [Lattice::setPhysicalDimensions\(\)](#).

6.12.3.3 FLatticePatchSetUp

```
constexpr unsigned int FLatticePatchSetUp = 0x01 [constexpr]
```

lattice patch construction checking flag

Definition at line 39 of file [LatticePatch.h](#).

Referenced by [LatticePatch::checkFlag\(\)](#), [LatticePatch::derotate\(\)](#), [LatticePatch::exchangeGhostCells\(\)](#), [LatticePatch::rotateIntoEigen\(\)](#), and [LatticePatch::~~LatticePatch\(\)](#).

6.12.3.4 GhostLayersInitialized

```
constexpr unsigned int GhostLayersInitialized = 0x04 [constexpr]
```

lattice patch construction checking flag

Definition at line 41 of file [LatticePatch.h](#).

Referenced by [LatticePatch::checkFlag\(\)](#), [LatticePatch::exchangeGhostCells\(\)](#), and [LatticePatch::rotateIntoEigen\(\)](#).

6.12.3.5 TranslocationLookupSetUp

```
constexpr unsigned int TranslocationLookupSetUp = 0x02 [constexpr]
```

lattice patch construction checking flag

Definition at line 40 of file [LatticePatch.h](#).

Referenced by [LatticePatch::checkFlag\(\)](#), [LatticePatch::derotate\(\)](#), [LatticePatch::generateTranslocationLookup\(\)](#), and [LatticePatch::rotateIntoEigen\(\)](#).

6.13 LatticePatch.h

[Go to the documentation of this file.](#)

```

00001 ////////////////////////////////////////////////////////////////////
00002 /// @file LatticePatch.h
00003 /// @brief Declaration of the lattice and lattice patches
00004 ////////////////////////////////////////////////////////////////////
00005
00006 #pragma once
00007
00008 // IO
00009 #include <iomanip>
00010 #include <iostream>
00011 #include <sstream>
00012
00013 // string, container, algorithm
00014 #include <string>
00015 // #include <string_view>
00016 #include <array>
00017 #include <vector>
00018 #include <algorithm>
00019
00020 // MPI & OpenMP
00021 #include <mpi.h>
00022 #if defined(_OPENMP)
00023 #include <omp.h>
00024 #endif
00025
00026 // Sundials
00027 #include <cvode/cvode.h> /* prototypes for CVODE fcts. */
00028 #include <nvector/nvector_parallel.h> /* definition of N_Vector and macros */
00029 #include <sundials/sundials_types.h> /* definition of type sunrealtype */
00030
00031 // stencils
00032 #include "DerivationStencils.h"
00033
00034 /// lattice construction checking flag
00035 constexpr unsigned int FLatticeDimensionSet = 0x01;
00036
00037 ///@{
00038 /** lattice patch construction checking flag */
00039 constexpr unsigned int FLatticePatchSetUp = 0x01;
00040 constexpr unsigned int TranslocationLookupSetUp = 0x02;
00041 constexpr unsigned int GhostLayersInitialized = 0x04;
00042 constexpr unsigned int BuffersInitialized = 0x08;
00043 ///@}
00044
00045 /** @brief Lattice class for the construction of the enveloping discrete
00046  * simulation space */
00047 class Lattice {
00048 private:
00049     /// physical size of the lattice in x-direction
00050     sunrealtype tot_lx;
00051     /// physical size of the lattice in y-direction
00052     sunrealtype tot_ly;
00053     /// physical size of the lattice in z-direction
00054     sunrealtype tot_lz;
00055     /// number of points in x-direction
00056     sunindextype tot_nx;
00057     /// number of points in y-direction
00058     sunindextype tot_ny;
00059     /// number of points in z-direction
00060     sunindextype tot_nz;
00061     /// total number of lattice points
00062     sunindextype tot_noP;
00063     /// dimension of each data point set once and for all
00064     static constexpr int dataPointDimension = 6;
00065     /// number of lattice points times data dimension of each point
00066     sunindextype tot_noDP;
00067     /// physical distance between lattice points in x-direction
00068     sunrealtype dx;
00069     /// physical distance between lattice points in y-direction
00070     sunrealtype dy;
00071     /// physical distance between lattice points in z-direction
00072     sunrealtype dz;
00073     /// stencil order
00074     const int stencilOrder;
00075     /// required width of ghost layers (depends on the stencil order)
00076     const int ghostLayerWidth;
00077     /// lattice status flags
00078     unsigned int statusFlags;
00079
00080 public:
00081     /// number of MPI processes
00082     int n_prc;

```

```

00083  /// number of MPI process
00084  int my_prc;
00085  /// personal communicator of the lattice
00086  MPI_Comm comm;
00087  /// function to create and deploy the cartesian communicator
00088  void initializeCommunicator(const int Nx, const int Ny,
00089                             const int Nz, const bool per);
00090  /// default construction
00091  Lattice(const int St0);
00092  /// SUNContext object
00093  SUNContext sunctx;
00094  /// component function for resizing the discrete dimensions of the lattice
00095  void setDiscreteDimensions(const sunindextype _nx,
00096                             const sunindextype _ny, const sunindextype _nz);
00097  /// component function for resizing the physical size of the lattice
00098  void setPhysicalDimensions(const sunrealtype _lx,
00099                             const sunrealtype _ly, const sunrealtype _lz);
00100  ///@{
00101  /** getter function */
00102  [[nodiscard]] const sunrealtype &get_tot_lx() const { return tot_lx; }
00103  [[nodiscard]] const sunrealtype &get_tot_ly() const { return tot_ly; }
00104  [[nodiscard]] const sunrealtype &get_tot_lz() const { return tot_lz; }
00105  [[nodiscard]] const sunindextype &get_tot_nx() const { return tot_nx; }
00106  [[nodiscard]] const sunindextype &get_tot_ny() const { return tot_ny; }
00107  [[nodiscard]] const sunindextype &get_tot_nz() const { return tot_nz; }
00108  [[nodiscard]] const sunindextype &get_tot_noP() const { return tot_noP; }
00109  [[nodiscard]] const sunindextype &get_tot_noDP() const { return tot_noDP; }
00110  [[nodiscard]] const sunrealtype &get_dx() const { return dx; }
00111  [[nodiscard]] const sunrealtype &get_dy() const { return dy; }
00112  [[nodiscard]] const sunrealtype &get_dz() const { return dz; }
00113  [[nodiscard]] constexpr int get_dataPointDimension() const {
00114      return dataPointDimension;
00115  }
00116  [[nodiscard]] const int &get_stencilOrder() const { return stencilOrder; }
00117  [[nodiscard]] const int &get_ghostLayerWidth() const {
00118      return ghostLayerWidth;
00119  }
00120  ///@}
00121 };
00122
00123 /** @brief LatticePatch class for the construction of the patches in the
00124  * enveloping lattice */
00125 class LatticePatch {
00126 private:
00127     /// origin of the patch in physical space; x-coordinate
00128     sunrealtype x0;
00129     /// origin of the patch in physical space; y-coordinate
00130     sunrealtype y0;
00131     /// origin of the patch in physical space; z-coordinate
00132     sunrealtype z0;
00133     /// inner position of lattice-patch in the lattice patchwork; x-points
00134     sunindextype LIx;
00135     /// inner position of lattice-patch in the lattice patchwork; y-points
00136     sunindextype LIy;
00137     /// inner position of lattice-patch in the lattice patchwork; z-points
00138     sunindextype LIz;
00139     /// physical size of the lattice-patch in the x-dimension
00140     sunrealtype lx;
00141     /// physical size of the lattice-patch in the y-dimension
00142     sunrealtype ly;
00143     /// physical size of the lattice-patch in the z-dimension
00144     sunrealtype lz;
00145     /// number of points in the lattice patch in the x-dimension
00146     sunindextype nx;
00147     /// number of points in the lattice patch in the y-dimension
00148     sunindextype ny;
00149     /// number of points in the lattice patch in the z-dimension
00150     sunindextype nz;
00151     /// physical distance between lattice points in x-direction
00152     sunrealtype dx;
00153     /// physical distance between lattice points in y-direction
00154     sunrealtype dy;
00155     /// physical distance between lattice points in z-direction
00156     sunrealtype dz;
00157     /// lattice patch status flags
00158     unsigned int statusFlags;
00159     /// pointer to the enveloping lattice
00160     const Lattice *envelopeLattice;
00161     /// aid (auxilliary) vector including ghost cells to compute the derivatives
00162     std::vector<sunrealtype> uAux;
00163     ///@{
00164     /** translocation lookup table */
00165     std::vector<sunindextype> uTox, uToy, uToz, xTou, yTou, zTou;
00166     ///@}
00167     ///@{
00168     /** buffer to save spatial derivative values */
00169     std::vector<sunrealtype> buffX, buffY, buffZ;

```

```

00170     ///  

00171     ///  

00172     /** buffer for passing ghost cell data */  

00173     std::vector<sunrealtype> ghostCellLeft, ghostCellRight, ghostCellLeftToSend,  

00174         ghostCellRightToSend, ghostCellsToSend, ghostCells;  

00175     ///  

00176     ///  

00177     /** ghost cell translocation lookup table */  

00178     std::vector<sunindextype> lgcTox, rgcTox, lgcToy, rgcToy, lgcToz, rgcToz;  

00179     ///  

00180     ///  

00181     /** Rotate and translocate an input array according to a lookup into an output  

00182      * array */  

00183     inline void rotateToX(sunrealtype *outArray, const sunrealtype *inArray,  

00184         const std::vector<sunindextype> &lookup);  

00185     inline void rotateToY(sunrealtype *outArray, const sunrealtype *inArray,  

00186         const std::vector<sunindextype> &lookup);  

00187     inline void rotateToZ(sunrealtype *outArray, const sunrealtype *inArray,  

00188         const std::vector<sunindextype> &lookup);  

00189     ///  

00190 public:  

00191     ///  

00192     ///  

00193     ///  

00194     N_Vector u;  

00195     ///  

00196     N_Vector du;  

00197     ///  

00198     sunrealtype *uData;  

00199     ///  

00200     sunrealtype *uAuxData;  

00201     ///  

00202     sunrealtype *duData;  

00203     ///  

00204     /** pointer to halo data */  

00205     sunrealtype *gCLData, *gCRData;  

00206     ///  

00207     ///  

00208     std::array<sunrealtype *, 3> buffData;  

00209     ///  

00210     LatticePatch();  

00211     ///  

00212     ~LatticePatch();  

00213     ///  

00214     friend int generatePatchwork(const Lattice &envelopeLattice,  

00215         LatticePatch &patchToMold, const int DLx,  

00216         const int DLy, const int DLz);  

00217     ///  

00218     sunindextype discreteSize(int dir=0) const;  

00219     ///  

00220     sunrealtype origin(const int dir) const;  

00221     ///  

00222     sunrealtype getDelta(const int dir) const;  

00223     ///  

00224     void generateTranslocationLookup();  

00225     ///  

00226     void rotateIntoEigen(const int dir);  

00227     ///  

00228     void derotate(int dir, sunrealtype *buffOut);  

00229     ///  

00230     void initializeBuffers();  

00231     ///  

00232     void exchangeGhostCells(const int dir);  

00233     ///  

00234     void derive(const int dir);  

00235     ///  

00236     void checkFlag(unsigned int flag) const;  

00237 };  

00238  

00239 ///  

00240 void errorKill(const std::string &errorMessage);  

00241  

00242 ///  

00243 int check_error(int error, const char *funcname, int id);  

00244  

00245 ///  

00246 int check_retval(void *returnvalue, const char *funcname, int opt, int id);  

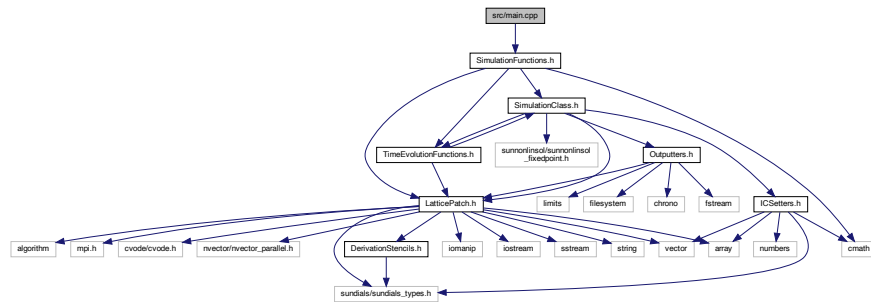
00247

```

6.14 src/main.cpp File Reference

Main function to configure the user's simulation settings.

```
#include "SimulationFunctions.h"
Include dependency graph for main.cpp:
```



Functions

- `int main (int argc, char *argv[])`

6.14.1 Detailed Description

Main function to configure the user's simulation settings.

Definition in file [main.cpp](#).

6.14.2 Function Documentation

6.14.2.1 main()

```
int main (
    int argc,
    char * argv[ ] )
```

Determine the output directory.

A "SimResults" folder will be created if non-existent with a subdirectory named in the identifier format "yy-mm-dd_hh-MM-ss" that contains the csv files

A 1D simulation with specified

A 2D simulation with specified

A 3D simulation with specified

Definition at line 9 of file [main.cpp](#).

```
00010 {
00011     /** Determine the output directory.
00012      * A "SimResults" folder will be created if non-existent
00013      * with a subdirectory named in the identifier format
00014      * "yy-mm-dd_hh-MM-ss" that contains the csv files */
```

```

00015     constexpr auto outputDirectory = "/path/to/directory/";
00016
00017     if(!filesystem::exists(outputDirectory)) {
00018         cerr<<"\nOutput directory nonexistent.\n";
00019         exit(1);
00020     }
00021
00022     // Initialize MPI environment
00023     int provided;
00024     MPI_Init_thread(&argc, &argv, MPI_THREAD_SINGLE, &provided);
00025
00026
00027     //----- BEGIN OF CONFIGURATION -----//
00028
00029     //////////// -- 1D -- ////////////
00030     /** A 1D simulation with specified */
00031     /*
00032     /**/ Specify your settings here /**/
00033     constexpr array<sunrealtype,2> CNodeTolerances={1.0e-16,1.0e-16}; /// - relative and absolute
00034     tolerances of the CNode solver
00035     constexpr int StencilOrder=13; /// - accuracy order of the
00036     stencils in the range 1-13
00037     constexpr sunrealtype physical_sidelength=300e-6; /// - physical length of the
00038     lattice in meters
00039     constexpr sunindextype latticepoints=6e3; /// - number of lattice points
00040     constexpr bool periodic=true; /// - periodic or vanishing
00041     boundary values
00042     int processOrder=3; /// - included processes of the
00043     weak-field expansion, see README.md
00044     constexpr sunrealtype simulationTime=100.0e-6; /// - physical total
00045     simulation time
00046     constexpr int numberOfSteps=100; /// - discrete time steps
00047     constexpr int outputStep=100; /// - output step multiples
00048     constexpr char outputStyle='c'; /// - output in csv (c) or
00049     binary (b) format
00050
00051     /// Add electromagnetic waves.
00052     planewave planel; /// A plane wave with
00053     planel.k = {1e5,0,0}; /// - wavevector (normalized to \f$ 1/\lambda \f$)
00054     planel.p = {0,0,0.1}; /// - amplitude/polarization
00055     planel.phi = {0,0,0}; /// - phase shift
00056     planewave plane2; /// Another plane wave with
00057     plane2.k = {-1e6,0,0}; /// - wavevector (normalized to \f$ 1/\lambda \f$)
00058     plane2.p = {0,0,0.5}; /// - amplitude/polarization
00059     plane2.phi = {0,0,0}; /// - phase shift
00060     // Do not comment out this vector, even if no plane wave is used. But if, emplace used plane
00061     waves.
00062     vector<planewave> planewaves;
00063     //planewaves.emplace_back(planel);
00064     //planewaves.emplace_back(plane2);
00065
00066     gaussian1D gauss1; /// A Gaussian wave with
00067     gauss1.k = {1.0e6,0,0}; /// - wavevector (normalized to \f$ 1/\lambda \f$)
00068     gauss1.p = {0,0,0.1}; /// - polarization/amplitude
00069     gauss1.x0 = {100e-6,0,0}; /// - shift from origin
00070     gauss1.phig = 5e-6; /// - width
00071     gauss1.phi = {0,0,0}; /// - phase shift
00072     gaussian1D gauss2; /// Another Gaussian with
00073     gauss2.k = {-0.2e6,0,0}; /// - wavevector (normalized to \f$ 1/\lambda \f$)
00074     gauss2.p = {0,0,0.5}; /// - polarization/amplitude
00075     gauss2.x0 = {200e-6,0,0}; /// - shift from origin
00076     gauss2.phig = 15e-6; /// - width
00077     gauss2.phi = {0,0,0}; /// - phase shift
00078     // Do not comment out this vector, even if no Gaussian wave is used. But if, emplace used Gaussian
00079     waves.
00080     vector<gaussian1D> Gaussians1D;
00081     Gaussians1D.emplace_back(gauss1);
00082     Gaussians1D.emplace_back(gauss2);
00083
00084     /// Do not change this below ///
00085     int *interactions = &processOrder;
00086     Sim1D(CNodeTolerances,StencilOrder,physical_sidelength,latticepoints,
00087         periodic,interactions,simulationTime,numberOfSteps,
00088         outputDirectory,outputStep,outputStyle,
00089         planewaves,Gaussians1D);
00090
00091     */
00092     ////////////
00093
00094     //////////// -- 2D -- ////////////
00095     /** A 2D simulation with specified */
00096     /*
00097     /**/ Specify your settings here /**/
00098     constexpr array<sunrealtype,2> CNodeTolerances={1.0e-12,1.0e-12}; /// - relative and absolute
00099     tolerances of the CNode solver
00100     constexpr int StencilOrder=13; /// - accuracy order of the
00101     stencils in the range 1-13

```

```

00091     constexpr array<sunrealtype,2> physical_sidelengths={80e-6,80e-6}; /// - physical length of the
        lattice in the given dimensions in meters
00092     constexpr array<sunindextype,2> latticepoints_per_dim={800,800};    /// - number of lattice points
        per dimension
00093     constexpr array<int,2> patches_per_dim={2,2};                      /// - slicing of discrete
        dimensions into patches
00094     constexpr bool periodic=true;                                       /// - periodic or vanishing
        boundary values
00095     int processOrder=3;                                                 /// - included processes of the
        weak-field expansion, see README.md
00096     constexpr sunrealtype simulationTime=40e-6l;                       /// - physical total simulation
        time
00097     constexpr int numberOfSteps=100;                                    /// - discrete time steps
00098     constexpr int outputStep=100;                                       /// - output step multiples
00099     constexpr char outputStyle='c';                                     /// - output in csv (c) or
        binary (b) format

00100
00101     /// Add electromagnetic waves.
00102     planewave plane1;           /// A plane wave with
00103     plane1.k = {1e5,0,0};       /// - wavevector (normalized to  $\frac{1}{\lambda}$ )
00104     plane1.p = {0,0,0.1};       /// - amplitude/polarization
00105     plane1.phi = {0,0,0};       /// - phase shift
00106     planewave plane2;           /// Another plane wave with
00107     plane2.k = {-1e6,0,0};      /// - wavevector
00108     plane2.p = {0,0,0.5};       /// - amplitude/polarization
00109     plane2.phi = {0,0,0};       /// - phase shift
00110     // Do not comment out this vector, even if no plane wave is used. But if, emplace used plane
        waves.
00111     vector<planewave> planewaves;
00112     //planewaves.emplace_back(plane1);
00113     //planewaves.emplace_back(plane2);
00114
00115     gaussian2D gauss1;          /// A Gaussian wave with
00116     gauss1.x0 = {40e-6,40e-6};  /// - center it approaches
00117     gauss1.axis = {1,0};        /// - normalized direction _from_ which the wave approaches the
        center
00118     gauss1.amp = 0.5;           /// - amplitude
00119     gauss1.phip = 2*atan(0);     /// - polarization rotation from TE-mode (z-axis)
00120     gauss1.w0 = 2.3e-6;         /// - taille
00121     gauss1.zr = 16.619e-6;      /// - Rayleigh length
00122     /// the wavelength is determined by the relation  $\frac{1}{\lambda} = \frac{\pi w_0^2}{z_R}$ 
00123     gauss1.ph0 = 2e-5;          /// - beam center
00124     gauss1.phA = 0.45e-5;       /// - beam length
00125     gaussian2D gauss2;          /// Another Gaussian wave with
00126     gauss2.x0 = {40e-6,40e-6};  /// - center it approaches
00127     gauss2.axis = {-0.7071,0.7071}; /// - normalized direction from which the wave approaches the
        center
00128     gauss2.amp = 0.5;           /// - amplitude
00129     gauss2.phip = 2*atan(0);     /// - polarization rotation fom TE-mode (z-axis)
00130     gauss2.w0 = 2.3e-6;         /// - taille
00131     gauss2.zr = 16.619e-6;      /// - Rayleigh length
00132     gauss2.ph0 = 2e-5;          /// - beam center
00133     gauss2.phA = 0.45e-5;       /// - beam length
00134     // Do not comment out this vector, even if no Gaussian wave is used. But if, emplace used Gaussian
        waves.
00135     vector<gaussian2D> Gaussians2D;
00136     Gaussians2D.emplace_back(gauss1);
00137     Gaussians2D.emplace_back(gauss2);
00138
00139     /// Do not change this below ///
00140     static_assert(latticepoints_per_dim[0]*patches_per_dim[0]==0 &&
        latticepoints_per_dim[1]*patches_per_dim[1]==0,
        "The number of lattice points in each dimension must be "
        "divisible by the number of patches in that direction.");
00141     int * interactions = &processOrder;
00142     Sim2D(CVodeTolerances,StencilOrder,physical_sidelengths,
        latticepoints_per_dim,patches_per_dim,periodic,interactions,
00143     simulationTime,numberOfSteps,outputDirectory,outputStep,
        outputStyle,planewaves,Gaussians2D);
00144
00145     */
00150     //////////////////////////////////////
00151
00152
00153     ////////////////////////////////////// -- 3D -- //////////////////////////////////////
00154     /** A 3D simulation with specified */
00155     /*
00156     /// Specify your settings here ///
00157     constexpr array<sunrealtype,2> CVodeTolerances={1.0e-12,1.0e-12};    /// - relative and
        absolute tolerances of the CVode solver
00158     constexpr int StencilOrder=13;                                       /// - accuracy order of
        the stencils in the range 1-13
00159     constexpr array<sunrealtype,3> physical_sidelengths={80e-6,80e-6,20e-6}; /// - physical dimensions
        in meters
00160     constexpr array<sunindextype,3> latticepoints_per_dim={800,800,200}; /// - number of lattice
        points in any dimension
00161     constexpr array<int,3> patches_per_dim= {8,8,2};                    /// - slicing of discrete
        dimensions into patches

```



```

00162     constexpr bool periodic=true;                                     /// - perodic or
non-periodic boundaries
00163     int processOrder=3;                                             /// - processes of the
weak-field expansion, see README.md
00164     constexpr sunrealtype simulationTime=40e-6;                    /// - physical total
simulation time
00165     constexpr int numberOfSteps=40;                                /// - discrete time steps

00166     constexpr int outputStep=20;                                    /// - output step
multiples
00167     constexpr char outputStyle='b';                                /// - output in csv (c)
or binary (b) format

00168     /// Add electromagnetic waves.
00169     planewave plane1;                                              /// A plane wave with
00170     plane1.k = {1e5,0,0};                                          /// - wavevector (normalized to \f$ 1/\lambda \f$)
00171     plane1.p = {0,0,0.1};                                          /// - amplitude/polarization
00172     plane1.phi = {0,0,0};                                          /// - phase shift
00173     planewave plane2;                                              /// Another plane wave with
00174     plane2.k = {-1e6,0,0};                                          /// - wavevector (normalized to \f$ 1/\lambda \f$)
00175     plane2.p = {0,0,0.5};                                          /// - amplitude/polarization
00176     plane2.phi = {0,0,0};                                          /// - phase shift
00177     // Do not comment out this vector, even if no plane wave is used. But if, emplace used plane
00178     waves.
00179     vector<planewave> planewaves;
00180     //planewaves.emplace_back(plane1);
00181     //planewaves.emplace_back(plane2);
00182
00183     gaussian3D gauss1;                                             /// A Gaussian wave with
00184     gauss1.x0 = {40e-6,40e-6,10e-6};                               /// - center it approaches
00185     gauss1.axis = {1,0,0};                                          /// - normalized direction _from_ which the wave approaches
the center
00186     gauss1.amp = 0.05;                                             /// - amplitude
00187     gauss1.phip = 2*atan(0);                                        /// - polarization rotation from TE-mode (z-axis)
00188     gauss1.w0 = 2.3e-6;                                            /// - taille
00189     gauss1.zr = 16.619e-6;                                         /// - Rayleigh length
00190     /// the wavelength is determined by the relation \f$ \lambda = \pi*w_0^2/z_R \f$
00191     gauss1.ph0 = 2e-5;                                             /// - beam center
00192     gauss1.phA = 0.45e-5;                                          /// - beam length
00193     gaussian3D gauss2;                                             /// Another Gaussian wave with
00194     gauss2.x0 = {40e-6,40e-6,10e-6};                               /// - center it approaches
00195     gauss2.axis = {0,1,0};                                          /// - normalized direction from which the wave approaches the
center
00196     gauss2.amp = 0.05;                                             /// - amplitude
00197     gauss2.phip = 2*atan(0);                                        /// - polarization rotation from TE-mode (z-axis)
00198     gauss2.w0 = 2.3e-6;                                            /// - taille
00199     gauss2.zr = 16.619e-6;                                         /// - Rayleigh length
00200     gauss2.ph0 = 2e-5;                                             /// - beam center
00201     gauss2.phA = 0.45e-5;                                          /// - beam length
00202     // Do not comment out this vector, even if no Gaussian wave is used. But if, emplace used Gaussian
00203     waves.
00204     vector<gaussian3D> Gaussians3D;
00205     Gaussians3D.emplace_back(gauss1);
00206     Gaussians3D.emplace_back(gauss2);
00207
00207     /// Do not change this below ///
00208     static_assert(latticepoints_per_dim[0]*patches_per_dim[0]==0 &&
00209         latticepoints_per_dim[1]*patches_per_dim[1]==0 &&
00210         latticepoints_per_dim[2]*patches_per_dim[2]==0,
00211         "The number of lattice points in each dimension must be "
00212         "divisible by the number of patches in that direction.");
00213     static_assert(latticepoints_per_dim[0]/patches_per_dim[0] ==
00214         latticepoints_per_dim[1]/patches_per_dim[1] &&
00215         latticepoints_per_dim[0]/patches_per_dim[0] ==
00216         latticepoints_per_dim[2]/patches_per_dim[2],
00217         "At 3D simulations you are forced to make patches cubic in terms of "
00218         "lattice points as this is decisive for computational efficiency.");
00219     int *interactions = &processOrder;
00220     Sim3D(CVodeTolerances,StencilOrder,physical_sidelengths,
00221         latticepoints_per_dim,patches_per_dim,periodic,interactions,
00222         simulationTime,numberOfSteps,outputDirectory,outputStep,
00223         outputStyle,planewaves,Gaussians3D);
00224     */
00225     //////////////////////////////////////
00226
00227     //----- END OF CONFIGURATION -----//
00228
00229     // Finalize MPI environment
00230     MPI_Finalize();
00231
00232     return 0;
00233 }

```

6.15 main.cpp

[Go to the documentation of this file.](#)

```
00001 /// @file main.cpp
00002 /// @brief Main function to configure the user's simulation settings
00003
00004
00005 #include "SimulationFunctions.h" /* complete simulation functions and all headers */
00006
00007 using namespace std;
00008
00009 int main(int argc, char *argv[])
00010 {
00011     /** Determine the output directory.
00012      * A "SimResults" folder will be created if non-existent
00013      * with a subdirectory named in the identifier format
00014      * "yy-mm-dd_hh-MM-ss" that contains the csv files */
00015     constexpr auto outputDirectory = "/path/to/directory/";
00016
00017     if(!filesystem::exists(outputDirectory)) {
00018         cerr<<"\nOutput directory nonexistent.\n";
00019         exit(1);
00020     }
00021
00022     // Initialize MPI environment
00023     int provided;
00024     MPI_Init_thread(&argc, &argv, MPI_THREAD_SINGLE, &provided);
00025
00026
00027     //----- BEGIN OF CONFIGURATION -----//
00028
00029     ////////////////////////////////// -- 1D -- //////////////////////////////////
00030     /** A 1D simulation with specified */
00031     /**
00032      * Specify your settings here */
00033     constexpr array<sunrealtype,2> CNodeTolerances={1.0e-16,1.0e-16}; /// - relative and absolute
00034     tolerances of the CNode solver
00035     constexpr int StencilOrder=13; /// - accuracy order of the
00036     stencils in the range 1-13
00037     constexpr sunrealtype physical_sidelength=300e-6; /// - physical length of the
00038     lattice in meters
00039     constexpr sunindextype latticepoints=6e3; /// - number of lattice points
00040     constexpr bool periodic=true; /// - periodic or vanishing
00041     boundary values
00042     int processOrder=3; /// - included processes of the
00043     weak-field expansion, see README.md
00044     constexpr sunrealtype simulationTime=100.0e-6l; /// - physical total
00045     simulation time
00046     constexpr int numberOfSteps=100; /// - discrete time steps
00047     constexpr int outputStep=100; /// - output step multiples
00048     constexpr char outputStyle='c'; /// - output in csv (c) or
00049     binary (b) format
00050
00051     /// Add electromagnetic waves.
00052     planewave plane1; /// A plane wave with
00053     plane1.k = {1e5,0,0}; /// - wavevector (normalized to \f$ 1/\lambda \f$)
00054     plane1.p = {0,0,0.1}; /// - amplitude/polarization
00055     plane1.phi = {0,0,0}; /// - phase shift
00056     planewave plane2; /// Another plane wave with
00057     plane2.k = {-1e6,0,0}; /// - wavevector (normalized to \f$ 1/\lambda \f$)
00058     plane2.p = {0,0,0.5}; /// - amplitude/polarization
00059     plane2.phi = {0,0,0}; /// - phase shift
00060     // Do not comment out this vector, even if no plane wave is used. But if, emplace used plane
00061     waves.
00062     vector<planewave> planewaves;
00063     //planewaves.emplace_back(plane1);
00064     //planewaves.emplace_back(plane2);
00065
00066     gaussian1D gauss1; /// A Gaussian wave with
00067     gauss1.k = {1.0e6,0,0}; /// - wavevector (normalized to \f$ 1/\lambda \f$)
00068     gauss1.p = {0,0,0.1}; /// - polarization/amplitude
00069     gauss1.x0 = {100e-6,0,0}; /// - shift from origin
00070     gauss1.phig = 5e-6; /// - width
00071     gauss1.phi = {0,0,0}; /// - phase shift
00072     gaussian1D gauss2; /// Another Gaussian with
00073     gauss2.k = {-0.2e6,0,0}; /// - wavevector (normalized to \f$ 1/\lambda \f$)
00074     gauss2.p = {0,0,0.5}; /// - polarization/amplitude
00075     gauss2.x0 = {200e-6,0,0}; /// - shift from origin
00076     gauss2.phig = 15e-6; /// - width
00077     gauss2.phi = {0,0,0}; /// - phase shift
00078     // Do not comment out this vector, even if no Gaussian wave is used. But if, emplace used Gaussian
00079     waves.
00080     vector<gaussian1D> Gaussians1D;
00081     Gaussians1D.emplace_back(gauss1);
00082     Gaussians1D.emplace_back(gauss2);
```

```

00074
00075     /// Do not change this below ///
00076     int *interactions = &processOrder;
00077     Sim1D(CVodeTolerances,StencilOrder,physical_sidelength,latticepoints,
00078           periodic,interactions,simulationTime,numberOfSteps,
00079           outputDirectory,outputStep,outputStyle,
00080           planewaves,Gaussians1D);
00081     */
00082     //////////////////////////////////////
00083
00084
00085     ////////////////////////////////// -- 2D -- //////////////////////////////////
00086     /** A 2D simulation with specified */
00087     /*
00088     /// Specify your settings here ///
00089     constexpr array<sunrealtype,2> CVodeTolerances={1.0e-12,1.0e-12}; /// - relative and absolute
00090     tolerances of the CVode solver
00091     constexpr int StencilOrder=13; /// - accuracy order of the
00092     stencils in the range 1-13
00093     constexpr array<sunrealtype,2> physical_sidelengths={80e-6,80e-6}; /// - physical length of the
00094     lattice in the given dimensions in meters
00095     constexpr array<sunindextype,2> latticepoints_per_dim={800,800}; /// - number of lattice points
00096     per dimension
00097     constexpr array<int,2> patches_per_dim={2,2}; /// - slicing of discrete
00098     dimensions into patches
00099     constexpr bool periodic=true; /// - periodic or vanishing
00100     boundary values
00101     int processOrder=3; /// - included processes of the
00102     weak-field expansion, see README.md
00103     constexpr sunrealtype simulationTime=40e-6l; /// - physical total simulation
00104     time
00105     constexpr int numberOfSteps=100; /// - discrete time steps
00106     constexpr int outputStep=100; /// - output step multiples
00107     constexpr char outputStyle='c'; /// - output in csv (c) or
00108     binary (b) format
00109
00110     /// Add electromagnetic waves.
00111     planewave planel; /// A plane wave with
00112     planel.k = {1e5,0,0}; /// - wavevector (normalized to \f$ 1/\lambda \f$)
00113     planel.p = {0,0,0.1}; /// - amplitude/polarization
00114     planel.phi = {0,0,0}; /// - phase shift
00115     planewave plane2; /// Another plane wave with
00116     plane2.k = {-1e6,0,0}; /// - wavevector
00117     plane2.p = {0,0,0.5}; /// - amplitude/polarization
00118     plane2.phi = {0,0,0}; /// - phase shift
00119     // Do not comment out this vector, even if no plane wave is used. But if, emplace used plane
00120     waves.
00121     vector<planewave> planewaves;
00122     //planewaves.emplace_back(planel);
00123     //planewaves.emplace_back(plane2);
00124
00125     gaussian2D gauss1; /// A Gaussian wave with
00126     gauss1.x0 = {40e-6,40e-6}; /// - center it approaches
00127     gauss1.axis = {1,0}; /// - normalized direction _from_ which the wave approaches the
00128     center
00129     gauss1.amp = 0.5; /// - amplitude
00130     gauss1.phip = 2*atan(0); /// - polarization rotation from TE-mode (z-axis)
00131     gauss1.w0 = 2.3e-6; /// - taille
00132     gauss1.zr = 16.619e-6; /// - Rayleigh length
00133     /// the wavelength is determined by the relation \f$ \lambda = \pi w_0^2 / z_R \f$
00134     gauss1.ph0 = 2e-5; /// - beam center
00135     gauss1.phA = 0.45e-5; /// - beam length
00136     gaussian2D gauss2; /// Another Gaussian wave with
00137     gauss2.x0 = {40e-6,40e-6}; /// - center it approaches
00138     gauss2.axis = {-0.7071,0.7071}; /// - normalized direction from which the wave approaches the
00139     center
00140     gauss2.amp = 0.5; /// - amplitude
00141     gauss2.phip = 2*atan(0); /// - polarization rotation fom TE-mode (z-axis)
00142     gauss2.w0 = 2.3e-6; /// - taille
00143     gauss2.zr = 16.619e-6; /// - Rayleigh length
00144     gauss2.ph0 = 2e-5; /// - beam center
00145     gauss2.phA = 0.45e-5; /// - beam length
00146     // Do not comment out this vector, even if no Gaussian wave is used. But if, emplace used Gaussian
00147     waves.
00148     vector<gaussian2D> Gaussians2D;
00149     Gaussians2D.emplace_back(gauss1);
00150     Gaussians2D.emplace_back(gauss2);
00151
00152     /// Do not change this below ///
00153     static_assert(latticepoints_per_dim[0]*patches_per_dim[0]==0 &&
00154                   latticepoints_per_dim[1]*patches_per_dim[1]==0,
00155                   "The number of lattice points in each dimension must be "
00156                   "divisible by the number of patches in that direction.");
00157     int * interactions = &processOrder;
00158     Sim2D(CVodeTolerances,StencilOrder,physical_sidelengths,
00159           latticepoints_per_dim,patches_per_dim,periodic,interactions,
00160           simulationTime,numberOfSteps,outputDirectory,outputStep,

```

```

00148         outputStyle,planewaves,Gaussians2D);
00149     */
00150     //////////////////////////////////////
00151
00152
00153     ////////////////////////////////// -- 3D -- //////////////////////////////////
00154     /** A 3D simulation with specified */
00155     /*
00156     /// Specify your settings here ///
00157     constexpr array<sunrealtype,2> CNodeTolerances={1.0e-12,1.0e-12};          /// - relative and
absolute tolerances of the CNode solver
00158     constexpr int StencilOrder=13;                                          /// - accuracy order of
the stencils in the range 1-13
00159     constexpr array<sunrealtype,3> physical_sidelengths={80e-6,80e-6,20e-6}; /// - physical dimensions
in meters
00160     constexpr array<sunindextype,3> latticepoints_per_dim={800,800,200};    /// - number of lattice
points in any dimension
00161     constexpr array<int,3> patches_per_dim= {8,8,2};                      /// - slicing of discrete
dimensions into patches
00162     constexpr bool periodic=true;                                          /// - perodic or
non-periodic boundaries
00163     int processOrder=3;                                                    /// - processes of the
weak-field expansion, see README.md
00164     constexpr sunrealtype simulationTime=40e-6;                          /// - physical total
simulation time
00165     constexpr int numberOfSteps=40;                                        /// - discrete time steps
00166     constexpr int outputStep=20;                                          /// - output step
multiples
00167     constexpr char outputStyle='b';                                       /// - output in csv (c)
or binary (b) format
00168
00169     /// Add electromagnetic waves.
00170     planewave plane1;                                                      /// A plane wave with
00171     plane1.k = {1e5,0,0};                                                  /// - wavevector (normalized to \f$ 1/\lambda \f$)
00172     plane1.p = {0,0,0.1};                                                  /// - amplitude/polarization
00173     plane1.phi = {0,0,0};                                                  /// - phase shift
00174     planewave plane2;                                                      /// Another plane wave with
00175     plane2.k = {-1e6,0,0};                                                  /// - wavevector (normalized to \f$ 1/\lambda \f$)
00176     plane2.p = {0,0,0.5};                                                  /// - amplitude/polarization
00177     plane2.phi = {0,0,0};                                                  /// - phase shift
00178     // Do not comment out this vector, even if no plane wave is used. But if, emplace used plane
waves.
00179     vector<planewave> planewaves;
00180     //planewaves.emplace_back(plane1);
00181     //planewaves.emplace_back(plane2);
00182
00183     gaussian3D gauss1;                                                      /// A Gaussian wave with
00184     gauss1.x0 = {40e-6,40e-6,10e-6};                                       /// - center it approaches
00185     gauss1.axis = {1,0,0};                                                  /// - normalized direction _from_ which the wave approaches
the center
00186     gauss1.amp = 0.05;                                                      /// - amplitude
00187     gauss1.phip = 2*atan(0);                                                 /// - polarization rotation from TE-mode (z-axis)
00188     gauss1.w0 = 2.3e-6;                                                      /// - taille
00189     gauss1.zr = 16.619e-6;                                                  /// - Rayleigh length
00190     /// the wavelength is determined by the relation \f$ \lambda = \pi*w_0^2/z_R \f$
00191     gauss1.ph0 = 2e-5;                                                      /// - beam center
00192     gauss1.phA = 0.45e-5;                                                  /// - beam length
00193     gaussian3D gauss2;                                                      /// Another Gaussian wave with
00194     gauss2.x0 = {40e-6,40e-6,10e-6};                                       /// - center it approaches
00195     gauss2.axis = {0,1,0};                                                  /// - normalized direction from which the wave approaches the
center
00196     gauss2.amp = 0.05;                                                      /// - amplitude
00197     gauss2.phip = 2*atan(0);                                                 /// - polarization rotation from TE-mode (z-axis)
00198     gauss2.w0 = 2.3e-6;                                                      /// - taille
00199     gauss2.zr = 16.619e-6;                                                  /// - Rayleigh length
00200     gauss2.ph0 = 2e-5;                                                      /// - beam center
00201     gauss2.phA = 0.45e-5;                                                  /// - beam length
00202     // Do not comment out this vector, even if no Gaussian wave is used. But if, emplace used Gaussian
waves.
00203     vector<gaussian3D> Gaussians3D;
00204     Gaussians3D.emplace_back(gauss1);
00205     Gaussians3D.emplace_back(gauss2);
00206
00207     /// Do not change this below ///
00208     static_assert(latticepoints_per_dim[0]*patches_per_dim[0]==0 &&
latticepoints_per_dim[1]*patches_per_dim[1]==0 &&
latticepoints_per_dim[2]*patches_per_dim[2]==0,
00209         "The number of lattice points in each dimension must be "
00210         "divisible by the number of patches in that direction.");
00211     static_assert(latticepoints_per_dim[0]/patches_per_dim[0] ==
latticepoints_per_dim[1]/patches_per_dim[1] &&
latticepoints_per_dim[0]/patches_per_dim[0] ==
latticepoints_per_dim[2]/patches_per_dim[2],
00212         "At 3D simulations you are forced to make patches cubic in terms of "
00213         "lattice points as this is decisive for computational efficiency.");
00214     int *interactions = &processOrder;
00215     Sim3D(CNodeTolerances,StencilOrder,physical_sidelengths,

```

```

00221         latticepoints_per_dim, patches_per_dim, periodic, interactions,
00222         simulationTime, numberOfSteps, outputDirectory, outputStep,
00223         outputStyle, planewaves, Gaussians3D);
00224     */
00225     //////////////////////////////////////
00226
00227     //----- END OF CONFIGURATION -----//
00228
00229     // Finalize MPI environment
00230     MPI_Finalize();
00231
00232     return 0;
00233 }

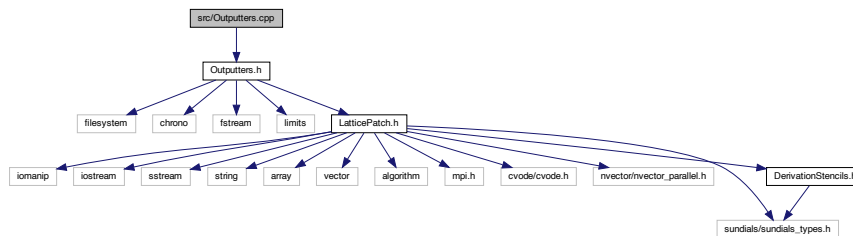
```

6.16 src/Outputters.cpp File Reference

Generation of output writing to disk.

```
#include "Outputters.h"
```

Include dependency graph for Outputters.cpp:



6.16.1 Detailed Description

Generation of output writing to disk.

Definition in file [Outputters.cpp](#).

