

HEWES : Heisenberg-Euler Weak-Field Expansion Simulator

v0.2.5

Generated by Doxygen 1.9.6

1 HEWES : Heisenberg-Euler Weak-Field Expansion Simulator	1
1.1 HEWES: Heisenberg-Euler Weak-Field Expansion Simulator	1
1.1.1 Contents	2
1.1.2 Required software	2
1.1.3 Short user manual	2
1.1.3.1 Note on simulation settings	3
1.1.3.2 Note on resource occupation	4
1.1.3.3 Note on the output analysis	4
1.1.4 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	30
5.4.2.5 x0	30
5.5 gaussian2D Struct Reference	30
5.5.1 Detailed Description	30

5.5.2 Field Documentation	31
5.5.2.1 amp	31
5.5.2.2 axis	31
5.5.2.3 ph0	31
5.5.2.4 phA	31
5.5.2.5 phip	32
5.5.2.6 w0	32
5.5.2.7 x0	32
5.5.2.8 zr	32
5.6 gaussian3D Struct Reference	33
5.6.1 Detailed Description	33
5.6.2 Field Documentation	33
5.6.2.1 amp	33
5.6.2.2 axis	33
5.6.2.3 ph0	34
5.6.2.4 phA	34
5.6.2.5 phip	34
5.6.2.6 w0	34
5.6.2.7 x0	35
5.6.2.8 zr	35
5.7 ICSetter Class Reference	35
5.7.1 Detailed Description	36
5.7.2 Member Function Documentation	36
5.7.2.1 add()	36
5.7.2.2 addGauss1D()	37
5.7.2.3 addGauss2D()	38
5.7.2.4 addGauss3D()	38
5.7.2.5 addPlaneWave1D()	39
5.7.2.6 addPlaneWave2D()	40
5.7.2.7 addPlaneWave3D()	40
5.7.2.8 eval()	41
5.7.3 Field Documentation	41
5.7.3.1 gauss1Ds	42
5.7.3.2 gauss2Ds	42
5.7.3.3 gauss3Ds	42
5.7.3.4 planeWaves1D	42
5.7.3.5 planeWaves2D	43
5.7.3.6 planeWaves3D	43
5.8 Lattice Class Reference	43
5.8.1 Detailed Description	45
5.8.2 Constructor & Destructor Documentation	45
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()	48
5.8.3.9 get_tot_lz()	49
5.8.3.10 get_tot_noDP()	49
5.8.3.11 get_tot_noP()	49
5.8.3.12 get_tot_nx()	50
5.8.3.13 get_tot_ny()	50
5.8.3.14 get_tot_nz()	50
5.8.3.15 setDiscreteDimensions()	51
5.8.3.16 setPhysicalDimensions()	51
5.8.4 Field Documentation	52
5.8.4.1 comm	52
5.8.4.2 dataPointDimension	52
5.8.4.3 dx	53
5.8.4.4 dy	53
5.8.4.5 dz	53
5.8.4.6 ghostLayerWidth	53
5.8.4.7 my_prc	54
5.8.4.8 n_prc	54
5.8.4.9 statusFlags	54
5.8.4.10 stencilOrder	54
5.8.4.11 sunctx	55
5.8.4.12 tot_lx	55
5.8.4.13 tot_ly	55
5.8.4.14 tot_lz	55
5.8.4.15 tot_noDP	56
5.8.4.16 tot_noP	56
5.8.4.17 tot_nx	56
5.8.4.18 tot_ny	56
5.8.4.19 tot_nz	57
5.9 LatticePatch Class Reference	57
5.9.1 Detailed Description	60
5.9.2 Constructor & Destructor Documentation	60
5.9.2.1 LatticePatch()	61
5.9.2.2 ~LatticePatch()	61

5.9.3 Member Function Documentation	62
5.9.3.1 checkFlag()	62
5.9.3.2 derive()	63
5.9.3.3 derotate()	68
5.9.3.4 discreteSize()	70
5.9.3.5 exchangeGhostCells()	71
5.9.3.6 generateTranslocationLookup()	73
5.9.3.7 getDelta()	75
5.9.3.8 initializeBuffers()	76
5.9.3.9 origin()	77
5.9.3.10 rotateIntoEigen()	78
5.9.3.11 rotateToX()	79
5.9.3.12 rotateToY()	80
5.9.3.13 rotateToZ()	81
5.9.4 Friends And Related Function Documentation	82
5.9.4.1 generatePatchwork	82
5.9.5 Field Documentation	84
5.9.5.1 buffData	84
5.9.5.2 buffX	84
5.9.5.3 buffY	84
5.9.5.4 buffZ	85
5.9.5.5 du	85
5.9.5.6 duData	85
5.9.5.7 duLocal	85
5.9.5.8 dx	86
5.9.5.9 dy	86
5.9.5.10 dz	86
5.9.5.11 envelopeLattice	86
5.9.5.12 gCLData	87
5.9.5.13 gCRData	87
5.9.5.14 ghostCellLeft	87
5.9.5.15 ghostCellLeftToSend	87
5.9.5.16 ghostCellRight	88
5.9.5.17 ghostCellRightToSend	88
5.9.5.18 ghostCells	88
5.9.5.19 ghostCellsToSend	88
5.9.5.20 ID	88
5.9.5.21 lgcTox	89
5.9.5.22 lgcToy	89
5.9.5.23 lgcToz	89
5.9.5.24 Llx	89
5.9.5.25 Lly	90

5.9.5.26 Llz	90
5.9.5.27 lx	90
5.9.5.28 ly	90
5.9.5.29 lz	91
5.9.5.30 nx	91
5.9.5.31 ny	91
5.9.5.32 nz	91
5.9.5.33 rgcTox	92
5.9.5.34 rgcToy	92
5.9.5.35 rgcToz	92
5.9.5.36 statusFlags	92
5.9.5.37 u	93
5.9.5.38 uAux	93
5.9.5.39 uAuxData	93
5.9.5.40 uData	93
5.9.5.41 uLocal	94
5.9.5.42 uTox	94
5.9.5.43 uToy	94
5.9.5.44 uToz	94
5.9.5.45 x0	95
5.9.5.46 xTou	95
5.9.5.47 y0	95
5.9.5.48 yTou	95
5.9.5.49 z0	96
5.9.5.50 zTou	96
5.10 OutputManager Class Reference	96
5.10.1 Detailed Description	97
5.10.2 Constructor & Destructor Documentation	97
5.10.2.1 OutputManager()	97
5.10.3 Member Function Documentation	97
5.10.3.1 generateOutputFolder()	98
5.10.3.2 getSimCode()	99
5.10.3.3 outUState()	99
5.10.3.4 set_outputStyle()	101
5.10.3.5 SimCodeGenerator()	102
5.10.4 Field Documentation	102
5.10.4.1 outputStyle	102
5.10.4.2 Path	103
5.10.4.3 simCode	103
5.11 PlaneWave Class Reference	103
5.11.1 Detailed Description	104
5.11.2 Field Documentation	104

5.11.2.1 kx	104
5.11.2.2 ky	105
5.11.2.3 kz	105
5.11.2.4 phix	105
5.11.2.5 phiy	105
5.11.2.6 phiz	106
5.11.2.7 px	106
5.11.2.8 py	106
5.11.2.9 pz	106
5.12 planewave Struct Reference	107
5.12.1 Detailed Description	107
5.12.2 Field Documentation	107
5.12.2.1 k	107
5.12.2.2 p	107
5.12.2.3 phi	108
5.13 PlaneWave1D Class Reference	108
5.13.1 Detailed Description	109
5.13.2 Constructor & Destructor Documentation	109
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	113
5.14.2.1 PlaneWave2D()	113
5.14.3 Member Function Documentation	113
5.14.3.1 addToSpace()	114
5.15 PlaneWave3D Class Reference	114
5.15.1 Detailed Description	115
5.15.2 Constructor & Destructor Documentation	116
5.15.2.1 PlaneWave3D()	116
5.15.3 Member Function Documentation	116
5.15.3.1 addToSpace()	117
5.16 Simulation Class Reference	117
5.16.1 Detailed Description	119
5.16.2 Constructor & Destructor Documentation	119
5.16.2.1 Simulation()	119
5.16.2.2 ~Simulation()	120
5.16.3 Member Function Documentation	120
5.16.3.1 addInitialConditions()	120
5.16.3.2 addPeriodicCLayerInX()	122
5.16.3.3 addPeriodicCLayerInXY()	123

5.16.3.4 advanceToTime()	124
5.16.3.5 checkFlag()	125
5.16.3.6 checkNoFlag()	126
5.16.3.7 initializeCVODEobject()	127
5.16.3.8 initializePatchwork()	129
5.16.3.9 outAllFieldData()	130
5.16.3.10 setDiscreteDimensionsOfLattice()	131
5.16.3.11 setInitialConditions()	132
5.16.3.12 setPhysicalDimensionsOfLattice()	133
5.16.3.13 start()	134
5.16.4 Field Documentation	135
5.16.4.1 cvode_mem	135
5.16.4.2 icsettings	136
5.16.4.3 lattice	136
5.16.4.4 latticePatch	136
5.16.4.5 NLS	136
5.16.4.6 outputManager	137
5.16.4.7 statusFlags	137
5.16.4.8 t	137
5.17 TimeEvolution Class Reference	137
5.17.1 Detailed Description	138
5.17.2 Member Function Documentation	138
5.17.2.1 f()	138
5.17.3 Field Documentation	139
5.17.3.1 c	139
5.17.3.2 TimeEvolver	139
6 File Documentation	141
6.1 README.md File Reference	141
6.2 src/DerivationStencils.cpp File Reference	141
6.2.1 Detailed Description	141
6.3 DerivationStencils.cpp	142
6.4 src/DerivationStencils.h File Reference	142
6.4.1 Detailed Description	144
6.4.2 Function Documentation	144
6.4.2.1 s10b() [1/2]	144
6.4.2.2 s10b() [2/2]	145
6.4.2.3 s10c() [1/2]	145
6.4.2.4 s10c() [2/2]	146
6.4.2.5 s10f() [1/2]	146
6.4.2.6 s10f() [2/2]	147
6.4.2.7 s11b() [1/2]	147

6.4.2.8 <code>s11b()</code> [2/2]	148
6.4.2.9 <code>s11f()</code> [1/2]	148
6.4.2.10 <code>s11f()</code> [2/2]	149
6.4.2.11 <code>s12b()</code> [1/2]	149
6.4.2.12 <code>s12b()</code> [2/2]	150
6.4.2.13 <code>s12c()</code> [1/2]	150
6.4.2.14 <code>s12c()</code> [2/2]	151
6.4.2.15 <code>s12f()</code> [1/2]	151
6.4.2.16 <code>s12f()</code> [2/2]	152
6.4.2.17 <code>s13b()</code> [1/2]	152
6.4.2.18 <code>s13b()</code> [2/2]	153
6.4.2.19 <code>s13f()</code> [1/2]	153
6.4.2.20 <code>s13f()</code> [2/2]	154
6.4.2.21 <code>s1b()</code> [1/2]	154
6.4.2.22 <code>s1b()</code> [2/2]	155
6.4.2.23 <code>s1f()</code> [1/2]	155
6.4.2.24 <code>s1f()</code> [2/2]	156
6.4.2.25 <code>s2b()</code> [1/2]	156
6.4.2.26 <code>s2b()</code> [2/2]	157
6.4.2.27 <code>s2c()</code> [1/2]	157
6.4.2.28 <code>s2c()</code> [2/2]	158
6.4.2.29 <code>s2f()</code> [1/2]	158
6.4.2.30 <code>s2f()</code> [2/2]	159
6.4.2.31 <code>s3b()</code> [1/2]	159
6.4.2.32 <code>s3b()</code> [2/2]	160
6.4.2.33 <code>s3f()</code> [1/2]	160
6.4.2.34 <code>s3f()</code> [2/2]	161
6.4.2.35 <code>s4b()</code> [1/2]	161
6.4.2.36 <code>s4b()</code> [2/2]	162
6.4.2.37 <code>s4c()</code> [1/2]	162
6.4.2.38 <code>s4c()</code> [2/2]	163
6.4.2.39 <code>s4f()</code> [1/2]	163
6.4.2.40 <code>s4f()</code> [2/2]	164
6.4.2.41 <code>s5b()</code> [1/2]	164
6.4.2.42 <code>s5b()</code> [2/2]	165
6.4.2.43 <code>s5f()</code> [1/2]	165
6.4.2.44 <code>s5f()</code> [2/2]	166
6.4.2.45 <code>s6b()</code> [1/2]	166
6.4.2.46 <code>s6b()</code> [2/2]	167
6.4.2.47 <code>s6c()</code> [1/2]	167
6.4.2.48 <code>s6c()</code> [2/2]	168
6.4.2.49 <code>s6f()</code> [1/2]	168

6.4.2.50 <code>s6f()</code> [2/2]	169
6.4.2.51 <code>s7b()</code> [1/2]	169
6.4.2.52 <code>s7b()</code> [2/2]	170
6.4.2.53 <code>s7f()</code> [1/2]	170
6.4.2.54 <code>s7f()</code> [2/2]	171
6.4.2.55 <code>s8b()</code> [1/2]	171
6.4.2.56 <code>s8b()</code> [2/2]	172
6.4.2.57 <code>s8c()</code> [1/2]	172
6.4.2.58 <code>s8c()</code> [2/2]	173
6.4.2.59 <code>s8f()</code> [1/2]	173
6.4.2.60 <code>s8f()</code> [2/2]	174
6.4.2.61 <code>s9b()</code> [1/2]	174
6.4.2.62 <code>s9b()</code> [2/2]	175
6.4.2.63 <code>s9f()</code> [1/2]	175
6.4.2.64 <code>s9f()</code> [2/2]	176
6.5 <code>DerivationStencils.h</code>	176
6.6 <code>src/ICSetters.cpp</code> File Reference	180
6.6.1 Detailed Description	180
6.7 <code>ICSetters.cpp</code>	180
6.8 <code>src/ICSetters.h</code> File Reference	185
6.8.1 Detailed Description	186
6.9 <code>ICSetters.h</code>	186
6.10 <code>src/LatticePatch.cpp</code> File Reference	189
6.10.1 Detailed Description	190
6.10.2 Function Documentation	190
6.10.2.1 <code>check_retval()</code>	190
6.10.2.2 <code>errorKill()</code>	191
6.10.2.3 <code>generatePatchwork()</code>	192
6.11 <code>LatticePatch.cpp</code>	194
6.12 <code>src/LatticePatch.h</code> File Reference	206
6.12.1 Detailed Description	207
6.12.2 Function Documentation	208
6.12.2.1 <code>check_retval()</code>	208
6.12.2.2 <code>errorKill()</code>	209
6.12.3 Variable Documentation	210
6.12.3.1 <code>BuffersInitialized</code>	210
6.12.3.2 <code>FLatticeDimensionSet</code>	210
6.12.3.3 <code>FLatticePatchSetUp</code>	210
6.12.3.4 <code>GhostLayersInitialized</code>	210
6.12.3.5 <code>TranslocationLookupSetUp</code>	211
6.13 <code>LatticePatch.h</code>	211
6.14 <code>src/main.cpp</code> File Reference	214

6.14.1 Detailed Description	214
6.14.2 Function Documentation	215
6.14.2.1 main()	215
6.15 main.cpp	222
6.16 src/Outputters.cpp File Reference	226
6.16.1 Detailed Description	226
6.17 Outputters.cpp	226
6.18 src/Outputters.h File Reference	228
6.18.1 Detailed Description	229
6.19 Outputters.h	229
6.20 src/SimulationClass.cpp File Reference	230
6.20.1 Detailed Description	230
6.21 SimulationClass.cpp	230
6.22 src/SimulationClass.h File Reference	234
6.22.1 Detailed Description	235
6.22.2 Variable Documentation	235
6.22.2.1 CvoidObjectSetUp	236
6.22.2.2 LatticeDiscreteSetUp	236
6.22.2.3 LatticePatchworkSetUp	236
6.22.2.4 LatticePhysicalSetUp	236
6.22.2.5 SimulationStarted	237
6.23 SimulationClass.h	237
6.24 src/SimulationFunctions.cpp File Reference	238
6.24.1 Detailed Description	239
6.24.2 Function Documentation	239
6.24.2.1 Sim1D()	239
6.24.2.2 Sim2D()	242
6.24.2.3 Sim3D()	245
6.24.2.4 timer()	248
6.25 SimulationFunctions.cpp	249
6.26 src/SimulationFunctions.h File Reference	252
6.26.1 Detailed Description	254
6.26.2 Function Documentation	254
6.26.2.1 Sim1D()	254
6.26.2.2 Sim2D()	257
6.26.2.3 Sim3D()	260
6.26.2.4 timer()	263
6.27 SimulationFunctions.h	264
6.28 src/TimeEvolutionFunctions.cpp File Reference	265
6.28.1 Detailed Description	265
6.28.2 Function Documentation	265
6.28.2.1 linear1DProp()	266

6.28.2.2 linear2DProp()	267
6.28.2.3 linear3DProp()	269
6.28.2.4 nonlinear1DProp()	271
6.28.2.5 nonlinear2DProp()	276
6.28.2.6 nonlinear3DProp()	280
6.29 TimeEvolutionFunctions.cpp	284
6.30 src/TimeEvolutionFunctions.h File Reference	293
6.30.1 Detailed Description	294
6.30.2 Function Documentation	294
6.30.2.1 linear1DProp()	294
6.30.2.2 linear2DProp()	296
6.30.2.3 linear3DProp()	298
6.30.2.4 nonlinear1DProp()	300
6.30.2.5 nonlinear2DProp()	305
6.30.2.6 nonlinear3DProp()	309
6.31 TimeEvolutionFunctions.h	313
Index	315

Chapter 1

HEWES : Heisenberg-Euler Weak-Field Expansion Simulator

1.1 HEWES: Heisenberg-Euler Weak-Field Expansion Simulator

The Heisenberg-Euler Weak-Field Expansion Simulator is a solver for the all-optical QED vacuum. Vacuum polarization, due to omnipresent quantum fluctuations, supplements Maxwell's linear equations of electromagnetism by nonlinear photon-photon interactions. HEWES solves the nonlinear equations of motion for electromagnetic waves in the weak-field limit of the Heisenberg-Euler effective theory of QED with up to six-photon processes.

There is a [paper](#) that introduces the algorithm and shows remarkable results and a [Mendeley Data repository](#) with extra and supplementary materials.

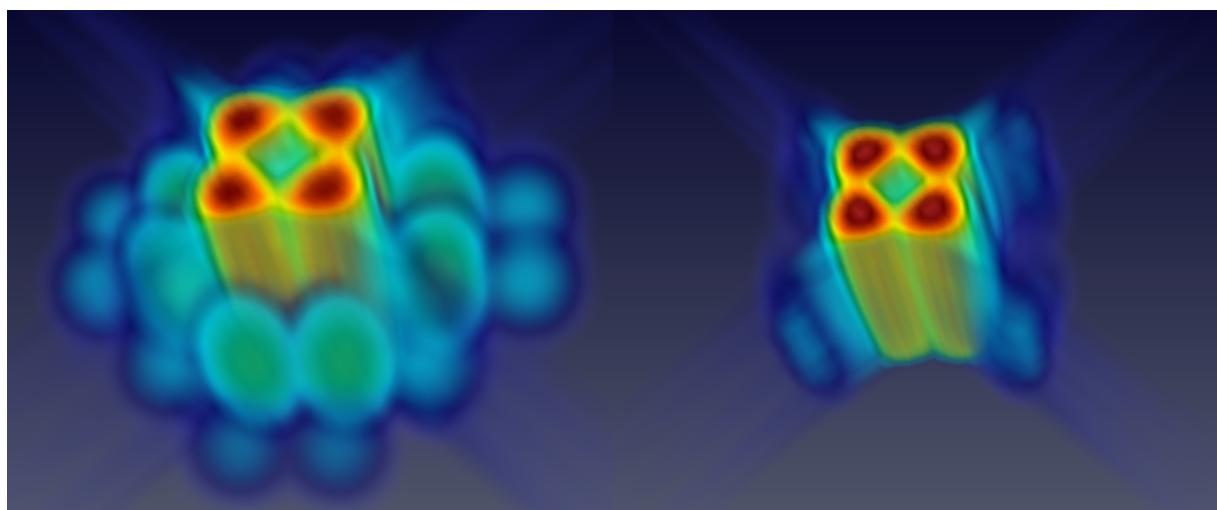


Figure 1.1 3D Harmonic Generation

1.1.1 Contents

- Required software
- Short user manual
 - Note on simulation settings
 - Note on resource occupation
 - Note on the output analysis
- Authors

1.1.2 Required software

CMake is used for building and a recent *C++* compiler version is required since features up to the *C++20* standard are used.

An *MPI* implementation supporting the *MPI* 3.1 standard is strongly recommended to make use of multiprocessing.

OpenMP is optional to enforce more vectorization and enable multithreading. The latter extra layer of parallelization is useful for performance only when a very large number of compute nodes is occupied.

The *CVODE* solver is fetched on-the-fly through *CMake*.

If *CVODE* (or the whole *SUNDIALS* package) is installed manually: Version 6 is required, the code is presumably compliant with the upcoming version 7. Enable *MPI* and *OpenMP*, if desired. For optimal performance the `CMAKE_BUILD_TYPE` should be "Release". Edit the `SUNDIALS_DIR` variable in the `CMakeLists.txt` to the installation directory.

1.1.3 Short user manual

In order to build the executable with *CMake*, execute, e.g., in the src directory, `cmake -S . -Bbuild` and then `cmake --build build`.

On *Windows* a subsequent installation of the *SUNDIALS* modules is required. With *CMake* this can be done via `cmake --install build --config Debug`. The installation type has to be "Debug" with *MSVC* due to some issue with *SUNDIALS*. *MSMPI* is required, even though it only complies with *MPI*-2.0.

You have full control over all high-level simulation settings via command line arguments.

- First, the general settings are specified:
 - The path to the project directory. Therein, a `SimResults` folder is created and therein a folder named after the timestamp of the start of the simulation.
 - Whether you want to simulate in 1D, 2D, or 3D.
 - The relative and absolute integration tolerances of the *CVODE* solver.
Recommended values are between 1e-12 and 1e-16.
 - the order of accuracy of the numerical scheme (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 (relevant 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 slices in any dimension.
Note: In the 3D case patches should be chosen cubic in terms of lattice points. This is decisive for computational efficiency.
- Whether you want to simulate in the linear vacuum (0), on top of the linear vacuum only 4-photon processes (1), only 6-photon processes (2), or 4- and 6-photon processes (3).
- 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 field data to be written to disk.
- The output format. It can be *CSV* (comma-separated-values) or 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 are written into one file and the step number is the name of the file.
- Second, the electromagnetic waves are chosen and their parameters specified. You can choose plane waves (not much physical content, but useful for checks) and implementations of Gaussian pulses in 1D, 2D, and 3D. To see which command line argument is which parameter, see the comments in the short example *Bash* run scripts which are preconfigured for `1D`, `2D`, and `3D` simulations. (One example is also provided as a *Windows Powershell script*.) Amplitudes are given in units of the critical field strength (Schwinger limit). Position and propagation parameters on the y- and z-axis are only effective if the grid has an extent in the corresponding dimension.
A description of the wave implementations is given in the *Doxxygen*-generated [code reference](#). Note that the 3D Gaussians, as they are implemented up to now, are propagated only in the xy-plane. More waveform implementations will follow in subsequent versions of the code.

The boundaries are periodic.

Note that in 2D and 3D simulations the number of *MPI* processes has to coincide with the actual number of patches, as described above.

If the program was built with *OpenMP* support, the environment variable `OMP_NUM_THREADS` needs to be set.

It can be useful to save the run script along with the output as a log of the simulation settings for later reference.

Monitor `stdout` and `stderr` during the run (or redirect into files). The starting timestamp, 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 cannot be guaranteed and issues of this kind can hardly be predicted. On top, they are even system-dependent. Piece of advice: Only pass decimals for the grid settings and initial conditions.

CVODE warnings and errors are reported on `stdout` and `stderr`.

1.1.3.1 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 fourth of the Nyquist frequency to minimize nonphysical dispersion effects on the lattice. 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 in the example scripts.

1.1.3.2 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. There, the communication load plays a major role which in turn depends on the order of the numerical scheme. This is because the the order of the scheme determines how many neighboring grid points are taken into account for the finite differences derivatives.

Simulations in 1D are relatively cheap and can easily be run on a modern notebook within some seconds. 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 and still take only some minutes. 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. This means an increased communication load. Even hundreds or thousands of cores can be kept busy for many hours or days. The output size quickly amounts to hundreds of gigabytes for just a single state. This hurdle forms a practical limit to the grid resolution.

Some scaling tests are shown in the [paper](#).

If the output is in binary form, the size can be easily calculated. Per step it is the number of grid points times six (the number of field components) times 8 bytes.

1.1.3.3 Note on the 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 the simplest solution for smaller simulations and a portable way that also works fast and is straightforward to analyze.

Or, the option strictly recommended for 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 the help of *MPI IO*, and hence only possible if *MPI* is used. 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 therein a folder named after the starting timestamp of the simulation (in the form `yy-mm-dd_hh-MM-ss`) is created. This is 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. For its use, the byte order of the reading machine has to be the same as that of the writing machine.

More information describing settings and analysis procedures used for actual scientific results are given in an open-access [paper](#) and a collection of corresponding analysis notebooks are uploaded to a *Mendeley Data repository*. Some small example *Python* analysis scripts can be found in the examples. The `first steps` demonstrate how the simulated data is correctly 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. A script with the purpose to extract the ratio of polarization flipped photons of a laser pulse due to vacuum birefringence can be found in the examples as `birefringence.py`. Visualization requires tools like *Paraview*, as used for the cover figures.

1.1.4 Authors

- Arnau Pons Domenech
- Hartmut Ruhl (hartmut.ruhl@physik.uni-muenchen.de)
- Andreas Lindner (and.lindner@physik.uni-muenchen.de)
- Baris Ölmez (b.oelez@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	33
ICSetter	35
Lattice	43
LatticePatch	57
OutputManager	96
PlaneWave	103
PlaneWave1D	108
PlaneWave2D	111
PlaneWave3D	114
planewave	107
Simulation	117
TimeEvolution	137

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	33
ICSetter	ICSetter class to initialize wave types with default parameters	35
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	57
OutputManager	Output Manager class to generate and coordinate output writing to disk	96
PlaneWave	Super-class for plane waves	103
planewave	Plane wave structure	107
PlaneWave1D	Class for plane waves in 1D	108
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	117
TimeEvolution	Monostate TimeEvolution class to propagate the field data in time in a given order of the HE weak-field expansion	137

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	141
src/DerivationStencils.h	
Definition of derivation stencils from order 1 to 13	142
src/ICSetters.cpp	
Implementation of the plane wave and Gaussian wave packets	180
src/ICSetters.h	
Declaration of the plane wave and Gaussian wave packets	185
src/LatticePatch.cpp	
Construction of the overall envelope lattice and the lattice patches	189
src/LatticePatch.h	
Declaration of the lattice and lattice patches	206
src/main.cpp	
Main function to configure the user's simulation settings	214
src/Outputters.cpp	
Generation of output writing to disk	226
src/Outputters.h	
OutputManager class to outstream simulation data	228
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)	230
src/SimulationClass.h	
Class for the Simulation object calling all functionality: from wave settings over lattice construction, time evolution and outputs initialization of the CVode object	234
src/SimulationFunctions.cpp	
Implementation of the complete simulation functions for 1D, 2D, and 3D, as called in the main function	238
src/SimulationFunctions.h	
Full simulation functions for 1D, 2D, and 3D used in main.cpp	252
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	265
src/TimeEvolutionFunctions.h	
Functions to propagate data vectors in time according to Maxwell's equations, and various orders in the HE weak-field expansion	293

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

Private Attributes

- surrealtype `kx`
wavenumber k_x
- surrealtype `ky`
wavenumber k_y
- surrealtype `kz`
wavenumber k_z
- surrealtype `px`
polarization & amplitude in x-direction, p_x
- surrealtype `py`
polarization & amplitude in y-direction, p_y
- surrealtype `pz`
polarization & amplitude in z-direction, p_z
- surrealtype `phix`
phase shift in x-direction, ϕ_x
- surrealtype `phiy`
phase shift in y-direction, ϕ_y
- surrealtype `phiz`

- sunrealtype `x0z`
center of pulse 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 \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{                                /* E-field vector */
00157     px * cos(kScalarX - phix) * envelopeAmp, /* \f$ E_x \f$ */
00158     py * cos(kScalarX - phiy) * envelopeAmp, /* \f$ E_y \f$ */
00159     pz * cos(kScalarX - phiz) * envelopeAmp}; /* \f$ E_z \f$ */
00160   // Put E-field into space
00161   pTo6Space[0] += E[0];
00162   pTo6Space[1] += E[1];
00163   pTo6Space[2] += E[2];
00164   // and B-field
00165   pTo6Space[3] += (ky * E[2] - kz * E[1]) / wavelength;
00166   pTo6Space[4] += (kz * E[0] - kx * E[2]) / wavelength;
00167   pTo6Space[5] += (kx * E[1] - ky * E[0]) / wavelength;
00168
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 kx

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

wavenumber k_x

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

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

5.1.4.2 ky

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

wavenumber k_y

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

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

5.1.4.3 kz

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

wavenumber k_z

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

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

5.1.4.4 phig

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

Private Attributes

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

Definition at line 172 of file [ICSetters.cpp](#).

```
00175
00176     dis = dis_;           /** - center it approaches */
00177     axis = axis_;         /** - direction from 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_;          /** - shift from 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 }
```

References [A1](#), [A2](#), [Amp](#), [axis](#), [dis](#), [lambda](#), [Ph0](#), [PhA](#), [phip](#), [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 /*+z*z*/)
00204         - zg * zg); // radial distance to propagation axis
00205     // \f$ w(z) = w_0\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) = \arctan(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$-
00211     sunrealtype Rz = nan("0x12345"); // beam curvature
00212     if (abs(gz) > 1e-15)
00213         Rz = zg * (1 + (zr * zr / zg / zg));
00214     else
00215         Rz = 1e308;
00216     // wavenumber \f$ k = 2\pi/\lambda \f$-
00217     const sunrealtype k = 2 * std::numbers::pi / lambda;
00218     // \f$ \Phi_F = kr^2/(2*R(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-\Phi_0)^2/PhA^2}\cos(\Phi_F) \f$-
00222     // CVode is a diva, no chance to remove the square in the second exponential
00223     // -> h too small
00224     const sunrealtype G2D = sqrt(w0 / wz) * exp(-r * r / wz / wz) *
00225         exp(-(zg - Ph0) * (zg - Ph0) / PhA / PhA) *
00226         cos(PhF); // gauss shape
00227     // \f$ c_\alpha = \vec{e}_x\cdot\vec{e}_z \f$-
00228     // projection components; do like this for CVode 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 [ICSetters.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 [ICSetters.h](#).

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

5.2.4.3 Amp

`sunrealtype Gauss2D::Amp [private]`

amplitude A

Definition at line [146](#) of file [ICSetters.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 (shift) 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}, unrealtype Amp_=1.0, unrealtype phip_=0, unrealtype w0_=1e-5, unrealtype zr_=4e-5, unrealtype Ph0_=2e-5, unrealtype PhA_=0.45e-5)`
construction with default parameters
- void `addToSpace` (`unrealtype x, unrealtype y, unrealtype z, unrealtype *pTo6Space`) const
function for the actual implementation in space

Private Attributes

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

5.3.1 Detailed Description

class for Gaussian pulses in 3D

They are given in the form $\vec{E} = A \vec{\epsilon} \frac{\omega_0}{\omega(z)} \exp(-r/\omega(z))^2 \exp\left(-((z_g - \Phi_0)/\Phi_A)^2\right) \cos\left(\frac{k r^2}{2R(z)} + g(z) - k z_g\right)$ 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 from TE-mode
- taille
- Rayleigh length
- shift from 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 from TE-mode */
00258     w0 = w0_;      /** - taille */
00259     zr = zr_;       /** - Rayleigh length */
00260     Ph0 = Ph0_;    /** - shift from center */
00261     PhA = PhA_;   /** - beam length */
00262     lambda = std::numbers::pi * w0 * w0 / zr;
00263     A1 = Amp * cos(phip);
00264     A2 = Amp * sin(phip);
00265 }
```

References [A1](#), [A2](#), [Amp](#), [axis](#), [dis](#), [lambda](#), [Ph0](#), [PhA](#), [phip](#), [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 (shift) 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::phip [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](#).

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

5.4.2.2 p

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

amplitude & polarization vector

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

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

5.4.2.3 phi

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

phase shift

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

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

5.4.2.4 phig

`sunrealtype gaussian1D::phig`

`width`

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

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

5.4.2.5 x0

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

`shift from origin`

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

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

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](#).

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

5.5.2.2 axis

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

direction from where it comes

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

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

5.5.2.3 ph0

`sunrealtype gaussian2D::ph0`

beam center

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

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

5.5.2.4 phA

`sunrealtype gaussian2D::phA`

beam length

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

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

5.5.2.5 **phip**

sunrealtype gaussian2D::phip

polarization rotation

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

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

5.5.2.6 **w0**

sunrealtype gaussian2D::w0

taille

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

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

5.5.2.7 **x0**

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

center

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

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

5.5.2.8 **zr**

sunrealtype gaussian2D::zr

Rayleigh length

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

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

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 `phi0`
- sunrealtype `phiA`

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](#).

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

5.6.2.2 axis

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

direction from where it comes

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

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

5.6.2.3 ph0

sunrealtype gaussian3D::ph0

beam center

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

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

5.6.2.4 phA

sunrealtype gaussian3D::phA

beam length

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

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

5.6.2.5 phip

sunrealtype gaussian3D::phip

polarization rotation

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

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

5.6.2.6 w0

sunrealtype gaussian3D::w0

taille

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

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

5.6.2.7 x0

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

center

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

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

5.6.2.8 zr

```
sunrealtype gaussian3D::zr
```

Rayleigh length

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

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

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

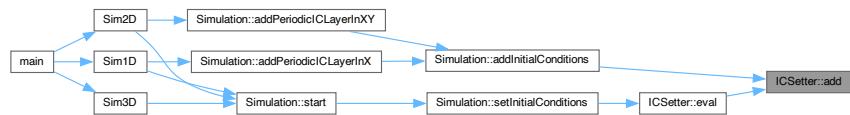
Definition at line 309 of file [ICSetters.cpp](#).

```
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 }
```

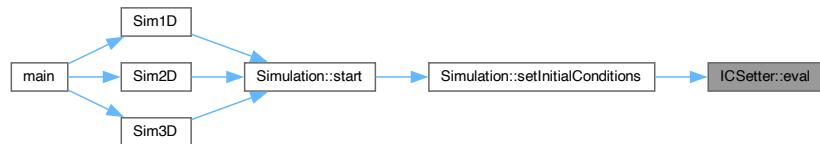
References [add\(\)](#).

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

- [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

- void * `comm`
- SUNContext `sunctx`
SUNContext object.

Static Public Attributes

- static constexpr int `n_prc` = 1
number of processes
- static constexpr int `my_prc` = 0
process number

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 59 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 34 of file LatticePatch.cpp.
00034             : stencilOrder(Sto),
00035     ghostLayerWidth(Sto/2+1) {
00036     statusFlags = 0;
00037 }
```

References [statusFlags](#).

5.8.3 Member Function Documentation

5.8.3.1 get_dataPointDimension()

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

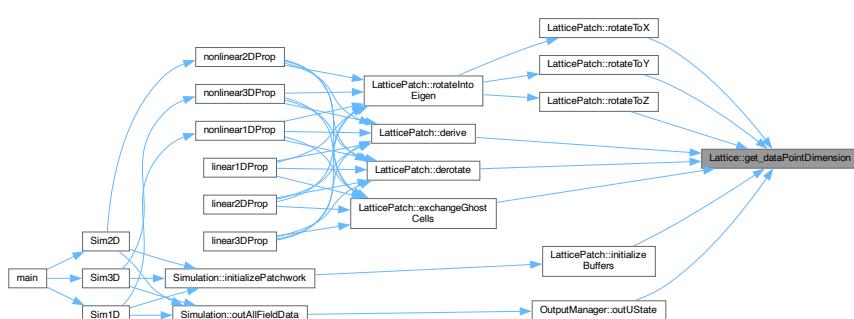
Definition at line 134 of file [LatticePatch.h](#).

```
00134     return dataPointDimension;
00135 }
00136 }
```

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 131 of file [LatticePatch.h](#).

```
00131 { return dx; }
```

References [dx](#).

5.8.3.3 get_dy()

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

getter function

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

```
00132 { return dy; }
```

References [dy](#).

5.8.3.4 get_dz()

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

getter function

Definition at line 133 of file [LatticePatch.h](#).

```
00133 { return dz; }
```

References [dz](#).

5.8.3.5 get_ghostLayerWidth()

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

getter function

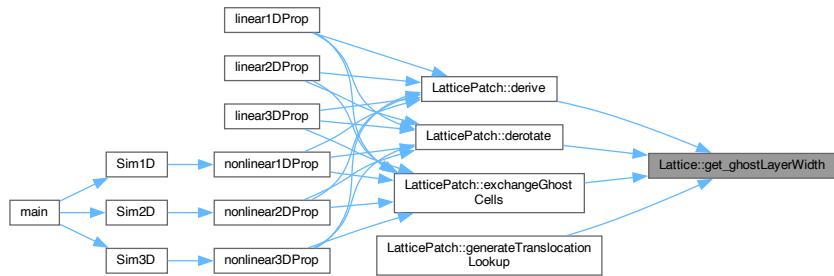
Definition at line 138 of file [LatticePatch.h](#).

```
00138
00139     return ghostLayerWidth;
00140 }
```

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

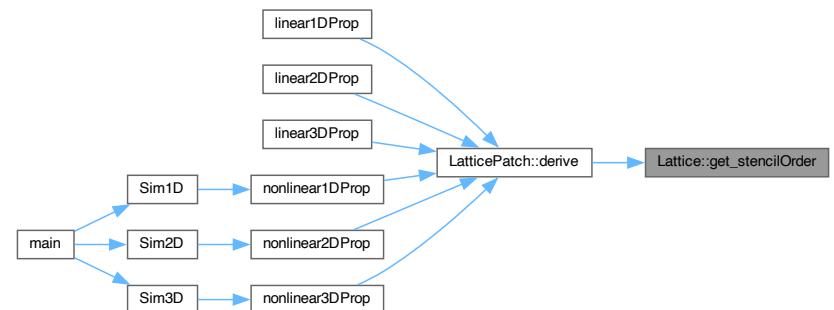
Definition at line 137 of file [LatticePatch.h](#).

```
00137 { 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 123 of file [LatticePatch.h](#).

```
00123 { 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 124 of file [LatticePatch.h](#).

```
00124 { 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 125 of file [LatticePatch.h](#).
00125 { **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 130 of file [LatticePatch.h](#).
00130 { **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 129 of file [LatticePatch.h](#).
00129 { **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 126 of file [LatticePatch.h](#).

```
00126 { 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 127 of file [LatticePatch.h](#).

```
00127 { 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 128 of file [LatticePatch.h](#).

```
00128 { return tot_nz; }
```

References [tot_nz](#).

5.8.3.15 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 40 of file [LatticePatch.cpp](#).

```
00041 // copy the given data for number of points
00042 tot_nx = _nx;
00043 tot_ny = _ny;
00044 tot_nz = _nz;
00045 // compute the resulting number of points and datapoints
00046 tot_noP = tot_nx * tot_ny * tot_nz;
00047 tot_noDP = dataPointDimension * tot_noP;
00048 // compute the new Delta, the physical resolution
00049 dx = tot_lx / tot_nx;
00050 dy = tot_ly / tot_ny;
00051 dz = tot_lz / tot_nz;
00052
00053 }
```

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:



5.8.3.16 setPhysicalDimensions()

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

component function for resizing the physical size of the lattice

Set the physical size of the lattice.

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

```
00057
00058 tot_lx = _lx;
00059 tot_ly = _ly;
00060 tot_lz = _lz;
00061 // calculate physical distance between points
```

```

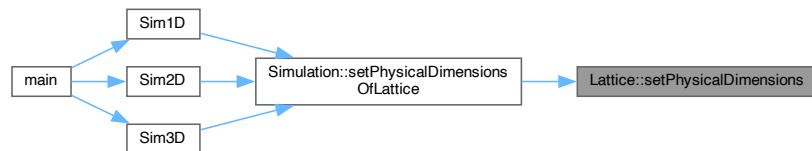
00062     dx = tot_lx / tot_nx;
00063     dy = tot_ly / tot_ny;
00064     dz = tot_lz / tot_nz;
00065     statusFlags |= FLatticeDimensionSet;
00066 }

```

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

```
void* Lattice::comm
```

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

Referenced by [LatticePatch::exchangeGhostCells\(\)](#), [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 76 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 80 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 82 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 84 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 88 of file [LatticePatch.h](#).

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

5.8.4.7 my_prc

```
constexpr int Lattice::my_prc = 0 [static], [constexpr]
```

process number

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

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

5.8.4.8 n_prc

```
constexpr int Lattice::n_prc = 1 [static], [constexpr]
```

number of processes

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

5.8.4.9 statusFlags

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

lattice status flags

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

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

5.8.4.10 stencilOrder

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

stencil order

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

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

5.8.4.11 sunctx

```
SUNContext Lattice::sunctx
```

SUNContext object.

Definition at line 114 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 62 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 64 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 66 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 78 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 74 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 68 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 70 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 72 of file [LatticePatch.h](#).

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

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

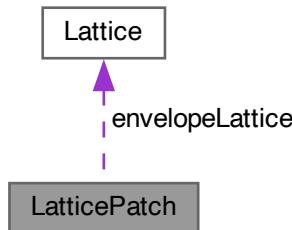
- src/[LatticePatch.h](#)
- src/[LatticePatch.cpp](#)

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; if not, abort

Data Fields

- int **ID**
 - ID of the LatticePatch, corresponds to process number (for debugging)*
- N_Vector **uLocal**
 - NVector for saving field components u=(E,B) in lattice points.*
- N_Vector **u**
- N_Vector **duLocal**
 - NVector for saving temporal derivatives of the field data.*
- N_Vector **du**
- sunrealtype * **uData**
 - pointer to field data*
- sunrealtype * **duData**
 - pointer to time-derivative data*
- sunrealtype * **uAuxData**
 - pointer to auxiliary data vector*
- 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 (auxilliarily) 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 146 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 73 of file [LatticePatch.cpp](#).

```
00073     {
00074     // set default origin coordinates to (0,0,0)
00075     x0 = y0 = z0 = 0;
00076     // set default position in Lattice-Patchwork to (0,0,0)
00077     Llx = Lly = Llz = 0;
00078     // set default physical length for lattice patch to (0,0,0)
00079     lx = ly = lz = 0;
00080     // set default discrete length for lattice patch to (0,1,1)
00081     /* This is done in this manner as even in 1D simulations require a 1 point
00082      * width */
00083     nx = 0;
00084     ny = nz = 1;
00085
00086     // u is not initialized as it wouldn't make any sense before the dimensions
00087     // are set idem for the enveloping lattice
00088
00089     // set default statusFlags to non set
00090     statusFlags = 0;
00091 }
```

References [Llx](#), [Lly](#), [Llz](#), [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 94 of file [LatticePatch.cpp](#).

```
00094     {
00095     // Deallocate memory for solution vector
00096     if (statusFlags & FLatticePatchSetUp) {
00097         // Destroy data vectors
00098 #if defined(_MPI)
00099 #if defined(_OPENMP)
00100     N_VDestroy(u);
00101     N_VDestroy(du);
00102     N_VDestroy_OpenMP(uLocal);
00103     N_VDestroy_OpenMP(duLocal);
00104 #else
00105     N_VDestroy_Parallel(u);
00106     N_VDestroy_Parallel(du);
00107 #endif
00108 #elif defined(_OPENMP)
00109     N_VDestroy_OpenMP(u);
00110     N_VDestroy_OpenMP(du);
00111 #else
00112     N_VDestroy_Serial(u);
00113     N_VDestroy_Serial(du);
00114 #endif
00115 }
00116 }
```

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

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; if not, abort

Check if all flags are set.

Definition at line 670 of file [LatticePatch.cpp](#).

```
00670
00671     if (!(statusFlags & flag)) {
00672         std::string errorMessage;
00673         switch (flag) {
00674             case FLatticePatchSetUp:
00675                 errorMessage = "The Lattice patch was not set up please make sure to "
00676                             "initialize a Lattice topology";
00677                 break;
00678             case TranslocationLookupSetUp:
00679                 errorMessage = "The translocation lookup tables have not been generated, "
00680                             "please be sure to run generateTranslocationLookup()";
00681                 break;
00682             case GhostLayersInitialized:
00683                 errorMessage = "The space for the ghost layers has not been allocated, "
00684                             "please be sure that the ghost cells are initialized ";
00685                 break;
00686             case BuffersInitialized:
00687                 errorMessage = "The space for the buffers has not been allocated, please "
00688                             "be sure to run initializeBuffers()";
00689                 break;
00690             default:
00691                 errorMessage = "Uppss, you've made a non-standard error, sadly I can't "
00692                             "help you there";
00693                 break;
00694         }
00695         errorKill(errorMessage);
00696     }
00697     return;
00698 }
```

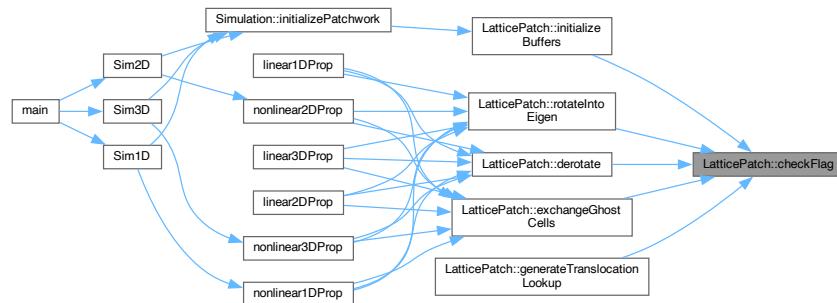
References [BuffersInitialized](#), [errorKill\(\)](#), [FLatticePatchSetUp](#), [GhostLayersInitialized](#), [statusFlags](#), and [TranslocationLookupSetUp](#).

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 701 of file [LatticePatch.cpp](#).

```
00701     {
00702     // ghost layer width adjusted to the chosen stencil order
00703     const int gLW = envelopeLattice->get_ghostLayerWidth();
00704     // dimensionality of data points -> 6
00705     const int dPD = envelopeLattice->get_dataPointDimension();
00706     // total width of patch in given direction including ghost layers at ends
00707     const sunindextype dirWidth = discreteSize(dir) + 2 * gLW;
00708     // width of patch only in given direction
00709     const sunindextype dirWidth0 = discreteSize(dir);
00710     // size of plane perpendicular to given dimension
00711     const sunindextype perpPlainSize = discreteSize() / discreteSize(dir);
00712     // physical distance between points in that direction
00713     sunrealtype dxi = nan("0x12345");
00714     switch (dir) {
00715     case 1:
00716         dxi = dx;
00717         break;
00718     case 2:
00719         dxi = dy;
00720         break;
00721     case 3:
00722         dxi = dz;
00723         break;
00724     default:
00725         dxi = 1;
00726         errorKill("Tried to derive in the wrong direction");
00727         break;
00728     }
00729     // Derive according to chosen stencil accuracy order
00730     const int order = envelopeLattice->get_stencilOrder();
00731     switch (order) {
00732     case 1: // gLW=1
00733         #pragma omp parallel for default(none) \
00734             shared(perpPlainSize, dxi, dirWidth, dirWidth0, gLW, dPD, uAux)
00735         for (sunindextype i = 0; i < perpPlainSize; i++) {
00736             #pragma omp simd
00737             for (sunindextype j = (i * dirWidth + gLW) * dPD;
00738                 j < (i * dirWidth + gLW + dirWidth0) * dPD; j += dPD) {
00739                 uAux[j + 0 - gLW * dPD] = slb(&uAux[j + 0]) / dxi;
00740                 uAux[j + 1 - gLW * dPD] = slb(&uAux[j + 1]) / dxi;
00741                 uAux[j + 2 - gLW * dPD] = slf(&uAux[j + 2]) / dxi;
00742             }
00743         }
00744     }
```

```

00742     uAux[j + 3 - gLW * dPD] = s1f(&uAux[j + 3]) / dxii;
00743     uAux[j + 4 - gLW * dPD] = s1f(&uAux[j + 4]) / dxii;
00744     uAux[j + 5 - gLW * dPD] = s1f(&uAux[j + 5]) / dxii;
00745 }
00746 }
00747 break;
00748 case 2: // gLW=2
00749 #pragma omp parallel for default(none) \
00750 shared(perpPlainSize, dxii, dirWidth, dirWidthO, gLW, dPD, uAux)
00751 for (sunindextype i = 0; i < perpPlainSize; i++) {
00752 #pragma omp simd
00753 for (sunindextype j = (i * dirWidth + gLW) * dPD;
00754 j < (i * dirWidth + gLW + dirWidthO) * dPD; j += dPD) {
00755     uAux[j + 0 - gLW * dPD] = s2b(&uAux[j + 0]) / dxii;
00756     uAux[j + 1 - gLW * dPD] = s2b(&uAux[j + 1]) / dxii;
00757     uAux[j + 2 - gLW * dPD] = s2f(&uAux[j + 2]) / dxii;
00758     uAux[j + 3 - gLW * dPD] = s2f(&uAux[j + 3]) / dxii;
00759     uAux[j + 4 - gLW * dPD] = s2c(&uAux[j + 4]) / dxii;
00760     uAux[j + 5 - gLW * dPD] = s2c(&uAux[j + 5]) / dxii;
00761 }
00762 }
00763 break;
00764 case 3: // gLW=2
00765 #pragma omp parallel for default(none) \
00766 shared(perpPlainSize, dxii, dirWidth, dirWidthO, gLW, dPD, uAux)
00767 for (sunindextype i = 0; i < perpPlainSize; i++) {
00768 #pragma omp simd
00769 for (sunindextype j = (i * dirWidth + gLW) * dPD;
00770 j < (i * dirWidth + gLW + dirWidthO) * dPD; j += dPD) {
00771     uAux[j + 0 - gLW * dPD] = s3b(&uAux[j + 0]) / dxii;
00772     uAux[j + 1 - gLW * dPD] = s3b(&uAux[j + 1]) / dxii;
00773     uAux[j + 2 - gLW * dPD] = s3f(&uAux[j + 2]) / dxii;
00774     uAux[j + 3 - gLW * dPD] = s3f(&uAux[j + 3]) / dxii;
00775     uAux[j + 4 - gLW * dPD] = s3f(&uAux[j + 4]) / dxii;
00776     uAux[j + 5 - gLW * dPD] = s3f(&uAux[j + 5]) / dxii;
00777 }
00778 }
00779 break;
00780 case 4: // gLW=3
00781 #pragma omp parallel for default(none) \
00782 shared(perpPlainSize, dxii, dirWidth, dirWidthO, gLW, dPD, uAux)
00783 for (sunindextype i = 0; i < perpPlainSize; i++) {
00784 #pragma omp simd
00785 for (sunindextype j = (i * dirWidth + gLW) * dPD;
00786 j < (i * dirWidth + gLW + dirWidthO) * dPD; j += dPD) {
00787     uAux[j + 0 - gLW * dPD] = s4b(&uAux[j + 0]) / dxii;
00788     uAux[j + 1 - gLW * dPD] = s4b(&uAux[j + 1]) / dxii;
00789     uAux[j + 2 - gLW * dPD] = s4f(&uAux[j + 2]) / dxii;
00790     uAux[j + 3 - gLW * dPD] = s4f(&uAux[j + 3]) / dxii;
00791     uAux[j + 4 - gLW * dPD] = s4c(&uAux[j + 4]) / dxii;
00792     uAux[j + 5 - gLW * dPD] = s4c(&uAux[j + 5]) / dxii;
00793 }
00794 }
00795 break;
00796 case 5: // gLW=3
00797 #pragma omp parallel for default(none) \
00798 shared(perpPlainSize, dxii, dirWidth, dirWidthO, gLW, dPD, uAux)
00799 for (sunindextype i = 0; i < perpPlainSize; i++) {
00800 #pragma omp simd
00801 for (sunindextype j = (i * dirWidth + gLW) * dPD;
00802 j < (i * dirWidth + gLW + dirWidthO) * dPD; j += dPD) {
00803     uAux[j + 0 - gLW * dPD] = s5b(&uAux[j + 0]) / dxii;
00804     uAux[j + 1 - gLW * dPD] = s5b(&uAux[j + 1]) / dxii;
00805     uAux[j + 2 - gLW * dPD] = s5f(&uAux[j + 2]) / dxii;
00806     uAux[j + 3 - gLW * dPD] = s5f(&uAux[j + 3]) / dxii;
00807     uAux[j + 4 - gLW * dPD] = s5f(&uAux[j + 4]) / dxii;
00808     uAux[j + 5 - gLW * dPD] = s5f(&uAux[j + 5]) / dxii;
00809 }
00810 }
00811 break;
00812 case 6: // gLW=4
00813 #pragma omp parallel for default(none) \
00814 shared(perpPlainSize, dxii, dirWidth, dirWidthO, gLW, dPD, uAux)
00815 for (sunindextype i = 0; i < perpPlainSize; i++) {
00816 #pragma omp simd
00817 for (sunindextype j = (i * dirWidth + gLW) * dPD;
00818 j < (i * dirWidth + gLW + dirWidthO) * dPD; j += dPD) {
00819     uAux[j + 0 - gLW * dPD] = s6b(&uAux[j + 0]) / dxii;
00820     uAux[j + 1 - gLW * dPD] = s6b(&uAux[j + 1]) / dxii;
00821     uAux[j + 2 - gLW * dPD] = s6f(&uAux[j + 2]) / dxii;
00822     uAux[j + 3 - gLW * dPD] = s6f(&uAux[j + 3]) / dxii;
00823     uAux[j + 4 - gLW * dPD] = s6c(&uAux[j + 4]) / dxii;
00824     uAux[j + 5 - gLW * dPD] = s6c(&uAux[j + 5]) / dxii;
00825 }
00826 }
00827 break;
00828 case 7: // gLW=4

```

```

00829 #pragma omp parallel for default(none) \
00830 shared(perpPlainSize, dxi, dirWidth, dirWidthO, gLW, dPD, uAux)
00831 for (sunindextype i = 0; i < perpPlainSize; i++) {
00832     #pragma omp simd
00833     for (sunindextype j = (i * dirWidth + gLW) * dPD;
00834         j < (i * dirWidth + gLW + dirWidthO) * dPD; j += dPD) {
00835         uAux[j + 0 - gLW * dPD] = s7b(&uAux[j + 0]) / dxi;
00836         uAux[j + 1 - gLW * dPD] = s7b(&uAux[j + 1]) / dxi;
00837         uAux[j + 2 - gLW * dPD] = s7f(&uAux[j + 2]) / dxi;
00838         uAux[j + 3 - gLW * dPD] = s7f(&uAux[j + 3]) / dxi;
00839         uAux[j + 4 - gLW * dPD] = s7f(&uAux[j + 4]) / dxi;
00840         uAux[j + 5 - gLW * dPD] = s7f(&uAux[j + 5]) / dxi;
00841     }
00842 }
00843 break;
00844 case 8: // gLW=5
00845 #pragma omp parallel for default(none) \
00846 shared(perpPlainSize, dxi, dirWidth, dirWidthO, gLW, dPD, uAux)
00847 for (sunindextype i = 0; i < perpPlainSize; i++) {
00848     #pragma omp simd
00849     for (sunindextype j = (i * dirWidth + gLW) * dPD;
00850         j < (i * dirWidth + gLW + dirWidthO) * dPD; j += dPD) {
00851         uAux[j + 0 - gLW * dPD] = s8b(&uAux[j + 0]) / dxi;
00852         uAux[j + 1 - gLW * dPD] = s8b(&uAux[j + 1]) / dxi;
00853         uAux[j + 2 - gLW * dPD] = s8f(&uAux[j + 2]) / dxi;
00854         uAux[j + 3 - gLW * dPD] = s8f(&uAux[j + 3]) / dxi;
00855         uAux[j + 4 - gLW * dPD] = s8c(&uAux[j + 4]) / dxi;
00856         uAux[j + 5 - gLW * dPD] = s8c(&uAux[j + 5]) / dxi;
00857     }
00858 }
00859 break;
00860 case 9: // gLW=5
00861 #pragma omp parallel for default(none) \
00862 shared(perpPlainSize, dxi, dirWidth, dirWidthO, gLW, dPD, uAux)
00863 for (sunindextype i = 0; i < perpPlainSize; i++) {
00864     #pragma omp simd
00865     for (sunindextype j = (i * dirWidth + gLW) * dPD;
00866         j < (i * dirWidth + gLW + dirWidthO) * dPD; j += dPD) {
00867         uAux[j + 0 - gLW * dPD] = s9b(&uAux[j + 0]) / dxi;
00868         uAux[j + 1 - gLW * dPD] = s9b(&uAux[j + 1]) / dxi;
00869         uAux[j + 2 - gLW * dPD] = s9f(&uAux[j + 2]) / dxi;
00870         uAux[j + 3 - gLW * dPD] = s9f(&uAux[j + 3]) / dxi;
00871         uAux[j + 4 - gLW * dPD] = s9f(&uAux[j + 4]) / dxi;
00872         uAux[j + 5 - gLW * dPD] = s9f(&uAux[j + 5]) / dxi;
00873     }
00874 }
00875 break;
00876 case 10: // gLW=6
00877 #pragma omp parallel for default(none) \
00878 shared(perpPlainSize, dxi, dirWidth, dirWidthO, gLW, dPD, uAux)
00879 for (sunindextype i = 0; i < perpPlainSize; i++) {
00880     #pragma omp simd
00881     for (sunindextype j = (i * dirWidth + gLW) * dPD;
00882         j < (i * dirWidth + gLW + dirWidthO) * dPD; j += dPD) {
00883         uAux[j + 0 - gLW * dPD] = s10b(&uAux[j + 0]) / dxi;
00884         uAux[j + 1 - gLW * dPD] = s10b(&uAux[j + 1]) / dxi;
00885         uAux[j + 2 - gLW * dPD] = s10f(&uAux[j + 2]) / dxi;
00886         uAux[j + 3 - gLW * dPD] = s10f(&uAux[j + 3]) / dxi;
00887         uAux[j + 4 - gLW * dPD] = s10c(&uAux[j + 4]) / dxi;
00888         uAux[j + 5 - gLW * dPD] = s10c(&uAux[j + 5]) / dxi;
00889     }
00890 }
00891 break;
00892 case 11: // gLW=6
00893 #pragma omp parallel for default(none) \
00894 shared(perpPlainSize, dxi, dirWidth, dirWidthO, gLW, dPD, uAux)
00895 for (sunindextype i = 0; i < perpPlainSize; i++) {
00896     #pragma omp simd
00897     for (sunindextype j = (i * dirWidth + gLW) * dPD;
00898         j < (i * dirWidth + gLW + dirWidthO) * dPD; j += dPD) {
00899         uAux[j + 0 - gLW * dPD] = s11b(&uAux[j + 0]) / dxi;
00900         uAux[j + 1 - gLW * dPD] = s11b(&uAux[j + 1]) / dxi;
00901         uAux[j + 2 - gLW * dPD] = s11f(&uAux[j + 2]) / dxi;
00902         uAux[j + 3 - gLW * dPD] = s11f(&uAux[j + 3]) / dxi;
00903         uAux[j + 4 - gLW * dPD] = s11f(&uAux[j + 4]) / dxi;
00904         uAux[j + 5 - gLW * dPD] = s11f(&uAux[j + 5]) / dxi;
00905     }
00906 }
00907 break;
00908 case 12: // gLW=7
00909 #pragma omp parallel for default(none) \
00910 shared(perpPlainSize, dxi, dirWidth, dirWidthO, gLW, dPD, uAux)
00911 for (sunindextype i = 0; i < perpPlainSize; i++) {
00912     #pragma omp simd
00913     for (sunindextype j = (i * dirWidth + gLW) * dPD;
00914         j < (i * dirWidth + gLW + dirWidthO) * dPD; j += dPD) {
00915         uAux[j + 0 - gLW * dPD] = s12b(&uAux[j + 0]) / dxi;

```

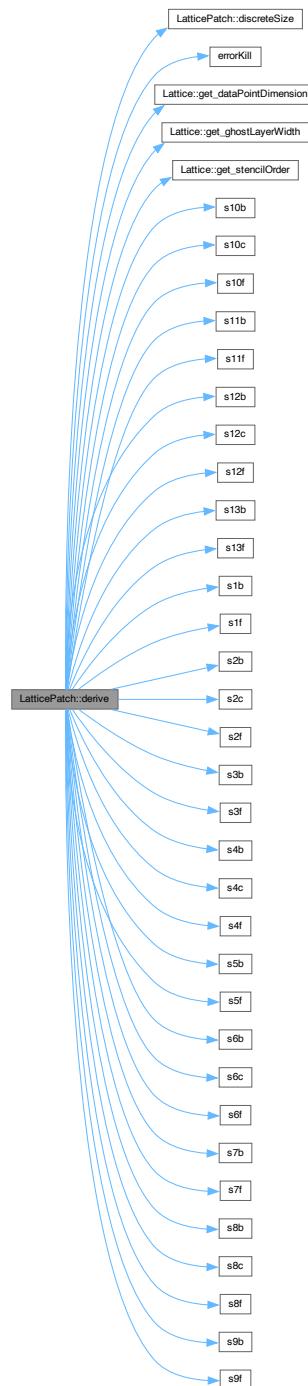
```

00916     uAux[j + 1 - gLW * dPD] = s12b(&uAux[j + 1]) / dx;
00917     uAux[j + 2 - gLW * dPD] = s12f(&uAux[j + 2]) / dx;
00918     uAux[j + 3 - gLW * dPD] = s12f(&uAux[j + 3]) / dx;
00919     uAux[j + 4 - gLW * dPD] = s12c(&uAux[j + 4]) / dx;
00920     uAux[j + 5 - gLW * dPD] = s12c(&uAux[j + 5]) / dx;
00921 }
00922 }
00923 break;
00924 case 13: // gLW=7
00925 // For all points in the plane perpendicular to the given direction
00926 #pragma omp parallel for default(none) \
00927 shared(perpPlainSize, dx, dirWidth, dirWidthO, gLW, dPD, uAux)
00928 for (sunindextype i = 0; i < perpPlainSize; i++) {
00929     // iterate through the derivation direction
00930     #pragma omp simd
00931     for (sunindextype j = (i * dirWidth
00932             + gLW /*to shift left by gLW below */) * dPD;
00933         j < (i * dirWidth + gLW + dirWidthO) * dPD; j += dPD) {
00934         // Compute the stencil derivative for any of the six field components
00935         // and update position by ghost width shift
00936         uAux[j + 0 - gLW * dPD] = s13b(&uAux[j + 0]) / dx;
00937         uAux[j + 1 - gLW * dPD] = s13b(&uAux[j + 1]) / dx;
00938         uAux[j + 2 - gLW * dPD] = s13f(&uAux[j + 2]) / dx;
00939         uAux[j + 3 - gLW * dPD] = s13f(&uAux[j + 3]) / dx;
00940         uAux[j + 4 - gLW * dPD] = s13f(&uAux[j + 4]) / dx;
00941         uAux[j + 5 - gLW * dPD] = s13f(&uAux[j + 5]) / dx;
00942     }
00943 }
00944 break;
00945 default:
00946     errorKill("Please set an existing stencil order");
00947     break;
00948 }
00949 }
00950 }
```

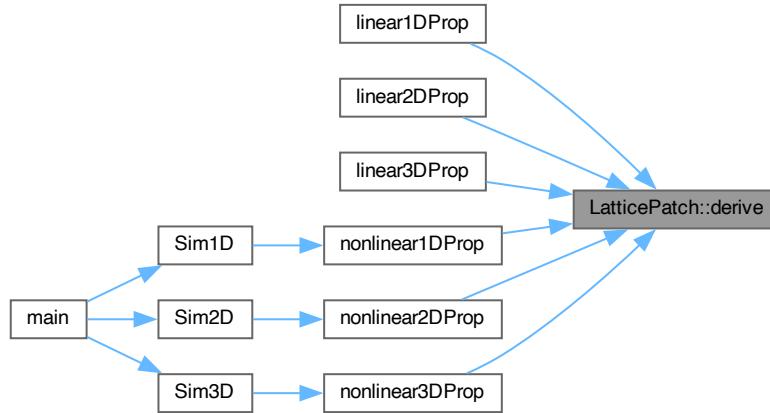
References [discreteSize\(\)](#), [dx](#), [dy](#), [dz](#), [envelopeLattice](#), [errorKill\(\)](#), [Lattice::get_dataPointDimension\(\)](#), [Lattice::get_ghostLayerWidth\(\)](#), [Lattice::get_stencilOrder\(\)](#), [s10b\(\)](#), [s10c\(\)](#), [s10f\(\)](#), [s11b\(\)](#), [s11f\(\)](#), [s12b\(\)](#), [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 478 of file [LatticePatch.cpp](#).

```
00478 // Check that the lattice as well as the translocation lookups have been set
00479 // up;
00480 checkFlag(FLatticePatchSetUp);
00481 checkFlag(TranslocationLookupSetUp);
00482 const int dPD = envelopeLattice->get_dataPointDimension();
00483 const int gLW = envelopeLattice->get_ghostLayerWidth();
00484 const sunindextype totalNP = discreteSize();
00485 sunindextype ii = 0, target = 0;
00486 switch (dir) {
00487 case 1:
00488 #pragma omp parallel for simd \
00489 private(ii, target) \
00490 shared(dPD, gLW, totalNP, uTox, uAux, buffOut) \
00491 schedule(static)
00492 for (sunindextype i = 0; i < totalNP; i++) {
00493     // get correct indices in u and rotation space
00494     target = dPD * i;
00495     ii = dPD * (uTox[i] - gLW);
00496     buffOut[target + 0] = uAux[5 + ii];
00497     buffOut[target + 1] = (-uAux[ii] + uAux[2 + ii]) / 2.;
00498     buffOut[target + 2] = (uAux[1 + ii] - uAux[3 + ii]) / 2.;
00499     buffOut[target + 3] = uAux[4 + ii];
00500     buffOut[target + 4] = (uAux[1 + ii] + uAux[3 + ii]) / 2.;
00501     buffOut[target + 5] = (uAux[ii] + uAux[2 + ii]) / 2.;
00502 }
00503 break;
00504 case 2:
00505 #pragma omp parallel for simd \
00506 private(ii, target) \
```

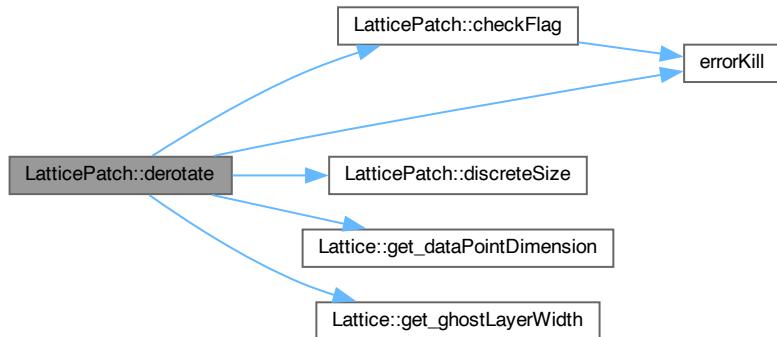
```

00508     shared(dPD, gLW, totalNP, uTox, uAux, buffOut) \
00509     schedule(static)
00510     for (sunindextype i = 0; i < totalNP; i++) {
00511         target = dPD * i;
00512         ii = dPD * (uToy[i] - gLW);
00513         buffOut[target + 0] = (uAux[ii] - uAux[2 + ii]) / 2.;
00514         buffOut[target + 1] = uAux[5 + ii];
00515         buffOut[target + 2] = (-uAux[1 + ii] + uAux[3 + ii]) / 2.;
00516         buffOut[target + 3] = (uAux[1 + ii] + uAux[3 + ii]) / 2.;
00517         buffOut[target + 4] = uAux[4 + ii];
00518         buffOut[target + 5] = (uAux[ii] + uAux[2 + ii]) / 2.;
00519     }
00520     break;
00521 case 3:
00522     #pragma omp parallel for simd \
00523     private(ii, target) \
00524     shared(dPD, gLW, totalNP, uTox, uAux, buffOut) \
00525     schedule(static)
00526     for (sunindextype i = 0; i < totalNP; i++) {
00527         target = dPD * i;
00528         ii = dPD * (uToz[i] - gLW);
00529         buffOut[target + 0] = (-uAux[ii] + uAux[2 + ii]) / 2.;
00530         buffOut[target + 1] = (uAux[1 + ii] - uAux[3 + ii]) / 2.;
00531         buffOut[target + 2] = uAux[5 + ii];
00532         buffOut[target + 3] = (uAux[1 + ii] + uAux[3 + ii]) / 2.;
00533         buffOut[target + 4] = (uAux[ii] + uAux[2 + ii]) / 2.;
00534         buffOut[target + 5] = uAux[4 + ii];
00535     }
00536     break;
00537 default:
00538     errorKill("Tried to derotate from the wrong direction");
00539     break;
00540 }
00541 }
```

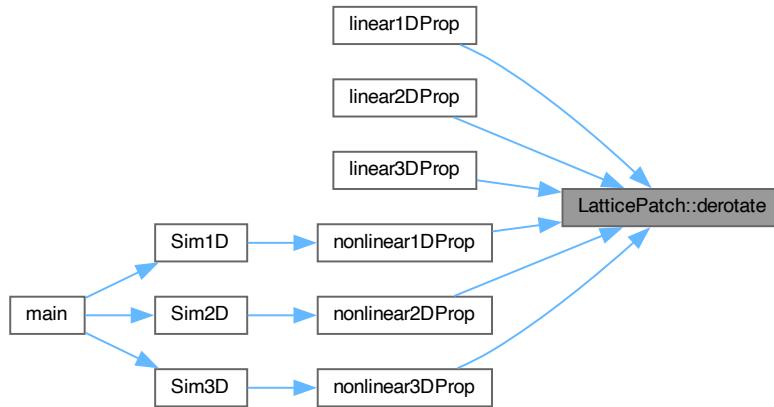
References [checkFlag\(\)](#), [discreteSize\(\)](#), [envelopeLattice](#), [errorKill\(\)](#), [FLatticePatchSetUp](#), [Lattice::get_dataPointDimension\(\)](#), [Lattice::get_ghostLayerWidth\(\)](#), [TranslocationLookupSetUp](#), [uAux](#), [uTox](#), [uToy](#), and [uToz](#).

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

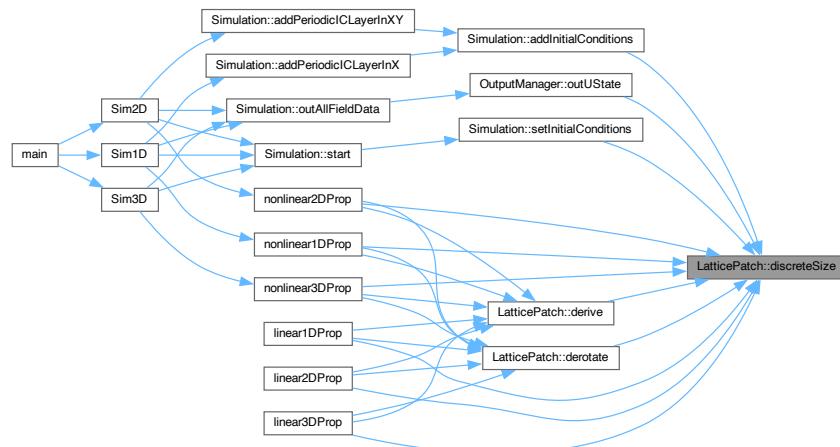
Definition at line [235](#) of file [LatticePatch.cpp](#).

```
00235
00236     switch (dir) {
00237         case 0:
00238             return nx * ny * nz;
00239         case 1:
00240             return nx;
00241         case 2:
00242             return ny;
00243         case 3:
00244             return nz;
00245         // case 4: return uAux.size(); // for debugging
00246     default:
00247         return -1;
00248     }
00249 }
```

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 559 of file [LatticePatch.cpp](#).

```
00559
00560 // Check that the lattice has been set up
00561 checkFlag(FLatticeDimensionSet);
00562 checkFlag(FLatticePatchSetUp);
00563 // Variables to per dimension calculate the halo indices, and distance to
00564 // other side halo boundary
00565 int mx = 1, my = 1, mz = 1, distToRight = 1;
00566 const int gLW = envelopeLattice->get_ghostLayerWidth();
00567 // In the chosen direction m is set to ghost layer width while the others
00568 // remain to form the plane
00569 switch (dir) {
00570 case 1:
00571     mx = gLW;
00572     my = ny;
00573     mz = nz;
00574     distToRight = (nx - gLW);
00575     break;
00576 case 2:
00577     mx = nx;
00578     my = gLW;
00579     mz = nz;
00580     distToRight = nx * (ny - gLW);
00581     break;
00582 case 3:
00583     mx = nx;
00584     my = ny;
00585     mz = gLW;
00586     distToRight = nx * ny * (nz - gLW);
00587     break;
00588 }
00589 // total number of exchanged points
00590 const int dPD = envelopeLattice->get_dataPointDimension();
00591 const sunindextype exchangeSize = mx * my * mz * dPD;
00592 // provide size of the halos for ghost cells
```

```

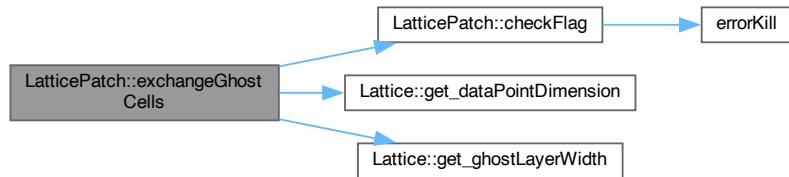
00593     ghostCellLeft.resize(exchangeSize);
00594     ghostCellRight.resize(ghostCellLeft.size());
00595     ghostCellLeftToSend.resize(ghostCellLeft.size());
00596     ghostCellRightToSend.resize(ghostCellLeft.size());
00597     statusFlags |= GhostLayersInitialized;
00598
00599 // Initialize running index li for the halo buffers, and index ui of uData for
00600 // data transfer
00601 sunindextype li = 0, ui = 0;
00602 // Fill the halo buffers
00603 #pragma omp parallel for default(none) \
00604 private(ui, li) \
00605 shared(nx, ny, mx, my, mz, dPD, distToRight, uData, \
00606         ghostCellLeftToSend, ghostCellRightToSend)
00607 for (sunindextype iz = 0; iz < mz; iz++) {
00608     for (sunindextype iy = 0; iy < my; iy++) {
00609         // uData vector start index of halo data to be transferred
00610         // with each z-step add the whole xy-plane and with y-step the x-range ->
00611         // iterate all x-ranges
00612         ui = (iz * nx * ny + iy * nx) * dPD;
00613         // increase halo index by transferred items of previous iteration steps
00614         li = (iz * my * mx + iy * mx) * dPD;
00615         // copy left halo data from uData to buffer, transfer size is given by
00616         // x-length (not x-range)
00617         std::copy(&uData[ui], &uData[ui + mx * dPD], &ghostCellLeftToSend[li]);
00618         ui += distToRight * dPD;
00619         std::copy(&uData[ui], &uData[ui + mx * dPD], &ghostCellRightToSend[li]);
00620     }
00621 }
00622
00623 #if !defined(_MPI)
00624     std::copy(&ghostCellLeftToSend[0], &ghostCellLeftToSend[exchangeSize],
00625               &ghostCellRight[0]);
00626     std::copy(&ghostCellRightToSend[0], &ghostCellRightToSend[exchangeSize],
00627               &ghostCellLeft[0]);
00628
00629 #elif defined(_MPI)
00630     /* Send and receive the data to and from neighboring latticePatches */
00631     // Adjust direction to cartesian communicator
00632     int dim = 2; // default for dir==1
00633     if (dir == 2) {
00634         dim = 1;
00635     } else if (dir == 3) {
00636         dim = 0;
00637     }
00638     int rank_source = 0, rank_dest = 0;
00639     MPI_Cart_shift(envelopeLattice->comm, dim, -1, &rank_source,
00640                    &rank_dest); // s.t. rank_dest is left & v.v.
00641
00642     // nonblocking Irecv/Isend
00643
00644     MPI_Request requests[4];
00645     MPI_Irecv(&ghostCellRight[0], exchangeSize, MPI_SUNREALTYPE, rank_source, 1,
00646                envelopeLattice->comm, &requests[0]);
00647     MPI_Isend(&ghostCellLeftToSend[0], exchangeSize, MPI_SUNREALTYPE, rank_dest,
00648                1, envelopeLattice->comm, &requests[1]);
00649     MPI_Irecv(&ghostCellLeft[0], exchangeSize, MPI_SUNREALTYPE, rank_dest, 2,
00650                envelopeLattice->comm, &requests[2]);
00651     MPI_Isend(&ghostCellRightToSend[0], exchangeSize, MPI_SUNREALTYPE,
00652                rank_source, 2, envelopeLattice->comm, &requests[3]);
00653     MPI_Waitall(4, requests, MPI_STATUS_IGNORE);
00654
00655     // blocking Sendrecv:
00656     /*
00657     MPI_Sendrecv(&ghostCellLeftToSend[0], exchangeSize, MPI_SUNREALTYPE,
00658                  rank_dest, 1, &ghostCellRight[0], exchangeSize, MPI_SUNREALTYPE,
00659                  rank_source, 1, envelopeLattice->comm, MPI_STATUS_IGNORE);
00660     MPI_Sendrecv(&ghostCellRightToSend[0], exchangeSize, MPI_SUNREALTYPE,
00661                  rank_source, 2, &ghostCellLeft[0], exchangeSize, MPI_SUNREALTYPE,
00662                  rank_dest, 2, envelopeLattice->comm, MPI_STATUS_IGNORE);
00663     */
00664 #endif
00665     gCLData = &ghostCellLeft[0];
00666     gCRData = &ghostCellRight[0];
00667 }

```

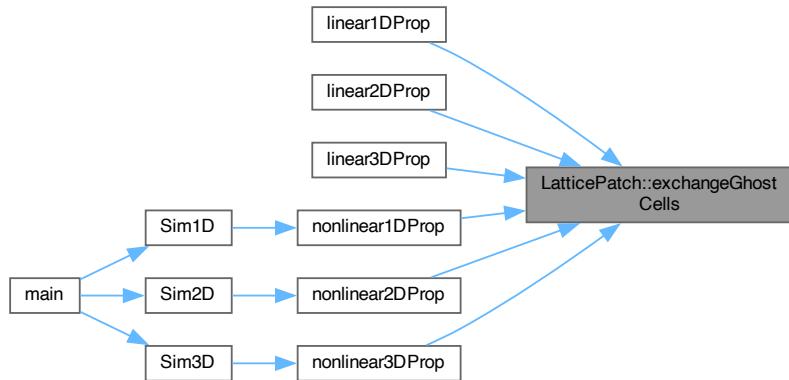
References `checkFlag()`, `Lattice::comm`, `envelopeLattice`, `FLatticeDimensionSet`, `FLatticePatchSetUp`, `gCLData`, `gCRData`, `Lattice::get_dataPointDimension()`, `Lattice::get_ghostLayerWidth()`, `ghostCellLeft`, `ghostCellLeftToSend`, `ghostCellRight`, `ghostCellRightToSend`, `GhostLayersInitialized`, `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 285 of file [LatticePatch.cpp](#).

```

00285
00286 // Check that the lattice has been set up
00287 checkFlag(FLatticeDimensionSet);
00288 // lengths for auxilliary layers, including ghost layers
00289 const int gLW = envelopeLattice->get_ghostLayerWidth();
00290 const sunindextype mx = nx + 2 * gLW;
00291 const sunindextype my = ny + 2 * gLW;
00292 const sunindextype mz = nz + 2 * gLW;
00293 // sizes for lookup vectors
00294 const sunindextype totalNP = nx * ny * nz;
00295 const sunindextype haloXSize = mx * ny * nz;
00296 const sunindextype haloYSIZE = nx * my * nz;
00297 const sunindextype haloZSize = nx * ny * mz;
  
```

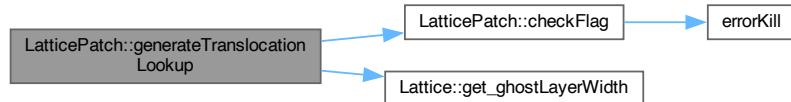
```

00298 // generate u->uAux
00299 uTox.resize(totalNP);
00300 uToy.resize(totalNP);
00301 uToz.resize(totalNP);
00302 // generate uAux->u with length including halo
00303 xTou.resize(haloXSize);
00304 yTou.resize(haloYSize);
00305 zTou.resize(haloZSize);
00306 // same for ghost layer lookup tables
00307 const sunindextype ghostXSize = gLW * ny * nz;
00308 const sunindextype ghostYSize = gLW * nx * nz;
00309 const sunindextype ghostZSize = gLW * nx * ny;
00310 lgcTox.resize(ghostXSize);
00311 rgcTox.resize(ghostXSize);
00312 lgcToy.resize(ghostYSize);
00313 rgcToy.resize(ghostYSize);
00314 lgcToz.resize(ghostZSize);
00315 rgcToz.resize(ghostZSize);
00316 // variables for cartesian position in the 3D discrete lattice
00317 sunindextype px = 0, py = 0, pz = 0;
00318 // Fill the lookup tables
00319 #pragma omp parallel default(none) \
00320 private(px, py, pz) \
00321 shared(uTox, uToy, uToz, xTou, yTou, zTou, \
00322     nx, ny, mx, my, mz, gLW, totalNP, \
00323     lgcTox, rgcTox, lgcToy, rgcToy, lgcToz, rgcToz, \
00324     ghostXSize, ghostYSize, ghostZSize)
00325 {
00326 #pragma omp for simd schedule(static)
00327 for (sunindextype i = 0; i < totalNP; i++) { // loop over the patch
00328     // calculate cartesian coordinates
00329     px = i % nx;
00330     py = (i / nx) % ny;
00331     pz = (i / nx) / ny;
00332     // fill lookups extended by halos (useful for y and z direction)
00333     uTox[i] = (px + gLW) + py * mx +
00334         pz * mx * ny; // unroll (de-flatten) cartesian dimension
00335     xTou[px + py * mx + pz * mx * ny] =
00336         i; // match cartesian point to u location
00337     uToy[i] = (py + gLW) + pz * my + px * my * nz;
00338     yTou[py + pz * my + px * my * nz] = i;
00339     uToz[i] = (pz + gLW) + px * mz + py * mz * nx;
00340     zTou[pz + px * mz + py * mz * nx] = i;
00341 }
00342 #pragma omp for simd schedule(static)
00343 for (sunindextype i = 0; i < ghostXSize; i++) {
00344     px = i % gLW;
00345     py = (i / gLW) % ny;
00346     pz = (i / gLW) / ny;
00347     lgcTox[i] = px + py * mx + pz * mx * ny;
00348     rgcTox[i] = px + nx + gLW + py * mx + pz * mx * ny;
00349 }
00350 #pragma omp for simd schedule(static)
00351 for (sunindextype i = 0; i < ghostYSize; i++) {
00352     px = i % nx;
00353     py = (i / nx) % gLW;
00354     pz = (i / nx) / gLW;
00355     lgcToy[i] = py + pz * my + px * my * nz;
00356     rgcToy[i] = py + ny + gLW + pz * my + px * my * nz;
00357 }
00358 #pragma omp for simd schedule(static)
00359 for (sunindextype i = 0; i < ghostZSize; i++) {
00360     px = i % nx;
00361     py = (i / nx) % ny;
00362     pz = (i / nx) / ny;
00363     lgcToz[i] = pz + px * mz + py * mz * nx;
00364     rgcToz[i] = pz + nz + gLW + px * mz + py * mz * nx;
00365 }
00366 }
00367 statusFlags |= TranslocationLookupSetUp;
00368 }

```

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 267 of file [LatticePatch.cpp](#).

```
00267
00268     switch (dir) {
00269     case 1:
00270         return dx;
00271     case 2:
00272         return dy;
00273     case 3:
00274         return dz;
00275     default:
00276         errorKill(
00277             "LatticePatch::getDelta function called with wrong dir parameter");
00278         return -1;
00279     }
00280 }
```

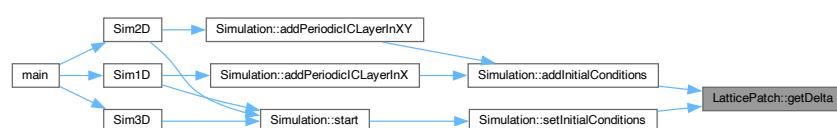
References [dx](#), [dy](#), [dz](#), and [errorKill\(\)](#).

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 544 of file [LatticePatch.cpp](#).

```
00544 {
00545     // Check that the lattice has been set up
00546     checkFlag(FLatticeDimensionSet);
00547     const int dPD = envelopeLattice->get_dataPointDimension();
00548     buffX.resize(nx * ny * nz * dPD);
00549     buffY.resize(nx * ny * nz * dPD);
00550     buffZ.resize(nx * ny * nz * dPD);
00551     // Set pointers used for propagation functions
00552     buffData[0] = &buffX[0];
00553     buffData[1] = &buffY[0];
00554     buffData[2] = &buffZ[0];
00555     statusFlags |= BuffersInitialized;
00556 }
```

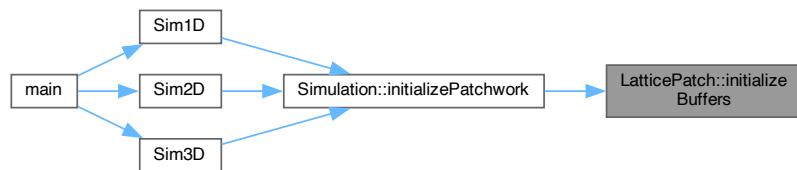
References [buffData](#), [BuffersInitialized](#), [buffX](#), [buffY](#), [buffZ](#), [checkFlag\(\)](#), [envelopeLattice](#), [FLatticeDimensionSet](#), [Lattice::get_dataPointDimension\(\)](#), [nx](#), [ny](#), [nz](#), and [statusFlags](#).

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 252 of file [LatticePatch.cpp](#).

```
00252
00253     switch (dir) {
00254         case 1:
00255             return x0;
00256         case 2:
00257             return y0;
00258         case 3:
00259             return z0;
00260     default:
00261         errorKill("LatticePatch::origin function called with wrong dir parameter");
00262         return -1;
00263     }
00264 }
```

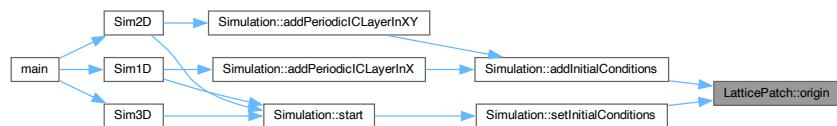
References [errorKill\(\)](#), [x0](#), [y0](#), and [z0](#).

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

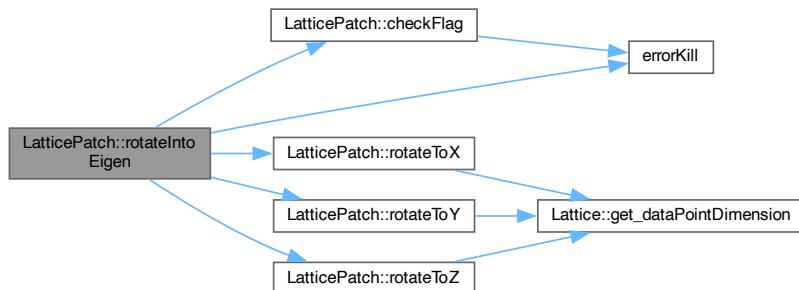
Definition at line 373 of file [LatticePatch.cpp](#).

```
00373
00374     // Check that the lattice, ghost layers as well as the translocation lookups
00375     // have been set up;
00376     checkFlag(FLatticePatchSetUp);
00377     checkFlag(TranslocationLookupSetUp);
00378     checkFlag(GhostLayersInitialized); // this check is only after call to
00379     // exchange ghost cells
00380     switch (dir) {
00381     case 1:
00382         rotateToX(uAuxData, gCLData, lgcTox);
00383         rotateToX(uAuxData, uData, uTox);
00384         rotateToX(uAuxData, gCRData, rgcTox);
00385         break;
00386     case 2:
00387         rotateToY(uAuxData, gCLData, lgcToy);
00388         rotateToY(uAuxData, uData, uToy);
00389         rotateToY(uAuxData, gCRData, rgcToy);
00390         break;
00391     case 3:
00392         rotateToZ(uAuxData, gCLData, lgcToz);
00393         rotateToZ(uAuxData, uData, uToz);
00394         rotateToZ(uAuxData, gCRData, rgcToz);
00395         break;
00396     default:
00397         errorKill("Tried to rotate into the wrong direction");
00398         break;
00399     }
00400 }
```

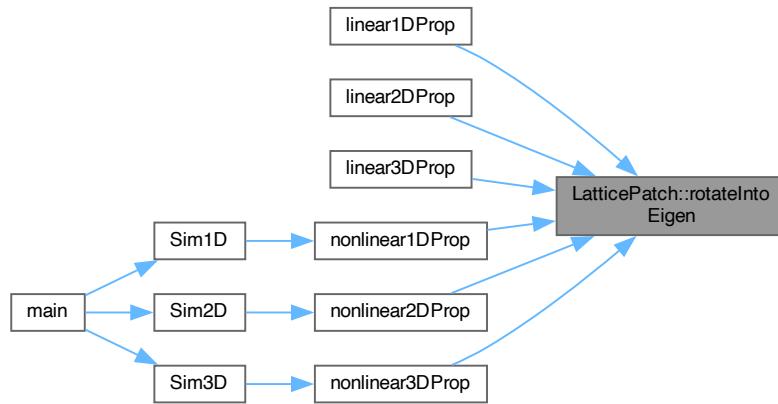
References [checkFlag\(\)](#), [errorKill\(\)](#), [FLatticePatchSetUp](#), [gCLData](#), [gCRData](#), [GhostLayersInitialized](#), [lgcTox](#), [lgcToy](#), [lgcToz](#), [rgcTox](#), [rgcToy](#), [rgcToz](#), [rotateToX\(\)](#), [rotateToY\(\)](#), [rotateToZ\(\)](#), [TranslocationLookupSetUp](#), [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 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         // get correct u-vector and spatial indices along previously defined lookup
00416         // tables
00417         target = dPD * lookup[i];
00418         ii = dPD * i;
00419         outArray[target + 0] = -inArray[1 + ii] + inArray[5 + ii];
00420         outArray[target + 1] = inArray[2 + ii] + inArray[4 + ii];
00421         outArray[target + 2] = inArray[1 + ii] + inArray[5 + ii];
00422         outArray[target + 3] = -inArray[2 + ii] + inArray[4 + ii];
00423         outArray[target + 4] = inArray[3 + ii];
00424         outArray[target + 5] = inArray[ii];
00425     }
00426 }
  
```

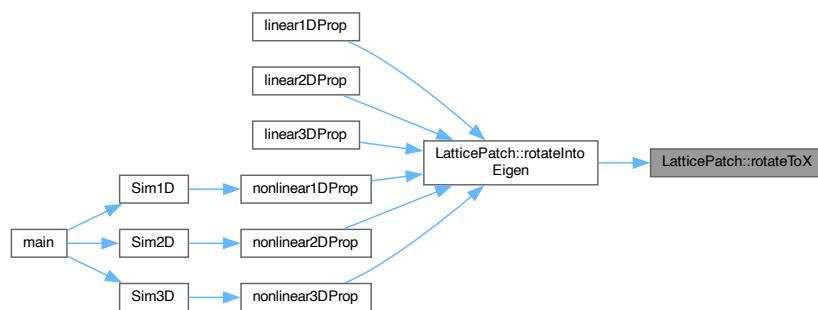
References [envelopeLattice](#), and [Lattice::get_dataPointDimension\(\)](#).

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 430 of file [LatticePatch.cpp](#).

```

00432
00433     sunindextype ii = 0, target = 0;
00434     const int dPD = envelopeLattice->get_dataPointDimension();
00435     const sunindextype size = lookup.size();
00436     #pragma omp parallel for simd \
00437     private(target, ii) \
00438     shared(lookup, outArray, inArray, size, dPD) \
00439     schedule(static)
00440     for (sunindextype i = 0; i < size; i++) {
00441         target = dPD * lookup[i];
00442         ii = dPD * i;
00443         outArray[target + 0] = inArray[ii] + inArray[5 + ii];
00444         outArray[target + 1] = -inArray[2 + ii] + inArray[3 + ii];
00445         outArray[target + 2] = -inArray[ii] + inArray[5 + ii];
00446         outArray[target + 3] = inArray[2 + ii] + inArray[3 + ii];
00447         outArray[target + 4] = inArray[4 + ii];
00448         outArray[target + 5] = inArray[1 + ii];
00449     }
00450 }
  
```

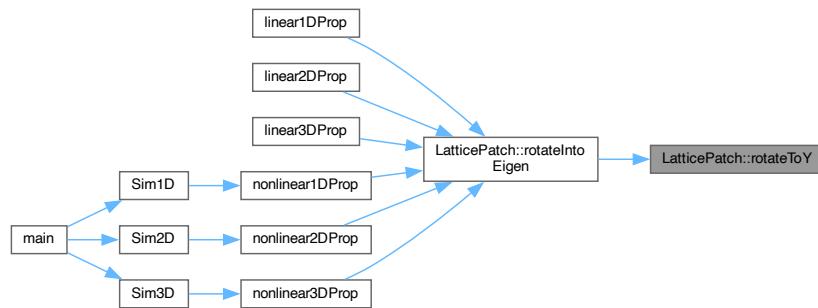
References [envelopeLattice](#), and [Lattice::get_dataPointDimension\(\)](#).

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 Rz into eigenspace of Z matrix and write to auxiliary vector

Definition at line 454 of file [LatticePatch.cpp](#).

```

00456
00457     sunindextype ii = 0, target = 0;
00458     const sunindextype size = lookup.size();
00459     const int dPD = envelopeLattice->get_dataPointDimension();
00460     #pragma omp parallel for simd \
00461     private(target, ii) \
00462     shared(lookup, outArray, inArray, size, dPD) \
00463     schedule(static)
00464     for (sunindextype i = 0; i < size; i++) {
00465         target = dPD * lookup[i];
00466         ii = dPD * i;
00467         outArray[target + 0] = -inArray[ii] + inArray[4 + ii];
00468         outArray[target + 1] = inArray[1 + ii] + inArray[3 + ii];
00469         outArray[target + 2] = inArray[ii] + inArray[4 + ii];

```

```

00470     outArray[target + 3] = -inArray[1 + ii] + inArray[3 + ii];
00471     outArray[target + 4] = inArray[5 + ii];
00472     outArray[target + 5] = inArray[2 + ii];
00473 }
00474 }
```

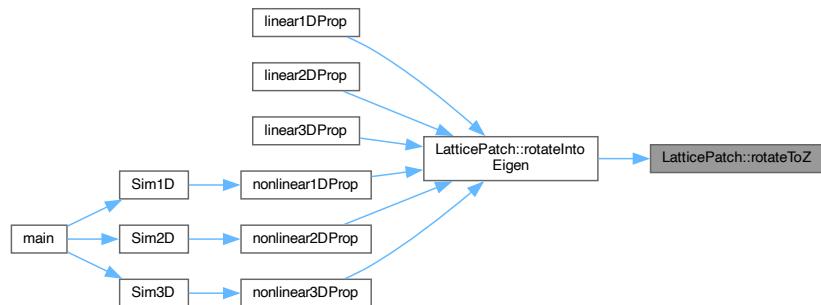
References [envelopeLattice](#), and [Lattice::get_dataPointDimension\(\)](#).

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 119 of file [LatticePatch.cpp](#).

00121 {

```

00122 // Retrieve the ghost layer depth
00123 const int gLW = envelopeLattice.get_ghostLayerWidth();
00124 // Retrieve the data point dimension
00125 const int dPD = envelopeLattice.get_dataPointDimension();
00126 // MPI process/patch
00127 const int my_prc = envelopeLattice.my_prc;
00128 // Determine thickness of the slice
00129 const sunindextype tot_NOXP = envelopeLattice.get_tot_nx();
00130 const sunindextype tot_NOYP = envelopeLattice.get_tot_ny();
00131 const sunindextype tot_NOZP = envelopeLattice.get_tot_nz();
00132 // position of the patch in the lattice of patches -> process associated to
00133 // position
00134 const sunindextype LIx = my_prc % DLx;
00135 const sunindextype LIy = (my_prc / DLx) % DLy;
00136 const sunindextype LIz = (my_prc / DLx) / DLy;
00137 // Determine the number of points in the patch and first absolute points in
00138 // each dimension
00139 const sunindextype local_NOXP = tot_NOXP / DLx;
00140 const sunindextype local_NOYP = tot_NOYP / DLy;
00141 const sunindextype local_NOZP = tot_NOZP / DLz;
00142 // absolute positions of the first point in each dimension
00143 const sunindextype firstXPoint = local_NOXP * LIx;
00144 const sunindextype firstYPoint = local_NOYP * LIy;
00145 const sunindextype firstZPoint = local_NOZP * LIz;
00146 #if defined(_MPI)
00147 // total number of points in a patch
00148 const sunindextype local_NODP = dPD * local_NOXP * local_NOYP * local_NOZP;
00149 #endif
00150 // Set patch up with above derived quantities
00151 patchToMold.dx = envelopeLattice.get_dx();
00152 patchToMold.dy = envelopeLattice.get_dy();
00153 patchToMold.dz = envelopeLattice.get_dz();
00154 patchToMold.x0 = firstXPoint * patchToMold.dx;
00155 patchToMold.y0 = firstYPoint * patchToMold.dy;
00156 patchToMold.z0 = firstZPoint * patchToMold.dz;
00157 patchToMold.LIx = LIx;
00158 patchToMold.LIy = LIy;
00159 patchToMold.LIz = LIz;
00160 patchToMold.nx = local_NOXP;
00161 patchToMold.ny = local_NOYP;
00162 patchToMold.nz = local_NOZP;
00163 patchToMold.lx = patchToMold.nx * patchToMold.dx;
00164 patchToMold.ly = patchToMold.ny * patchToMold.dy;
00165 patchToMold.lz = patchToMold.nz * patchToMold.dz;
00166
00167 #if defined(_MPI)
00168 #if defined(_OPENMP) // OpenMP and MPI+X NVectors interoperability
00169 // OpenMP NVectors with local patch size
00170 int num_threads = 1;
00171 num_threads = omp_get_max_threads();
00172 patchToMold.ulocal = N_VNew_OpenMP(local_NODP, num_threads,
00173 envelopeLattice.sunctx);
00174 patchToMold.dulocal = N_VNew_OpenMP(local_NODP, num_threads,
00175 envelopeLattice.sunctx);
00176 // MPI+X NVectors containing local OpenMP NVectors
00177 patchToMold.u = N_VMake_MPIPlusX(envelopeLattice.comm, patchToMold.ulocal,
00178 envelopeLattice.sunctx);
00179 patchToMold.du = N_VMake_MPIPlusX(envelopeLattice.comm, patchToMold.dulocal,
00180 envelopeLattice.sunctx);
00181 // Pointers to local vectors
00182 patchToMold.uData = N_VGetArrayPointer_MPIPlusX(patchToMold.u);
00183 patchToMold.duData = N_VGetArrayPointer_MPIPlusX(patchToMold.du);
00184 #else // only MPI
00185 // MPI NVectors with local patch and global lattice size
00186 patchToMold.u =
00187 N_VNew_Parallel(envelopeLattice.comm, local_NODP,
00188 envelopeLattice.get_tot_noDP(), envelopeLattice.sunctx);
00189 patchToMold.du =
00190 N_VNew_Parallel(envelopeLattice.comm, local_NODP,
00191 envelopeLattice.get_tot_noDP(), envelopeLattice.sunctx);
00192 patchToMold.uData = NV_DATA_P(patchToMold.u);
00193 patchToMold.duData = NV_DATA_P(patchToMold.du);
00194 #endif
00195 #elif defined(_OPENMP) // only OpenMP
00196 // OpenMP NVectors with global lattice size
00197 int num_threads = 1;
00198 num_threads = omp_get_max_threads();
00199 patchToMold.u =
00200 N_VNew_OpenMP(envelopeLattice.get_tot_noDP(), num_threads,
00201 envelopeLattice.sunctx);
00202 patchToMold.du =
00203 N_VNew_OpenMP(envelopeLattice.get_tot_noDP(), num_threads,
00204 envelopeLattice.sunctx);
00205 patchToMold.uData = NV_DATA_OMP(patchToMold.u);
00206 patchToMold.duData = NV_DATA_OMP(patchToMold.du);
00207 #else // just serial
00208 // Serial NVectors with global lattice size

```

```

00209 patchToMold.u =
00210     N_VNew_Serial(envelopeLattice.get_tot_noDP(), envelopeLattice.sunctx);
00211 patchToMold.du =
00212     N_VNew_Serial(envelopeLattice.get_tot_noDP(), envelopeLattice.sunctx);
00213 patchToMold.uData = NV_DATA_S(patchToMold.u);
00214 patchToMold.duData = NV_DATA_S(patchToMold.du);
00215 #endif
00216
00217 // Allocate space for auxiliary uAux so that the lattice and all possible
00218 // directions of ghost Layers fit
00219 const sunindextype s1 = patchToMold.nx, s2 = patchToMold.ny,
00220     s3 = patchToMold.nz;
00221 const sunindextype s_min = std::min(s1, std::min(s2, s3));
00222 patchToMold.uAux.resize(s1 * s2 * s3 / s_min * (s_min + 2 * gLW) * dPD);
00223 patchToMold.uAuxData = &patchToMold.uAux[0];
00224 patchToMold.envelopeLattice = &envelopeLattice;
00225 // Set patch "name" to process number -> only for debugging
00226 // patchToMold.ID=my_prc;
00227 // set flag
00228 patchToMold.statusFlags = FLatticePatchSetUp;
00229 patchToMold.generateTranslocationLookup();
00230 return 0;
00231 }

```

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 229 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 190 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 190 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 190 of file [LatticePatch.h](#).