6.17 Outputters.cpp

[Go to the documentation of this file.](#)

```

00001 //////////////////////////////////////
00002 /// @file Outputters.cpp
00003 /// @brief Generation of output writing to disk
00004 //////////////////////////////////////
00005
00006 #include "Outputters.h"
00007
00008 namespace fs = std::filesystem;
00009 namespace chrono = std::chrono;
00010
00011 /// Directly generate the simCode at construction
00012 OutputManager::OutputManager() {
00013     simCode = SimCodeGenerator();
00014     outputStyle = 'c';
00015 }
00016
00017 /// Generate the identifier number reverse from year to minute in the format
00018 /// yy-mm-dd_hh-MM-ss
00019 std::string OutputManager::SimCodeGenerator() {
00020     const chrono::time_point<chrono::system_clock> now{

```

```

00021     chrono::system_clock::now());
00022     const chrono::year_month_day ymd{chrono::floor<chrono::days>(now)};
00023     const auto tod = now - chrono::floor<chrono::days>(now);
00024     const chrono::hh_mm_ss hms{tod};
00025
00026     std::stringstream temp;
00027     temp << std::setfill('0') << std::setw(2)
00028         << static_cast<int>(ymd.year() - chrono::years(2000)) << "-"
00029         << std::setfill('0') << std::setw(2)
00030         << static_cast<unsigned>(ymd.month()) << "-"
00031         << std::setfill('0') << std::setw(2)
00032         << static_cast<unsigned>(ymd.day()) << "_"
00033         << std::setfill('0') << std::setw(2) << hms.hours().count()
00034         << "-" << std::setfill('0')
00035         << std::setw(2) << hms.minutes().count() << "-"
00036         << std::setfill('0') << std::setw(2)
00037         << hms.seconds().count();
00038     // << "-" << hms.subseconds().count(); // subseconds render the filename
00039     // too large
00040     return temp.str();
00041 }
00042
00043 /** Generate the folder to save the data to by one process:
00044  * In the given directory it creates a directory "SimResults" and a directory
00045  * with the simCode. The relevant part of the main file is written to a
00046  * "config.txt" file in that directory to log the settings. */
00047 void OutputManager::generateOutputFolder(const std::string &dir) {
00048     // Do this only once for the first process
00049     int myPrc;
00050     MPI_Comm_rank(MPI_COMM_WORLD, &myPrc);
00051     if (myPrc == 0) {
00052         if (!fs::is_directory(dir))
00053             fs::create_directory(dir);
00054         if (!fs::is_directory(dir + "/SimResults"))
00055             fs::create_directory(dir + "/SimResults");
00056         if (!fs::is_directory(dir + "/SimResults/" + simCode))
00057             fs::create_directory(dir + "/SimResults/" + simCode);
00058     }
00059     // path variable for the output generation
00060     Path = dir + "/SimResults/" + simCode + "/";
00061
00062     // Logging configurations from main.cpp
00063     std::ifstream fin("main.cpp");
00064     std::ofstream fout(Path + "config.txt");
00065     std::string line;
00066     int begin=1000;
00067     for (int i = 1; !fin.eof(); i++) {
00068         getline(fin, line);
00069         if (line.starts_with(" //----- B")) {
00070             begin=i;
00071         }
00072         if (i < begin) {
00073             continue;
00074         }
00075         fout << line << std::endl;
00076         if (line.starts_with(" //----- E")) {
00077             break;
00078         }
00079     }
00080     return;
00081 }
00082
00083 void OutputManager::set_outputStyle(const char _outputStyle){
00084     outputStyle = _outputStyle;
00085 }
00086
00087 /** Write the field data either in csv format to one file per each process
00088  * (patch) or in binary form to a single file. Files are stores inthe simCode
00089  * directory. For csv files the state (simulation step) denotes the
00090  * prefix and the suffix after an underscore is given by the process/patch
00091  * number. Binary files are simply named after the step number. */
00092 void OutputManager::outUState(const int &state, const Lattice &lattice,
00093     const LatticePatch &latticePatch) {
00094     switch(outputStyle){
00095         case 'c': { // one csv file per process
00096             std::ofstream ofs;
00097             ofs.open(Path + std::to_string(state) + "-"
00098                 + std::to_string(lattice.my_prc) + ".csv");
00099             // Precision of sunrealtype in significant decimal digits; 15 for IEEE double
00100             ofs << std::setprecision(std::numeric_limits<sunrealtype>::digits10);
00101
00102             // Walk through each lattice point
00103             const sunindextype totalNP = latticePatch.discreteSize();
00104             for (sunindextype i = 0; i < totalNP * 6; i += 6) {
00105                 // Six columns to contain the field data: Ex,Ey,Ez,Bx,By,Bz
00106                 ofs << latticePatch.uData[i + 0] << "," << latticePatch.uData[i + 1] << ","
00107                     << latticePatch.uData[i + 2] << "," << latticePatch.uData[i + 3] << ","

```

```

00108         « latticePatch.uData[i + 4] « "," « latticePatch.uData[i + 5]
00109         « std::endl;
00110     }
00111     ofs.close();
00112     break;
00113 }
00114
00115     case 'b': { // a single binary file
00116         // Open the output file
00117         MPI_File fh;
00118         const std::string filename = Path+std::to_string(state);
00119         MPI_File_open(lattice.comm,&filename[0],MPI_MODE_WRONLY|MPI_MODE_CREATE,
00120             MPI_INFO_NULL,&fh);
00121         // number of datapoints in the patch with process offset
00122         const sunindextype count = latticePatch.discreteSize()*
00123             lattice.getDataPointDimension();
00124         MPI_Offset offset = lattice.my_prc*count*sizeof(MPI_SUNREALTYPE);
00125         // Go to offset in file and write data to it; maximal precision in
00126         // "native" representation
00127         MPI_File_set_view(fh,offset,MPI_SUNREALTYPE,MPI_SUNREALTYPE,"native",
00128             MPI_INFO_NULL);
00129         MPI_Request write_request;
00130         MPI_File_iwrite_all(fh,latticePatch.uData,count,MPI_SUNREALTYPE,
00131             &write_request);
00132         MPI_Wait(&write_request,MPI_STATUS_IGNORE);
00133         MPI_File_close(&fh);
00134         break;
00135     }
00136     default: {
00137         errorKill("No valid output style defined."
00138             " Choose between (c): one csv file per process,"
00139             " (b) one binary file");
00140         break;
00141     }
00142 }
00143 }
00144

```

6.18 src/Outputters.h File Reference

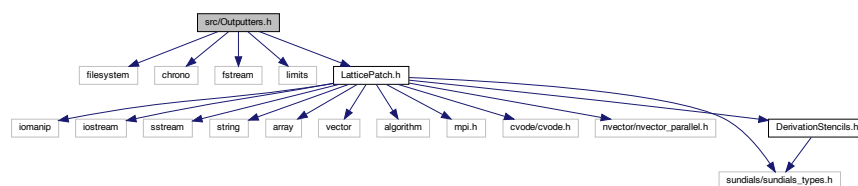
[OutputManager](#) class to outstream simulation data.

```

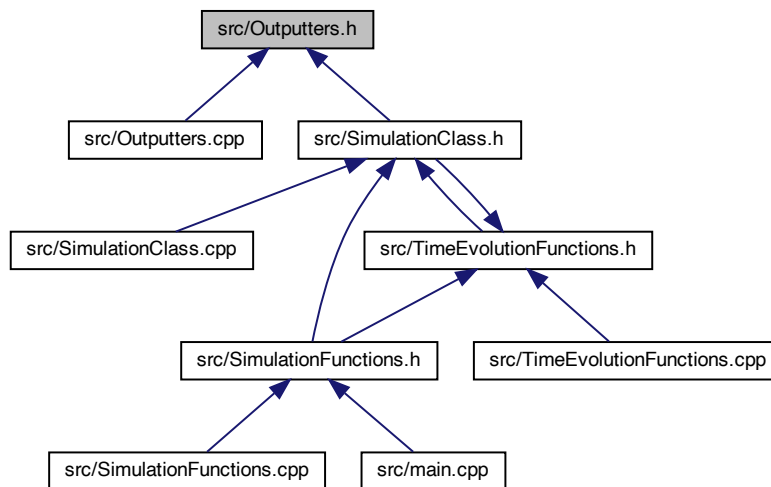
#include <filesystem>
#include <chrono>
#include <fstream>
#include <limits>
#include "LatticePatch.h"

```

Include dependency graph for Outputters.h:



This graph shows which files directly or indirectly include this file:



Data Structures

- class [OutputManager](#)

Output Manager class to generate and coordinate output writing to disk.

6.18.1 Detailed Description

[OutputManager](#) class to outstream simulation data.

Definition in file [Outputters.h](#).

6.19 Outputters.h

[Go to the documentation of this file.](#)

```

00001 ///////////////////////////////////////////////////////////////////
00002 /// @file Outputters.h
00003 /// @brief OutputManager class to outstream simulation data
00004 ///////////////////////////////////////////////////////////////////
00005
00006 #pragma once
00007
00008 // perform operations on the filesystem
00009 #include <filesystem>
00010
00011 // output controlling with limits and timestep
00012 #include <chrono>
00013 #include <fstream>
00014 #include <limits>
00015
00016 // project subfile header
00017 #include "LatticePatch.h"
00018
00019 /** @brief Output Manager class to generate and coordinate output writing to
00020  * disk */
00021 class OutputManager {

```



```

00022 private:
00023     /// function to create the Code of the Simulations
00024     static std::string SimCodeGenerator();
00025     /// variable to save the SimCode generated at execution
00026     std::string simCode;
00027     /// variable for the path to the output folder
00028     std::string Path;
00029     /// output style; csv or binary
00030     char outputStyle;
00031 public:
00032     /// default constructor
00033     OutputManager();
00034     /// function that creates folder to save simulation data
00035     void generateOutputFolder(const std::string &dir);
00036     /// set the output style
00037     void set_outputStyle(const char _outputStyle);
00038     /// function to write data to disk in specified way
00039     void outUState(const int &state, const Lattice &lattice,
00040                  const LatticePatch &latticePatch);
00041     /// simCode getter function
00042     [[nodiscard]] const std::string &getSimCode() const { return simCode; }
00043 };
00044

```

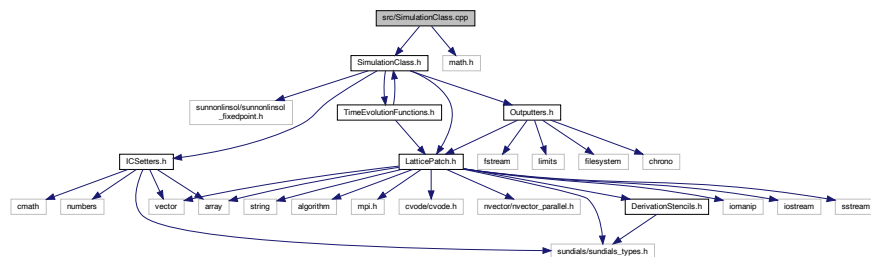
6.20 src/SimulationClass.cpp File Reference

Interface to the whole [Simulation](#) procedure: from wave settings over lattice construction, time evolution and outputs (also all relevant CVODE steps are performed here)

```
#include "SimulationClass.h"
```

```
#include <math.h>
```

Include dependency graph for SimulationClass.cpp:



6.20.1 Detailed Description

Interface to the whole [Simulation](#) procedure: from wave settings over lattice construction, time evolution and outputs (also all relevant CVODE steps are performed here)

Definition in file [SimulationClass.cpp](#).

6.21 SimulationClass.cpp

[Go to the documentation of this file.](#)

```

00001 ///////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
00002 /// @file SimulationClass.cpp
00003 /// @brief Interface to the whole Simulation procedure:
00004 /// from wave settings over lattice construction, time evolution and outputs
00005 /// (also all relevant CVODE steps are performed here)
00006 ///////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

```

```

00007
00008 #include "SimulationClass.h"
00009
00010 #include <math.h>
00011
00012 /// Along with the simulation object, create the cartesian communicator and
00013 /// SUNContext object
00014 Simulation::Simulation(const int Nx, const int Ny, const int Nz,
00015     const int StencilOrder, const bool periodicity) :
00016     lattice(StencilOrder){
00017     statusFlags = 0;
00018     t = 0;
00019     // Initialize the cartesian communicator
00020     lattice.initializeCommunicator(Nx, Ny, Nz, periodicity);
00021
00022     // Create the SUNContext object associated with the thread of execution
00023     int retval = 0;
00024     retval = SUNContext_Create(&lattice.comm, &lattice.sunctx);
00025     if (check_retval(&retval, "SUNContext_Create", 1, lattice.my_prc))
00026         MPI_Abort(lattice.comm, 1);
00027 }
00028
00029 /// Free the CCode solver memory and Sundials context object with the finish of
00030 /// the simulation
00031 Simulation::~Simulation() {
00032     // Free solver memory
00033     if (statusFlags & CcodeObjectSetup) {
00034         CcodeFree(&ccode_mem);
00035         SUNNonlinSolFree(NLS);
00036         SUNContext_Free(&lattice.sunctx);
00037     }
00038 }
00039
00040 /// Set the discrete dimensions, the number of points per dimension
00041 void Simulation::setDiscreteDimensionsOfLattice(const sunindextype nx,
00042     const sunindextype ny, const sunindextype nz) {
00043     checkNoFlag(LatticePatchworkSetup);
00044     lattice.setDiscreteDimensions(nx, ny, nz);
00045     statusFlags |= LatticeDiscreteSetup;
00046 }
00047
00048 /// Set the physical dimensions with lengths in micro meters
00049 void Simulation::setPhysicalDimensionsOfLattice(const sunrealtype lx,
00050     const sunrealtype ly, const sunrealtype lz) {
00051     checkNoFlag(LatticePatchworkSetup);
00052     lattice.setPhysicalDimensions(lx, ly, lz);
00053     statusFlags |= LatticePhysicalSetup;
00054 }
00055
00056 /// Check that the lattice dimensions are set up and generate the patchwork
00057 void Simulation::initializePatchwork(const int nx, const int ny,
00058     const int nz) {
00059     checkFlag(LatticeDiscreteSetup);
00060     checkFlag(LatticePhysicalSetup);
00061
00062     // Generate the patchwork
00063     generatePatchwork(lattice, latticePatch, nx, ny, nz);
00064     latticePatch.initializeBuffers();
00065
00066     statusFlags |= LatticePatchworkSetup;
00067 }
00068
00069 /// Configure CCODE
00070 void Simulation::initializeCCODEObject(const sunrealtype reltol,
00071     const sunrealtype abstol) {
00072     checkFlag(SimulationStarted);
00073
00074     // CCode settings return value
00075     int retval = 0;
00076
00077     // Create CCODE object -- returns a pointer to the ccode memory structure
00078     // with Adams method (Adams-Moulton formula) solver chosen for non-stiff ODE
00079     ccode_mem = CcodeCreate(CV_ADAMS, lattice.sunctx);
00080
00081     // Specify user data and attach it to the main ccode memory block
00082     retval = CcodeSetUserData(
00083         ccode_mem,
00084         &latticePatch); // patch contains the user data as used in CVRhsFn
00085     if (check_retval(&retval, "CcodeSetUserData", 1, lattice.my_prc))
00086         MPI_Abort(lattice.comm, 1);
00087
00088     // Initialize CCODE solver
00089     retval = CcodeInit(ccode_mem, TimeEvolution::f, 0,
00090         latticePatch.u); // allocate memory, CVRhsFn f, t_i=0, u
00091                         // contains the initial values
00092     if (check_retval(&retval, "CcodeInit", 1, lattice.my_prc))
00093         MPI_Abort(lattice.comm, 1);

```

```

00094
00095 // Create fixed point nonlinear solver object (suitable for non-stiff ODE) and
00096 // attach it to CCode
00097 NLS = SUNNonlinSol_FixedPoint(latticePatch.u, 0, lattice.sunctx);
00098 retval = CCodeSetNonlinearSolver(cvode_mem, NLS);
00099 if (check_retval(&retval, "CCodeSetNonlinearSolver", 1, lattice.my_prc))
00100     MPI_Abort(lattice.comm, 1);
00101
00102 // Anderson damping factor
00103 retval = SUNNonlinSolSetDamping_FixedPoint(NLS, 1);
00104 if (check_retval(&retval, "SUNNonlinSolSetDamping_FixedPoint", 1,
00105     lattice.my_prc)) MPI_Abort(lattice.comm, 1);
00106
00107 // Specify integration tolerances -- a scalar relative tolerance and scalar
00108 // absolute tolerance
00109 retval = CCodeSStolerances(cvode_mem, reltol, abstol);
00110 if (check_retval(&retval, "CCodeSStolerances", 1, lattice.my_prc))
00111     MPI_Abort(lattice.comm, 1);
00112
00113 // Specify the maximum number of steps to be taken by the solver in its
00114 // attempt to reach the next tout
00115 retval = CCodeSetMaxNumSteps(cvode_mem, 10000);
00116 if (check_retval(&retval, "CCodeSetMaxNumSteps", 1, lattice.my_prc))
00117     MPI_Abort(lattice.comm, 1);
00118
00119 // maximum number of warnings for too small h
00120 retval = CCodeSetMaxHnilWarns(cvode_mem, 3);
00121 if (check_retval(&retval, "CCodeSetMaxHnilWarns", 1, lattice.my_prc))
00122     MPI_Abort(lattice.comm, 1);
00123
00124 statusFlags |= CCodeObjectSetUp;
00125 }
00126
00127 /// Check if the lattice patchwork is set up and set the initial conditions
00128 void Simulation::start() {
00129     checkFlag(LatticeDiscreteSetUp);
00130     checkFlag(LatticePhysicalSetUp);
00131     checkFlag(LatticePatchworkSetUp);
00132     checkNoFlag(SimulationStarted);
00133     checkNoFlag(CCodeObjectSetUp);
00134     setInitialConditions();
00135     statusFlags |= SimulationStarted;
00136 }
00137
00138 /// Set initial conditions: Fill the lattice points with the initial field
00139 /// values
00140 void Simulation::setInitialConditions() {
00141     const sunrealtype dx = latticePatch.getDelta(1);
00142     const sunrealtype dy = latticePatch.getDelta(2);
00143     const sunrealtype dz = latticePatch.getDelta(3);
00144     const sunindextype nx = latticePatch.discreteSize(1);
00145     const sunindextype ny = latticePatch.discreteSize(2);
00146     const sunindextype totalNP = latticePatch.discreteSize();
00147     const sunrealtype x0 = latticePatch.origin(1);
00148     const sunrealtype y0 = latticePatch.origin(2);
00149     const sunrealtype z0 = latticePatch.origin(3);
00150     sunindextype px = 0, py = 0, pz = 0;
00151     #pragma omp parallel for default(none) \
00152     shared(nx, ny, totalNP, dx, dy, dz, x0, y0, z0) \
00153     firstprivate(px, py, pz) schedule(static)
00154     for (sunindextype i = 0; i < totalNP * 6; i += 6) {
00155         px = (i / 6) % nx;
00156         py = ((i / 6) / nx) % ny;
00157         pz = ((i / 6) / nx) / ny;
00158         // Call the 'eval' function to fill the lattice points with the field data
00159         icsettings.eval(static_cast<sunrealtype>(px) * dx + x0,
00160             static_cast<sunrealtype>(py) * dy + y0,
00161             static_cast<sunrealtype>(pz) * dz + z0, &latticePatch.uData[i]);
00162     }
00163     return;
00164 }
00165
00166 /// Use parameters to add periodic IC layers
00167 void Simulation::addInitialConditions(const sunindextype xm,
00168     const sunindextype ym,
00169     const sunindextype zm /* zm=0 always */) {
00170     const sunrealtype dx = latticePatch.getDelta(1);
00171     const sunrealtype dy = latticePatch.getDelta(2);
00172     const sunrealtype dz = latticePatch.getDelta(3);
00173     const sunindextype nx = latticePatch.discreteSize(1);
00174     const sunindextype ny = latticePatch.discreteSize(2);
00175     const sunindextype totalNP = latticePatch.discreteSize();
00176     // Correct for demanded displacement, rest as for setInitialConditions
00177     const sunrealtype x0 = latticePatch.origin(1) + xm*lattice.get_tot_lx();
00178     const sunrealtype y0 = latticePatch.origin(2) + ym*lattice.get_tot_ly();
00179     const sunrealtype z0 = latticePatch.origin(3) + zm*lattice.get_tot_lz();
00180     sunindextype px = 0, py = 0, pz = 0;

```

```

00181     for (sunindextype i = 0; i < totalNP * 6; i += 6) {
00182         px = (i / 6) % nx;
00183         py = ((i / 6) / nx) % ny;
00184         pz = ((i / 6) / nx) / ny;
00185         icsettings.add(static_cast<sunrealttype>(px) * dx + x0,
00186                       static_cast<sunrealttype>(py) * dy + y0,
00187                       static_cast<sunrealttype>(pz) * dz + z0, &latticePatch.uData[i]);
00188     }
00189     return;
00190 }
00191
00192 /// Add initial conditions in one dimension
00193 void Simulation::addPeriodicICLayerInX() {
00194     addInitialConditions(-1, 0, 0);
00195     addInitialConditions(1, 0, 0);
00196     return;
00197 }
00198
00199 /// Add initial conditions in two dimensions
00200 void Simulation::addPeriodicICLayerInXY() {
00201     addInitialConditions(-1, -1, 0);
00202     addInitialConditions(-1, 0, 0);
00203     addInitialConditions(-1, 1, 0);
00204     addInitialConditions(0, 1, 0);
00205     addInitialConditions(0, -1, 0);
00206     addInitialConditions(1, -1, 0);
00207     addInitialConditions(1, 0, 0);
00208     addInitialConditions(1, 1, 0);
00209     return;
00210 }
00211
00212 /// Advance the solution in time -> integrate the ODE over an interval t
00213 void Simulation::advanceToTime(const sunrealttype &tEnd) {
00214     checkFlag(SimulationStarted);
00215     int retval = 0;
00216     retval = CVode(cvode_mem, tEnd, latticePatch.u, &t,
00217                  CV_NORMAL); // CV_NORMAL: internal steps to reach tEnd, then
00218                             // interpolate to return latticePatch.u, return time
00219                             // reached by the solver as t
00220     if (check_retval(&retval, "CVode", 1, lattice.my_prc))
00221         MPI_Abort(lattice.comm, 1);
00222 }
00223
00224 /// Write specified simulation steps to disk
00225 void Simulation::outAllFieldData(const int &state) {
00226     checkFlag(SimulationStarted);
00227     outputManager.outUState(state, lattice, latticePatch);
00228 }
00229
00230 /// Check presence of configuration flags
00231 void Simulation::checkFlag(unsigned int flag) const {
00232     if (!(statusFlags & flag)) {
00233         std::string errorMessage;
00234         switch (flag) {
00235             case LatticeDiscreteSetUp:
00236                 errorMessage = "The discrete size of the Simulation has not been set up";
00237                 break;
00238             case LatticePhysicalSetUp:
00239                 errorMessage = "The physical size of the Simulation has not been set up";
00240                 break;
00241             case LatticePatchworkSetUp:
00242                 errorMessage = "The patchwork for the Simulation has not been set up";
00243                 break;
00244             case CvodeObjectSetUp:
00245                 errorMessage = "The CVODE object has not been initialized";
00246                 break;
00247             case SimulationStarted:
00248                 errorMessage = "The Simulation has not been started";
00249                 break;
00250             default:
00251                 errorMessage = "Uppss, you've made a non-standard error, sadly I can't "
00252                                "help you there";
00253                 break;
00254         }
00255         errorKill(errorMessage);
00256     }
00257     return;
00258 }
00259
00260 /// Check absence of configuration flags
00261 void Simulation::checkNoFlag(unsigned int flag) const {
00262     if ((statusFlags & flag)) {
00263         std::string errorMessage;
00264         switch (flag) {
00265             case LatticeDiscreteSetUp:
00266                 errorMessage =
00267                     "The discrete size of the Simulation has already been set up";

```

```

00268         break;
00269     case LatticePhysicalSetUp:
00270         errorMessage =
00271             "The physical size of the Simulation has already been set up";
00272         break;
00273     case LatticePatchworkSetUp:
00274         errorMessage = "The patchwork for the Simulation has already been set up";
00275         break;
00276     case CvodeObjectSetUp:
00277         errorMessage = "The CVODE object has already been initialized";
00278         break;
00279     case SimulationStarted:
00280         errorMessage = "The simulation has already started, some changes are no "
00281             "longer possible";
00282         break;
00283     default:
00284         errorMessage = "Uppss, you've made a non-standard error, sadly I can't "
00285             "help you there";
00286         break;
00287     }
00288     errorKill(errorMessage);
00289 }
00290 return;
00291 }

```

6.22 src/SimulationClass.h File Reference

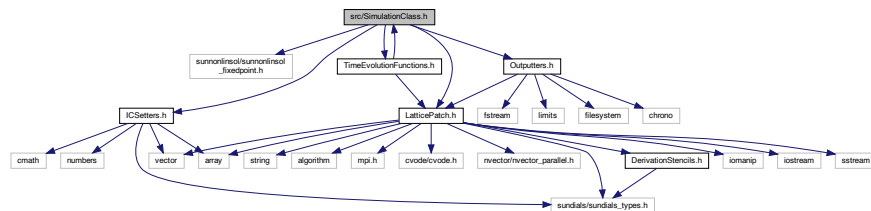
Class for the [Simulation](#) object calling all functionality: from wave settings over lattice construction, time evolution and outputs initialization of the CCode object.

```

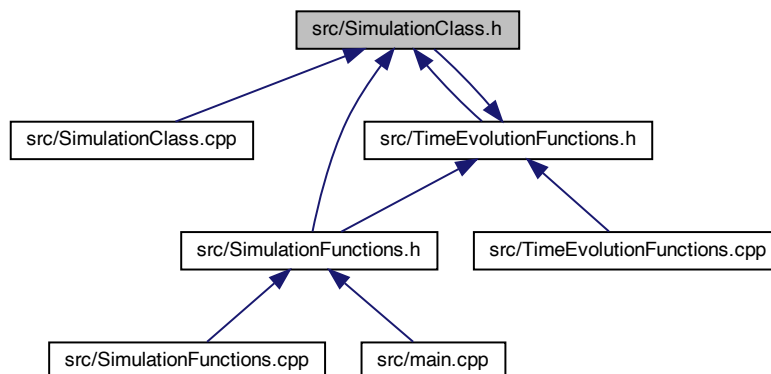
#include "sunnonlinsol/sunnonlinsol_fixedpoint.h"
#include "ICSetters.h"
#include "LatticePatch.h"
#include "Outputters.h"
#include "TimeEvolutionFunctions.h"

```

Include dependency graph for SimulationClass.h:



This graph shows which files directly or indirectly include this file:



Data Structures

- class [Simulation](#)

[Simulation](#) class to instantiate the whole walkthrough of a [Simulation](#).

Variables

- constexpr unsigned int [LatticeDiscreteSetUp](#) = 0x01
- constexpr unsigned int [LatticePhysicalSetUp](#) = 0x02
- constexpr unsigned int [LatticePatchworkSetUp](#) = 0x04
- constexpr unsigned int [CvodeObjectSetUp](#) = 0x08
- constexpr unsigned int [SimulationStarted](#) = 0x10

6.22.1 Detailed Description

Class for the [Simulation](#) object calling all functionality: from wave settings over lattice construction, time evolution and outputs initialization of the CCode object.

Definition in file [SimulationClass.h](#).

6.22.2 Variable Documentation

6.22.2.1 CvodeObjectSetUp

```
constexpr unsigned int CvodeObjectSetUp = 0x08 [constexpr]
```

simulation checking flag

Definition at line 24 of file [SimulationClass.h](#).

Referenced by [Simulation::checkFlag\(\)](#), [Simulation::checkNoFlag\(\)](#), [Simulation::initializeCVODEobject\(\)](#), [Simulation::start\(\)](#), and [Simulation::~~Simulation\(\)](#).

6.22.2.2 LatticeDiscreteSetUp

```
constexpr unsigned int LatticeDiscreteSetUp = 0x01 [constexpr]
```

simulation checking flag

Definition at line 21 of file [SimulationClass.h](#).

Referenced by [Simulation::checkFlag\(\)](#), [Simulation::checkNoFlag\(\)](#), [Simulation::initializePatchwork\(\)](#), [Simulation::setDiscreteDimensionality\(\)](#), and [Simulation::start\(\)](#).

6.22.2.3 LatticePatchworkSetUp

```
constexpr unsigned int LatticePatchworkSetUp = 0x04 [constexpr]
```

simulation checking flag

Definition at line 23 of file [SimulationClass.h](#).

Referenced by [Simulation::checkFlag\(\)](#), [Simulation::checkNoFlag\(\)](#), [Simulation::initializePatchwork\(\)](#), [Simulation::setDiscreteDimensionality\(\)](#), [Simulation::setPhysicalDimensionsOfLattice\(\)](#), and [Simulation::start\(\)](#).

6.22.2.4 LatticePhysicalSetUp

```
constexpr unsigned int LatticePhysicalSetUp = 0x02 [constexpr]
```

simulation checking flag

Definition at line 22 of file [SimulationClass.h](#).

Referenced by [Simulation::checkFlag\(\)](#), [Simulation::checkNoFlag\(\)](#), [Simulation::initializePatchwork\(\)](#), [Simulation::setPhysicalDimensionsOfLattice\(\)](#), and [Simulation::start\(\)](#).

6.22.2.5 SimulationStarted

```
constexpr unsigned int SimulationStarted = 0x10 [constexpr]
```

simulation checking flag

Definition at line 25 of file [SimulationClass.h](#).

Referenced by [Simulation::advanceToTime\(\)](#), [Simulation::checkFlag\(\)](#), [Simulation::checkNoFlag\(\)](#), [Simulation::initializeCVODEobject\(\)](#), [Simulation::outAllFieldData\(\)](#), and [Simulation::start\(\)](#).

6.23 SimulationClass.h

[Go to the documentation of this file.](#)

```
00001 ///////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
00002 /// @file SimulationClass.h
00003 /// @brief Class for the Simulation object calling all functionality:
00004 /// from wave settings over lattice construction, time evolution and outputs
00005 /// initialization of the CNode object
00006 ///////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
00007
00008 #pragma once
00009
00010 /* access to the fixed point SUNNonlinearSolver */
00011 #include "sunnonlinsol/sunnonlinsol_fixedpoint.h"
00012
00013 // project subfile headers
00014 #include "ICSetters.h"
00015 #include "LatticePatch.h"
00016 #include "Outputters.h"
00017 #include "TimeEvolutionFunctions.h"
00018
00019 //@{
00020 /** simulation checking flag */
00021 constexpr unsigned int LatticeDiscreteSetUp = 0x01;
00022 constexpr unsigned int LatticePhysicalSetUp = 0x02;
00023 constexpr unsigned int LatticePatchworkSetUp = 0x04; // not used anymore
00024 constexpr unsigned int CnodeObjectSetUp = 0x08;
00025 constexpr unsigned int SimulationStarted = 0x10;
00026 //@}
00027
00028 /** @brief Simulation class to instantiate the whole walkthrough of a Simulation
00029  */
00030 class Simulation {
00031 private:
00032     /// Lattice object
00033     Lattice lattice;
00034     /// LatticePatch object
00035     LatticePatch latticePatch;
00036     /// current time of the simulation
00037     sunrealtype t;
00038     /// simulation status flags
00039     unsigned int statusFlags;
00040
00041 public:
00042     /// IC Setter object
00043     ICSetter icsettings;
00044     /// Output Manager object
00045     OutputManager outputManager;
00046     /// pointer to CNode memory object
00047     void *cnode_mem;
00048     /// nonlinear solver object
00049     SUNNonlinearSolver NLS;
00050     /// constructor function for the creation of the cartesian communicator
00051     Simulation(const int Nx, const int Ny, const int Nz, const int StencilOrder,
00052               const bool periodicity);
00053     /// destructor function freeing CNode memory and Sundials context
00054     ~Simulation();
00055     /// reference to the cartesian communicator of the lattice (for debugging)
00056     MPI_Comm *get_cart_comm() { return &lattice.comm; }
00057     /// function to set discrete dimensions of the lattice
00058     void setDiscreteDimensionsOfLattice(const sunindextype _tot_nx,
00059                                       const sunindextype _tot_ny, const sunindextype _tot_nz);
00060     /// function to set physical dimensions of the lattice
00061     void setPhysicalDimensionsOfLattice(const sunrealtype lx,
00062                                       const sunrealtype ly, const sunrealtype lz);
```



```

00063  /// function to initialize the Patchwork
00064  void initializePatchwork(const int nx, const int ny, const int nz);
00065  /// function to initialize the CVODE object with all requirements
00066  void initializeCVODEobject(const sunrealtype reltol,
00067                             const sunrealtype abstol);
00068  /// function to start the simulation for time iteration
00069  void start();
00070  /// functions to set the initial field configuration onto the lattice
00071  void setInitialConditions();
00072  /// functions to add initial periodic field configurations
00073  void addInitialConditions(const sunindextype xm, const sunindextype ym,
00074                             const sunindextype zm = 0);
00075  /// function to add a periodic IC layer in one dimension
00076  void addPeriodicICLayerInX();
00077  /// function to add periodic IC layers in two dimensions
00078  void addPeriodicICLayerInXY();
00079  /// function to advance solution in time with CVODE
00080  void advanceToTime(const sunrealtype &tEnd);
00081  /// function to write field data to disk
00082  void outAllFieldData(const int &state);
00083  /// function to check if flag has been set
00084  void checkFlag(unsigned int flag) const;
00085  /// function to check if flag has not been set
00086  // message and cause an abort on all ranks
00087  void checkNoFlag(unsigned int flag) const;
00088 };
00089

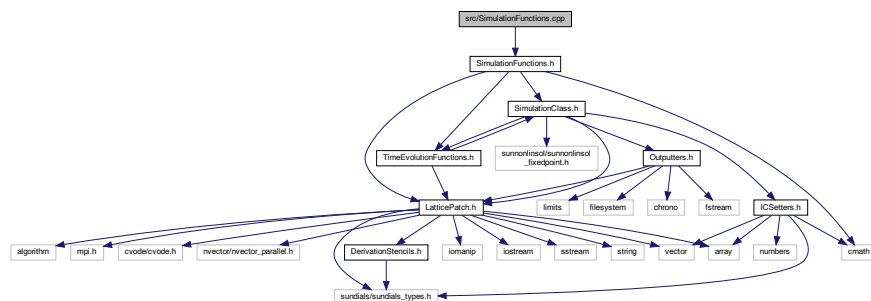
```

6.24 src/SimulationFunctions.cpp File Reference

Implementation of the complete simulation functions for 1D, 2D, and 3D, as called in the main function.

```
#include "SimulationFunctions.h"
```

Include dependency graph for SimulationFunctions.cpp:



Functions

- void **timer** (double &t1, double &t2)
MPI timer function.
- void **Sim1D** (const std::array< sunrealtype, 2 > CVOdeTol, const int StencilOrder, const sunrealtype phys_dim, const sunindextype disc_dim, const bool periodic, int *interactions, const sunrealtype endTime, const int numberOfSteps, const std::string outputDirectory, const int outputStep, const char outputStyle, const std::vector< **planewave** > &planes, const std::vector< **gaussian1D** > &gaussians)
complete 1D Simulation function
- void **Sim2D** (const std::array< sunrealtype, 2 > CVOdeTol, int const StencilOrder, const std::array< sunrealtype, 2 > phys_dims, const std::array< sunindextype, 2 > disc_dims, const std::array< int, 2 > patches, const bool periodic, int *interactions, const sunrealtype endTime, const int numberOfSteps, const std::string outputDirectory, const int outputStep, const char outputStyle, const std::vector< **planewave** > &planes, const std::vector< **gaussian2D** > &gaussians)

complete 2D [Simulation](#) function

- void [Sim3D](#) (const std::array< sunrealtype, 2 > CNodeTol, const int StencilOrder, const std::array< sunrealtype, 3 > phys_dims, const std::array< sunindextype, 3 > disc_dims, const std::array< int, 3 > patches, const bool periodic, int *interactions, const sunrealtype endTime, const int numberOfSteps, const std::string outputDirectory, const int outputStep, const char outputStyle, const std::vector< [planewave](#) > &planes, const std::vector< [gaussian3D](#) > &gaussians)

complete 3D [Simulation](#) function

6.24.1 Detailed Description

Implementation of the complete simulation functions for 1D, 2D, and 3D, as called in the main function.

Definition in file [SimulationFunctions.cpp](#).

6.24.2 Function Documentation

6.24.2.1 Sim1D()

```
void Sim1D (
    const std::array< sunrealtype, 2 > CNodeTol,
    const int StencilOrder,
    const sunrealtype phys_dim,
    const sunindextype disc_dim,
    const bool periodic,
    int * interactions,
    const sunrealtype endTime,
    const int numberOfSteps,
    const std::string outputDirectory,
    const int outputStep,
    const char outputStyle,
    const std::vector< planewave > & planes,
    const std::vector< gaussian1D > & gaussians )
```

complete 1D [Simulation](#) function

Conduct the complete 1D simulation process

Definition at line 21 of file [SimulationFunctions.cpp](#).

```
00028                                     {
00029
00030     // MPI data
00031     double ts = MPI_Wtime();
00032     int myPrc = 0, nPrc = 0;
00033     MPI_Comm_size(MPI_COMM_WORLD, &nPrc);
00034     MPI_Comm_rank(MPI_COMM_WORLD, &myPrc);
00035
00036     // Check feasibility of the patchwork decomposition
00037     if (myPrc == 0) {
00038         if (disc_dim % nPrc != 0) {
00039             errorKill("The number of lattice points must be "
00040                     "divisible by the number of processes.");
00041         }
00042     }
00043
00044     // Initialize the simulation, set up the cartesian communicator
00045     std::array<int, 3> patches = {nPrc, 1, 1};
00046     Simulation sim(patches[0], patches[1], patches[2], StencilOrder, periodic);
```

```

00047
00048 // Configure the patchwork
00049 sim.setPhysicalDimensionsOfLattice(phys_dim,1,1);
00050 sim.setDiscreteDimensionsOfLattice(disc_dim,1,1);
00051 sim.initializePatchwork(patches[0], patches[1], patches[2]);
00052
00053 // Add em-waves
00054 for (const auto &gauss : gaussians)
00055     sim.icsettings.addGauss1D(gauss.k, gauss.p, gauss.x0, gauss.phig,
00056                               gauss.phi);
00057 for (const auto &plane : planes)
00058     sim.icsettings.addPlaneWave1D(plane.k, plane.p, plane.phi);
00059
00060 // Check that the patchwork is ready and set the initial conditions
00061 sim.start();
00062 sim.addPeriodicICLayerInX();
00063
00064 // Initialize CNode with abs and rel tolerances
00065 sim.initializeCNodeObject(CNodeTol[0], CNodeTol[1]);
00066
00067 // Configure the time evolution function
00068 TimeEvolution::c = interactions;
00069 TimeEvolution::TimeEvolver = nonlinear1DProp;
00070
00071 // Configure the output
00072 sim.outputManager.generateOutputFolder(outputDirectory);
00073 if (!myPrc) {
00074     std::cout << "Simulation code: " << sim.outputManager.getSimCode()
00075                 << std::endl;
00076 }
00077 sim.outputManager.set_outputStyle(outputStyle);
00078
00079 //MPI_Barrier(MPI_COMM_WORLD); // insure correct output
00080 //sim.outAllFieldData(0); // output of initial state
00081 // Conduct the propagation in space and time
00082 for (int step = 1; step <= numberOfSteps; step++) {
00083     sim.advanceToTime(endTime / numberOfSteps * step);
00084     if (step % outputStep == 0) {
00085         MPI_Barrier(MPI_COMM_WORLD); // insure correct output
00086         sim.outAllFieldData(step);
00087     }
00088     double tn = MPI_Wtime();
00089     if (!myPrc) {
00090         std::cout << "\rStep " << step << "\t\t" << std::flush;
00091         timer(ts, tn);
00092     }
00093 }
00094
00095 return;
00096 }

```

References [ICSetter::addGauss1D\(\)](#), [Simulation::addPeriodicICLayerInX\(\)](#), [ICSetter::addPlaneWave1D\(\)](#), [Simulation::advanceToTime\(\)](#), [TimeEvolution::c](#), [errorKill\(\)](#), [OutputManager::generateOutputFolder\(\)](#), [OutputManager::getSimCode\(\)](#), [Simulation::icsettings](#), [Simulation::initializeCNodeObject\(\)](#), [Simulation::initializePatchwork\(\)](#), [nonlinear1DProp\(\)](#), [Simulation::outAllFieldData\(\)](#), [Simulation::outputManager](#), [OutputManager::set_outputStyle\(\)](#), [Simulation::setDiscreteDimensionsOfLattice\(\)](#), [Simulation::setPhysicalDimensionsOfLattice\(\)](#), [Simulation::start\(\)](#), [TimeEvolution::TimeEvolver](#), and [timer\(\)](#).


```

const std::array< int, 2 > patches,
const bool periodic,
int * interactions,
const sunrealtype endTime,
const int numberOfSteps,
const std::string outputDirectory,
const int outputStep,
const char outputStyle,
const std::vector< planewave > & planes,
const std::vector< gaussian2D > & gaussians )

```

complete 2D [Simulation](#) function

Conduct the complete 2D simulation process

Definition at line 99 of file [SimulationFunctions.cpp](#).

```

00107                                     {
00108
00109     // MPI data
00110     double ts = MPI_Wtime();
00111     int myPrc = 0, nPrc = 0; // Get process rank and number of processes
00112     MPI_Comm_rank(MPI_COMM_WORLD,
00113                   &myPrc); // Return process rank, number \in [1,nPrc]
00114     MPI_Comm_size(MPI_COMM_WORLD,
00115                   &nPrc); // Return number of processes (communicator size)
00116
00117     // Check feasibility of the patchwork decomposition
00118     if (myPrc == 0) {
00119         if (nPrc != patches[0] * patches[1]) {
00120             errorKill(
00121                 "The number of MPI processes must match the number of patches.");
00122         }
00123     }
00124
00125     // Initialize the simulation, set up the cartesian communicator
00126     Simulation sim(patches[0], patches[1], 1, StencilOrder, periodic);
00127
00128     // Configure the patchwork
00129     sim.setPhysicalDimensionsOfLattice(phys_dims[0],
00130                                       phys_dims[1],
00131                                       1); // spacing of the lattice
00132     sim.setDiscreteDimensionsOfLattice(
00133         disc_dims[0], disc_dims[1], 1); // Spacing equivalence to points
00134     sim.initializePatchwork(patches[0], patches[1], 1);
00135
00136     // Add em-waves
00137     for (const auto &gauss : gaussians)
00138         sim.icsettings.addGauss2D(gauss.x0, gauss.axis, gauss.amp, gauss.phip,
00139                                   gauss.w0, gauss.zr, gauss.ph0, gauss.phA);
00140     for (const auto &plane : planes)
00141         sim.icsettings.addPlaneWave2D(plane.k, plane.p, plane.phi);
00142
00143     // Check that the patchwork is ready and set the initial conditions
00144     sim.start(); // Check if the lattice is set up, set initial field
00145                 // configuration
00146     sim.addPeriodicICLayerInXY(); // insure periodicity in propagation directions
00147
00148     // Initialize CVode with rel and abs tolerances
00149     sim.initializeCVODEobject(CVodeTol[0], CVodeTol[1]);
00150
00151     // Configure the time evolution function
00152     TimeEvolution::c = interactions;
00153     TimeEvolution::TimeEvolver = nonlinear2DProp;
00154
00155     // Configure the output
00156     sim.outputManager.generateOutputFolder(outputDirectory);
00157     if (!myPrc) {
00158         std::cout << "Simulation code: " << sim.outputManager.getSimCode()
00159                   << std::endl;
00160     }
00161     sim.outputManager.set_outputStyle(outputStyle);
00162
00163     //MPI_Barrier(MPI_COMM_WORLD); // insure correct output
00164     //sim.outAllFieldData(0); // output of initial state
00165     // Conduct the propagation in space and time
00166     for (int step = 1; step <= numberOfSteps; step++) {
00167         sim.advanceToTime(endTime / numberOfSteps * step);
00168         if (step % outputStep == 0) {
00169             MPI_Barrier(MPI_COMM_WORLD); // insure correct output
00170             sim.outAllFieldData(step);

```

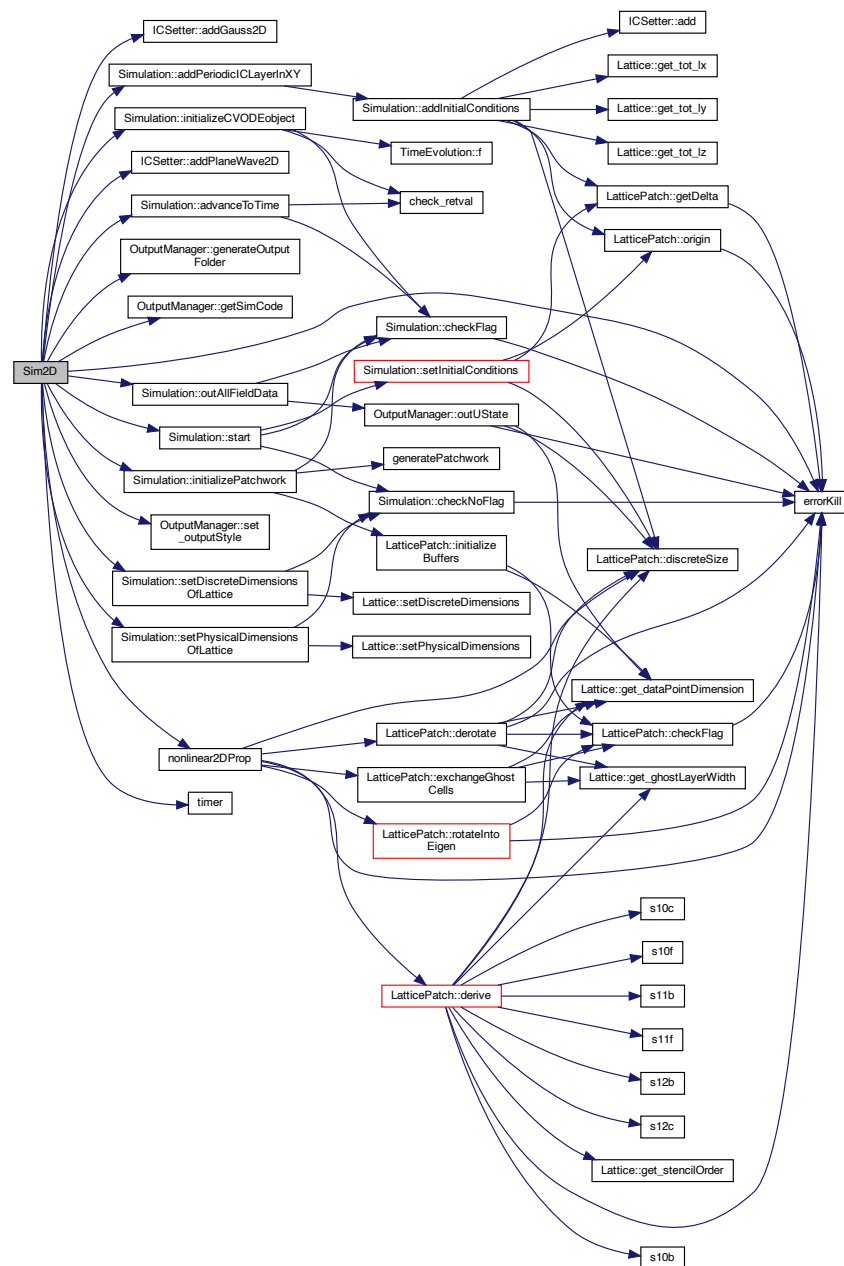
```

00171     }
00172     double tn = MPI_Wtime();
00173     if (!myPrc) {
00174         std::cout << "\rStep " << step << "\t\t" << std::flush;
00175         timer(ts, tn);
00176     }
00177 }
00178
00179 return;
00180 }

```

References [ICSetter::addGauss2D\(\)](#), [Simulation::addPeriodicCLayerInXY\(\)](#), [ICSetter::addPlaneWave2D\(\)](#), [Simulation::advanceToTime\(\)](#), [TimeEvolution::c](#), [errorKill\(\)](#), [OutputManager::generateOutputFolder\(\)](#), [OutputManager::getSimCode\(\)](#), [Simulation::icsettings](#), [Simulation::initializeCVODEobject\(\)](#), [Simulation::initializePatchwork\(\)](#), [nonlinear2DProp\(\)](#), [Simulation::outAllFieldData\(\)](#), [Simulation::outputManager](#), [OutputManager::set_outputStyle\(\)](#), [Simulation::setDiscreteDimensionsOfLattice\(\)](#), [Simulation::setPhysicalDimensionsOfLattice\(\)](#), [Simulation::start\(\)](#), [TimeEvolution::TimeEvolver](#), and [timer\(\)](#).