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

5.9.5.5 du

```
N_Vector LatticePatch::du
```

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

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

5.9.5.6 duData

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

pointer to time-derivative data

Definition at line 221 of file [LatticePatch.h](#).

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

5.9.5.7 duLocal

```
N_Vector LatticePatch::duLocal
```

NVector for saving temporal derivatives of the field data.

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

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

5.9.5.8 dx

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

physical distance between lattice points in x-direction

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

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

5.9.5.9 dy

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

physical distance between lattice points in y-direction

Definition at line 175 of file [LatticePatch.h](#).

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

5.9.5.10 dz

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

physical distance between lattice points in z-direction

Definition at line 177 of file [LatticePatch.h](#).

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

5.9.5.11 envelopeLattice

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

pointer to the enveloping lattice

Definition at line 181 of file [LatticePatch.h](#).

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

5.9.5.12 gCLData

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

pointer to halo data

Definition at line 226 of file [LatticePatch.h](#).

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

5.9.5.13 gCRData

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

pointer to halo data

Definition at line 226 of file [LatticePatch.h](#).

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

5.9.5.14 ghostCellLeft

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

buffer for passing ghost cell data

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

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

5.9.5.15 ghostCellLeftToSend

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

buffer for passing ghost cell data

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

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

5.9.5.16 ghostCellRight

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

buffer for passing ghost cell data

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

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

5.9.5.17 ghostCellRightToSend

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

buffer for passing ghost cell data

Definition at line 195 of file [LatticePatch.h](#).

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

5.9.5.18 ghostCells

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

buffer for passing ghost cell data

Definition at line 195 of file [LatticePatch.h](#).

5.9.5.19 ghostCellsToSend

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

buffer for passing ghost cell data

Definition at line 195 of file [LatticePatch.h](#).

5.9.5.20 ID

```
int LatticePatch::ID
```

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

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

5.9.5.21 lgcTox

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

ghost cell translocation lookup table

Definition at line 199 of file [LatticePatch.h](#).

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

5.9.5.22 lgcToy

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

ghost cell translocation lookup table

Definition at line 199 of file [LatticePatch.h](#).

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

5.9.5.23 lgcToz

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

ghost cell translocation lookup table

Definition at line 199 of file [LatticePatch.h](#).

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

5.9.5.24 LIx

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

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

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

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

5.9.5.25 **Liy**

```
sunindextype LatticePatch::Liy [private]
```

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

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

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

5.9.5.26 **Liz**

```
sunindextype LatticePatch::Liz [private]
```

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

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

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

5.9.5.27 **lx**

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

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

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

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

5.9.5.28 **ly**

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

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

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

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

5.9.5.29 lz

`sunrealtype LatticePatch::lz [private]`

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

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

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

5.9.5.30 nx

`sunindextype LatticePatch::nx [private]`

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

Definition at line 167 of file [LatticePatch.h](#).

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

5.9.5.31 ny

`sunindextype LatticePatch::ny [private]`

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

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

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

5.9.5.32 nz

`sunindextype LatticePatch::nz [private]`

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

Definition at line 171 of file [LatticePatch.h](#).

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

5.9.5.33 rgcTox

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

ghost cell translocation lookup table

Definition at line 199 of file [LatticePatch.h](#).

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

5.9.5.34 rgcToy

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

ghost cell translocation lookup table

Definition at line 199 of file [LatticePatch.h](#).

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

5.9.5.35 rgcToz

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

ghost cell translocation lookup table

Definition at line 199 of file [LatticePatch.h](#).

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

5.9.5.36 statusFlags

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

lattice patch status flags

Definition at line 179 of file [LatticePatch.h](#).

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

5.9.5.37 u

```
N_Vector LatticePatch::u
```

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

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

5.9.5.38 uAux

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

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

Definition at line 183 of file [LatticePatch.h](#).

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

5.9.5.39 uAuxData

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

pointer to auxiliary data vector

Definition at line 223 of file [LatticePatch.h](#).

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

5.9.5.40 uData

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

pointer to field data

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

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

5.9.5.41 uLocal

```
N_Vector LatticePatch::uLocal
```

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

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

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

5.9.5.42 uTox

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

translocation lookup table

Definition at line 186 of file [LatticePatch.h](#).

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

5.9.5.43 uToy

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

translocation lookup table

Definition at line 186 of file [LatticePatch.h](#).

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

5.9.5.44 uToz

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

translocation lookup table

Definition at line 186 of file [LatticePatch.h](#).

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

5.9.5.45 x0

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

origin of the patch in physical space; x-coordinate

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

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

5.9.5.46 xTou

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

translocation lookup table

Definition at line 186 of file [LatticePatch.h](#).

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

5.9.5.47 y0

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

origin of the patch in physical space; y-coordinate

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

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

5.9.5.48 yTou

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

translocation lookup table

Definition at line 186 of file [LatticePatch.h](#).

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

5.9.5.49 z0

`sunrealtype LatticePatch::z0 [private]`

origin of the patch in physical space; z-coordinate

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

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

5.9.5.50 zTou

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

translocation lookup table

Definition at line 186 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 direcory "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.

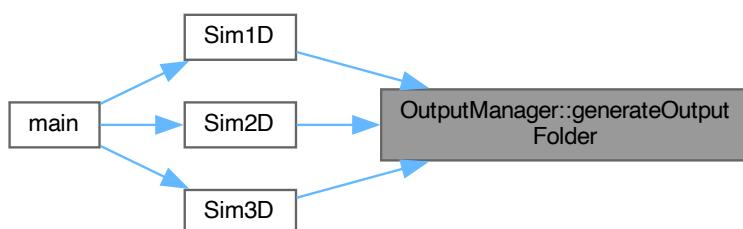
Definition at line 47 of file [Outputters.cpp](#).

```
00047 // Do this only once for the first process
00048 int myPrc = 0;
00049 #if defined(_MPI)
00050 MPI_Comm_rank(MPI_COMM_WORLD, &myPrc);
00051 #endif
00052 if (myPrc == 0) {
00053     if (!fs::is_directory(dir))
00054         fs::create_directory(dir);
00055     if (!fs::is_directory(dir + "/SimResults"))
00056         fs::create_directory(dir + "/SimResults");
00057     if (!fs::is_directory(dir + "/SimResults/" + simCode))
00058         fs::create_directory(dir + "/SimResults/" + simCode);
00059 }
00060 // path variable for the output generation
00061 Path = dir + "/SimResults/" + simCode + "/";
00062
00063 // Logging configurations from main.cpp -> no more necessary
00064 /*
00065 std::ifstream fin("main.cpp");
00066 std::ofstream fout(Path + "config.txt");
00067 std::string line;
00068 int begin = 1000;
00069 for (int i = 1; !fin.eof(); i++) {
00070     getline(fin, line);
00071     if (line.starts_with("      //----- B"))
00072         begin=i;
00073     else
00074         continue;
00075     if (i < begin)
00076         continue;
00077     fout << line << std::endl;
00078     if (line.starts_with("      //----- E"))
00079         break;
00080 }
00081 */
00082
00083 return;
00084 }
```

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

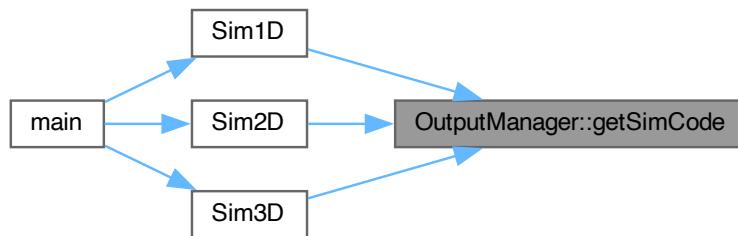
Definition at line 42 of file [Outputters.h](#).

```
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 in the 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 96 of file [Outputters.cpp](#).

```
00097 {
00098     switch(outputStyle) {
00099         case 'c': { // one csv file per process
00100             std::ofstream ofs;
00101             ofs.open(Path + std::to_string(state) + "_"
00102                     + std::to_string(lattice.my_prc) + ".csv");
00103             // Precision of sunrealtypes in significant decimal digits; 15 for IEEE double
00104             ofs << std::setprecision(std::numeric_limits<sunrealtypes>::digits10);
00105
00106             // Walk through each lattice point
00107             const sunindextype totalNP = latticePatch.discreteSize();
00108             for (sunindextype i = 0; i < totalNP * 6; i += 6) {
00109                 // Six columns to contain the field data: Ex,Ey,Ez,Bx,By,Bz
00110                 ofs << latticePatch.uData[i + 0] << "," << latticePatch.uData[i + 1] << ","
00111             }
00112         }
00113     }
00114 }
```

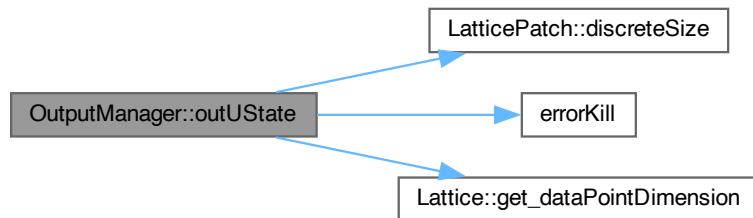
```

00111     « latticePatch.uData[i + 2] « "," « latticePatch.uData[i + 3] « ","
00112     « latticePatch.uData[i + 4] « "," « latticePatch.uData[i + 5]
00113     « std::endl;
00114 }
00115 ofs.close();
00116 break;
00117 }
00118 #if defined(_MPI)
00119     case 'b': { // a single binary file
00120         // Open the output file
00121         MPI_File fh;
00122         const std::string filename = Path+std::to_string(state);
00123         MPI_File_open(lattice.comm,&filename[0],MPI_MODE_WRONLY|MPI_MODE_CREATE,
00124                         MPI_INFO_NULL,&fh);
00125         // number of datapoints in the patch with process offset
00126         const sunindextype count = latticePatch.discreteSize()*
00127             lattice.get_dataPointDimension();
00128         MPI_Offset offset = lattice.my_prc*count*sizeof(MPI_SUNREALTYPE);
00129         // Go to offset in file and write data to it; maximal precision in
00130         // "native" representation
00131         MPI_File_set_view(fh,offset,MPI_SUNREALTYPE,MPI_SUNREALTYPE,"native",
00132                         MPI_INFO_NULL);
00133 #if !defined(WIN32) // MSMPI does not yet support nonblocking collective write
00134         MPI_Request write_request;
00135         MPI_File_iwrite_all(fh,latticePatch.uData,count,MPI_SUNREALTYPE,
00136                         &write_request);
00137         MPI_Wait(&write_request,MPI_STATUS_IGNORE);
00138 #else
00139         MPI_Status status;
00140         MPI_File_write_at_all(fh,offset,latticePatch.uData,count,MPI_SUNREALTYPE,
00141                         &status);
00142 #endif
00143         MPI_File_close(&fh);
00144         break;
00145     }
00146 #endif
00147     default: {
00148         errorKill("No valid output style defined.\n"
00149                 "Choose between csv: one CSV file per process, and "
00150                 "binary: one binary file. In case MPI is not used, you "
00151                 "may only choose csv.");
00152         break;
00153     }
00154 }
00155 }
```

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

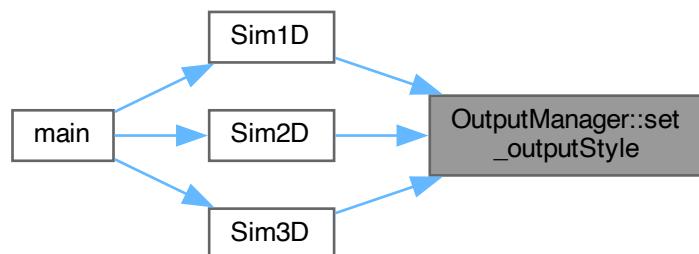
Definition at line 87 of file [Outputters.cpp](#).

```
00087  
00088     outputStyle = _outputStyle;  
00089 }
```

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   const chrono::time_point<chrono::system_clock> now{
00020     chrono::system_clock::now();
00021   const chrono::year_month_day ymd(chrono::floor<chrono::days>(now));
00022   const auto tod = now - chrono::floor<chrono::days>(now);
00023   const chrono::hh_mm_ss hms{tod};
00024
00025   std::stringstream temp;
00026   temp << std::setfill('0') << std::setw(2)
00027     << static_cast<int>(ymd.year() - chrono::years(2000)) << "-"
00028     << std::setfill('0') << std::setw(2)
00029     << static_cast<unsigned>(ymd.month()) << "-"
00030     << std::setfill('0') << std::setw(2)
00031     << static_cast<unsigned>(ymd.day()) << "_"
00032     << std::setfill('0') << std::setw(2) << hms.hours().count()
00033     << "-" << std::setfill('0')
00034     << std::setfill('0') << std::setw(2)
00035     << std::setfill('0') << std::setw(2)
00036     << hms.minutes().count() << "-"
00037     << std::setfill('0') << std::setw(2)
00038     //<< "_" << hms.seconds().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:

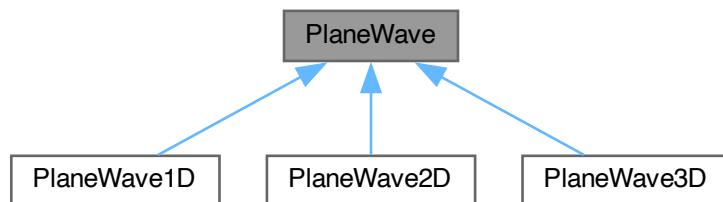
- [src/Outputters.h](#)
- [src/Outputters.cpp](#)

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 **phyi**
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]
```

polarization & amplitude 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](#).

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

5.12.2.2 p

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

amplitde & polarization vector

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

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

5.12.2.3 phi

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

phase shift

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

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

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

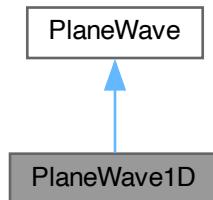
- [src/SimulationFunctions.h](#)

5.13 PlaneWave1D Class Reference

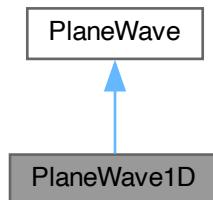
class for plane waves in 1D

```
#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

Protected Attributes inherited from `PlaneWave`

- `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

5.13.1 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 \cdot \vec{x} \cdot \vec{v} \f$ */
00034 // Plane wave definition
00035     const std::array<sunrealtype, 3> E{{
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:

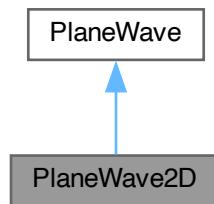
- [src/ICSetters.h](#)
- [src/ICSetters.cpp](#)

5.14 PlaneWave2D Class Reference

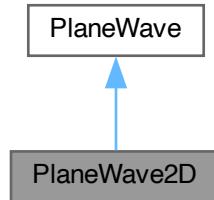
class for plane waves in 2D

```
#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

Protected Attributes inherited from [PlaneWave](#)

- `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

5.14.1 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{{
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:

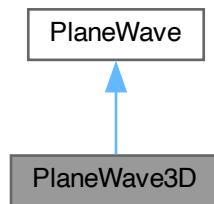
- [src/ICSetters.h](#)
- [src/ICSetters.cpp](#)

5.15 PlaneWave3D Class Reference

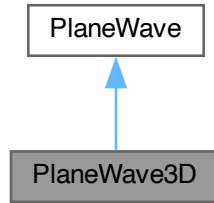
class for plane waves in 3D

```
#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

Protected Attributes inherited from [PlaneWave](#)

- `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

5.15.1 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); /* 1/\lambda */
00107     const sunrealtype kScalarX = (kx * x + ky * y + kz * z) * 2 *
00108         std::numbers::pi; /* 2\pi \vec{k} \cdot \vec{x} */
00109     // Plane wave definition
00110     const std::array<sunrealtype, 3> E{ /* E-field vector */
00111         px * cos(kScalarX - phix), /* E_x */
00112         py * cos(kScalarX - phiy), /* E_y */
00113         pz * cos(kScalarX - phiz)}; /* E_z */
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:

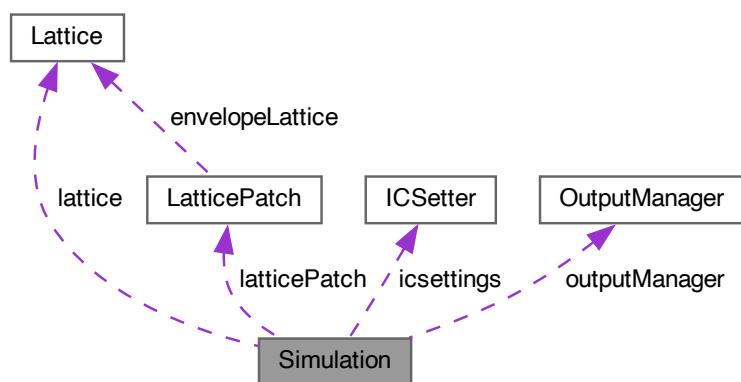
- [src/ICSetters.h](#)
- [src/ICSetters.cpp](#)

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 CVode memory and Sundials context
- 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 `initializeCVODEobject` (const sunrealtype reltol, const sunrealtype abstol)
function to initialize the CVODE 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 CVODE
- 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 * `cvode_mem`
pointer to CVode 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

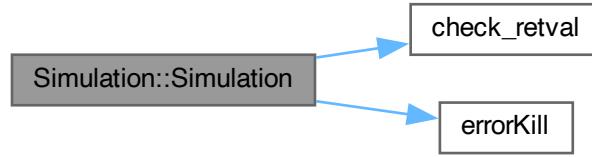
Along with the simulation object, create the cartesian communicator and SUNContext object

Definition at line 14 of file `SimulationClass.cpp`.

```
00015     :
00016     lattice(StencilOrder) {
00017     statusFlags = 0;
00018     t = 0;
00019 #if defined(_MPI)
00020     // Initialize the cartesian communicator
00021     lattice.initializeCommunicator(Nx, Ny, Nz, periodicity);
00022 #endif
00023
00024     // Create the SUNContext object associated with the thread of execution
00025     int retval = 0;
00026     retval = SUNContext_Create(&lattice.comm, &lattice.sunctx);
00027     if (check_retval(&retval, "SUNContext_Create", 1, lattice.my_prc))
00028         errorKill("at SUNContext_Create.");
00029 }
```

References `check_retval()`, `Lattice::comm`, `errorKill()`, `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 CVode memory and Sundials context

Free the CVode solver memory and Sundials context object with the finish of the simulation

Definition at line 33 of file [SimulationClass.cpp](#).

```

00033     {
00034     // Free solver memory
00035     if (statusFlags & CvodeObjectSetUp) {
00036         CVodeFree(&cvode_mem);
00037         SUNNonlinSolFree(NLS);
00038         SUNContext_Free(&lattice.sunctx);
00039     }
00040 }
```

References [cvode_mem](#), [CvodeObjectSetUp](#), [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 174 of file [SimulationClass.cpp](#).

```

00176     {
00177     const sunrealtype dx = latticePatch.getDelta(1);
```

```

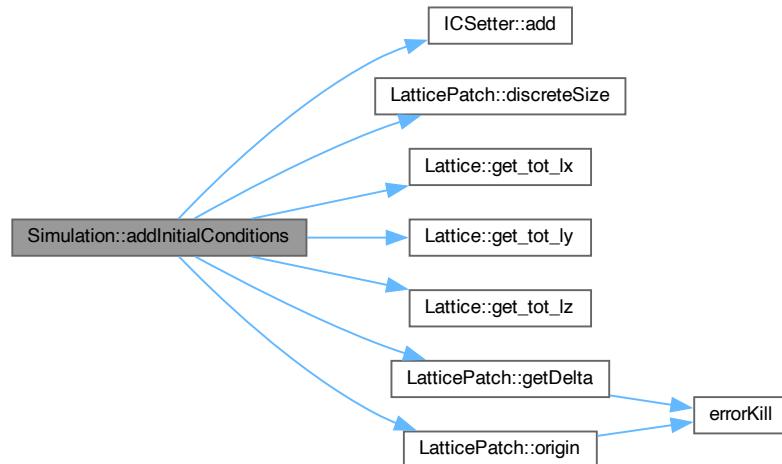
00178 const sunrealtype dy = latticePatch.getDelta(2);
00179 const sunrealtype dz = latticePatch.getDelta(3);
00180 const sunindextype nx = latticePatch.discreteSize(1);
00181 const sunindextype ny = latticePatch.discreteSize(2);
00182 const sunindextype totalNP = latticePatch.discreteSize();
00183 // Correct for demanded displacement, rest as for setInitialConditions
00184 const sunrealtype x0 = latticePatch.origin(1) + xm*lattice.get_tot_lx();
00185 const sunrealtype y0 = latticePatch.origin(2) + ym*lattice.get_tot_ly();
00186 const sunrealtype z0 = latticePatch.origin(3) + zm*lattice.get_tot_lz();
00187 sunindextype px = 0, py = 0, pz = 0;
00188 for (sunindextype i = 0; i < totalNP * 6; i += 6) {
00189     px = (i / 6) % nx;
00190     py = ((i / 6) / nx) % ny;
00191     pz = (((i / 6) / nx) / ny);
00192     icsettings.add(static_cast<sunrealtype>(px) * dx + x0,
00193                     static_cast<sunrealtype>(py) * dy + y0,
00194                     static_cast<sunrealtype>(pz) * dz + z0, &latticePatch.uData[i]);
00195 }
00196 return;
00197 }

```

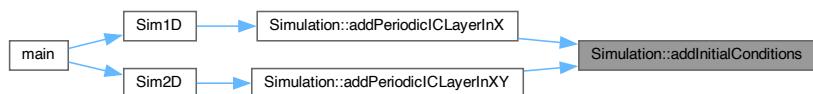
References [ICSetter::add\(\)](#), [LatticePatch::discreteSize\(\)](#), [Lattice::get_tot_lx\(\)](#), [Lattice::get_tot_ly\(\)](#), [Lattice::get_tot_lz\(\)](#), [LatticePatch::getDelta\(\)](#), [icsettings](#), [lattice](#), [latticePatch](#), [LatticePatch::origin\(\)](#), and [LatticePatch::uData](#).

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.

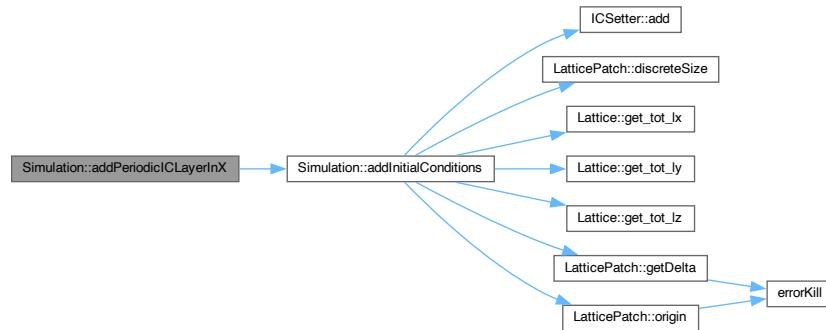
Definition at line 200 of file [SimulationClass.cpp](#).

```
00200     {
00201     addInitialConditions(-1, 0, 0);
00202     addInitialConditions(1, 0, 0);
00203     return;
00204 }
```

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.

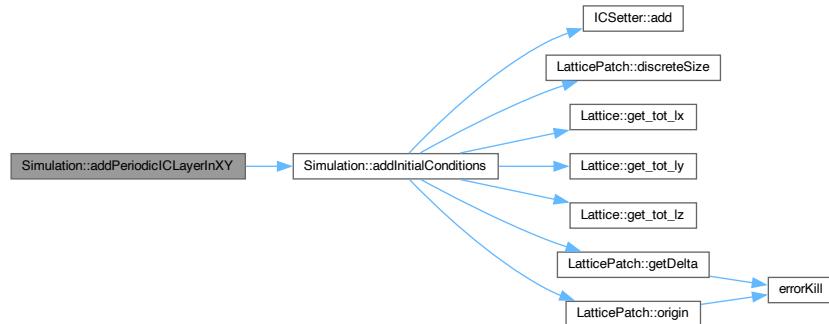
Definition at line 207 of file [SimulationClass.cpp](#).

```
00207
00208     addInitialConditions(-1, -1, 0);
00209     addInitialConditions(-1, 0, 0);
00210     addInitialConditions(-1, 1, 0);
00211     addInitialConditions(0, 1, 0);
00212     addInitialConditions(0, -1, 0);
00213     addInitialConditions(1, -1, 0);
00214     addInitialConditions(1, 0, 0);
00215     addInitialConditions(1, 1, 0);
00216     return;
00217 }
```

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.

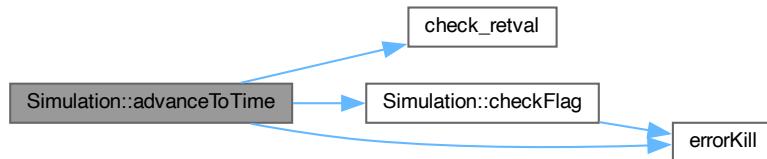
Definition at line 220 of file [SimulationClass.cpp](#).

```
00220
00221     checkFlag(SimulationStarted);
00222     int retval = 0;
00223     retval = CVode(cvode_mem, tEnd, latticePatch.u, &t,
00224                  CV_NORMAL); // CV_NORMAL: internal steps to reach tEnd, then
00225                  // interpolate to return latticePatch.u, return time
00226                  // reached by the solver as t
00227     if (check_retval(&retval, "CVode", 1, lattice.my_prc))
00228         errorKill("at CVode integration.");
00229 }
```

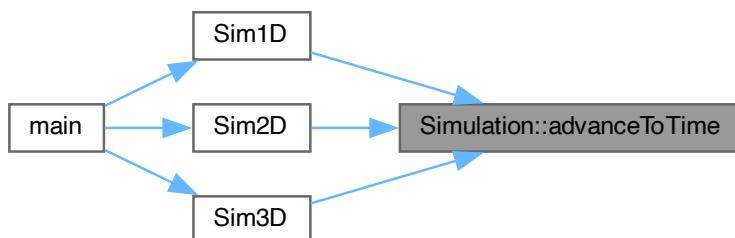
References [check_retval\(\)](#), [checkFlag\(\)](#), [cvode_mem](#), [errorKill\(\)](#), [lattice](#), [latticePatch](#), [Lattice::my_prc](#), [SimulationStarted](#), [t](#), and [LatticePatch::u](#).

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 238 of file [SimulationClass.cpp](#).

```
00238
00239     if (!(statusFlags & flag)) {
00240         std::string errorMessage;
00241         switch (flag) {
00242             case LatticeDiscreteSetUp:
00243                 errorMessage = "The discrete size of the Simulation has not been set up";
00244                 break;
00245             case LatticePhysicalSetUp:
00246                 errorMessage = "The physical size of the Simulation has not been set up";
00247                 break;
00248             case LatticePatchworkSetUp:
00249                 errorMessage = "The patchwork for the Simulation has not been set up";
00250                 break;
00251             case CvodeObjectsetUp:
00252                 errorMessage = "The CVODE object has not been initialized";
00253                 break;
00254             case SimulationStarted:
00255                 errorMessage = "The Simulation has not been started";
00256                 break;
00257             default:
00258                 errorMessage = "Uppss, you've made a non-standard error, sadly I can't "
00259                             "help you there";
00260                 break;
00261         }
00262         errorKill(errorMessage);
00263     }
00264     return;
00265 }
```

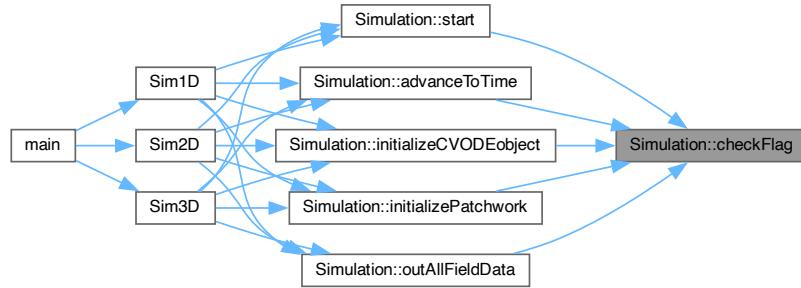
References [CvodeObjectSetUp](#), [errorKill\(\)](#), [LatticeDiscreteSetUp](#), [LatticePatchworkSetUp](#), [LatticePhysicalSetUp](#), [SimulationStarted](#), and [statusFlags](#).

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 268 of file `SimulationClass.cpp`.

```
00268
00269     if ((statusFlags & flag)) {
00270         std::string errorMessage;
00271         switch (flag) {
00272             case LatticeDiscreteSetUp:
00273                 errorMessage =
00274                     "The discrete size of the Simulation has already been set up";
00275                 break;
00276             case LatticePhysicalSetUp:
00277                 errorMessage =
00278                     "The physical size of the Simulation has already been set up";
00279                 break;
00280             case LatticePatchworkSetUp:
00281                 errorMessage = "The patchwork for the Simulation has already been set up";
00282                 break;
00283             case CvodeObjectsetUp:
00284                 errorMessage = "The CVODE object has already been initialized";
00285                 break;
00286             case SimulationStarted:
00287                 errorMessage = "The simulation has already started, some changes are no "
00288                             "longer possible";
00289                 break;
00290             default:
00291                 errorMessage = "Uppss, you've made a non-standard error, sadly I can't "
00292                             "help you there";
00293                 break;
00294         }
00295         errorKill(errorMessage);
00296     }
00297     return;
00298 }
```

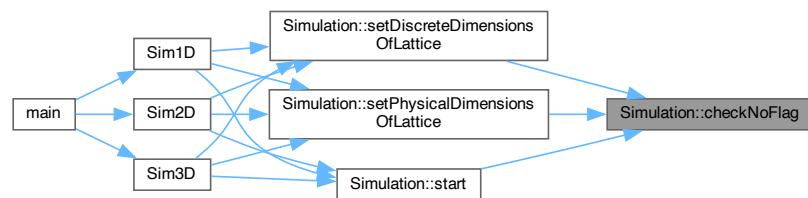
References `CvodeObjectsetUp`, `errorKill()`, `LatticeDiscreteSetUp`, `LatticePatchworkSetUp`, `LatticePhysicalSetUp`, `SimulationStarted`, and `statusFlags`.

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 initializeCVODEobject()

```
void Simulation::initializeCVODEobject (
    const sunrealtype reltol,
    const sunrealtype abstol )
```

function to initialize the CVODE object with all requirements

Configure CVODE.

Definition at line 76 of file [SimulationClass.cpp](#).

```
00077     {
00078     checkFlag(SimulationStarted);
00079
00080     // CVode settings return value
00081     int retval = 0;
00082
00083     // Create CVODE object -- returns a pointer to the cvode memory structure
00084     // with Adams method (Adams-Moulton formula) solver chosen for non-stiff ODE
00085     cvode_mem = CVodeCreate(CV_ADAMS, lattice.sunctx);
00086
00087     // Specify user data and attach it to the main cvode memory block
00088     retval = CVodeSetUserData(
00089         cvode_mem,
00090         &latticePatch); // patch contains the user data as used in CVRhsFn
00091     if (check(retval, "CVodeSetUserData", 1, lattice.my_prc))
00092         errorKill("at CVodeSetUserData.");
00093
00094     // Initialize CVODE solver
00095     retval = CVodeInit(cvode_mem, TimeEvolution::f, 0,
00096                         latticePatch.u); // allocate memory, CVRhsFn f, t_i=0, u
00097                                         // contains the initial values
00098     if (check(retval, "CVodeInit", 1, lattice.my_prc))
00099         errorKill("at CVodeInit.");
```

```

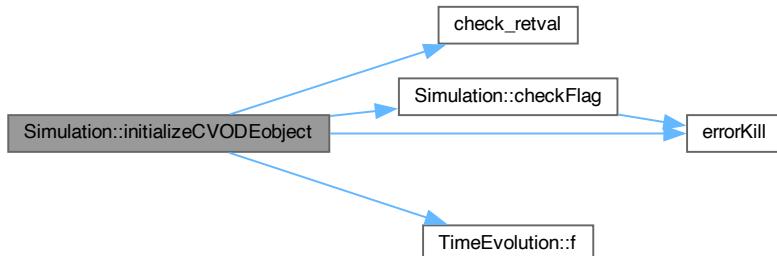
00100 // Create fixed point nonlinear solver object (suitable for non-stiff ODE) and
00101 // attach it to CVode
00102 NLS = SUNNonlinSol_FixedPoint(latticePatch.u, 0, lattice.sunctx);
00103 retval = CVodeSetNonlinearSolver(cvode_mem, NLS);
00104 if (check_retval(&retval, "CVodeSetNonlinearSolver", 1, lattice.my_prc))
00105     errorKill("at CVodeSetNonlinearSolver.");
00106
00107 // Anderson damping factor
00108 retval = SUNNonlinSolSetDamping_FixedPoint(NLS,1);
00109 if (check_retval(&retval, "SUNNonlinSolSetDamping_FixedPoint", 1,
00110     lattice.my_prc))
00111     errorKill("at SUNNonlinSolSetDamping_FixedPoint.");
00112
00113 // Specify integration tolerances -- a scalar relative tolerance and scalar
00114 // absolute tolerance
00115 retval = CVodeSStolerances(cvode_mem, reltol, abstol);
00116 if (check_retval(&retval, "CVodeSStolerances", 1, lattice.my_prc))
00117     errorKill("at CVodeSStolerances.");
00118
00119 // Specify the maximum number of steps to be taken by the solver in its
00120 // attempt to reach the next tout
00121 retval = CVodeSetMaxNumSteps(cvode_mem, 10000);
00122 if (check_retval(&retval, "CVodeSetMaxNumSteps", 1, lattice.my_prc))
00123     errorKill("at CVodeSetMaxNumSteps.");
00124
00125 // maximum number of warnings for too small h
00126 retval = CVodeSetMaxHnilWarns(cvode_mem, 3);
00127 if (check_retval(&retval, "CVodeSetMaxHnilWarns", 1, lattice.my_prc))
00128     errorKill("at CVodeSetMaxHnilWarns.");
00129
00130 statusFlags |= CvodeObjectSetUp;
00131
00132 }

```

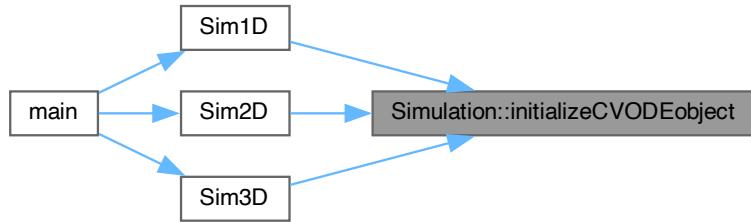
References `check_retval()`, `checkFlag()`, `cvode_mem`, `CvodeObjectSetUp`, `errorKill()`, `TimeEvolution::f()`, `lattice`, `latticePatch`, `Lattice::my_prc`, `NLS`, `SimulationStarted`, `statusFlags`, `Lattice::sunctx`, and `LatticePatch::u`.

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

Here is the call graph for this function:



Here is the caller graph for this function:



5.16.3.8 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.

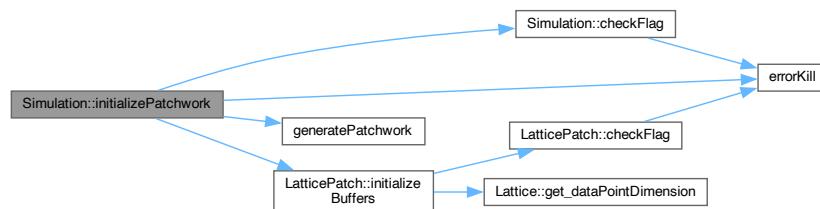
Definition at line 59 of file [SimulationClass.cpp](#).

```
00060      {
00061      checkFlag(LatticeDiscreteSetUp);
00062      checkFlag(LatticePhysicalSetUp);
00063
00064      // Generate the patchwork
00065      #if !defined(_MPI)
00066      if( nx>1 || ny>1 || nz>1 )
00067          errorKill("Splitting the lattice does not work without MPI.");
00068      #endif
00069      generatePatchwork(lattice, latticePatch, nx, ny, nz);
00070      latticePatch.initializeBuffers();
00071
00072      statusFlags |= LatticePatchworkSetUp;
00073 }
```

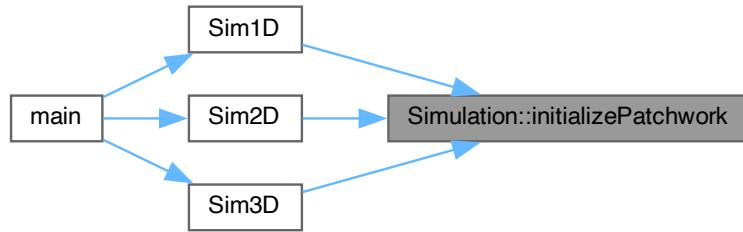
References [checkFlag\(\)](#), [errorKill\(\)](#), [generatePatchwork\(\)](#), [LatticePatch::initializeBuffers\(\)](#), [lattice](#), [LatticeDiscreteSetUp](#), [latticePatch](#), [LatticePatchworkSetUp](#), [LatticePhysicalSetUp](#), and [statusFlags](#).

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

Here is the call graph for this function:



Here is the caller graph for this function:



5.16.3.9 outAllFieldData()

```
void Simulation::outAllFieldData (
    const int & state )
```

function to write field data to disk

Write specified simulation steps to disk.

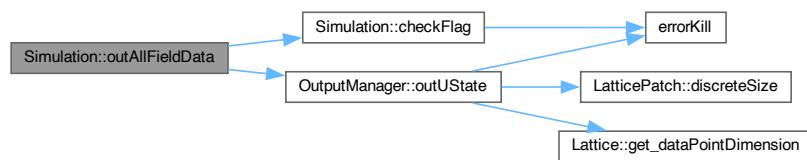
Definition at line 232 of file [SimulationClass.cpp](#).

```
00232
00233     checkFlag(SimulationStarted);
00234     outputManager.outUState(state, lattice, latticePatch);
00235 }
```

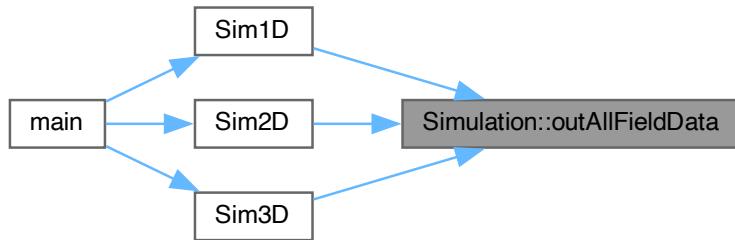
References [checkFlag\(\)](#), [lattice](#), [latticePatch](#), [outputManager](#), [OutputManager::outUState\(\)](#), and [SimulationStarted](#).

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

Here is the call graph for this function:



Here is the caller graph for this function:



5.16.3.10 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.

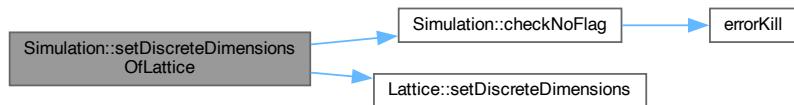
Definition at line 43 of file [SimulationClass.cpp](#).

```
00044
00045     checkNoFlag(LatticePatchworkSetUp);
00046     lattice.setDiscreteDimensions(nx, ny, nz);
00047     statusFlags |= LatticeDiscreteSetUp;
00048 }
```

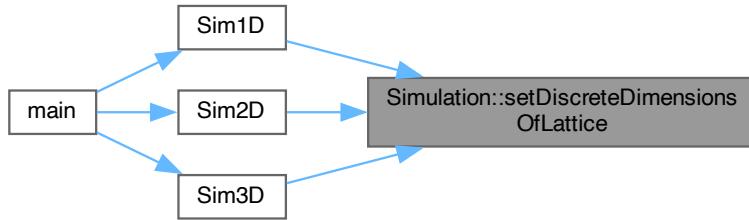
References [checkNoFlag\(\)](#), [lattice](#), [LatticeDiscreteSetUp](#), [LatticePatchworkSetUp](#), [Lattice::setDiscreteDimensions\(\)](#), and [statusFlags](#).

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

Here is the call graph for this function:



Here is the caller graph for this function:



5.16.3.11 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 147 of file [SimulationClass.cpp](#).

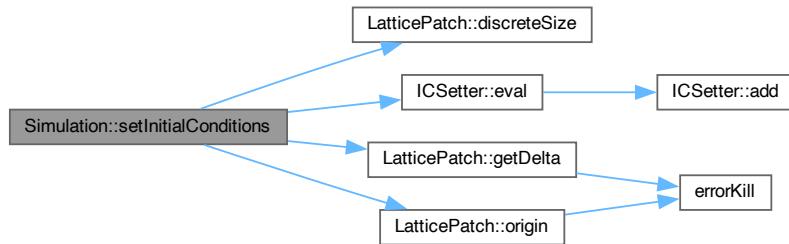
```

00147     {
00148     const sunrealtyp dx = latticePatch.getDelta(1);
00149     const sunrealtyp dy = latticePatch.getDelta(2);
00150     const sunrealtyp dz = latticePatch.getDelta(3);
00151     const sunindextyp nx = latticePatch.discreteSize(1);
00152     const sunindextyp ny = latticePatch.discreteSize(2);
00153     const sunindextyp totalNP = latticePatch.discreteSize();
00154     const sunrealtyp x0 = latticePatch.origin(1);
00155     const sunrealtyp y0 = latticePatch.origin(2);
00156     const sunrealtyp z0 = latticePatch.origin(3);
00157     sunindextyp px = 0, py = 0, pz = 0;
00158     #pragma omp parallel for default(none) \
00159     shared(nx, ny, totalNP, dx, dy, dz, x0, y0, z0) \
00160     firstprivate(px, py, pz) schedule(static)
00161     for (sunindextyp i = 0; i < totalNP * 6; i += 6) {
00162         px = (i / 6) % nx;
00163         py = ((i / 6) / nx) % ny;
00164         pz = ((i / 6) / nx) / ny;
00165         // Call the 'eval' function to fill the lattice points with the field data
00166         icsettings.eval(static_cast<sunrealtyp>(px) * dx + x0,
00167                         static_cast<sunrealtyp>(py) * dy + y0,
00168                         static_cast<sunrealtyp>(pz) * dz + z0, &latticePatch.uData[i]);
00169     }
00170     return;
00171 }
```

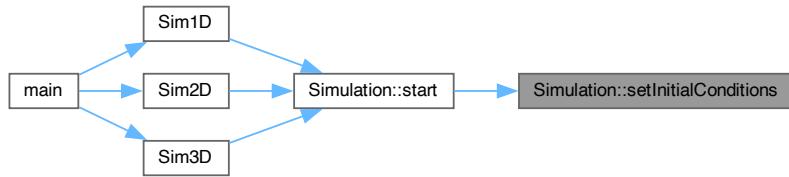
References [LatticePatch::discreteSize\(\)](#), [ICSetter::eval\(\)](#), [LatticePatch::getDelta\(\)](#), [icsettings](#), [latticePatch](#), [LatticePatch::origin\(\)](#), and [LatticePatch::uData](#).

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

Here is the call graph for this function:



Here is the caller graph for this function:



5.16.3.12 `setPhysicalDimensionsOfLattice()`

```

void Simulation::setPhysicalDimensionsOfLattice (
    const sunrealtypelx,
    const sunrealtypely,
    const sunrealtypelz )
  
```

function to set physical dimensions of the lattice

Set the physical dimensions with lengths in micro meters.

Definition at line 51 of file [SimulationClass.cpp](#).

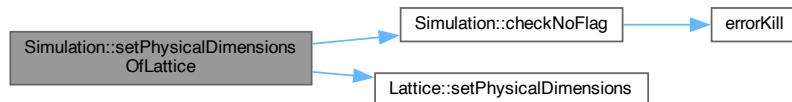
```

00052
00053     checkNoFlag(LatticePatchworkSetUp);
00054     lattice.setPhysicalDimensions(lx, ly, lz);
00055     statusFlags |= LatticePhysicalSetUp;
00056 }
  
```

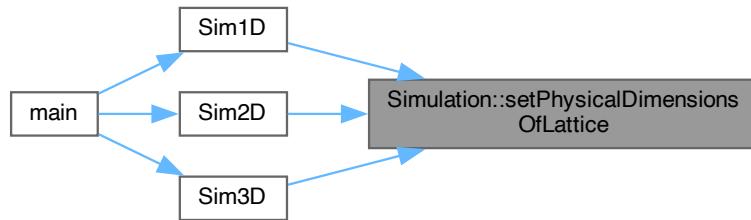
References [checkNoFlag\(\)](#), [lattice](#), [LatticePatchworkSetUp](#), [LatticePhysicalSetUp](#), [Lattice::setPhysicalDimensions\(\)](#), and [statusFlags](#).

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

Here is the call graph for this function:



Here is the caller graph for this function:



5.16.3.13 `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 135 of file `SimulationClass.cpp`.

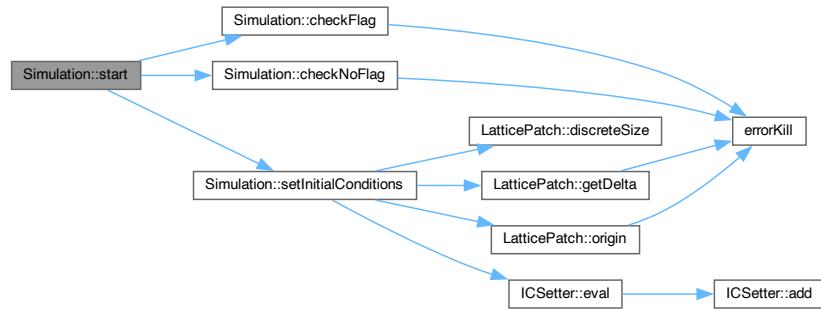
```

00135     {
00136     checkFlag(LatticeDiscreteSetUp);
00137     checkFlag(LatticePhysicalSetUp);
00138     checkFlag(LatticePatchworkSetUp);
00139     checkNoFlag(SimulationStarted);
00140     checkNoFlag(CvodeObjectSetUp);
00141     setInitialConditions();
00142     statusFlags |= SimulationStarted;
00143 }
```

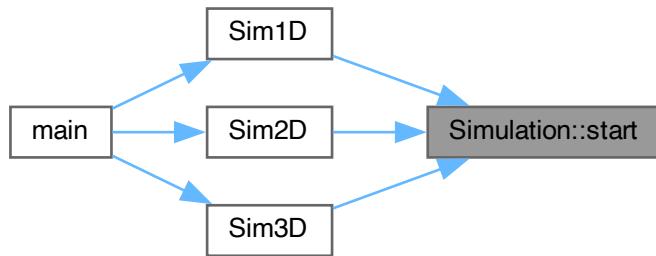
References `checkFlag()`, `checkNoFlag()`, `CvodeObjectSetUp`, `LatticeDiscreteSetUp`, `LatticePatchworkSetUp`, `LatticePhysicalSetUp`, `setInitialConditions()`, `SimulationStarted`, and `statusFlags`.

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\(\)](#), [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. derivatievees with NVector pointers
00018 #if defined(_MPI)
00019 #if !defined(_OPENMP)
00020     sunrealtype *udata = NV_DATA_P(u),
00021             *dudata = NV_DATA_P(udot);
00022 #elif defined(_OPENMP)
00023     sunrealtype *udata = N_VGetArrayPointer_MPIPlusX(u),
00024             *dudata = N_VGetArrayPointer_MPIPlusX(udot);
00025 #endif
00026 #elif defined(_OPENMP)
00027     sunrealtype *udata = NV_DATA_OMP(u),
00028             *dudata = NV_DATA_OMP(udot);
00029 #else
00030     sunrealtype *udata = NV_DATA_S(u),
00031             *dudata = NV_DATA_S(udot);
00032 #endif
00033
00034 // Store original data location of the patch
00035 sunrealtype *originaluData = data->uData,
00036             *originalduData = data->duData;
00037
00038 // Point patch data to arguments of f
00039 data->uData = udata;
00040 data->duData = dudata;
00041
00042 // Time-evolve these arguments (the field data) with specific propagator below
00043 TimeEvolver(data, u, udot, c);
00044
```

```

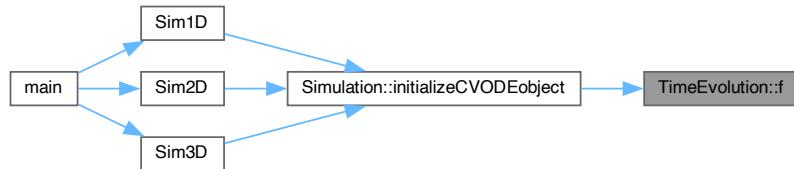
00045 // Refer patch data back to original location
00046 data->uData = originaluData;
00047 data->duData = originalduData;
00048
00049 return (0);
00050 }

```

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

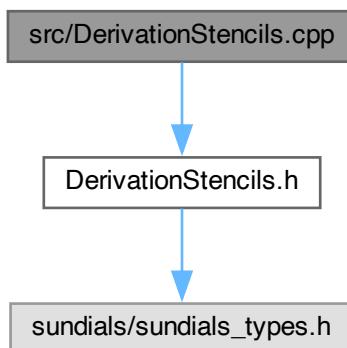
File Documentation

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

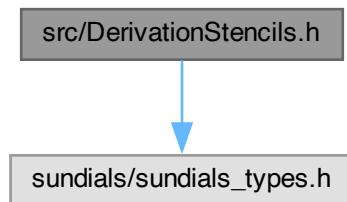
[Go to the documentation of this file.](#)

```
00001 ///////////////////////////////////////////////////////////////////
00002 /// @file DerivationStencils.cpp
00003 /// @brief Empty. All definitions in the header.
00004 ///////////////////////////////////////////////////////////////////
00005 #include "DerivationStencils.h"
```

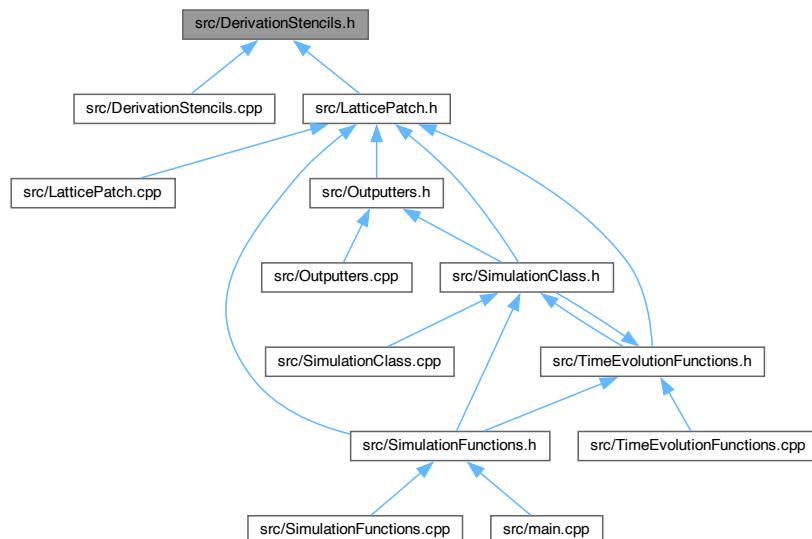
6.4 src/DerivationStencils.h File Reference

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 (
    unrealtype 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]

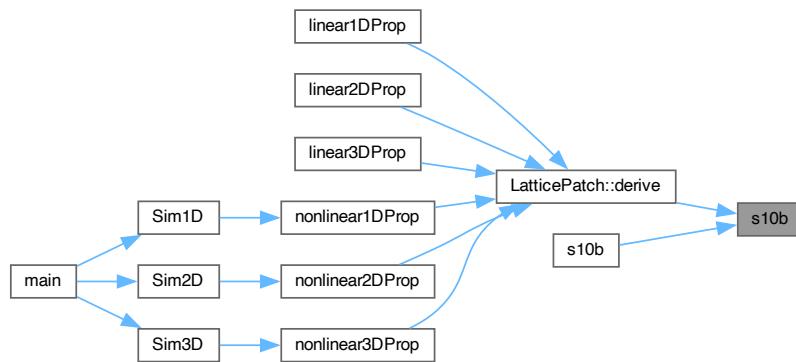
```
sunrealtype s10b (
    unrealtype const * udata,
    const int dPD ) [inline]
```

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 (
    unrealtype 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]

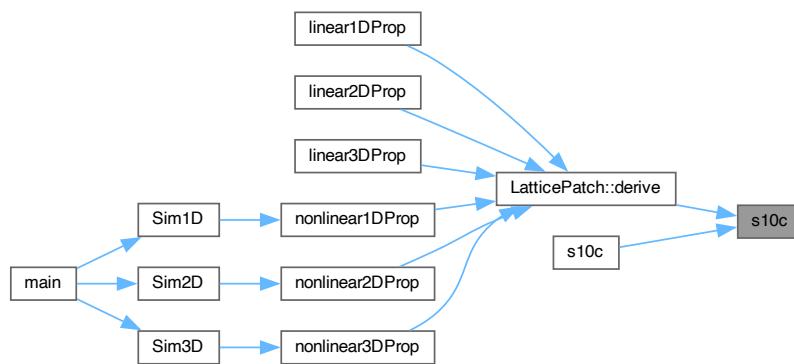
```
sunrealtype s10c (
    unrealtype const * udata,
    const int dPD ) [inline]
```

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 (
    unrealtype 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]

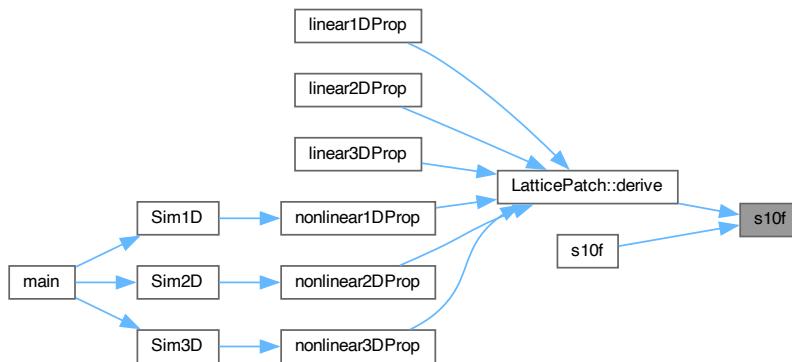
```
sunrealtype s10f (
    unrealtype const * udata,
    const int dPD ) [inline]
```

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 (
    unrealtype 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]

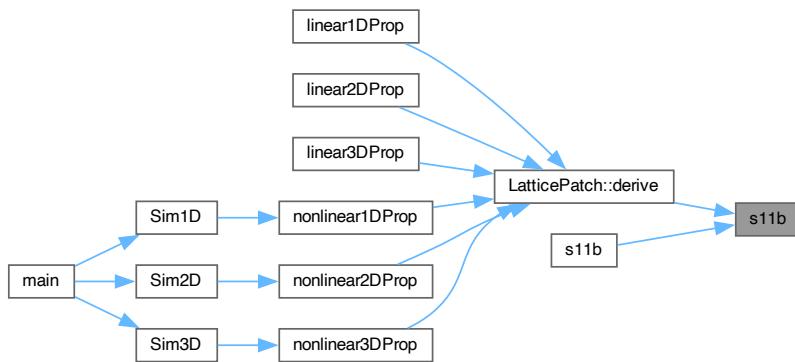
```
sunrealtype s11b (
    unrealtype const * udata,
    const int dPD ) [inline]
```

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 (
    unrealtype 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]

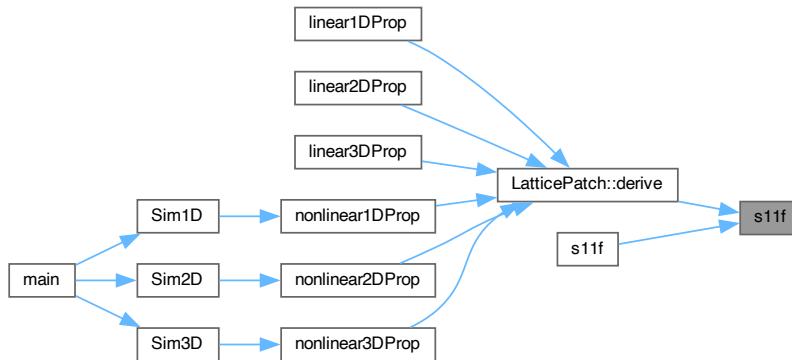
```
sunrealtype s11f (
    unrealtype const * udata,
    const int dPD ) [inline]
```

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 (
    unrealtype 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]

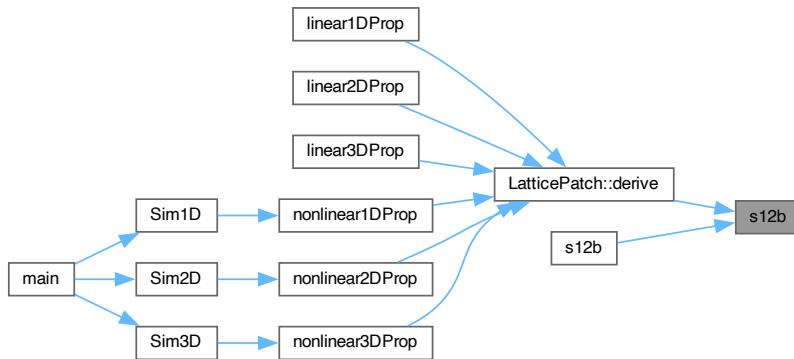
```
sunrealtype s12b (
    unrealtype const * udata,
    const int dPD ) [inline]
```

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 (
    unrealtype 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]

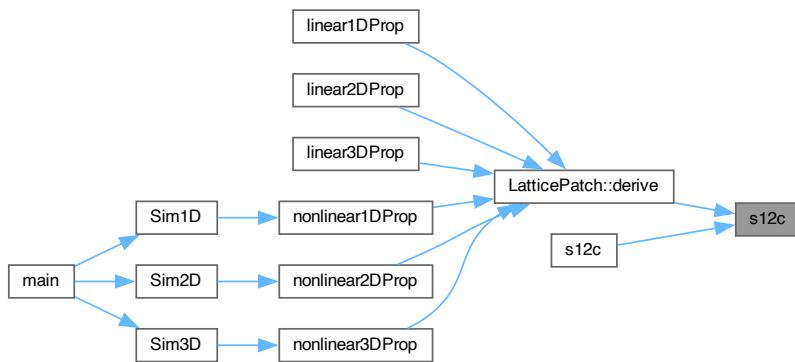
```
sunrealtype s12c (
    unrealtype const * udata,
    const int dPD ) [inline]
```

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 (
    unrealtype 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]

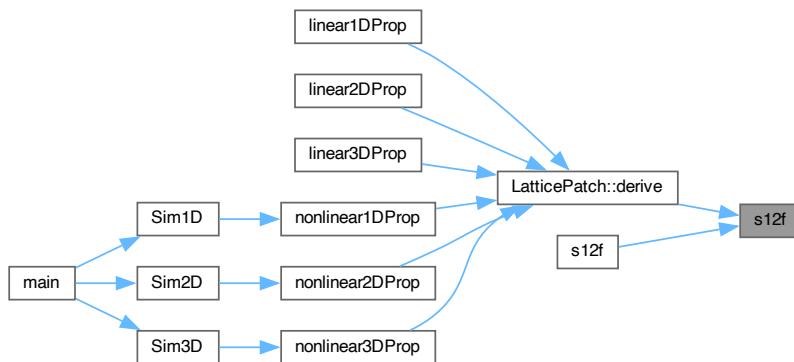
```
sunrealtype s12f (
    unrealtype const * udata,
    const int dPD ) [inline]
```

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 (
    unrealtype 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]

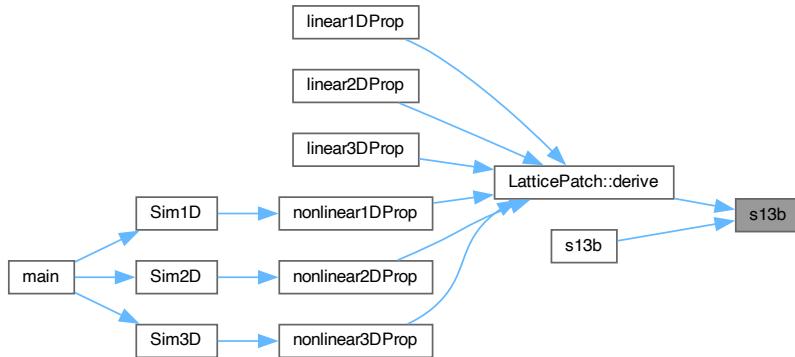
```
sunrealtype s13b (
    unrealtype const * udata,
    const int dPD ) [inline]
```

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 (
    unrealtype 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]

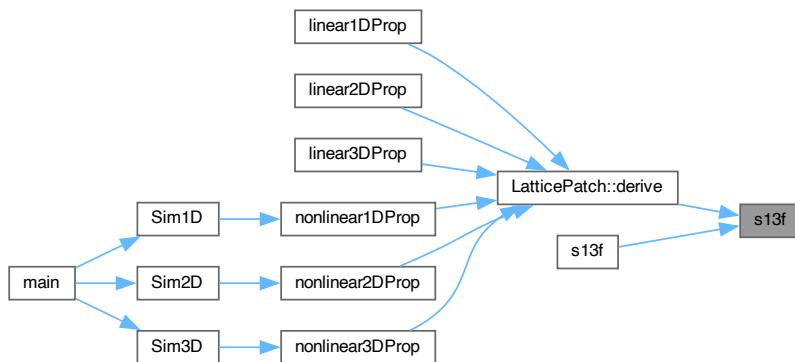
```
sunrealtype s13f (
    unrealtype const * udata,
    const int dPD ) [inline]
```

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 (
    unrealtype 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]

```

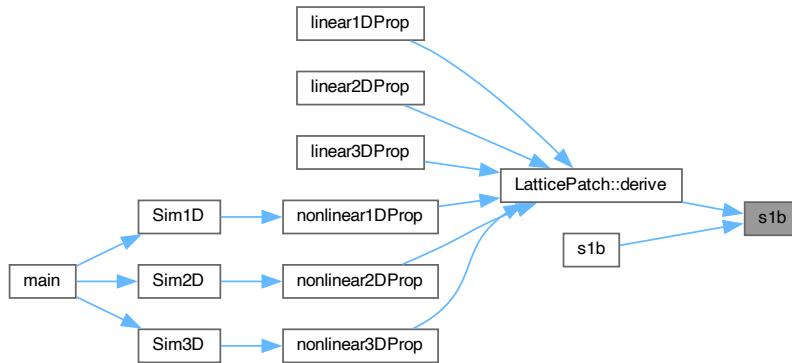
sunrealtype s1b (
    unrealtype const * udata,
    const int dPD ) [inline]

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 (
    unrealtype 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]

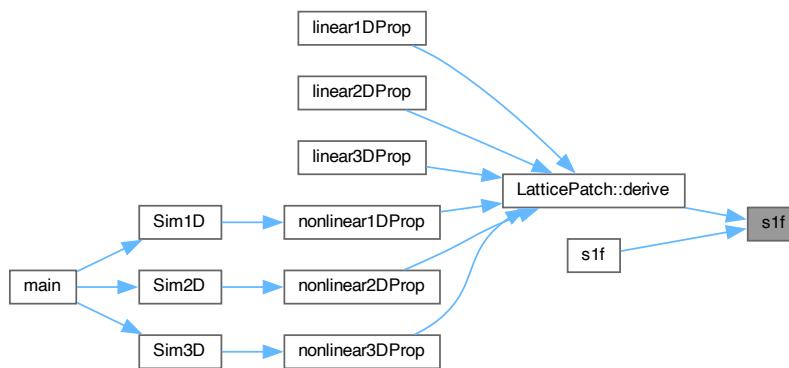
```
sunrealtype s1f (
    unrealtype const * udata,
    const int dPD ) [inline]
```

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 (
    unrealtype 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]

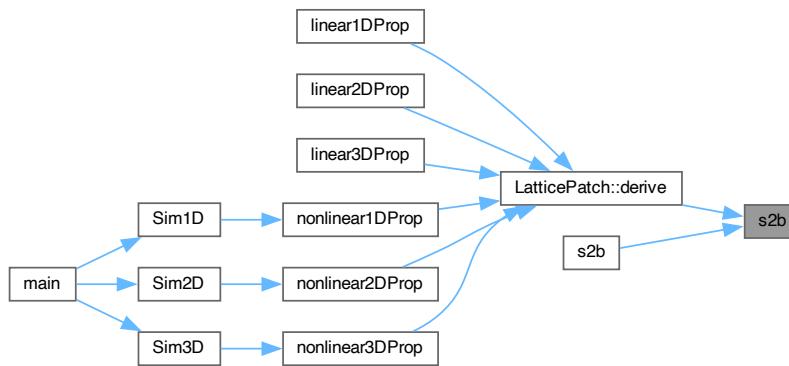
```
sunrealtype s2b (
    unrealtype const * udata,
    const int dPD ) [inline]
```

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 (
    unrealtype 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]

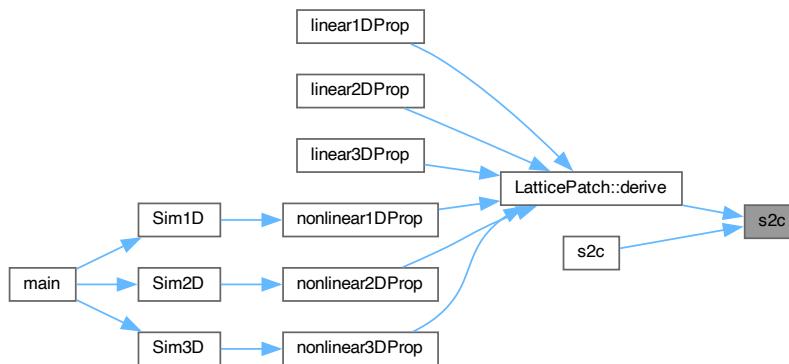
```
sunrealtype s2c (
    unrealtype const * udata,
    const int dPD ) [inline]
```

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 (
    unrealtype 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]

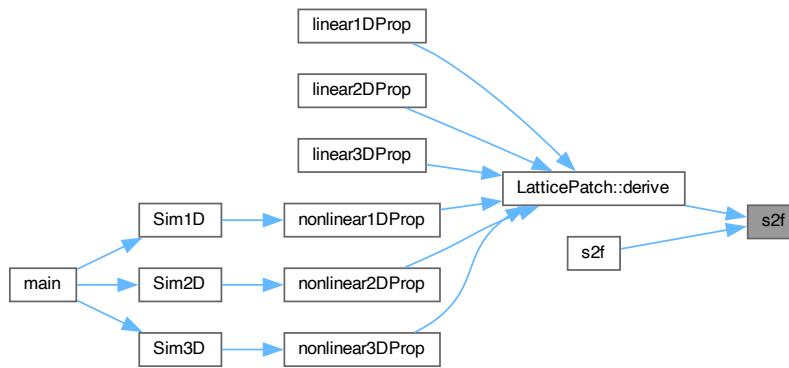
```
sunrealtype s2f (
    unrealtype const * udata,
    const int dPD ) [inline]
```

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 (
    unrealtype 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]

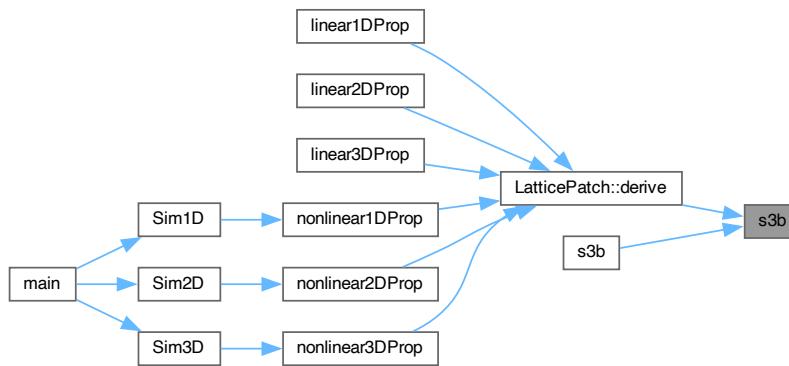
```
sunrealtype s3b (
    unrealtype const * udata,
    const int dPD ) [inline]
```

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 (
    unrealtype 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]

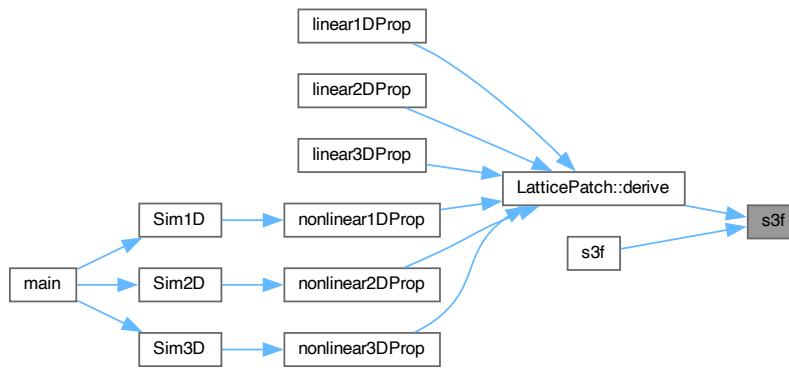
```
sunrealtype s3f (
    unrealtype const * udata,
    const int dPD ) [inline]
```

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 (
    unrealtype 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]

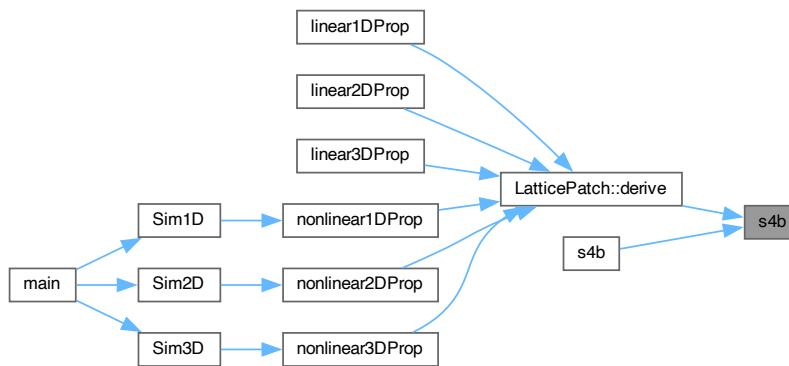
```
sunrealtype s4b (
    unrealtype const * udata,
    const int dPD ) [inline]
```

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 (
    unrealtype 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]

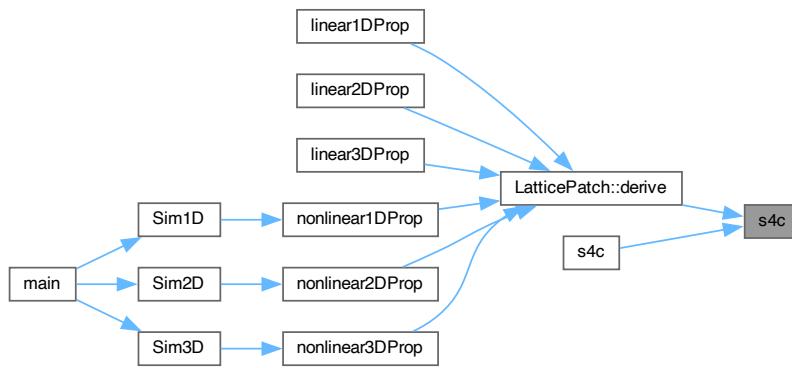
```
sunrealtype s4c (
    unrealtype const * udata,
    const int dPD ) [inline]
```

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 (
    unrealtype 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]

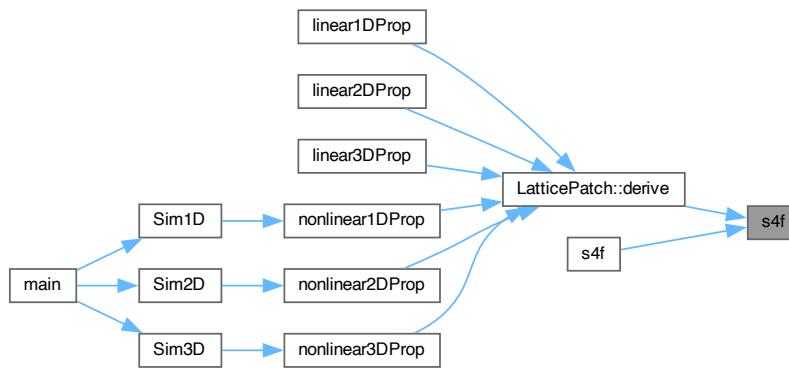
```
sunrealtype s4f (
    unrealtype const * udata,
    const int dPD ) [inline]
```

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 (
    unrealtype 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]

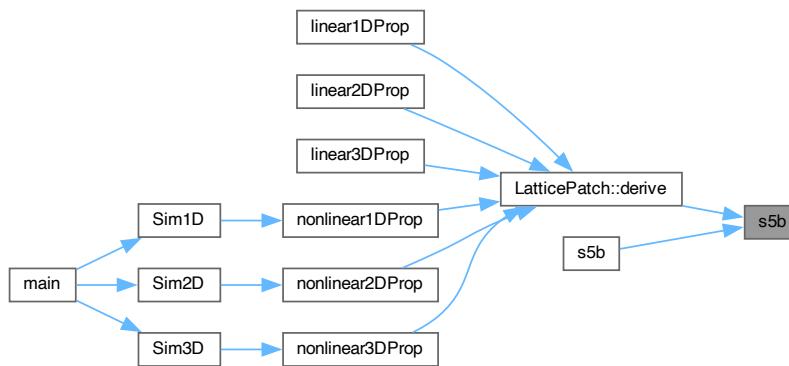
```
sunrealtype s5b (
    unrealtype const * udata,
    const int dPD ) [inline]
```

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 (
    unrealtype 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]

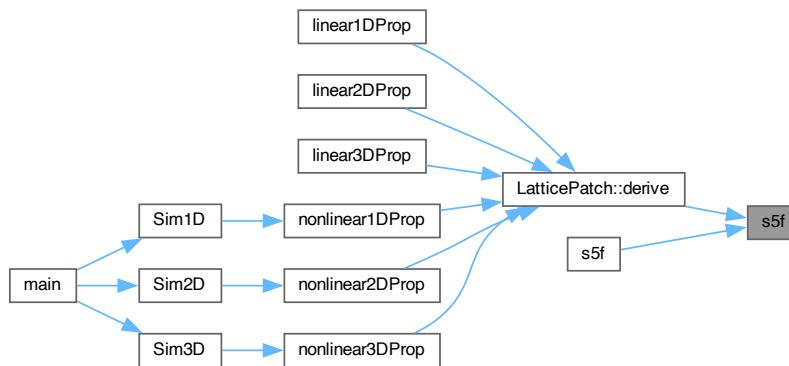
```
sunrealtype s5f (
    unrealtype const * udata,
    const int dPD ) [inline]
```

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 (
    unrealtype 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]

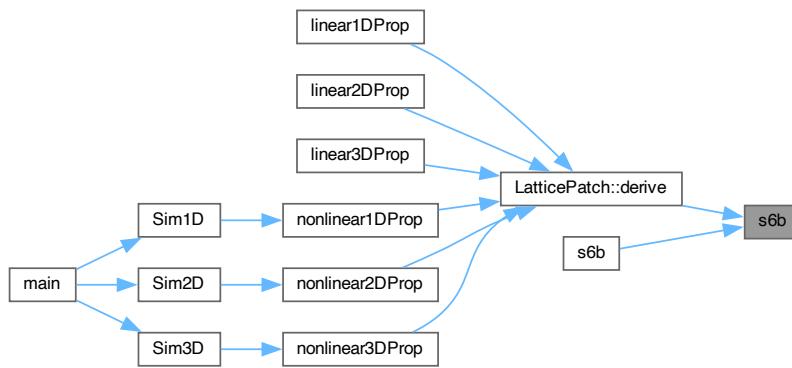
```
sunrealtype s6b (
    unrealtype const * udata,
    const int dPD ) [inline]
```

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 (
    unrealtype 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]

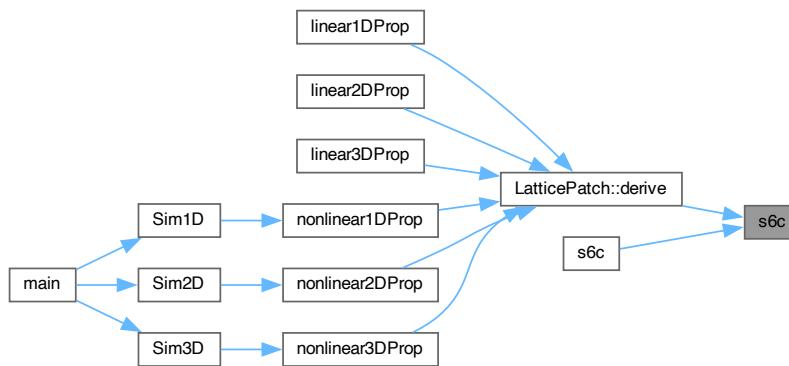
```
sunrealtype s6c (
    unrealtype const * udata,
    const int dPD ) [inline]
```

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 (
    unrealtype 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]

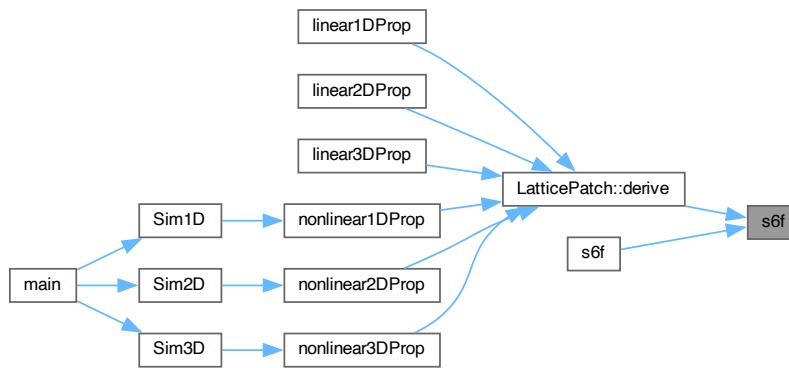
```
sunrealtype s6f (
    unrealtype const * udata,
    const int dPD ) [inline]
```

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 (
    unrealtype 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]

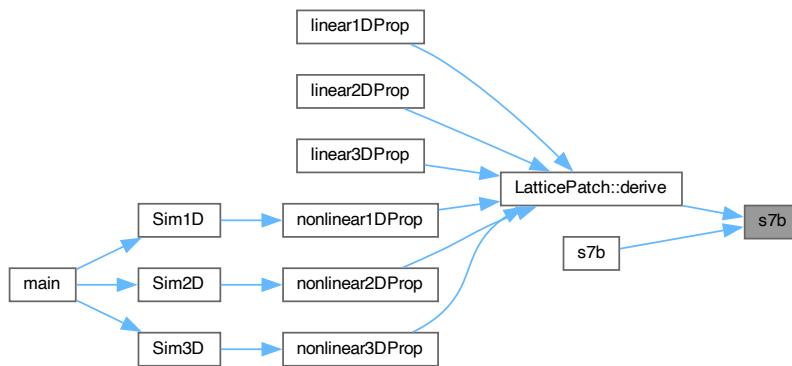
```
sunrealtype s7b (
    unrealtype const * udata,
    const int dPD ) [inline]
```

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 (
    unrealtype 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]

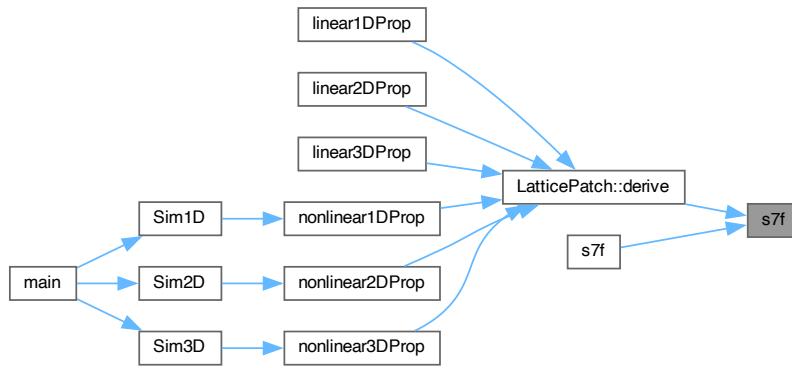
```
sunrealtype s7f (
    unrealtype const * udata,
    const int dPD ) [inline]
```

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 (
    unrealtype 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]

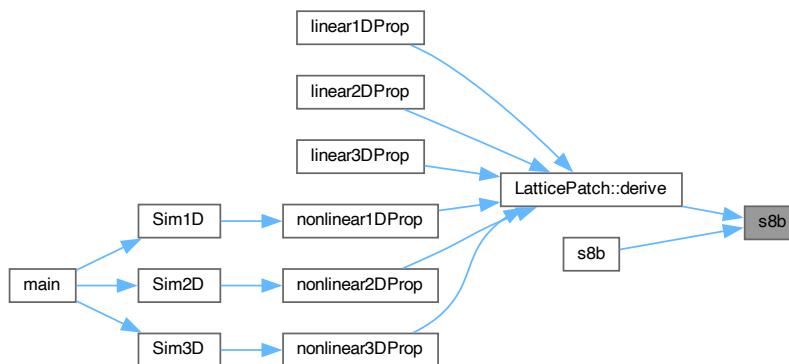
```
sunrealtype s8b (
    unrealtype const * udata,
    const int dPD ) [inline]
```

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 (
    unrealtype 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]

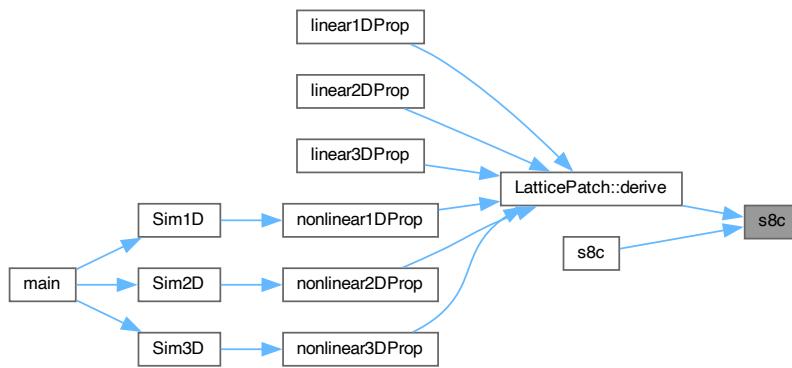
```
sunrealtype s8c (
    unrealtype const * udata,
    const int dPD ) [inline]
```

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 (
    unrealtype 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]

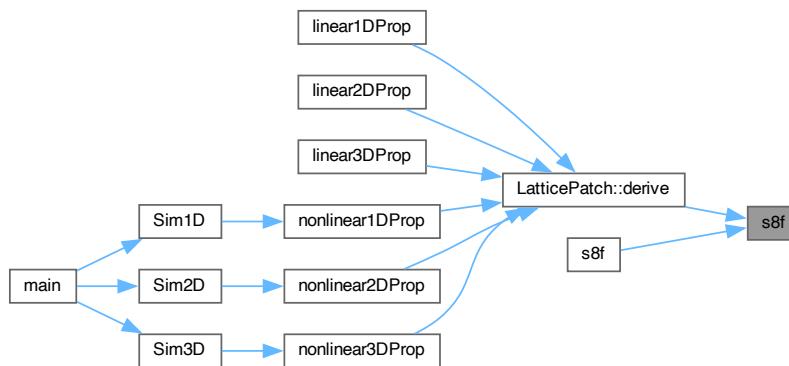
```
sunrealtype s8f (
    unrealtype const * udata,
    const int dPD ) [inline]
```

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 (
    unrealtype 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]

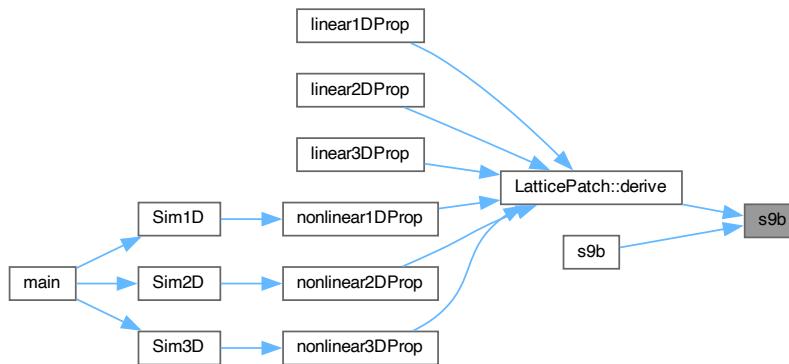
```
sunrealtype s9b (
    unrealtype const * udata,
    const int dPD ) [inline]
```

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 (
    unrealtype 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]

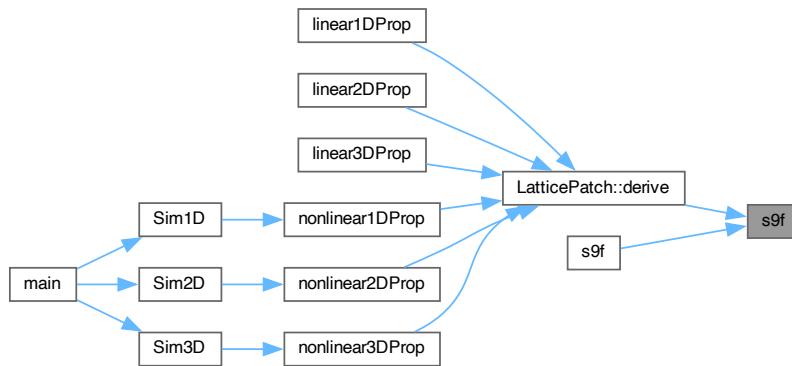
```
sunrealtype s9f (
    unrealtype const * udata,
    const int dPD ) [inline]
```

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 unrealtype */
00009
00010 //////////////////////////////////////////////////////////////////
00011 // Stencils with variable dPD -- data point dimension //
00012 //////////////////////////////////////////////////////////////////
00013
00014 // Downwind (forward) differentiating
00015 #pragma omp declare simd uniform(dPD) notinbranch
00016 inline unrealtype slf(unrealtype 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 unrealtype slb(unrealtype 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 unrealtype s2f(unrealtype const *udata, const int dPD) {
00027     return 1.0 / 2.0 * udata[-2 * dPD] - 2.0 / 1.0 * udata[-1 * dPD] +
00028         1.0 / 2.0 * udata[0] + 2.0 / 1.0 * udata[1 * dPD];
00029 }
```

```

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 slf(sunrealtype const *udata) { return slf(udata, 6); }
00250 inline sunrealtype slb(sunrealtype const *udata) { return slb(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); }

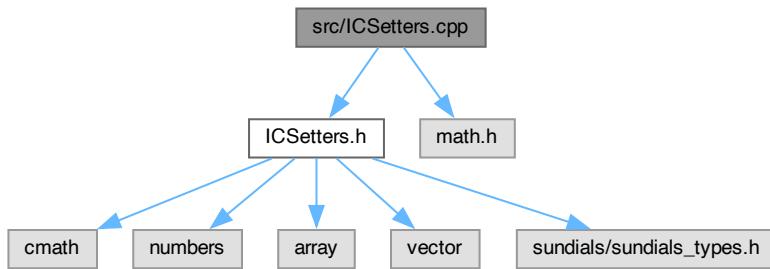
```

```
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); /* 1/\lambda */
00032     const sunrealtype kScalarX = (kx * x + ky * y + kz * z) * 2 *
00033         std::numbers::pi; /* \f$ 2\pi \ \vec{k} \cdot \vec{E} */
00034     // Plane wave definition
00035     const std::array<sunrealtype, 3> E{{
00036         px * cos(kScalarX - phix), /* \f$ E_x */
00037         py * cos(kScalarX - phiy), /* \f$ E_y */
00038         pz * cos(kScalarX - phiz)} }; /* \f$ E_z */
00039
00040     // Put E-field into space
00041     pTo6Space[0] += E[0];
00042     pTo6Space[1] += E[1];
00043     pTo6Space[2] += E[2];
00044     // and B-field
00045     pTo6Space[3] += (ky * E[2] - kz * E[1]) / wavelength;
00046     pTo6Space[4] += (kz * E[0] - kx * E[2]) / wavelength;
00047     pTo6Space[5] += (kx * E[1] - ky * E[0]) / wavelength;
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 */
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_{ix} \f$ */
00061     phiy = phi[1]; /* - phase shift in y-direction \f$ \phi_{iy} \f$ */
00062     phiz = phi[2]; /* - phase shift in z-direction \f$ \phi_{iz} \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); /* 1/\lambda */
00070     const sunrealtype kScalarX = (kx * x + ky * y + kz * z) * 2 *
00071         std::numbers::pi; /* \f$ 2\pi \ \vec{k} \cdot \vec{E} */
00072     // Plane wave definition
00073     const std::array<sunrealtype, 3> E{{
00074         px * cos(kScalarX - phix), /* \f$ E_x */
00075         py * cos(kScalarX - phiy), /* \f$ E_y */
00076         pz * cos(kScalarX - phiz)} }; /* \f$ E_z */
00077
00078     // Put E-field into space
00079     pTo6Space[0] += E[0];
00080     pTo6Space[1] += E[1];
00081     pTo6Space[2] += E[2];
00082     // and B-field
00083     pTo6Space[3] += (ky * E[2] - kz * E[1]) / wavelength;
00084     pTo6Space[4] += (kz * E[0] - kx * E[2]) / wavelength;
00085     pTo6Space[5] += (kx * E[1] - ky * E[0]) / wavelength;
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 */
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_{ix} \f$ */
00098     phiy = phi[1]; /* - phase shift in y-direction \f$ \phi_{iy} \f$ */
00099     phiz = phi[2]; /* - phase shift in z-direction \f$ \phi_{iz} \f$ */
00100 }
00101
00102 /** PlaneWave3D implementation in space */
00103 void PlaneWave3D::addToSpace(sunrealtype x, sunrealtype y, sunrealtype z,
00104     sunrealtype *pTo6Space) const {
00105     const sunrealtype wavelength =
00106         sqrt(kx * kx + ky * ky + kz * kz); /* 1/\lambda */
00107     const sunrealtype kScalarX = (kx * x + ky * y + kz * z) * 2 *
00108         std::numbers::pi; /* \f$ 2\pi \ \vec{k} \cdot \vec{E} */
00109     // Plane wave definition
00110     const std::array<sunrealtype, 3> E{/* E-field vector \f$ \vec{E} */

```

```

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<sunrealtype, 3> k, std::array<sunrealtype, 3> p,
00126                     std::array<sunrealtype, 3> xo, unrealtype phig_,
00127                     std::array<sunrealtype, 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(unrealtype 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 \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<sunrealtype, 3> E{
00157     {px * cos(kScalarX - phix) * envelopeAmp, /* E-field vector */
00158      py * cos(kScalarX - phiy) * envelopeAmp, /* \f$ E_y \f$ */
00159      pz * cos(kScalarX - phiz) * envelopeAmp}; /* \f$ E_z \f$ */
00160 // Put E-field into space
00161 pTo6Space[0] += E[0];
00162 pTo6Space[1] += E[1];
00163 pTo6Space[2] += E[2];
00164 // and B-field
00165 pTo6Space[3] += (ky * E[2] - kz * E[1]) / wavelength;
00166 pTo6Space[4] += (kz * E[0] - kx * E[2]) / wavelength;
00167 pTo6Space[5] += (kx * E[1] - ky * E[0]) / wavelength;
00168 }
00169
00170 /**
00171 Gauss2D construction with */
00172 Gauss2D::Gauss2D(std::array<sunrealtype, 3> dis_,
00173                     std::array<sunrealtype, 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 from 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_; /* - shift from 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(unrealtype 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 sunrealtypet 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 sunrealtypet r = sqrt((x * x + y * y /*+z*z*/) -
00204     zg * zg); // radial distance to propagation axis
00205 // \f$ w(z) = w_0\sqrt{1+(z_g/z_R)^2} \f$
00206 // waist at position z
00207 const sunrealtypet wz = w0 * sqrt(1 + (zg * zg / zr / zr));
00208 // \f$ g(z) = atan(z_g/z_r) \f$
00209 const sunrealtypet gz = atan(zg / zr); // Gouy phase
00210 // \f$ R(z) = z_g*(1+(z_r/z_g)^2) \f$
00211 sunrealtypet 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 sunrealtypet k = 2 * std::numbers::pi / lambda;
00218 // \f$ \Phi_F = kr^2/(2*R(z))+g(z)-kz_g \f$
00219 const sunrealtypet PhF =
00220     -k * r * r / (2 * Rz) + gz - k * zg; // to be inserted into cosine
00221 // \f$ G = \sqrt{w_0/w_z}\mathrm{e}^{-(r/w_z)^2}\mathrm{e}^{-(z-g-Ph0)^2/PhA^2}\cos(\Phi_F) \f$
00222 // CVode is a diva, no chance to remove the square in the second exponential
00223 // -> h too small
00224 const sunrealtypet 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{c} \cdot \vec{a} \f$
00228 // projection components; do like this for CVode convergence -> otherwise
00229 // results in machine error values for non-existant field components if
00230 // axis[0] and axis[1] are given
00231 const sunrealtypet ca =
00232     axis[0]; // x-component of propagation axis which is given as parameter
00233 // no z-component for 2D propagation
00234 const sunrealtypet 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<sunrealtypet, 3> dis_,
00248                     std::array<sunrealtypet, 3> axis_,
00249                     sunrealtypet Amp_,
00250                     // std::array<sunrealtypet, 3> pol_,
00251                     sunrealtypet phip_, sunrealtypet w0_, sunrealtypet zr_,
00252                     sunrealtypet Ph0_, sunrealtypet 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 from TE-mode */
00258     w0 = w0_; //** - taille */
00259     zr = zr_; //** - Rayleigh length */
00260     Ph0 = Ph0_; //** - shift from center */
00261     PhA = PhA_; //** - beam length */
00262     lambda = std::numbers::pi * w0 * w0 / zr;
00263     A1 = Amp * cos(phip);
00264     A2 = Amp * sin(phip);
00265 }
00266
00267 /** Gauss3D implementation in space */
00268 void Gauss3D::addToSpace(sunrealtypet x, sunrealtypet y, sunrealtypet z,
00269                           sunrealtypet *pTo6Space) const {
00270     x -= dis[0];
00271     y -= dis[1];
00272     z -= dis[2];
00273     const sunrealtypet zg = x * axis[0] + y * axis[1] + z * axis[2];
00274     const sunrealtypet r = sqrt((x * x + y * y + z * z) - zg * zg);
00275     const sunrealtypet wz = w0 * sqrt(1 + (zg * zg / zr / zr));
00276     const sunrealtypet gz = atan(zg / zr);
00277     sunrealtypet Rz = nan("0x12345");
00278     if (abs(zg) > 1e-15)
00279         Rz = zg * (1 + (zr * zr / zg / zg));
00280     else
00281         Rz = 1e308;
00282     const sunrealtypet k = 2 * std::numbers::pi / lambda;
00283     const sunrealtypet PhF = -k * r * r / (2 * Rz) + gz - k * zg;
00284     const sunrealtypet G3D = (w0 / wz) * exp(-r * r / wz / wz) *

```

```

00285           exp(-(zg - Ph0) * (zg - Ph0) / PhA / PhA) * cos(PhF);
00286   const sunrealtypes ca = axis[0];
00287   const sunrealtypes 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(sunrealtypes x, sunrealtypes y, sunrealtypes z,
00298                      sunrealtypes *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(sunrealtypes x, sunrealtypes y, sunrealtypes z,
00310                      sunrealtypes *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<sunrealtypes, 3> k,
00327                               std::array<sunrealtypes, 3> p,
00328                               std::array<sunrealtypes, 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<sunrealtypes, 3> k,
00334                               std::array<sunrealtypes, 3> p,
00335                               std::array<sunrealtypes, 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<sunrealtypes, 3> k,
00341                               std::array<sunrealtypes, 3> p,
00342                               std::array<sunrealtypes, 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<sunrealtypes, 3> k,
00348                           std::array<sunrealtypes, 3> p,
00349                           std::array<sunrealtypes, 3> xo, sunrealtypes phig_,
00350                           std::array<sunrealtypes, 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<sunrealtypes, 3> dis_,
00356                           std::array<sunrealtypes, 3> axis_,
00357                           sunrealtypes Amp_, sunrealtypes phip_, sunrealtypes w0_,
00358                           sunrealtypes zr_, sunrealtypes Ph0_, sunrealtypes 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<sunrealtypes, 3> dis_,
00366                           std::array<sunrealtypes, 3> axis_,
00367                           sunrealtypes Amp_, sunrealtypes phip_, sunrealtypes w0_,
00368                           sunrealtypes zr_, sunrealtypes Ph0_, sunrealtypes PhA_)
00369 {
00370   gauss3Ds.emplace_back(
00371     Gauss3D(dis_, axis_, Amp_, phip_, w0_, zr_, Ph0_, PhA_));

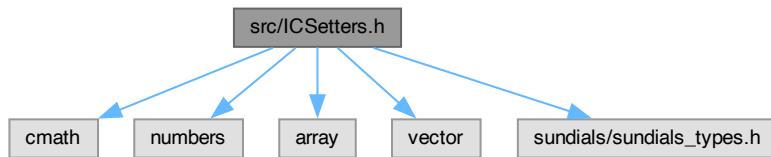
```

```
00372 }
```