Here is the call graph for this function:



6.24.2.3 Sim3D()

```
void Sim3D (
    const std::array< sunrealtype, 2 > CVodeTol,
    const int StencilOrder,
    const std::array< sunrealtype, 3 > phys_dims,
    const std::array< sunindextype, 3 > disc_dims,
    const std::array< int, 3 > patches,
    const bool periodic,
    int * interactions,
    const sunrealtype endTime,
    const int numberOfSteps,
    const std::string outputDirectory,
    const int outputStep,
    const char outputStyle,
    const std::vector< planewave > & planes,
    const std::vector< gaussian3D > & gaussians )
```

complete 3D [Simulation](#) function

Conduct the complete 3D simulation process

Definition at line 183 of file [SimulationFunctions.cpp](#).

```
00191                                     {
00192
00193     // MPI data
00194     double ts = MPI_Wtime();
00195     int myPrc = 0, nPrc = 0; // Get process rank and number of process
00196     MPI_Comm_rank(MPI_COMM_WORLD,
00197                   &myPrc); // rank of the process inside the world communicator
00198     MPI_Comm_size(MPI_COMM_WORLD,
00199                   &nPrc); // Size of the communicator is the number of processes
00200
00201     // Check feasibility of the patchwork decomposition
00202     if (myPrc == 0) {
00203         if (nPrc != patches[0] * patches[1] * patches[2]) {
00204             errorKill(
00205                 "The number of MPI processes must match the number of patches.");
00206         }
00207         /*
00208         if ( ( disc_dims[0] / patches[0] != disc_dims[1] / patches[1] ) |
00209             ( disc_dims[0] / patches[0] != disc_dims[2] / patches[2] ) ) {
00210             std::clog
00211                 « "\nWarning: Patches should be cubic in terms of the lattice "
00212                 "points for the computational efficiency of larger simulations.\n";
00213         }
00214         */
00215     }
00216
00217     // Initialize the simulation, set up the cartesian communicator
00218     Simulation sim(patches[0], patches[1], patches[2],
00219                   StencilOrder, periodic); // Simulation object with slicing
00220
00221     // Create the SUNContext object associated with the thread of execution
00222     sim.setPhysicalDimensionsOfLattice(phys_dims[0], phys_dims[1],
00223                                       phys_dims[2]); // spacing of the box
00224     sim.setDiscreteDimensionsOfLattice(
00225         disc_dims[0], disc_dims[1],
00226         disc_dims[2]); // Spacing equivalence to points
00227     sim.initializePatchwork(patches[0], patches[1], patches[2]);
00228
00229     // Add em-waves
00230     for (const auto &plane : planes)
00231         sim.icsettings.addPlaneWave3D(plane.k, plane.p, plane.phi);
00232     for (const auto &gauss : gaussians)
00233         sim.icsettings.addGauss3D(gauss.x0, gauss.axis, gauss.amp, gauss.phip,
00234                                   gauss.w0, gauss.zr, gauss.ph0, gauss.phA);
00235
00236     // Check that the patchwork is ready and set the initial conditions
00237     sim.start();
00238 }
```

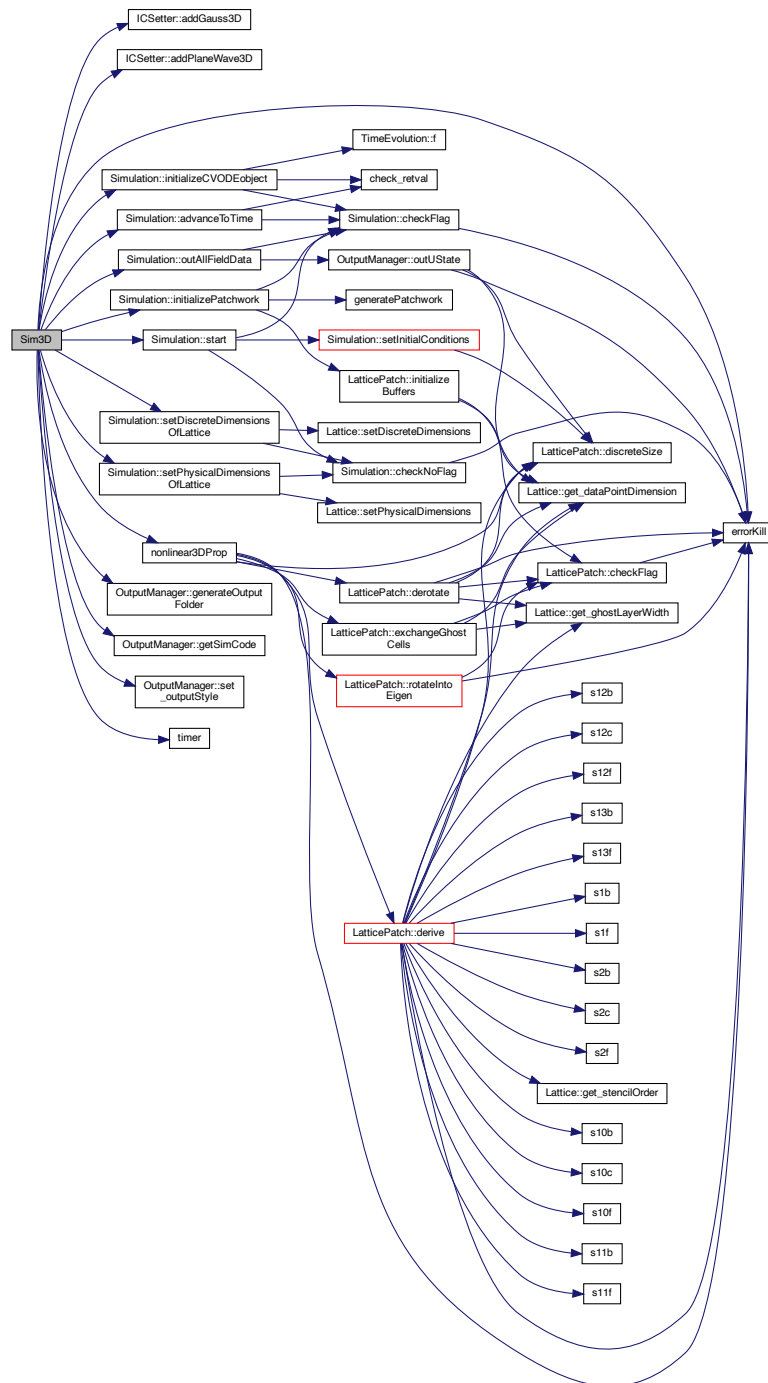
```

00239 // Initialize CNode with abs and rel tolerances
00240 sim.initializeCNodeObject(CNodeTol[0], CNodeTol[1]);
00241
00242 // Configure the time evolution function
00243 TimeEvolution::c = interactions;
00244 TimeEvolution::TimeEvolver = nonlinear3DProp;
00245
00246 // Configure the output
00247 sim.outputManager.generateOutputFolder(outputDirectory);
00248 if (!myProc) {
00249     std::cout << "Simulation code: " << sim.outputManager.getSimCode()
00250     << std::endl;
00251 }
00252 sim.outputManager.set_outputStyle(outputStyle);
00253
00254 //MPI_Barrier(MPI_COMM_WORLD); // insure correct output
00255 //sim.outAllFieldData(0); // output of initial state
00256 // Conduct the propagation in space and time
00257 for (int step = 1; step <= numberOfSteps; step++) {
00258     sim.advanceToTime(endTime / numberOfSteps * step);
00259     if (step % outputStep == 0) {
00260         MPI_Barrier(MPI_COMM_WORLD); // insure correct output
00261         sim.outAllFieldData(step);
00262     }
00263     double tn = MPI_Wtime();
00264     if (!myProc) {
00265         std::cout << "\rStep " << step << "\t\t" << std::flush;
00266         timer(ts, tn);
00267     }
00268 }
00269 return;
00270 }

```

References [ICSetter::addGauss3D\(\)](#), [ICSetter::addPlaneWave3D\(\)](#), [Simulation::advanceToTime\(\)](#), [TimeEvolution::c](#), [errorKill\(\)](#), [OutputManager::generateOutputFolder\(\)](#), [OutputManager::getSimCode\(\)](#), [Simulation::icsettings](#), [Simulation::initializeCNodeObject\(\)](#), [Simulation::initializePatchwork\(\)](#), [nonlinear3DProp\(\)](#), [Simulation::outAllFieldData\(\)](#), [Simulation::outputManager](#), [OutputManager::set_outputStyle\(\)](#), [Simulation::setDiscreteDimensionsOfLattice\(\)](#), [Simulation::setPhysicalDimensionsOfLattice\(\)](#), [Simulation::start\(\)](#), [TimeEvolution::TimeEvolver](#), and [timer\(\)](#).

Here is the call graph for this function:



6.24.2.4 timer()

```
void timer (
    double & t1,
    double & t2 ) [inline]
```

MPI timer function.

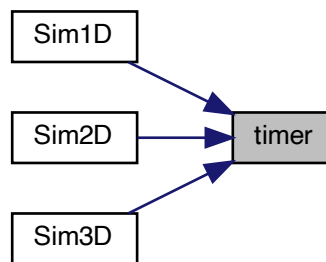
Calculate and print the total simulation time

Definition at line 10 of file [SimulationFunctions.cpp](#).

```
00010 {
00011     printf("Elapsed time:  %fs\n", (t2 - t1));
00012 }
```

Referenced by [Sim1D\(\)](#), [Sim2D\(\)](#), and [Sim3D\(\)](#).

Here is the caller graph for this function:



6.25 SimulationFunctions.cpp

[Go to the documentation of this file.](#)

```
00001 //////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
00002 // @file SimulationFunctions.cpp
00003 // @brief Implementation of the complete simulation functions for
00004 // 1D, 2D, and 3D, as called in the main function
00005 //////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
00006
00007 #include "SimulationFunctions.h"
00008
00009 /** Calculate and print the total simulation time */
00010 inline void timer(double &t1, double &t2) {
00011     printf("Elapsed time:  %fs\n", (t2 - t1));
00012 }
00013
00014 // Instantiate and preliminarily initialize the time evolver
00015 // non-const statics to be defined in actual simulation process
00016 int *TimeEvolution::c = nullptr;
00017 void (*TimeEvolution::TimeEvolver)(LatticePatch *, N_Vector, N_Vector,
00018                                     int *) = nonlinear1DProp;
00019
00020 /** Conduct the complete 1D simulation process */
00021 void Sim1D(const std::array<sunrealtype,2> CNodeTol, const int StencilOrder,
00022            const sunrealtype phys_dim, const sunindextype disc_dim,
00023            const bool periodic, int *interactions,
00024            const sunrealtype endTime, const int numberOfSteps,
00025            const std::string outputDirectory, const int outputStep,
00026            const char outputStyle,
00027            const std::vector<planewave> &planes,
00028            const std::vector<gaussian1D> &gaussians) {
00029
00030     // MPI data
00031     double ts = MPI_Wtime();
00032     int myPrc = 0, nPrc = 0;
00033     MPI_Comm_size(MPI_COMM_WORLD, &nPrc);
00034     MPI_Comm_rank(MPI_COMM_WORLD, &myPrc);
00035
00036     // Check feasibility of the patchwork decomposition
```

```

00037     if (myPrc == 0) {
00038         if (disc_dim % nPrc != 0) {
00039             errorKill("The number of lattice points must be "
00040                 "divisible by the number of processes.");
00041         }
00042     }
00043
00044     // Initialize the simulation, set up the cartesian communicator
00045     std::array<int, 3> patches = {nPrc, 1, 1};
00046     Simulation sim(patches[0], patches[1], patches[2], StencilOrder, periodic);
00047
00048     // Configure the patchwork
00049     sim.setPhysicalDimensionsOfLattice(phys_dim, 1, 1);
00050     sim.setDiscreteDimensionsOfLattice(disc_dim, 1, 1);
00051     sim.initializePatchwork(patches[0], patches[1], patches[2]);
00052
00053     // Add em-waves
00054     for (const auto &gauss : gaussians)
00055         sim.icsettings.addGauss1D(gauss.k, gauss.p, gauss.x0, gauss.phig,
00056             gauss.phi);
00057     for (const auto &plane : planes)
00058         sim.icsettings.addPlaneWave1D(plane.k, plane.p, plane.phi);
00059
00060     // Check that the patchwork is ready and set the initial conditions
00061     sim.start();
00062     sim.addPeriodicICLayerInX();
00063
00064     // Initialize CNode with abs and rel tolerances
00065     sim.initializeCNodeObject(CNodeTol[0], CNodeTol[1]);
00066
00067     // Configure the time evolution function
00068     TimeEvolution::c = interactions;
00069     TimeEvolution::TimeEvolver = nonlinear1DProp;
00070
00071     // Configure the output
00072     sim.outputManager.generateOutputFolder(outputDirectory);
00073     if (!myPrc) {
00074         std::cout << "Simulation code: " << sim.outputManager.getSimCode()
00075             << std::endl;
00076     }
00077     sim.outputManager.set_outputStyle(outputStyle);
00078
00079     //MPI_Barrier(MPI_COMM_WORLD); // insure correct output
00080     //sim.outAllFieldData(0); // output of initial state
00081     // Conduct the propagation in space and time
00082     for (int step = 1; step <= numberOfSteps; step++) {
00083         sim.advanceToTime(endTime / numberOfSteps * step);
00084         if (step % outputStep == 0) {
00085             MPI_Barrier(MPI_COMM_WORLD); // insure correct output
00086             sim.outAllFieldData(step);
00087         }
00088         double tn = MPI_Wtime();
00089         if (!myPrc) {
00090             std::cout << "\rStep " << step << "\t\t" << std::flush;
00091             timer(ts, tn);
00092         }
00093     }
00094
00095     return;
00096 }
00097
00098 /** Conduct the complete 2D simulation process */
00099 void Sim2D(const std::array<sunrealtype, 2> CNodeTol, int const StencilOrder,
00100     const std::array<sunrealtype, 2> phys_dims,
00101     const std::array<sunindextype, 2> disc_dims,
00102     const std::array<int, 2> patches, const bool periodic, int *interactions,
00103     const sunrealtype endTime, const int numberOfSteps,
00104     const std::string outputDirectory, const int outputStep,
00105     const char outputStyle,
00106     const std::vector<planewave> &planes,
00107     const std::vector<gaussian2D> &gaussians) {
00108
00109     // MPI data
00110     double ts = MPI_Wtime();
00111     int myPrc = 0, nPrc = 0; // Get process rank and number of processes
00112     MPI_Comm_rank(MPI_COMM_WORLD,
00113         &myPrc); // Return process rank, number \in [1,nPrc]
00114     MPI_Comm_size(MPI_COMM_WORLD,
00115         &nPrc); // Return number of processes (communicator size)
00116
00117     // Check feasibility of the patchwork decomposition
00118     if (myPrc == 0) {
00119         if (nPrc != patches[0] * patches[1]) {
00120             errorKill(
00121                 "The number of MPI processes must match the number of patches.");
00122         }
00123     }

```

```

00124
00125 // Initialize the simulation, set up the cartesian communicator
00126 Simulation sim(patch[0], patch[1], 1, StencilOrder, periodic);
00127
00128 // Configure the patchwork
00129 sim.setPhysicalDimensionsOfLattice(phys_dims[0],
00130                                   phys_dims[1],
00131                                   1); // spacing of the lattice
00132
00133 sim.setDiscreteDimensionsOfLattice(
00134   disc_dims[0], disc_dims[1], 1); // Spacing equivalence to points
00135
00136 sim.initializePatchwork(patch[0], patch[1], 1);
00137
00138 // Add em-waves
00139 for (const auto &gauss : gaussians)
00140   sim.icsettings.addGauss2D(gauss.x0, gauss.axis, gauss.amp, gauss.phip,
00141                             gauss.w0, gauss.zr, gauss.ph0, gauss.phA);
00142
00143 for (const auto &plane : planes)
00144   sim.icsettings.addPlaneWave2D(plane.k, plane.p, plane.phi);
00145
00146 // Check that the patchwork is ready and set the initial conditions
00147 sim.start(); // Check if the lattice is set up, set initial field
00148 // configuration
00149 sim.addPeriodicICLayerInXY(); // insure periodicity in propagation directions
00150
00151 // Initialize CNode with rel and abs tolerances
00152 sim.initializeCNodeObject(CNodeTol[0], CNodeTol[1]);
00153
00154 // Configure the time evolution function
00155 TimeEvolution::c = interactions;
00156 TimeEvolution::TimeEvolver = nonlinear2DProp;
00157
00158 // Configure the output
00159 sim.outputManager.generateOutputFolder(outputDirectory);
00160
00161 if (!myPrc) {
00162   std::cout << "Simulation code: " << sim.outputManager.getSimCode()
00163             << std::endl;
00164 }
00165
00166 sim.outputManager.set_outputStyle(outputStyle);
00167
00168 //MPI_Barrier(MPI_COMM_WORLD); // insure correct output
00169 //sim.outAllFieldData(0); // output of initial state
00170 // Conduct the propagation in space and time
00171 for (int step = 1; step <= numberOfSteps; step++) {
00172   sim.advanceToTime(endTime / numberOfSteps * step);
00173   if (step % outputStep == 0) {
00174     MPI_Barrier(MPI_COMM_WORLD); // insure correct output
00175     sim.outAllFieldData(step);
00176   }
00177   double tn = MPI_Wtime();
00178   if (!myPrc) {
00179     std::cout << "\rStep " << step << "\t\t" << std::flush;
00180     timer(ts, tn);
00181   }
00182 }
00183
00184 return;
00185 }
00186
00187 /** Conduct the complete 3D simulation process */
00188 void Sim3D(const std::array<sunrealttype,2> CNodeTol, const int StencilOrder,
00189           const std::array<sunrealttype,3> phys_dims,
00190           const std::array<sunindextype,3> disc_dims,
00191           const std::array<int,3> patches,
00192           const bool periodic, int *interactions, const sunrealttype endTime,
00193           const int numberOfSteps, const std::string outputDirectory,
00194           const int outputStep, const char outputStyle,
00195           const std::vector<planewave> &planes,
00196           const std::vector<gaussian3D> &gaussians) {
00197
00198   // MPI data
00199   double ts = MPI_Wtime();
00200   int myPrc = 0, nPrc = 0; // Get process rank and number of process
00201   MPI_Comm_rank(MPI_COMM_WORLD,
00202                 &myPrc); // rank of the process inside the world communicator
00203   MPI_Comm_size(MPI_COMM_WORLD,
00204                 &nPrc); // Size of the communicator is the number of processes
00205
00206   // Check feasibility of the patchwork decomposition
00207   if (myPrc == 0) {
00208     if (nPrc != patches[0] * patches[1] * patches[2]) {
00209       errorKill(
00210         "The number of MPI processes must match the number of patches.");
00211     }
00212   }
00213   /*
00214   if ( ( disc_dims[0] / patches[0] != disc_dims[1] / patches[1] ) |
00215        ( disc_dims[0] / patches[0] != disc_dims[2] / patches[2] ) ) {
00216     std::clog

```

```

00211         « "\nWarning: Patches should be cubic in terms of the lattice "
00212         "points for the computational efficiency of larger simulations.\n";
00213     }
00214     */
00215 }
00216
00217 // Initialize the simulation, set up the cartesian communicator
00218 Simulation sim(patches[0], patches[1], patches[2],
00219             StencilOrder, periodic); // Simulation object with slicing
00220
00221 // Create the SUNContext object associated with the thread of execution
00222 sim.setPhysicalDimensionsOfLattice(phys_dims[0], phys_dims[1],
00223                                   phys_dims[2]); // spacing of the box
00224 sim.setDiscreteDimensionsOfLattice(
00225     disc_dims[0], disc_dims[1],
00226     disc_dims[2]); // Spacing equivalence to points
00227 sim.initializePatchwork(patches[0], patches[1], patches[2]);
00228
00229 // Add em-waves
00230 for (const auto &plane : planes)
00231     sim.icsettings.addPlaneWave3D(plane.k, plane.p, plane.phi);
00232 for (const auto &gauss : gaussians)
00233     sim.icsettings.addGauss3D(gauss.x0, gauss.axis, gauss.amp, gauss.phip,
00234                               gauss.w0, gauss.zr, gauss.ph0, gauss.phA);
00235
00236 // Check that the patchwork is ready and set the initial conditions
00237 sim.start();
00238
00239 // Initialize CNode with abs and rel tolerances
00240 sim.initializeCNodeObject(CNodeTol[0], CNodeTol[1]);
00241
00242 // Configure the time evolution function
00243 TimeEvolution::c = interactions;
00244 TimeEvolution::TimeEvolver = nonlinear3DProp;
00245
00246 // Configure the output
00247 sim.outputManager.generateOutputFolder(outputDirectory);
00248 if (!myProc) {
00249     std::cout << "Simulation code: " << sim.outputManager.getSimCode()
00250               << std::endl;
00251 }
00252 sim.outputManager.set_outputStyle(outputStyle);
00253
00254 //MPI_Barrier(MPI_COMM_WORLD); // insure correct output
00255 //sim.outAllFieldData(0); // output of initial state
00256 // Conduct the propagation in space and time
00257 for (int step = 1; step <= numberOfSteps; step++) {
00258     sim.advanceToTime(endTime / numberOfSteps * step);
00259     if (step % outputStep == 0) {
00260         MPI_Barrier(MPI_COMM_WORLD); // insure correct output
00261         sim.outAllFieldData(step);
00262     }
00263     double tn = MPI_Wtime();
00264     if (!myProc) {
00265         std::cout << "\rStep " << step << "\t\t" << std::flush;
00266         timer(ts, tn);
00267     }
00268 }
00269 return;
00270 }

```

6.26 src/SimulationFunctions.h File Reference

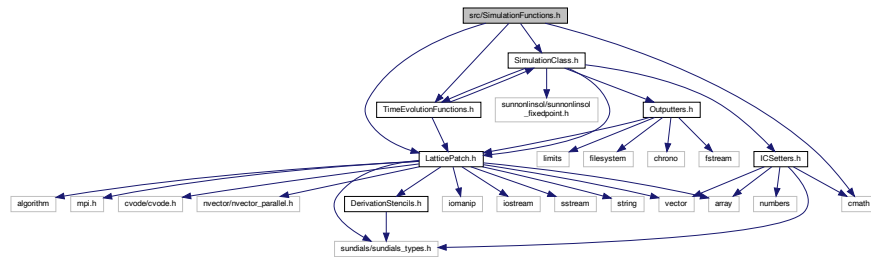
Full simulation functions for 1D, 2D, and 3D used in [main.cpp](#).

```

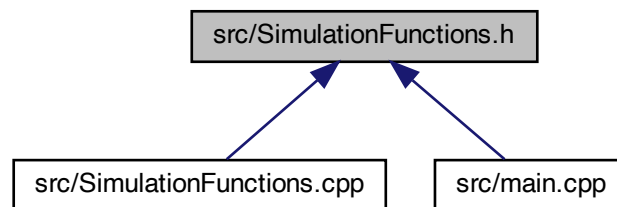
#include <cmath>
#include "LatticePatch.h"
#include "SimulationClass.h"
#include "TimeEvolutionFunctions.h"

```

Include dependency graph for SimulationFunctions.h:



This graph shows which files directly or indirectly include this file:



Data Structures

- struct [planewave](#)
plane wave structure
- struct [gaussian1D](#)
1D Gaussian wave structure
- struct [gaussian2D](#)
2D Gaussian wave structure
- struct [gaussian3D](#)
3D Gaussian wave structure

Functions

- void [Sim1D](#) (const std::array< sunrealtype, 2 >, const int, const sunrealtype, const sunindextype, const bool, int *, const sunrealtype, const int, const std::string, const int, const char, const std::vector< [planewave](#) > &, const std::vector< [gaussian1D](#) > &)
complete 1D Simulation function
- void [Sim2D](#) (const std::array< sunrealtype, 2 >, const int, const std::array< sunrealtype, 2 >, const std::array< sunindextype, 2 >, const std::array< int, 2 >, const bool, int *, const sunrealtype, const int, const std::string, const int, const char, const std::vector< [planewave](#) > &, const std::vector< [gaussian2D](#) > &)
complete 2D Simulation function

- void [Sim3D](#) (const std::array< sunrealtype, 2 >, const int, const std::array< sunrealtype, 3 >, const std::array< sunindextype, 3 >, const std::array< int, 3 >, const bool, int *, const sunrealtype, const int, const std::string, const int, const char, const std::vector< [planewave](#) > &, const std::vector< [gaussian3D](#) > &)
complete 3D Simulation function
- void [timer](#) (double &, double &)
MPI timer function.

6.26.1 Detailed Description

Full simulation functions for 1D, 2D, and 3D used in [main.cpp](#).

Definition in file [SimulationFunctions.h](#).

6.26.2 Function Documentation

6.26.2.1 Sim1D()

```
void Sim1D (
    const std::array< sunrealtype, 2 > CVodeTol,
    const int StencilOrder,
    const sunrealtype phys_dim,
    const sunindextype disc_dim,
    const bool periodic,
    int * interactions,
    const sunrealtype endTime,
    const int numberOfSteps,
    const std::string outputDirectory,
    const int outputStep,
    const char outputStyle,
    const std::vector< planewave > & planes,
    const std::vector< gaussian1D > & gaussians )
```

complete 1D [Simulation](#) function

Conduct the complete 1D simulation process

Definition at line 21 of file [SimulationFunctions.cpp](#).

```
00028 {
00029
00030     // MPI data
00031     double ts = MPI_Wtime();
00032     int myPrc = 0, nPrc = 0;
00033     MPI_Comm_size(MPI_COMM_WORLD, &nPrc);
00034     MPI_Comm_rank(MPI_COMM_WORLD, &myPrc);
00035
00036     // Check feasibility of the patchwork decomposition
00037     if (myPrc == 0) {
00038         if (disc_dim % nPrc != 0) {
00039             errorKill("The number of lattice points must be "
00040                     "divisible by the number of processes.");
00041         }
00042     }
00043
00044     // Initialize the simulation, set up the cartesian communicator
00045     std::array<int, 3> patches = {nPrc, 1, 1};
00046     Simulation sim(patches[0], patches[1], patches[2], StencilOrder, periodic);
00047 }
```

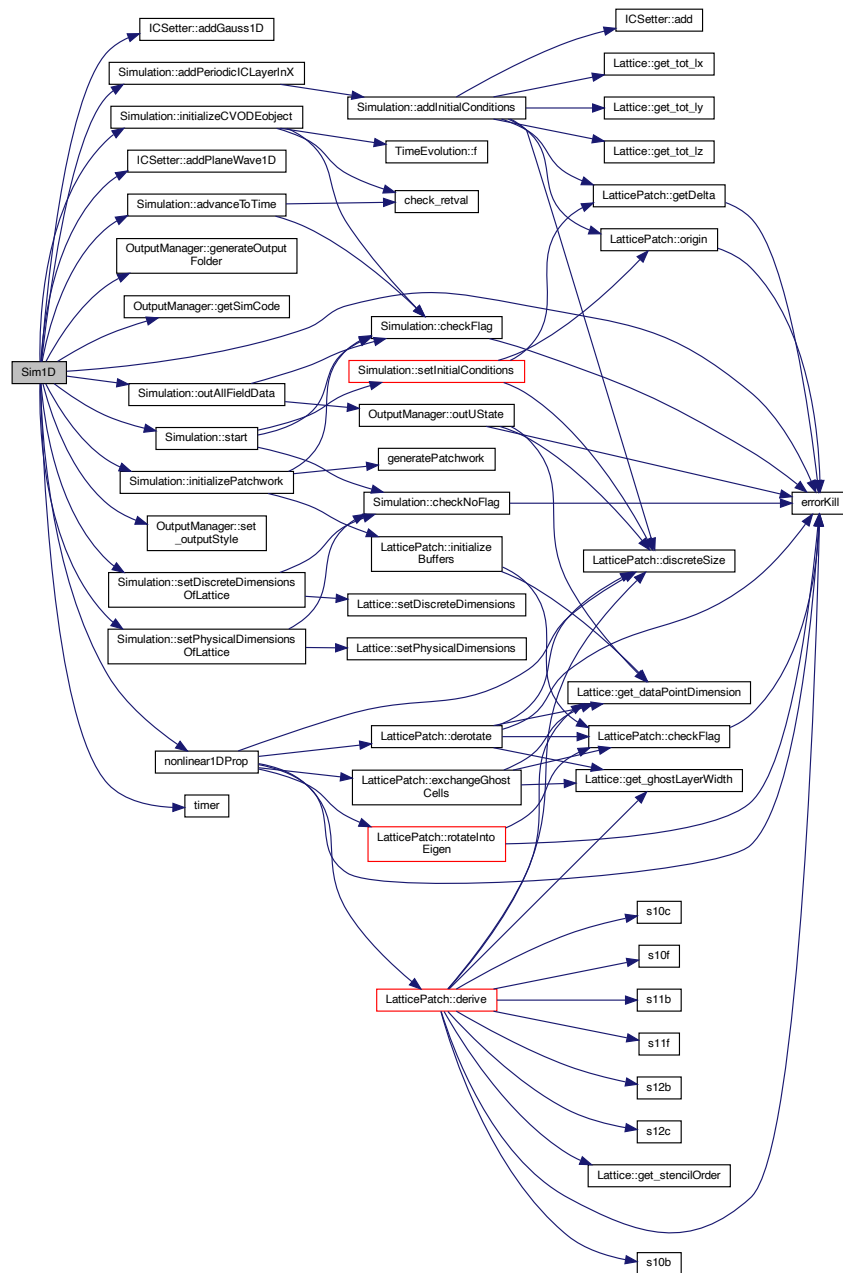
```

00048 // Configure the patchwork
00049 sim.setPhysicalDimensionsOfLattice(phys_dim,1,1);
00050 sim.setDiscreteDimensionsOfLattice(disc_dim,1,1);
00051 sim.initializePatchwork(patches[0], patches[1], patches[2]);
00052
00053 // Add em-waves
00054 for (const auto &gauss : gaussians)
00055     sim.icsettings.addGauss1D(gauss.k, gauss.p, gauss.x0, gauss.phig,
00056                             gauss.phi);
00057 for (const auto &plane : planes)
00058     sim.icsettings.addPlaneWave1D(plane.k, plane.p, plane.phi);
00059
00060 // Check that the patchwork is ready and set the initial conditions
00061 sim.start();
00062 sim.addPeriodicICLayerInX();
00063
00064 // Initialize CVOde with abs and rel tolerances
00065 sim.initializeCVOdeObject(CVodeTol[0], CVodeTol[1]);
00066
00067 // Configure the time evolution function
00068 TimeEvolution::c = interactions;
00069 TimeEvolution::TimeEvolver = nonlinear1DProp;
00070
00071 // Configure the output
00072 sim.outputManager.generateOutputFolder(outputDirectory);
00073 if (!myProc) {
00074     std::cout << "Simulation code: " << sim.outputManager.getSimCode()
00075                 << std::endl;
00076 }
00077 sim.outputManager.set_outputStyle(outputStyle);
00078
00079 //MPI_Barrier(MPI_COMM_WORLD); // insure correct output
00080 //sim.outAllFieldData(0); // output of initial state
00081 // Conduct the propagation in space and time
00082 for (int step = 1; step <= numberOfSteps; step++) {
00083     sim.advanceToTime(endTime / numberOfSteps * step);
00084     if (step % outputStep == 0) {
00085         MPI_Barrier(MPI_COMM_WORLD); // insure correct output
00086         sim.outAllFieldData(step);
00087     }
00088     double tn = MPI_Wtime();
00089     if (!myProc) {
00090         std::cout << "\rStep " << step << "\t\t" << std::flush;
00091         timer(ts, tn);
00092     }
00093 }
00094
00095 return;
00096 }

```

References [ICSetter::addGauss1D\(\)](#), [Simulation::addPeriodicICLayerInX\(\)](#), [ICSetter::addPlaneWave1D\(\)](#), [Simulation::advanceToTime\(\)](#), [TimeEvolution::c](#), [errorKill\(\)](#), [OutputManager::generateOutputFolder\(\)](#), [OutputManager::getSimCode\(\)](#), [Simulation::icsettings](#), [Simulation::initializeCVOdeObject\(\)](#), [Simulation::initializePatchwork\(\)](#), [nonlinear1DProp\(\)](#), [Simulation::outAllFieldData\(\)](#), [Simulation::outputManager](#), [OutputManager::set_outputStyle\(\)](#), [Simulation::setDiscreteDimensionsOfLattice\(\)](#), [Simulation::setPhysicalDimensionsOfLattice\(\)](#), [Simulation::start\(\)](#), [TimeEvolution::TimeEvolver](#), and [timer\(\)](#).

Here is the call graph for this function:



6.26.2.2 Sim2D()

```
void Sim2D (
    const std::array< sunrealtype, 2 > CVodeTol,
    int const StencilOrder,
    const std::array< sunrealtype, 2 > phys_dims,
    const std::array< sunindextype, 2 > disc_dims,
```

```

const std::array< int, 2 > patches,
const bool periodic,
int * interactions,
const sunrealtype endTime,
const int numberOfSteps,
const std::string outputDirectory,
const int outputStep,
const char outputStyle,
const std::vector< planewave > & planes,
const std::vector< gaussian2D > & gaussians )

```

complete 2D [Simulation](#) function

Conduct the complete 2D simulation process

Definition at line 99 of file [SimulationFunctions.cpp](#).

```

00107                                     {
00108
00109     // MPI data
00110     double ts = MPI_Wtime();
00111     int myPrc = 0, nPrc = 0; // Get process rank and number of processes
00112     MPI_Comm_rank(MPI_COMM_WORLD,
00113                   &myPrc); // Return process rank, number \in [1,nPrc]
00114     MPI_Comm_size(MPI_COMM_WORLD,
00115                   &nPrc); // Return number of processes (communicator size)
00116
00117     // Check feasibility of the patchwork decomposition
00118     if (myPrc == 0) {
00119         if (nPrc != patches[0] * patches[1]) {
00120             errorKill(
00121                 "The number of MPI processes must match the number of patches.");
00122         }
00123     }
00124
00125     // Initialize the simulation, set up the cartesian communicator
00126     Simulation sim(patches[0], patches[1], 1, StencilOrder, periodic);
00127
00128     // Configure the patchwork
00129     sim.setPhysicalDimensionsOfLattice(phys_dims[0],
00130                                       phys_dims[1],
00131                                       1); // spacing of the lattice
00132     sim.setDiscreteDimensionsOfLattice(
00133         disc_dims[0], disc_dims[1], 1); // Spacing equivalence to points
00134     sim.initializePatchwork(patches[0], patches[1], 1);
00135
00136     // Add em-waves
00137     for (const auto &gauss : gaussians)
00138         sim.icsettings.addGauss2D(gauss.x0, gauss.axis, gauss.amp, gauss.phip,
00139                                   gauss.w0, gauss.zr, gauss.ph0, gauss.phA);
00140     for (const auto &plane : planes)
00141         sim.icsettings.addPlaneWave2D(plane.k, plane.p, plane.phi);
00142
00143     // Check that the patchwork is ready and set the initial conditions
00144     sim.start(); // Check if the lattice is set up, set initial field
00145                 // configuration
00146     sim.addPeriodicICLayerInXY(); // insure periodicity in propagation directions
00147
00148     // Initialize CVode with rel and abs tolerances
00149     sim.initializeCVODEobject(CVodeTol[0], CVodeTol[1]);
00150
00151     // Configure the time evolution function
00152     TimeEvolution::c = interactions;
00153     TimeEvolution::TimeEvolver = nonlinear2DProp;
00154
00155     // Configure the output
00156     sim.outputManager.generateOutputFolder(outputDirectory);
00157     if (!myPrc) {
00158         std::cout << "Simulation code: " << sim.outputManager.getSimCode()
00159                   << std::endl;
00160     }
00161     sim.outputManager.set_outputStyle(outputStyle);
00162
00163     //MPI_Barrier(MPI_COMM_WORLD); // insure correct output
00164     //sim.outAllFieldData(0); // output of initial state
00165     // Conduct the propagation in space and time
00166     for (int step = 1; step <= numberOfSteps; step++) {
00167         sim.advanceToTime(endTime / numberOfSteps * step);
00168         if (step % outputStep == 0) {
00169             MPI_Barrier(MPI_COMM_WORLD); // insure correct output
00170             sim.outAllFieldData(step);

```

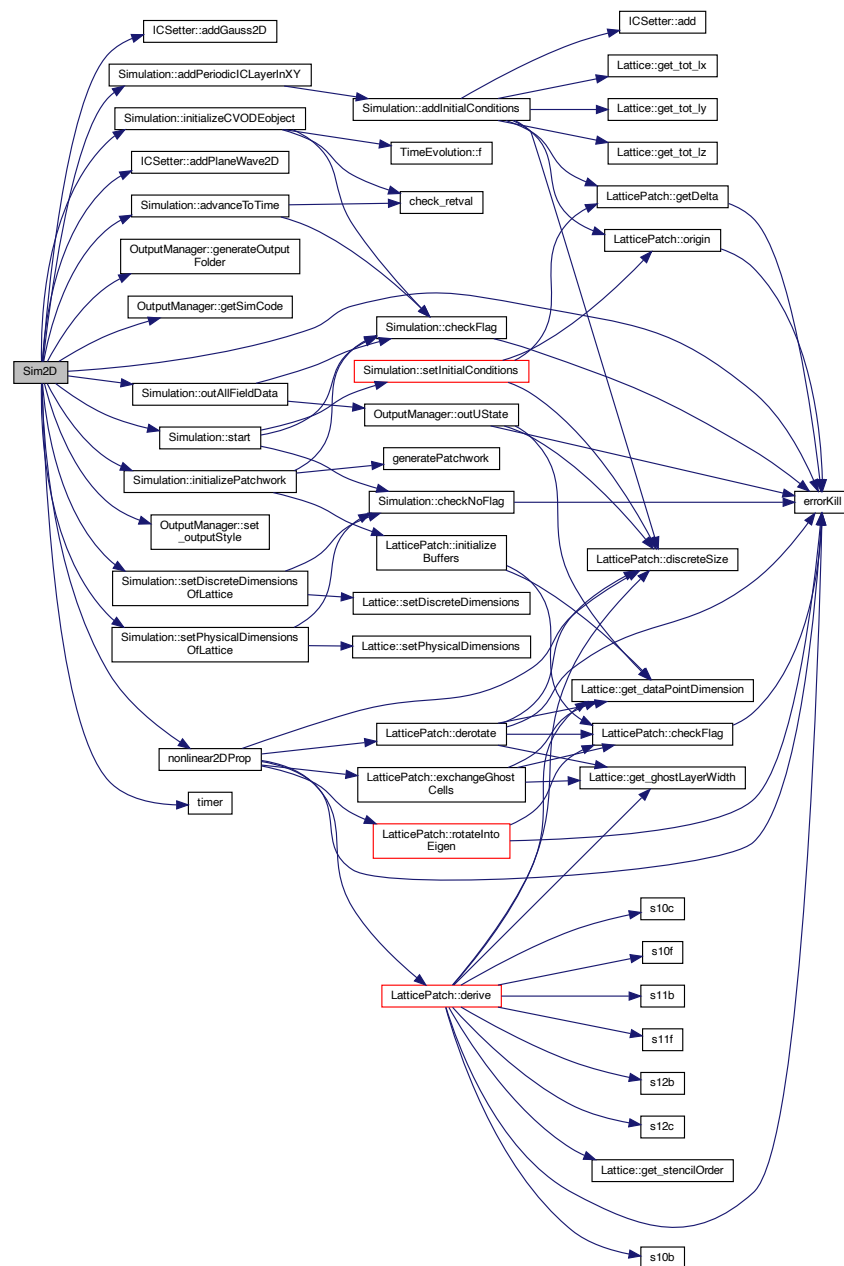
```

00171     }
00172     double tn = MPI_Wtime();
00173     if (!myPrc) {
00174         std::cout << "\rStep " << step << "\t\t" << std::flush;
00175         timer(ts, tn);
00176     }
00177 }
00178
00179 return;
00180 }

```

References [ICSetter::addGauss2D\(\)](#), [Simulation::addPeriodicCLayerInXY\(\)](#), [ICSetter::addPlaneWave2D\(\)](#), [Simulation::advanceToTime\(\)](#), [TimeEvolution::c](#), [errorKill\(\)](#), [OutputManager::generateOutputFolder\(\)](#), [OutputManager::getSimCode\(\)](#), [Simulation::icsettings](#), [Simulation::initializeCVODEobject\(\)](#), [Simulation::initializePatchwork\(\)](#), [nonlinear2DProp\(\)](#), [Simulation::outAllFieldData\(\)](#), [Simulation::outputManager](#), [OutputManager::set_outputStyle\(\)](#), [Simulation::setDiscreteDimensionsOfLattice\(\)](#), [Simulation::setPhysicalDimensionsOfLattice\(\)](#), [Simulation::start\(\)](#), [TimeEvolution::TimeEvolver](#), and [timer\(\)](#).