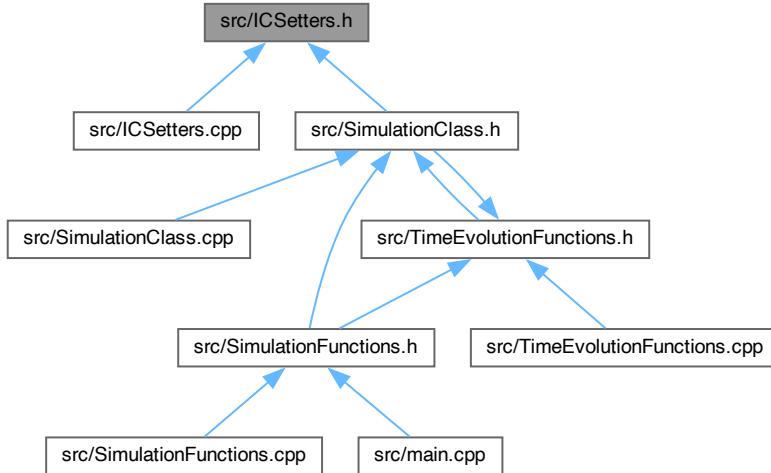
6.8 src/ICSetters.h File Reference

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 sunrealtypes */
00015
00016 /** @brief super-class for plane waves
00017 *
00018 * They are given in the form  $\vec{E} = \vec{E}_0 \cos \left( \vec{k} \cdot \vec{x} - \phi \right)$ 
00019 */
00020 class PlaneWave {
00021 protected:
00022     /// wavenumber  $k_x$ 
00023     sunrealtypes kx;
00024     /// wavenumber  $k_y$ 
00025     sunrealtypes ky;
00026     /// wavenumber  $k_z$ 
00027     sunrealtypes kz;
00028     /// polarization & amplitude in x-direction,  $p_x$ 
00029     sunrealtypes px;
00030     /// polarization & amplitude in y-direction,  $p_y$ 
00031     sunrealtypes py;
00032     /// polarization & amplitude in z-direction,  $p_z$ 
00033     sunrealtypes pz;
00034     /// phase shift in x-direction,  $\phi_x$ 
00035     sunrealtypes phix;
00036     /// phase shift in y-direction,  $\phi_y$ 
00037     sunrealtypes phiy;
00038     /// phase shift in z-direction,  $\phi_z$ 
00039     sunrealtypes 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, unrealtype y, unrealtype z,
00051      unrealtype *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, unrealtype y, unrealtype z,
00063         unrealtype *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(unrealtype x, unrealtype y, unrealtype z,
00075         unrealtype *pTo6Space) const;
00076 };
00077
00078 /** @brief class for Gaussian pulses in 1D
00079 */
00080 * They are given in the form  $\vec{E} = \vec{p} \times \exp(-(\vec{x} - \vec{x}_0)^2 / \Phi_g^2)$ ,  $\cos(\vec{k} \cdot \vec{x})$ 
00081 */
00082
00083 class Gauss1D {
00084 private:
00085     /// wavenumber  $k_x$ 
00086     unrealtype kx;
00087     /// wavenumber  $k_y$ 
00088     unrealtype ky;
00089     /// wavenumber  $k_z$ 
00090     unrealtype kz;
00091     /// polarization & amplitude in x-direction,  $p_x$ 
00092     unrealtype px;
00093     /// polarization & amplitude in y-direction,  $p_y$ 
00094     unrealtype py;
00095     /// polarization & amplitude in z-direction,  $p_z$ 
00096     unrealtype pz;
00097     /// phase shift in x-direction,  $\phi_x$ 
00098     unrealtype phix;
00099     /// phase shift in y-direction,  $\phi_y$ 
00100    unrealtype phiy;
00101    /// phase shift in z-direction,  $\phi_z$ 
00102    unrealtype phiz;
00103    /// center of pulse in x-direction,  $x_0$ 
00104    unrealtype x0;
00105    /// center of pulse in y-direction,  $y_0$ 
00106    unrealtype y0;
00107    /// center of pulse in z-direction,  $z_0$ 
00108    unrealtype z0;
00109    /// pulse width  $\Phi_g$ 
00110    unrealtype 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         unrealtype phig_ = 1.0,
00118         std::array<sunrealtype, 3> phi = {0, 0, 0});
00119     /// function for the actual implementation in space
00120     void addToSpace(unrealtype x, unrealtype y, unrealtype z,
00121         unrealtype *pTo6Space) const;
00122
00123 public:
00124 };
00125
00126 /** @brief class for Gaussian pulses in 2D
00127 */
00128 * They are given in the form
00129 *  $\vec{E} = \vec{\epsilon} \exp(-r/\omega(z))^2 \exp(-((z-g)/\Phi_A)^2) \cos(\frac{k}{r^2}(2R(z)) + g(z) - k)$  with
00130 *  $\vec{\epsilon} = \sqrt{\frac{\omega_0}{\omega(z)}} \exp(-((z-g)/\Phi_A)^2)$ 
00131

```

```

00132 * - propagation direction (subtracted distance to origin) \f$ z_g \f$
00133 * - radial distance to propagation axis \f$ r = \sqrt{\vec{x}^2 - z_g^2} \f$
00134 * - \f$ k = 2\pi / \lambda \f$
00135 * - waist at position z, \f$ \omega(z) = w_0 \ , \ \sqrt{1+(z_g/z_R)^2} \f$
00136 * - Gouy phase \f$ g(z) = \tan^{-1}(z_g/z_r) \f$
00137 * - beam curvature \f$ R(z) = z_g \ , \ (1+(z_r/z_g)^2) \f$
00138 * obtained via the chosen parameters */
00139 class Gauss2D {
00140 private:
00141     /// distance maximum to origin
00142     std::array<surrealtype, 3> dis;
00143     /// normalized propagation axis
00144     std::array<surrealtype, 3> axis;
00145     /// amplitude \f$ A\f$
00146     surrealtype Amp;
00147     /// polarization rotation from TE-mode around propagation direction
00148     // that determines \f$ \vec{\epsilon}\f$ above
00149     surrealtype phip;
00150     /// taille \f$ \omega_0 \f$
00151     surrealtype w0;
00152     /// Rayleigh length \f$ z_R = \pi \omega_0^2 / \lambda \f$
00153     surrealtype zr;
00154     /// center of beam (shift) \f$ \Phi_0 \f$
00155     surrealtype Ph0;
00156     /// length of beam \f$ \Phi_A \f$
00157     surrealtype PhA;
00158     /// amplitude projection on TE-mode
00159     surrealtype A1;
00160     /// amplitude projection on xy-plane
00161     surrealtype A2;
00162     /// wavelength \f$ \lambda \f$
00163     surrealtype lambda;
00164
00165 public:
00166     /// construction with default parameters
00167     Gauss2D(std::array<surrealtype, 3> dis_ = {0, 0, 0},
00168               std::array<surrealtype, 3> axis_ = {1, 0, 0},
00169               surrealtype Amp_ = 1.0,
00170               surrealtype phip_ = 0, surrealtype w0_ = 1e-5,
00171               surrealtype zr_ = 4e-5,
00172               surrealtype Ph0_ = 2e-5, surrealtype PhA_ = 0.45e-5);
00173     /// function for the actual implementation in space
00174     void addToSpace(surrealtype x, surrealtype y, surrealtype z,
00175                     surrealtype *pTo6Space) const;
00176
00177 public:
00178 };
00179
00180 /** @brief class for Gaussian pulses in 3D
00181 *
00182 * They are given in the form
00183 * \f$ \vec{E} = \vec{\epsilon} \exp(-r/\omega(z))^2 \exp(-((z_g-\Phi_0)/\Phi_A)^2) \cos(\frac{k}{r^2}(2R(z)) + g(z) - k) \f$ with
00184 * - propagation direction (subtracted distance to origin) \f$ z_g \f$
00185 * - radial distance to propagation axis \f$ r = \sqrt{\vec{x}^2 - z_g^2} \f$
00186 * - \f$ k = 2\pi / \lambda \f$
00187 * - waist at position z, \f$ \omega(z) = w_0 \ , \ \sqrt{1+(z_g/z_R)^2} \f$
00188 * - Gouy phase \f$ g(z) = \tan^{-1}(z_g/z_r) \f$
00189 * - beam curvature \f$ R(z) = z_g \ , \ (1+(z_r/z_g)^2) \f$
00190 * obtained via the chosen parameters */
00191
00192 class Gauss3D {
00193 private:
00194     /// distance maximum to origin
00195     std::array<surrealtype, 3> dis;
00196     /// normalized propagation axis
00197     std::array<surrealtype, 3> axis;
00198     /// amplitude \f$ A\f$
00199     surrealtype Amp;
00200     /// polarization rotation from TE-mode around propagation direction
00201     // that determines \f$ \vec{\epsilon}\f$ above
00202     surrealtype phip;
00203     // polarization
00204     std::array<surrealtype, 3> pol;
00205     /// taille \f$ \omega_0 \f$
00206     surrealtype w0;
00207     surrealtype zr;
00208     /// Rayleigh length \f$ z_R = \pi \omega_0^2 / \lambda \f$
00209     surrealtype zr;
00210     /// center of beam (shift) \f$ \Phi_0 \f$
00211     surrealtype Ph0;
00212     /// length of beam \f$ \Phi_A \f$
00213     surrealtype PhA;
00214     /// amplitude projection on TE-mode (z-axis)
00215     surrealtype A1;
00216     /// amplitude projection on xy-plane
00217     surrealtype A2;
00218     /// wavelength \f$ \lambda \f$
```

```

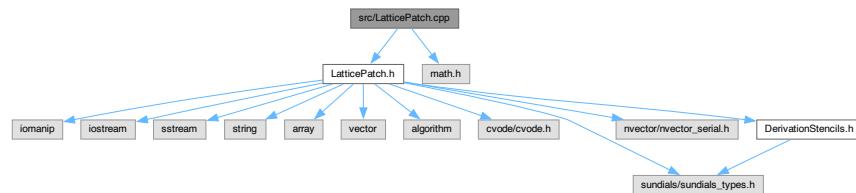
00219     sunrealtype lambda;
00220
00221 public:
00222     /// construction with default parameters
00223     Gauss3D(std::array<sunrealtype, 3> dis_ = {0, 0, 0},
00224             std::array<sunrealtype, 3> axis_ = {1, 0, 0},
00225             unrealtype Amp_ = 1.0,
00226             unrealtype phip_ = 0,
00227             // unrealtype pol_ = {0, 0, 1},
00228             unrealtype w0_ = 1e-5, unrealtype zr_ = 4e-5,
00229             unrealtype Ph0_ = 2e-5, unrealtype PhA_ = 0.45e-5);
00230     /// function for the actual implementation in space
00231     void addToSpace(unrealtype x, unrealtype y, unrealtype z,
00232                     unrealtype *pTo6Space) const;
00233
00234 public:
00235 };
00236
00237 /** @brief ICSsetter class to initialize wave types with default parameters */
00238 class ICSsetter {
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(unrealtype x, unrealtype y, unrealtype z,
00256               unrealtype *pTo6Space);
00257     /// function to fill the lattice space with initial field values
00258     // of all field vector containers
00259     void add(unrealtype x, unrealtype y, unrealtype z,
00260              unrealtype *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                      unrealtype 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                      unrealtype Amp_ = 1.0, unrealtype phip_ = 0,
00283                      unrealtype w0_ = 1e-5, unrealtype zr_ = 4e-5,
00284                      unrealtype Ph0_ = 2e-5, unrealtype 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                      unrealtype Amp_ = 1.0, unrealtype phip_ = 0,
00289                      unrealtype w0_ = 1e-5, unrealtype zr_ = 4e-5,
00290                      unrealtype Ph0_ = 2e-5, unrealtype PhA_ = 0.45e-5);
00291 };
00292

```

6.10 src/LatticePatch.cpp File Reference

Construction 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_retval](#) (void *returnvalue, const char *funcname, int opt, int id)

helper function to check CVode errors

6.10.1 Detailed Description

Construction of the overall envelope lattice and the lattice patches.

Definition in file [LatticePatch.cpp](#).

6.10.2 Function Documentation

6.10.2.1 [check_retval\(\)](#)

```
int check_retval (
    void * returnvalue,
    const char * funcname,
    int opt,
    int id )
```

helper function to check CVode errors

Check function return value. Adapted from CVode 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 979 of file [LatticePatch.cpp](#).

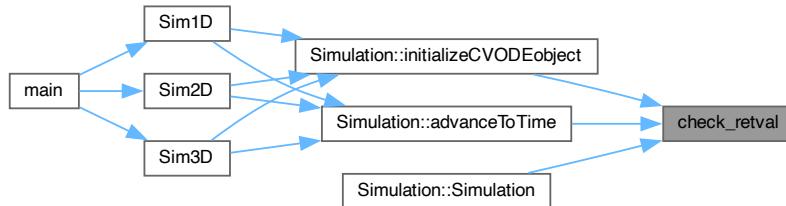
```
00979     int *retval = nullptr;
```

```

00981
00982     /* Check if SUNDIALS function returned NULL pointer - no memory allocated */
00983     if (opt == 0 && returnvalue == nullptr) {
00984         fprintf(stderr,
00985             "\nSUNDIALS_ERROR(process %d): %s() failed - "
00986             "returned NULL pointer\n\n", id, funcname);
00987         return (1);
00988     }
00989
00990     /* Check if retval < 0 */
00991     else if (opt == 1) {
00992         retval = (int *)returnvalue;
00993         char *flagname = CVodeGetReturnFlagName(*retval);
00994         if (*retval < 0) {
00995             fprintf(stderr, "\nSUNDIALS_ERROR(process %d): %s() failed "
00996                 "with retval = %d: %s\n\n",
00997                 id, funcname, *retval, flagname);
00998         }
00999     }
01000 }
01001
01002     /* Check if function returned NULL pointer - no memory allocated */
01003     else if (opt == 2 && returnvalue == nullptr) {
01004         fprintf(stderr,
01005             "\nMEMORY_ERROR(process %d): %s() failed - "
01006             "returned NULL pointer\n\n", id, funcname);
01007     }
01008 }
01009
01010 return (0);
01011 }
```

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

Here is the caller graph for this function:



6.10.2.2 errorKill()

```
void errorKill (
    const std::string & errorMessage )
```

Print a specific error message to stderr.

helper function for error messages

Definition at line 955 of file [LatticePatch.cpp](#).

```

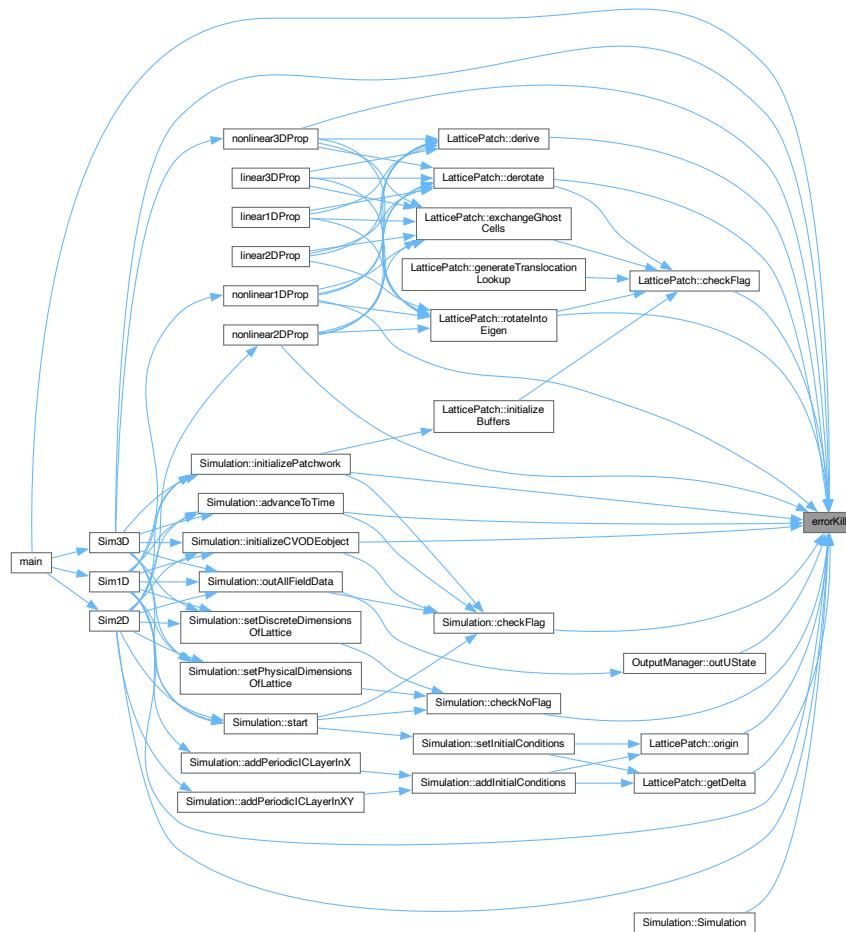
00955
00956     int my_prc=0;
00957 #if defined(_MPI)
00958     MPI_Comm_rank(MPI_COMM_WORLD,&my_prc);
00959 #endif
00960     if (my_prc==0) {
00961         std::cerr << std::endl << "Error: " << errorMessage
```

```

00962     << "\nAborting..." << std::endl;
00963 #if defined(_MPI)
00964     MPI_Abort(MPI_COMM_WORLD, 1);
00965 #else
00966     exit(1);
00967 #endif
00968     return;
00969 }
00970 }
```

Referenced by [Simulation::advanceToTime\(\)](#), [LatticePatch::checkFlag\(\)](#), [Simulation::checkFlag\(\)](#), [Simulation::checkNoFlag\(\)](#), [LatticePatch::derive\(\)](#), [LatticePatch::derotate\(\)](#), [LatticePatch::getDelta\(\)](#), [Simulation::initializeCVODEobject\(\)](#), [Simulation::initializePatchwork\(\)](#), [main\(\)](#), [nonlinear1DProp\(\)](#), [nonlinear2DProp\(\)](#), [nonlinear3DProp\(\)](#), [LatticePatch::origin\(\)](#), [OutputManager::outUState\(\)](#), [LatticePatch::rotateIntoEigen\(\)](#), [Sim1D\(\)](#), [Sim2D\(\)](#), [Sim3D\(\)](#), and [Simulation::Simulation\(\)](#).

Here is the caller graph for this function:



6.10.2.3 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 119 of file [LatticePatch.cpp](#).

```
00121
00122 // Retrieve the ghost layer depth
00123 const int gLW = envelopeLattice.get_ghostLayerWidth();
00124 // Retrieve the data point dimension
00125 const int dPD = envelopeLattice.get_dataPointDimension();
00126 // MPI process/patch
00127 const int my_prc = envelopeLattice.my_prc;
00128 // Determine thicknes of the slice
00129 const sunindextype tot_NOXP = envelopeLattice.get_tot_nx();
00130 const sunindextype tot_NOYP = envelopeLattice.get_tot_ny();
00131 const sunindextype tot_NOZP = envelopeLattice.get_tot_nz();
00132 // position of the patch in the lattice of patches -> process associated to
00133 // position
00134 const sunindextype LIx = my_prc % DLx;
00135 const sunindextype LIy = (my_prc / DLx) % DLy;
00136 const sunindextype LIz = (my_prc / DLx) / DLy;
00137 // Determine the number of points in the patch and first absolute points in
00138 // each dimension
00139 const sunindextype local_NOXP = tot_NOXP / DLx;
00140 const sunindextype local_NOYP = tot_NOYP / DLy;
00141 const sunindextype local_NOZP = tot_NOZP / DLz;
00142 // absolute positions of the first point in each dimension
00143 const sunindextype firstXPoint = local_NOXP * LIx;
00144 const sunindextype firstYPoint = local_NOYP * LIy;
00145 const sunindextype firstZPoint = local_NOZP * LIz;
00146 #if defined(_MPI)
00147 // total number of points in a patch
00148 const sunindextype local_NODP = dPD * local_NOXP * local_NOYP * local_NOZP;
00149 #endif
00150 // Set patch up with above derived quantities
00151 patchToMold.dx = envelopeLattice.get_dx();
00152 patchToMold.dy = envelopeLattice.get_dy();
00153 patchToMold.dz = envelopeLattice.get_dz();
00154 patchToMold.x0 = firstXPoint * patchToMold.dx;
00155 patchToMold.y0 = firstYPoint * patchToMold.dy;
00156 patchToMold.z0 = firstZPoint * patchToMold.dz;
00157 patchToMold.LIx = LIx;
00158 patchToMold.LIy = LIy;
00159 patchToMold.LIz = LIz;
00160 patchToMold.nx = local_NOXP;
00161 patchToMold.ny = local_NOYP;
00162 patchToMold.nz = local_NOZP;
00163 patchToMold.lx = patchToMold.nx * patchToMold.dx;
00164 patchToMold.ly = patchToMold.ny * patchToMold.dy;
00165 patchToMold.lz = patchToMold.nz * patchToMold.dz;
00166
00167 #if defined(_MPI)
00168 #if defined(_OPENMP) // OpenMP and MPI+X NVectors interoperability
00169 // OpenMP NVectors with local patch size
00170 int num_threads = 1;
00171 num_threads = omp_get_max_threads();
00172 patchToMold.uLocal = N_VNew_OpenMP(local_NODP, num_threads,
00173 envelopeLattice.sunctx);
00174 patchToMold.duLocal = N_VNew_OpenMP(local_NODP, num_threads,
00175 envelopeLattice.sunctx);
00176 // MPI+X NVectors containing local OpenMP NVectors
00177 patchToMold.u = N_VMake_MPIPlusX(envelopeLattice.comm, patchToMold.uLocal,
00178 envelopeLattice.sunctx);
00179 patchToMold.du = N_VMake_MPIPlusX(envelopeLattice.comm, patchToMold.duLocal,
00180 envelopeLattice.sunctx);
00181 // Pointers to local vectors
00182 patchToMold.uData = N_VGetArrayPointer_MPIPlusX(patchToMold.u);
00183 patchToMold.duData = N_VGetArrayPointer_MPIPlusX(patchToMold.du);
00184 #else // only MPI
00185 // MPI NVectors with local patch and global lattice size
00186 patchToMold.u =
00187 N_VNew_Parallel(envelopeLattice.comm, local_NODP,
00188 envelopeLattice.get_tot_noDP(), envelopeLattice.sunctx);
00189 patchToMold.du =
00190 N_VNew_Parallel(envelopeLattice.comm, local_NODP,
00191 envelopeLattice.get_tot_noDP(), envelopeLattice.sunctx);
00192 patchToMold.uData = NV_DATA_P(patchToMold.u);
00193 patchToMold.duData = NV_DATA_P(patchToMold.du);
00194 #endif
00195 #elif defined(_OPENMP) // only OpenMP
00196 // OpenMP NVectors with global lattice size
```

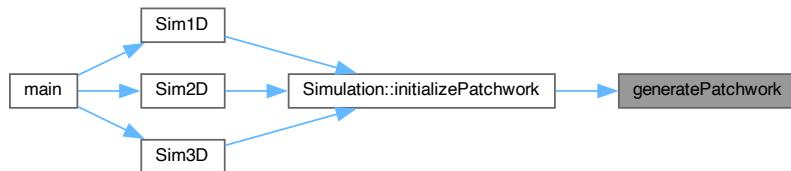
```

00197     int num_threads = 1;
00198     num_threads = omp_get_max_threads();
00199     patchToMold.u =
00200         N_VNew_OpenMP(envelopeLattice.get_tot_noDP(), num_threads,
00201             envelopeLattice.sunctx);
00202     patchToMold.du =
00203         N_VNew_OpenMP(envelopeLattice.get_tot_noDP(), num_threads,
00204             envelopeLattice.sunctx);
00205     patchToMold.uData = NV_DATA_OMP(patchToMold.u);
00206     patchToMold.duData = NV_DATA_OMP(patchToMold.du);
00207 #else // just serial
00208     // Serial NVectors with global lattice size
00209     patchToMold.u =
00210         N_VNew_Serial(envelopeLattice.get_tot_noDP(), envelopeLattice.sunctx);
00211     patchToMold.du =
00212         N_VNew_Serial(envelopeLattice.get_tot_noDP(), envelopeLattice.sunctx);
00213     patchToMold.uData = NV_DATA_S(patchToMold.u);
00214     patchToMold.duData = NV_DATA_S(patchToMold.du);
00215 #endif
00216
00217     // Allocate space for auxiliary uAux so that the lattice and all possible
00218     // directions of ghost Layers fit
00219     const sunindextype s1 = patchToMold.nx, s2 = patchToMold.ny,
00220         s3 = patchToMold.nz;
00221     const sunindextype s_min = std::min(s1, std::min(s2, s3));
00222     patchToMold.uAux.resize(s1 * s2 * s3 / s_min * (s_min + 2 * gLW) * dPD);
00223     patchToMold.uAuxData = &patchToMold.uAux[0];
00224     patchToMold.envelopeLattice = &envelopeLattice;
00225     // Set patch "name" to process number -> only for debugging
00226     // patchToMold.ID=my_prc;
00227     // set flag
00228     patchToMold.statusFlags = FLatticePatchSetUp;
00229     patchToMold.generateTranslocationLookup();
00230     return 0;
00231 }

```

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 #if defined(_MPI)
00015 /// Initialize the cartesian communicator
00016 void Lattice::initializeCommunicator(const int Nx, const int Ny,
00017     const int Nz, const bool per) {
00018     const int dims[3] = {Nz, Ny, Nx};
00019     const int periods[3] = {static_cast<int>(per), static_cast<int>(per),

```

```

00020           static_cast<int>(per));
00021     // Create the cartesian communicator for MPI_COMM_WORLD
00022     MPI_Cart_create(MPI_COMM_WORLD, 3, dims, periods, 1, &comm);
00023     // Set MPI variables of the lattice
00024     MPI_Comm_size(comm, &(n_prc));
00025     MPI_Comm_rank(comm, &(my_prc));
00026     // Associate name to the communicator to identify it -> for debugging and
00027     // nicer error messages
00028     constexpr char lattice_comm_name[] = "Lattice";
00029     MPI_Comm_set_name(comm, lattice_comm_name);
00030   }
00031 #endif
00032
00033 /// Construct the lattice and set the stencil order
00034 Lattice::Lattice(const int StO) : stencilOrder(StO),
00035   ghostLayerWidth(StO/2+1) {
00036   statusFlags = 0;
00037 }
00038
00039 /// Set the number of points in each dimension of the lattice
00040 void Lattice::setDiscreteDimensions(const sunindextype _nx,
00041   const sunindextype _ny, const sunindextype _nz) {
00042   // copy the given data for number of points
00043   tot_nx = _nx;
00044   tot_ny = _ny;
00045   tot_nz = _nz;
00046   // compute the resulting number of points and datapoints
00047   tot_noP = tot_nx * tot_ny * tot_nz;
00048   tot_noDP = dataPointDimension * tot_noP;
00049   // compute the new Delta, the physical resolution
00050   dx = tot_lx / tot_nx;
00051   dy = tot_ly / tot_ny;
00052   dz = tot_lz / tot_nz;
00053 }
00054
00055 /// Set the physical size of the lattice
00056 void Lattice::setPhysicalDimensions(const sunrealtype _lx,
00057   const sunrealtype _ly, const sunrealtype _lz) {
00058   tot_lx = _lx;
00059   tot_ly = _ly;
00060   tot_lz = _lz;
00061   // calculate physical distance between points
00062   dx = tot_lx / tot_nx;
00063   dy = tot_ly / tot_ny;
00064   dz = tot_lz / tot_nz;
00065   statusFlags |= FLatticeDimensionSet;
00066 }
00067
00068 /////////////////////////////////
00069 //// Implementation of LatticePatch component functions /////
00070 /////////////////////////////////
00071
00072 /// Construct the lattice patch
00073 LatticePatch::LatticePatch() {
00074   // set default origin coordinates to (0,0,0)
00075   x0 = y0 = z0 = 0;
00076   // set default position in Lattice-Patchwork to (0,0,0)
00077   LIx = LIy = LIz = 0;
00078   // set default physical length for lattice patch to (0,0,0)
00079   lx = ly = lz = 0;
00080   // set default discrete length for lattice patch to (0,1,1)
00081   /* This is done in this manner as even in 1D simulations require a 1 point
00082    * width */
00083   nx = 0;
00084   ny = nz = 1;
00085
00086   // u is not initialized as it wouldn't make any sense before the dimensions
00087   // are set idem for the enveloping lattice
00088
00089   // set default statusFlags to non set
00090   statusFlags = 0;
00091 }
00092
00093 /// Destruct the patch and thereby destroy the NVectors
00094 LatticePatch::~LatticePatch() {
00095   // Deallocate memory for solution vector
00096   if (statusFlags & FLatticePatchSetUp) {
00097     // Destroy data vectors
00098 #if defined(_MPI)
00099 #if defined(_OPENMP)
00100     N_VDestroy(u);
00101     N_VDestroy(du);
00102     N_VDestroy_OpenMP(uLocal);
00103     N_VDestroy_OpenMP(duLocal);
00104 #else
00105     N_VDestroy_Parallel(u);
00106     N_VDestroy_Parallel(du);

```

```

00107 #endif
00108 #elif defined(_OPENMP)
00109     N_VDestroy_OpenMP(u);
00110     N_VDestroy_OpenMP(du);
00111 #else
00112     N_VDestroy_Serial(u);
00113     N_VDestroy_Serial(du);
00114 #endif
00115 }
00116 }
00117
00118 /// Set up the patchwork
00119 int generatePatchwork(const Lattice &envelopeLattice,
00120     LatticePatch &patchToMold,
00121     const int DLx, const int DLy, const int DLz) {
00122     // Retrieve the ghost layer depth
00123     const int gLW = envelopeLattice.get_ghostLayerWidth();
00124     // Retrieve the data point dimension
00125     const int dPD = envelopeLattice.get_dataPointDimension();
00126     // MPI process/patch
00127     const int my_prc = envelopeLattice.my_prc;
00128     // Determine thickness of the slice
00129     const sunindextype tot_NOXP = envelopeLattice.get_tot_nx();
00130     const sunindextype tot_NOYP = envelopeLattice.get_tot_ny();
00131     const sunindextype tot_NOZP = envelopeLattice.get_tot_nz();
00132     // position of the patch in the lattice of patches -> process associated to
00133     // position
00134     const sunindextype LIx = my_prc % DLx;
00135     const sunindextype LIy = (my_prc / DLx) % DLy;
00136     const sunindextype LIz = (my_prc / DLx) / DLy;
00137     // Determine the number of points in the patch and first absolute points in
00138     // each dimension
00139     const sunindextype local_NOXP = tot_NOXP / DLx;
00140     const sunindextype local_NOYP = tot_NOYP / DLy;
00141     const sunindextype local_NOZP = tot_NOZP / DLz;
00142     // absolute positions of the first point in each dimension
00143     const sunindextype firstXPoint = local_NOXP * LIx;
00144     const sunindextype firstYPoint = local_NOYP * LIy;
00145     const sunindextype firstZPoint = local_NOZP * LIz;
00146 #if defined(_MPI)
00147     // total number of points in a patch
00148     const sunindextype local_NODP = dPD * local_NOXP * local_NOYP * local_NOZP;
00149 #endif
00150     // Set patch up with above derived quantities
00151     patchToMold.dx = envelopeLattice.get_dx();
00152     patchToMold.dy = envelopeLattice.get_dy();
00153     patchToMold.dz = envelopeLattice.get_dz();
00154     patchToMold.x0 = firstXPoint * patchToMold.dx;
00155     patchToMold.y0 = firstYPoint * patchToMold.dy;
00156     patchToMold.z0 = firstZPoint * patchToMold.dz;
00157     patchToMold.LIx = LIx;
00158     patchToMold.LIy = LIy;
00159     patchToMold.LIz = LIz;
00160     patchToMold.nx = local_NOXP;
00161     patchToMold.ny = local_NOYP;
00162     patchToMold.nz = local_NOZP;
00163     patchToMold.lx = patchToMold.nx * patchToMold.dx;
00164     patchToMold.ly = patchToMold.ny * patchToMold.dy;
00165     patchToMold.lz = patchToMold.nz * patchToMold.dz;
00166
00167 #if defined(_MPI)
00168 #if defined(_OPENMP) // OpenMP and MPI+X NVectors interoperability
00169     // OpenMP NVectors with local patch size
00170     int num_threads = 1;
00171     num_threads = omp_get_max_threads();
00172     patchToMold.uLocal = N_VNew_OpenMP(local_NODP, num_threads,
00173         envelopeLattice.sunctx);
00174     patchToMold.duLocal = N_VNew_OpenMP(local_NODP, num_threads,
00175         envelopeLattice.sunctx);
00176     // MPI+X NVectors containing local OpenMP NVectors
00177     patchToMold.u = N_VMake_MPIPlusX(envelopeLattice.comm, patchToMold.uLocal,
00178         envelopeLattice.sunctx);
00179     patchToMold.du = N_VMake_MPIPlusX(envelopeLattice.comm, patchToMold.duLocal,
00180         envelopeLattice.sunctx);
00181     // Pointers to local vectors
00182     patchToMold.uData = N_VGetArrayPointer_MPIPlusX(patchToMold.u);
00183     patchToMold.duData = N_VGetArrayPointer_MPIPlusX(patchToMold.du);
00184 #else // only MPI
00185     // MPI NVectors with local patch and global lattice size
00186     patchToMold.u =
00187         N_VNew_Parallel(envelopeLattice.comm, local_NODP,
00188             envelopeLattice.get_tot_noDP(), envelopeLattice.sunctx);
00189     patchToMold.du =
00190         N_VNew_Parallel(envelopeLattice.comm, local_NODP,
00191             envelopeLattice.get_tot_noDP(), envelopeLattice.sunctx);
00192     patchToMold.uData = NV_DATA_P(patchToMold.u);
00193     patchToMold.duData = NV_DATA_P(patchToMold.du);

```

```

00194 #endif
00195 #elif defined(_OPENMP) // only OpenMP
00196 // OpenMP NVectors with global lattice size
00197 int num_threads = 1;
00198 num_threads = omp_get_max_threads();
00199 patchToMold.u =
00200     N_VNew_OpenMP(envelopeLattice.get_tot_noDP(), num_threads,
00201         envelopeLattice.sunctx);
00202 patchToMold.du =
00203     N_VNew_OpenMP(envelopeLattice.get_tot_noDP(), num_threads,
00204         envelopeLattice.sunctx);
00205 patchToMold.uData = NV_DATA_OMP(patchToMold.u);
00206 patchToMold.duData = NV_DATA_OMP(patchToMold.du);
00207 #else // just serial
00208 // Serial NVectors with global lattice size
00209 patchToMold.u =
00210     N_VNew_Serial(envelopeLattice.get_tot_noDP(), envelopeLattice.sunctx);
00211 patchToMold.du =
00212     N_VNew_Serial(envelopeLattice.get_tot_noDP(), envelopeLattice.sunctx);
00213 patchToMold.uData = NV_DATA_S(patchToMold.u);
00214 patchToMold.duData = NV_DATA_S(patchToMold.du);
00215 #endif
00216
00217 // Allocate space for auxiliary uAux so that the lattice and all possible
00218 // directions of ghost Layers fit
00219 const sunindextype s1 = patchToMold.nx, s2 = patchToMold.ny,
00220     s3 = patchToMold.nz;
00221 const sunindextype s_min = std::min(s1, std::min(s2, s3));
00222 patchToMold.uAux.resize(s1 * s2 * s3 / s_min * (s_min + 2 * gLW) * dPD);
00223 patchToMold.uAuxData = &patchToMold.uAux[0];
00224 patchToMold.envelopeLattice = &envelopeLattice;
00225 // Set patch "name" to process number -> only for debugging
00226 // patchToMold.ID=my_prc;
00227 // set flag
00228 patchToMold.statusFlags = FLatticePatchSetUp;
00229 patchToMold.generateTranslocationLookup();
00230 return 0;
00231 }
00232
00233 /// Return the discrete size of the patch: number of lattice patch points in
00234 /// specified dimension
00235 sunindextype LatticePatch::discreteSize(int dir) const {
00236     switch (dir) {
00237     case 0:
00238         return nx * ny * nz;
00239     case 1:
00240         return nx;
00241     case 2:
00242         return ny;
00243     case 3:
00244         return nz;
00245     // case 4: return uAux.size(); // for debugging
00246     default:
00247         return -1;
00248     }
00249 }
00250
00251 /// Return the physical origin of the patch in a dimension
00252 sunrealtype LatticePatch::origin(const int dir) const {
00253     switch (dir) {
00254     case 1:
00255         return x0;
00256     case 2:
00257         return y0;
00258     case 3:
00259         return z0;
00260     default:
00261         errorKill("LatticePatch::origin function called with wrong dir parameter");
00262         return -1;
00263     }
00264 }
00265
00266 /// Return the distance between points in the patch in a dimension
00267 sunrealtype LatticePatch::getDelta(const int dir) const {
00268     switch (dir) {
00269     case 1:
00270         return dx;
00271     case 2:
00272         return dy;
00273     case 3:
00274         return dz;
00275     default:
00276         errorKill(
00277             "LatticePatch::getDelta function called with wrong dir parameter");
00278         return -1;
00279     }
00280 }

```

```

00281  /** In order to avoid cache misses:
00282   * create vectors to translate u vector into space coordinates and vice versa
00283   * and same for left and right ghost layers to space */
00284 void LatticePatch::generateTranslocationLookup() {
00285   // Check that the lattice has been set up
00286   checkFlag(FLatticeDimensionSet);
00287   // lengths for auxilliary layers, including ghost layers
00288   const int gLW = envelopeLattice->get_ghostLayerWidth();
00289   const sunindextype mx = nx + 2 * gLW;
00290   const sunindextype my = ny + 2 * gLW;
00291   const sunindextype mz = nz + 2 * gLW;
00292   // sizes for lookup vectors
00293   const sunindextype totalNP = nx * ny * nz;
00294   const sunindextype haloXSize = mx * ny * nz;
00295   const sunindextype haloYSIZE = nx * my * nz;
00296   const sunindextype haloZSize = nx * ny * mz;
00297   // generate u->uAux
00298   uTox.resize(totalNP);
00299   uToy.resize(totalNP);
00300   uToz.resize(totalNP);
00301   // generate uAux->u with length including halo
00302   xTou.resize(haloXSize);
00303   yTou.resize(haloYSIZE);
00304   zTou.resize(haloZSize);
00305   // same for ghost layer lookup tables
00306   const sunindextype ghostXSize = gLW * ny * nz;
00307   const sunindextype ghostYSIZE = gLW * nx * nz;
00308   const sunindextype ghostZSize = gLW * nx * ny;
00309   lgcTox.resize(ghostXSize);
00310   rgcTox.resize(ghostXSize);
00311   lgcToy.resize(ghostYSIZE);
00312   rgcToy.resize(ghostYSIZE);
00313   lgcToz.resize(ghostZSize);
00314   rgcToz.resize(ghostZSize);
00315   // variables for cartesian position in the 3D discrete lattice
00316   // sunindextype px = 0, py = 0, pz = 0;
00317   // Fill the lookup tables
00318   #pragma omp parallel default(none) \
00319   private(px, py, pz) \
00320   shared(uTox, uToy, uToz, xTou, yTou, zTou, \
00321         nx, ny, mx, my, mz, gLW, totalNP, \
00322         lgcTox, rgcTox, lgcToy, rgcToy, lgcToz, rgcToz, \
00323         ghostXSize, ghostYSIZE, ghostZSize)
00324 {
00325   #pragma omp for simd schedule(static)
00326   for (sunindextype i = 0; i < totalNP; i++) { // loop over the patch
00327     // calculate cartesian coordinates
00328     px = i % nx;
00329     py = (i / nx) % ny;
00330     pz = (i / nx) / ny;
00331     // fill lookups extended by halos (useful for y and z direction)
00332     uTox[i] = (px + gLW) + py * mx +
00333               pz * mx * ny; // unroll (de-flatten) cartesian dimension
00334     xTou[px + py * mx + pz * mx * ny] =
00335       i; // match cartesian point to u location
00336     uToy[i] = (py + gLW) + pz * my + px * my * nz;
00337     yTou[py + pz * my + px * my * nz] = i;
00338     uToz[i] = (pz + gLW) + px * mz + py * mz * nx;
00339     zTou[pz + px * mz + py * mz * nx] = i;
00340   }
00341   #pragma omp for simd schedule(static)
00342   for (sunindextype i = 0; i < ghostXSize; i++) {
00343     px = i % gLW;
00344     py = (i / gLW) % ny;
00345     pz = (i / gLW) / ny;
00346     lgcTox[i] = px + py * mx + pz * mx * ny;
00347     rgcTox[i] = px + nx + gLW + py * mx + pz * mx * ny;
00348   }
00349   #pragma omp for simd schedule(static)
00350   for (sunindextype i = 0; i < ghostYSIZE; i++) {
00351     px = i % nx;
00352     py = (i / nx) % gLW;
00353     pz = (i / nx) / gLW;
00354     lgcToy[i] = py + pz * my + px * my * nz;
00355     rgcToy[i] = py + ny + gLW + pz * my + px * my * nz;
00356   }
00357   #pragma omp for simd schedule(static)
00358   for (sunindextype i = 0; i < ghostZSize; i++) {
00359     px = i % nx;
00360     py = (i / nx) % ny;
00361     pz = (i / nx) / ny;
00362     lgcToz[i] = pz + px * mz + py * mz * nx;
00363     rgcToz[i] = pz + nz + gLW + px * mz + py * mz * nx;
00364   }
00365 }
00366 }
00367 statusFlags |= TranslocationLookupSetUp;

```

```

00368 }
00369
00370 /** Rotate into eigenraum along R matrices of paper using the rotation
00371 * methods;
00372 * uAuxData gets the rotated left-halo-, inner-patch-, right-halo-data */
00373 void LatticePatch::rotateIntoEigen(const int dir) {
00374 // Check that the lattice, ghost layers as well as the translocation lookups
00375 // have been set up;
00376 checkFlag(FLatticePatchSetUp);
00377 checkFlag(TranslocationLookupSetUp);
00378 checkFlag(GhostLayersInitialized); // this check is only after call to
00379 // exchange ghost cells
00380 switch (dir) {
00381 case 1:
00382     rotateToX(uAuxData, gCLData, lgcToX);
00383     rotateToX(uAuxData, uData, uToX);
00384     rotateToX(uAuxData, gCRData, rgcToX);
00385     break;
00386 case 2:
00387     rotateToY(uAuxData, gCLData, lgcToY);
00388     rotateToY(uAuxData, uData, uToY);
00389     rotateToY(uAuxData, gCRData, rgcToY);
00390     break;
00391 case 3:
00392     rotateToZ(uAuxData, gCLData, lgcToZ);
00393     rotateToZ(uAuxData, uData, uToZ);
00394     rotateToZ(uAuxData, gCRData, rgcToZ);
00395     break;
00396 default:
00397     errorKill("Tried to rotate into the wrong direction");
00398     break;
00399 }
00400 }
00401
00402 /// Rotate halo and inner-patch data vectors with rotation matrix Rx into
00403 /// eigenspace of Z matrix and write to auxiliary vector
00404 inline void LatticePatch::rotateToX(sunrealtype *outArray,
00405                                     const unrealtype *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         // get correct u-vector and spatial indices along previously defined lookup
00416         // tables
00417         target = dPD * lookup[i];
00418         ii = dPD * i;
00419         outArray[target + 0] = -inArray[1 + ii] + inArray[5 + ii];
00420         outArray[target + 1] = inArray[2 + ii] + inArray[4 + ii];
00421         outArray[target + 2] = inArray[1 + ii] + inArray[5 + ii];
00422         outArray[target + 3] = -inArray[2 + ii] + inArray[4 + ii];
00423         outArray[target + 4] = inArray[3 + ii];
00424         outArray[target + 5] = inArray[ii];
00425     }
00426 }
00427
00428 /// Rotate halo and inner-patch data vectors with rotation matrix Ry into
00429 /// eigenspace of Z matrix and write to auxiliary vector
00430 inline void LatticePatch::rotateToY(sunrealtype *outArray,
00431                                     const unrealtype *inArray,
00432                                     const std::vector<sunindextype> &lookup) {
00433     sunindextype ii = 0, target = 0;
00434     const int dPD = envelopeLattice->get_dataPointDimension();
00435     const sunindextype size = lookup.size();
00436 #pragma omp parallel for simd \
00437 private(target, ii) \
00438 shared(lookup, outArray, inArray, size, dPD) \
00439 schedule(static)
00440     for (sunindextype i = 0; i < size; i++) {
00441         target = dPD * lookup[i];
00442         ii = dPD * i;
00443         outArray[target + 0] = inArray[ii] + inArray[5 + ii];
00444         outArray[target + 1] = -inArray[2 + ii] + inArray[3 + ii];
00445         outArray[target + 2] = -inArray[ii] + inArray[5 + ii];
00446         outArray[target + 3] = inArray[2 + ii] + inArray[3 + ii];
00447         outArray[target + 4] = inArray[4 + ii];
00448         outArray[target + 5] = inArray[1 + ii];
00449     }
00450 }
00451
00452 /// Rotate halo and inner-patch data vectors with rotation matrix Rz into
00453 /// eigenspace of Z matrix and write to auxiliary vector
00454 inline void LatticePatch::rotateToZ(sunrealtype *outArray,

```

```

00455                               const sunrealtype *inArray,
00456                               const std::vector<sunindextype> &lookup) {
00457     sunindextype ii = 0, target = 0;
00458     const sunindextype size = lookup.size();
00459     const int dPD = envelopeLattice->get_dataPointDimension();
00460     #pragma omp parallel for simd \
00461     private(target, ii) \
00462     shared(lookup, outArray, inArray, size, dPD) \
00463     schedule(static)
00464     for (sunindextype i = 0; i < size; i++) {
00465         target = dPD * lookup[i];
00466         ii = dPD * i;
00467         outArray[target + 0] = -inArray[ii] + inArray[4 + ii];
00468         outArray[target + 1] = inArray[1 + ii] + inArray[3 + ii];
00469         outArray[target + 2] = inArray[ii] + inArray[4 + ii];
00470         outArray[target + 3] = -inArray[1 + ii] + inArray[3 + ii];
00471         outArray[target + 4] = inArray[5 + ii];
00472         outArray[target + 5] = inArray[2 + ii];
00473     }
00474 }
00475
00476 /// Derotate uAux with transposed rotation matrices and write to derivative
00477 /// buffer -- normalization is done here by the factor 1/2
00478 void LatticePatch::derotate(int dir, sunrealtype *buffOut) {
00479     // Check that the lattice as well as the translocation lookups have been set
00480     // up;
00481     checkFlag(FLatticePatchSetUp);
00482     checkFlag(TranslocationLookupSetUp);
00483     const int dPD = envelopeLattice->get_dataPointDimension();
00484     const int gLW = envelopeLattice->get_ghostLayerWidth();
00485     const sunindextype totalNP = discreteSize();
00486     sunindextype ii = 0, target = 0;
00487     switch (dir) {
00488     case 1:
00489         #pragma omp parallel for simd \
00490         private(ii, target) \
00491         shared(dPD, gLW, totalNP, uTox, uAux, buffOut) \
00492         schedule(static)
00493         for (sunindextype i = 0; i < totalNP; i++) {
00494             // get correct indices in u and rotation space
00495             target = dPD * i;
00496             ii = dPD * (uTox[i] - gLW);
00497             buffOut[target + 0] = uAux[5 + ii];
00498             buffOut[target + 1] = (-uAux[ii] + uAux[2 + ii]) / 2.;
00499             buffOut[target + 2] = (uAux[1 + ii] - uAux[3 + ii]) / 2.;
00500             buffOut[target + 3] = uAux[4 + ii];
00501             buffOut[target + 4] = (uAux[1 + ii] + uAux[3 + ii]) / 2.;
00502             buffOut[target + 5] = (uAux[ii] + uAux[2 + ii]) / 2.;
00503         }
00504         break;
00505     case 2:
00506         #pragma omp parallel for simd \
00507         private(ii, target) \
00508         shared(dPD, gLW, totalNP, uTox, uAux, buffOut) \
00509         schedule(static)
00510         for (sunindextype i = 0; i < totalNP; i++) {
00511             target = dPD * i;
00512             ii = dPD * (uTox[i] - gLW);
00513             buffOut[target + 0] = (uAux[ii] - uAux[2 + ii]) / 2.;
00514             buffOut[target + 1] = uAux[5 + ii];
00515             buffOut[target + 2] = (-uAux[1 + ii] + uAux[3 + ii]) / 2.;
00516             buffOut[target + 3] = (uAux[1 + ii] + uAux[3 + ii]) / 2.;
00517             buffOut[target + 4] = uAux[4 + ii];
00518             buffOut[target + 5] = (uAux[ii] + uAux[2 + ii]) / 2.;
00519         }
00520         break;
00521     case 3:
00522         #pragma omp parallel for simd \
00523         private(ii, target) \
00524         shared(dPD, gLW, totalNP, uTox, uAux, buffOut) \
00525         schedule(static)
00526         for (sunindextype i = 0; i < totalNP; i++) {
00527             target = dPD * i;
00528             ii = dPD * (uToz[i] - gLW);
00529             buffOut[target + 0] = (-uAux[ii] + uAux[2 + ii]) / 2.;
00530             buffOut[target + 1] = (uAux[1 + ii] - uAux[3 + ii]) / 2.;
00531             buffOut[target + 2] = uAux[5 + ii];
00532             buffOut[target + 3] = (uAux[1 + ii] + uAux[3 + ii]) / 2.;
00533             buffOut[target + 4] = (uAux[ii] + uAux[2 + ii]) / 2.;
00534             buffOut[target + 5] = uAux[4 + ii];
00535         }
00536         break;
00537     default:
00538         errorKill("Tried to derotate from the wrong direction");
00539         break;
00540     }
00541 }
```

```

00542
00543 // Create buffers to save derivative values, optimizing computational load
00544 void LatticePatch::initializeBuffers() {
00545 // Check that the lattice has been set up
00546 checkFlag(FLatticeDimensionSet);
00547 const int dPD = envelopeLattice->get_dataPointDimension();
00548 buffX.resize(nx * ny * nz * dPD);
00549 buffY.resize(nx * ny * nz * dPD);
00550 buffZ.resize(nx * ny * nz * dPD);
00551 // Set pointers used for propagation functions
00552 buffData[0] = &buffX[0];
00553 buffData[1] = &buffY[0];
00554 buffData[2] = &buffZ[0];
00555 statusFlags |= BuffersInitialized;
00556 }
00557
00558 /// Perform the ghost cell exchange in a specified direction
00559 void LatticePatch::exchangeGhostCells(const int dir) {
00560 // Check that the lattice has been set up
00561 checkFlag(FLatticeDimensionSet);
00562 checkFlag(FLatticePatchSetUp);
00563 // Variables to per dimension calculate the halo indices, and distance to
00564 // other side halo boundary
00565 int mx = 1, my = 1, mz = 1, distToRight = 1;
00566 const int gLW = envelopeLattice->get_ghostLayerWidth();
00567 // In the chosen direction m is set to ghost layer width while the others
00568 // remain to form the plane
00569 switch (dir) {
00570 case 1:
00571     mx = gLW;
00572     my = ny;
00573     mz = nz;
00574     distToRight = (nx - gLW);
00575     break;
00576 case 2:
00577     mx = nx;
00578     my = gLW;
00579     mz = nz;
00580     distToRight = nx * (ny - gLW);
00581     break;
00582 case 3:
00583     mx = nx;
00584     my = ny;
00585     mz = gLW;
00586     distToRight = nx * ny * (nz - gLW);
00587     break;
00588 }
00589 // total number of exchanged points
00590 const int dPD = envelopeLattice->get_dataPointDimension();
00591 const sunindextype exchangeSize = mx * my * mz * dPD;
00592 // provide size of the halos for ghost cells
00593 ghostCellLeft.resize(exchangeSize);
00594 ghostCellRight.resize(ghostCellLeft.size());
00595 ghostCellLeftToSend.resize(ghostCellLeft.size());
00596 ghostCellRightToSend.resize(ghostCellLeft.size());
00597 statusFlags |= GhostLayersInitialized;
00598
00599 // Initialize running index li for the halo buffers, and index ui of uData for
00600 // data transfer
00601 sunindextype li = 0, ui = 0;
00602 // Fill the halo buffers
00603 #pragma omp parallel for default(none) \
00604 private(ui, li) \
00605 shared(nx, ny, mx, my, mz, dPD, distToRight, uData, \
00606         ghostCellLeftToSend, ghostCellRightToSend)
00607 for (sunindextype iz = 0; iz < mz; iz++) {
00608     for (sunindextype iy = 0; iy < my; iy++) {
00609         // uData vector start index of halo data to be transferred
00610         // with each z-step add the whole xy-plane and with y-step the x-range ->
00611         // iterate all x-ranges
00612         ui = (iz * nx * ny + iy * nx) * dPD;
00613         // increase halo index by transferred items of previous iteration steps
00614         li = (iz * my * mx + iy * mx) * dPD;
00615         // copy left halo data from uData to buffer, transfer size is given by
00616         // x-length (not x-range)
00617         std::copy(&uData[ui], &uData[ui + mx * dPD], &ghostCellLeftToSend[li]);
00618         ui += distToRight * dPD;
00619         std::copy(&uData[ui], &uData[ui + mx * dPD], &ghostCellRightToSend[li]);
00620     }
00621 }
00622
00623 #if !defined(_MPI)
00624 std::copy(&ghostCellLeftToSend[0], &ghostCellLeftToSend[exchangeSize],
00625             &ghostCellRight[0]);
00626 std::copy(&ghostCellRightToSend[0], &ghostCellRightToSend[exchangeSize],
00627             &ghostCellLeft[0]);
00628

```

```

00629 #elif defined(_MPI)
00630 /* Send and receive the data to and from neighboring latticePatches */
00631 // Adjust direction to cartesian communicator
00632 int dim = 2; // default for dir==1
00633 if (dir == 2) {
00634     dim = 1;
00635 } else if (dir == 3) {
00636     dim = 0;
00637 }
00638 int rank_source = 0, rank_dest = 0;
00639 MPI_Cart_shift(envelopeLattice->comm, dim, -1, &rank_source,
00640                 &rank_dest); // s.t. rank_dest is left & v.v.
00641
00642 // nonblocking Irecv/Isend
00643
00644 MPI_Request requests[4];
00645 MPI_Irecv(&ghostCellRight[0], exchangeSize, MPI_SUNREALTYPE, rank_source, 1,
00646 envelopeLattice->comm, &requests[0]);
00647 MPI_Isend(&ghostCellLeftToSend[0], exchangeSize, MPI_SUNREALTYPE, rank_dest,
00648 1, envelopeLattice->comm, &requests[1]);
00649 MPI_Irecv(&ghostCellLeft[0], exchangeSize, MPI_SUNREALTYPE, rank_dest, 2,
00650 envelopeLattice->comm, &requests[2]);
00651 MPI_Isend(&ghostCellRightToSend[0], exchangeSize, MPI_SUNREALTYPE,
00652 rank_source, 2, envelopeLattice->comm, &requests[3]);
00653 MPI_Waitall(4, requests, MPI_STATUS_IGNORE);
00654
00655 // blocking Sendrecv:
00656 /*
00657 MPI_Sendrecv(&ghostCellLeftToSend[0], exchangeSize, MPI_SUNREALTYPE,
00658             rank_dest, 1, &ghostCellRight[0], exchangeSize, MPI_SUNREALTYPE,
00659             rank_source, 1, envelopeLattice->comm, MPI_STATUS_IGNORE);
00660 MPI_Sendrecv(&ghostCellRightToSend[0], exchangeSize, MPI_SUNREALTYPE,
00661             rank_source, 2, &ghostCellLeft[0], exchangeSize, MPI_SUNREALTYPE,
00662             rank_dest, 2, envelopeLattice->comm, MPI_STATUS_IGNORE);
00663 */
00664 #endif
00665 gCLData = &ghostCellLeft[0];
00666 gCRData = &ghostCellRight[0];
00667 }
00668
00669 /// Check if all flags are set
00670 void LatticePatch::checkFlag(unsigned int flag) const {
00671     if (!(statusFlags & flag)) {
00672         std::string errorMessage;
00673         switch (flag) {
00674             case FlatticePatchSetUp:
00675                 errorMessage = "The Lattice patch was not set up please make sure to "
00676                             "initialize a Lattice topology";
00677                 break;
00678             case TranslocationLookupSetUp:
00679                 errorMessage = "The translocation lookup tables have not been generated, "
00680                             "please be sure to run generateTranslocationLookup()";
00681                 break;
00682             case GhostLayersInitialized:
00683                 errorMessage = "The space for the ghost layers has not been allocated, "
00684                             "please be sure that the ghost cells are initialized ";
00685                 break;
00686             case BuffersInitialized:
00687                 errorMessage = "The space for the buffers has not been allocated, please "
00688                             "be sure to run initializeBuffers()";
00689                 break;
00690             default:
00691                 errorMessage = "Uppss, you've made a non-standard error, sadly I can't "
00692                             "help you there";
00693                 break;
00694         }
00695         errorKill(errorMessage);
00696     }
00697     return;
00698 }
00699
00700 /// Calculate derivatives in the patch (uAux) in the specified direction
00701 void LatticePatch::derive(const int dir) {
00702     // ghost layer width adjusted to the chosen stencil order
00703     const int gLW = envelopeLattice->get_ghostLayerWidth();
00704     // dimensionality of data points -> 6
00705     const int dPD = envelopeLattice->get_dataPointDimension();
00706     // total width of patch in given direction including ghost layers at ends
00707     const sunindextype dirWidth = discreteSize(dir) + 2 * gLW;
00708     // width of patch only in given direction
00709     const sunindextype dirWidth0 = discreteSize(dir);
00710     // size of plane perpendicular to given dimension
00711     const sunindextype perpPlainSize = discreteSize() / discreteSize(dir);
00712     // physical distance between points in that direction
00713     sunrealtype dxi = nan("0x12345");
00714     switch (dir) {
00715         case 1:

```

```

00716     dxi = dx;
00717     break;
00718 case 2:
00719     dxi = dy;
00720     break;
00721 case 3:
00722     dxi = dz;
00723     break;
00724 default:
00725     dxi = 1;
00726     errorKill("Tried to derive in the wrong direction");
00727     break;
00728 }
// Derive according to chosen stencil accuracy order
00729 const int order = envelopeLattice->get_stencilOrder();
00730 switch (order) {
00731 case 1: // gLW=1
00732 #pragma omp parallel for default(none) \
00733     shared(perpPlainSize, dxi, dirWidth, dirWidthO, gLW, dPD, uAux)
00734     for (sunindextype i = 0; i < perpPlainSize; i++) {
00735         #pragma omp simd
00736         for (sunindextype j = (i * dirWidth + gLW) * dPD;
00737             j < (i * dirWidth + gLW + dirWidthO) * dPD; j += dPD) {
00738             uAux[j + 0 - gLW * dPD] = s1b(&uAux[j + 0]) / dxi;
00739             uAux[j + 1 - gLW * dPD] = s1b(&uAux[j + 1]) / dxi;
00740             uAux[j + 2 - gLW * dPD] = s1f(&uAux[j + 2]) / dxi;
00741             uAux[j + 3 - gLW * dPD] = s1f(&uAux[j + 3]) / dxi;
00742             uAux[j + 4 - gLW * dPD] = s1f(&uAux[j + 4]) / dxi;
00743             uAux[j + 5 - gLW * dPD] = s1f(&uAux[j + 5]) / dxi;
00744         }
00745     }
00746 }
00747 break;
00748 case 2: // gLW=2
00749 #pragma omp parallel for default(none) \
00750     shared(perpPlainSize, dxi, dirWidth, dirWidthO, gLW, dPD, uAux)
00751     for (sunindextype i = 0; i < perpPlainSize; i++) {
00752         #pragma omp simd
00753         for (sunindextype j = (i * dirWidth + gLW) * dPD;
00754             j < (i * dirWidth + gLW + dirWidthO) * dPD; j += dPD) {
00755             uAux[j + 0 - gLW * dPD] = s2b(&uAux[j + 0]) / dxi;
00756             uAux[j + 1 - gLW * dPD] = s2b(&uAux[j + 1]) / dxi;
00757             uAux[j + 2 - gLW * dPD] = s2f(&uAux[j + 2]) / dxi;
00758             uAux[j + 3 - gLW * dPD] = s2f(&uAux[j + 3]) / dxi;
00759             uAux[j + 4 - gLW * dPD] = s2c(&uAux[j + 4]) / dxi;
00760             uAux[j + 5 - gLW * dPD] = s2c(&uAux[j + 5]) / dxi;
00761         }
00762     }
00763 }
00764 case 3: // gLW=2
00765 #pragma omp parallel for default(none) \
00766     shared(perpPlainSize, dxi, dirWidth, dirWidthO, gLW, dPD, uAux)
00767     for (sunindextype i = 0; i < perpPlainSize; i++) {
00768         #pragma omp simd
00769         for (sunindextype j = (i * dirWidth + gLW) * dPD;
00770             j < (i * dirWidth + gLW + dirWidthO) * dPD; j += dPD) {
00771             uAux[j + 0 - gLW * dPD] = s3b(&uAux[j + 0]) / dxi;
00772             uAux[j + 1 - gLW * dPD] = s3b(&uAux[j + 1]) / dxi;
00773             uAux[j + 2 - gLW * dPD] = s3f(&uAux[j + 2]) / dxi;
00774             uAux[j + 3 - gLW * dPD] = s3f(&uAux[j + 3]) / dxi;
00775             uAux[j + 4 - gLW * dPD] = s3f(&uAux[j + 4]) / dxi;
00776             uAux[j + 5 - gLW * dPD] = s3f(&uAux[j + 5]) / dxi;
00777         }
00778     }
00779 }
00780 break;
00781 case 4: // gLW=3
00782 #pragma omp parallel for default(none) \
00783     shared(perpPlainSize, dxi, dirWidth, dirWidthO, gLW, dPD, uAux)
00784     for (sunindextype i = 0; i < perpPlainSize; i++) {
00785         #pragma omp simd
00786         for (sunindextype j = (i * dirWidth + gLW) * dPD;
00787             j < (i * dirWidth + gLW + dirWidthO) * dPD; j += dPD) {
00788             uAux[j + 0 - gLW * dPD] = s4b(&uAux[j + 0]) / dxi;
00789             uAux[j + 1 - gLW * dPD] = s4b(&uAux[j + 1]) / dxi;
00790             uAux[j + 2 - gLW * dPD] = s4f(&uAux[j + 2]) / dxi;
00791             uAux[j + 3 - gLW * dPD] = s4f(&uAux[j + 3]) / dxi;
00792             uAux[j + 4 - gLW * dPD] = s4c(&uAux[j + 4]) / dxi;
00793             uAux[j + 5 - gLW * dPD] = s4c(&uAux[j + 5]) / dxi;
00794     }
00795 }
00796 break;
00797 case 5: // gLW=3
00798 #pragma omp parallel for default(none) \
00799     shared(perpPlainSize, dxi, dirWidth, dirWidthO, gLW, dPD, uAux)
00800     for (sunindextype i = 0; i < perpPlainSize; i++) {
00801         #pragma omp simd
00802         for (sunindextype j = (i * dirWidth + gLW) * dPD;
00803             j < (i * dirWidth + gLW + dirWidthO) * dPD; j += dPD) {

```

```

00803     uAux[j + 0 - gLW * dPD] = s5b(&uAux[j + 0]) / dxi;
00804     uAux[j + 1 - gLW * dPD] = s5b(&uAux[j + 1]) / dxi;
00805     uAux[j + 2 - gLW * dPD] = s5f(&uAux[j + 2]) / dxi;
00806     uAux[j + 3 - gLW * dPD] = s5f(&uAux[j + 3]) / dxi;
00807     uAux[j + 4 - gLW * dPD] = s5f(&uAux[j + 4]) / dxi;
00808     uAux[j + 5 - gLW * dPD] = s5f(&uAux[j + 5]) / dxi;
00809 }
00810 }
00811 break;
00812 case 6: // gLW=4
00813 #pragma omp parallel for default(none) \
00814 shared(perpPlainSize, dxi, dirWidth, dirWidthO, gLW, dPD, uAux)
00815 for (sunindextype i = 0; i < perpPlainSize; i++) {
00816     #pragma omp simd
00817     for (sunindextype j = (i * dirWidth + gLW) * dPD;
00818         j < (i * dirWidth + gLW + dirWidthO) * dPD; j += dPD) {
00819         uAux[j + 0 - gLW * dPD] = s6b(&uAux[j + 0]) / dxi;
00820         uAux[j + 1 - gLW * dPD] = s6b(&uAux[j + 1]) / dxi;
00821         uAux[j + 2 - gLW * dPD] = s6f(&uAux[j + 2]) / dxi;
00822         uAux[j + 3 - gLW * dPD] = s6f(&uAux[j + 3]) / dxi;
00823         uAux[j + 4 - gLW * dPD] = s6c(&uAux[j + 4]) / dxi;
00824         uAux[j + 5 - gLW * dPD] = s6c(&uAux[j + 5]) / dxi;
00825     }
00826 }
00827 break;
00828 case 7: // gLW=4
00829 #pragma omp parallel for default(none) \
00830 shared(perpPlainSize, dxi, dirWidth, dirWidthO, gLW, dPD, uAux)
00831 for (sunindextype i = 0; i < perpPlainSize; i++) {
00832     #pragma omp simd
00833     for (sunindextype j = (i * dirWidth + gLW) * dPD;
00834         j < (i * dirWidth + gLW + dirWidthO) * dPD; j += dPD) {
00835         uAux[j + 0 - gLW * dPD] = s7b(&uAux[j + 0]) / dxi;
00836         uAux[j + 1 - gLW * dPD] = s7b(&uAux[j + 1]) / dxi;
00837         uAux[j + 2 - gLW * dPD] = s7f(&uAux[j + 2]) / dxi;
00838         uAux[j + 3 - gLW * dPD] = s7f(&uAux[j + 3]) / dxi;
00839         uAux[j + 4 - gLW * dPD] = s7f(&uAux[j + 4]) / dxi;
00840         uAux[j + 5 - gLW * dPD] = s7f(&uAux[j + 5]) / dxi;
00841     }
00842 }
00843 break;
00844 case 8: // gLW=5
00845 #pragma omp parallel for default(none) \
00846 shared(perpPlainSize, dxi, dirWidth, dirWidthO, gLW, dPD, uAux)
00847 for (sunindextype i = 0; i < perpPlainSize; i++) {
00848     #pragma omp simd
00849     for (sunindextype j = (i * dirWidth + gLW) * dPD;
00850         j < (i * dirWidth + gLW + dirWidthO) * dPD; j += dPD) {
00851         uAux[j + 0 - gLW * dPD] = s8b(&uAux[j + 0]) / dxi;
00852         uAux[j + 1 - gLW * dPD] = s8b(&uAux[j + 1]) / dxi;
00853         uAux[j + 2 - gLW * dPD] = s8f(&uAux[j + 2]) / dxi;
00854         uAux[j + 3 - gLW * dPD] = s8f(&uAux[j + 3]) / dxi;
00855         uAux[j + 4 - gLW * dPD] = s8c(&uAux[j + 4]) / dxi;
00856         uAux[j + 5 - gLW * dPD] = s8c(&uAux[j + 5]) / dxi;
00857     }
00858 }
00859 break;
00860 case 9: // gLW=5
00861 #pragma omp parallel for default(none) \
00862 shared(perpPlainSize, dxi, dirWidth, dirWidthO, gLW, dPD, uAux)
00863 for (sunindextype i = 0; i < perpPlainSize; i++) {
00864     #pragma omp simd
00865     for (sunindextype j = (i * dirWidth + gLW) * dPD;
00866         j < (i * dirWidth + gLW + dirWidthO) * dPD; j += dPD) {
00867         uAux[j + 0 - gLW * dPD] = s9b(&uAux[j + 0]) / dxi;
00868         uAux[j + 1 - gLW * dPD] = s9b(&uAux[j + 1]) / dxi;
00869         uAux[j + 2 - gLW * dPD] = s9f(&uAux[j + 2]) / dxi;
00870         uAux[j + 3 - gLW * dPD] = s9f(&uAux[j + 3]) / dxi;
00871         uAux[j + 4 - gLW * dPD] = s9f(&uAux[j + 4]) / dxi;
00872         uAux[j + 5 - gLW * dPD] = s9f(&uAux[j + 5]) / dxi;
00873     }
00874 }
00875 break;
00876 case 10: // gLW=6
00877 #pragma omp parallel for default(none) \
00878 shared(perpPlainSize, dxi, dirWidth, dirWidthO, gLW, dPD, uAux)
00879 for (sunindextype i = 0; i < perpPlainSize; i++) {
00880     #pragma omp simd
00881     for (sunindextype j = (i * dirWidth + gLW) * dPD;
00882         j < (i * dirWidth + gLW + dirWidthO) * dPD; j += dPD) {
00883         uAux[j + 0 - gLW * dPD] = s10b(&uAux[j + 0]) / dxi;
00884         uAux[j + 1 - gLW * dPD] = s10b(&uAux[j + 1]) / dxi;
00885         uAux[j + 2 - gLW * dPD] = s10f(&uAux[j + 2]) / dxi;
00886         uAux[j + 3 - gLW * dPD] = s10f(&uAux[j + 3]) / dxi;
00887         uAux[j + 4 - gLW * dPD] = s10c(&uAux[j + 4]) / dxi;
00888         uAux[j + 5 - gLW * dPD] = s10c(&uAux[j + 5]) / dxi;
00889     }
}

```

```

00890     }
00891     break;
00892 case 11: // gLW=6
00893 #pragma omp parallel for default(none) \
00894 shared(perpPlainSize, dxi, dirWidth, dirWidthO, gLW, dPD, uAux)
00895 for (sunindextype i = 0; i < perpPlainSize; i++) {
00896     #pragma omp simd
00897     for (sunindextype j = (i * dirWidth + gLW) * dPD;
00898         j < (i * dirWidth + gLW + dirWidthO) * dPD; j += dPD) {
00899         uAux[j + 0 - gLW * dPD] = s11b(&uAux[j + 0]) / dxi;
00900         uAux[j + 1 - gLW * dPD] = s11b(&uAux[j + 1]) / dxi;
00901         uAux[j + 2 - gLW * dPD] = s11f(&uAux[j + 2]) / dxi;
00902         uAux[j + 3 - gLW * dPD] = s11f(&uAux[j + 3]) / dxi;
00903         uAux[j + 4 - gLW * dPD] = s11f(&uAux[j + 4]) / dxi;
00904         uAux[j + 5 - gLW * dPD] = s11f(&uAux[j + 5]) / dxi;
00905     }
00906 }
00907 break;
00908 case 12: // gLW=7
00909 #pragma omp parallel for default(none) \
00910 shared(perpPlainSize, dxi, dirWidth, dirWidthO, gLW, dPD, uAux)
00911 for (sunindextype i = 0; i < perpPlainSize; i++) {
00912     #pragma omp simd
00913     for (sunindextype j = (i * dirWidth + gLW) * dPD;
00914         j < (i * dirWidth + gLW + dirWidthO) * dPD; j += dPD) {
00915         uAux[j + 0 - gLW * dPD] = s12b(&uAux[j + 0]) / dxi;
00916         uAux[j + 1 - gLW * dPD] = s12b(&uAux[j + 1]) / dxi;
00917         uAux[j + 2 - gLW * dPD] = s12f(&uAux[j + 2]) / dxi;
00918         uAux[j + 3 - gLW * dPD] = s12f(&uAux[j + 3]) / dxi;
00919         uAux[j + 4 - gLW * dPD] = s12c(&uAux[j + 4]) / dxi;
00920         uAux[j + 5 - gLW * dPD] = s12c(&uAux[j + 5]) / dxi;
00921     }
00922 }
00923 break;
00924 case 13: // gLW=7
00925 // For all points in the plane perpendicular to the given direction
00926 #pragma omp parallel for default(none) \
00927 shared(perpPlainSize, dxi, dirWidth, dirWidthO, gLW, dPD, uAux)
00928 for (sunindextype i = 0; i < perpPlainSize; i++) {
00929     // iterate through the derivation direction
00930     #pragma omp simd
00931     for (sunindextype j = (i * dirWidth
00932             + gLW /*to shift left by gLW below */) * dPD;
00933         j < (i * dirWidth + gLW + dirWidthO) * dPD; j += dPD) {
00934         // Compute the stencil derivative for any of the six field components
00935         // and update position by ghost width shift
00936         uAux[j + 0 - gLW * dPD] = s13b(&uAux[j + 0]) / dxi;
00937         uAux[j + 1 - gLW * dPD] = s13b(&uAux[j + 1]) / dxi;
00938         uAux[j + 2 - gLW * dPD] = s13f(&uAux[j + 2]) / dxi;
00939         uAux[j + 3 - gLW * dPD] = s13f(&uAux[j + 3]) / dxi;
00940         uAux[j + 4 - gLW * dPD] = s13f(&uAux[j + 4]) / dxi;
00941         uAux[j + 5 - gLW * dPD] = s13f(&uAux[j + 5]) / dxi;
00942     }
00943 }
00944 break;
00945
00946 default:
00947     errorKill("Please set an existing stencil order");
00948     break;
00949 }
00950 }
00951
00952 ////////////// Helper functions ///////////
00953
00954 // Print a specific error message to stderr
00955 void errorKill(const std::string & errorMessage) {
00956     int my_prc=0;
00957 #if defined(_MPI)
00958     MPI_Comm_rank(MPI_COMM_WORLD,&my_prc);
00959 #endif
00960     if (my_prc==0) {
00961         std::cerr << std::endl << "Error: " << errorMessage
00962         << "\nAborting..." << std::endl;
00963 #if defined(_MPI)
00964     MPI_Abort(MPI_COMM_WORLD, 1);
00965 #else
00966     exit(1);
00967 #endif
00968     return;
00969 }
00970 }
00971
00972 /** Check function return value. Adapted from CVode examples.
00973     opt == 0 means SUNDIALS function allocates memory so check if
00974         returned NULL pointer
00975     opt == 1 means SUNDIALS function returns an integer value so check if
00976         retval < 0

```

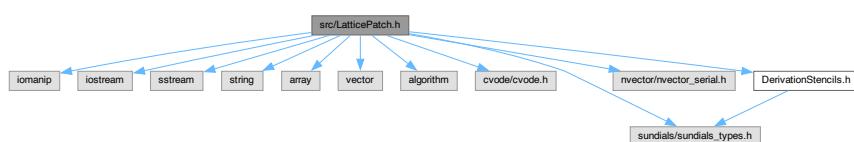
```

00977     opt == 2 means function allocates memory so check if returned
00978     NULL pointer */
00979 int check_retval(void *returnvalue, const char *funcname, int opt, int id) {
00980     int *retval = nullptr;
00981
00982     /* Check if SUNDIALS function returned NULL pointer - no memory allocated */
00983     if (opt == 0 && returnvalue == nullptr) {
00984         fprintf(stderr,
00985             "\nSUNDIALS_ERROR(process %d): %s() failed - "
00986             "returned NULL pointer\n\n", id, funcname);
00987         return (1);
00988     }
00989
00990     /* Check if retval < 0 */
00991     else if (opt == 1) {
00992         retval = (int *)returnvalue;
00993         char *flagname = CVodeGetReturnFlagName(*retval);
00994         if (*retval < 0) {
00995             fprintf(stderr, "\nSUNDIALS_ERROR(process %d): %s() failed "
00996                 "with retval = %d: %s\n\n",
00997                 id, funcname, *retval, flagname);
00998             return (1);
00999         }
01000     }
01001
01002     /* Check if function returned NULL pointer - no memory allocated */
01003     else if (opt == 2 && returnvalue == nullptr) {
01004         fprintf(stderr,
01005             "\nMEMORY_ERROR(process %d): %s() failed - "
01006             "returned NULL pointer\n\n", id, funcname);
01007         return (1);
01008     }
01009
01010     return (0);
01011 }
```

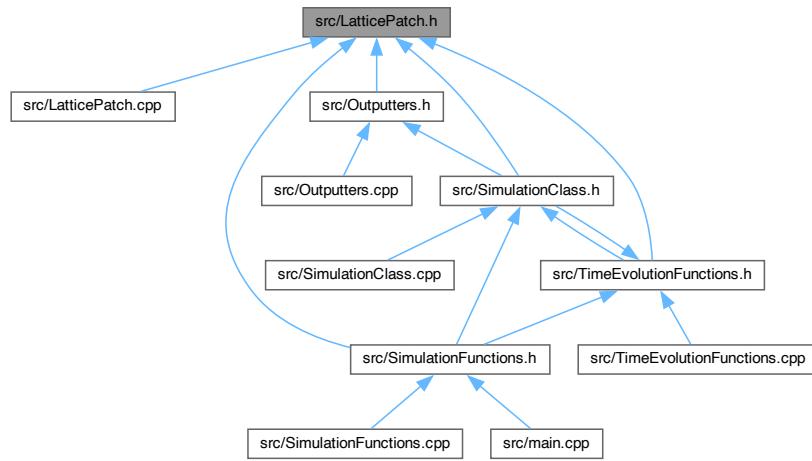
6.12 src/LatticePatch.h File Reference

Declaration of the lattice and lattice patches.

```
#include <iomanip>
#include <iostream>
#include <sstream>
#include <string>
#include <array>
#include <vector>
#include <algorithm>
#include <cvode/cvode.h>
#include <sundials/sundials_types.h>
#include <nvector/nvector_serial.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_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_retval()

```
int check_retval (
    void * returnvalue,
    const char * funcname,
    int opt,
    int id )
```

helper function to check CVode errors

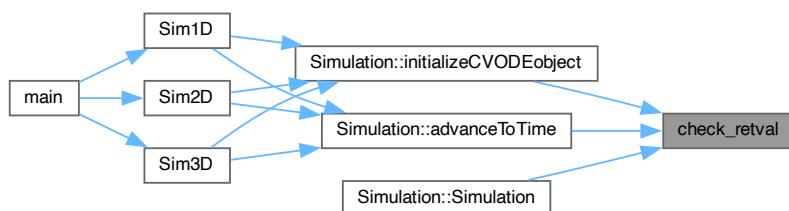
Check function return value. Adapted from CVode examples. opt == 0 means SUNDIALS function allocates memory so check if returned NULL pointer opt == 1 means SUNDIALS function returns an integer value so check if retval < 0 opt == 2 means function allocates memory so check if returned NULL pointer

Definition at line 979 of file [LatticePatch.cpp](#).

```
00979
00980     int *retval = nullptr;
00981
00982     /* Check if SUNDIALS function returned NULL pointer - no memory allocated */
00983     if (opt == 0 && returnvalue == nullptr) {
00984         fprintf(stderr,
00985             "\nSUNDIALS_ERROR(process %d): %s() failed - "
00986             "returned NULL pointer\n\n", id, funcname);
00987         return (1);
00988     }
00989
00990     /* Check if retval < 0 */
00991     else if (opt == 1) {
00992         retval = (int *)returnvalue;
00993         char *flagname = CVodeGetReturnFlagName(*retval);
00994         if (*retval < 0) {
00995             fprintf(stderr, "\nSUNDIALS_ERROR(process %d): %s() failed "
00996                 "with retval = %d: %s\n\n",
00997                 id, funcname, *retval, flagname);
00998             return (1);
00999         }
01000     }
01001
01002     /* Check if function returned NULL pointer - no memory allocated */
01003     else if (opt == 2 && returnvalue == nullptr) {
01004         fprintf(stderr,
01005             "\nMEMORY_ERROR(process %d): %s() failed - "
01006             "returned NULL pointer\n\n", id, funcname);
01007         return (1);
01008     }
01009
01010     return (0);
01011 }
```

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

Here is the caller graph for this function:



6.12.2.2 errorKill()

```
void errorKill (
    const std::string & errorMessage )
```

helper function for error messages

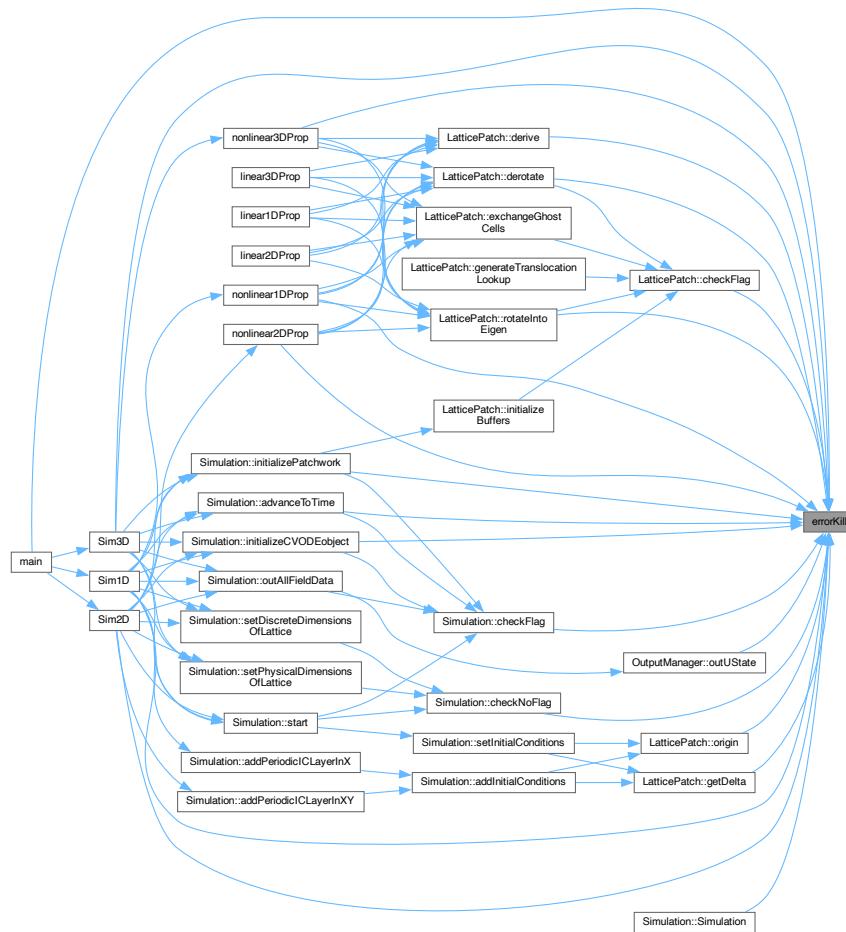
helper function for error messages

Definition at line 955 of file [LatticePatch.cpp](#).

```
00955
00956     int my_prc=0;
00957 #if defined(_MPI)
00958     MPI_Comm_rank(MPI_COMM_WORLD,&my_prc);
00959 #endif
00960     if (my_prc==0) {
00961         std::cerr << std::endl << "Error: " << errorMessage
00962         << "\nAborting..." << std::endl;
00963 #if defined(_MPI)
00964     MPI_Abort(MPI_COMM_WORLD, 1);
00965 #else
00966     exit(1);
00967 #endif
00968     return;
00969 }
00970 }
```

Referenced by [Simulation::advanceToTime\(\)](#), [LatticePatch::checkFlag\(\)](#), [Simulation::checkFlag\(\)](#), [Simulation::checkNoFlag\(\)](#), [LatticePatch::derive\(\)](#), [LatticePatch::derotate\(\)](#), [LatticePatch::getDelta\(\)](#), [Simulation::initializeCVODEobject\(\)](#), [Simulation::initializePatchwork\(\)](#), [main\(\)](#), [nonlinear1DProp\(\)](#), [nonlinear2DProp\(\)](#), [nonlinear3DProp\(\)](#), [LatticePatch::origin\(\)](#), [OutputManager::outUState\(\)](#), [LatticePatch::rotateIntoEigen\(\)](#), [Sim1D\(\)](#), [Sim2D\(\)](#), [Sim3D\(\)](#), and [Simulation::Simulation\(\)](#).

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 54 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 47 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 51 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 53 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 52 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 #if defined(_MPI)
00022 #include <mpi.h>
00023 #endif
00024 #if defined(_OPENMP)
00025 #include <omp.h>
00026 #endif
00027
00028 // Sundials
00029 #include <cvode/cvode.h>           /* prototypes for CVODE fcts. */
00030 #include <sundials/sundials_types.h> /* definition of type sunrealtype */
00031 #if defined(_MPI)
00032 #include <nvector/nvector_parallel.h> /* definition of MPI N_Vector */
00033 #if defined(_OPENMP)
00034 #include <nvector/nvector_openmp.h>   /* definition of OpenMP N_Vector */
00035 #include <nvector/nvector_mpiplusx.h> /* definition of MPI+X N_Vector */
00036 #endif
00037 #elif defined(_OPENMP)
00038 #include <nvector/nvector_openmp.h>   /* definition of OpenMP N_Vector */
00039 #else
00040 #include <nvector/nvector_serial.h>    /* definition of standard N_Vector */
00041 #endif
00042
00043 // stencils
00044 #include "DerivationStencils.h"
00045
00046 /// lattice construction checking flag
00047 constexpr unsigned int FLatticeDimensionSet = 0x01;
00048
00049 /**
00050 ** lattice patch construction checking flag */
00051 constexpr unsigned int FLatticePatchSetUp = 0x01;
00052 constexpr unsigned int TranslocationLookupSetUp = 0x02;
00053 constexpr unsigned int GhostLayersInitialized = 0x04;
00054 constexpr unsigned int BuffersInitialized = 0x08;
00055 /**
00056 /** @brief Lattice class for the construction of the enveloping discrete
00057 * simulation space */
00058 class Lattice {
00059 private:
00060     /// physical size of the lattice in x-direction
00061     sunrealtype tot_lx;
```

```

00063     /// physical size of the lattice in y-direction
00064     sunrealtype tot_ly;
00065     /// physical size of the lattice in z-direction
00066     sunrealtype tot_lz;
00067     /// number of points in x-direction
00068     sunindextype tot_nx;
00069     /// number of points in y-direction
00070     sunindextype tot_ny;
00071     /// number of points in z-direction
00072     sunindextype tot_nz;
00073     /// total number of lattice points
00074     sunindextype tot_noP;
00075     /// dimension of each data point set once and for all
00076     static constexpr int dataPointDimension = 6;
00077     /// number of lattice points times data dimension of each point
00078     sunindextype tot_noDP;
00079     /// physical distance between lattice points in x-direction
00080     sunrealtype dx;
00081     /// physical distance between lattice points in y-direction
00082     sunrealtype dy;
00083     /// physical distance between lattice points in z-direction
00084     sunrealtype dz;
00085     /// stencil order
00086     const int stencilOrder;
00087     /// required width of ghost layers (depends on the stencil order)
00088     const int ghostLayerWidth;
00089     /// lattice status flags
00090     unsigned int statusFlags;
00091
00092 public:
00093 #if defined(_MPI)
00094     /// number of MPI processes
00095     int n_prc;
00096     /// process number
00097     int my_prc;
00098     /// personal communicator of the lattice
00099     MPI_Comm comm;
00100    /// function to create and deploy the cartesian communicator
00101    void initializeCommunicator(const int Nx, const int Ny,
00102                               const int Nz, const bool per);
00103 #else
00104     /// number of processes
00105     static constexpr int n_prc = 1;
00106     /// process number
00107     static constexpr int my_prc = 0;
00108     // communicator as (null) pointer
00109     void* comm;
00110 #endif
00111     /// default construction
00112     Lattice(const int StO);
00113     /// SUNContext object
00114     SUNContext sunctx;
00115     /// component function for resizing the discrete dimensions of the lattice
00116     void setDiscreteDimensions(const sunindextype _nx,
00117                                const sunindextype _ny, const sunindextype _nz);
00118     /// component function for resizing the physical size of the lattice
00119     void setPhysicalDimensions(const sunrealtype _lx,
00120                               const sunrealtype _ly, const sunrealtype _lz);
00121     /**@{
00122     /** @brief getter function */
00123     [[nodiscard]] const sunrealtype &get_tot_lx() const { return tot_lx; }
00124     [[nodiscard]] const sunrealtype &get_tot_ly() const { return tot_ly; }
00125     [[nodiscard]] const sunrealtype &get_tot_lz() const { return tot_lz; }
00126     [[nodiscard]] const sunindextype &get_tot_nx() const { return tot_nx; }
00127     [[nodiscard]] const sunindextype &get_tot_ny() const { return tot_ny; }
00128     [[nodiscard]] const sunindextype &get_tot_nz() const { return tot_nz; }
00129     [[nodiscard]] const sunindextype &get_tot_noP() const { return tot_noP; }
00130     [[nodiscard]] const sunindextype &get_tot_noDP() const { return tot_noDP; }
00131     [[nodiscard]] const sunrealtype &get_dx() const { return dx; }
00132     [[nodiscard]] const sunrealtype &get_dy() const { return dy; }
00133     [[nodiscard]] const sunrealtype &get_dz() const { return dz; }
00134     [[nodiscard]] constexpr int get_dataPointDimension() const {
00135         return dataPointDimension;
00136     }
00137     [[nodiscard]] const int &get_stencilOrder() const { return stencilOrder; }
00138     [[nodiscard]] const int &get_ghostLayerWidth() const {
00139         return ghostLayerWidth;
00140     }
00141     /**@}
00142 };
00143
00144 /** @brief LatticePatch class for the construction of the patches in the
00145 * enveloping lattice */
00146 class LatticePatch {
00147 private:
00148     /// origin of the patch in physical space; x-coordinate
00149     sunrealtype x0;

```

```

00150  /// origin of the patch in physical space; y-coordinate
00151  surrealtype y0;
00152  /// origin of the patch in physical space; z-coordinate
00153  surrealtype z0;
00154  /// inner position of lattice-patch in the lattice patchwork; x-points
00155  sunindextype LIx;
00156  /// inner position of lattice-patch in the lattice patchwork; y-points
00157  sunindextype LIy;
00158  /// inner position of lattice-patch in the lattice patchwork; z-points
00159  sunindextype LIz;
00160  /// physical size of the lattice-patch in the x-dimension
00161  surrealtype lx;
00162  /// physical size of the lattice-patch in the y-dimension
00163  surrealtype ly;
00164  /// physical size of the lattice-patch in the z-dimension
00165  surrealtype lz;
00166  /// number of points in the lattice patch in the x-dimension
00167  sunindextype nx;
00168  /// number of points in the lattice patch in the y-dimension
00169  sunindextype ny;
00170  /// number of points in the lattice patch in the z-dimension
00171  sunindextype nz;
00172  /// physical distance between lattice points in x-direction
00173  surrealtype dx;
00174  /// physical distance between lattice points in y-direction
00175  surrealtype dy;
00176  /// physical distance between lattice points in z-direction
00177  surrealtype dz;
00178  /// lattice patch status flags
00179  unsigned int statusFlags;
00180  /// pointer to the enveloping lattice
00181  const Lattice *envelopeLattice;
00182  /// aid (auxilliarily) vector including ghost cells to compute the derivatives
00183  std::vector<surrealtype> uAux;
00184  /**
00185  ** translocation lookup table */
00186  std::vector<sunindextype> uTox, uToy, uToz, xTou, yTou, zTou;
00187  /**
00188  /**
00189  ** buffer to save spatial derivative values */
00190  std::vector<surrealtype> buffX, buffY, buffZ;
00191  /**
00192  /**
00193  ** buffer for passing ghost cell data */
00194  std::vector<surrealtype> ghostCellLeft, ghostCellRight, ghostCellLeftToSend,
00195  ghostCellRightToSend, ghostCellsToSend, ghostCells;
00196  /**
00197  /**
00198  ** ghost cell translocation lookup table */
00199  std::vector<sunindextype> lgcTox, rgcTox, lgcToy, rgcToy, lgcToz, rgcToz;
00200  /**
00201  /**
00202  /** Rotate and translocate an input array according to a lookup into an output
00203  * array */
00204  inline void rotateToX(surrealtype *outArray, const surrealtype *inArray,
00205  const std::vector<sunindextype> &lookup);
00206  inline void rotateToY(surrealtype *outArray, const surrealtype *inArray,
00207  const std::vector<sunindextype> &lookup);
00208  inline void rotateToZ(surrealtype *outArray, const surrealtype *inArray,
00209  const std::vector<sunindextype> &lookup);
00210 /**
00211 public:
00212  /// ID of the LatticePatch, corresponds to process number (for debugging)
00213  int ID;
00214  /// NVector for saving field components u=(E,B) in lattice points
00215  N_Vector uLocal, u;
00216  /// NVector for saving temporal derivatives of the field data
00217  N_Vector duLocal, du;
00218  /// pointer to field data
00219  surrealtype *uData;
00220  /// pointer to time-derivative data
00221  surrealtype *duData;
00222  /// pointer to auxiliary data vector
00223  surrealtype *uAuxData;
00224 /**
00225  ** pointer to halo data */
00226  surrealtype *gCLData, *gCRData;
00227 /**
00228  ** pointer to spatial derivative data buffers
00229  std::array<surrealtype *, 3> buffData;
00230  /// constructor setting up a default first lattice patch
00231  LatticePatch();
00232  /// destructor freeing parallel vectors
00233  ~LatticePatch();
00234  /// friend function for creating the patchwork slicing of the overall lattice
00235  friend int generatePatchwork(const Lattice &envelopeLattice,
00236                                LatticePatch &patchToMold, const int DLx,
```

```

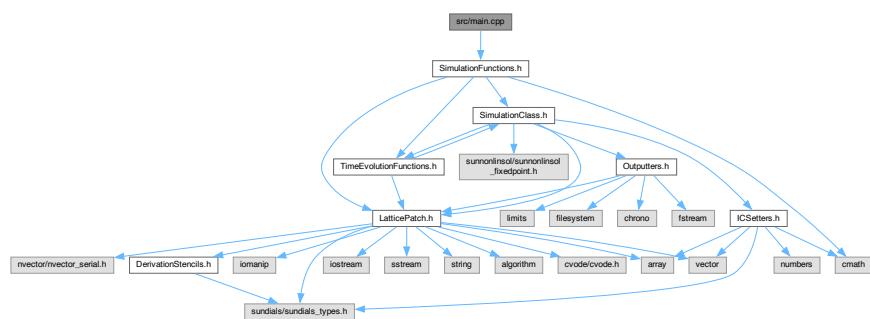
00237             const int DLy, const int DLz);
00238     /// function to get the discrete size of the LatticePatch
00239     sunindextype discreteSize(int dir=0) const;
00240     /// function to get the origin of the patch
00241     sunrealtype origin(const int dir) const;
00242     /// function to get distance between points
00243     sunrealtype getDelta(const int dir) const;
00244     /// function to fill out the lookup tables for cache efficiency
00245     void generateTranslocationLookup();
00246     /// function to rotate u into Z-matrix eigenraum
00247     void rotateIntoEigen(const int dir);
00248     /// function to derotate uAux into dudata lattice direction of x
00249     void derotate(int dir, sunrealtype *buffOut);
00250     /// initialize buffers to save derivatives
00251     void initializeBuffers();
00252     /// function to exchange ghost cells
00253     void exchangeGhostCells(const int dir);
00254     /// function to derive the centered values in uAux and save them noncentered
00255     void derive(const int dir);
00256     /// function to check if a flag has been set; if not, abort
00257     void checkFlag(unsigned int flag) const;
00258 };
00259
00260 /// helper function for error messages
00261 void errorKill(const std::string & errorMessage);
00262
00263 /// helper function to check CVode errors
00264 int check_retval(void *returnvalue, const char *funcname, int opt, int id);
00265

```

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

- relative and absolute tolerances of the CVode solver
- accuracy order of the stencils in the range 1-13
- physical length of the lattice in meters
- number of lattice points
- periodic or vanishing boundary values
- included processes of the weak-field expansion, see [README.md](#)
- physical total simulation time
- discrete time steps
- output step multiples
- output in csv (c) or binary (b) format

Add electromagnetic waves.

A plane wave with

- wavevector (normalized to $1/\lambda$)
- amplitude/polarization
- phase shift

Another plane wave with

- wavevector (normalized to $1/\lambda$)
- amplitude/polarization
- phase shift

A Gaussian wave with

- wavevector (normalized to $1/\lambda$)
- polarization/amplitude
- shift from origin
- width
- phase shift

Another Gaussian with

- wavevector (normalized to $1/\lambda$)
- polarization/amplitude
- shift from origin
- width
- phase shift

A 2D simulation with specified

- relative and absolute tolerances of the CVode solver
- accuracy order of the stencils in the range 1-13
- physical length of the lattice in the given dimensions in meters
- number of lattice points per dimension
- slicing of discrete dimensions into patches
- periodic or vanishing boundary values
- included processes of the weak-field expansion, see [README.md](#)
- physical total simulation time
- discrete time steps
- output step multiples
- output in csv (c) or binary (b) format

Add electromagnetic waves.

A plane wave with

- wavevector (normalized to $1/\lambda$)
- amplitude/polarization
- phase shift

Another plane wave with

- wavevector

- amplitude/polarization
- phase shift

A Gaussian wave with

- center it approaches
- normalized direction *from* which the wave approaches the center
- amplitude
- polarization rotation from TE-mode (z-axis)
- taille
- Rayleigh length

the wavelength is determined by the relation $\lambda = \pi * w_0^2 / z_R$

- beam center
- beam length

Another Gaussian wave with

- center it approaches
- normalized direction from which the wave approaches the center
- amplitude
- polarization rotation fom TE-mode (z-axis)
- taille
- Rayleigh length
- beam center
- beam length

A 3D simulation with specified

- relative and absolute tolerances of the CVode solver
- accuracy order of the stencils in the range 1-13
- physical dimensions in meters
- number of lattice points in any dimension
- slicing of discrete dimensions into patches
- perodic or non-periodic boundaries
- processes of the weak-field expansion, see [README.md](#)
- physical total simulation time

- discrete time steps
- output step multiples
- output in csv (c) or binary (b) format

Add electromagnetic waves.