Here is the call graph for this function:



6.26.2.3 Sim3D()

```
void Sim3D (
    const std::array< sunrealtype, 2 > CNodeTol,
    const int StencilOrder,
    const std::array< sunrealtype, 3 > phys_dims,
    const std::array< sunindextype, 3 > disc_dims,
    const std::array< int, 3 > patches,
    const bool periodic,
    int * interactions,
    const sunrealtype endTime,
    const int numberOfSteps,
    const std::string outputDirectory,
    const int outputStep,
    const char outputStyle,
    const std::vector< planewave > & planes,
    const std::vector< gaussian3D > & gaussians )
```

complete 3D [Simulation](#) function

Conduct the complete 3D simulation process

Definition at line 183 of file [SimulationFunctions.cpp](#).

```
00191                                     {
00192
00193     // MPI data
00194     double ts = MPI_Wtime();
00195     int myPrc = 0, nPrc = 0; // Get process rank and number of process
00196     MPI_Comm_rank(MPI_COMM_WORLD,
00197                   &myPrc); // rank of the process inside the world communicator
00198     MPI_Comm_size(MPI_COMM_WORLD,
00199                   &nPrc); // Size of the communicator is the number of processes
00200
00201     // Check feasibility of the patchwork decomposition
00202     if (myPrc == 0) {
00203         if (nPrc != patches[0] * patches[1] * patches[2]) {
00204             errorKill(
00205                 "The number of MPI processes must match the number of patches.");
00206         }
00207         /*
00208         if ( ( disc_dims[0] / patches[0] != disc_dims[1] / patches[1] ) |
00209             ( disc_dims[0] / patches[0] != disc_dims[2] / patches[2] ) ) {
00210             std::clog
00211                 « "\nWarning: Patches should be cubic in terms of the lattice "
00212                 "points for the computational efficiency of larger simulations.\n";
00213         }
00214         */
00215     }
00216
00217     // Initialize the simulation, set up the cartesian communicator
00218     Simulation sim(patches[0], patches[1], patches[2],
00219                   StencilOrder, periodic); // Simulation object with slicing
00220
00221     // Create the SUNContext object associated with the thread of execution
00222     sim.setPhysicalDimensionsOfLattice(phys_dims[0], phys_dims[1],
00223                                       phys_dims[2]); // spacing of the box
00224     sim.setDiscreteDimensionsOfLattice(
00225         disc_dims[0], disc_dims[1],
00226         disc_dims[2]); // Spacing equivalence to points
00227     sim.initializePatchwork(patches[0], patches[1], patches[2]);
00228
00229     // Add em-waves
00230     for (const auto &plane : planes)
00231         sim.icsettings.addPlaneWave3D(plane.k, plane.p, plane.phi);
00232     for (const auto &gauss : gaussians)
00233         sim.icsettings.addGauss3D(gauss.x0, gauss.axis, gauss.amp, gauss.phip,
00234                                   gauss.w0, gauss.zr, gauss.ph0, gauss.phA);
00235
00236     // Check that the patchwork is ready and set the initial conditions
00237     sim.start();
00238 }
```

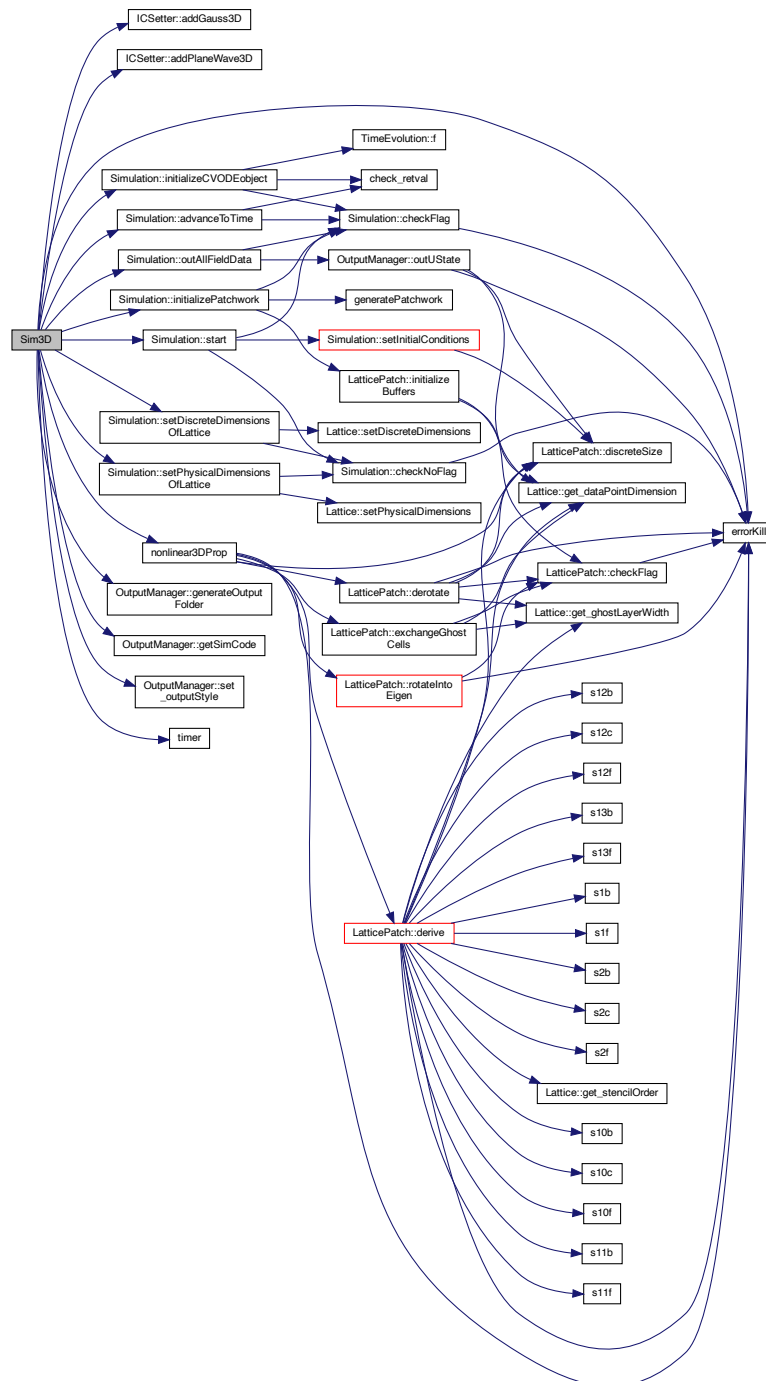
```

00239 // Initialize CNode with abs and rel tolerances
00240 sim.initializeCNodeObject(CNodeTol[0], CNodeTol[1]);
00241
00242 // Configure the time evolution function
00243 TimeEvolution::c = interactions;
00244 TimeEvolution::TimeEvolver = nonlinear3DProp;
00245
00246 // Configure the output
00247 sim.outputManager.generateOutputFolder(outputDirectory);
00248 if (!myProc) {
00249     std::cout << "Simulation code: " << sim.outputManager.getSimCode()
00250         << std::endl;
00251 }
00252 sim.outputManager.set_outputStyle(outputStyle);
00253
00254 //MPI_Barrier(MPI_COMM_WORLD); // insure correct output
00255 //sim.outAllFieldData(0); // output of initial state
00256 // Conduct the propagation in space and time
00257 for (int step = 1; step <= numberOfSteps; step++) {
00258     sim.advanceToTime(endTime / numberOfSteps * step);
00259     if (step % outputStep == 0) {
00260         MPI_Barrier(MPI_COMM_WORLD); // insure correct output
00261         sim.outAllFieldData(step);
00262     }
00263     double tn = MPI_Wtime();
00264     if (!myProc) {
00265         std::cout << "\rStep " << step << "\t\t" << std::flush;
00266         timer(ts, tn);
00267     }
00268 }
00269 return;
00270 }

```

References [ICSetter::addGauss3D\(\)](#), [ICSetter::addPlaneWave3D\(\)](#), [Simulation::advanceToTime\(\)](#), [TimeEvolution::c](#), [errorKill\(\)](#), [OutputManager::generateOutputFolder\(\)](#), [OutputManager::getSimCode\(\)](#), [Simulation::icsettings](#), [Simulation::initializeCNodeObject\(\)](#), [Simulation::initializePatchwork\(\)](#), [nonlinear3DProp\(\)](#), [Simulation::outAllFieldData\(\)](#), [Simulation::outputManager](#), [OutputManager::set_outputStyle\(\)](#), [Simulation::setDiscreteDimensionsOfLattice\(\)](#), [Simulation::setPhysicalDimensionsOfLattice\(\)](#), [Simulation::start\(\)](#), [TimeEvolution::TimeEvolver](#), and [timer\(\)](#).

Here is the call graph for this function:



6.26.2.4 timer()

```

void timer (
    double & t1,
    double & t2 ) [inline]
  
```

MPI timer function.

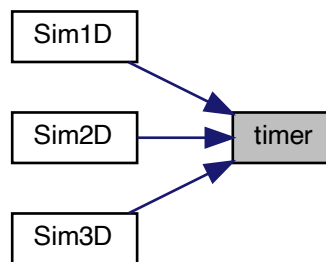
Calculate and print the total simulation time

Definition at line 10 of file [SimulationFunctions.cpp](#).

```
00010 {
00011     printf("Elapsed time:  %fs\n", (t2 - t1));
00012 }
```

Referenced by [Sim1D\(\)](#), [Sim2D\(\)](#), and [Sim3D\(\)](#).

Here is the caller graph for this function:



6.27 SimulationFunctions.h

[Go to the documentation of this file.](#)

```
00001 //////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
00002 /// @file SimulationFunctions.h
00003 /// @brief Full simulation functions for 1D, 2D, and 3D used in main.cpp
00004 //////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
00005
00006 #pragma once
00007
00008 // math
00009 #include <cmath>
00010
00011 // project subfile headers
00012 #include "LatticePatch.h"
00013 #include "SimulationClass.h"
00014 #include "TimeEvolutionFunctions.h"
00015
00016 /***** EM-wave structures *****/
00017
00018 /// plane wave structure
00019 struct planewave {
00020     std::array<sunrealtype, 3> k; /**< wavevector (normalized to \f$ 1/\lambda \f$) */
00021     std::array<sunrealtype, 3> p; /**< amplitude & polarization vector */
00022     std::array<sunrealtype, 3> phi; /**< phase shift */
00023 };
00024
00025 /// 1D Gaussian wave structure
00026 struct gaussian1D {
00027     std::array<sunrealtype, 3> k; /**< wavevector (normalized to \f$ 1/\lambda \f$) */
00028     std::array<sunrealtype, 3> p; /**< amplitude & polarization vector */
00029     std::array<sunrealtype, 3> x0; /**< shift from origin */
00030     sunrealtype phig; /**< width */
00031     std::array<sunrealtype, 3> phi; /**< phase shift */
00032 };
00033
00034 /// 2D Gaussian wave structure
00035 struct gaussian2D {
00036     std::array<sunrealtype, 3> x0; /**< center */
```

```

00037 std::array<sunrealtype, 3> axis; /**< direction from where it comes */
00038 sunrealtype amp; /**< amplitude */
00039 sunrealtype phip; /**< polarization rotation */
00040 sunrealtype w0; /**< taille */
00041 sunrealtype zr; /**< Rayleigh length */
00042 sunrealtype ph0; /**< beam center */
00043 sunrealtype phA; /**< beam length */
00044 };
00045
00046 /// 3D Gaussian wave structure
00047 struct gaussian3D {
00048     std::array<sunrealtype, 3> x0; /**< center */
00049     std::array<sunrealtype, 3> axis; /**< direction from where it comes */
00050     sunrealtype amp; /**< amplitude */
00051     sunrealtype phip; /**< polarization rotation */
00052     sunrealtype w0; /**< taille */
00053     sunrealtype zr; /**< Rayleigh length */
00054     sunrealtype ph0; /**< beam center */
00055     sunrealtype phA; /**< beam length */
00056 };
00057
00058 /******* simulation function declarations *****/
00059
00060 /// complete 1D Simulation function
00061 void Sim1D(const std::array<sunrealtype,2>, const int, const sunrealtype,
00062     const sunindextype, const bool, int *, const sunrealtype, const int,
00063     const std::string, const int, const char,
00064     const std::vector<planewave> &,
00065     const std::vector<gaussian1D> &);
00066 /// complete 2D Simulation function
00067 void Sim2D(const std::array<sunrealtype,2>, const int,
00068     const std::array<sunrealtype,2>,
00069     const std::array<sunindextype,2>, const std::array<int,2>,
00070     const bool, int *,
00071     const sunrealtype, const int, const std::string,
00072     const int, const char,
00073     const std::vector<planewave> &, const std::vector<gaussian2D> &);
00074 /// complete 3D Simulation function
00075 void Sim3D(const std::array<sunrealtype,2>, const int,
00076     const std::array<sunrealtype,3>,
00077     const std::array<sunindextype,3>, const std::array<int,3>,
00078     const bool, int *,
00079     const sunrealtype, const int, const std::string,
00080     const int, const char,
00081     const std::vector<planewave> &, const std::vector<gaussian3D> &);
00082
00083 /// MPI timer function
00084 void timer(double &, double &);

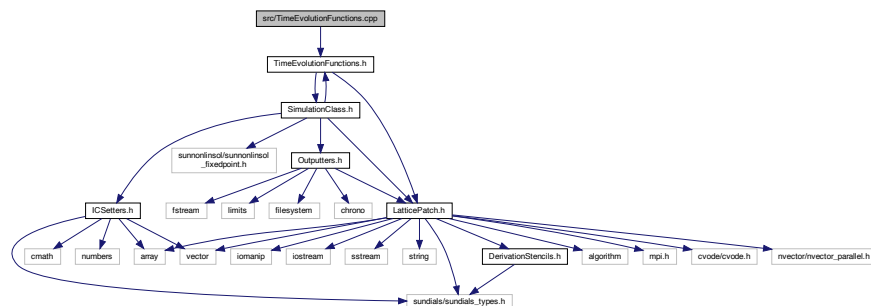
```

6.28 src/TimeEvolutionFunctions.cpp File Reference

Implementation of functions to propagate data vectors in time according to Maxwell's equations, and various orders in the HE weak-field expansion.

```
#include "TimeEvolutionFunctions.h"
```

Include dependency graph for TimeEvolutionFunctions.cpp:



Functions

- void [linear1DProp](#) ([LatticePatch](#) *data, [N_Vector](#) u, [N_Vector](#) udot, int *c)
only under-the-hood-callable Maxwell propagation in 1D;
- void [nonlinear1DProp](#) ([LatticePatch](#) *data, [N_Vector](#) u, [N_Vector](#) udot, int *c)
nonlinear 1D HE propagation
- void [linear2DProp](#) ([LatticePatch](#) *data, [N_Vector](#) u, [N_Vector](#) udot, int *c)
only under-the-hood-callable Maxwell propagation in 2D
- void [nonlinear2DProp](#) ([LatticePatch](#) *data, [N_Vector](#) u, [N_Vector](#) udot, int *c)
nonlinear 2D HE propagation
- void [linear3DProp](#) ([LatticePatch](#) *data, [N_Vector](#) u, [N_Vector](#) udot, int *c)
only under-the-hood-callable Maxwell propagation in 3D
- void [nonlinear3DProp](#) ([LatticePatch](#) *data, [N_Vector](#) u, [N_Vector](#) udot, int *c)
nonlinear 3D HE propagation

6.28.1 Detailed Description

Implementation of functions to propagate data vectors in time according to Maxwell's equations, and various orders in the HE weak-field expansion.

Definition in file [TimeEvolutionFunctions.cpp](#).

6.28.2 Function Documentation

6.28.2.1 [linear1DProp\(\)](#)

```
void linear1DProp (
    LatticePatch * data,
    N\_Vector u,
    N\_Vector udot,
    int * c )
```

only under-the-hood-callable Maxwell propagation in 1D;

Maxwell propagation function for 1D – only for reference.

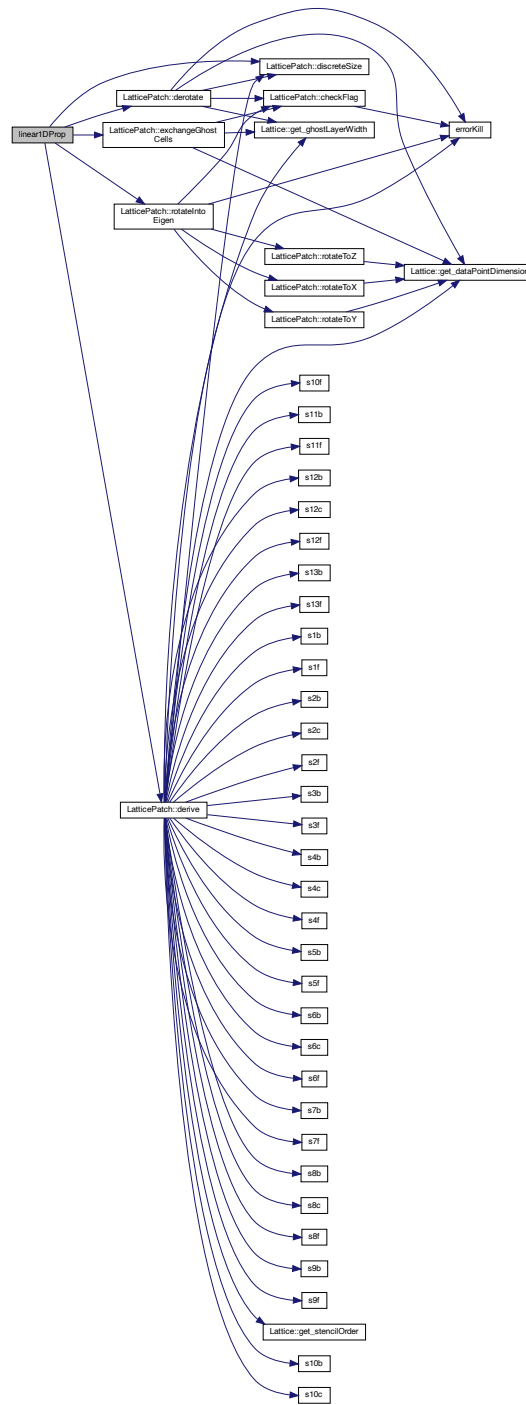
Definition at line 43 of file [TimeEvolutionFunctions.cpp](#).

```
00043 {
00044
00045     // pointers to temporal and spatial derivative data
00046     sunrealtype *duData = data->duData;
00047     sunrealtype *dxData = data->buffData[1 - 1];
00048
00049     // sequence along any dimension according to the scheme:
00050     data->exchangeGhostCells(1); // -> exchange halos
00051     data->rotateIntoEigen(
00052         1); // -> rotate all data to prepare derivative operation
00053     data->derive(1); // -> perform derivative approximation operation on it
00054     data->derotate(
00055         1, dxData); // -> derotate derived data for ensuing time-evolution
00056
00057     const sunindextype totalNP = data->discreteSize();
00058     sunindextype pp = 0;
00059     for (sunindextype i = 0; i < totalNP; i++) {
00060         pp = i * 6;
```

```
00061      /*
00062      simple vacuum Maxwell equations for the temporal derivatives using the
00063      spatial derivative only in x-direction without polarization or
00064      magnetization terms
00065      */
00066      duData[pp + 0] = 0;
00067      duData[pp + 1] = -dxData[pp + 5];
00068      duData[pp + 2] = dxData[pp + 4];
00069      duData[pp + 3] = 0;
00070      duData[pp + 4] = dxData[pp + 2];
00071      duData[pp + 5] = -dxData[pp + 1];
00072  }
00073 }
```

References [LatticePatch::buffData](#), [LatticePatch::derive\(\)](#), [LatticePatch::derotate\(\)](#), [LatticePatch::discreteSize\(\)](#), [LatticePatch::duData](#), [LatticePatch::exchangeGhostCells\(\)](#), and [LatticePatch::rotateIntoEigen\(\)](#).

Here is the call graph for this function:



6.28.2.2 linear2DProp()

```
void linear2DProp (
    LatticePatch * data,
```

```

    N_Vector u,
    N_Vector udot,
    int * c )

```

only under-the-hood-callable Maxwell propagation in 2D

Maxwell propagation function for 2D – only for reference.

Definition at line 286 of file [TimeEvolutionFunctions.cpp](#).

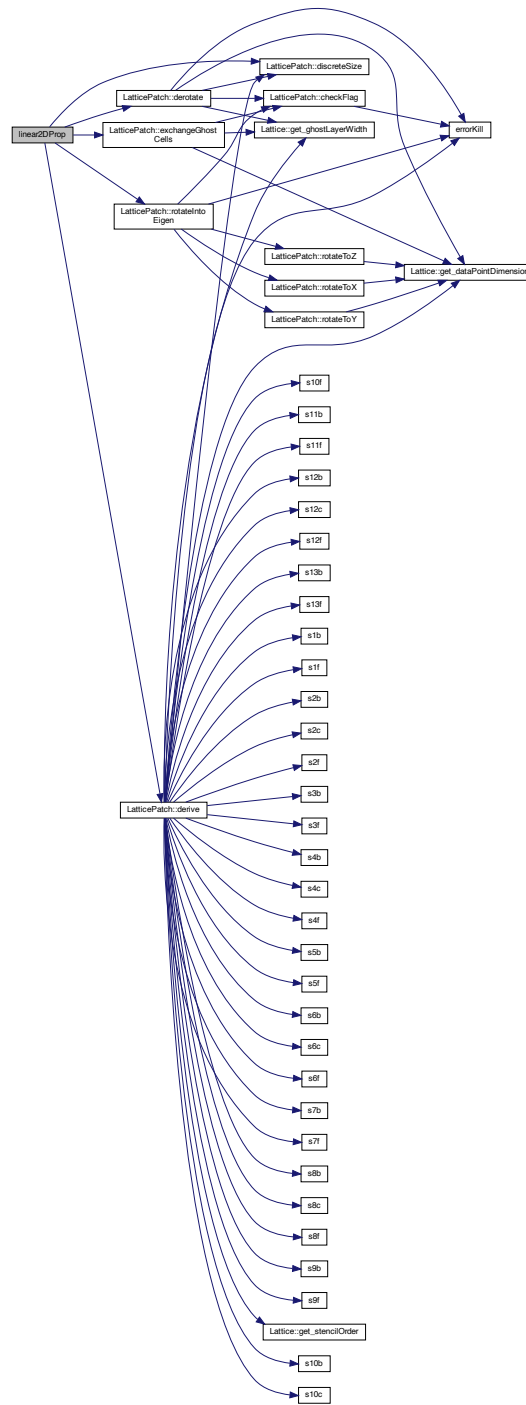
```

00286                                     {
00287
00288     sunrealtype *duData = data->duData;
00289     sunrealtype *dxData = data->buffData[1 - 1];
00290     sunrealtype *dyData = data->buffData[2 - 1];
00291
00292     data->exchangeGhostCells(1);
00293     data->rotateIntoEigen(1);
00294     data->derive(1);
00295     data->derotate(1, dxData);
00296     data->exchangeGhostCells(2);
00297     data->rotateIntoEigen(2);
00298     data->derive(2);
00299     data->derotate(2, dyData);
00300
00301     const sunindextype totalNP = data->discreteSize();
00302     sunindextype pp = 0;
00303     for (sunindextype i = 0; i < totalNP; i++) {
00304         pp = i * 6;
00305         duData[pp + 0] = dyData[pp + 5];
00306         duData[pp + 1] = -dxData[pp + 5];
00307         duData[pp + 2] = -dyData[pp + 3] + dxData[pp + 4];
00308         duData[pp + 3] = -dyData[pp + 2];
00309         duData[pp + 4] = dxData[pp + 2];
00310         duData[pp + 5] = dyData[pp + 0] - dxData[pp + 1];
00311     }
00312 }

```

References [LatticePatch::buffData](#), [LatticePatch::derive\(\)](#), [LatticePatch::derotate\(\)](#), [LatticePatch::discreteSize\(\)](#), [LatticePatch::duData](#), [LatticePatch::exchangeGhostCells\(\)](#), and [LatticePatch::rotateIntoEigen\(\)](#).

Here is the call graph for this function:



6.28.2.3 linear3DProp()

```
void linear3DProp (
    LatticePatch * data,
```

```

    N_Vector u,
    N_Vector udot,
    int * c )

```

only under-the-hood-callable Maxwell propagation in 3D

Maxwell propagation function for 3D – only for reference.

Definition at line 494 of file [TimeEvolutionFunctions.cpp](#).

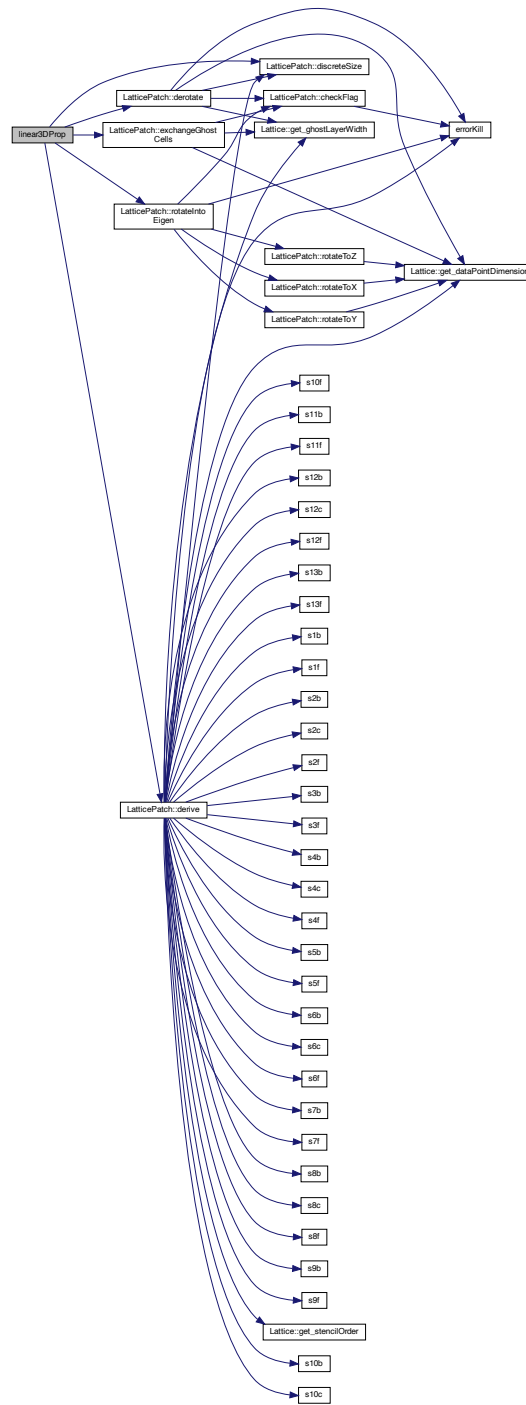
```

00494
00495
00496     sunrealtype *duData = data->duData;
00497     sunrealtype *dxData = data->buffData[1 - 1];
00498     sunrealtype *dyData = data->buffData[2 - 1];
00499     sunrealtype *dzData = data->buffData[3 - 1];
00500
00501     data->exchangeGhostCells(1);
00502     data->rotateIntoEigen(1);
00503     data->derive(1);
00504     data->derotate(1, dxData);
00505     data->exchangeGhostCells(2);
00506     data->rotateIntoEigen(2);
00507     data->derive(2);
00508     data->derotate(2, dyData);
00509     data->exchangeGhostCells(3);
00510     data->rotateIntoEigen(3);
00511     data->derive(3);
00512     data->derotate(3, dzData);
00513
00514     const sunindextype totalNP = data->discreteSize();
00515     sunindextype pp = 0;
00516     for (sunindextype i = 0; i < totalNP; i++) {
00517         pp = i * 6;
00518         duData[pp + 0] = dyData[pp + 5] - dzData[pp + 4];
00519         duData[pp + 1] = dzData[pp + 3] - dxData[pp + 5];
00520         duData[pp + 2] = dxData[pp + 4] - dyData[pp + 3];
00521         duData[pp + 3] = -dyData[pp + 2] + dzData[pp + 1];
00522         duData[pp + 4] = -dzData[pp + 0] + dxData[pp + 2];
00523         duData[pp + 5] = -dxData[pp + 1] + dyData[pp + 0];
00524     }
00525 }

```

References [LatticePatch::buffData](#), [LatticePatch::derive\(\)](#), [LatticePatch::derotate\(\)](#), [LatticePatch::discreteSize\(\)](#), [LatticePatch::duData](#), [LatticePatch::exchangeGhostCells\(\)](#), and [LatticePatch::rotateIntoEigen\(\)](#).

Here is the call graph for this function:



6.28.2.4 nonlinear1DProp()

```

void nonlinear1DProp (
    LatticePatch * data,

```

```

    N_Vector u,
    N_Vector udot,
    int * c )

```

nonlinear 1D HE propagation

HE propagation function for 1D.

Definition at line 76 of file [TimeEvolutionFunctions.cpp](#).

```

00076                                                                 {
00077
00078     // NVector pointers to provided field values and their temp. derivatives
00079     sunrealtype *udata = NV_DATA_P(u),
00080                 *dudata = NV_DATA_P(udot);
00081
00082     // pointer to spatial derivatives via patch data
00083     sunrealtype *dxData = data->buffData[1 - 1];
00084
00085     // same sequence as in the linear case
00086     data->exchangeGhostCells(1);
00087     data->rotateIntoEigen(1);
00088     data->derive(1);
00089     data->derotate(1, dxData);
00090
00091     /*
00092     F and G are nonzero in the nonlinear case,
00093     polarization and magnetization derivatives
00094     w.r.t. E- and B-field go into the e.o.m.
00095     */
00096     static sunrealtype f, g; // em field invariants F, G
00097     // derivatives of HE Lagrangian w.r.t. field invariants
00098     static sunrealtype lf, lff, lfg, lg, lgg;
00099     // matrix to hold derivatives of polarization and magnetization
00100     static std::array<sunrealtype, 21> JMM;
00101     // array to hold E^2 and B^2 components
00102     static std::array<sunrealtype, 6> Quad;
00103     // array to hold intermediate temp. derivatives of E and B
00104     static std::array<sunrealtype, 6> h;
00105     // determinant needed for explicit matrix inversion
00106     static sunrealtype detC = nan("0x12345");
00107
00108     // number of points in the patch
00109     const sunindextype totalNP = data->discreteSize();
00110     #pragma omp parallel for default(none) \
00111     private(JMM, Quad, h, detC) \
00112     shared(totalNP, c, f, g, lf, lff, lfg, lg, lgg, udata, dudata, dxData) \
00113     schedule(static)
00114     for (sunindextype pp = 0; pp < totalNP * 6;
00115          pp += 6) { // loop over all 6dim points in the patch
00116         // em field Lorentz invariants F and G
00117         f = 0.5 * ((Quad[0] = udata[pp] * udata[pp]) +
00118                  (Quad[1] = udata[pp + 1] * udata[pp + 1]) +
00119                  (Quad[2] = udata[pp + 2] * udata[pp + 2]) -
00120                  (Quad[3] = udata[pp + 3] * udata[pp + 3]) -
00121                  (Quad[4] = udata[pp + 4] * udata[pp + 4]) -
00122                  (Quad[5] = udata[pp + 5] * udata[pp + 5]));
00123         g = udata[pp] * udata[pp + 3] + udata[pp + 1] * udata[pp + 4] +
00124            udata[pp + 2] * udata[pp + 5];
00125         // process/expansion order and corresponding derivative values of L
00126         // w.r.t. F, G
00127         switch (*c) {
00128             case 0: // linear Maxwell vacuum
00129                 lf = 0;
00130                 lff = 0;
00131                 lfg = 0;
00132                 lg = 0;
00133                 lgg = 0;
00134                 break;
00135             case 1: // only 4-photon processes
00136                 lf = 0.000206527095658582755255648 * f;
00137                 lff = 0.000206527095658582755255648;
00138                 lfg = 0;
00139                 lg = 0.0003614224174025198216973841 * g;
00140                 lgg = 0.0003614224174025198216973841;
00141                 break;
00142             case 2: // only 6-photon processes
00143                 lf = 0.000354046449700427580438254 * f * f +
00144                    0.000191775160254398272737387 * g * g;
00145                 lff = 0.0007080928994008551608765075 * f;
00146                 lfg = 0.0003835503205087965454747749 * g;
00147                 lg = 0.0003835503205087965454747749 * f * g;
00148                 lgg = 0.0003835503205087965454747749 * f;
00149                 break;

```



```

00150     case 3: // 4- and 6-photon processes
00151         lf = (0.000206527095658582755255648 + 0.000354046449700427580438254 * f) *
00152             f +
00153             0.000191775160254398272737387 * g * g;
00154         lff = 0.000206527095658582755255648 + 0.0007080928994008551608765075 * f;
00155         lfg = 0.0003835503205087965454747749 * g;
00156         lg = (0.0003614224174025198216973841 +
00157             0.0003835503205087965454747749 * f) *
00158             g;
00159         lgg = 0.0003614224174025198216973841 + 0.0003835503205087965454747749 * f;
00160         break;
00161     default:
00162         errorKill(
00163             "You need to specify a correct order in the weak-field expansion.");
00164     }
00165
00166     // derivatives of polarization and magnetization w.r.t. E and B
00167     // Jpx(Ex)
00168     JMM[0] = lf + lff * Quad[0] +
00169         udata[3 + pp] * (2 * lfg * udata[pp] + lgg * udata[3 + pp]);
00170     // Jpx(Ey)
00171     JMM[1] =
00172         lff * udata[pp] * udata[1 + pp] + lfg * udata[1 + pp] * udata[3 + pp] +
00173         lfg * udata[pp] * udata[4 + pp] + lgg * udata[3 + pp] * udata[4 + pp];
00174     // Jpy(Ey)
00175     JMM[2] = lf + lff * Quad[1] +
00176         udata[4 + pp] * (2 * lfg * udata[1 + pp] + lgg * udata[4 + pp]);
00177     // Jpx(Ez) = Jpz(Ex)
00178     JMM[3] =
00179         lff * udata[pp] * udata[2 + pp] + lfg * udata[2 + pp] * udata[3 + pp] +
00180         lfg * udata[pp] * udata[5 + pp] + lgg * udata[3 + pp] * udata[5 + pp];
00181     // Jpy(Ez) = Jpz(Ey)
00182     JMM[4] = lff * udata[1 + pp] * udata[2 + pp] +
00183         lfg * udata[2 + pp] * udata[4 + pp] +
00184         lfg * udata[1 + pp] * udata[5 + pp] +
00185         lgg * udata[4 + pp] * udata[5 + pp];
00186     // Jpz(Ez)
00187     JMM[5] = lf + lff * Quad[2] +
00188         udata[5 + pp] * (2 * lfg * udata[2 + pp] + lgg * udata[5 + pp]);
00189     // Jpx(Bx) = Jmx(Ex)
00190     JMM[6] = lg + lfg * (Quad[0] - Quad[3 + 0]) +
00191         (-lff + lgg) * udata[pp] * udata[3 + pp];
00192     // Jpy(Bx) = Jmx(Ey)
00193     JMM[7] = -(udata[3 + pp] * (lff * udata[1 + pp] + lfg * udata[4 + pp])) +
00194         udata[pp] * (lfg * udata[1 + pp] + lgg * udata[4 + pp]);
00195     // Jpz(Bx) = Jmx(Ez)
00196     JMM[8] = -(udata[3 + pp] * (lff * udata[2 + pp] + lfg * udata[5 + pp])) +
00197         udata[pp] * (lfg * udata[2 + pp] + lgg * udata[5 + pp]);
00198     // Jmx(Bx)
00199     JMM[9] = -lf + lgg * Quad[0] +
00200         udata[3 + pp] * (-2 * lfg * udata[pp] + lff * udata[3 + pp]);
00201     // Jpx(By) = Jmy(Ex)
00202     JMM[10] = udata[1 + pp] * (lfg * udata[pp] + lgg * udata[3 + pp]) -
00203         (lff * udata[pp] + lfg * udata[3 + pp]) * udata[4 + pp];
00204     // Jpy(By) = Jmy(Ey)
00205     JMM[11] = lg + lfg * (Quad[1] - Quad[4 + 0]) +
00206         (-lff + lgg) * udata[1 + pp] * udata[4 + pp];
00207     // Jpz(By) = Jmy(Ez)
00208     JMM[12] = -(udata[4 + pp] * (lff * udata[2 + pp] + lfg * udata[5 + pp])) +
00209         udata[1 + pp] * (lfg * udata[2 + pp] + lgg * udata[5 + pp]);
00210     // Jmx(By) = Jmy(Bx)
00211     JMM[13] = lgg * udata[pp] * udata[1 + pp] +
00212         lff * udata[3 + pp] * udata[4 + pp] -
00213         lfg * (udata[1 + pp] * udata[3 + pp] + udata[pp] * udata[4 + pp]);
00214     // Jmy(By)
00215     JMM[14] = -lf + lgg * Quad[1] +
00216         udata[4 + pp] * (-2 * lfg * udata[1 + pp] + lff * udata[4 + pp]);
00217     // Jmz(Ex) = Jpx(Bz)
00218     JMM[15] = udata[2 + pp] * (lfg * udata[pp] + lgg * udata[3 + pp]) -
00219         (lff * udata[pp] + lfg * udata[3 + pp]) * udata[5 + pp];
00220     // Jmz(Ey) = Jpy(Bz)
00221     JMM[16] = udata[2 + pp] * (lfg * udata[1 + pp] + lgg * udata[4 + pp]) -
00222         (lff * udata[1 + pp] + lfg * udata[4 + pp]) * udata[5 + pp];
00223     // Jpz(Bz) = Jmz(Ez)
00224     JMM[17] = lg + lfg * (Quad[2] - Quad[5 + 0]) +
00225         (-lff + lgg) * udata[2 + pp] * udata[5 + pp];
00226     // Jmz(Bx) = Jmx(Bz)
00227     JMM[18] = lgg * udata[pp] * udata[2 + pp] +
00228         lff * udata[3 + pp] * udata[5 + pp] -
00229         lfg * (udata[2 + pp] * udata[3 + pp] + udata[pp] * udata[5 + pp]);
00230     // Jmy(Bz) = Jmz(By)
00231     JMM[19] =
00232         lgg * udata[1 + pp] * udata[2 + pp] +
00233         lff * udata[4 + pp] * udata[5 + pp] -
00234         lfg * (udata[2 + pp] * udata[4 + pp] + udata[1 + pp] * udata[5 + pp]);
00235     // Jmz(Bz)
00236     JMM[20] = -lf + lgg * Quad[2] +

```

```

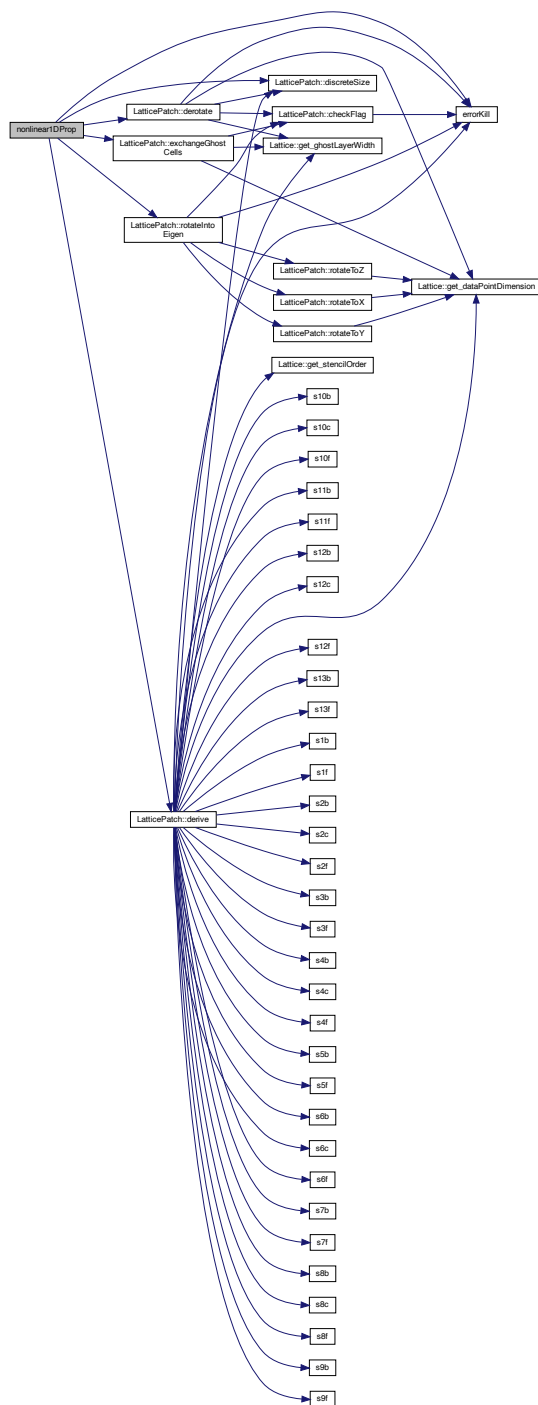
00237         udata[5 + pp] * (-2 * lfg * udata[2 + pp] + lff * udata[5 + pp]);
00238
00239     // apply Z
00240     // top block: -QJm(E)*E, Q-QJm(B)*B
00241     h[0] = 0;
00242     h[1] = dxData[pp] * JMM[15] + dxData[1 + pp] * JMM[16] +
00243           dxData[2 + pp] * JMM[17] + dxData[3 + pp] * JMM[18] +
00244           dxData[4 + pp] * JMM[19] + dxData[5 + pp] * (-1 + JMM[20]);
00245     h[2] = -(dxData[pp] * JMM[10]) - dxData[1 + pp] * JMM[11] -
00246           dxData[2 + pp] * JMM[12] - dxData[3 + pp] * JMM[13] +
00247           dxData[4 + pp] * (1 - JMM[14]) - dxData[5 + pp] * JMM[19];
00248     // bottom blocks: -Q*E
00249     h[3] = 0;
00250     h[4] = dxData[2 + pp];
00251     h[5] = -dxData[1 + pp];
00252     // (1+A)^-1 applies only to E components
00253     // -Jp(B)*B
00254     h[0] -= h[3] * JMM[6] + h[4] * JMM[10] + h[5] * JMM[15];
00255     h[1] -= h[3] * JMM[7] + h[4] * JMM[11] + h[5] * JMM[16];
00256     h[2] -= h[3] * JMM[8] + h[4] * JMM[12] + h[5] * JMM[17];
00257     // apply C^-1 explicitly, with C=1+Jp(E)
00258     dudata[pp + 0] =
00259         h[2] * (- (JMM[3] * (1 + JMM[2])) + JMM[1] * JMM[4]) +
00260         h[1] * (JMM[3] * JMM[4] - JMM[1] * (1 + JMM[5])) +
00261         h[0] * (1 - JMM[4] * JMM[4] + JMM[5] + JMM[2] * (1 + JMM[5]));
00262     dudata[pp + 1] =
00263         h[2] * (JMM[3] * JMM[1] - (1 + JMM[0]) * JMM[4]) +
00264         h[1] * (1 - JMM[3] * JMM[3] + JMM[5] + JMM[0] * (1 + JMM[5])) +
00265         h[0] * (JMM[4] * JMM[3] - JMM[1] * (1 + JMM[5]));
00266     dudata[pp + 2] =
00267         h[2] * (1 - JMM[1] * JMM[1] + JMM[2] + JMM[0] * (1 + JMM[2])) +
00268         h[1] * (JMM[1] * JMM[3] - (1 + JMM[0]) * JMM[4]) +
00269         h[0] * (-((1 + JMM[2]) * JMM[3]) + JMM[1] * JMM[4]);
00270     detC = // determinant of C
00271         -((1 + JMM[2]) * (-1 + JMM[3] * JMM[3])) +
00272         (JMM[3] * JMM[1] - JMM[4]) * JMM[4] + JMM[5] + JMM[2] * JMM[5] +
00273         JMM[0] * (1 + JMM[2] - JMM[4] * JMM[4] + (1 + JMM[2]) * JMM[5]) -
00274         JMM[1] * (- (JMM[4] * JMM[3]) + JMM[1] * (1 + JMM[5]));
00275     dudata[pp + 0] /= detC;
00276     dudata[pp + 1] /= detC;
00277     dudata[pp + 2] /= detC;
00278     dudata[pp + 3] = h[3];
00279     dudata[pp + 4] = h[4];
00280     dudata[pp + 5] = h[5];
00281 }
00282 return;
00283 }

```

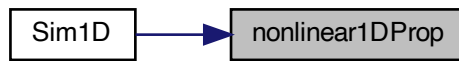
References [LatticePatch::buffData](#), [LatticePatch::derive\(\)](#), [LatticePatch::derotate\(\)](#), [LatticePatch::discreteSize\(\)](#), [errorKill\(\)](#), [LatticePatch::exchangeGhostCells\(\)](#), and [LatticePatch::rotateIntoEigen\(\)](#).

Referenced by [Sim1D\(\)](#).

Here is the call graph for this function:



Here is the caller graph for this function:



6.28.2.5 nonlinear2DProp()

```

void nonlinear2DProp (
    LatticePatch * data,
    N_Vector u,
    N_Vector udot,
    int * c )
  
```

nonlinear 2D HE propagation

HE propagation function for 2D.

Definition at line 315 of file [TimeEvolutionFunctions.cpp](#).

```

00315
00316
00317     sunrealtype *udata = NV_DATA_P(u),
00318               *dudata = NV_DATA_P(udot);
00319
00320     sunrealtype *dxData = data->buffData[1 - 1];
00321     sunrealtype *dyData = data->buffData[2 - 1];
00322
00323     data->exchangeGhostCells(1);
00324     data->rotateIntoEigen(1);
00325     data->derive(1);
00326     data->derotate(1, dxData);
00327     data->exchangeGhostCells(2);
00328     data->rotateIntoEigen(2);
00329     data->derive(2);
00330     data->derotate(2, dyData);
00331
00332     static sunrealtype f, g;
00333     static sunrealtype lf, lff, lfg, lg, lgg;
00334     static std::array<sunrealtype, 21> JMM;
00335     static std::array<sunrealtype, 6> Quad;
00336     static std::array<sunrealtype, 6> h;
00337     static sunrealtype detC;
00338
00339     const sunindextype totalNP = data->discreteSize();
00340     #pragma omp parallel for default(none) \
00341     private(JMM, Quad, h, detC) \
00342     shared(totalNP, c, f, g, lf, lff, lfg, lg, lgg, udata, dudata, \
00343            dxData, dyData) \
00344     schedule(static)
00345     for (sunindextype pp = 0; pp < totalNP * 6; pp += 6) {
00346         f = 0.5 * ((Quad[0] = udata[pp] * udata[pp]) +
00347                  (Quad[1] = udata[pp + 1] * udata[pp + 1]) +
00348                  (Quad[2] = udata[pp + 2] * udata[pp + 2]) -
00349                  (Quad[3] = udata[pp + 3] * udata[pp + 3]) -
00350                  (Quad[4] = udata[pp + 4] * udata[pp + 4]) -
00351                  (Quad[5] = udata[pp + 5] * udata[pp + 5]));
00352         g = udata[pp] * udata[pp + 3] + udata[pp + 1] * udata[pp + 4] +
00353             udata[pp + 2] * udata[pp + 5];
00354         switch (*c) {
00355         case 0:
00356             lf = 0;
00357             lff = 0;
  
```

```

00358     lfg = 0;
00359     lg = 0;
00360     lgg = 0;
00361     break;
00362 case 1:
00363     lf = 0.000206527095658582755255648 * f;
00364     lff = 0.000206527095658582755255648;
00365     lfg = 0;
00366     lg = 0.0003614224174025198216973841 * g;
00367     lgg = 0.0003614224174025198216973841;
00368     break;
00369 case 2:
00370     lf = 0.000354046449700427580438254 * f * f +
00371         0.000191775160254398272737387 * g * g;
00372     lff = 0.0007080928994008551608765075 * f;
00373     lfg = 0.0003835503205087965454747749 * g;
00374     lg = 0.0003835503205087965454747749 * f * g;
00375     lgg = 0.0003835503205087965454747749 * f;
00376     break;
00377 case 3:
00378     lf = (0.000206527095658582755255648 + 0.000354046449700427580438254 * f) *
00379         f +
00380         0.000191775160254398272737387 * g * g;
00381     lff = 0.000206527095658582755255648 + 0.000708092899400855160876508 * f;
00382     lfg = 0.0003835503205087965454747749 * g;
00383     lg = (0.000361422417402519821697384 + 0.000383550320508796545474775 * f) *
00384         g;
00385     lgg = 0.000361422417402519821697384 + 0.000383550320508796545474775 * f;
00386     break;
00387 default:
00388     errorKill(
00389         "You need to specify a correct order in the weak-field expansion.");
00390 }
00391
00392 JMM[0] = lf + lff * Quad[0] +
00393         udata[3 + pp] * (2 * lfg * udata[pp] + lgg * udata[3 + pp]);
00394 JMM[1] =
00395     lff * udata[pp] * udata[1 + pp] + lfg * udata[1 + pp] * udata[3 + pp] +
00396     lfg * udata[pp] * udata[4 + pp] + lgg * udata[3 + pp] * udata[4 + pp];
00397 JMM[2] = lf + lff * Quad[1] +
00398         udata[4 + pp] * (2 * lfg * udata[1 + pp] + lgg * udata[4 + pp]);
00399 JMM[3] =
00400     lff * udata[pp] * udata[2 + pp] + lfg * udata[2 + pp] * udata[3 + pp] +
00401     lfg * udata[pp] * udata[5 + pp] + lgg * udata[3 + pp] * udata[5 + pp];
00402 JMM[4] = lff * udata[1 + pp] * udata[2 + pp] +
00403         lfg * udata[2 + pp] * udata[4 + pp] +
00404         lfg * udata[1 + pp] * udata[5 + pp] +
00405         lgg * udata[4 + pp] * udata[5 + pp];
00406 JMM[5] = lf + lff * Quad[2] +
00407         udata[5 + pp] * (2 * lfg * udata[2 + pp] + lgg * udata[5 + pp]);
00408 JMM[6] = lg + lfg * (Quad[0] - Quad[3 + 0]) +
00409         (-lff + lgg) * udata[pp] * udata[3 + pp];
00410 JMM[7] = -(udata[3 + pp] * (lff * udata[1 + pp] + lfg * udata[4 + pp])) +
00411         udata[pp] * (lfg * udata[1 + pp] + lgg * udata[4 + pp]);
00412 JMM[8] = -(udata[3 + pp] * (lff * udata[2 + pp] + lfg * udata[5 + pp])) +
00413         udata[pp] * (lfg * udata[2 + pp] + lgg * udata[5 + pp]);
00414 JMM[9] = -lf + lgg * Quad[0] +
00415         udata[3 + pp] * (-2 * lfg * udata[pp] + lff * udata[3 + pp]);
00416 JMM[10] = udata[1 + pp] * (lfg * udata[pp] + lgg * udata[3 + pp]) -
00417         (lff * udata[pp] + lfg * udata[3 + pp]) * udata[4 + pp];
00418 JMM[11] = lg + lfg * (Quad[1] - Quad[4 + 0]) +
00419         (-lff + lgg) * udata[1 + pp] * udata[4 + pp];
00420 JMM[12] = -(udata[4 + pp] * (lff * udata[2 + pp] + lfg * udata[5 + pp])) +
00421         udata[1 + pp] * (lfg * udata[2 + pp] + lgg * udata[5 + pp]);
00422 JMM[13] = lgg * udata[pp] * udata[1 + pp] +
00423         lff * udata[3 + pp] * udata[4 + pp] -
00424         lfg * (udata[1 + pp] * udata[3 + pp] + udata[pp] * udata[4 + pp]);
00425 JMM[14] = -lf + lgg * Quad[1] +
00426         udata[4 + pp] * (-2 * lfg * udata[1 + pp] + lff * udata[4 + pp]);
00427 JMM[15] = udata[2 + pp] * (lfg * udata[pp] + lgg * udata[3 + pp]) -
00428         (lff * udata[pp] + lfg * udata[3 + pp]) * udata[5 + pp];
00429 JMM[16] = udata[2 + pp] * (lfg * udata[1 + pp] + lgg * udata[4 + pp]) -
00430         (lff * udata[1 + pp] + lfg * udata[4 + pp]) * udata[5 + pp];
00431 JMM[17] = lg + lfg * (Quad[2] - Quad[5 + 0]) +
00432         (-lff + lgg) * udata[2 + pp] * udata[5 + pp];
00433 JMM[18] = lgg * udata[pp] * udata[2 + pp] +
00434         lff * udata[3 + pp] * udata[5 + pp] -
00435         lfg * (udata[2 + pp] * udata[3 + pp] + udata[pp] * udata[5 + pp]);
00436 JMM[19] =
00437     lgg * udata[1 + pp] * udata[2 + pp] +
00438     lff * udata[4 + pp] * udata[5 + pp] -
00439     lfg * (udata[2 + pp] * udata[4 + pp] + udata[1 + pp] * udata[5 + pp]);
00440 JMM[20] = -lf + lgg * Quad[2] +
00441         udata[5 + pp] * (-2 * lfg * udata[2 + pp] + lff * udata[5 + pp]);
00442
00443 h[0] = 0;
00444 h[1] = dxData[pp] * JMM[15] + dxData[1 + pp] * JMM[16] +

```

```

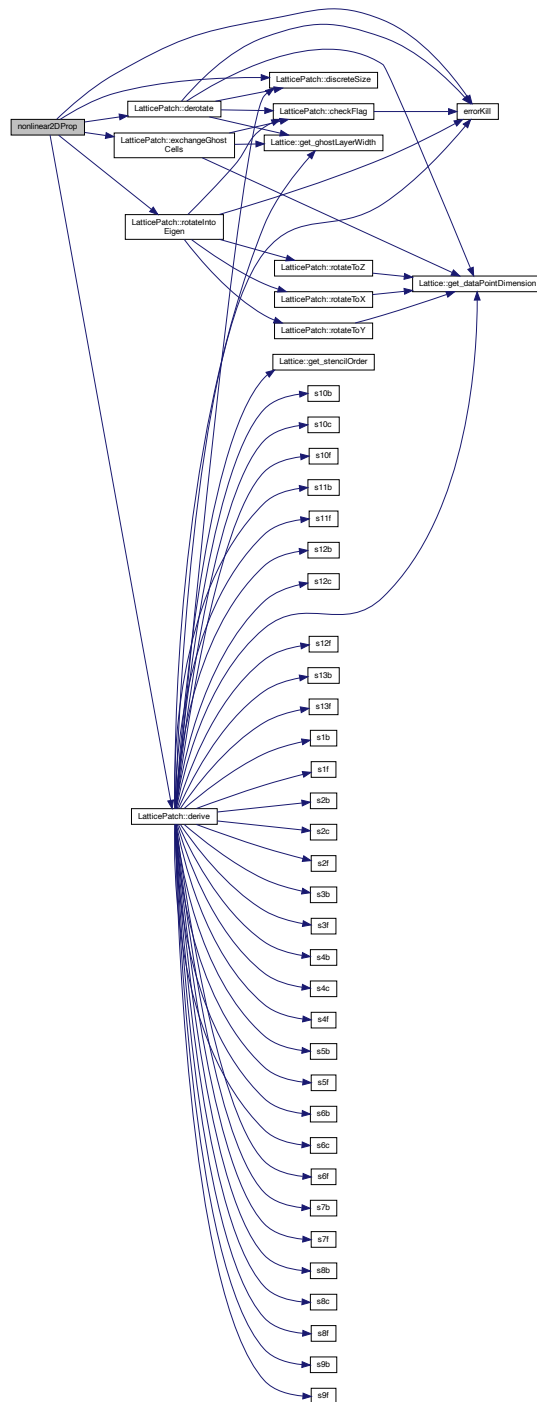
00445         dxData[2 + pp] * JMM[17] + dxData[3 + pp] * JMM[18] +
00446         dxData[4 + pp] * JMM[19] + dxData[5 + pp] * (-1 + JMM[20]));
00447     h[2] = -(dxData[pp] * JMM[10]) - dxData[1 + pp] * JMM[11] -
00448         dxData[2 + pp] * JMM[12] - dxData[3 + pp] * JMM[13] +
00449         dxData[4 + pp] * (1 - JMM[14]) - dxData[5 + pp] * JMM[19];
00450     h[3] = 0;
00451     h[4] = dxData[2 + pp];
00452     h[5] = -dxData[1 + pp];
00453     h[0] += -(dyData[pp] * JMM[15]) - dyData[1 + pp] * JMM[16] -
00454         dyData[2 + pp] * JMM[17] - dyData[3 + pp] * JMM[18] -
00455         dyData[4 + pp] * JMM[19] + dyData[5 + pp] * (1 - JMM[20]);
00456     h[1] += 0;
00457     h[2] += dyData[pp] * JMM[6] + dyData[1 + pp] * JMM[7] +
00458         dyData[2 + pp] * JMM[8] + dyData[3 + pp] * (-1 + JMM[9]) +
00459         dyData[4 + pp] * JMM[13] + dyData[5 + pp] * JMM[18];
00460     h[3] += -dyData[2 + pp];
00461     h[4] += 0;
00462     h[5] += dyData[pp];
00463     h[0] -= h[3] * JMM[6] + h[4] * JMM[10] + h[5] * JMM[15];
00464     h[1] -= h[3] * JMM[7] + h[4] * JMM[11] + h[5] * JMM[16];
00465     h[2] -= h[3] * JMM[8] + h[4] * JMM[12] + h[5] * JMM[17];
00466     dudata[pp + 0] =
00467         h[2] * (- (JMM[3] * (1 + JMM[2])) + JMM[1] * JMM[4]) +
00468         h[1] * (JMM[3] * JMM[4] - JMM[1] * (1 + JMM[5])) +
00469         h[0] * (1 - JMM[4] * JMM[4] + JMM[5] + JMM[2] * (1 + JMM[5]));
00470     dudata[pp + 1] =
00471         h[2] * (JMM[3] * JMM[1] - (1 + JMM[0]) * JMM[4]) +
00472         h[1] * (1 - JMM[3] * JMM[3] + JMM[5] + JMM[0] * (1 + JMM[5])) +
00473         h[0] * (JMM[4] * JMM[3] - JMM[1] * (1 + JMM[5]));
00474     dudata[pp + 2] =
00475         h[2] * (1 - JMM[1] * JMM[1] + JMM[2] + JMM[0] * (1 + JMM[2])) +
00476         h[1] * (JMM[1] * JMM[3] - (1 + JMM[0]) * JMM[4]) +
00477         h[0] * (-((1 + JMM[2]) * JMM[3]) + JMM[1] * JMM[4]);
00478     detC =
00479         -((1 + JMM[2]) * (-1 + JMM[3] * JMM[3])) +
00480         (JMM[3] * JMM[1] - JMM[4]) * JMM[4] + JMM[5] + JMM[2] * JMM[5] +
00481         JMM[0] * (1 + JMM[2] - JMM[4] * JMM[4] + (1 + JMM[2]) * JMM[5]) -
00482         JMM[1] * (- (JMM[4] * JMM[3]) + JMM[1] * (1 + JMM[5]));
00483     dudata[pp + 0] /= detC;
00484     dudata[pp + 1] /= detC;
00485     dudata[pp + 2] /= detC;
00486     dudata[pp + 3] = h[3];
00487     dudata[pp + 4] = h[4];
00488     dudata[pp + 5] = h[5];
00489 }
00490 return;
00491 }

```

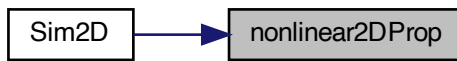
References [LatticePatch::buffData](#), [LatticePatch::derive\(\)](#), [LatticePatch::derotate\(\)](#), [LatticePatch::discreteSize\(\)](#), [errorKill\(\)](#), [LatticePatch::exchangeGhostCells\(\)](#), and [LatticePatch::rotateIntoEigen\(\)](#).

Referenced by [Sim2D\(\)](#).

Here is the call graph for this function:



Here is the caller graph for this function:



6.28.2.6 nonlinear3DProp()

```

void nonlinear3DProp (
    LatticePatch * data,
    N_Vector u,
    N_Vector udot,
    int * c )
  
```

nonlinear 3D HE propagation

HE propagation function for 3D.

Definition at line 528 of file [TimeEvolutionFunctions.cpp](#).

```

00528
00529
00530     sunrealtype *udata = NV_DATA_P(u),
00531     *dudata = NV_DATA_P(udot);
00532
00533     sunrealtype *dxData = data->buffData[1 - 1];
00534     sunrealtype *dyData = data->buffData[2 - 1];
00535     sunrealtype *dzData = data->buffData[3 - 1];
00536
00537     data->exchangeGhostCells(1);
00538     data->rotateIntoEigen(1);
00539     data->derive(1);
00540     data->derotate(1,dxData);
00541     data->exchangeGhostCells(2);
00542     data->rotateIntoEigen(2);
00543     data->derive(2);
00544     data->derotate(2,dyData);
00545     data->exchangeGhostCells(3);
00546     data->rotateIntoEigen(3);
00547     data->derive(3);
00548     data->derotate(3,dzData);
00549
00550     static sunrealtype f, g;
00551     static sunrealtype lf, lff, lfg, lg, lgg;
00552     static std::array<sunrealtype, 21> JMM;
00553     static std::array<sunrealtype, 6> Quad;
00554     static std::array<sunrealtype, 6> h;
00555     static sunrealtype detC = nan("0x12345");
00556
00557     const sunindextype totalNP = data->discreteSize();
00558     #pragma omp parallel for default(none) \
00559     private(JMM, Quad, h, detC) \
00560     shared(totalNP, c, f, g, lf, lff, lfg, lg, lgg, udata, dudata, \
00561           dxData, dyData, dzData) \
00562     schedule(static)
00563     for (sunindextype pp = 0; pp < totalNP * 6; pp += 6) {
00564         f = 0.5 * ((Quad[0] = udata[pp] * udata[pp]) +
00565                 (Quad[1] = udata[pp + 1] * udata[pp + 1]) +
00566                 (Quad[2] = udata[pp + 2] * udata[pp + 2]) -
00567                 (Quad[3] = udata[pp + 3] * udata[pp + 3]) -
00568                 (Quad[4] = udata[pp + 4] * udata[pp + 4]) -
00569                 (Quad[5] = udata[pp + 5] * udata[pp + 5]));
00570         g = udata[pp] * udata[pp + 3] + udata[pp + 1] * udata[pp + 4] +
  
```



```

00571         udata[pp + 2] * udata[pp + 5];
00572     switch (*c) {
00573     case 0:
00574         lf = 0;
00575         lff = 0;
00576         lfg = 0;
00577         lg = 0;
00578         lgg = 0;
00579         break;
00580     case 1:
00581         lf = 0.000206527095658582755255648 * f;
00582         lff = 0.000206527095658582755255648;
00583         lfg = 0;
00584         lg = 0.0003614224174025198216973841 * g;
00585         lgg = 0.0003614224174025198216973841;
00586         break;
00587     case 2:
00588         lf = 0.000354046449700427580438254 * f * f +
00589             0.000191775160254398272737387 * g * g;
00590         lff = 0.0007080928994008551608765075 * f;
00591         lfg = 0.0003835503205087965454747749 * g;
00592         lg = 0.0003835503205087965454747749 * f * g;
00593         lgg = 0.0003835503205087965454747749 * f;
00594         break;
00595     case 3:
00596         lf = (0.000206527095658582755255648 + 0.000354046449700427580438254 * f) *
00597             f +
00598             0.000191775160254398272737387 * g * g;
00599         lff = 0.000206527095658582755255648 + 0.000708092899400855160876508 * f;
00600         lfg = 0.0003835503205087965454747749 * g;
00601         lg = (0.000361422417402519821697384 + 0.000383550320508796545474775 * f) *
00602             g;
00603         lgg = 0.000361422417402519821697384 + 0.000383550320508796545474775 * f;
00604         break;
00605     default:
00606         errorKill(
00607             "You need to specify a correct order in the weak-field expansion.");
00608     }
00609
00610     JMM[0] = lf + lff * Quad[0] +
00611         udata[3 + pp] * (2 * lfg * udata[pp] + lgg * udata[3 + pp]);
00612     JMM[1] =
00613         lff * udata[pp] * udata[1 + pp] + lfg * udata[1 + pp] * udata[3 + pp] +
00614         lfg * udata[pp] * udata[4 + pp] + lgg * udata[3 + pp] * udata[4 + pp];
00615     JMM[2] = lf + lff * Quad[1] +
00616         udata[4 + pp] * (2 * lfg * udata[1 + pp] + lgg * udata[4 + pp]);
00617     JMM[3] =
00618         lff * udata[pp] * udata[2 + pp] + lfg * udata[2 + pp] * udata[3 + pp] +
00619         lfg * udata[pp] * udata[5 + pp] + lgg * udata[3 + pp] * udata[5 + pp];
00620     JMM[4] = lff * udata[1 + pp] * udata[2 + pp] +
00621         lfg * udata[2 + pp] * udata[4 + pp] +
00622         lfg * udata[1 + pp] * udata[5 + pp] +
00623         lgg * udata[4 + pp] * udata[5 + pp];
00624     JMM[5] = lf + lff * Quad[2] +
00625         udata[5 + pp] * (2 * lfg * udata[2 + pp] + lgg * udata[5 + pp]);
00626     JMM[6] = lg + lfg * (Quad[0] - Quad[3 + 0]) +
00627         (-lff + lgg) * udata[pp] * udata[3 + pp];
00628     JMM[7] = -(udata[3 + pp] * (lff * udata[1 + pp] + lfg * udata[4 + pp])) +
00629         udata[pp] * (lfg * udata[1 + pp] + lgg * udata[4 + pp]);
00630     JMM[8] = -(udata[3 + pp] * (lff * udata[2 + pp] + lfg * udata[5 + pp])) +
00631         udata[pp] * (lfg * udata[2 + pp] + lgg * udata[5 + pp]);
00632     JMM[9] = -lf + lgg * Quad[0] +
00633         udata[3 + pp] * (-2 * lfg * udata[pp] + lff * udata[3 + pp]);
00634     JMM[10] = udata[1 + pp] * (lfg * udata[pp] + lgg * udata[3 + pp]) -
00635         (lff * udata[pp] + lfg * udata[3 + pp]) * udata[4 + pp];
00636     JMM[11] = lg + lfg * (Quad[1] - Quad[4 + 0]) +
00637         (-lff + lgg) * udata[1 + pp] * udata[4 + pp];
00638     JMM[12] = -(udata[4 + pp] * (lff * udata[2 + pp] + lfg * udata[5 + pp])) +
00639         udata[1 + pp] * (lfg * udata[2 + pp] + lgg * udata[5 + pp]);
00640     JMM[13] = lgg * udata[pp] * udata[1 + pp] +
00641         lff * udata[3 + pp] * udata[4 + pp] -
00642         lfg * (udata[1 + pp] * udata[3 + pp] + udata[pp] * udata[4 + pp]);
00643     JMM[14] = -lf + lgg * Quad[1] +
00644         udata[4 + pp] * (-2 * lfg * udata[1 + pp] + lff * udata[4 + pp]);
00645     JMM[15] = udata[2 + pp] * (lfg * udata[pp] + lgg * udata[3 + pp]) -
00646         (lff * udata[pp] + lfg * udata[3 + pp]) * udata[5 + pp];
00647     JMM[16] = udata[2 + pp] * (lfg * udata[1 + pp] + lgg * udata[4 + pp]) -
00648         (lff * udata[1 + pp] + lfg * udata[4 + pp]) * udata[5 + pp];
00649     JMM[17] = lg + lfg * (Quad[2] - Quad[5 + 0]) +
00650         (-lff + lgg) * udata[2 + pp] * udata[5 + pp];
00651     JMM[18] = lgg * udata[pp] * udata[2 + pp] +
00652         lff * udata[3 + pp] * udata[5 + pp] -
00653         lfg * (udata[2 + pp] * udata[3 + pp] + udata[pp] * udata[5 + pp]);
00654     JMM[19] =
00655         lgg * udata[1 + pp] * udata[2 + pp] +
00656         lff * udata[4 + pp] * udata[5 + pp] -
00657         lfg * (udata[2 + pp] * udata[4 + pp] + udata[1 + pp] * udata[5 + pp]);

```

```

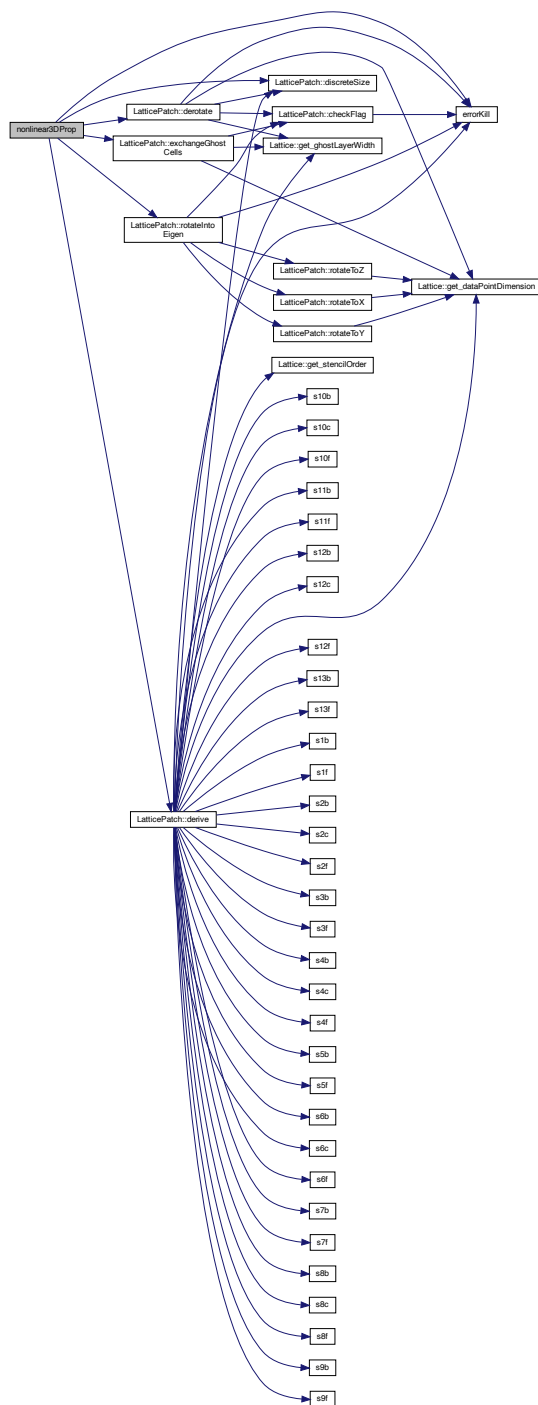
00658     JMM[20] = -lf + lgg * Quad[2] +
00659             udata[5 + pp] * (-2 * lfg * udata[2 + pp] + lff * udata[5 + pp]);
00660
00661     h[0] = 0;
00662     h[1] = dxData[pp] * JMM[15] + dxData[1 + pp] * JMM[16] +
00663           dxData[2 + pp] * JMM[17] + dxData[3 + pp] * JMM[18] +
00664           dxData[4 + pp] * JMM[19] + dxData[5 + pp] * (-1 + JMM[20]);
00665     h[2] = -(dxData[pp] * JMM[10]) - dxData[1 + pp] * JMM[11] -
00666           dxData[2 + pp] * JMM[12] - dxData[3 + pp] * JMM[13] +
00667           dxData[4 + pp] * (1 - JMM[14]) - dxData[5 + pp] * JMM[19];
00668     h[3] = 0;
00669     h[4] = dxData[2 + pp];
00670     h[5] = -dxData[1 + pp];
00671     h[0] += -(dyData[pp] * JMM[15]) - dyData[1 + pp] * JMM[16] -
00672           dyData[2 + pp] * JMM[17] - dyData[3 + pp] * JMM[18] -
00673           dyData[4 + pp] * JMM[19] + dyData[5 + pp] * (1 - JMM[20]);
00674     h[1] += 0;
00675     h[2] += dyData[pp] * JMM[6] + dyData[1 + pp] * JMM[7] +
00676           dyData[2 + pp] * JMM[8] + dyData[3 + pp] * (-1 + JMM[9]) +
00677           dyData[4 + pp] * JMM[13] + dyData[5 + pp] * JMM[18];
00678     h[3] += -dyData[2 + pp];
00679     h[4] += 0;
00680     h[5] += dyData[pp];
00681     h[0] += dzData[pp] * JMM[10] + dzData[1 + pp] * JMM[11] +
00682           dzData[2 + pp] * JMM[12] + dzData[3 + pp] * JMM[13] +
00683           dzData[4 + pp] * (-1 + JMM[14]) + dzData[5 + pp] * JMM[19];
00684     h[1] += -(dzData[pp] * JMM[6]) - dzData[1 + pp] * JMM[7] -
00685           dzData[2 + pp] * JMM[8] + dzData[3 + pp] * (1 - JMM[9]) -
00686           dzData[4 + pp] * JMM[13] - dzData[5 + pp] * JMM[18];
00687     h[2] += 0;
00688     h[3] += dzData[1 + pp];
00689     h[4] += -dzData[pp];
00690     h[5] += 0;
00691     h[0] -= h[3] * JMM[6] + h[4] * JMM[10] + h[5] * JMM[15];
00692     h[1] -= h[3] * JMM[7] + h[4] * JMM[11] + h[5] * JMM[16];
00693     h[2] -= h[3] * JMM[8] + h[4] * JMM[12] + h[5] * JMM[17];
00694     dudata[pp + 0] =
00695         h[2] * (-JMM[3] * (1 + JMM[2])) + JMM[1] * JMM[4] +
00696         h[1] * (JMM[3] * JMM[4] - JMM[1] * (1 + JMM[5])) +
00697         h[0] * (1 - JMM[4] * JMM[4] + JMM[5] + JMM[2] * (1 + JMM[5]));
00698     dudata[pp + 1] =
00699         h[2] * (JMM[3] * JMM[1] - (1 + JMM[0]) * JMM[4]) +
00700         h[1] * (1 - JMM[3] * JMM[3] + JMM[5] + JMM[0] * (1 + JMM[5])) +
00701         h[0] * (JMM[4] * JMM[3] - JMM[1] * (1 + JMM[5]));
00702     dudata[pp + 2] =
00703         h[2] * (1 - JMM[1] * JMM[1] + JMM[2] + JMM[0] * (1 + JMM[2])) +
00704         h[1] * (JMM[1] * JMM[3] - (1 + JMM[0]) * JMM[4]) +
00705         h[0] * (-((1 + JMM[2]) * JMM[3]) + JMM[1] * JMM[4]);
00706     detC =
00707         -((1 + JMM[2]) * (-1 + JMM[3] * JMM[3])) +
00708         (JMM[3] * JMM[1] - JMM[4]) * JMM[4] + JMM[5] + JMM[2] * JMM[5] +
00709         JMM[0] * (1 + JMM[2] - JMM[4] * JMM[4] + (1 + JMM[2]) * JMM[5]) -
00710         JMM[1] * (-JMM[4] * JMM[3] + JMM[1] * (1 + JMM[5]));
00711     dudata[pp + 0] /= detC;
00712     dudata[pp + 1] /= detC;
00713     dudata[pp + 2] /= detC;
00714     dudata[pp + 3] = h[3];
00715     dudata[pp + 4] = h[4];
00716     dudata[pp + 5] = h[5];
00717 }
00718 return;
00719 }

```

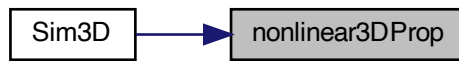
References [LatticePatch::buffData](#), [LatticePatch::derive\(\)](#), [LatticePatch::derotate\(\)](#), [LatticePatch::discreteSize\(\)](#), [errorKill\(\)](#), [LatticePatch::exchangeGhostCells\(\)](#), and [LatticePatch::rotateIntoEigen\(\)](#).

Referenced by [Sim3D\(\)](#).

Here is the call graph for this function:



Here is the caller graph for this function:



6.29 TimeEvolutionFunctions.cpp

[Go to the documentation of this file.](#)

```

00001 ///////////////////////////////////////////////////////////////////
00002 /// @file TimeEvolutionFunctions.cpp
00003 /// @brief Implementation of functions to propagate
00004 /// data vectors in time according to Maxwell's equations,
00005 /// and various orders in the HE weak-field expansion
00006 ///////////////////////////////////////////////////////////////////
00007
00008 #include "TimeEvolutionFunctions.h"
00009
00010 /// CNode right-hand-side function (CVRhsFn)
00011 int TimeEvolution::f(sunrealtype t, N_Vector u, N_Vector udot, void *data_loc) {
00012
00013     // Set recover pointer to provided lattice patch where the field data resides
00014     LatticePatch *data = static_cast<LatticePatch *>(data_loc);
00015
00016     // update circle
00017     // Access provided field values and temp. derivatives with NVector pointers
00018     sunrealtype *uData = NV_DATA_P(u),
00019                 *duData = NV_DATA_P(udot);
00020
00021     // Store original data location of the patch
00022     sunrealtype *originaluData = data->uData,
00023                 *originalduData = data->duData;
00024
00025     // Point patch data to arguments of f
00026     data->uData = uData;
00027     data->duData = duData;
00028
00029     // Time-evolve these arguments (the field data) with specific propagator below
00030     TimeEvolver(data, u, udot, c);
00031
00032     // Refer patch data back to original location
00033     data->uData = originaluData;
00034     data->duData = originalduData;
00035
00036     return (0);
00037 }
00038
00039 /// only under-the-hood-callable Maxwell propagation in 1D;
00040 /// unused parameters 2-4 for compliance with CVRhsFn - field data is here
00041 /// accessed implicitly via user data (lattice patch);
00042 /// same effect as the respective nonlinear function without nonlinear terms
00043 void linear1DProp(LatticePatch *data, N_Vector u, N_Vector udot, int *c) {
00044
00045     // pointers to temporal and spatial derivative data
00046     sunrealtype *duData = data->duData;
00047     sunrealtype *dxData = data->buffData[1 - 1];
00048
00049     // sequence along any dimension according to the scheme:
00050     data->exchangeGhostCells(1); // -> exchange halos
00051     data->rotateIntoEigen(
00052         1); // -> rotate all data to prepare derivative operation
00053     data->derive(1); // -> perform derivative approximation operation on it
00054     data->derotate(
00055         1, dxData); // -> derotate derived data for ensuing time-evolution
00056
00057     const sunindextype totalNP = data->discreteSize();
00058     sunindextype pp = 0;
00059     for (sunindextype i = 0; i < totalNP; i++) {
00060         pp = i * 6;
00061         /*
  
```

```

00062     simple vacuum Maxwell equations for the temporal derivatives using the
00063     spatial derivative only in x-direction without polarization or
00064     magnetization terms
00065     */
00066     duData[pp + 0] = 0;
00067     duData[pp + 1] = -dxData[pp + 5];
00068     duData[pp + 2] = dxData[pp + 4];
00069     duData[pp + 3] = 0;
00070     duData[pp + 4] = dxData[pp + 2];
00071     duData[pp + 5] = -dxData[pp + 1];
00072 }
00073 }
00074
00075 /// nonlinear 1D HE propagation
00076 void nonlinear1DProp(LatticePatch *data, N_Vector u, N_Vector udot, int *c) {
00077
00078     // N_Vector pointers to provided field values and their temp. derivatives
00079     sunrealtype *udata = NV_DATA_P(u),
00080                 *dudata = NV_DATA_P(udot);
00081
00082     // pointer to spatial derivatives via patch data
00083     sunrealtype *dxData = data->buffData[1 - 1];
00084
00085     // same sequence as in the linear case
00086     data->exchangeGhostCells(1);
00087     data->rotateIntoEigen(1);
00088     data->derive(1);
00089     data->derotate(1, dxData);
00090
00091     /*
00092     F and G are nonzero in the nonlinear case,
00093     polarization and magnetization derivatives
00094     w.r.t. E- and B-field go into the e.o.m.
00095     */
00096     static sunrealtype f, g; // em field invariants F, G
00097     // derivatives of HE Lagrangian w.r.t. field invariants
00098     static sunrealtype lf, lff, lfg, lg, lgg;
00099     // matrix to hold derivatives of polarization and magnetization
00100     static std::array<sunrealtype, 21> JMM;
00101     // array to hold E^2 and B^2 components
00102     static std::array<sunrealtype, 6> Quad;
00103     // array to hold intermediate temp. derivatives of E and B
00104     static std::array<sunrealtype, 6> h;
00105     // determinant needed for explicit matrix inversion
00106     static sunrealtype detC = nan("0x12345");
00107
00108     // number of points in the patch
00109     const sunindextype totalNP = data->discreteSize();
00110     #pragma omp parallel for default(none) \
00111     private(JMM, Quad, h, detC) \
00112     shared(totalNP, c, f, g, lf, lff, lfg, lg, lgg, udata, dudata, dxData) \
00113     schedule(static)
00114     for (sunindextype pp = 0; pp < totalNP * 6;
00115         pp += 6) { // loop over all 6dim points in the patch
00116         // em field Lorentz invariants F and G
00117         f = 0.5 * ((Quad[0] = udata[pp] * udata[pp]) +
00118                 (Quad[1] = udata[pp + 1] * udata[pp + 1]) +
00119                 (Quad[2] = udata[pp + 2] * udata[pp + 2]) -
00120                 (Quad[3] = udata[pp + 3] * udata[pp + 3]) -
00121                 (Quad[4] = udata[pp + 4] * udata[pp + 4]) -
00122                 (Quad[5] = udata[pp + 5] * udata[pp + 5]));
00123         g = udata[pp] * udata[pp + 3] + udata[pp + 1] * udata[pp + 4] +
00124             udata[pp + 2] * udata[pp + 5];
00125         // process/expansion order and corresponding derivative values of L
00126         // w.r.t. F, G
00127         switch (*c) {
00128         case 0: // linear Maxwell vacuum
00129             lf = 0;
00130             lff = 0;
00131             lfg = 0;
00132             lg = 0;
00133             lgg = 0;
00134             break;
00135         case 1: // only 4-photon processes
00136             lf = 0.000206527095658582755255648 * f;
00137             lff = 0.000206527095658582755255648;
00138             lfg = 0;
00139             lg = 0.0003614224174025198216973841 * g;
00140             lgg = 0.0003614224174025198216973841;
00141             break;
00142         case 2: // only 6-photon processes
00143             lf = 0.000354046449700427580438254 * f * f +
00144                 0.000191775160254398272737387 * g * g;
00145             lff = 0.0007080928994008551608765075 * f;
00146             lfg = 0.0003835503205087965454747749 * g;
00147             lg = 0.0003835503205087965454747749 * f * g;
00148             lgg = 0.0003835503205087965454747749 * f;