A plane wave with

- wavevector (normalized to $1/\lambda$)
- amplitude/polarization
- phase shift

Another plane wave with

- wavevector
- amplitude/polarization
- phase shift

A Gaussian wave with

- center it approaches
- normalized direction *from* which the wave approaches the center
- amplitude
- polarization rotation from TE-mode (z-axis)
- taille
- Rayleigh length

the wavelength is determined by the relation $\lambda = \pi * w_0^2 / z_R$

- beam center
- beam length

Another Gaussian wave with

- center it approaches
- normalized direction from which the wave approaches the center
- amplitude
- polarization rotation from TE-mode (z-axis)
- taille
- Rayleigh length

- beam center
- beam length

Definition at line 9 of file main.cpp.

```

00010 {
00011 #if defined(_MPI)
00012     // Initialize MPI environment
00013     int provided;
00014     MPI_Init_thread(&argc, &argv, MPI_THREAD_SINGLE, &provided);
00015 #endif
00016     // Check if all CL args are provided
00017     if(argc != 92) {
00018         errorKill("\nPlease check the command-line arguments.\n");
00019     }
00020     /** Determine the output directory.
00021     * A "SimResults" folder will be created if non-existent
00022     * with a subdirectory named in the identifier format
00023     * "yy-mm-dd_hh-MM-ss" that contains the csv files      */
00024     string outputDirectory = argv[1]; // command line
00025
00026     if(!filesystem::exists(outputDirectory)) {
00027         errorKill("\nOutput directory nonexistent.\n");
00028     }
00029
00030     //----- BEGIN OF CONFIGURATION -----//
00031
00032     if (stoi(argv[2]) == 1) {
00033         ////////////// -- 1D -- ///////////
00034         /** A 1D simulation with specified */
00035         array<sunrealtype,2> CVodeTolerances={stod(argv[3]),stod(argv[4])}; // - relative and absolute
00036         tolerances of the CVode solver
00037         int StencilOrder=stoi(argv[5]); // - accuracy order of the stencils in the range 1-13
00038         unrealtype physical_sidelength=stod(argv[6]); // - physical length of the lattice in meters
00039         sunintdtype latticepoints=stoi(argv[9]); // - number of lattice points
00040         constexpr bool periodic=true; // - periodic or vanishing boundary values
00041         int processOrder=stoi(argv[15]); // - included processes of the weak-field expansion, see
00042             README.md
00043         unrealtype simulationTime=stod(argv[16]); // - physical total simulation time
00044         int numberOfSteps=stoi(argv[17]); // - discrete time steps
00045         int outputStep=stoi(argv[18]); // - output step multiples
00046         char outputStyle=*(argv[19]); // - output in csv (c) or binary (b) format
00047
00048         // Add electromagnetic waves.
00049         vector<planewave> planewaves;
00050         vector<gaussian1D> Gaussians1D;
00051         if (stoi(argv[20])) { // if use of plane waves
00052             planewave plane1; // A plane wave with
00053             plane1.k = {stod(argv[22]),stod(argv[23]),stod(argv[24])}; // - wavevector (normalized to
00054             // f$ 1/\lambda \f$)
00055             plane1.p = {stod(argv[25]),stod(argv[26]),stod(argv[27])}; // - amplitude/polarization
00056             plane1.phi = {stod(argv[28]),stod(argv[29]),stod(argv[30])}; // - phase shift
00057             planewave plane2; // Another plane wave with
00058             plane2.k = {stod(argv[31]),stod(argv[32]),stod(argv[33])}; // - wavevector (normalized to
00059             // f$ 1/\lambda \f$)
00060             plane2.p = {stod(argv[34]),stod(argv[35]),stod(argv[36])}; // - amplitude/polarization
00061             plane2.phi = {stod(argv[37]),stod(argv[38]),stod(argv[39])}; // - phase shift
00062             if (stoi(argv[21])>2) {
00063                 cerr<<"You can only use up to two plane waves." << endl;
00064                 exit(1);
00065             }
00066             if (stoi(argv[21])>0) planewaves.emplace_back(plane1);
00067             if (stoi(argv[21])==2) planewaves.emplace_back(plane2);
00068
00069         if (stoi(argv[40])) { // if use of Gaussians
00070             gaussian1D gauss1; // A Gaussian wave with
00071             gauss1.k = {stod(argv[42]),stod(argv[43]),stod(argv[44])}; // - wavevector (normalized to
00072             // f$ 1/\lambda \f$)
00073             gauss1.p = {stod(argv[45]),stod(argv[46]),stod(argv[47])}; // - polarization/amplitude
00074             gauss1.x0 = {stod(argv[48]),stod(argv[49]),stod(argv[50])}; // - shift from origin
00075             gauss1.phig = stod(argv[51]); // - width
00076             gauss1.phi = {stod(argv[52]),stod(argv[53]),stod(argv[54])}; // - phase shift
00077             gaussian1D gauss2; // Another Gaussian with
00078             gauss2.k = {stod(argv[55]),stod(argv[56]),stod(argv[57])}; // - wavevector (normalized to
00079             // f$ 1/\lambda \f$)
00080             gauss2.p = {stod(argv[58]),stod(argv[59]),stod(argv[60])}; // - polarization/amplitude
00081             gauss2.x0 = {stod(argv[61]),stod(argv[62]),stod(argv[63])}; // - shift from origin
00082             gauss2.phig = stod(argv[64]); // - width
00083             gauss2.phi = {stod(argv[65]),stod(argv[66]),stod(argv[67])}; // - phase shift
00084             if (stoi(argv[41])>2) {
00085                 cerr<<"You can only use up to two Gaussian pulses." << endl;
00086                 exit(1);
00087             }
00088             if (stoi(argv[41])>0) Gaussians1D.emplace_back(gauss1);
00089             if (stoi(argv[41])==2) Gaussians1D.emplace_back(gauss2);

```

```

00084     }
00085     ///////////////////////////////////////////////////////////////////
00086     int *interactions = &processOrder;
00087     Sim1D(CVodeTolerances,StencilOrder,physical_sidelength,latticepoints,
00088             periodic,interactions,simulationTime,numberOfSteps,
00089             outputDirectory,outputStep,outputStyle,
00090             planewaves,Gaussians1D);
00091 }
00092 ///////////////////////////////////////////////////////////////////
00093
00094
00095 if (stoi(argv[2]) == 2) {
00096 ////////////////////////////////////////////////////////////////// -- 2D -- //////////////////////////////////////////////////////////////////
00097 /** A 2D simulation with specified */
00098 array<sunrealtype,2> CVodeTolerances={stod(argv[3]),stod(argv[4])}; // - relative and absolute
tolerances of the CVode solver
00099 int StencilOrder=stoi(argv[5]); // - accuracy order of the stencils in the range 1-13
00100 array<sunrealtype,2> physical_sidelengths={stod(argv[6]),stod(argv[7])}; // - physical length of
the lattice in the given dimensions in meters
00101 array<sunindextype,2> latticepoints_per_dim={stoi(argv[9]),stoi(argv[10])}; // - number of
lattice points per dimension
00102 array<int,2> patches_per_dim={stoi(argv[12]),stoi(argv[13])}; // - slicing of discrete
dimensions into patches
00103 constexpr bool periodic=true; // - periodic or vanishing boundary values
00104 int processOrder=stoi(argv[15]); // - included processes of the weak-field expansion, see
README.md
00105 unrealtype simulationTime=stod(argv[16]); // - physical total simulation time
00106 int numberOfSteps=stoi(argv[17]); // - discrete time steps
00107 int outputStep=stoi(argv[18]); // - output step multiples
00108 char outputStyle=*(argv[19]); // - output in csv (c) or binary (b) format
00109
00110 // Add electromagnetic waves.
00111 vector<planewave> planewaves;
00112 vector<gaussian2D> Gaussians2D;
00113 if (stoi(argv[20])) { // if use of plane waves
00114     planewave plane1; // A plane wave with
00115     plane1.k = {stod(argv[22]),stod(argv[23]),stod(argv[24])}; // - wavevector (normalized to
\f$ 1/\lambda \f$)
00116     plane1.p = {stod(argv[25]),stod(argv[26]),stod(argv[27])}; // - amplitude/polarization
00117     plane1.phi = {stod(argv[28]),stod(argv[29]),stod(argv[30])}; // - phase shift
00118     planewave plane2; // Another plane wave with
00119     plane2.k = {stod(argv[31]),stod(argv[32]),stod(argv[33])}; // - wavevector
00120     plane2.p = {stod(argv[34]),stod(argv[35]),stod(argv[36])}; // - amplitude/polarization
00121     plane2.phi = {stod(argv[37]),stod(argv[38]),stod(argv[39])}; // - phase shift
00122     if (stoi(argv[21])>2) {
00123         cerr<<"You can only use up to two plane waves." << endl;
00124         exit(1);
00125     }
00126     if (stoi(argv[21])>0) planewaves.emplace_back(plane1);
00127     if (stoi(argv[21])==2) planewaves.emplace_back(plane2);
00128 }
00129 if (stoi(argv[68])) { // if use of Gaussians
00130     gaussian2D gauss1; // A Gaussian wave with
00131     gauss1.x0 = {stod(argv[70]),stod(argv[71])}; // - center it approaches
00132     gauss1.axis = {stod(argv[73]),stod(argv[74])}; // - normalized direction _from_ which the
wave approaches the center
00133     gauss1.amp = stod(argv[75]); // - amplitude
00134     gauss1.phi_p = atan(stod(argv[76])); // - polarization rotation from TE-mode (z-axis)
00135     gauss1.w0 = stod(argv[77]); // - taille
00136     gauss1.zr = stod(argv[78]); // - Rayleigh length
00137     // the wavelength is determined by the relation \f$ \lambda = \pi * w_0^2 / z_R \f$
00138     gauss1.ph0 = stod(argv[79]); // - beam center
00139     gauss1.phA = stod(argv[80]); // - beam length
00140     gaussian2D gauss2; // Another Gaussian wave with
00141     gauss2.x0 = {stod(argv[81]),stod(argv[82])}; // - center it approaches
00142     gauss2.axis = {stod(argv[84]),stod(argv[85])}; // - normalized direction from which the wave
approaches the center
00143     gauss2.amp = stod(argv[86]); // - amplitude
00144     gauss2.phi_p = atan(stod(argv[87])); // - polarization rotation from TE-mode (z-axis)
00145     gauss2.w0 = stod(argv[88]); // - taille
00146     gauss2.zr = stod(argv[89]); // - Rayleigh length
00147     gauss2.ph0 = stod(argv[90]); // - beam center
00148     gauss2.phA = stod(argv[91]); // - beam length
00149     if (stoi(argv[69])>2) {
00150         cerr<<"You can only use up to two Gaussian pulses." << endl;
00151         exit(1);
00152     }
00153     if (stoi(argv[69])>0) Gaussians2D.emplace_back(gauss1);
00154     if (stoi(argv[69])==2) Gaussians2D.emplace_back(gauss2);
00155 }
00156 //////////////////////////////////////////////////////////////////
00157 int * interactions = &processOrder;
00158 Sim2D(CVodeTolerances,StencilOrder,physical_sidelengths,
00159             latticepoints_per_dim,patches_per_dim,periodic,interactions,
00160             simulationTime,numberOfSteps,outputDirectory,outputStep,
00161             outputStyle,planewaves,Gaussians2D);
00162 }

```

```

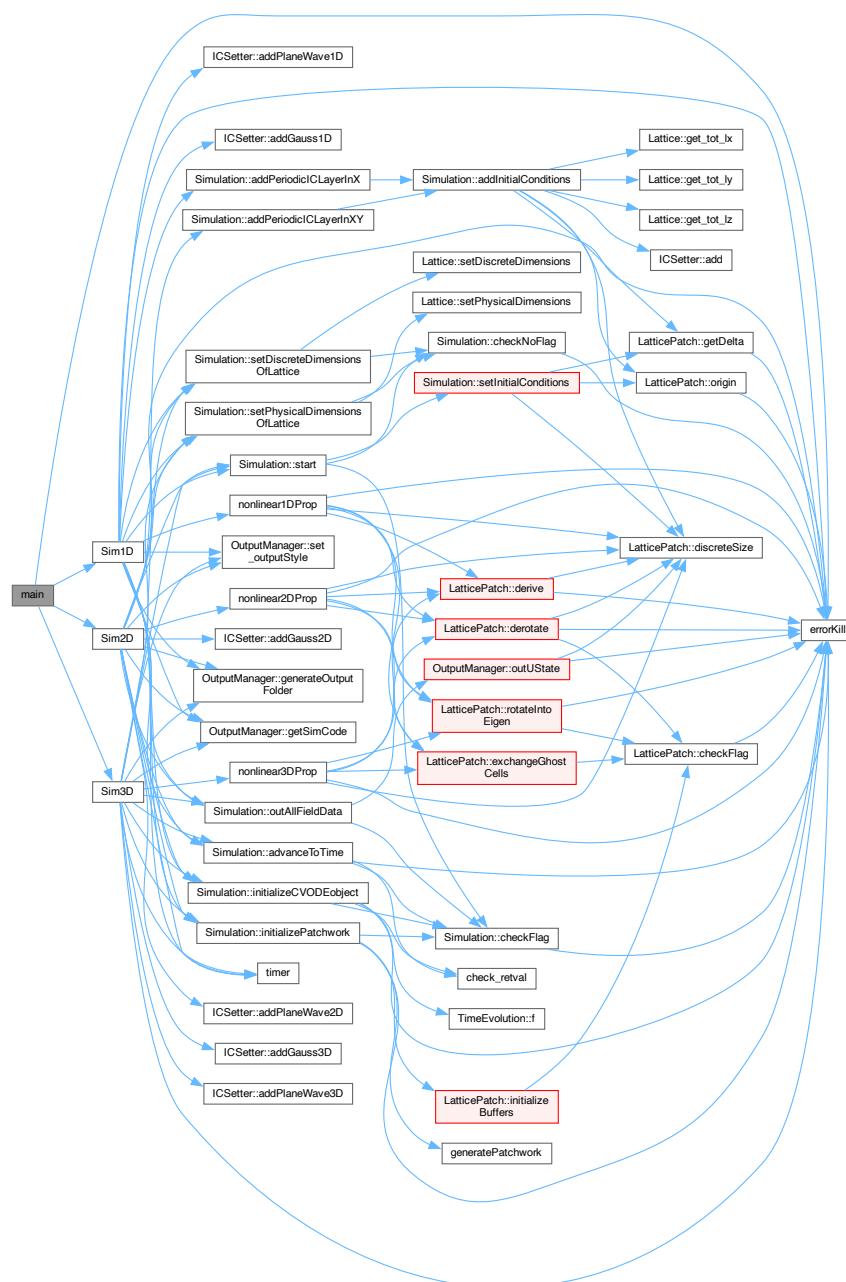
00163 ///////////////////////////////////////////////////////////////////
00164
00165
00166 if (stoi(argv[2]) == 3) {
00167 ////////////////////////////////////////////////////////////////// -- 3D -- //////////////////////////////////////////////////////////////////
00168 /** A 3D simulation with specified */
00169 array<sunrealtype,2> CVodeTolerances={stod(argv[3]),stod(argv[4])}; /// - relative and absolute
00170 tolerances of the CVode solver
00171 int StencilOrder=stoi(argv[5]); /// - accuracy order of the stencils in the range 1-13
00172 array<sunrealtype,3> physical_sidelengths={stod(argv[6]),stod(argv[7]),stod(argv[8])}; /// -
00173 physical dimensions in meters
00174 array<sunindextype,3> latticepoints_per_dim={stoi(argv[9]),stoi(argv[10]),stoi(argv[11])}; /// -
00175 number of lattice points in any dimension
00176 array<int,3> patches_per_dim={stoi(argv[12]),stoi(argv[13]),stoi(argv[14])}; /// - slicing of
00177 discrete dimensions into patches
00178 constexpr bool periodic=true; /// - perodic or non-periodic boundaries
00179 int processOrder=stoi(argv[15]); /// - processes of the weak-field expansion, see README.md
00180 unrealtype simulationTime=stod(argv[16]); /// - physical total simulation time
00181 int numberOfSteps=stoi(argv[17]); /// - discrete time steps
00182 int outputStep=stoi(argv[18]); /// - output step multiples
00183 char outputStyle==*(argv[19]); /// - output in csv (c) or binary (b) format
00184
00185 // Add electromagnetic waves.
00186 vector<planewave> planewaves;
00187 vector<gaussian3D> Gaussians3D;
00188 if (stoi(argv[20])) { // if use of plane waves
00189     planewave plane1; // A plane wave with
00190     plane1.k = {stod(argv[22]),stod(argv[23]),stod(argv[24])}; /// - wavevector (normalized to
00191     \f$ 1/\lambda \f$)
00192     plane1.p = {stod(argv[25]),stod(argv[26]),stod(argv[27])}; /// - amplitude/polarization
00193     plane1.phi = {stod(argv[28]),stod(argv[29]),stod(argv[30])}; /// - phase shift
00194     planewave plane2; // Another plane wave with
00195     plane2.k = {stod(argv[31]),stod(argv[32]),stod(argv[33])}; /// - wavevector
00196     plane2.p = {stod(argv[34]),stod(argv[35]),stod(argv[36])}; /// - amplitude/polarization
00197     plane2.phi = {stod(argv[37]),stod(argv[38]),stod(argv[39])}; /// - phase shift
00198     if (stoi(argv[21])>2) {
00199         cerr<<"You can only use up to two plane waves." << endl;
00200         exit(1);
00201     }
00202     if (stoi(argv[21])>0) planewaves.emplace_back(plane1);
00203     if (stoi(argv[21])==2) planewaves.emplace_back(plane2);
00204 }
00205 if (stoi(argv[68])) { // if use of Gaussians
00206     gaussian3D gauss1; // A Gaussian wave with
00207     gauss1.x0 = {stod(argv[70]),stod(argv[71]),stod(argv[72])}; /// - center it approaches
00208     gauss1.axis = {stod(argv[73]),stod(argv[74]),0}; /// - normalized direction _from_ which the
00209     wave approaches the center
00210     gauss1.amp = stod(argv[75]); /// - amplitude
00211     gauss1.phip = atan(stod(argv[76])); /// - polarization rotation from TE-mode (z-axis)
00212     gauss1.w0 = stod(argv[77]); /// - taille
00213     gauss1.zr = stod(argv[78]); /// - Rayleigh length
00214     // the wavelength is determined by the relation \f$ \lambda = \pi * w_0^2 / z_R \f$
00215     gauss1.ph0 = stod(argv[79]); /// - beam center
00216     gauss1.phA = stod(argv[80]); /// - beam length
00217     gaussian3D gauss2; // Another Gaussian wave with
00218     gauss2.x0 = {stod(argv[81]),stod(argv[82]),stod(argv[83])}; /// - center it approaches
00219     gauss2.axis = {stod(argv[84]),stod(argv[85]),0}; /// - normalized direction from which the
00220     wave approaches the center
00221     gauss2.amp = stod(argv[86]); /// - amplitude
00222     gauss2.phip = atan(stod(argv[87])); /// - polarization rotation from TE-mode (z-axis)
00223     gauss2.w0 = stod(argv[88]); /// - taille
00224     gauss2.zr = stod(argv[89]); /// - Rayleigh length
00225     gauss2.ph0 = stod(argv[90]); /// - beam center
00226     gauss2.phA = stod(argv[91]); /// - beam length
00227     if (stoi(argv[69])>2) {
00228         cerr<<"You can only use up to two Gaussian pulses." << endl;
00229         exit(1);
00230     }
00231     if (stoi(argv[69])>0) Gaussians3D.emplace_back(gauss1);
00232     if (stoi(argv[69])==2) Gaussians3D.emplace_back(gauss2);
00233 }
00234 //////////////////////////////////////////////////////////////////
00235
00236 //----- END OF CONFIGURATION -----//
00237
00238 #if defined(_MPI)
00239 // Finalize MPI environment
00240 MPI_Finalize();
00241 #endif
00242

```

```
00243     return 0;
00244 }
```

References gaussian2D::amp, gaussian3D::amp, gaussian2D::axis, gaussian3D::axis, errorKill(), planewave::k, gaussian1D::k, planewave::p, gaussian1D::p, gaussian2D::ph0, gaussian3D::ph0, gaussian2D::phA, gaussian3D::phA, planewave::phi, gaussian1D::phi, gaussian1D::phig, gaussian2D::phip, gaussian3D::phip, Sim1D(), Sim2D(), Sim3D(), gaussian2D::w0, gaussian3D::w0, gaussian1D::x0, gaussian2D::x0, gaussian3D::x0, gaussian2D::zr, and gaussian3D::zr.

Here is the call graph for this function:



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 #if defined(_MPI)
00012     // Initialize MPI environment
00013     int provided;
00014     MPI_Init_thread(&argc, &argv, MPI_THREAD_SINGLE, &provided);
00015 #endif
00016     // Check if all CL args are provided
00017     if(argc != 92) {
00018         errorKill("\nPlease check the command-line arguments.\n");
00019     }
00020     /** Determine the output directory.
00021      * A "SimResults" folder will be created if non-existent
00022      * with a subdirectory named in the identifier format
00023      * "yy-mm-dd_hh-MM-ss" that contains the csv files      */
00024     string outputDirectory = argv[1]; // command line
00025
00026     if(!filesystem::exists(outputDirectory)) {
00027         errorKill("\nOutput directory nonexistent.\n");
00028     }
00029
00030     //----- BEGIN OF CONFIGURATION -----//
00031
00032     if (stoi(argv[2]) == 1) {
00033         ////////////// -- 1D --
00034         /** A 1D simulation with specified */
00035         array<sunrealtype,2> CVodeTolerances={stod(argv[3]),stod(argv[4])}; // - relative and absolute
00036         tolerances of the CVode solver
00037         int StencilOrder=stoi(argv[5]); // - accuracy order of the stencils in the range 1-13
00038         unrealtype physical_sidelength=stod(argv[6]); // - physical length of the lattice in meters
00039         sunidextype latticepoints=stoi(argv[9]); // - number of lattice points
00040         constexpr bool periodic=true; // - periodic or vanishing boundary values
00041         int processOrder=stoi(argv[15]); // - included processes of the weak-field expansion, see
00042         README.md
00043         unrealtype simulationTime=stod(argv[16]); // - physical total simulation time
00044         int numberOfSteps=stoi(argv[17]); // - discrete time steps
00045         int outputStep=stoi(argv[18]); // - output step multiples
00046         char outputStyle=*(argv[19]); // - output in csv (c) or binary (b) format
00047
00048     // Add electromagnetic waves.
00049     vector<planewave> planewaves;
00050     vector<gaussian1D> Gaussians1D;
00051     if (stoi(argv[20])) { // if use of plane waves
00052         planewave plane1; // A plane wave with
00053         plane1.k = {stod(argv[22]),stod(argv[23]),stod(argv[24])}; // - wavevector (normalized to
00054         //f$ 1/\lambda \f$)
00055         plane1.p = {stod(argv[25]),stod(argv[26]),stod(argv[27])}; // - amplitude/polarization
00056         plane1.phi = {stod(argv[28]),stod(argv[29]),stod(argv[30])}; // - phase shift
00057         planewave plane2; // Another plane wave with
00058         plane2.k = {stod(argv[31]),stod(argv[32]),stod(argv[33])}; // - wavevector (normalized to
00059         //f$ 1/\lambda \f$)
00060         plane2.p = {stod(argv[34]),stod(argv[35]),stod(argv[36])}; // - amplitude/polarization
00061         plane2.phi = {stod(argv[37]),stod(argv[38]),stod(argv[39])}; // - phase shift
00062         if (stoi(argv[21])>2) {
00063             cerr<<"You can only use up to two plane waves." << endl;
00064             exit(1);
00065         }
00066         if (stoi(argv[21])>0) planewaves.emplace_back(plane1);
00067         if (stoi(argv[21])==2) planewaves.emplace_back(plane2);
00068
00069     if (stoi(argv[40])) { // if use of Gaussians
00070         gaussian1D gauss1; // A Gaussian wave with
00071         gauss1.k = {stod(argv[42]),stod(argv[43]),stod(argv[44])}; // - wavevector (normalized to
00072         //f$ 1/\lambda \f$)
00073         gauss1.p = {stod(argv[45]),stod(argv[46]),stod(argv[47])}; // - polarization/amplitude
00074         gauss1.x0 = {stod(argv[48]),stod(argv[49]),stod(argv[50])}; // - shift from origin
00075         gauss1.phig = stod(argv[51]); // - width
00076         gauss1.phi = {stod(argv[52]),stod(argv[53]),stod(argv[54])}; // - phase shift
00077         gaussian1D gauss2; // Another Gaussian with
00078         gauss2.k = {stod(argv[55]),stod(argv[56]),stod(argv[57])}; // - wavevector (normalized to
00079         //f$ 1/\lambda \f$)
00080         gauss2.p = {stod(argv[58]),stod(argv[59]),stod(argv[60])}; // - polarization/amplitude
00081         gauss2.x0 = {stod(argv[61]),stod(argv[62]),stod(argv[63])}; // - shift from origin
00082         gauss2.phig = stod(argv[64]); // - width
00083         gauss2.phi = {stod(argv[65]),stod(argv[66]),stod(argv[67])}; // - phase shift
00084         if (stoi(argv[41])>2) {
00085             cerr<<"You can only use up to two Gaussian pulses." << endl;
00086             exit(1);
00087         }

```

```

00082     if (stoi(argv[41])>0) Gaussians1D.emplace_back(gauss1);
00083     if (stoi(argv[41])==2) Gaussians1D.emplace_back(gauss2);
00084 }
00085 ///////////////////////////////////////////////////////////////////
00086 int *interactions = &processOrder;
00087 Sim1D(CVodeTolerances,StencilOrder,physical_sidelength,latticepoints,
00088         periodic,interactions,simulationTime,numberOfSteps,
00089         outputDirectory,outputStep,outputStyle,
00090         planewaves,Gaussians1D);
00091 }
00092 ///////////////////////////////////////////////////////////////////
00093
00094
00095 if (stoi(argv[2]) == 2) {
00096 ////////////////////////////////////////////////////////////////// -- 2D -- ///////////////////////////////////////////////////////////////////
00097 /* A 2D simulation with specified */
00098 array<sunrealtype,2> CVodeTolerances={stod(argv[3]),stod(argv[4])}; // - relative and absolute
00099 tolerances of the CVode solver
00100 int StencilOrder=stoi(argv[5]); // - accuracy order of the stencils in the range 1-13
00101 array<sunrealtype,2> physical_sidelengths={stod(argv[6]),stod(argv[7])}; // - physical length of
00102 the lattice in the given dimensions in meters
00103 array<sunindextype,2> latticepoints_per_dim={stoi(argv[9]),stoi(argv[10])}; // - number of
00104 lattice points per dimension
00105 array<int,2> patches_per_dim={stoi(argv[12]),stoi(argv[13])}; // - slicing of discrete
00106 dimensions into patches
00107 constexpr bool periodic=true; // - periodic or vanishing boundary values
00108 int processOrder=stoi(argv[15]); // - included processes of the weak-field expansion, see
00109 README.md
00110 unrealtype simulationTime=stod(argv[16]); // - physical total simulation time
00111 int numberOfSteps=stoi(argv[17]); // - discrete time steps
00112 int outputStep=stoi(argv[18]); // - output step multiples
00113 char outputStyle=*(argv[19]); // - output in csv (c) or binary (b) format
00114
00115 // Add electromagnetic waves.
00116 vector<planewave> planewaves;
00117 vector<gaussian2D> Gaussians2D;
00118 if (stoi(argv[20])) { // if use of plane waves
00119     planewave plane1; // A plane wave with
00120     plane1.k = {stod(argv[22]),stod(argv[23]),stod(argv[24])}; // - wavevector (normalized to
00121     \f$ 1/\lambda \f$)
00122     plane1.p = {stod(argv[25]),stod(argv[26]),stod(argv[27])}; // - amplitude/polarization
00123     plane1.phi = {stod(argv[28]),stod(argv[29]),stod(argv[30])}; // - phase shift
00124     planewave plane2; // Another plane wave with
00125     plane2.k = {stod(argv[31]),stod(argv[32]),stod(argv[33])}; // - wavevector
00126     plane2.p = {stod(argv[34]),stod(argv[35]),stod(argv[36])}; // - amplitude/polarization
00127     plane2.phi = {stod(argv[37]),stod(argv[38]),stod(argv[39])}; // - phase shift
00128     if (stoi(argv[21])>2) {
00129         cerr<<"You can only use up to two plane waves." << endl;
00130         exit(1);
00131     }
00132     if (stoi(argv[21])>0) planewaves.emplace_back(plane1);
00133     if (stoi(argv[21])==2) planewaves.emplace_back(plane2);
00134 }
00135 if (stoi(argv[68])) { // if use of Gaussians
00136     gaussian2D gauss1; // A Gaussian wave with
00137     gauss1.x0 = {stod(argv[70]),stod(argv[71])}; // - center it approaches
00138     gauss1.axis = {stod(argv[73]),stod(argv[74])}; // - normalized direction _from_ which the
00139     wave approaches the center
00140     gauss1.amp = stod(argv[75]); // - amplitude
00141     gauss1.phip = atan(stod(argv[76])); // - polarization rotation from TE-mode (z-axis)
00142     gauss1.w0 = stod(argv[77]); // - taille
00143     gauss1.zr = stod(argv[78]); // - Rayleigh length
00144     // the wavelength is determined by the relation \f$ \lambda = \pi * w_0^2 / z_R \f$
00145     gauss1.pho = stod(argv[79]); // - beam center
00146     gauss1.phA = stod(argv[80]); // - beam length
00147     gaussian2D gauss2; // Another Gaussian wave with
00148     gauss2.x0 = {stod(argv[81]),stod(argv[82])}; // - center it approaches
00149     gauss2.axis = {stod(argv[84]),stod(argv[85])}; // - normalized direction from which the wave
00150     approaches the center
00151     gauss2.amp = stod(argv[86]); // - amplitude
00152     gauss2.phip = atan(stod(argv[87])); // - polarization rotation from TE-mode (z-axis)
00153     gauss2.w0 = stod(argv[88]); // - taille
00154     gauss2.zr = stod(argv[89]); // - Rayleigh length
00155     gauss2.pho = stod(argv[90]); // - beam center
00156     gauss2.phA = stod(argv[91]); // - beam length
00157     if (stoi(argv[69])>2) {
00158         cerr<<"You can only use up to two Gaussian pulses." << endl;
00159         exit(1);
00160     }
00161     if (stoi(argv[69])>0) Gaussians2D.emplace_back(gauss1);
00162     if (stoi(argv[69])==2) Gaussians2D.emplace_back(gauss2);
00163 }
00164 ///////////////////////////////////////////////////////////////////
00165 int *interactions = &processOrder;
00166 Sim2D(CVodeTolerances,StencilOrder,physical_sidelengths,
00167         latticepoints_per_dim,patches_per_dim,periodic,interactions,
00168         simulationTime,numberOfSteps,outputDirectory,outputStep,
00169         outputStyle);

```

```

00161         outputStyle,planewaves,Gaussians2D);
00162     }
00163     ///////////////////////////////////////////////////////////////////
00164
00165
00166     if (stoi(argv[2]) == 3) {
00167     ////////////////////////////////////////////////////////////////// - 3D -- ///////////////////////////////////////////////////////////////////
00168     /** A 3D simulation with specified */
00169     array<sunrealtype,2> CVodeTolerances={stod(argv[3]),stod(argv[4])}; // - relative and absolute
00170     tolerances of the CVode solver
00171     int StencilOrder=stoi(argv[5]); // - accuracy order of the stencils in the range 1-13
00172     array<unrealtype,3> physical_sidelengths={stod(argv[6]),stod(argv[7]),stod(argv[8])}; // -
00173     physical dimensions in meters
00174     array<uninttype,3> latticepoints_per_dim={stoi(argv[9]),stoi(argv[10]),stoi(argv[11])}; // -
00175     number of lattice points in any dimension
00176     array<int,3> patches_per_dim={stoi(argv[12]),stoi(argv[13]),stoi(argv[14])}; // - slicing of
00177     discrete dimensions into patches
00178     constexpr bool periodic=true; // - periodic or non-periodic boundaries
00179     int processOrder=stoi(argv[15]); // - processes of the weak-field expansion, see README.md
00180     unrealtype simulationTime=stod(argv[16]); // - physical total simulation time
00181     int numberOfSteps=stoi(argv[17]); // - discrete time steps
00182     int outputStep=stoi(argv[18]); // - output step multiples
00183     char outputStyle=*(argv[19]); // - output in csv (c) or binary (b) format
00184
00185     // Add electromagnetic waves.
00186     vector<planewave> planewaves;
00187     vector<gaussian3D> Gaussians3D;
00188     if (stoi(argv[20])) { // if use of plane waves
00189         planewave plane1; // A plane wave with
00190         plane1.k = {stod(argv[22]),stod(argv[23]),stod(argv[24])}; // - wavevector (normalized to
00191         \f$ 1/\lambda \f$)
00192         plane1.p = {stod(argv[25]),stod(argv[26]),stod(argv[27])}; // - amplitude/polarization
00193         plane1.phi = {stod(argv[28]),stod(argv[29]),stod(argv[30])}; // - phase shift
00194         planewave plane2; // Another plane wave with
00195         plane2.k = {stod(argv[31]),stod(argv[32]),stod(argv[33])}; // - wavevector
00196         plane2.p = {stod(argv[34]),stod(argv[35]),stod(argv[36])}; // - amplitude/polarization
00197         plane2.phi = {stod(argv[37]),stod(argv[38]),stod(argv[39])}; // - phase shift
00198         if (stoi(argv[21])>2) {
00199             cerr<<"You can only use up to two plane waves." << endl;
00200             exit(1);
00201         }
00202         if (stoi(argv[21])>0) planewaves.emplace_back(plane1);
00203         if (stoi(argv[21])==2) planewaves.emplace_back(plane2);
00204     }
00205     if (stoi(argv[68])) { // if use of Gaussians
00206         gaussian3D gauss1; // A Gaussian wave with
00207         gauss1.x0 = {stod(argv[70]),stod(argv[71]),stod(argv[72])}; // - center it approaches
00208         gauss1.axis = {stod(argv[73]),stod(argv[74]),0}; // - normalized direction _from_ which the
00209         wave approaches the center
00210         gauss1.amp = stod(argv[75]); // - amplitude
00211         gauss1.phip = atan(stod(argv[76])); // - polarization rotation from TE-mode (z-axis)
00212         gauss1.w0 = stod(argv[77]); // - taille
00213         gauss1.zr = stod(argv[78]); // - Rayleigh length
00214         // the wavelength is determined by the relation \f$ \lambda = \pi * w_0^2 / z_R \f$
00215         gauss1.ph0 = stod(argv[79]); // - beam center
00216         gauss1.phA = stod(argv[80]); // - beam length
00217         gaussian3D gauss2; // Another Gaussian wave with
00218         gauss2.x0 = {stod(argv[81]),stod(argv[82]),stod(argv[83])}; // - center it approaches
00219         gauss2.axis = {stod(argv[84]),stod(argv[85]),0}; // - normalized direction from which the
00220         wave approaches the center
00221         gauss2.amp = stod(argv[86]); // - amplitude
00222         gauss2.phip = atan(stod(argv[87])); // - polarization rotation from TE-mode (z-axis)
00223         gauss2.w0 = stod(argv[88]); // - taille
00224         gauss2.zr = stod(argv[89]); // - Rayleigh length
00225         gauss2.ph0 = stod(argv[90]); // - beam center
00226         gauss2.phA = stod(argv[91]); // - beam length
00227         if (stoi(argv[69])>2) {
00228             cerr<<"You can only use up to two Gaussian pulses." << endl;
00229             exit(1);
00230         }
00231         if (stoi(argv[69])>0) Gaussians3D.emplace_back(gauss1);
00232         if (stoi(argv[69])==2) Gaussians3D.emplace_back(gauss2);
00233     }
00234     //////////////////////////////////////////////////////////////////
00235
00236     //----- END OF CONFIGURATION -----//
00237
00238 #if defined(_MPI)
00239     // Finalize MPI environment
00240     MPI_Finalize();

```

```

00241 #endif
00242
00243     return 0;
00244 }

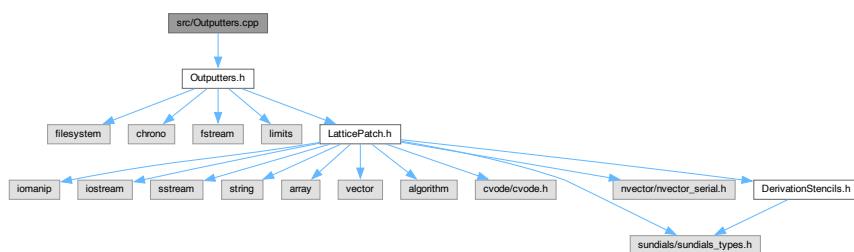
```

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;
0010
0011 /// Directly generate the simCode at construction
0012 OutputManager::OutputManager() {
0013     simCode = SimCodeGenerator();
0014     outputStyle = 'c';
0015 }
0016
0017 /// Generate the identifier number reverse from year to minute in the format
0018 /// yy-mm-dd_hh-MM-ss
0019 std::string OutputManager::SimCodeGenerator() {
0020     const chrono::time_point<chrono::system_clock> now{
0021         chrono::system_clock::now()};
0022     const chrono::year_month_day ymd{chrono::floor<chrono::days>(now)};
0023     const auto tod = now - chrono::floor<chrono::days>(now);
0024     const chrono::hh_mm_ss hms{tod};
0025
0026     std::stringstream temp;
0027     temp << std::setfill('0') << std::setw(2)
0028         << static_cast<int>(ymd.year() - chrono::years(2000)) << "-"
0029         << 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 direcory "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 = 0;
00050 #if defined(_MPI)
00051     MPI_Comm_rank(MPI_COMM_WORLD, &myPrc);
00052 #endif
00053     if (myPrc == 0) {
00054         if (!fs::is_directory(dir))
00055             fs::create_directory(dir);
00056         if (!fs::is_directory(dir + "/SimResults"))
00057             fs::create_directory(dir + "/SimResults");
00058         if (!fs::is_directory(dir + "/SimResults/" + simCode))
00059             fs::create_directory(dir + "/SimResults/" + simCode);
00060     }
00061     // path variable for the output generation
00062     Path = dir + "/SimResults/" + simCode + "/";
00063
00064     // Logging configurations from main.cpp -> no more necessary
00065     /*
00066     std::ifstream fin("main.cpp");
00067     std::ofstream fout(Path + "config.txt");
00068     std::string line;
00069     int begin = 1000;
00070     for (int i = 1; !fin.eof(); i++) {
00071         getline(fin, line);
00072         if (line.starts_with("      //----- B")) {
00073             begin=i;
00074         }
00075         if (i < begin) {
00076             continue;
00077         }
00078         fout << line << std::endl;
00079         if (line.starts_with("      //----- E")) {
00080             break;
00081         }
00082     }
00083     */
00084     return;
00085 }
00086
00087 void OutputManager::set_outputStyle(const char _outputStyle) {
00088     outputStyle = _outputStyle;
00089 }
00090
00091 /** Write the field data either in csv format to one file per each process
00092 * (patch) or in binary form to a single file. Files are stores inthe simCode
00093 * directory. For csv files the state (simulation step) denotes the
00094 * prefix and the suffix after an underscore is given by the process/patch
00095 * number. Binary files are simply named after the step number. */
00096 void OutputManager::outUState(const int &state, const Lattice &lattice,
00097     const LatticePatch &latticePatch) {
00098     switch(outputStyle) {
00099     case 'c': { // one csv file per process
00100         std::ofstream ofs;
00101         ofs.open(Path + std::to_string(state) + "_"
00102             + std::to_string(lattice.my_prc) + ".csv");
00103         // Precision of sunrealtype in significant decimal digits; 15 for IEEE double
00104         ofs << std::setprecision(std::numeric_limits<sunrealtype>::digits10);
00105
00106         // Walk through each lattice point
00107         const sunindextype totalNP = latticePatch.discreteSize();
00108         for (sunindextype i = 0; i < totalNP * 6; i += 6) {
00109             // Six columns to contain the field data: Ex,Ey,Ez,Bx,By,Bz
00110             ofs << latticePatch.uData[i + 0] << "," << latticePatch.uData[i + 1] << ","
00111             << latticePatch.uData[i + 2] << "," << latticePatch.uData[i + 3] << ","
00112             << latticePatch.uData[i + 4] << "," << latticePatch.uData[i + 5]
00113             << std::endl;
00114         }
00115         ofs.close();
00116         break;

```

```

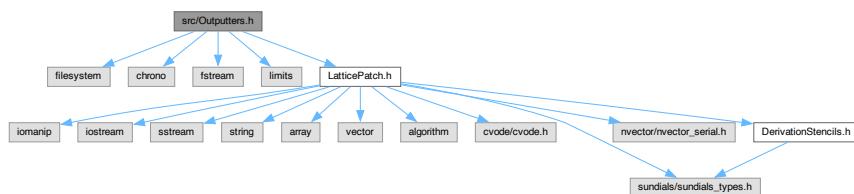
00117             }
00118 #if defined(_MPI)
00119     case 'b': { // a single binary file
00120     // Open the output file
00121     MPI_File fh;
00122     const std::string filename = Path+std::to_string(state);
00123     MPI_File_open(lattice.comm,&filename[0],MPI_MODE_WRONLY|MPI_MODE_CREATE,
00124         MPI_INFO_NULL,&fh);
00125     // number of datapoints in the patch with process offset
00126     const sunindextype count = latticePatch.discreteSize()*
00127         lattice.get_dataPointDimension();
00128     MPI_Offset offset = lattice.my_prc*count*sizeof(MPI_SUNREALTYPE);
00129     // Go to offset in file and write data to it; maximal precision in
00130     // "native" representation
00131     MPI_File_set_view(fh,offset,MPI_SUNREALTYPE,MPI_SUNREALTYPE,"native",
00132         MPI_INFO_NULL);
00133 #if !defined(WIN32) // MSMPI does not yet support nonblocking collective write
00134     MPI_Request write_request;
00135     MPI_File_iwrite_all(fh,latticePatch.uData,count,MPI_SUNREALTYPE,
00136         &write_request);
00137     MPI_Wait(&write_request,MPI_STATUS_IGNORE);
00138 #else
00139     MPI_Status status;
00140     MPI_File_write_at_all(fh,offset,latticePatch.uData,count,MPI_SUNREALTYPE,
00141         &status);
00142 #endif
00143     MPI_File_close(&fh);
00144     break;
00145 }
00146 #endif
00147     default: {
00148     errorKill("No valid output style defined.\n"
00149         "Choose between csv: one CSV file per process, and "
00150         "binary: one binary file. In case MPI is not used, you "
00151         "may only choose csv.");
00152     break;
00153 }
00154 }
00155 }
00156

```

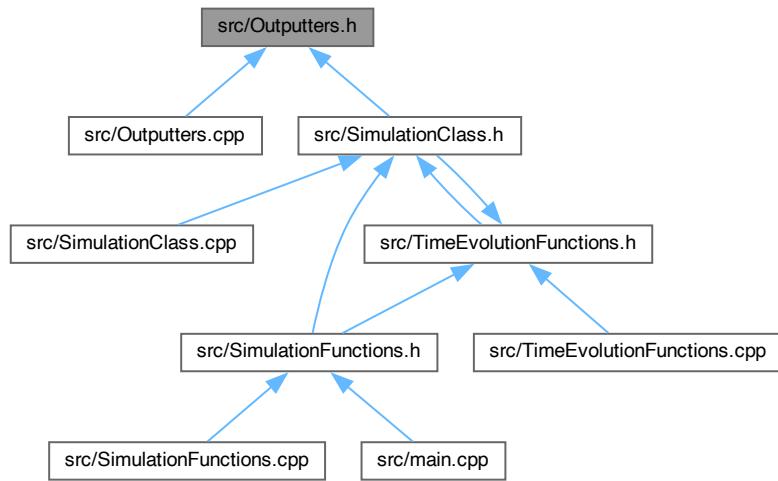
6.18 src/Outputters.h File Reference

[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:
  
```

```

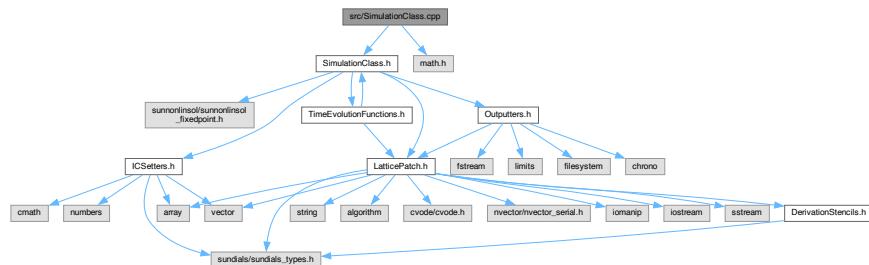
00023  /// function to create the Code of the Simulations
00024  static std::string SimCodeGenerator();
00025  /// variable to save the SimCode generated at execution
00026  std::string simCode;
00027  /// variable for the path to the output folder
00028  std::string Path;
00029  /// output style; csv 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
0010 #include <math.h>
0011
0012 /// Along with the simulation object, create the cartesian communicator and
0013 /// SUNContext object
0014 Simulation::Simulation(const int Nx, const int Ny, const int Nz,
0015     const int StencilOrder, const bool periodicity) :
0016     lattice(StencilOrder){
0017     statusFlags = 0;
0018     t = 0;
0019 #if defined(_MPI)
0020     // Initialize the cartesian communicator
0021     lattice.initializeCommunicator(Nx, Ny, Nz, periodicity);
0022 #endif
0023
0024     // Create the SUNContext object associated with the thread of execution
0025     int retval = 0;
0026     retval = SUNContext_Create(&lattice.comm, &lattice.sunctx);
0027     if (check_retval(&retval, "SUNContext_Create", 1, lattice.my_prc))
0028         errorKill("at SUNContext_Create.");
0029 }
0030
0031 /// Free the CVode solver memory and Sundials context object with the finish of
0032 /// the simulation
0033 Simulation::~Simulation() {
0034     // Free solver memory
0035     if (statusFlags & CvodeObjectSetUp) {
0036         CVodeFree(&cvode_mem);
0037         SUNNonlinSolFree(NLS);
0038         SUNContext_Free(&lattice.sunctx);
0039     }
0040 }
0041
0042 /// Set the discrete dimensions, the number of points per dimension
0043 void Simulation::setDiscreteDimensionsOfLattice(const sunindextype nx,
0044     const sunindextype ny, const sunindextype nz) {
0045     checkNoFlag(LatticePatchworkSetUp);
0046     lattice.setDiscreteDimensions(nx, ny, nz);
0047     statusFlags |= LatticeDiscreteSetUp;
0048 }
0049
0050 /// Set the physical dimensions with lengths in micro meters
0051 void Simulation::setPhysicalDimensionsOfLattice(const sunrealtype lx,
0052     const sunrealtype ly, const sunrealtype lz) {
0053     checkNoFlag(LatticePatchworkSetUp);
0054     lattice.setPhysicalDimensions(lx, ly, lz);
0055     statusFlags |= LatticePhysicalSetUp;
0056 }
0057
0058 /// Check that the lattice dimensions are set up and generate the patchwork
0059 void Simulation::initializePatchwork(const int nx, const int ny,
0060     const int nz) {
0061     checkFlag(LatticeDiscreteSetUp);
0062     checkFlag(LatticePhysicalSetUp);
0063
0064     // Generate the patchwork
0065 #if !defined(_MPI)
0066     if( nx>1 || ny>1 || nz>1 )
0067         errorKill("Splitting the lattice does not work without MPI.");
0068 #endif
0069     generatePatchwork(lattice, latticePatch, nx, ny, nz);
0070     latticePatch.initializeBuffers();
0071
0072     statusFlags |= LatticePatchworkSetUp;
0073 }
0074
0075 /// Configure CVODE
0076 void Simulation::initializeCVODEobject(const sunrealtype reltol,
0077     const sunrealtype abstol) {
0078     checkFlag(SimulationStarted);
0079
0080     // CVode settings return value
0081     int retval = 0;
0082
0083     // Create CVODE object -- returns a pointer to the cvode memory structure
0084     // with Adams method (Adams-Moulton formula) solver chosen for non-stiff ODE
0085     cvode_mem = CVodeCreate(CV_ADAMS, lattice.sunctx);
0086
0087     // Specify user data and attach it to the main cvode memory block
0088     retval = CVodeSetUserData(
0089         cvode_mem,
0090         &latticePatch); // patch contains the user data as used in CVRhsFn
0091     if (check_retval(&retval, "CVodeSetUserData", 1, lattice.my_prc))
0092         errorKill("at CVodeSetUserData.");
0093

```

```

00094 // Initialize CVODE solver
00095 retval = CVodeInit(cvode_mem, TimeEvolution::f, 0,
00096                     latticePatch.u); // allocate memory, CVRhsFn f, t_i=0, u
00097                     // contains the initial values
00098 if (check_retval(&retval, "CVodeInit", 1, lattice.my_prc))
00099     errorKill("at CVodeInit.");
00100
00101 // Create fixed point nonlinear solver object (suitable for non-stiff ODE) and
00102 // attach it to CVode
00103 NLS = SUNNonlinSol_FixedPoint(latticePatch.u, 0, lattice.sunctx);
00104 retval = CVodeSetNonlinearSolver(cvode_mem, NLS);
00105 if (check_retval(&retval, "CVodeSetNonlinearSolver", 1, lattice.my_prc))
00106     errorKill("at CVodeSetNonlinearSolver.");
00107
00108 // Anderson damping factor
00109 retval = SUNNonlinSolSetDamping_FixedPoint(NLS,1);
00110 if (check_retval(&retval, "SUNNonlinSolSetDamping_FixedPoint", 1,
00111                 lattice.my_prc))
00112     errorKill("at SUNNonlinSolSetDamping_FixedPoint.");
00113
00114 // Specify integration tolerances -- a scalar relative tolerance and scalar
00115 // absolute tolerance
00116 retval = CVodeSStolerances(cvode_mem, reltol, abstol);
00117 if (check_retval(&retval, "CVodeSStolerances", 1, lattice.my_prc))
00118     errorKill("at CVodeSStolerances.");
00119
00120 // Specify the maximum number of steps to be taken by the solver in its
00121 // attempt to reach the next tout
00122 retval = CVodeSetMaxNumSteps(cvode_mem, 10000);
00123 if (check_retval(&retval, "CVodeSetMaxNumSteps", 1, lattice.my_prc))
00124     errorKill("at CVodeSetMaxNumSteps.");
00125
00126 // maximum number of warnings for too small h
00127 retval = CVodeSetMaxHnilWarns(cvode_mem,3);
00128 if (check_retval(&retval, "CVodeSetMaxHnilWarns", 1, lattice.my_prc))
00129     errorKill("at CVodeSetMaxHnilWarns.");
00130
00131 statusFlags |= CvodeObjectSetUp;
00132 }
00133
00134 /// Check if the lattice patchwork is set up and set the initial conditions
00135 void Simulation::start() {
00136     checkFlag(LatticeDiscreteSetUp);
00137     checkFlag(LatticePhysicalSetUp);
00138     checkFlag(LatticePatchworkSetUp);
00139     checkNoFlag(SimulationStarted);
00140     checkNoFlag(CvodeObjectSetUp);
00141     setInitialConditions();
00142     statusFlags |= SimulationStarted;
00143 }
00144
00145 /// Set initial conditions: Fill the lattice points with the initial field
00146 /// values
00147 void Simulation::setInitialConditions() {
00148     const sunrealtype dx = latticePatch.getDelta(1);
00149     const sunrealtype dy = latticePatch.getDelta(2);
00150     const sunrealtype dz = latticePatch.getDelta(3);
00151     const sunindextype nx = latticePatch.discreteSize(1);
00152     const sunindextype ny = latticePatch.discreteSize(2);
00153     const sunindextype totalNP = latticePatch.discreteSize();
00154     const sunrealtype x0 = latticePatch.origin(1);
00155     const sunrealtype y0 = latticePatch.origin(2);
00156     const sunrealtype z0 = latticePatch.origin(3);
00157     sunindextype px = 0, py = 0, pz = 0;
00158 #pragma omp parallel for default(none) \
00159 shared(nx, ny, totalNP, dx, dy, dz, x0, y0, z0) \
00160 firstprivate(px, py, pz) schedule(static)
00161     for (sunindextype i = 0; i < totalNP * 6; i += 6) {
00162         px = (i / 6) % nx;
00163         py = ((i / 6) / nx) % ny;
00164         pz = ((i / 6) / nx) / ny;
00165         // Call the 'eval' function to fill the lattice points with the field data
00166         icsettings.eval(static_cast<sunrealtype>(px) * dx + x0,
00167                         static_cast<sunrealtype>(py) * dy + y0,
00168                         static_cast<sunrealtype>(pz) * dz + z0, &latticePatch.uData[i]);
00169     }
00170     return;
00171 }
00172
00173 /// Use parameters to add periodic IC layers
00174 void Simulation::addInitialConditions(const sunindextype xm,
00175                                         const sunindextype ym,
00176                                         const sunindextype zm /* zm=0 always */ ) {
00177     const sunrealtype dx = latticePatch.getDelta(1);
00178     const sunrealtype dy = latticePatch.getDelta(2);
00179     const sunrealtype dz = latticePatch.getDelta(3);
00180     const sunindextype nx = latticePatch.discreteSize(1);

```