```

```

00149         break;
00150     case 3: // 4- and 6-photon processes
00151         lf = (0.000206527095658582755255648 + 0.000354046449700427580438254 * f) *
00152             f +
00153             0.000191775160254398272737387 * g * g;
00154         lff = 0.000206527095658582755255648 + 0.0007080928994008551608765075 * f;
00155         lfg = 0.0003835503205087965454747749 * g;
00156         lg = (0.0003614224174025198216973841 +
00157             0.0003835503205087965454747749 * f) *
00158             g;
00159         lgg = 0.0003614224174025198216973841 + 0.0003835503205087965454747749 * f;
00160         break;
00161     default:
00162         errorKill(
00163             "You need to specify a correct order in the weak-field expansion.");
00164     }
00165
00166     // derivatives of polarization and magnetization w.r.t. E and B
00167     // Jpx(Ex)
00168     JMM[0] = lf + lff * Quad[0] +
00169         udata[3 + pp] * (2 * lfg * udata[pp] + lgg * udata[3 + pp]);
00170     // Jpx(Ey)
00171     JMM[1] =
00172         lff * udata[pp] * udata[1 + pp] + lfg * udata[1 + pp] * udata[3 + pp] +
00173         lfg * udata[pp] * udata[4 + pp] + lgg * udata[3 + pp] * udata[4 + pp];
00174     // Jpy(Ey)
00175     JMM[2] = lf + lff * Quad[1] +
00176         udata[4 + pp] * (2 * lfg * udata[1 + pp] + lgg * udata[4 + pp]);
00177     // Jpx(Ez) = Jpz(Ex)
00178     JMM[3] =
00179         lff * udata[pp] * udata[2 + pp] + lfg * udata[2 + pp] * udata[3 + pp] +
00180         lfg * udata[pp] * udata[5 + pp] + lgg * udata[3 + pp] * udata[5 + pp];
00181     // Jpy(Ez) = Jpz(Ey)
00182     JMM[4] = lff * udata[1 + pp] * udata[2 + pp] +
00183         lfg * udata[2 + pp] * udata[4 + pp] +
00184         lfg * udata[1 + pp] * udata[5 + pp] +
00185         lgg * udata[4 + pp] * udata[5 + pp];
00186     // Jpz(Ez)
00187     JMM[5] = lf + lff * Quad[2] +
00188         udata[5 + pp] * (2 * lfg * udata[2 + pp] + lgg * udata[5 + pp]);
00189     // Jpx(Bx) = Jmx(Ex)
00190     JMM[6] = lg + lfg * (Quad[0] - Quad[3 + 0]) +
00191         (-lff + lgg) * udata[pp] * udata[3 + pp];
00192     // Jpy(Bx) = Jmx(Ey)
00193     JMM[7] = -(udata[3 + pp] * (lff * udata[1 + pp] + lfg * udata[4 + pp])) +
00194         udata[pp] * (lfg * udata[1 + pp] + lgg * udata[4 + pp]);
00195     // Jpz(Bx) = Jmx(Ez)
00196     JMM[8] = -(udata[3 + pp] * (lff * udata[2 + pp] + lfg * udata[5 + pp])) +
00197         udata[pp] * (lfg * udata[2 + pp] + lgg * udata[5 + pp]);
00198     // Jmx(Bx)
00199     JMM[9] = -lf + lgg * Quad[0] +
00200         udata[3 + pp] * (-2 * lfg * udata[pp] + lff * udata[3 + pp]);
00201     // Jpx(By) = Jmy(Ex)
00202     JMM[10] = udata[1 + pp] * (lfg * udata[pp] + lgg * udata[3 + pp]) -
00203         (lff * udata[pp] + lfg * udata[3 + pp]) * udata[4 + pp];
00204     // Jpy(By) = Jmy(Ey)
00205     JMM[11] = lg + lfg * (Quad[1] - Quad[4 + 0]) +
00206         (-lff + lgg) * udata[1 + pp] * udata[4 + pp];
00207     // Jpz(By) = Jmy(Ez)
00208     JMM[12] = -(udata[4 + pp] * (lff * udata[2 + pp] + lfg * udata[5 + pp])) +
00209         udata[1 + pp] * (lfg * udata[2 + pp] + lgg * udata[5 + pp]);
00210     // Jmx(By) = Jmy(Bx)
00211     JMM[13] = lgg * udata[pp] * udata[1 + pp] +
00212         lff * udata[3 + pp] * udata[4 + pp] -
00213         lfg * (udata[1 + pp] * udata[3 + pp] + udata[pp] * udata[4 + pp]);
00214     // Jmy(By)
00215     JMM[14] = -lf + lgg * Quad[1] +
00216         udata[4 + pp] * (-2 * lfg * udata[1 + pp] + lff * udata[4 + pp]);
00217     // Jmz(Ex) = Jpx(Bz)
00218     JMM[15] = udata[2 + pp] * (lfg * udata[pp] + lgg * udata[3 + pp]) -
00219         (lff * udata[pp] + lfg * udata[3 + pp]) * udata[5 + pp];
00220     // Jmz(Ey) = Jpy(Bz)
00221     JMM[16] = udata[2 + pp] * (lfg * udata[1 + pp] + lgg * udata[4 + pp]) -
00222         (lff * udata[1 + pp] + lfg * udata[4 + pp]) * udata[5 + pp];
00223     // Jpz(Bz) = Jmz(Ez)
00224     JMM[17] = lg + lfg * (Quad[2] - Quad[5 + 0]) +
00225         (-lff + lgg) * udata[2 + pp] * udata[5 + pp];
00226     // Jmz(Bx) = Jmx(Bz)
00227     JMM[18] = lgg * udata[pp] * udata[2 + pp] +
00228         lff * udata[3 + pp] * udata[5 + pp] -
00229         lfg * (udata[2 + pp] * udata[3 + pp] + udata[pp] * udata[5 + pp]);
00230     // Jmy(Bz) = Jmz(By)
00231     JMM[19] =
00232         lgg * udata[1 + pp] * udata[2 + pp] +
00233         lff * udata[4 + pp] * udata[5 + pp] -
00234         lfg * (udata[2 + pp] * udata[4 + pp] + udata[1 + pp] * udata[5 + pp]);
00235     // Jmz(Bz)

```

```

00236     JMM[20] = -lf + lgg * Quad[2] +
00237             udata[5 + pp] * (-2 * lfg * udata[2 + pp] + lff * udata[5 + pp]);
00238
00239     // apply Z
00240     // top block: -QJm(E)*E, Q-QJm(B)*B
00241     h[0] = 0;
00242     h[1] = dxData[pp] * JMM[15] + dxData[1 + pp] * JMM[16] +
00243           dxData[2 + pp] * JMM[17] + dxData[3 + pp] * JMM[18] +
00244           dxData[4 + pp] * JMM[19] + dxData[5 + pp] * (-1 + JMM[20]);
00245     h[2] = -(dxData[pp] * JMM[10]) - dxData[1 + pp] * JMM[11] -
00246           dxData[2 + pp] * JMM[12] - dxData[3 + pp] * JMM[13] +
00247           dxData[4 + pp] * (1 - JMM[14]) - dxData[5 + pp] * JMM[19];
00248     // bottom blocks: -Q*E
00249     h[3] = 0;
00250     h[4] = dxData[2 + pp];
00251     h[5] = -dxData[1 + pp];
00252     // (1+A)^-1 applies only to E components
00253     // -Jp(B)*B
00254     h[0] -= h[3] * JMM[6] + h[4] * JMM[10] + h[5] * JMM[15];
00255     h[1] -= h[3] * JMM[7] + h[4] * JMM[11] + h[5] * JMM[16];
00256     h[2] -= h[3] * JMM[8] + h[4] * JMM[12] + h[5] * JMM[17];
00257     // apply C^-1 explicitly, with C=1+Jp(E)
00258     dudata[pp + 0] =
00259         h[2] * (-(JMM[3] * (1 + JMM[2])) + JMM[1] * JMM[4]) +
00260         h[1] * (JMM[3] * JMM[4] - JMM[1] * (1 + JMM[5])) +
00261         h[0] * (1 - JMM[4] * JMM[4] + JMM[5] + JMM[2] * (1 + JMM[5]));
00262     dudata[pp + 1] =
00263         h[2] * (JMM[3] * JMM[1] - (1 + JMM[0]) * JMM[4]) +
00264         h[1] * (1 - JMM[3] * JMM[3] + JMM[5] + JMM[0] * (1 + JMM[5])) +
00265         h[0] * (JMM[4] * JMM[3] - JMM[1] * (1 + JMM[5]));
00266     dudata[pp + 2] =
00267         h[2] * (1 - JMM[1] * JMM[1] + JMM[2] + JMM[0] * (1 + JMM[2])) +
00268         h[1] * (JMM[1] * JMM[3] - (1 + JMM[0]) * JMM[4]) +
00269         h[0] * (-(1 + JMM[2]) * JMM[3] + JMM[1] * JMM[4]);
00270     detC = // determinant of C
00271         -((1 + JMM[2]) * (-1 + JMM[3] * JMM[3])) +
00272         (JMM[3] * JMM[1] - JMM[4] * JMM[4] + JMM[5] + JMM[2] * JMM[5] +
00273         JMM[0] * (1 + JMM[2] - JMM[4] * JMM[4] + (1 + JMM[2]) * JMM[5]) -
00274         JMM[1] * (-(JMM[4] * JMM[3] + JMM[1] * (1 + JMM[5])));
00275     dudata[pp + 0] /= detC;
00276     dudata[pp + 1] /= detC;
00277     dudata[pp + 2] /= detC;
00278     dudata[pp + 3] = h[3];
00279     dudata[pp + 4] = h[4];
00280     dudata[pp + 5] = h[5];
00281 }
00282 return;
00283 }
00284
00285 /// only under-the-hood-callable Maxwell propagation in 2D
00286 void linear2DProp(LatticePatch *data, N_Vector u, N_Vector udot, int *c) {
00287
00288     sunrealtype *duData = data->duData;
00289     sunrealtype *dxData = data->buffData[1 - 1];
00290     sunrealtype *dyData = data->buffData[2 - 1];
00291
00292     data->exchangeGhostCells(1);
00293     data->rotateIntoEigen(1);
00294     data->derive(1);
00295     data->derotate(1, dxData);
00296     data->exchangeGhostCells(2);
00297     data->rotateIntoEigen(2);
00298     data->derive(2);
00299     data->derotate(2, dyData);
00300
00301     const sunindextype totalNP = data->discreteSize();
00302     sunindextype pp = 0;
00303     for (sunindextype i = 0; i < totalNP; i++) {
00304         pp = i * 6;
00305         duData[pp + 0] = dyData[pp + 5];
00306         duData[pp + 1] = -dxData[pp + 5];
00307         duData[pp + 2] = -dyData[pp + 3] + dxData[pp + 4];
00308         duData[pp + 3] = -dyData[pp + 2];
00309         duData[pp + 4] = dxData[pp + 2];
00310         duData[pp + 5] = dyData[pp + 0] - dxData[pp + 1];
00311     }
00312 }
00313
00314 /// nonlinear 2D HE propagation
00315 void nonlinear2DProp(LatticePatch *data, N_Vector u, N_Vector udot, int *c) {
00316
00317     sunrealtype *udata = NV_DATA_P(u),
00318                 *dudata = NV_DATA_P(udot);
00319
00320     sunrealtype *dxData = data->buffData[1 - 1];
00321     sunrealtype *dyData = data->buffData[2 - 1];
00322

```

```

00323 data->exchangeGhostCells(1);
00324 data->rotateIntoEigen(1);
00325 data->derive(1);
00326 data->derotate(1, dxData);
00327 data->exchangeGhostCells(2);
00328 data->rotateIntoEigen(2);
00329 data->derive(2);
00330 data->derotate(2, dyData);
00331
00332 static sunrealtype f, g;
00333 static sunrealtype lf, lff, lfg, lg, lgg;
00334 static std::array<sunrealtype, 21> JMM;
00335 static std::array<sunrealtype, 6> Quad;
00336 static std::array<sunrealtype, 6> h;
00337 static sunrealtype detC;
00338
00339 const sunindextype totalNP = data->discreteSize();
00340 #pragma omp parallel for default(none) \
00341 private(JMM, Quad, h, detC) \
00342 shared(totalNP, c, f, g, lf, lff, lfg, lg, lgg, udata, dudata, \
00343         dxData, dyData) \
00344 schedule(static)
00345 for (sunindextype pp = 0; pp < totalNP * 6; pp += 6) {
00346     f = 0.5 * ((Quad[0] = udata[pp] * udata[pp]) +
00347              (Quad[1] = udata[pp + 1] * udata[pp + 1]) +
00348              (Quad[2] = udata[pp + 2] * udata[pp + 2]) -
00349              (Quad[3] = udata[pp + 3] * udata[pp + 3]) -
00350              (Quad[4] = udata[pp + 4] * udata[pp + 4]) -
00351              (Quad[5] = udata[pp + 5] * udata[pp + 5]));
00352     g = udata[pp] * udata[pp + 3] + udata[pp + 1] * udata[pp + 4] +
00353         udata[pp + 2] * udata[pp + 5];
00354     switch (*c) {
00355     case 0:
00356         lf = 0;
00357         lff = 0;
00358         lfg = 0;
00359         lg = 0;
00360         lgg = 0;
00361         break;
00362     case 1:
00363         lf = 0.000206527095658582755255648 * f;
00364         lff = 0.000206527095658582755255648;
00365         lfg = 0;
00366         lg = 0.0003614224174025198216973841 * g;
00367         lgg = 0.0003614224174025198216973841;
00368         break;
00369     case 2:
00370         lf = 0.000354046449700427580438254 * f * f +
00371             0.000191775160254398272737387 * g * g;
00372         lff = 0.0007080928994008551608765075 * f;
00373         lfg = 0.0003835503205087965454747749 * g;
00374         lg = 0.0003835503205087965454747749 * f * g;
00375         lgg = 0.0003835503205087965454747749 * f;
00376         break;
00377     case 3:
00378         lf = (0.000206527095658582755255648 + 0.000354046449700427580438254 * f) *
00379             f +
00380             0.000191775160254398272737387 * g * g;
00381         lff = 0.000206527095658582755255648 + 0.000708092899400855160876508 * f;
00382         lfg = 0.0003835503205087965454747749 * g;
00383         lg = (0.000361422417402519821697384 + 0.000383550320508796545474775 * f) *
00384             g;
00385         lgg = 0.000361422417402519821697384 + 0.000383550320508796545474775 * f;
00386         break;
00387     default:
00388         errorKill(
00389             "You need to specify a correct order in the weak-field expansion.");
00390     }
00391
00392     JMM[0] = lf + lff * Quad[0] +
00393         udata[3 + pp] * (2 * lfg * udata[pp] + lgg * udata[3 + pp]);
00394     JMM[1] =
00395         lff * udata[pp] * udata[1 + pp] + lfg * udata[1 + pp] * udata[3 + pp] +
00396         lfg * udata[pp] * udata[4 + pp] + lgg * udata[3 + pp] * udata[4 + pp];
00397     JMM[2] = lf + lff * Quad[1] +
00398         udata[4 + pp] * (2 * lfg * udata[1 + pp] + lgg * udata[4 + pp]);
00399     JMM[3] =
00400         lff * udata[pp] * udata[2 + pp] + lfg * udata[2 + pp] * udata[3 + pp] +
00401         lfg * udata[pp] * udata[5 + pp] + lgg * udata[3 + pp] * udata[5 + pp];
00402     JMM[4] = lff * udata[1 + pp] * udata[2 + pp] +
00403         lfg * udata[2 + pp] * udata[4 + pp] +
00404         lfg * udata[1 + pp] * udata[5 + pp] +
00405         lgg * udata[4 + pp] * udata[5 + pp];
00406     JMM[5] = lf + lff * Quad[2] +
00407         udata[5 + pp] * (2 * lfg * udata[2 + pp] + lgg * udata[5 + pp]);
00408     JMM[6] = lg + lfg * (Quad[0] - Quad[3 + 0]) +
00409         (-lff + lgg) * udata[pp] * udata[3 + pp];

```



```

00410 JMM[7] = -(udata[3 + pp] * (lff * udata[1 + pp] + lfg * udata[4 + pp])) +
00411         udata[pp] * (lfg * udata[1 + pp] + lgg * udata[4 + pp]);
00412 JMM[8] = -(udata[3 + pp] * (lff * udata[2 + pp] + lfg * udata[5 + pp])) +
00413         udata[pp] * (lfg * udata[2 + pp] + lgg * udata[5 + pp]);
00414 JMM[9] = -lf + lgg * Quad[0] +
00415         udata[3 + pp] * (-2 * lfg * udata[pp] + lff * udata[3 + pp]);
00416 JMM[10] = udata[1 + pp] * (lfg * udata[pp] + lgg * udata[3 + pp]) -
00417         (lff * udata[pp] + lfg * udata[3 + pp]) * udata[4 + pp];
00418 JMM[11] = lg + lfg * (Quad[1] - Quad[4 + 0]) +
00419         (-lff + lgg) * udata[1 + pp] * udata[4 + pp];
00420 JMM[12] = -(udata[4 + pp] * (lff * udata[2 + pp] + lfg * udata[5 + pp])) +
00421         udata[1 + pp] * (lfg * udata[2 + pp] + lgg * udata[5 + pp]);
00422 JMM[13] = lgg * udata[pp] * udata[1 + pp] +
00423         lff * udata[3 + pp] * udata[4 + pp] -
00424         lfg * (udata[1 + pp] * udata[3 + pp] + udata[pp] * udata[4 + pp]);
00425 JMM[14] = -lf + lgg * Quad[1] +
00426         udata[4 + pp] * (-2 * lfg * udata[1 + pp] + lff * udata[4 + pp]);
00427 JMM[15] = udata[2 + pp] * (lfg * udata[pp] + lgg * udata[3 + pp]) -
00428         (lff * udata[pp] + lfg * udata[3 + pp]) * udata[5 + pp];
00429 JMM[16] = udata[2 + pp] * (lfg * udata[1 + pp] + lgg * udata[4 + pp]) -
00430         (lff * udata[1 + pp] + lfg * udata[4 + pp]) * udata[5 + pp];
00431 JMM[17] = lg + lfg * (Quad[2] - Quad[5 + 0]) +
00432         (-lff + lgg) * udata[2 + pp] * udata[5 + pp];
00433 JMM[18] = lgg * udata[pp] * udata[2 + pp] +
00434         lff * udata[3 + pp] * udata[5 + pp] -
00435         lfg * (udata[2 + pp] * udata[3 + pp] + udata[pp] * udata[5 + pp]);
00436 JMM[19] =
00437         lgg * udata[1 + pp] * udata[2 + pp] +
00438         lff * udata[4 + pp] * udata[5 + pp] -
00439         lfg * (udata[2 + pp] * udata[4 + pp] + udata[1 + pp] * udata[5 + pp]);
00440 JMM[20] = -lf + lgg * Quad[2] +
00441         udata[5 + pp] * (-2 * lfg * udata[2 + pp] + lff * udata[5 + pp]);
00442
00443 h[0] = 0;
00444 h[1] = dxData[pp] * JMM[15] + dxData[1 + pp] * JMM[16] +
00445         dxData[2 + pp] * JMM[17] + dxData[3 + pp] * JMM[18] +
00446         dxData[4 + pp] * JMM[19] + dxData[5 + pp] * (-1 + JMM[20]);
00447 h[2] = -(dxData[pp] * JMM[10]) - dxData[1 + pp] * JMM[11] -
00448         dxData[2 + pp] * JMM[12] - dxData[3 + pp] * JMM[13] +
00449         dxData[4 + pp] * (1 - JMM[14]) - dxData[5 + pp] * JMM[19];
00450 h[3] = 0;
00451 h[4] = dxData[2 + pp];
00452 h[5] = -dxData[1 + pp];
00453 h[0] += -(dyData[pp] * JMM[15]) - dyData[1 + pp] * JMM[16] -
00454         dyData[2 + pp] * JMM[17] - dyData[3 + pp] * JMM[18] -
00455         dyData[4 + pp] * JMM[19] + dyData[5 + pp] * (1 - JMM[20]);
00456 h[1] += 0;
00457 h[2] += dyData[pp] * JMM[6] + dyData[1 + pp] * JMM[7] +
00458         dyData[2 + pp] * JMM[8] + dyData[3 + pp] * (-1 + JMM[9]) +
00459         dyData[4 + pp] * JMM[13] + dyData[5 + pp] * JMM[18];
00460 h[3] += -dyData[2 + pp];
00461 h[4] += 0;
00462 h[5] += dyData[pp];
00463 h[0] -= h[3] * JMM[6] + h[4] * JMM[10] + h[5] * JMM[15];
00464 h[1] -= h[3] * JMM[7] + h[4] * JMM[11] + h[5] * JMM[16];
00465 h[2] -= h[3] * JMM[8] + h[4] * JMM[12] + h[5] * JMM[17];
00466 dudata[pp + 0] =
00467         h[2] * (-(JMM[3] * (1 + JMM[2])) + JMM[1] * JMM[4]) +
00468         h[1] * (JMM[3] * JMM[4] - JMM[1] * (1 + JMM[5])) +
00469         h[0] * (1 - JMM[4] * JMM[4] + JMM[5] + JMM[2] * (1 + JMM[5]));
00470 dudata[pp + 1] =
00471         h[2] * (JMM[3] * JMM[1] - (1 + JMM[0]) * JMM[4]) +
00472         h[1] * (1 - JMM[3] * JMM[3] + JMM[3] + JMM[5] + JMM[0] * (1 + JMM[5])) +
00473         h[0] * (JMM[4] * JMM[3] - JMM[1] * (1 + JMM[5]));
00474 dudata[pp + 2] =
00475         h[2] * (1 - JMM[1] * JMM[1] + JMM[2] + JMM[0] * (1 + JMM[2])) +
00476         h[1] * (JMM[1] * JMM[3] - (1 + JMM[0]) * JMM[4]) +
00477         h[0] * (-(1 + JMM[2]) * JMM[3] + JMM[1] * JMM[4]);
00478 detC =
00479         -((1 + JMM[2]) * (-1 + JMM[3] * JMM[3])) +
00480         (JMM[3] * JMM[1] - JMM[4] * JMM[4] + JMM[5] + JMM[2] * JMM[5] +
00481         JMM[0] * (1 + JMM[2] - JMM[4] * JMM[4] + (1 + JMM[2]) * JMM[5]) -
00482         JMM[1] * (-(JMM[4] * JMM[3]) + JMM[1] * (1 + JMM[5])));
00483 dudata[pp + 0] /= detC;
00484 dudata[pp + 1] /= detC;
00485 dudata[pp + 2] /= detC;
00486 dudata[pp + 3] = h[3];
00487 dudata[pp + 4] = h[4];
00488 dudata[pp + 5] = h[5];
00489 }
00490 return;
00491 }
00492
00493 /// only under-the-hood-callable Maxwell propagation in 3D
00494 void linear3DProp(LatticePatch *data, N_Vector u, N_Vector udot, int *c) {
00495
00496     sunrealtype *duData = data->duData;

```

```

00497     sunrealtype *dxData = data->buffData[1 - 1];
00498     sunrealtype *dyData = data->buffData[2 - 1];
00499     sunrealtype *dzData = data->buffData[3 - 1];
00500
00501     data->exchangeGhostCells(1);
00502     data->rotateIntoEigen(1);
00503     data->derive(1);
00504     data->derotate(1, dxData);
00505     data->exchangeGhostCells(2);
00506     data->rotateIntoEigen(2);
00507     data->derive(2);
00508     data->derotate(2, dyData);
00509     data->exchangeGhostCells(3);
00510     data->rotateIntoEigen(3);
00511     data->derive(3);
00512     data->derotate(3, dzData);
00513
00514     const sunindextype totalNP = data->discreteSize();
00515     sunindextype pp = 0;
00516     for (sunindextype i = 0; i < totalNP; i++) {
00517         pp = i * 6;
00518         duData[pp + 0] = dyData[pp + 5] - dzData[pp + 4];
00519         duData[pp + 1] = dzData[pp + 3] - dxData[pp + 5];
00520         duData[pp + 2] = dxData[pp + 4] - dyData[pp + 3];
00521         duData[pp + 3] = -dyData[pp + 2] + dzData[pp + 1];
00522         duData[pp + 4] = -dzData[pp + 0] + dxData[pp + 2];
00523         duData[pp + 5] = -dxData[pp + 1] + dyData[pp + 0];
00524     }
00525 }
00526
00527 /// nonlinear 3D HE propagation
00528 void nonlinear3DProp(LatticePatch *data, N_Vector u, N_Vector udot, int *c) {
00529
00530     sunrealtype *udata = NV_DATA_P(u),
00531                 *dudata = NV_DATA_P(udot);
00532
00533     sunrealtype *dxData = data->buffData[1 - 1];
00534     sunrealtype *dyData = data->buffData[2 - 1];
00535     sunrealtype *dzData = data->buffData[3 - 1];
00536
00537     data->exchangeGhostCells(1);
00538     data->rotateIntoEigen(1);
00539     data->derive(1);
00540     data->derotate(1, dxData);
00541     data->exchangeGhostCells(2);
00542     data->rotateIntoEigen(2);
00543     data->derive(2);
00544     data->derotate(2, dyData);
00545     data->exchangeGhostCells(3);
00546     data->rotateIntoEigen(3);
00547     data->derive(3);
00548     data->derotate(3, dzData);
00549
00550     static sunrealtype f, g;
00551     static sunrealtype lf, lff, lfg, lg, lgg;
00552     static std::array<sunrealtype, 21> JMM;
00553     static std::array<sunrealtype, 6> Quad;
00554     static std::array<sunrealtype, 6> h;
00555     static sunrealtype detC = nan("0x12345");
00556
00557     const sunindextype totalNP = data->discreteSize();
00558     #pragma omp parallel for default(none) \
00559     private(JMM, Quad, h, detC) \
00560     shared(totalNP, c, f, g, lf, lff, lfg, lg, lgg, udata, dudata, \
00561            dxData, dyData, dzData) \
00562     schedule(static)
00563     for (sunindextype pp = 0; pp < totalNP * 6; pp += 6) {
00564         f = 0.5 * ((Quad[0] = udata[pp] * udata[pp]) +
00565                  (Quad[1] = udata[pp + 1] * udata[pp + 1]) +
00566                  (Quad[2] = udata[pp + 2] * udata[pp + 2]) -
00567                  (Quad[3] = udata[pp + 3] * udata[pp + 3]) -
00568                  (Quad[4] = udata[pp + 4] * udata[pp + 4]) -
00569                  (Quad[5] = udata[pp + 5] * udata[pp + 5]));
00570         g = udata[pp] * udata[pp + 3] + udata[pp + 1] * udata[pp + 4] +
00571             udata[pp + 2] * udata[pp + 5];
00572         switch (*c) {
00573             case 0:
00574                 lf = 0;
00575                 lff = 0;
00576                 lfg = 0;
00577                 lg = 0;
00578                 lgg = 0;
00579                 break;
00580             case 1:
00581                 lf = 0.000206527095658582755255648 * f;
00582                 lff = 0.000206527095658582755255648;
00583                 lfg = 0;

```

```

00584     lg = 0.0003614224174025198216973841 * g;
00585     lgg = 0.0003614224174025198216973841;
00586     break;
00587 case 2:
00588     lf = 0.000354046449700427580438254 * f * f +
00589         0.000191775160254398272737387 * g * g;
00590     lff = 0.0007080928994008551608765075 * f;
00591     lfg = 0.0003835503205087965454747749 * g;
00592     lg = 0.0003835503205087965454747749 * f * g;
00593     lgg = 0.0003835503205087965454747749 * f;
00594     break;
00595 case 3:
00596     lf = (0.000206527095658582755255648 + 0.000354046449700427580438254 * f) *
00597         f +
00598         0.000191775160254398272737387 * g * g;
00599     lff = 0.000206527095658582755255648 + 0.000708092899400855160876508 * f;
00600     lfg = 0.0003835503205087965454747749 * g;
00601     lg = (0.000361422417402519821697384 + 0.000383550320508796545474775 * f) *
00602         g;
00603     lgg = 0.000361422417402519821697384 + 0.000383550320508796545474775 * f;
00604     break;
00605 default:
00606     errorKill(
00607         "You need to specify a correct order in the weak-field expansion.");
00608 }
00609
00610 JMM[0] = lf + lff * Quad[0] +
00611     udata[3 + pp] * (2 * lfg * udata[pp] + lgg * udata[3 + pp]);
00612 JMM[1] =
00613     lff * udata[pp] * udata[1 + pp] + lfg * udata[1 + pp] * udata[3 + pp] +
00614     lfg * udata[pp] * udata[4 + pp] + lgg * udata[3 + pp] * udata[4 + pp];
00615 JMM[2] = lf + lff * Quad[1] +
00616     udata[4 + pp] * (2 * lfg * udata[1 + pp] + lgg * udata[4 + pp]);
00617 JMM[3] =
00618     lff * udata[pp] * udata[2 + pp] + lfg * udata[2 + pp] * udata[3 + pp] +
00619     lfg * udata[pp] * udata[5 + pp] + lgg * udata[3 + pp] * udata[5 + pp];
00620 JMM[4] = lff * udata[1 + pp] * udata[2 + pp] +
00621     lfg * udata[2 + pp] * udata[4 + pp] +
00622     lfg * udata[1 + pp] * udata[5 + pp] +
00623     lgg * udata[4 + pp] * udata[5 + pp];
00624 JMM[5] = lf + lff * Quad[2] +
00625     udata[5 + pp] * (2 * lfg * udata[2 + pp] + lgg * udata[5 + pp]);
00626 JMM[6] = lg + lfg * (Quad[0] - Quad[3 + 0]) +
00627     (-lff + lgg) * udata[pp] * udata[3 + pp];
00628 JMM[7] = -(udata[3 + pp] * (lff * udata[1 + pp] + lfg * udata[4 + pp])) +
00629     udata[pp] * (lfg * udata[1 + pp] + lgg * udata[4 + pp]);
00630 JMM[8] = -(udata[3 + pp] * (lff * udata[2 + pp] + lfg * udata[5 + pp])) +
00631     udata[pp] * (lfg * udata[2 + pp] + lgg * udata[5 + pp]);
00632 JMM[9] = -lf + lgg * Quad[0] +
00633     udata[3 + pp] * (-2 * lfg * udata[pp] + lff * udata[3 + pp]);
00634 JMM[10] = udata[1 + pp] * (lfg * udata[pp] + lgg * udata[3 + pp]) -
00635     (lff * udata[pp] + lfg * udata[3 + pp]) * udata[4 + pp];
00636 JMM[11] = lg + lfg * (Quad[1] - Quad[4 + 0]) +
00637     (-lff + lgg) * udata[1 + pp] * udata[4 + pp];
00638 JMM[12] = -(udata[4 + pp] * (lff * udata[2 + pp] + lfg * udata[5 + pp])) +
00639     udata[1 + pp] * (lfg * udata[2 + pp] + lgg * udata[5 + pp]);
00640 JMM[13] = lgg * udata[pp] * udata[1 + pp] +
00641     lff * udata[3 + pp] * udata[4 + pp] -
00642     lfg * (udata[1 + pp] * udata[3 + pp] + udata[pp] * udata[4 + pp]);
00643 JMM[14] = -lf + lgg * Quad[1] +
00644     udata[4 + pp] * (-2 * lfg * udata[1 + pp] + lff * udata[4 + pp]);
00645 JMM[15] = udata[2 + pp] * (lfg * udata[pp] + lgg * udata[3 + pp]) -
00646     (lff * udata[pp] + lfg * udata[3 + pp]) * udata[5 + pp];
00647 JMM[16] = udata[2 + pp] * (lfg * udata[1 + pp] + lgg * udata[4 + pp]) -
00648     (lff * udata[1 + pp] + lfg * udata[4 + pp]) * udata[5 + pp];
00649 JMM[17] = lg + lfg * (Quad[2] - Quad[5 + 0]) +
00650     (-lff + lgg) * udata[2 + pp] * udata[5 + pp];
00651 JMM[18] = lgg * udata[pp] * udata[2 + pp] +
00652     lff * udata[3 + pp] * udata[5 + pp] -
00653     lfg * (udata[2 + pp] * udata[3 + pp] + udata[pp] * udata[5 + pp]);
00654 JMM[19] =
00655     lgg * udata[1 + pp] * udata[2 + pp] +
00656     lff * udata[4 + pp] * udata[5 + pp] -
00657     lfg * (udata[2 + pp] * udata[4 + pp] + udata[1 + pp] * udata[5 + pp]);
00658 JMM[20] = -lf + lgg * Quad[2] +
00659     udata[5 + pp] * (-2 * lfg * udata[2 + pp] + lff * udata[5 + pp]);
00660
00661 h[0] = 0;
00662 h[1] = dxData[pp] * JMM[15] + dxData[1 + pp] * JMM[16] +
00663     dxData[2 + pp] * JMM[17] + dxData[3 + pp] * JMM[18] +
00664     dxData[4 + pp] * JMM[19] + dxData[5 + pp] * (-1 + JMM[20]);
00665 h[2] = -(dxData[pp] * JMM[10]) - dxData[1 + pp] * JMM[11] -
00666     dxData[2 + pp] * JMM[12] - dxData[3 + pp] * JMM[13] +
00667     dxData[4 + pp] * (1 - JMM[14]) - dxData[5 + pp] * JMM[19];
00668 h[3] = 0;
00669 h[4] = dxData[2 + pp];
00670 h[5] = -dxData[1 + pp];

```

```

00671     h[0] += -(dyData[pp] * JMM[15]) - dyData[1 + pp] * JMM[16] -
00672             dyData[2 + pp] * JMM[17] - dyData[3 + pp] * JMM[18] -
00673             dyData[4 + pp] * JMM[19] + dyData[5 + pp] * (1 - JMM[20]);
00674     h[1] += 0;
00675     h[2] += dyData[pp] * JMM[6] + dyData[1 + pp] * JMM[7] +
00676             dyData[2 + pp] * JMM[8] + dyData[3 + pp] * (-1 + JMM[9]) +
00677             dyData[4 + pp] * JMM[13] + dyData[5 + pp] * JMM[18];
00678     h[3] += -dyData[2 + pp];
00679     h[4] += 0;
00680     h[5] += dyData[pp];
00681     h[0] += dzData[pp] * JMM[10] + dzData[1 + pp] * JMM[11] +
00682             dzData[2 + pp] * JMM[12] + dzData[3 + pp] * JMM[13] +
00683             dzData[4 + pp] * (-1 + JMM[14]) + dzData[5 + pp] * JMM[19];
00684     h[1] += -(dzData[pp] * JMM[6]) - dzData[1 + pp] * JMM[7] -
00685             dzData[2 + pp] * JMM[8] + dzData[3 + pp] * (1 - JMM[9]) -
00686             dzData[4 + pp] * JMM[13] - dzData[5 + pp] * JMM[18];
00687     h[2] += 0;
00688     h[3] += dzData[1 + pp];
00689     h[4] += -dzData[pp];
00690     h[5] += 0;
00691     h[0] -= h[3] * JMM[6] + h[4] * JMM[10] + h[5] * JMM[15];
00692     h[1] -= h[3] * JMM[7] + h[4] * JMM[11] + h[5] * JMM[16];
00693     h[2] -= h[3] * JMM[8] + h[4] * JMM[12] + h[5] * JMM[17];
00694     dudata[pp + 0] =
00695         h[2] * (-(JMM[3] * (1 + JMM[2])) + JMM[1] * JMM[4]) +
00696         h[1] * (JMM[3] * JMM[4] - JMM[1] * (1 + JMM[5])) +
00697         h[0] * (1 - JMM[4] * JMM[4] + JMM[5] + JMM[2] * (1 + JMM[5]));
00698     dudata[pp + 1] =
00699         h[2] * (JMM[3] * JMM[1] - (1 + JMM[0]) * JMM[4]) +
00700         h[1] * (1 - JMM[3] * JMM[3] + JMM[5] + JMM[0] * (1 + JMM[5])) +
00701         h[0] * (JMM[4] * JMM[3] - JMM[1] * (1 + JMM[5]));
00702     dudata[pp + 2] =
00703         h[2] * (1 - JMM[1] * JMM[1] + JMM[2] + JMM[0] * (1 + JMM[2])) +
00704         h[1] * (JMM[1] * JMM[3] - (1 + JMM[0]) * JMM[4]) +
00705         h[0] * (-(1 + JMM[2]) * JMM[3] + JMM[1] * JMM[4]);
00706     detC =
00707         -(1 + JMM[2]) * (-1 + JMM[3] * JMM[3]) +
00708         (JMM[3] * JMM[1] - JMM[4]) * JMM[4] + JMM[5] + JMM[2] * JMM[5] +
00709         JMM[0] * (1 + JMM[2] - JMM[4] * JMM[4] + (1 + JMM[2]) * JMM[5]) -
00710         JMM[1] * (-(JMM[4] * JMM[3]) + JMM[1] * (1 + JMM[5]));
00711     dudata[pp + 0] /= detC;
00712     dudata[pp + 1] /= detC;
00713     dudata[pp + 2] /= detC;
00714     dudata[pp + 3] = h[3];
00715     dudata[pp + 4] = h[4];
00716     dudata[pp + 5] = h[5];
00717 }
00718 return;
00719 }

```

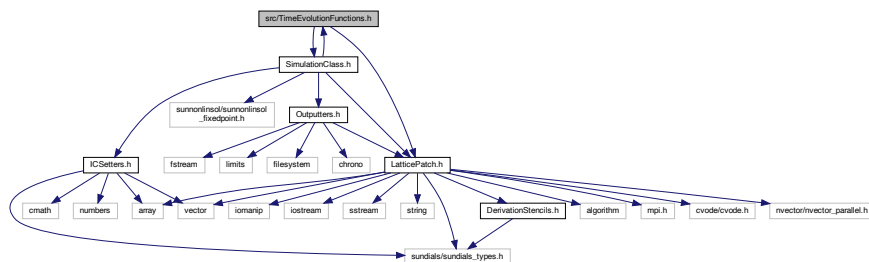
6.30 src/TimeEvolutionFunctions.h File Reference

Functions to propagate data vectors in time according to Maxwell's equations, and various orders in the HE weak-field expansion.

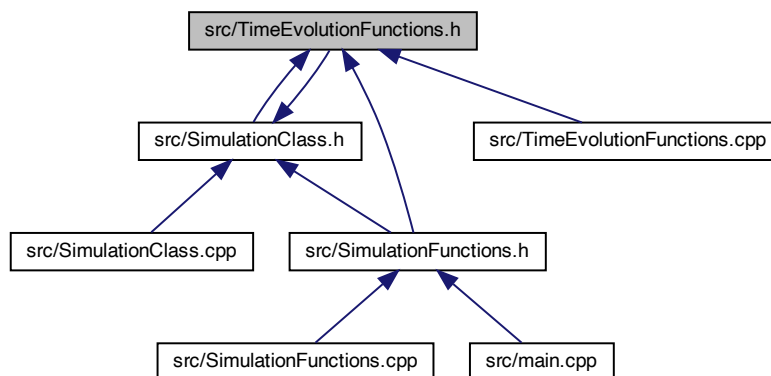
```
#include "LatticePatch.h"
```

```
#include "SimulationClass.h"
```

Include dependency graph for TimeEvolutionFunctions.h:



This graph shows which files directly or indirectly include this file:



Data Structures

- class [TimeEvolution](#)
monostate [TimeEvolution](#) class to propagate the field data in time in a given order of the HE weak-field expansion

Functions

- void [linear1DProp](#) ([LatticePatch](#) *data, [N_Vector](#) u, [N_Vector](#) udot, int *c)
Maxwell propagation function for 1D – only for reference.
- void [nonlinear1DProp](#) ([LatticePatch](#) *data, [N_Vector](#) u, [N_Vector](#) udot, int *c)
HE propagation function for 1D.
- void [linear2DProp](#) ([LatticePatch](#) *data, [N_Vector](#) u, [N_Vector](#) udot, int *c)
Maxwell propagation function for 2D – only for reference.
- void [nonlinear2DProp](#) ([LatticePatch](#) *data, [N_Vector](#) u, [N_Vector](#) udot, int *c)
HE propagation function for 2D.
- void [linear3DProp](#) ([LatticePatch](#) *data, [N_Vector](#) u, [N_Vector](#) udot, int *c)
Maxwell propagation function for 3D – only for reference.
- void [nonlinear3DProp](#) ([LatticePatch](#) *data, [N_Vector](#) u, [N_Vector](#) udot, int *c)
HE propagation function for 3D.

6.30.1 Detailed Description

Functions to propagate data vectors in time according to Maxwell's equations, and various orders in the HE weak-field expansion.

Definition in file [TimeEvolutionFunctions.h](#).

6.30.2 Function Documentation

6.30.2.1 linear1DProp()

```
void linear1DProp (
    LatticePatch * data,
    N_Vector u,
    N_Vector udot,
    int * c )
```

Maxwell propagation function for 1D – only for reference.

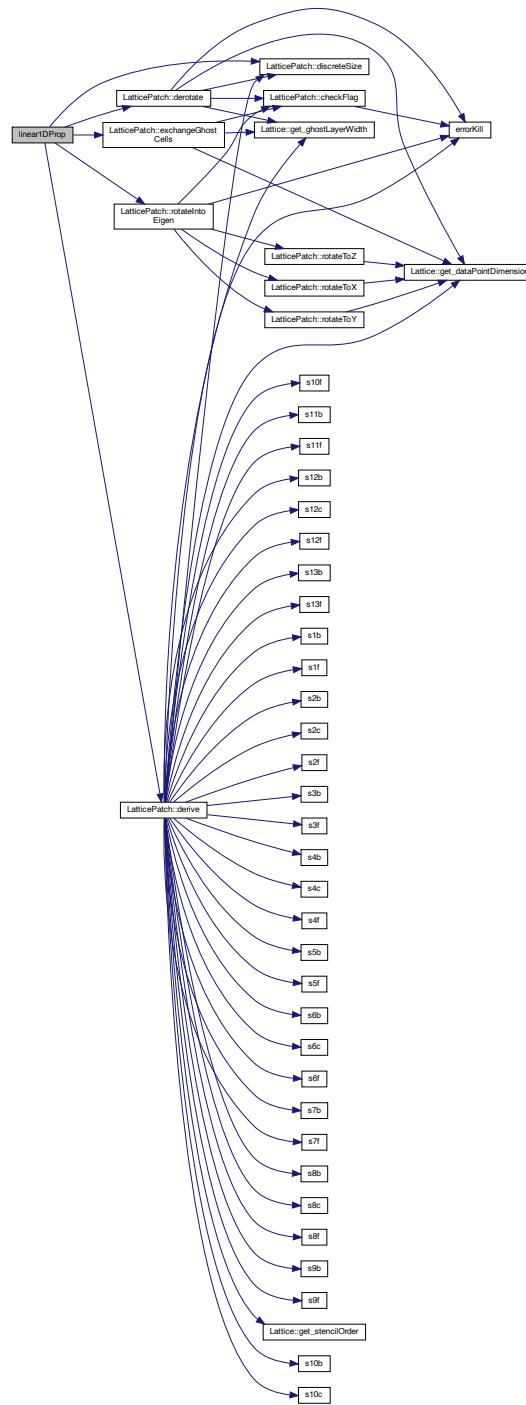
Maxwell propagation function for 1D – only for reference.

Definition at line 43 of file [TimeEvolutionFunctions.cpp](#).

```
00043 {
00044
00045 // pointers to temporal and spatial derivative data
00046 sunrealtype *duData = data->duData;
00047 sunrealtype *dxData = data->buffData[1 - 1];
00048
00049 // sequence along any dimension according to the scheme:
00050 data->exchangeGhostCells(1); // -> exchange halos
00051 data->rotateIntoEigen(
00052     1); // -> rotate all data to prepare derivative operation
00053 data->derive(1); // -> perform derivative approximation operation on it
00054 data->derotate(
00055     1, dxData); // -> derotate derived data for ensuing time-evolution
00056
00057 const sunindextype totalNP = data->discreteSize();
00058 sunindextype pp = 0;
00059 for (sunindextype i = 0; i < totalNP; i++) {
00060     pp = i * 6;
00061     /*
00062     simple vacuum Maxwell equations for the temporal derivatives using the
00063     spatial derivative only in x-direction without polarization or
00064     magnetization terms
00065     */
00066     duData[pp + 0] = 0;
00067     duData[pp + 1] = -dxData[pp + 5];
00068     duData[pp + 2] = dxData[pp + 4];
00069     duData[pp + 3] = 0;
00070     duData[pp + 4] = dxData[pp + 2];
00071     duData[pp + 5] = -dxData[pp + 1];
00072 }
00073 }
```

References [LatticePatch::buffData](#), [LatticePatch::derive\(\)](#), [LatticePatch::derotate\(\)](#), [LatticePatch::discreteSize\(\)](#), [LatticePatch::duData](#), [LatticePatch::exchangeGhostCells\(\)](#), and [LatticePatch::rotateIntoEigen\(\)](#).

Here is the call graph for this function:



6.30.2.2 linear2DProp()

```

void linear2DProp (
    LatticePatch * data,

```

```

    N_Vector u,
    N_Vector udot,
    int * c )

```

Maxwell propagation function for 2D – only for reference.

Maxwell propagation function for 2D – only for reference.

Definition at line 286 of file [TimeEvolutionFunctions.cpp](#).

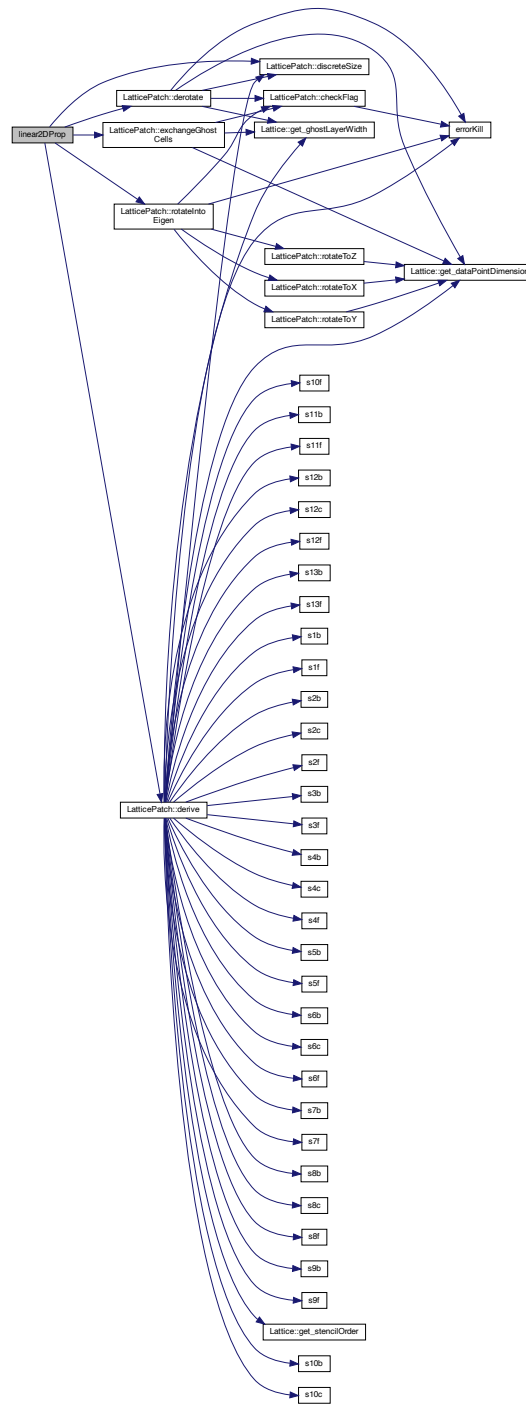
```

00286                                     {
00287
00288     sunrealtype *duData = data->duData;
00289     sunrealtype *dxData = data->buffData[1 - 1];
00290     sunrealtype *dyData = data->buffData[2 - 1];
00291
00292     data->exchangeGhostCells(1);
00293     data->rotateIntoEigen(1);
00294     data->derive(1);
00295     data->derotate(1, dxData);
00296     data->exchangeGhostCells(2);
00297     data->rotateIntoEigen(2);
00298     data->derive(2);
00299     data->derotate(2, dyData);
00300
00301     const sunindextype totalNP = data->discreteSize();
00302     sunindextype pp = 0;
00303     for (sunindextype i = 0; i < totalNP; i++) {
00304         pp = i * 6;
00305         duData[pp + 0] = dyData[pp + 5];
00306         duData[pp + 1] = -dxData[pp + 5];
00307         duData[pp + 2] = -dyData[pp + 3] + dxData[pp + 4];
00308         duData[pp + 3] = -dyData[pp + 2];
00309         duData[pp + 4] = dxData[pp + 2];
00310         duData[pp + 5] = dyData[pp + 0] - dxData[pp + 1];
00311     }
00312 }

```

References [LatticePatch::buffData](#), [LatticePatch::derive\(\)](#), [LatticePatch::derotate\(\)](#), [LatticePatch::discreteSize\(\)](#), [LatticePatch::duData](#), [LatticePatch::exchangeGhostCells\(\)](#), and [LatticePatch::rotateIntoEigen\(\)](#).

Here is the call graph for this function:



6.30.2.3 linear3DProp()

```
void linear3DProp (
    LatticePatch * data,
```

```

    N_Vector u,
    N_Vector udot,
    int * c )

```

Maxwell propagation function for 3D – only for reference.

Maxwell propagation function for 3D – only for reference.

Definition at line 494 of file [TimeEvolutionFunctions.cpp](#).

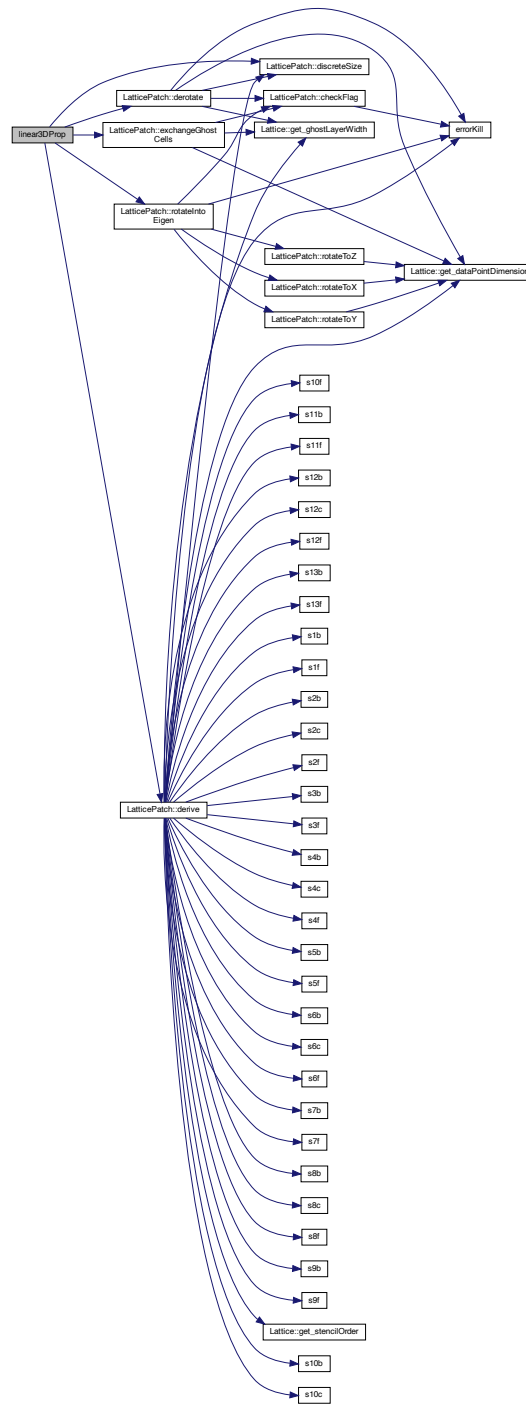
```

00494
00495
00496     sunrealtype *duData = data->duData;
00497     sunrealtype *dxData = data->buffData[1 - 1];
00498     sunrealtype *dyData = data->buffData[2 - 1];
00499     sunrealtype *dzData = data->buffData[3 - 1];
00500
00501     data->exchangeGhostCells(1);
00502     data->rotateIntoEigen(1);
00503     data->derive(1);
00504     data->derotate(1, dxData);
00505     data->exchangeGhostCells(2);
00506     data->rotateIntoEigen(2);
00507     data->derive(2);
00508     data->derotate(2, dyData);
00509     data->exchangeGhostCells(3);
00510     data->rotateIntoEigen(3);
00511     data->derive(3);
00512     data->derotate(3, dzData);
00513
00514     const sunindextype totalNP = data->discreteSize();
00515     sunindextype pp = 0;
00516     for (sunindextype i = 0; i < totalNP; i++) {
00517         pp = i * 6;
00518         duData[pp + 0] = dyData[pp + 5] - dzData[pp + 4];
00519         duData[pp + 1] = dzData[pp + 3] - dxData[pp + 5];
00520         duData[pp + 2] = dxData[pp + 4] - dyData[pp + 3];
00521         duData[pp + 3] = -dyData[pp + 2] + dzData[pp + 1];
00522         duData[pp + 4] = -dzData[pp + 0] + dxData[pp + 2];
00523         duData[pp + 5] = -dxData[pp + 1] + dyData[pp + 0];
00524     }
00525 }

```

References [LatticePatch::buffData](#), [LatticePatch::derive\(\)](#), [LatticePatch::derotate\(\)](#), [LatticePatch::discreteSize\(\)](#), [LatticePatch::duData](#), [LatticePatch::exchangeGhostCells\(\)](#), and [LatticePatch::rotateIntoEigen\(\)](#).

Here is the call graph for this function:



6.30.2.4 nonlinear1DProp()

```

void nonlinear1DProp (
    LatticePatch * data,

```

```

    N_Vector u,
    N_Vector udot,
    int * c )

```

HE propagation function for 1D.

HE propagation function for 1D.

Definition at line 76 of file [TimeEvolutionFunctions.cpp](#).

```

00076                                                                 {
00077
00078     // NVector pointers to provided field values and their temp. derivatives
00079     sunrealtype *udata = NV_DATA_P(u),
00080                 *dudata = NV_DATA_P(udot);
00081
00082     // pointer to spatial derivatives via patch data
00083     sunrealtype *dxData = data->buffData[1 - 1];
00084
00085     // same sequence as in the linear case
00086     data->exchangeGhostCells(1);
00087     data->rotateIntoEigen(1);
00088     data->derive(1);
00089     data->derotate(1, dxData);
00090
00091     /*
00092     F and G are nonzero in the nonlinear case,
00093     polarization and magnetization derivatives
00094     w.r.t. E- and B-field go into the e.o.m.
00095     */
00096     static sunrealtype f, g; // em field invariants F, G
00097     // derivatives of HE Lagrangian w.r.t. field invariants
00098     static sunrealtype lf, lff, lfg, lg, lgg;
00099     // matrix to hold derivatives of polarization and magnetization
00100     static std::array<sunrealtype, 21> JMM;
00101     // array to hold E^2 and B^2 components
00102     static std::array<sunrealtype, 6> Quad;
00103     // array to hold intermediate temp. derivatives of E and B
00104     static std::array<sunrealtype, 6> h;
00105     // determinant needed for explicit matrix inversion
00106     static sunrealtype detC = nan("0x12345");
00107
00108     // number of points in the patch
00109     const sunindextype totalNP = data->discreteSize();
00110     #pragma omp parallel for default(none) \
00111     private(JMM, Quad, h, detC) \
00112     shared(totalNP, c, f, g, lf, lff, lfg, lg, lgg, udata, dudata, dxData) \
00113     schedule(static)
00114     for (sunindextype pp = 0; pp < totalNP * 6;
00115         pp += 6) { // loop over all 6dim points in the patch
00116         // em field Lorentz invariants F and G
00117         f = 0.5 * ((Quad[0] = udata[pp] * udata[pp]) +
00118                 (Quad[1] = udata[pp + 1] * udata[pp + 1]) +
00119                 (Quad[2] = udata[pp + 2] * udata[pp + 2]) -
00120                 (Quad[3] = udata[pp + 3] * udata[pp + 3]) -
00121                 (Quad[4] = udata[pp + 4] * udata[pp + 4]) -
00122                 (Quad[5] = udata[pp + 5] * udata[pp + 5]));
00123         g = udata[pp] * udata[pp + 3] + udata[pp + 1] * udata[pp + 4] +
00124             udata[pp + 2] * udata[pp + 5];
00125         // process/expansion order and corresponding derivative values of L
00126         // w.r.t. F, G
00127         switch (*c) {
00128         case 0: // linear Maxwell vacuum
00129             lf = 0;
00130             lff = 0;
00131             lfg = 0;
00132             lg = 0;
00133             lgg = 0;
00134             break;
00135         case 1: // only 4-photon processes
00136             lf = 0.000206527095658582755255648 * f;
00137             lff = 0.000206527095658582755255648;
00138             lfg = 0;
00139             lg = 0.0003614224174025198216973841 * g;
00140             lgg = 0.0003614224174025198216973841;
00141             break;
00142         case 2: // only 6-photon processes
00143             lf = 0.000354046449700427580438254 * f * f +
00144                 0.000191775160254398272737387 * g * g;
00145             lff = 0.0007080928994008551608765075 * f;
00146             lfg = 0.0003835503205087965454747749 * g;
00147             lg = 0.0003835503205087965454747749 * f * g;
00148             lgg = 0.0003835503205087965454747749 * f;
00149             break;

```

```

00150     case 3: // 4- and 6-photon processes
00151         lf = (0.000206527095658582755255648 + 0.000354046449700427580438254 * f) *
00152             f +
00153             0.000191775160254398272737387 * g * g;
00154         lff = 0.000206527095658582755255648 + 0.0007080928994008551608765075 * f;
00155         lfg = 0.0003835503205087965454747749 * g;
00156         lg = (0.0003614224174025198216973841 +
00157             0.0003835503205087965454747749 * f) *
00158             g;
00159         lgg = 0.0003614224174025198216973841 + 0.0003835503205087965454747749 * f;
00160         break;
00161     default:
00162         errorKill(
00163             "You need to specify a correct order in the weak-field expansion.");
00164     }
00165
00166     // derivatives of polarization and magnetization w.r.t. E and B
00167     // Jpx(Ex)
00168     JMM[0] = lf + lff * Quad[0] +
00169         udata[3 + pp] * (2 * lfg * udata[pp] + lgg * udata[3 + pp]);
00170     // Jpx(Ey)
00171     JMM[1] =
00172         lff * udata[pp] * udata[1 + pp] + lfg * udata[1 + pp] * udata[3 + pp] +
00173         lfg * udata[pp] * udata[4 + pp] + lgg * udata[3 + pp] * udata[4 + pp];
00174     // Jpy(Ey)
00175     JMM[2] = lf + lff * Quad[1] +
00176         udata[4 + pp] * (2 * lfg * udata[1 + pp] + lgg * udata[4 + pp]);
00177     // Jpx(Ez) = Jpz(Ex)
00178     JMM[3] =
00179         lff * udata[pp] * udata[2 + pp] + lfg * udata[2 + pp] * udata[3 + pp] +
00180         lfg * udata[pp] * udata[5 + pp] + lgg * udata[3 + pp] * udata[5 + pp];
00181     // Jpy(Ez) = Jpz(Ey)
00182     JMM[4] = lff * udata[1 + pp] * udata[2 + pp] +
00183         lfg * udata[2 + pp] * udata[4 + pp] +
00184         lfg * udata[1 + pp] * udata[5 + pp] +
00185         lgg * udata[4 + pp] * udata[5 + pp];
00186     // Jpz(Ez)
00187     JMM[5] = lf + lff * Quad[2] +
00188         udata[5 + pp] * (2 * lfg * udata[2 + pp] + lgg * udata[5 + pp]);
00189     // Jpx(Bx) = Jmx(Ex)
00190     JMM[6] = lg + lfg * (Quad[0] - Quad[3 + 0]) +
00191         (-lff + lgg) * udata[pp] * udata[3 + pp];
00192     // Jpy(Bx) = Jmx(Ey)
00193     JMM[7] = -(udata[3 + pp] * (lff * udata[1 + pp] + lfg * udata[4 + pp])) +
00194         udata[pp] * (lfg * udata[1 + pp] + lgg * udata[4 + pp]);
00195     // Jpz(Bx) = Jmx(Ez)
00196     JMM[8] = -(udata[3 + pp] * (lff * udata[2 + pp] + lfg * udata[5 + pp])) +
00197         udata[pp] * (lfg * udata[2 + pp] + lgg * udata[5 + pp]);
00198     // Jmx(Bx)
00199     JMM[9] = -lf + lgg * Quad[0] +
00200         udata[3 + pp] * (-2 * lfg * udata[pp] + lff * udata[3 + pp]);
00201     // Jpx(By) = Jmy(Ex)
00202     JMM[10] = udata[1 + pp] * (lfg * udata[pp] + lgg * udata[3 + pp]) -
00203         (lff * udata[pp] + lfg * udata[3 + pp]) * udata[4 + pp];
00204     // Jpy(By) = Jmy(Ey)
00205     JMM[11] = lg + lfg * (Quad[1] - Quad[4 + 0]) +
00206         (-lff + lgg) * udata[1 + pp] * udata[4 + pp];
00207     // Jpz(By) = Jmy(Ez)
00208     JMM[12] = -(udata[4 + pp] * (lff * udata[2 + pp] + lfg * udata[5 + pp])) +
00209         udata[1 + pp] * (lfg * udata[2 + pp] + lgg * udata[5 + pp]);
00210     // Jmx(By) = Jmy(Bx)
00211     JMM[13] = lgg * udata[pp] * udata[1 + pp] +
00212         lff * udata[3 + pp] * udata[4 + pp] -
00213         lfg * (udata[1 + pp] * udata[3 + pp] + udata[pp] * udata[4 + pp]);
00214     // Jmy(By)
00215     JMM[14] = -lf + lgg * Quad[1] +
00216         udata[4 + pp] * (-2 * lfg * udata[1 + pp] + lff * udata[4 + pp]);
00217     // Jmz(Ex) = Jpx(Bz)
00218     JMM[15] = udata[2 + pp] * (lfg * udata[pp] + lgg * udata[3 + pp]) -
00219         (lff * udata[pp] + lfg * udata[3 + pp]) * udata[5 + pp];
00220     // Jmz(Ey) = Jpy(Bz)
00221     JMM[16] = udata[2 + pp] * (lfg * udata[1 + pp] + lgg * udata[4 + pp]) -
00222         (lff * udata[1 + pp] + lfg * udata[4 + pp]) * udata[5 + pp];
00223     // Jpz(Bz) = Jmz(Ez)
00224     JMM[17] = lg + lfg * (Quad[2] - Quad[5 + 0]) +
00225         (-lff + lgg) * udata[2 + pp] * udata[5 + pp];
00226     // Jmz(Bx) = Jmx(Bz)
00227     JMM[18] = lgg * udata[pp] * udata[2 + pp] +
00228         lff * udata[3 + pp] * udata[5 + pp] -
00229         lfg * (udata[2 + pp] * udata[3 + pp] + udata[pp] * udata[5 + pp]);
00230     // Jmy(Bz) = Jmz(By)
00231     JMM[19] =
00232         lgg * udata[1 + pp] * udata[2 + pp] +
00233         lff * udata[4 + pp] * udata[5 + pp] -
00234         lfg * (udata[2 + pp] * udata[4 + pp] + udata[1 + pp] * udata[5 + pp]);
00235     // Jmz(Bz)
00236     JMM[20] = -lf + lgg * Quad[2] +

```

```

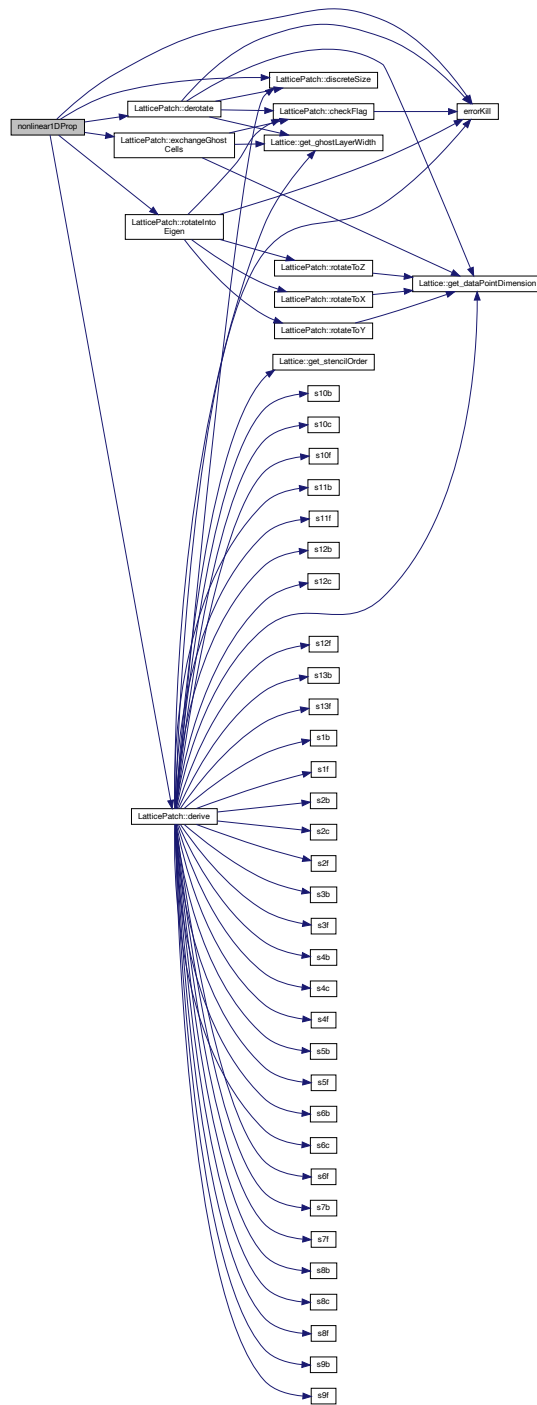
00237         udata[5 + pp] * (-2 * lfg * udata[2 + pp] + lff * udata[5 + pp]);
00238
00239     // apply Z
00240     // top block: -QJm(E)*E, Q-QJm(B)*B
00241     h[0] = 0;
00242     h[1] = dxData[pp] * JMM[15] + dxData[1 + pp] * JMM[16] +
00243           dxData[2 + pp] * JMM[17] + dxData[3 + pp] * JMM[18] +
00244           dxData[4 + pp] * JMM[19] + dxData[5 + pp] * (-1 + JMM[20]);
00245     h[2] = -(dxData[pp] * JMM[10]) - dxData[1 + pp] * JMM[11] -
00246           dxData[2 + pp] * JMM[12] - dxData[3 + pp] * JMM[13] +
00247           dxData[4 + pp] * (1 - JMM[14]) - dxData[5 + pp] * JMM[19];
00248     // bottom blocks: -Q*E
00249     h[3] = 0;
00250     h[4] = dxData[2 + pp];
00251     h[5] = -dxData[1 + pp];
00252     // (1+A)^-1 applies only to E components
00253     // -Jp(B)*B
00254     h[0] -= h[3] * JMM[6] + h[4] * JMM[10] + h[5] * JMM[15];
00255     h[1] -= h[3] * JMM[7] + h[4] * JMM[11] + h[5] * JMM[16];
00256     h[2] -= h[3] * JMM[8] + h[4] * JMM[12] + h[5] * JMM[17];
00257     // apply C^-1 explicitly, with C=1+Jp(E)
00258     dudata[pp + 0] =
00259         h[2] * (-JMM[3] * (1 + JMM[2])) + JMM[1] * JMM[4] +
00260         h[1] * (JMM[3] * JMM[4] - JMM[1] * (1 + JMM[5])) +
00261         h[0] * (1 - JMM[4] * JMM[4] + JMM[5] + JMM[2] * (1 + JMM[5]));
00262     dudata[pp + 1] =
00263         h[2] * (JMM[3] * JMM[1] - (1 + JMM[0]) * JMM[4]) +
00264         h[1] * (1 - JMM[3] * JMM[3] + JMM[5] + JMM[0] * (1 + JMM[5])) +
00265         h[0] * (JMM[4] * JMM[3] - JMM[1] * (1 + JMM[5]));
00266     dudata[pp + 2] =
00267         h[2] * (1 - JMM[1] * JMM[1] + JMM[2] + JMM[0] * (1 + JMM[2])) +
00268         h[1] * (JMM[1] * JMM[3] - (1 + JMM[0]) * JMM[4]) +
00269         h[0] * (-((1 + JMM[2]) * JMM[3]) + JMM[1] * JMM[4]);
00270     detC = // determinant of C
00271         -((1 + JMM[2]) * (-1 + JMM[3] * JMM[3])) +
00272         (JMM[3] * JMM[1] - JMM[4]) * JMM[4] + JMM[5] + JMM[2] * JMM[5] +
00273         JMM[0] * (1 + JMM[2] - JMM[4] * JMM[4] + (1 + JMM[2]) * JMM[5]) -
00274         JMM[1] * (-JMM[4] * JMM[3]) + JMM[1] * (1 + JMM[5]);
00275     dudata[pp + 0] /= detC;
00276     dudata[pp + 1] /= detC;
00277     dudata[pp + 2] /= detC;
00278     dudata[pp + 3] = h[3];
00279     dudata[pp + 4] = h[4];
00280     dudata[pp + 5] = h[5];
00281 }
00282 return;
00283 }

```

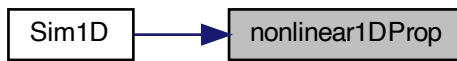
References [LatticePatch::buffData](#), [LatticePatch::derive\(\)](#), [LatticePatch::derotate\(\)](#), [LatticePatch::discreteSize\(\)](#), [errorKill\(\)](#), [LatticePatch::exchangeGhostCells\(\)](#), and [LatticePatch::rotateIntoEigen\(\)](#).

Referenced by [Sim1D\(\)](#).

Here is the call graph for this function:



Here is the caller graph for this function:



6.30.2.5 nonlinear2DProp()

```

void nonlinear2DProp (
    LatticePatch * data,
    N_Vector u,
    N_Vector udot,
    int * c )
  
```

HE propagation function for 2D.

HE propagation function for 2D.

Definition at line 315 of file [TimeEvolutionFunctions.cpp](#).

```

00315
00316
00317     sunrealtype *udata = NV_DATA_P(u),
00318               *dudata = NV_DATA_P(udot);
00319
00320     sunrealtype *dxData = data->buffData[1 - 1];
00321     sunrealtype *dyData = data->buffData[2 - 1];
00322
00323     data->exchangeGhostCells(1);
00324     data->rotateIntoEigen(1);
00325     data->derive(1);
00326     data->derotate(1, dxData);
00327     data->exchangeGhostCells(2);
00328     data->rotateIntoEigen(2);
00329     data->derive(2);
00330     data->derotate(2, dyData);
00331
00332     static sunrealtype f, g;
00333     static sunrealtype lf, lff, lfg, lg, lgg;
00334     static std::array<sunrealtype, 21> JMM;
00335     static std::array<sunrealtype, 6> Quad;
00336     static std::array<sunrealtype, 6> h;
00337     static sunrealtype detC;
00338
00339     const sunindextype totalNP = data->discreteSize();
00340     #pragma omp parallel for default(none) \
00341     private(JMM, Quad, h, detC) \
00342     shared(totalNP, c, f, g, lf, lff, lfg, lg, lgg, udata, dudata, \
00343            dxData, dyData) \
00344     schedule(static)
00345     for (sunindextype pp = 0; pp < totalNP * 6; pp += 6) {
00346         f = 0.5 * ((Quad[0] = udata[pp] * udata[pp]) +
00347                  (Quad[1] = udata[pp + 1] * udata[pp + 1]) +
00348                  (Quad[2] = udata[pp + 2] * udata[pp + 2]) -
00349                  (Quad[3] = udata[pp + 3] * udata[pp + 3]) -
00350                  (Quad[4] = udata[pp + 4] * udata[pp + 4]) -
00351                  (Quad[5] = udata[pp + 5] * udata[pp + 5]));
00352         g = udata[pp] * udata[pp + 3] + udata[pp + 1] * udata[pp + 4] +
00353             udata[pp + 2] * udata[pp + 5];
00354         switch (*c) {
00355             case 0:
00356                 lf = 0;
00357                 lff = 0;
  
```



```

00358     lfg = 0;
00359     lg = 0;
00360     lgg = 0;
00361     break;
00362 case 1:
00363     lf = 0.000206527095658582755255648 * f;
00364     lff = 0.000206527095658582755255648;
00365     lfg = 0;
00366     lg = 0.0003614224174025198216973841 * g;
00367     lgg = 0.0003614224174025198216973841;
00368     break;
00369 case 2:
00370     lf = 0.000354046449700427580438254 * f * f +
00371         0.000191775160254398272737387 * g * g;
00372     lff = 0.0007080928994008551608765075 * f;
00373     lfg = 0.0003835503205087965454747749 * g;
00374     lg = 0.0003835503205087965454747749 * f * g;
00375     lgg = 0.0003835503205087965454747749 * f;
00376     break;
00377 case 3:
00378     lf = (0.000206527095658582755255648 + 0.000354046449700427580438254 * f) *
00379         f +
00380         0.000191775160254398272737387 * g * g;
00381     lff = 0.000206527095658582755255648 + 0.000708092899400855160876508 * f;
00382     lfg = 0.0003835503205087965454747749 * g;
00383     lg = (0.000361422417402519821697384 + 0.000383550320508796545474775 * f) *
00384         g;
00385     lgg = 0.000361422417402519821697384 + 0.000383550320508796545474775 * f;
00386     break;
00387 default:
00388     errorKill(
00389         "You need to specify a correct order in the weak-field expansion.");
00390 }
00391
00392 JMM[0] = lf + lff * Quad[0] +
00393     udata[3 + pp] * (2 * lfg * udata[pp] + lgg * udata[3 + pp]);
00394 JMM[1] =
00395     lff * udata[pp] * udata[1 + pp] + lfg * udata[1 + pp] * udata[3 + pp] +
00396     lfg * udata[pp] * udata[4 + pp] + lgg * udata[3 + pp] * udata[4 + pp];
00397 JMM[2] = lf + lff * Quad[1] +
00398     udata[4 + pp] * (2 * lfg * udata[1 + pp] + lgg * udata[4 + pp]);
00399 JMM[3] =
00400     lff * udata[pp] * udata[2 + pp] + lfg * udata[2 + pp] * udata[3 + pp] +
00401     lfg * udata[pp] * udata[5 + pp] + lgg * udata[3 + pp] * udata[5 + pp];
00402 JMM[4] = lff * udata[1 + pp] * udata[2 + pp] +
00403     lfg * udata[2 + pp] * udata[4 + pp] +
00404     lfg * udata[1 + pp] * udata[5 + pp] +
00405     lgg * udata[4 + pp] * udata[5 + pp];
00406 JMM[5] = lf + lff * Quad[2] +
00407     udata[5 + pp] * (2 * lfg * udata[2 + pp] + lgg * udata[5 + pp]);
00408 JMM[6] = lg + lfg * (Quad[0] - Quad[3 + 0]) +
00409     (-lff + lgg) * udata[pp] * udata[3 + pp];
00410 JMM[7] = -(udata[3 + pp] * (lff * udata[1 + pp] + lfg * udata[4 + pp])) +
00411     udata[pp] * (lfg * udata[1 + pp] + lgg * udata[4 + pp]);
00412 JMM[8] = -(udata[3 + pp] * (lff * udata[2 + pp] + lfg * udata[5 + pp])) +
00413     udata[pp] * (lfg * udata[2 + pp] + lgg * udata[5 + pp]);
00414 JMM[9] = -lf + lgg * Quad[0] +
00415     udata[3 + pp] * (-2 * lfg * udata[pp] + lff * udata[3 + pp]);
00416 JMM[10] = udata[1 + pp] * (lfg * udata[pp] + lgg * udata[3 + pp]) -
00417     (lff * udata[pp] + lfg * udata[3 + pp]) * udata[4 + pp];
00418 JMM[11] = lg + lfg * (Quad[1] - Quad[4 + 0]) +
00419     (-lff + lgg) * udata[1 + pp] * udata[4 + pp];
00420 JMM[12] = -(udata[4 + pp] * (lff * udata[2 + pp] + lfg * udata[5 + pp])) +
00421     udata[1 + pp] * (lfg * udata[2 + pp] + lgg * udata[5 + pp]);
00422 JMM[13] = lgg * udata[pp] * udata[1 + pp] +
00423     lff * udata[3 + pp] * udata[4 + pp] -
00424     lfg * (udata[1 + pp] * udata[3 + pp] + udata[pp] * udata[4 + pp]);
00425 JMM[14] = -lf + lgg * Quad[1] +
00426     udata[4 + pp] * (-2 * lfg * udata[1 + pp] + lff * udata[4 + pp]);
00427 JMM[15] = udata[2 + pp] * (lfg * udata[pp] + lgg * udata[3 + pp]) -
00428     (lff * udata[pp] + lfg * udata[3 + pp]) * udata[5 + pp];
00429 JMM[16] = udata[2 + pp] * (lfg * udata[1 + pp] + lgg * udata[4 + pp]) -
00430     (lff * udata[1 + pp] + lfg * udata[4 + pp]) * udata[5 + pp];
00431 JMM[17] = lg + lfg * (Quad[2] - Quad[5 + 0]) +
00432     (-lff + lgg) * udata[2 + pp] * udata[5 + pp];
00433 JMM[18] = lgg * udata[pp] * udata[2 + pp] +
00434     lff * udata[3 + pp] * udata[5 + pp] -
00435     lfg * (udata[2 + pp] * udata[3 + pp] + udata[pp] * udata[5 + pp]);
00436 JMM[19] =
00437     lgg * udata[1 + pp] * udata[2 + pp] +
00438     lff * udata[4 + pp] * udata[5 + pp] -
00439     lfg * (udata[2 + pp] * udata[4 + pp] + udata[1 + pp] * udata[5 + pp]);
00440 JMM[20] = -lf + lgg * Quad[2] +
00441     udata[5 + pp] * (-2 * lfg * udata[2 + pp] + lff * udata[5 + pp]);
00442
00443 h[0] = 0;
00444 h[1] = dxData[pp] * JMM[15] + dxData[1 + pp] * JMM[16] +

```

```

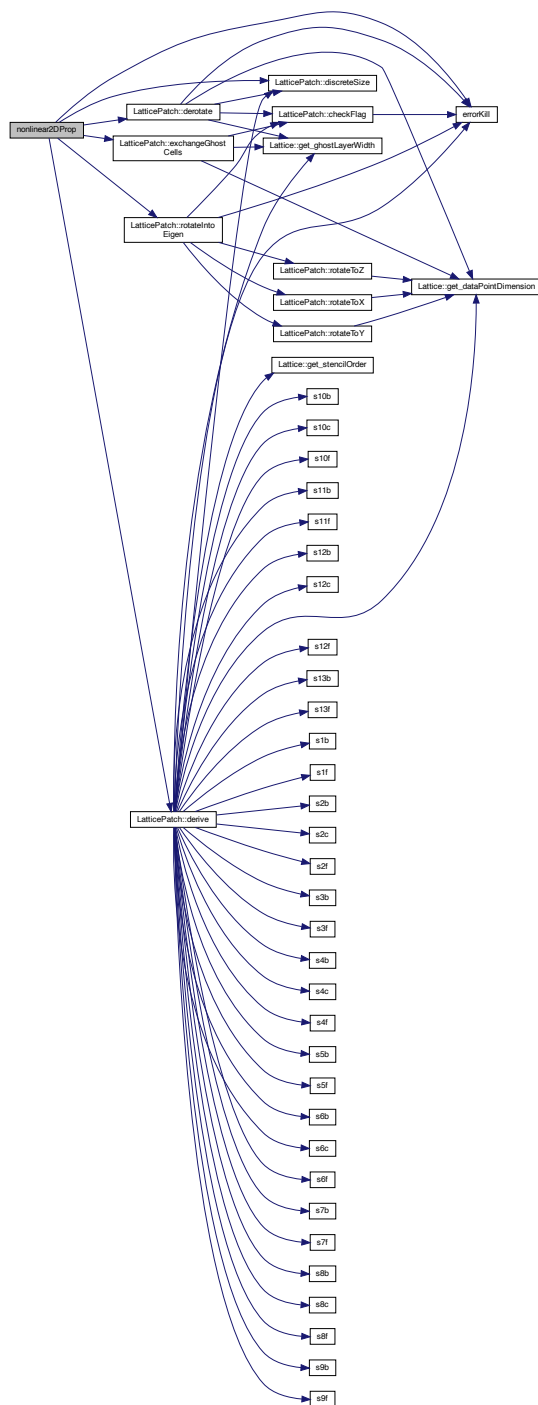
00445         dxData[2 + pp] * JMM[17] + dxData[3 + pp] * JMM[18] +
00446         dxData[4 + pp] * JMM[19] + dxData[5 + pp] * (-1 + JMM[20]));
00447     h[2] = -(dxData[pp] * JMM[10]) - dxData[1 + pp] * JMM[11] -
00448         dxData[2 + pp] * JMM[12] - dxData[3 + pp] * JMM[13] +
00449         dxData[4 + pp] * (1 - JMM[14]) - dxData[5 + pp] * JMM[19];
00450     h[3] = 0;
00451     h[4] = dxData[2 + pp];
00452     h[5] = -dxData[1 + pp];
00453     h[0] += -(dyData[pp] * JMM[15]) - dyData[1 + pp] * JMM[16] -
00454         dyData[2 + pp] * JMM[17] - dyData[3 + pp] * JMM[18] -
00455         dyData[4 + pp] * JMM[19] + dyData[5 + pp] * (1 - JMM[20]);
00456     h[1] += 0;
00457     h[2] += dyData[pp] * JMM[6] + dyData[1 + pp] * JMM[7] +
00458         dyData[2 + pp] * JMM[8] + dyData[3 + pp] * (-1 + JMM[9]) +
00459         dyData[4 + pp] * JMM[13] + dyData[5 + pp] * JMM[18];
00460     h[3] += -dyData[2 + pp];
00461     h[4] += 0;
00462     h[5] += dyData[pp];
00463     h[0] -= h[3] * JMM[6] + h[4] * JMM[10] + h[5] * JMM[15];
00464     h[1] -= h[3] * JMM[7] + h[4] * JMM[11] + h[5] * JMM[16];
00465     h[2] -= h[3] * JMM[8] + h[4] * JMM[12] + h[5] * JMM[17];
00466     dudata[pp + 0] =
00467         h[2] * (- (JMM[3] * (1 + JMM[2])) + JMM[1] * JMM[4]) +
00468         h[1] * (JMM[3] * JMM[4] - JMM[1] * (1 + JMM[5])) +
00469         h[0] * (1 - JMM[4] * JMM[4] + JMM[5] + JMM[2] * (1 + JMM[5]));
00470     dudata[pp + 1] =
00471         h[2] * (JMM[3] * JMM[1] - (1 + JMM[0]) * JMM[4]) +
00472         h[1] * (1 - JMM[3] * JMM[3] + JMM[5] + JMM[0] * (1 + JMM[5])) +
00473         h[0] * (JMM[4] * JMM[3] - JMM[1] * (1 + JMM[5]));
00474     dudata[pp + 2] =
00475         h[2] * (1 - JMM[1] * JMM[1] + JMM[2] + JMM[0] * (1 + JMM[2])) +
00476         h[1] * (JMM[1] * JMM[3] - (1 + JMM[0]) * JMM[4]) +
00477         h[0] * (-((1 + JMM[2]) * JMM[3]) + JMM[1] * JMM[4]);
00478     detC =
00479         -((1 + JMM[2]) * (-1 + JMM[3] * JMM[3])) +
00480         (JMM[3] * JMM[1] - JMM[4]) * JMM[4] + JMM[5] + JMM[2] * JMM[5] +
00481         JMM[0] * (1 + JMM[2] - JMM[4] * JMM[4] + (1 + JMM[2]) * JMM[5]) -
00482         JMM[1] * (- (JMM[4] * JMM[3]) + JMM[1] * (1 + JMM[5]));
00483     dudata[pp + 0] /= detC;
00484     dudata[pp + 1] /= detC;
00485     dudata[pp + 2] /= detC;
00486     dudata[pp + 3] = h[3];
00487     dudata[pp + 4] = h[4];
00488     dudata[pp + 5] = h[5];
00489 }
00490 return;
00491 }

```

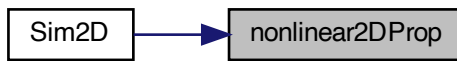
References [LatticePatch::buffData](#), [LatticePatch::derive\(\)](#), [LatticePatch::derotate\(\)](#), [LatticePatch::discreteSize\(\)](#), [errorKill\(\)](#), [LatticePatch::exchangeGhostCells\(\)](#), and [LatticePatch::rotateIntoEigen\(\)](#).

Referenced by [Sim2D\(\)](#).

Here is the call graph for this function:



Here is the caller graph for this function:



6.30.2.6 nonlinear3DProp()

```

void nonlinear3DProp (
    LatticePatch * data,
    N_Vector u,
    N_Vector udot,
    int * c )
  
```

HE propagation function for 3D.

HE propagation function for 3D.

Definition at line 528 of file [TimeEvolutionFunctions.cpp](#).

```

00528
00529
00530     sunrealtype *udata = NV_DATA_P(u),
00531     *dudata = NV_DATA_P(udot);
00532
00533     sunrealtype *dxData = data->buffData[1 - 1];
00534     sunrealtype *dyData = data->buffData[2 - 1];
00535     sunrealtype *dzData = data->buffData[3 - 1];
00536
00537     data->exchangeGhostCells(1);
00538     data->rotateIntoEigen(1);
00539     data->derive(1);
00540     data->derotate(1,dxData);
00541     data->exchangeGhostCells(2);
00542     data->rotateIntoEigen(2);
00543     data->derive(2);
00544     data->derotate(2,dyData);
00545     data->exchangeGhostCells(3);
00546     data->rotateIntoEigen(3);
00547     data->derive(3);
00548     data->derotate(3,dzData);
00549
00550     static sunrealtype f, g;
00551     static sunrealtype lf, lff, lfg, lg, lgg;
00552     static std::array<sunrealtype, 21> JMM;
00553     static std::array<sunrealtype, 6> Quad;
00554     static std::array<sunrealtype, 6> h;
00555     static sunrealtype detC = nan("0x12345");
00556
00557     const sunindextype totalNP = data->discreteSize();
00558     #pragma omp parallel for default(none) \
00559     private(JMM, Quad, h, detC) \
00560     shared(totalNP, c, f, g, lf, lff, lfg, lg, lgg, udata, dudata, \
00561           dxData, dyData, dzData) \
00562     schedule(static)
00563     for (sunindextype pp = 0; pp < totalNP * 6; pp += 6) {
00564         f = 0.5 * ((Quad[0] = udata[pp] * udata[pp]) +
00565                 (Quad[1] = udata[pp + 1] * udata[pp + 1]) +
00566                 (Quad[2] = udata[pp + 2] * udata[pp + 2]) -
00567                 (Quad[3] = udata[pp + 3] * udata[pp + 3]) -
00568                 (Quad[4] = udata[pp + 4] * udata[pp + 4]) -
00569                 (Quad[5] = udata[pp + 5] * udata[pp + 5]));
00570         g = udata[pp] * udata[pp + 3] + udata[pp + 1] * udata[pp + 4] +
  
```

```

00571         udata[pp + 2] * udata[pp + 5];
00572     switch (*c) {
00573     case 0:
00574         lf = 0;
00575         lff = 0;
00576         lfg = 0;
00577         lg = 0;
00578         lgg = 0;
00579         break;
00580     case 1:
00581         lf = 0.000206527095658582755255648 * f;
00582         lff = 0.000206527095658582755255648;
00583         lfg = 0;
00584         lg = 0.0003614224174025198216973841 * g;
00585         lgg = 0.0003614224174025198216973841;
00586         break;
00587     case 2:
00588         lf = 0.000354046449700427580438254 * f * f +
00589             0.000191775160254398272737387 * g * g;
00590         lff = 0.0007080928994008551608765075 * f;
00591         lfg = 0.0003835503205087965454747749 * g;
00592         lg = 0.0003835503205087965454747749 * f * g;
00593         lgg = 0.0003835503205087965454747749 * f;
00594         break;
00595     case 3:
00596         lf = (0.000206527095658582755255648 + 0.000354046449700427580438254 * f) *
00597             f +
00598             0.000191775160254398272737387 * g * g;
00599         lff = 0.000206527095658582755255648 + 0.000708092899400855160876508 * f;
00600         lfg = 0.0003835503205087965454747749 * g;
00601         lg = (0.000361422417402519821697384 + 0.000383550320508796545474775 * f) *
00602             g;
00603         lgg = 0.000361422417402519821697384 + 0.000383550320508796545474775 * f;
00604         break;
00605     default:
00606         errorKill(
00607             "You need to specify a correct order in the weak-field expansion.");
00608     }
00609
00610     JMM[0] = lf + lff * Quad[0] +
00611         udata[3 + pp] * (2 * lfg * udata[pp] + lgg * udata[3 + pp]);
00612     JMM[1] =
00613         lff * udata[pp] * udata[1 + pp] + lfg * udata[1 + pp] * udata[3 + pp] +
00614         lfg * udata[pp] * udata[4 + pp] + lgg * udata[3 + pp] * udata[4 + pp];
00615     JMM[2] = lf + lff * Quad[1] +
00616         udata[4 + pp] * (2 * lfg * udata[1 + pp] + lgg * udata[4 + pp]);
00617     JMM[3] =
00618         lff * udata[pp] * udata[2 + pp] + lfg * udata[2 + pp] * udata[3 + pp] +
00619         lfg * udata[pp] * udata[5 + pp] + lgg * udata[3 + pp] * udata[5 + pp];
00620     JMM[4] = lff * udata[1 + pp] * udata[2 + pp] +
00621         lfg * udata[2 + pp] * udata[4 + pp] +
00622         lfg * udata[1 + pp] * udata[5 + pp] +
00623         lgg * udata[4 + pp] * udata[5 + pp];
00624     JMM[5] = lf + lff * Quad[2] +
00625         udata[5 + pp] * (2 * lfg * udata[2 + pp] + lgg * udata[5 + pp]);
00626     JMM[6] = lg + lfg * (Quad[0] - Quad[3 + 0]) +
00627         (-lff + lgg) * udata[pp] * udata[3 + pp];
00628     JMM[7] = -(udata[3 + pp] * (lff * udata[1 + pp] + lfg * udata[4 + pp])) +
00629         udata[pp] * (lfg * udata[1 + pp] + lgg * udata[4 + pp]);
00630     JMM[8] = -(udata[3 + pp] * (lff * udata[2 + pp] + lfg * udata[5 + pp])) +
00631         udata[pp] * (lfg * udata[2 + pp] + lgg * udata[5 + pp]);
00632     JMM[9] = -lf + lgg * Quad[0] +
00633         udata[3 + pp] * (-2 * lfg * udata[pp] + lff * udata[3 + pp]);
00634     JMM[10] = udata[1 + pp] * (lfg * udata[pp] + lgg * udata[3 + pp]) -
00635         (lff * udata[pp] + lfg * udata[3 + pp]) * udata[4 + pp];
00636     JMM[11] = lg + lfg * (Quad[1] - Quad[4 + 0]) +
00637         (-lff + lgg) * udata[1 + pp] * udata[4 + pp];
00638     JMM[12] = -(udata[4 + pp] * (lff * udata[2 + pp] + lfg * udata[5 + pp])) +
00639         udata[1 + pp] * (lfg * udata[2 + pp] + lgg * udata[5 + pp]);
00640     JMM[13] = lgg * udata[pp] * udata[1 + pp] +
00641         lff * udata[3 + pp] * udata[4 + pp] -
00642         lfg * (udata[1 + pp] * udata[3 + pp] + udata[pp] * udata[4 + pp]);
00643     JMM[14] = -lf + lgg * Quad[1] +
00644         udata[4 + pp] * (-2 * lfg * udata[1 + pp] + lff * udata[4 + pp]);
00645     JMM[15] = udata[2 + pp] * (lfg * udata[pp] + lgg * udata[3 + pp]) -
00646         (lff * udata[pp] + lfg * udata[3 + pp]) * udata[5 + pp];
00647     JMM[16] = udata[2 + pp] * (lfg * udata[1 + pp] + lgg * udata[4 + pp]) -
00648         (lff * udata[1 + pp] + lfg * udata[4 + pp]) * udata[5 + pp];
00649     JMM[17] = lg + lfg * (Quad[2] - Quad[5 + 0]) +
00650         (-lff + lgg) * udata[2 + pp] * udata[5 + pp];
00651     JMM[18] = lgg * udata[pp] * udata[2 + pp] +
00652         lff * udata[3 + pp] * udata[5 + pp] -
00653         lfg * (udata[2 + pp] * udata[3 + pp] + udata[pp] * udata[5 + pp]);
00654     JMM[19] =
00655         lgg * udata[1 + pp] * udata[2 + pp] +
00656         lff * udata[4 + pp] * udata[5 + pp] -
00657         lfg * (udata[2 + pp] * udata[4 + pp] + udata[1 + pp] * udata[5 + pp]);

```

```

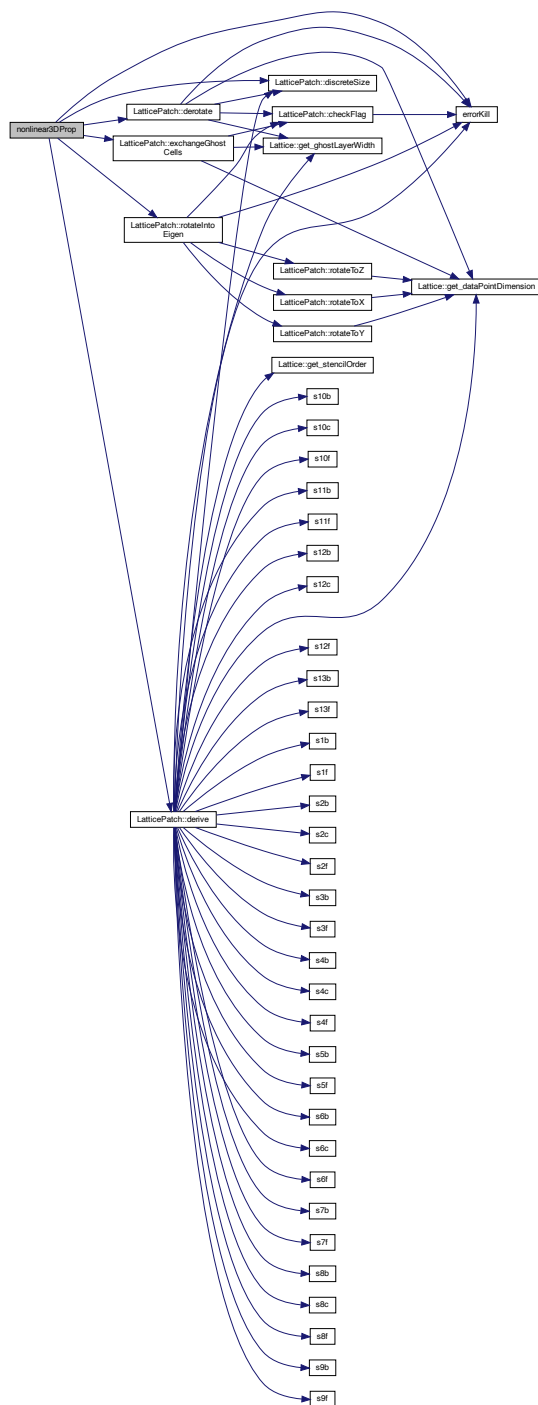
00658     JMM[20] = -lf + lgg * Quad[2] +
00659             udata[5 + pp] * (-2 * lfg * udata[2 + pp] + lff * udata[5 + pp]);
00660
00661     h[0] = 0;
00662     h[1] = dxData[pp] * JMM[15] + dxData[1 + pp] * JMM[16] +
00663           dxData[2 + pp] * JMM[17] + dxData[3 + pp] * JMM[18] +
00664           dxData[4 + pp] * JMM[19] + dxData[5 + pp] * (-1 + JMM[20]);
00665     h[2] = -(dxData[pp] * JMM[10]) - dxData[1 + pp] * JMM[11] -
00666           dxData[2 + pp] * JMM[12] - dxData[3 + pp] * JMM[13] +
00667           dxData[4 + pp] * (1 - JMM[14]) - dxData[5 + pp] * JMM[19];
00668     h[3] = 0;
00669     h[4] = dxData[2 + pp];
00670     h[5] = -dxData[1 + pp];
00671     h[0] += -(dyData[pp] * JMM[15]) - dyData[1 + pp] * JMM[16] -
00672           dyData[2 + pp] * JMM[17] - dyData[3 + pp] * JMM[18] -
00673           dyData[4 + pp] * JMM[19] + dyData[5 + pp] * (1 - JMM[20]);
00674     h[1] += 0;
00675     h[2] += dyData[pp] * JMM[6] + dyData[1 + pp] * JMM[7] +
00676           dyData[2 + pp] * JMM[8] + dyData[3 + pp] * (-1 + JMM[9]) +
00677           dyData[4 + pp] * JMM[13] + dyData[5 + pp] * JMM[18];
00678     h[3] += -dyData[2 + pp];
00679     h[4] += 0;
00680     h[5] += dyData[pp];
00681     h[0] += dzData[pp] * JMM[10] + dzData[1 + pp] * JMM[11] +
00682           dzData[2 + pp] * JMM[12] + dzData[3 + pp] * JMM[13] +
00683           dzData[4 + pp] * (-1 + JMM[14]) + dzData[5 + pp] * JMM[19];
00684     h[1] += -(dzData[pp] * JMM[6]) - dzData[1 + pp] * JMM[7] -
00685           dzData[2 + pp] * JMM[8] + dzData[3 + pp] * (1 - JMM[9]) -
00686           dzData[4 + pp] * JMM[13] - dzData[5 + pp] * JMM[18];
00687     h[2] += 0;
00688     h[3] += dzData[1 + pp];
00689     h[4] += -dzData[pp];
00690     h[5] += 0;
00691     h[0] -= h[3] * JMM[6] + h[4] * JMM[10] + h[5] * JMM[15];
00692     h[1] -= h[3] * JMM[7] + h[4] * JMM[11] + h[5] * JMM[16];
00693     h[2] -= h[3] * JMM[8] + h[4] * JMM[12] + h[5] * JMM[17];
00694     dudata[pp + 0] =
00695         h[2] * (-JMM[3] * (1 + JMM[2])) + JMM[1] * JMM[4] +
00696         h[1] * (JMM[3] * JMM[4] - JMM[1] * (1 + JMM[5])) +
00697         h[0] * (1 - JMM[4] * JMM[4] + JMM[5] + JMM[2] * (1 + JMM[5]));
00698     dudata[pp + 1] =
00699         h[2] * (JMM[3] * JMM[1] - (1 + JMM[0]) * JMM[4]) +
00700         h[1] * (1 - JMM[3] * JMM[3] + JMM[5] + JMM[0] * (1 + JMM[5])) +
00701         h[0] * (JMM[4] * JMM[3] - JMM[1] * (1 + JMM[5]));
00702     dudata[pp + 2] =
00703         h[2] * (1 - JMM[1] * JMM[1] + JMM[2] + JMM[0] * (1 + JMM[2])) +
00704         h[1] * (JMM[1] * JMM[3] - (1 + JMM[0]) * JMM[4]) +
00705         h[0] * (-((1 + JMM[2]) * JMM[3]) + JMM[1] * JMM[4]);
00706     detC =
00707         -((1 + JMM[2]) * (-1 + JMM[3] * JMM[3])) +
00708         (JMM[3] * JMM[1] - JMM[4]) * JMM[4] + JMM[5] + JMM[2] * JMM[5] +
00709         JMM[0] * (1 + JMM[2] - JMM[4] * JMM[4] + (1 + JMM[2]) * JMM[5]) -
00710         JMM[1] * (-JMM[4] * JMM[3] + JMM[1] * (1 + JMM[5]));
00711     dudata[pp + 0] /= detC;
00712     dudata[pp + 1] /= detC;
00713     dudata[pp + 2] /= detC;
00714     dudata[pp + 3] = h[3];
00715     dudata[pp + 4] = h[4];
00716     dudata[pp + 5] = h[5];
00717 }
00718 return;
00719 }

```

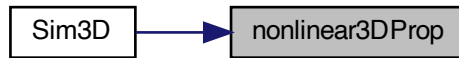
References [LatticePatch::buffData](#), [LatticePatch::derive\(\)](#), [LatticePatch::derotate\(\)](#), [LatticePatch::discreteSize\(\)](#), [errorKill\(\)](#), [LatticePatch::exchangeGhostCells\(\)](#), and [LatticePatch::rotateIntoEigen\(\)](#).

Referenced by [Sim3D\(\)](#).

Here is the call graph for this function:



Here is the caller graph for this function:



6.31 TimeEvolutionFunctions.h

[Go to the documentation of this file.](#)

```

00001 ///////////////////////////////////////////////////////////////////
00002 /// @file TimeEvolutionFunctions.h
00003 /// @brief Functions to propagate data vectors in time
00004 /// according to Maxwell's equations, and various
00005 /// orders in the HE weak-field expansion
00006 ///////////////////////////////////////////////////////////////////
00007
00008 #pragma once
00009
00010 #include "LatticePatch.h"
00011 #include "SimulationClass.h"
00012
00013 /** @brief monostate TimeEvolution class to propagate the field data in time in
00014  * a given order of the HE weak-field expansion */
00015 class TimeEvolution {
00016 public:
00017     /// choice which processes of the weak field expansion are included
00018     static int *c;
00019
00020     /// Pointer to functions for differentiation and time evolution
00021     static void (*TimeEvolver)(LatticePatch *, N_Vector, N_Vector, int *);
00022
00023     /// CVODE right hand side function (CVRhsFn) to provide IVP of the ODE
00024     static int f(sunrealtype t, N_Vector u, N_Vector udot, void *data_loc);
00025 };
00026
00027 /// Maxwell propagation function for 1D -- only for reference
00028 void linear1DProp(LatticePatch *data, N_Vector u, N_Vector udot, int *c);
00029 /// HE propagation function for 1D
00030 void nonlinear1DProp(LatticePatch *data, N_Vector u, N_Vector udot, int *c);
00031 /// Maxwell propagation function for 2D -- only for reference
00032 void linear2DProp(LatticePatch *data, N_Vector u, N_Vector udot, int *c);
00033 /// HE propagation function for 2D
00034 void nonlinear2DProp(LatticePatch *data, N_Vector u, N_Vector udot, int *c);
00035 /// Maxwell propagation function for 3D -- only for reference
00036 void linear3DProp(LatticePatch *data, N_Vector u, N_Vector udot, int *c);
00037 /// HE propagation function for 3D
00038 void nonlinear3DProp(LatticePatch *data, N_Vector u, N_Vector udot, int *c);
00039
  
```


Index

~LatticePatch
 LatticePatch, [62](#)
~Simulation
 Simulation, [119](#)

A1
 Gauss2D, [20](#)
 Gauss3D, [25](#)
A2
 Gauss2D, [20](#)
 Gauss3D, [26](#)
add
 ICSetter, [35](#)
addGauss1D
 ICSetter, [36](#)
addGauss2D
 ICSetter, [37](#)
addGauss3D
 ICSetter, [37](#)
addInitialConditions
 Simulation, [120](#)
addPeriodicICLayerInX
 Simulation, [121](#)
addPeriodicICLayerInXY
 Simulation, [122](#)
addPlaneWave1D
 ICSetter, [38](#)
addPlaneWave2D
 ICSetter, [39](#)
addPlaneWave3D
 ICSetter, [39](#)
addToSpace
 Gauss1D, [13](#)
 Gauss2D, [19](#)
 Gauss3D, [25](#)
 PlaneWave1D, [110](#)
 PlaneWave2D, [113](#)
 PlaneWave3D, [116](#)
advanceToTime
 Simulation, [122](#)
Amp
 Gauss2D, [20](#)
 Gauss3D, [26](#)
amp
 gaussian2D, [30](#)
 gaussian3D, [32](#)
axis
 Gauss2D, [20](#)
 Gauss3D, [26](#)
 gaussian2D, [30](#)

gaussian3D, [33](#)

buffData
 LatticePatch, [85](#)
BuffersInitialized
 LatticePatch.h, [211](#)
buffX
 LatticePatch, [85](#)
buffY
 LatticePatch, [85](#)
buffZ
 LatticePatch, [85](#)

c
 TimeEvolution, [138](#)
check_error
 LatticePatch.cpp, [191](#)
 LatticePatch.h, [209](#)
check_retval
 LatticePatch.cpp, [192](#)
 LatticePatch.h, [209](#)
checkFlag
 LatticePatch, [62](#)
 Simulation, [123](#)
checkNoFlag
 Simulation, [125](#)
comm
 Lattice, [53](#)
cnode_mem
 Simulation, [134](#)
CnodeObjectSetUp
 SimulationClass.h, [232](#)

dataPointDimension
 Lattice, [53](#)
DerivationStencils.h
 s10b, [142](#)
 s10c, [143](#), [144](#)
 s10f, [144](#), [145](#)
 s11b, [145](#), [146](#)
 s11f, [147](#)
 s12b, [148](#)
 s12c, [149](#), [150](#)
 s12f, [150](#), [151](#)
 s13b, [152](#)
 s13f, [153](#)
 s1b, [154](#), [155](#)
 s1f, [155](#), [156](#)
 s2b, [156](#), [157](#)
 s2c, [157](#), [158](#)

- s2f, [158](#), [159](#)
- s3b, [159](#), [160](#)
- s3f, [160](#), [161](#)
- s4b, [161](#), [162](#)
- s4c, [162](#), [163](#)
- s4f, [163](#), [164](#)
- s5b, [164](#), [165](#)
- s5f, [165](#), [166](#)
- s6b, [166](#), [167](#)
- s6c, [167](#), [168](#)
- s6f, [168](#), [169](#)
- s7b, [169](#), [170](#)
- s7f, [170](#), [171](#)
- s8b, [171](#), [172](#)
- s8c, [173](#)
- s8f, [174](#)
- s9b, [175](#)
- s9f, [176](#), [177](#)
- derive
 - LatticePatch, [64](#)
- derotate
 - LatticePatch, [69](#)
- dis
 - Gauss2D, [21](#)
 - Gauss3D, [26](#)
- discreteSize
 - LatticePatch, [71](#)
- du
 - LatticePatch, [86](#)
- duData
 - LatticePatch, [86](#)
- dx
 - Lattice, [53](#)
 - LatticePatch, [86](#)
- dy
 - Lattice, [54](#)
 - LatticePatch, [86](#)
- dz
 - Lattice, [54](#)
 - LatticePatch, [87](#)
- envelopeLattice
 - LatticePatch, [87](#)
- errorKill
 - LatticePatch.cpp, [193](#)
 - LatticePatch.h, [210](#)
- eval
 - ICSetter, [40](#)
- exchangeGhostCells
 - LatticePatch, [72](#)
- f
 - TimeEvolution, [137](#)
- FLatticeDimensionSet
 - LatticePatch.h, [211](#)
- FLatticePatchSetUp
 - LatticePatch.h, [212](#)
- Gauss1D, [11](#)
- addToSpace, [13](#)
- Gauss1D, [12](#)
- kx, [14](#)
- ky, [14](#)
- kz, [14](#)
- phig, [14](#)
- phix, [14](#)
- phiy, [15](#)
- phiz, [15](#)
- px, [15](#)
- py, [15](#)
- pz, [16](#)
- x0x, [16](#)
- x0y, [16](#)
- x0z, [16](#)
- gauss1Ds
 - ICSetter, [41](#)
- Gauss2D, [17](#)
- A1, [20](#)
- A2, [20](#)
- addToSpace, [19](#)
- Amp, [20](#)
- axis, [20](#)
- dis, [21](#)
- Gauss2D, [18](#)
- lambda, [21](#)
- Ph0, [21](#)
- PhA, [21](#)
- phip, [22](#)
- w0, [22](#)
- zr, [22](#)
- gauss2Ds
 - ICSetter, [41](#)
- Gauss3D, [23](#)
- A1, [25](#)
- A2, [26](#)
- addToSpace, [25](#)
- Amp, [26](#)
- axis, [26](#)
- dis, [26](#)
- Gauss3D, [24](#)
- lambda, [27](#)
- Ph0, [27](#)
- PhA, [27](#)
- phip, [27](#)
- w0, [28](#)
- zr, [28](#)
- gauss3Ds
 - ICSetter, [41](#)
- gaussian1D, [28](#)
- k, [29](#)
- p, [29](#)
- phi, [29](#)
- phig, [29](#)
- x0, [29](#)
- gaussian2D, [30](#)
- amp, [30](#)
- axis, [30](#)

- ph0, [31](#)
- phA, [31](#)
- phip, [31](#)
- w0, [31](#)
- x0, [31](#)
- zr, [32](#)
- gaussian3D, [32](#)
 - amp, [32](#)
 - axis, [33](#)
 - ph0, [33](#)
 - phA, [33](#)
 - phip, [33](#)
 - w0, [33](#)
 - x0, [34](#)
 - zr, [34](#)
- gCLData
 - LatticePatch, [87](#)
- gCRData
 - LatticePatch, [87](#)
- generateOutputFolder
 - OutputManager, [98](#)
- generatePatchwork
 - LatticePatch, [84](#)
 - LatticePatch.cpp, [194](#)
- generateTranslocationLookup
 - LatticePatch, [74](#)
- get_cart_comm
 - Simulation, [126](#)
- get_dataPointDimension
 - Lattice, [45](#)
- get_dx
 - Lattice, [45](#)
- get_dy
 - Lattice, [46](#)
- get_dz
 - Lattice, [46](#)
- get_ghostLayerWidth
 - Lattice, [46](#)
- get_stencilOrder
 - Lattice, [47](#)
- get_tot_lx
 - Lattice, [48](#)
- get_tot_ly
 - Lattice, [48](#)
- get_tot_lz
 - Lattice, [49](#)
- get_tot_noDP
 - Lattice, [49](#)
- get_tot_noP
 - Lattice, [50](#)
- get_tot_nx
 - Lattice, [50](#)
- get_tot_ny
 - Lattice, [50](#)
- get_tot_nz
 - Lattice, [50](#)
- getDelta
 - LatticePatch, [76](#)
- getSimCode
 - OutputManager, [99](#)
- ghostCellLeft
 - LatticePatch, [88](#)
- ghostCellLeftToSend
 - LatticePatch, [88](#)
- ghostCellRight
 - LatticePatch, [88](#)
- ghostCellRightToSend
 - LatticePatch, [88](#)
- ghostCells
 - LatticePatch, [89](#)
- ghostCellsToSend
 - LatticePatch, [89](#)
- GhostLayersInitialized
 - LatticePatch.h, [212](#)
- ghostLayerWidth
 - Lattice, [54](#)
- ICSetter, [34](#)
 - add, [35](#)
 - addGauss1D, [36](#)
 - addGauss2D, [37](#)
 - addGauss3D, [37](#)
 - addPlaneWave1D, [38](#)
 - addPlaneWave2D, [39](#)
 - addPlaneWave3D, [39](#)
 - eval, [40](#)
 - gauss1Ds, [41](#)
 - gauss2Ds, [41](#)
 - gauss3Ds, [41](#)
 - planeWaves1D, [42](#)
 - planeWaves2D, [42](#)
 - planeWaves3D, [42](#)
- icsettings
 - Simulation, [134](#)
- ID
 - LatticePatch, [89](#)
- initializeBuffers
 - LatticePatch, [77](#)
- initializeCommunicator
 - Lattice, [51](#)
- initializeCVODEobject
 - Simulation, [126](#)
- initializePatchwork
 - Simulation, [128](#)
- k
 - gaussian1D, [29](#)
 - planewave, [108](#)
- kx
 - Gauss1D, [14](#)
 - PlaneWave, [105](#)
- ky
 - Gauss1D, [14](#)
 - PlaneWave, [105](#)
- kz
 - Gauss1D, [14](#)
 - PlaneWave, [105](#)

- lambda
 - Gauss2D, 21
 - Gauss3D, 27
- Lattice, 43
 - comm, 53
 - dataPointDimension, 53
 - dx, 53
 - dy, 54
 - dz, 54
 - get_dataPointDimension, 45
 - get_dx, 45
 - get_dy, 46
 - get_dz, 46
 - get_ghostLayerWidth, 46
 - get_stencilOrder, 47
 - get_tot_lx, 48
 - get_tot_ly, 48
 - get_tot_lz, 49
 - get_tot_noDP, 49
 - get_tot_noP, 50
 - get_tot_nx, 50
 - get_tot_ny, 50
 - get_tot_nz, 50
 - ghostLayerWidth, 54
 - initializeCommunicator, 51
 - Lattice, 44
 - my_prc, 54
 - n_prc, 55
 - setDiscreteDimensions, 51
 - setPhysicalDimensions, 52
 - statusFlags, 55
 - stencilOrder, 55
 - sunctx, 55
 - tot_lx, 56
 - tot_ly, 56
 - tot_lz, 56
 - tot_noDP, 56
 - tot_noP, 57
 - tot_nx, 57
 - tot_ny, 57
 - tot_nz, 57
- lattice
 - Simulation, 135
- LatticeDiscreteSetUp
 - SimulationClass.h, 233
- LatticePatch, 58
 - ~LatticePatch, 62
 - buffData, 85
 - buffX, 85
 - buffY, 85
 - buffZ, 85
 - checkFlag, 62
 - derive, 64
 - derotate, 69
 - discreteSize, 71
 - du, 86
 - duData, 86
 - dx, 86
 - dy, 86
 - dz, 87
 - envelopeLattice, 87
 - exchangeGhostCells, 72
 - gCLData, 87
 - gCRData, 87
 - generatePatchwork, 84
 - generateTranslocationLookup, 74
 - getDelta, 76
 - ghostCellLeft, 88
 - ghostCellLeftToSend, 88
 - ghostCellRight, 88
 - ghostCellRightToSend, 88
 - ghostCells, 89
 - ghostCellsToSend, 89
 - ID, 89
 - initializeBuffers, 77
 - LatticePatch, 61
 - lgcTox, 89
 - lgcToy, 89
 - lgcToz, 90
 - Llx, 90
 - Lly, 90
 - Llz, 90
 - lx, 91
 - ly, 91
 - lz, 91
 - nx, 91
 - ny, 92
 - nz, 92
 - origin, 78
 - rgcTox, 92
 - rgcToy, 92
 - rgcToz, 93
 - rotateIntoEigen, 79
 - rotateToX, 80
 - rotateToY, 81
 - rotateToZ, 82
 - statusFlags, 93
 - u, 93
 - uAux, 93
 - uAuxData, 94
 - uData, 94
 - uTox, 94
 - uToy, 94
 - uToz, 95
 - x0, 95
 - xTou, 95
 - y0, 95
 - yTou, 96
 - z0, 96
 - zTou, 96
- latticePatch
 - Simulation, 135
- LatticePatch.cpp
 - check_error, 191
 - check_retval, 192
 - errorKill, 193

- generatePatchwork, [194](#)
- LatticePatch.h
 - BuffersInitialized, [211](#)
 - check_error, [209](#)
 - check_retval, [209](#)
 - errorKill, [210](#)
 - FLatticeDimensionSet, [211](#)
 - FLatticePatchSetUp, [212](#)
 - GhostLayersInitialized, [212](#)
 - TranslocationLookupSetUp, [212](#)
- LatticePatchworkSetUp
 - SimulationClass.h, [233](#)
- LatticePhysicalSetUp
 - SimulationClass.h, [233](#)
- lgcTox
 - LatticePatch, [89](#)
- lgcToy
 - LatticePatch, [89](#)
- lgcToz
 - LatticePatch, [90](#)
- linear1DProp
 - TimeEvolutionFunctions.cpp, [259](#)
 - TimeEvolutionFunctions.h, [287](#)
- linear2DProp
 - TimeEvolutionFunctions.cpp, [261](#)
 - TimeEvolutionFunctions.h, [289](#)
- linear3DProp
 - TimeEvolutionFunctions.cpp, [263](#)
 - TimeEvolutionFunctions.h, [291](#)
- Llx
 - LatticePatch, [90](#)
- Lly
 - LatticePatch, [90](#)
- Llz
 - LatticePatch, [90](#)
- lx
 - LatticePatch, [91](#)
- ly
 - LatticePatch, [91](#)
- lz
 - LatticePatch, [91](#)
- main
 - main.cpp, [216](#)
- main.cpp
 - main, [216](#)
- my_prc
 - Lattice, [54](#)
- n_prc
 - Lattice, [55](#)
- NLS
 - Simulation, [135](#)
- nonlinear1DProp
 - TimeEvolutionFunctions.cpp, [265](#)
 - TimeEvolutionFunctions.h, [293](#)
- nonlinear2DProp
 - TimeEvolutionFunctions.cpp, [270](#)
 - TimeEvolutionFunctions.h, [298](#)
- nonlinear3DProp
 - TimeEvolutionFunctions.cpp, [274](#)
 - TimeEvolutionFunctions.h, [302](#)
- nx
 - LatticePatch, [91](#)
- ny
 - LatticePatch, [92](#)
- nz
 - LatticePatch, [92](#)
- origin
 - LatticePatch, [78](#)
- outAllFieldData
 - Simulation, [129](#)
- OutputManager, [97](#)
 - generateOutputFolder, [98](#)
 - getSimCode, [99](#)
 - OutputManager, [97](#)
 - outputStyle, [103](#)
 - outUState, [100](#)
 - Path, [103](#)
 - set_outputStyle, [101](#)
 - simCode, [103](#)
 - SimCodeGenerator, [102](#)
- outputManager
 - Simulation, [135](#)
- outputStyle
 - OutputManager, [103](#)
- outUState
 - OutputManager, [100](#)
- p
 - gaussian1D, [29](#)
 - planewave, [108](#)
- Path
 - OutputManager, [103](#)
- Ph0
 - Gauss2D, [21](#)
 - Gauss3D, [27](#)
- ph0
 - gaussian2D, [31](#)
 - gaussian3D, [33](#)
- PhA
 - Gauss2D, [21](#)
 - Gauss3D, [27](#)
- phA
 - gaussian2D, [31](#)
 - gaussian3D, [33](#)
- phi
 - gaussian1D, [29](#)
 - planewave, [108](#)
- phig
 - Gauss1D, [14](#)
 - gaussian1D, [29](#)
- phip
 - Gauss2D, [22](#)
 - Gauss3D, [27](#)
 - gaussian2D, [31](#)
 - gaussian3D, [33](#)

- phix
 - Gauss1D, [14](#)
 - PlaneWave, [105](#)
- phiy
 - Gauss1D, [15](#)
 - PlaneWave, [106](#)
- phiz
 - Gauss1D, [15](#)
 - PlaneWave, [106](#)
- PlaneWave, [104](#)
 - kx, [105](#)
 - ky, [105](#)
 - kz, [105](#)
 - phix, [105](#)
 - phiy, [106](#)
 - phiz, [106](#)
 - px, [106](#)
 - py, [106](#)
 - pz, [107](#)
- planewave, [107](#)
 - k, [108](#)
 - p, [108](#)
 - phi, [108](#)
- PlaneWave1D, [109](#)
 - addToSpace, [110](#)
 - PlaneWave1D, [110](#)
- PlaneWave2D, [111](#)
 - addToSpace, [113](#)
 - PlaneWave2D, [112](#)
- PlaneWave3D, [114](#)
 - addToSpace, [116](#)
 - PlaneWave3D, [115](#)
- planeWaves1D
 - ICSetter, [42](#)
- planeWaves2D
 - ICSetter, [42](#)
- planeWaves3D
 - ICSetter, [42](#)
- px
 - Gauss1D, [15](#)
 - PlaneWave, [106](#)
- py
 - Gauss1D, [15](#)
 - PlaneWave, [106](#)
- pz
 - Gauss1D, [16](#)
 - PlaneWave, [107](#)
- README.md, [139](#)
- rgcTox
 - LatticePatch, [92](#)
- rgcToy
 - LatticePatch, [92](#)
- rgcToz
 - LatticePatch, [93](#)
- rotateIntoEigen
 - LatticePatch, [79](#)
- rotateToX
 - LatticePatch, [80](#)
- rotateToY
 - LatticePatch, [81](#)
- rotateToZ
 - LatticePatch, [82](#)
- s10b
 - DerivationStencils.h, [142](#)
- s10c
 - DerivationStencils.h, [143](#), [144](#)
- s10f
 - DerivationStencils.h, [144](#), [145](#)
- s11b
 - DerivationStencils.h, [145](#), [146](#)
- s11f
 - DerivationStencils.h, [147](#)
- s12b
 - DerivationStencils.h, [148](#)
- s12c
 - DerivationStencils.h, [149](#), [150](#)
- s12f
 - DerivationStencils.h, [150](#), [151](#)
- s13b
 - DerivationStencils.h, [152](#)
- s13f
 - DerivationStencils.h, [153](#)
- s1b
 - DerivationStencils.h, [154](#), [155](#)
- s1f
 - DerivationStencils.h, [155](#), [156](#)
- s2b
 - DerivationStencils.h, [156](#), [157](#)
- s2c
 - DerivationStencils.h, [157](#), [158](#)
- s2f
 - DerivationStencils.h, [158](#), [159](#)
- s3b
 - DerivationStencils.h, [159](#), [160](#)
- s3f
 - DerivationStencils.h, [160](#), [161](#)
- s4b
 - DerivationStencils.h, [161](#), [162](#)
- s4c
 - DerivationStencils.h, [162](#), [163](#)
- s4f
 - DerivationStencils.h, [163](#), [164](#)
- s5b
 - DerivationStencils.h, [164](#), [165](#)
- s5f
 - DerivationStencils.h, [165](#), [166](#)
- s6b
 - DerivationStencils.h, [166](#), [167](#)
- s6c
 - DerivationStencils.h, [167](#), [168](#)
- s6f
 - DerivationStencils.h, [168](#), [169](#)
- s7b
 - DerivationStencils.h, [169](#), [170](#)
- s7f
 - DerivationStencils.h, [170](#), [171](#)

- s8b
 - DerivationStencils.h, [171](#), [172](#)
- s8c
 - DerivationStencils.h, [173](#)
- s8f
 - DerivationStencils.h, [174](#)
- s9b
 - DerivationStencils.h, [175](#)
- s9f
 - DerivationStencils.h, [176](#), [177](#)
- set_outputStyle
 - OutputManager, [101](#)
- setDiscreteDimensions
 - Lattice, [51](#)
- setDiscreteDimensionsOfLattice
 - Simulation, [130](#)
- setInitialConditions
 - Simulation, [131](#)
- setPhysicalDimensions
 - Lattice, [52](#)
- setPhysicalDimensionsOfLattice
 - Simulation, [132](#)
- Sim1D
 - SimulationFunctions.cpp, [236](#)
 - SimulationFunctions.h, [249](#)
- Sim2D
 - SimulationFunctions.cpp, [238](#)
 - SimulationFunctions.h, [251](#)
- Sim3D
 - SimulationFunctions.cpp, [241](#)
 - SimulationFunctions.h, [254](#)
- simCode
 - OutputManager, [103](#)
- SimCodeGenerator
 - OutputManager, [102](#)
- Simulation, [116](#)
 - ~Simulation, [119](#)
 - addInitialConditions, [120](#)
 - addPeriodicICLayerInX, [121](#)
 - addPeriodicICLayerInXY, [122](#)
 - advanceToTime, [122](#)
 - checkFlag, [123](#)
 - checkNoFlag, [125](#)
 - cvmem, [134](#)
 - get_cart_comm, [126](#)
 - icsettings, [134](#)
 - initializeCVODEobject, [126](#)
 - initializePatchwork, [128](#)
 - lattice, [135](#)
 - latticePatch, [135](#)
 - NLS, [135](#)
 - outAllFieldData, [129](#)
 - outputManager, [135](#)
 - setDiscreteDimensionsOfLattice, [130](#)
 - setInitialConditions, [131](#)
 - setPhysicalDimensionsOfLattice, [132](#)
 - Simulation, [118](#)
 - start, [133](#)
 - statusFlags, [136](#)
 - t, [136](#)
- SimulationClass.h
 - CvmemObjectSetUp, [232](#)
 - LatticeDiscreteSetUp, [233](#)
 - LatticePatchworkSetUp, [233](#)
 - LatticePhysicalSetUp, [233](#)
 - SimulationStarted, [233](#)
- SimulationFunctions.cpp
 - Sim1D, [236](#)
 - Sim2D, [238](#)
 - Sim3D, [241](#)
 - timer, [243](#)
- SimulationFunctions.h
 - Sim1D, [249](#)
 - Sim2D, [251](#)
 - Sim3D, [254](#)
 - timer, [256](#)
- SimulationStarted
 - SimulationClass.h, [233](#)
- src/DerivationStencils.cpp, [139](#), [140](#)
- src/DerivationStencils.h, [140](#), [177](#)
- src/ICSetters.cpp, [181](#)
- src/ICSetters.h, [186](#), [187](#)
- src/LatticePatch.cpp, [191](#), [196](#)
- src/LatticePatch.h, [207](#), [213](#)
- src/main.cpp, [215](#), [220](#)
- src/Outputters.cpp, [223](#)
- src/Outputters.h, [225](#), [226](#)
- src/SimulationClass.cpp, [227](#)
- src/SimulationClass.h, [231](#), [234](#)
- src/SimulationFunctions.cpp, [235](#), [244](#)
- src/SimulationFunctions.h, [247](#), [257](#)
- src/TimeEvolutionFunctions.cpp, [258](#), [278](#)
- src/TimeEvolutionFunctions.h, [286](#), [306](#)
- start
 - Simulation, [133](#)
- statusFlags
 - Lattice, [55](#)
 - LatticePatch, [93](#)
 - Simulation, [136](#)
- stencilOrder
 - Lattice, [55](#)
- sunctx
 - Lattice, [55](#)
- t
 - Simulation, [136](#)
- TimeEvolution, [136](#)
 - c, [138](#)
 - f, [137](#)
 - TimeEvolver, [138](#)
- TimeEvolutionFunctions.cpp
 - linear1DProp, [259](#)
 - linear2DProp, [261](#)
 - linear3DProp, [263](#)
 - nonlinear1DProp, [265](#)
 - nonlinear2DProp, [270](#)
 - nonlinear3DProp, [274](#)

- TimeEvolutionFunctions.h
 - linear1DProp, [287](#)
 - linear2DProp, [289](#)
 - linear3DProp, [291](#)
 - nonlinear1DProp, [293](#)
 - nonlinear2DProp, [298](#)
 - nonlinear3DProp, [302](#)
- TimeEvolver
 - TimeEvolution, [138](#)
- timer
 - SimulationFunctions.cpp, [243](#)
 - SimulationFunctions.h, [256](#)
- tot_lx
 - Lattice, [56](#)
- tot_ly
 - Lattice, [56](#)
- tot_lz
 - Lattice, [56](#)
- tot_noDP
 - Lattice, [56](#)
- tot_noP
 - Lattice, [57](#)
- tot_nx
 - Lattice, [57](#)
- tot_ny
 - Lattice, [57](#)
- tot_nz
 - Lattice, [57](#)
- TranslocationLookupSetUp
 - LatticePatch.h, [212](#)
- u
 - LatticePatch, [93](#)
- uAux
 - LatticePatch, [93](#)
- uAuxData
 - LatticePatch, [94](#)
- uData
 - LatticePatch, [94](#)
- uTox
 - LatticePatch, [94](#)
- uToy
 - LatticePatch, [94](#)
- uToz
 - LatticePatch, [95](#)
- w0
 - Gauss2D, [22](#)
 - Gauss3D, [28](#)
 - gaussian2D, [31](#)
 - gaussian3D, [33](#)
- x0
 - gaussian1D, [29](#)
 - gaussian2D, [31](#)
 - gaussian3D, [34](#)
 - LatticePatch, [95](#)
- x0x
 - Gauss1D, [16](#)
- x0y
 - Gauss1D, [16](#)
- x0z
 - Gauss1D, [16](#)
- xTou
 - LatticePatch, [95](#)
- y0
 - LatticePatch, [95](#)
- yTou
 - LatticePatch, [96](#)
- z0
 - LatticePatch, [96](#)
- zr
 - Gauss2D, [22](#)
 - Gauss3D, [28](#)
 - gaussian2D, [32](#)
 - gaussian3D, [34](#)
- zTou
 - LatticePatch, [96](#)