```

00181 const sunindextype ny = latticePatch.discreteSize(2);
00182 const sunindextype totalNP = latticePatch.discreteSize();
00183 // Correct for demanded displacement, rest as for setInitialConditions
00184 const sunrealtype x0 = latticePatch.origin(1) + xm*lattice.get_tot_lx();
00185 const sunrealtype y0 = latticePatch.origin(2) + ym*lattice.get_tot_ly();
00186 const sunrealtype z0 = latticePatch.origin(3) + zm*lattice.get_tot_lz();
00187 sunindextype px = 0, py = 0, pz = 0;
00188 for (sunindextype i = 0; i < totalNP * 6; i += 6) {
00189     px = (i / 6) % nx;
00190     py = ((i / 6) / nx) % ny;
00191     pz = ((i / 6) / nx) / ny;
00192     icSettings.add(static_cast<sunrealtype>(px) * dx + x0,
00193                     static_cast<sunrealtype>(py) * dy + y0,
00194                     static_cast<sunrealtype>(pz) * dz + z0, &latticePatch.uData[i]);
00195 }
00196 return;
00197 }
00198
00199 /// Add initial conditions in one dimension
00200 void Simulation::addPeriodicICLayerInX() {
00201     addInitialConditions(-1, 0, 0);
00202     addInitialConditions(1, 0, 0);
00203     return;
00204 }
00205
00206 /// Add initial conditions in two dimensions
00207 void Simulation::addPeriodicICLayerInXY() {
00208     addInitialConditions(-1, -1, 0);
00209     addInitialConditions(-1, 0, 0);
00210     addInitialConditions(-1, 1, 0);
00211     addInitialConditions(0, 1, 0);
00212     addInitialConditions(0, -1, 0);
00213     addInitialConditions(1, -1, 0);
00214     addInitialConditions(1, 0, 0);
00215     addInitialConditions(1, 1, 0);
00216     return;
00217 }
00218
00219 /// Advance the solution in time -> integrate the ODE over an interval t
00220 void Simulation::advanceToTime(const sunrealtype &tEnd) {
00221     checkFlag(SimulationStarted);
00222     int retval = 0;
00223     retval = CVode(cvode_mem, tEnd, latticePatch.u, &t,
00224                   CV_NORMAL); // CV_NORMAL: internal steps to reach tEnd, then
00225                   // interpolate to return latticePatch.u, return time
00226                   // reached by the solver as t
00227     if (check(retval, "CVode", 1, lattice.my_prc))
00228         errorKill("at CVode integration.");
00229 }
00230
00231 /// Write specified simulation steps to disk
00232 void Simulation::outAllFieldData(const int &state) {
00233     checkFlag(SimulationStarted);
00234     outputManager.outUState(state, lattice, latticePatch);
00235 }
00236
00237 /// Check presence of configuration flags
00238 void Simulation::checkFlag(unsigned int flag) const {
00239     if (!(statusFlags & flag)) {
00240         std::string errorMessage;
00241         switch (flag) {
00242             case LatticeDiscreteSetUp:
00243                 errorMessage = "The discrete size of the Simulation has not been set up";
00244                 break;
00245             case LatticePhysicalSetUp:
00246                 errorMessage = "The physical size of the Simulation has not been set up";
00247                 break;
00248             case LatticePatchworkSetUp:
00249                 errorMessage = "The patchwork for the Simulation has not been set up";
00250                 break;
00251             case CvodeObjectSetUp:
00252                 errorMessage = "The CVODE object has not been initialized";
00253                 break;
00254             case SimulationStarted:
00255                 errorMessage = "The Simulation has not been started";
00256                 break;
00257             default:
00258                 errorMessage = "Uppss, you've made a non-standard error, sadly I can't "
00259                             "help you there";
00260                 break;
00261         }
00262         errorKill(errorMessage);
00263     }
00264     return;
00265 }
00266
00267 /// Check absence of configuration flags

```

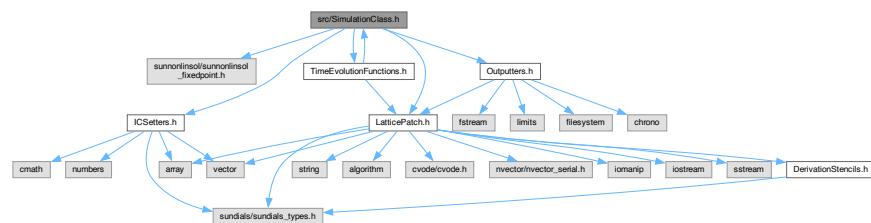
```

00268 void Simulation::checkNoFlag(unsigned int flag) const {
00269     if ((statusFlags & flag)) {
00270         std::string errorMessage;
00271         switch (flag) {
00272             case LatticeDiscreteSetUp:
00273                 errorMessage =
00274                     "The discrete size of the Simulation has already been set up";
00275                 break;
00276             case LatticePhysicalSetUp:
00277                 errorMessage =
00278                     "The physical size of the Simulation has already been set up";
00279                 break;
00280             case LatticePatchworkSetUp:
00281                 errorMessage = "The patchwork for the Simulation has already been set up";
00282                 break;
00283             case CvodeObjectSetUp:
00284                 errorMessage = "The CVODE object has already been initialized";
00285                 break;
00286             case SimulationStarted:
00287                 errorMessage = "The simulation has already started, some changes are no "
00288                     "longer possible";
00289                 break;
00290             default:
00291                 errorMessage = "Uppss, you've made a non-standard error, sadly I can't "
00292                     "help you there";
00293                 break;
00294         }
00295         errorKill(errorMessage);
00296     }
00297     return;
00298 }
```

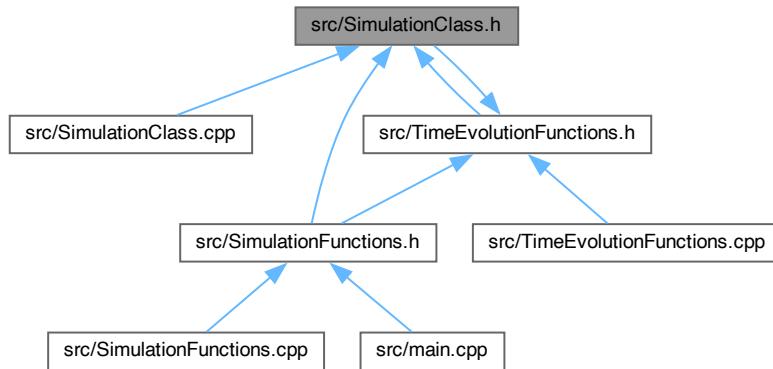
6.22 src/SimulationClass.h File Reference

Class for the **Simulation** object calling all functionality: from wave settings over lattice construction, time evolution and outputs initialization of the CVode 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 [CVode](#) 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::setDiscreteDimensions\(\)](#), 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::setDiscreteDimensions\(\)](#), [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::setPhysicalDimensions\(\)](#), 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 CVode 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 CvodeObjectSetUp = 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 CVode memory object
00047     void *cvode_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 CVode memory and Sundials context
00054     ~Simulation();
00055 #if defined(_MPI)
00056     /// reference to the cartesian communicator of the lattice (for debugging)
00057     MPI_Comm *get_cart_comm() { return &lattice.comm; }
00058 #endif
00059     /// function to set discrete dimensions of the lattice
00060     void setDiscreteDimensionsOfLattice(const sunindextype _tot_nx,
00061                                         const sunindextype _tot_ny, const sunindextype _tot_nz);
00062     /// function to set physical dimensions of the lattice
```

```

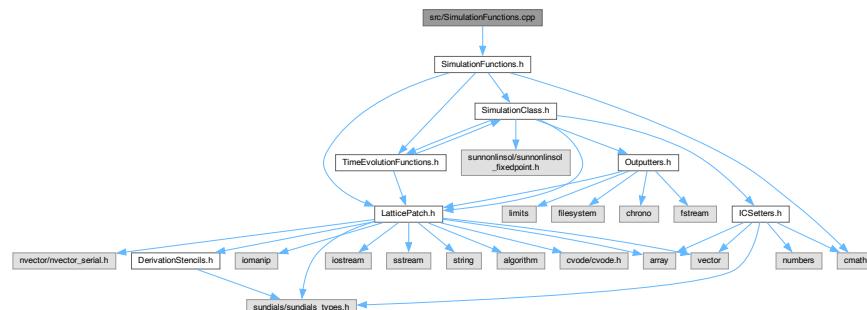
00063 void setPhysicalDimensionsOfLattice(const sunrealtypes lx,
00064     const sunrealtypes ly, const sunrealtypes lz);
00065     /// function to initialize the Patchwork
00066 void initializePatchwork(const int nx, const int ny, const int nz);
00067     /// function to initialize the CVODE object with all requirements
00068 void initializeCVODEobject(const sunrealtypes reltol,
00069     const sunrealtypes abstol);
00070     /// function to start the simulation for time iteration
00071 void start();
00072     /// functions to set the initial field configuration onto the lattice
00073 void setInitialConditions();
00074     /// functions to add initial periodic field configurations
00075 void addInitialConditions(const sunindextypes xm, const sunindextypes ym,
00076     const sunindextypes zm = 0);
00077     /// function to add a periodic IC layer in one dimension
00078 void addPeriodicICLayerInX();
00079     /// function to add periodic IC layers in two dimensions
00080 void addPeriodicICLayerInXY();
00081     /// function to advance solution in time with CVODE
00082 void advanceToTime(const sunrealtypes &tEnd);
00083     /// function to write field data to disk
00084 void outAllFieldData(const int &state);
00085     /// function to check if flag has been set
00086 void checkFlag(unsigned int flag) const;
00087     /// function to check if flag has not been set
00088 // message and cause an abort on all ranks
00089 void checkNoFlag(unsigned int flag) const;
00090 };
00091

```

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)
`timer function`
- void `Sim1D` (const std::array< sunrealtypes, 2 > CVodeTol, const int StencilOrder, const sunrealtypes phys_dim, const sunindextypes disc_dim, const bool periodic, int *interactions, const sunrealtypes 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< sunrealtypes, 2 > CVodeTol, int const StencilOrder, const std::array< sunrealtypes, 2 > phys_dims, const std::array< sunindextypes, 2 > disc_dims, const std::array< int, 2 > patches, const bool periodic, int *interactions, const sunrealtypes endTime, const int numberOfSteps, const std::string outputDirectory, const int outputStep, const char outputStyle, const std::vector< `planewave` > &planes, const std::vector< `gaussian2D` > &gaussians)

- 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*

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 > 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 int myPrc = 0, nPrc = 1;
00032 #if defined(_MPI)
00033 double ts = MPI_Wtime();
00034 MPI_Comm_size(MPI_COMM_WORLD, &nPrc);
00035 MPI_Comm_rank(MPI_COMM_WORLD, &myPrc);
00036
00037 // Check feasibility of the patchwork decomposition
00038 if (myPrc == 0) {
00039     if (disc_dim % nPrc != 0) {
00040         errorKill("The number of lattice points must be "
00041                 "divisible by the number of processes.");
00042     }
00043 }
00044 #elif !defined(_MPI) && defined(_OPENMP)
00045     double ts = omp_get_wtime();
00046 #endif
```

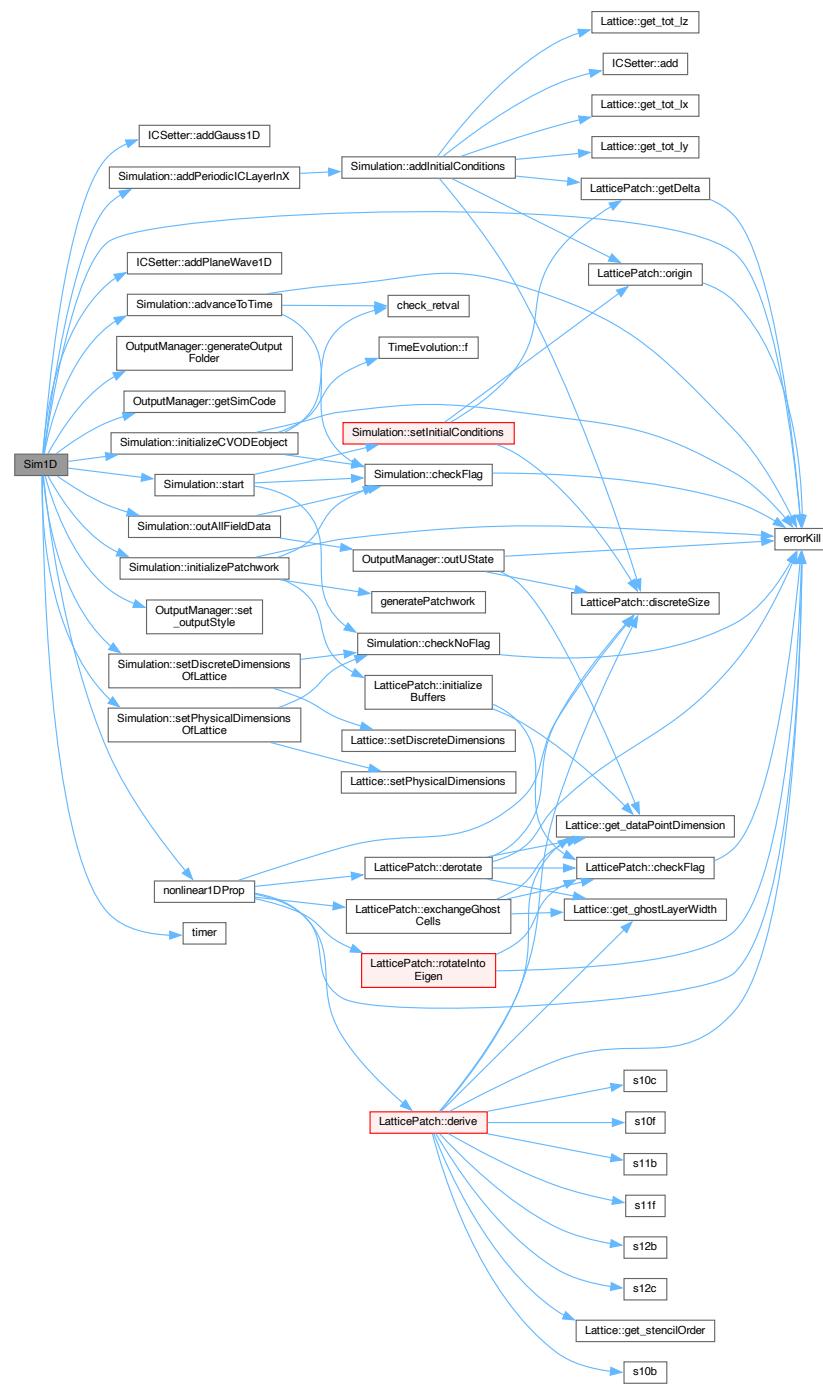
```

00047
00048 // Initialize the simulation, set up the cartesian communicator
00049 std::array<int, 3> patches = {nPrc, 1, 1};
00050 Simulation sim(patches[0], patches[1], patches[2], StencilOrder, periodic);
00051
00052 // Configure the patchwork
00053 sim.setPhysicalDimensionsOfLattice(phys_dim,1,1);
00054 sim.setDiscreteDimensionsOfLattice(disc_dim,1,1);
00055 sim.initializePatchwork(patches[0], patches[1], patches[2]);
00056
00057 // Add em-waves
00058 for (const auto &gauss : gaussians)
00059     sim.icsettings.addGauss1D(gauss.k, gauss.p, gauss.x0, gauss.phig,
00060                               gauss.phi);
00061 for (const auto &plane : planes)
00062     sim.icsettings.addPlaneWave1D(plane.k, plane.p, plane.phi);
00063
00064 // Check that the patchwork is ready and set the initial conditions
00065 sim.start();
00066 sim.addPeriodicICLayerInX();
00067
00068 // Initialize CVode with abs and rel tolerances
00069 sim.initializeCVODEObject(CVodeTol[0], CVodeTol[1]);
00070
00071 // Configure the time evolution function
00072 TimeEvolution::c = interactions;
00073 TimeEvolution::TimeEvolver = nonlinear1DProp;
00074
00075 // Configure the output
00076 sim.outputManager.generateOutputFolder(outputDirectory);
00077 if (!myPrc) {
00078     std::cout << "Simulation code: " << sim.outputManager.getSimCode()
00079     << std::endl;
00080 }
00081 sim.outputManager.set_outputStyle(outputStyle);
00082
00083 //MPI_Barrier(MPI_COMM_WORLD); // insure correct output
00084 //sim.outAllFieldData(0); // output of initial state
00085 // Conduct the propagation in space and time
00086 for (int step = 1; step <= numberOfSteps; step++) {
00087     sim.advanceToTime(endTime / numberOfSteps * step);
00088     if (step % outputStep == 0) {
00089 #if defined(_MPI)
00090         MPI_Barrier(MPI_COMM_WORLD); // insure correct output
00091 #endif
00092     sim.outAllFieldData(step);
00093 }
00094 #if defined(_MPI)
00095     double tn = MPI_Wtime();
00096 #elif !defined(_MPI) && defined(_OPENMP)
00097     double tn = omp_get_wtime();
00098 #endif
00099     if (!myPrc) {
00100         std::cout << "\rStep " << step << "\t\t" << std::flush;
00101 #if defined(_MPI) || defined(_OPENMP)
00102         timer(ts, tn);
00103 #endif
00104     }
00105 }
00106
00107 return;
00108 }
```

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\(\)](#).

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

Here is the call graph for this function:



Here is the caller graph for this function:



6.24.2.2 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 111 of file [SimulationFunctions.cpp](#).

```

00119
00120
00121 // MPI data
00122 int myPrc = 0, nPrc = 1; // Get process rank and number of processes
00123 #if defined(_MPI)
00124 double ts = MPI_Wtime();
00125 MPI_Comm_rank(MPI_COMM_WORLD,
00126                 &myPrc); // Return process rank, number \in [1,nPrc]
00127 MPI_Comm_size(MPI_COMM_WORLD,
00128                 &nPrc); // Return number of processes (communicator size)
00129
00130 // Check feasibility of the patchwork decomposition
00131 if (myPrc == 0) {
00132     if (nPrc != patches[0] * patches[1]) {
00133         errorKill(
00134             "The number of MPI processes must match the number of patches.");
00135     }
00136 }
00137 #elif !defined(_MPI) && defined(_OPENMP)
00138     double ts = omp_get_wtime();
00139 #endif
00140
00141 // Initialize the simulation, set up the cartesian communicator
00142 Simulation sim(patches[0], patches[1], 1, StencilOrder, periodic);
00143
00144 /* // Check that lattice communicator is unique; same as used in patchwork
00145 generation char cart_comm_name[MPI_MAX_OBJECT_NAME]; int cart_namelen;
00146 MPI_Comm_get_name(*sim.get_cart_comm(), cart_comm_name, &cart_namelen);

```

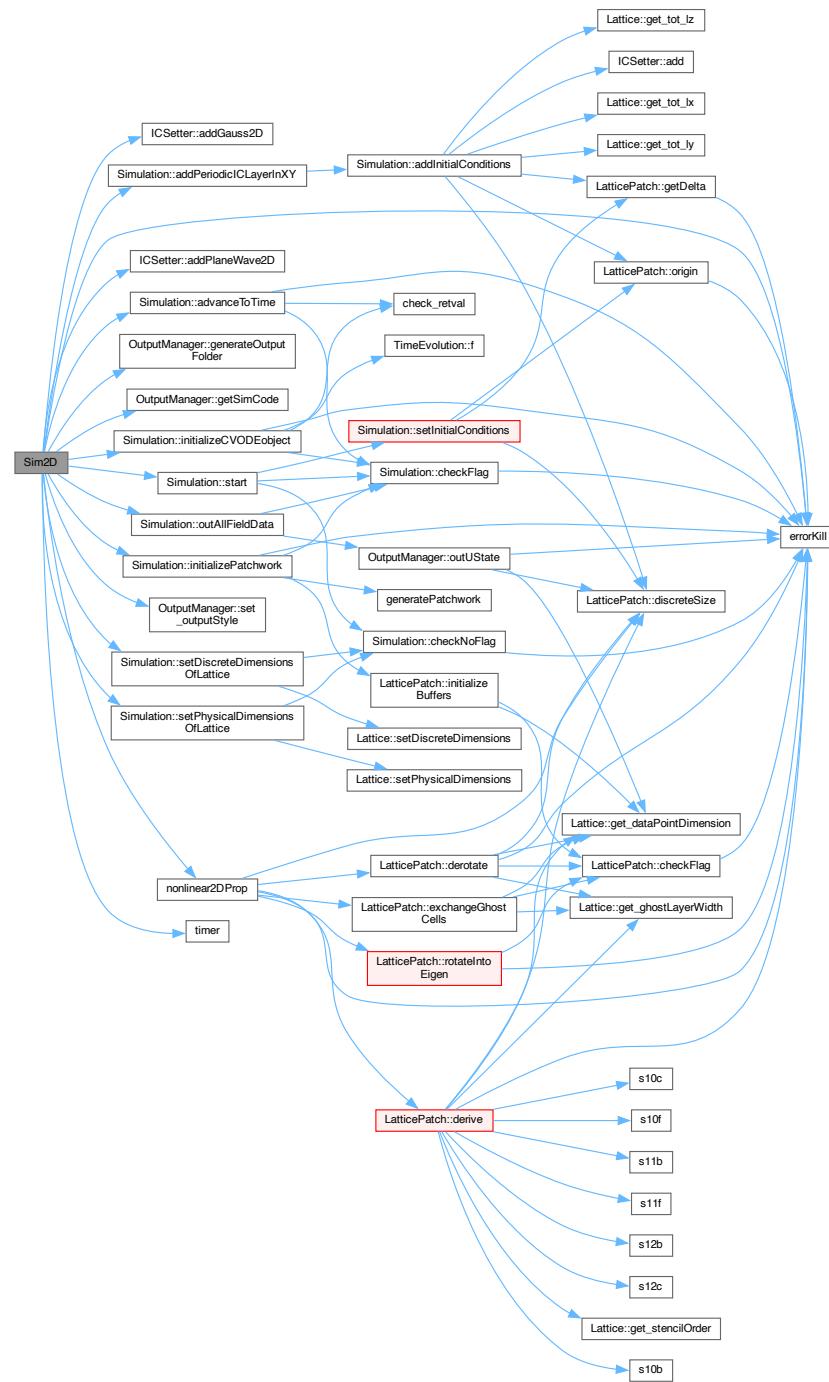
```

00147     printf("sim.get_cart_comm gives %s \n", cart_comm_name);
00148     */
00149
00150     // Configure the patchwork
00151     sim.setPhysicalDimensionsOfLattice(phys_dims[0],
00152                                         phys_dims[1],
00153                                         1); // spacing of the lattice
00154     sim.setDiscreteDimensionsOfLattice(
00155         disc_dims[0], disc_dims[1], 1); // Spacing equivalence to points
00156     sim.initializePatchwork(patches[0], patches[1], 1);
00157
00158     // Add em-waves
00159     for (const auto &gauss : gaussians)
00160         sim.icsettings.addGauss2D(gauss.x0, gauss.axis, gauss.amp, gauss.phip,
00161                                   gauss.w0, gauss.zr, gauss.ph0, gauss.phA);
00162     for (const auto &plane : planes)
00163         sim.icsettings.addPlaneWave2D(plane.k, plane.p, plane.phi);
00164
00165     // Check that the patchwork is ready and set the initial conditions
00166     sim.start(); // Check if the lattice is set up, set initial field
00167             // configuration
00168     sim.addPeriodicICLayerInXY(); // insure periodicity in propagation directions
00169
00170     // Initialize CVode with rel and abs tolerances
00171     sim.initializeCVODEobject(CVodeTol[0], CVodeTol[1]);
00172
00173     // Configure the time evolution function
00174     TimeEvolution::c = interactions;
00175     TimeEvolution::TimeEvolver = nonlinear2DProp;
00176
00177     // Configure the output
00178     sim.outputManager.generateOutputFolder(outputDirectory);
00179     if (!myPrc) {
00180         std::cout << "Simulation code: " << sim.outputManager.getSimCode()
00181             << std::endl;
00182     }
00183     sim.outputManager.set_outputStyle(outputStyle);
00184
00185     //MPI_Barrier(MPI_COMM_WORLD); // insure correct output
00186     //sim.outAllFieldData(0); // output of initial state
00187     // Conduct the propagation in space and time
00188     for (int step = 1; step <= number_of_steps; step++) {
00189         sim.advanceToTime(endTime / number_of_steps * step);
00190         if (step % output_step == 0) {
00191             #if defined(_MPI)
00192                 MPI_Barrier(MPI_COMM_WORLD); // insure correct output
00193             #endif
00194             sim.outAllFieldData(step);
00195         }
00196     #if defined(_MPI)
00197         double tn = MPI_Wtime();
00198     #elif !defined(_MPI) && defined(_OPENMP)
00199         double tn = omp_get_wtime();
00200     #endif
00201         if (!myPrc) {
00202             std::cout << "\rStep " << step << "\t\t" << std::flush;
00203     #if defined(_MPI) || defined(_OPENMP)
00204             timer(ts, tn);
00205     #endif
00206         }
00207     }
00208
00209     return;
00210 }
```

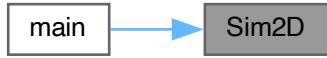
References [ICSetter::addGauss2D\(\)](#), [Simulation::addPeriodicICLayerInXY\(\)](#), [ICSetter::addPlaneWave2D\(\)](#), [Simulation::advanceToTime\(\)](#), [TimeEvolution::c](#), [errorKill\(\)](#), [OutputManager::generateOutputFolder\(\)](#), [OutputManager::getSimCode\(\)](#), [Simulation::icsettings](#), [Simulation::initializeCVODEobject\(\)](#), [Simulation::initializePatchwork\(\)](#), [nonlinear2DProp\(\)](#), [Simulation::outAllFieldData\(\)](#), [Simulation::outputManager](#), [OutputManager::set_outputStyle\(\)](#), [Simulation::setDiscreteDimensionsOfLattice\(\)](#), [Simulation::setPhysicalDimensionsOfLattice\(\)](#), [Simulation::start\(\)](#), [TimeEvolution::TimeEvolver](#), and [timer\(\)](#).

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

Here is the call graph for this function:



Here is the caller 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 213 of file [SimulationFunctions.cpp](#).

```

00221
00222
00223 // MPI data
00224 int myPrc = 0, nPrc = 1; // Get process rank and numer of process
00225 #if defined(_MPI)
00226 double ts = MPI_Wtime();
00227 MPI_Comm_rank(MPI_COMM_WORLD,
00228     &myPrc); // rank of the process inside the world communicator
00229 MPI_Comm_size(MPI_COMM_WORLD,
00230     &nPrc); // Size of the communicator is the number of processes
00231
00232 // Check feasibility of the patchwork decomposition
00233 if (myPrc == 0) {
00234     if (nPrc != patches[0] * patches[1] * patches[2]) {
00235         errorKill(
00236             "The number of MPI processes must match the number of patches.");
00237     }
00238     if ( ( disc_dims[0] / patches[0] != disc_dims[1] / patches[1] ) ||
00239         ( disc_dims[0] / patches[0] != disc_dims[2] / patches[2] ) ) {
00240         std::clog
00241             << "\nWarning: Patches should be cubic in terms of the lattice "
00242                 "points for the computational efficiency of larger simulations.\n";
00243     }
00244 }
00245 #elif !defined(_MPI) && defined(_OPENMP)
00246     double ts = omp_get_wtime();
00247 #endif
00248

```

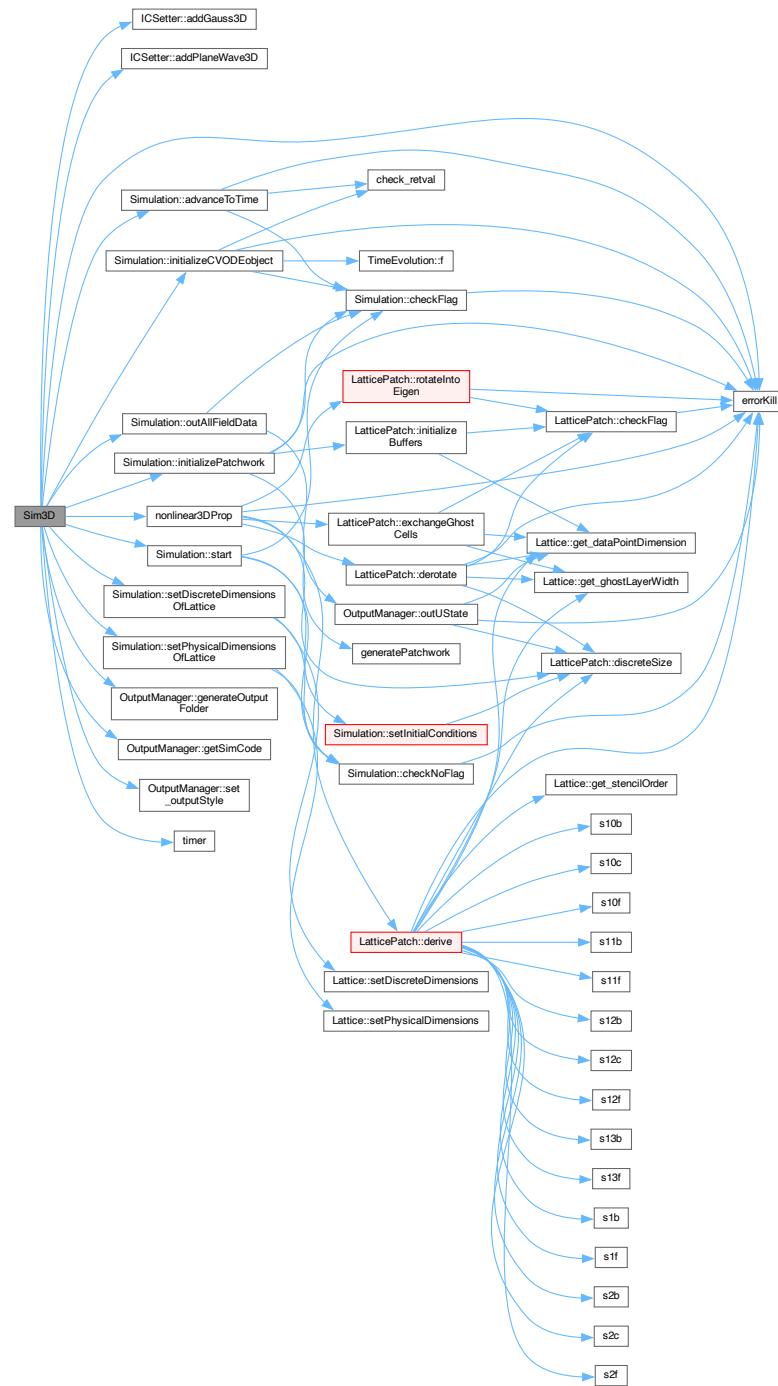
```

00249 // Initialize the simulation, set up the cartesian communicator
00250 Simulation sim(patches[0], patches[1], patches[2],
00251             StencilOrder, periodic); // Simulation object with slicing
00252
00253 // Create the SUNContext object associated with the thread of execution
00254 sim.setPhysicalDimensionsOfLattice(phys_dims[0], phys_dims[1],
00255                                         phys_dims[2]); // spacing of the box
00256 sim.setDiscreteDimensionsOfLattice(
00257     disc_dims[0], disc_dims[1],
00258     disc_dims[2]); // Spacing equivalence to points
00259 sim.initializePatchwork(patches[0], patches[1], patches[2]);
00260 //sim.initializeGhostCells();
00261
00262 // Add em-waves
00263 for (const auto &plane : planes)
00264     sim.icsettings.addPlaneWave3D(plane.k, plane.p, plane.phi);
00265 for (const auto &gauss : gaussians)
00266     sim.icsettings.addGauss3D(gauss.x0, gauss.axis, gauss.amp, gauss.phip,
00267                                gauss.w0, gauss.zr, gauss.ph0, gauss.phA);
00268
00269 // Check that the patchwork is ready and set the initial conditions
00270 sim.start();
00271
00272 // Initialize CVode with abs and rel tolerances
00273 sim.initializeCVODEobject(CVodeTol[0], CVodeTol[1]);
00274
00275 // Configure the time evolution function
00276 TimeEvolution::c = interactions;
00277 TimeEvolution::TimeEvolver = nonlinear3DProp;
00278
00279 // Configure the output
00280 sim.outputManager.generateOutputFolder(outputDirectory);
00281 if (!myPrc) {
00282     std::cout << "Simulation code: " << sim.outputManager.getSimCode()
00283             << std::endl;
00284 }
00285 sim.outputManager.set_outputStyle(outputStyle);
00286
00287 //MPI_Barrier(MPI_COMM_WORLD); // insure correct output
00288 //sim.outAllFieldData(0); // output of initial state
00289 // Conduct the propagation in space and time
00290 for (int step = 1; step <= numberofSteps; step++) {
00291     sim.advanceToTime(endTime / numberofSteps * step);
00292     if (step % outputStep == 0) {
00293 #if defined(_MPI)
00294         MPI_Barrier(MPI_COMM_WORLD); // insure correct output
00295 #endif
00296         sim.outAllFieldData(step);
00297     }
00298 #if defined(_MPI)
00299     double tn = MPI_Wtime();
00300 #elif !defined(_MPI) && defined(_OPENMP)
00301     double tn = omp_get_wtime();
00302 #endif
00303     if (!myPrc) {
00304         std::cout << "\rStep " << step << "\t\t" << std::flush;
00305 #if defined(_MPI) || defined(_OPENMP)
00306     timer(ts, tn);
00307 #endif
00308     }
00309 }
00310 return;
00311 }
```

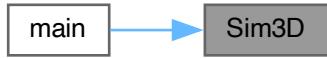
References [ICSetter::addGauss3D\(\)](#), [ICSetter::addPlaneWave3D\(\)](#), [Simulation::advanceToTime\(\)](#), [TimeEvolution::c](#), [errorKill\(\)](#), [OutputManager::generateOutputFolder\(\)](#), [OutputManager::getSimCode\(\)](#), [Simulation::icsettings](#), [Simulation::initializeCVODEobject\(\)](#), [Simulation::initializePatchwork\(\)](#), [nonlinear3DProp\(\)](#), [Simulation::outAllFieldData\(\)](#), [Simulation::outputManager](#), [OutputManager::set_outputStyle\(\)](#), [Simulation::setDiscreteDimensionsOfLattice\(\)](#), [Simulation::setPhysicalDimensionsOfLattice\(\)](#), [Simulation::start\(\)](#), [TimeEvolution::TimeEvolver](#), and [timer\(\)](#).

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

Here is the call graph for this function:



Here is the caller graph for this function:



6.24.2.4 timer()

```
void timer (
    double & t1,
    double & t2 ) [inline]
```

timer function

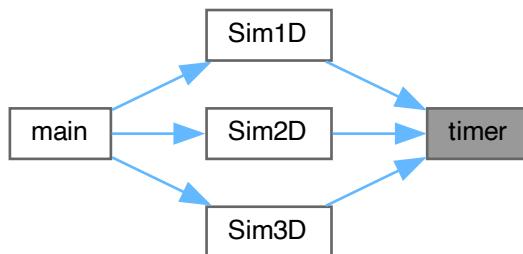
Calculate and print the elapsed wall time

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 elapsed wall 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_Vecotr, N_Vecotr,
00018                                     int *) = nonlinear1DProp;
00019
00020 /** Conduct the complete 1D simulation process */
00021 void Sim1D(const std::array<sunrealtype,2> CVodeTol, const int StencilOrder,
00022             const unrealtype phys_dim, const sunindextype disc_dim,
00023             const bool periodic, int *interactions,
00024             const unrealtype 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     int myPrc = 0, nPrc = 1;
00032 #if defined(_MPI)
00033     double ts = MPI_Wtime();
00034     MPI_Comm_size(MPI_COMM_WORLD, &nPrc);
00035     MPI_Comm_rank(MPI_COMM_WORLD, &myPrc);
00036
00037     // Check feasibility of the patchwork decomposition
00038     if (myPrc == 0) {
00039         if (disc_dim % nPrc != 0) {
00040             errorKill("The number of lattice points must be "
00041                       "divisible by the number of processes.");
00042         }
00043     }
00044 #elif !defined(_MPI) && defined(_OPENMP)
00045     double ts = omp_get_wtime();
00046 #endif
00047
00048     // Initialize the simulation, set up the cartesian communicator
00049     std::array<int, 3> patches = {nPrc, 1, 1};
00050     Simulation sim(patches[0], patches[1], patches[2], StencilOrder, periodic);
00051
00052     // Configure the patchwork
00053     sim.setPhysicalDimensionsOfLattice(phys_dim,1,1);
00054     sim.setDiscreteDimensionsOfLattice(disc_dim,1,1);
00055     sim.initializePatchwork(patches[0], patches[1], patches[2]);
00056
00057     // Add em-waves
00058     for (const auto &gauss : gaussians)
00059         sim.icsettings.addGauss1D(gauss.k, gauss.p, gauss.x0, gauss.phig,
00060                                   gauss.phi);
00061     for (const auto &plane : planes)
00062         sim.icsettings.addPlaneWave1D(plane.k, plane.p, plane.phi);
00063
00064     // Check that the patchwork is ready and set the initial conditions
00065     sim.start();
00066     sim.addPeriodicICLayerInX();
00067
00068     // Initialize CVode with abs and rel tolerances
00069     sim.initializeCVODEobject(CVodeTol[0], CVodeTol[1]);
00070
00071     // Configure the time evolution function
00072     TimeEvolution::c = interactions;
00073     TimeEvolution::TimeEvolver = nonlinear1DProp;
00074
00075     // Configure the output
00076     sim.outputManager.generateOutputFolder(outputDirectory);
00077     if (!myPrc) {
00078         std::cout << "Simulation code: " << sim.outputManager.getSimCode()
00079             << std::endl;
00080     }
00081     sim.outputManager.set_outputStyle(outputStyle);
00082

```

```

00083 //MPI_Barrier(MPI_COMM_WORLD); // insure correct output
00084 //sim.outAllFieldData(); // output of initial state
00085 // Conduct the propagation in space and time
00086 for (int step = 1; step <= numberofSteps; step++) {
00087     sim.advanceToTime(endTime / numberofSteps * step);
00088     if (step % outputStep == 0) {
00089 #if defined(_MPI)
00090         MPI_Barrier(MPI_COMM_WORLD); // insure correct output
00091 #endif
00092         sim.outAllFieldData(step);
00093     }
00094 #if defined(_MPI)
00095     double tn = MPI_Wtime();
00096 #elif !defined(_MPI) && defined(_OPENMP)
00097     double tn = omp_get_wtime();
00098 #endif
00099     if (!myPrc) {
00100         std::cout << "\rStep " << step << "\t\t" << std::flush;
00101 #if defined(_MPI) || defined(_OPENMP)
00102     timer(ts, tn);
00103 #endif
00104     }
00105 }
00106
00107 return;
00108 }
00109
00110 /** Conduct the complete 2D simulation process */
00111 void Sim2D(const std::array<sunrealtype,2> CCodeTol, int const StencilOrder,
00112             const std::array<sunrealtype,2> phys_dims,
00113             const std::array<sunindextype,2> disc_dims,
00114             const std::array<int,2> patches, const bool periodic, int *interactions,
00115             const unrealtype endTime, const int numberofSteps,
00116             const std::string outputDirectory, const int outputStep,
00117             const char outputStyle,
00118             const std::vector<planewave> &planes,
00119             const std::vector<gaussian2D> &gaussians) {
00120
00121 // MPI data
00122 int myPrc = 0, nPrc = 1; // Get process rank and number of processes
00123 #if defined(_MPI)
00124     double ts = MPI_Wtime();
00125     MPI_Comm_rank(MPI_COMM_WORLD,
00126                   &myPrc); // Return process rank, number \in [1,nPrc]
00127     MPI_Comm_size(MPI_COMM_WORLD,
00128                   &nPrc); // Return number of processes (communicator size)
00129
00130 // Check feasibility of the patchwork decomposition
00131 if (myPrc == 0) {
00132     if (nPrc != patches[0] * patches[1]) {
00133         errorKill(
00134             "The number of MPI processes must match the number of patches.");
00135     }
00136 }
00137 #elif !defined(_MPI) && defined(_OPENMP)
00138     double ts = omp_get_wtime();
00139 #endif
00140
00141 // Initialize the simulation, set up the cartesian communicator
00142 Simulation sim(patches[0], patches[1], 1, StencilOrder, periodic);
00143
00144 /* // Check that lattice communicator is unique; same as used in patchwork
00145 generation char cart_comm_name[MPI_MAX_OBJECT_NAME]; int cart_namelen;
00146 MPI_Comm_get_name(*sim.get_cart_comm(), cart_comm_name, &cart_namelen);
00147 printf("sim.get_cart_comm gives %s \n", cart_comm_name);
00148 */
00149
00150 // Configure the patchwork
00151 sim.setPhysicalDimensionsOfLattice(phys_dims[0],
00152                                     phys_dims[1],
00153                                     1); // spacing of the lattice
00154 sim.setDiscreteDimensionsOfLattice(
00155     disc_dims[0], disc_dims[1], 1); // Spacing equivalence to points
00156 sim.initializePatchwork(patches[0], patches[1], 1);
00157
00158 // Add em-waves
00159 for (const auto &gauss : gaussians)
00160     sim.icsettings.addGauss2D(gauss.x0, gauss.axis, gauss.amp, gauss.phip,
00161                               gauss.w0, gauss.zr, gauss.ph0, gauss.phA);
00162 for (const auto &plane : planes)
00163     sim.icsettings.addPlaneWave2D(plane.k, plane.p, plane.phi);
00164
00165 // Check that the patchwork is ready and set the initial conditions
00166 sim.start(); // Check if the lattice is set up, set initial field
00167 // configuration
00168 sim.addPeriodicICLayerInXY(); // insure periodicity in propagation directions
00169

```

```

00170 // Initialize CVode with rel and abs tolerances
00171 sim.initializeCVODEobject(CVodeTol[0], CVodeTol[1]);
00172
00173 // Configure the time evolution function
00174 TimeEvolution::c = interactions;
00175 TimeEvolution::TimeEvolver = nonlinear2DProp;
00176
00177 // Configure the output
00178 sim.outputManager.generateOutputFolder(outputDirectory);
00179 if (!myPrc) {
00180     std::cout << "Simulation code: " << sim.outputManager.getSimCode()
00181         << std::endl;
00182 }
00183 sim.outputManager.set_outputStyle(outputStyle);
00184
00185 //MPI_Barrier(MPI_COMM_WORLD); // insure correct output
00186 //sim.outAllFieldData(0); // output of initial state
00187 // Conduct the propagation in space and time
00188 for (int step = 1; step <= numberOfSteps; step++) {
00189     sim.advanceToTime(endTime / numberOfSteps * step);
00190     if (step % outputStep == 0) {
00191 #if defined(_MPI)
00192         MPI_Barrier(MPI_COMM_WORLD); // insure correct output
00193 #endif
00194     sim.outAllFieldData(step);
00195 }
00196 #if defined(_MPI)
00197     double tn = MPI_Wtime();
00198 #elif !defined(_MPI) && defined(_OPENMP)
00199     double tn = omp_get_wtime();
00200 #endif
00201     if (!myPrc) {
00202         std::cout << "\rStep " << step << "\t\t" << std::flush;
00203 #if defined(_MPI) || defined(_OPENMP)
00204     timer(ts, tn);
00205 #endif
00206 }
00207 }
00208
00209 return;
00210 }
00211
00212 /** Conduct the complete 3D simulation process */
00213 void Sim3D(const std::array<sunrealtype,2> CVodeTol, const int StencilOrder,
00214             const std::array<sunrealtype,3> phys_dims,
00215             const std::array<sunindextype,3> disc_dims,
00216             const std::array<int,3> patches,
00217             const bool periodic, int *interactions, const unrealtype endTime,
00218             const int numberofSteps, const std::string outputDirectory,
00219             const int outputStep, const char outputStyle,
00220             const std::vector<planewave> &planes,
00221             const std::vector<gaussian3D> &gaussians) {
00222
00223 // MPI data
00224 int myPrc = 0, nPrc = 1; // Get process rank and numer of process
00225 #if defined(_MPI)
00226 double ts = MPI_Wtime();
00227 MPI_Comm_rank(MPI_COMM_WORLD,
00228                 &myPrc); // rank of the process inside the world communicator
00229 MPI_Comm_size(MPI_COMM_WORLD,
00230                 &nPrc); // Size of the communicator is the number of processes
00231
00232 // Check feasibility of the patchwork decomposition
00233 if (myPrc == 0) {
00234     if (nPrc != patches[0] * patches[1] * patches[2]) {
00235         errorKill(
00236             "The number of MPI processes must match the number of patches.");
00237     }
00238     if ( ( disc_dims[0] / patches[0] != disc_dims[1] / patches[1] ) ||
00239         ( disc_dims[0] / patches[0] != disc_dims[2] / patches[2] ) ) {
00240         std::clog
00241             << "\nWarning: Patches should be cubic in terms of the lattice "
00242             "points for the computational efficiency of larger simulations.\n";
00243     }
00244 }
00245 #elif !defined(_MPI) && defined(_OPENMP)
00246     double ts = omp_get_wtime();
00247 #endif
00248
00249 // Initialize the simulation, set up the cartesian communicator
00250 Simulation sim(patches[0], patches[1], patches[2],
00251                 StencilOrder, periodic); // Simulation object with slicing
00252
00253 // Create the SUNContext object associated with the thread of execution
00254 sim.setPhysicalDimensionsOfLattice(phys_dims[0], phys_dims[1],
00255                                     phys_dims[2]); // spacing of the box
00256 sim.setDiscreteDimensionsOfLattice(

```

```

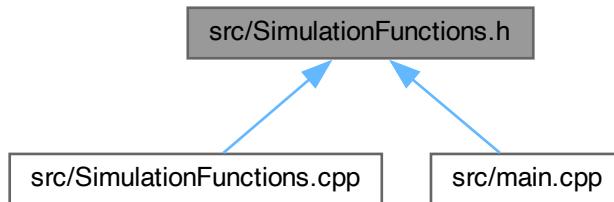
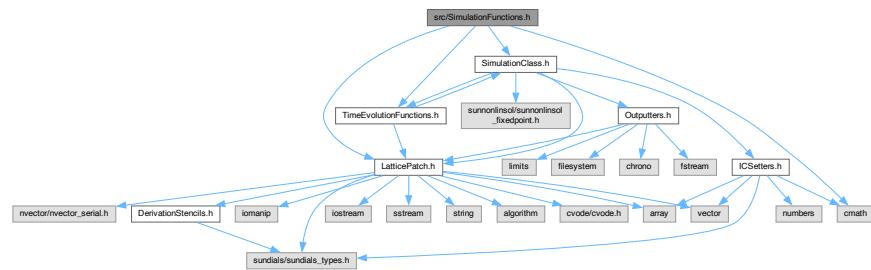
00257     disc_dims[0], disc_dims[1],
00258     disc_dims[2]); // Spacing equivalence to points
00259 sim.initializePatchwork(patches[0], patches[1], patches[2]);
00260 //sim.initializeGhostCells();
00261
00262 // Add em-waves
00263 for (const auto &plane : planes)
00264     sim.icsettings.addPlaneWave3D(plane.k, plane.p, plane.phi);
00265 for (const auto &gauss : gaussians)
00266     sim.icsettings.addGauss3D(gauss.x0, gauss.axis, gauss.amp, gauss.phip,
00267         gauss.w0, gauss.zr, gauss.ph0, gauss.phA);
00268
00269 // Check that the patchwork is ready and set the initial conditions
00270 sim.start();
00271
00272 // Initialize CVode with abs and rel tolerances
00273 sim.initializeCVODEObject(CVodeTol[0], CVodeTol[1]);
00274
00275 // Configure the time evolution function
00276 TimeEvolution::c = interactions;
00277 TimeEvolution::TimeEvolver = nonlinear3DProp;
00278
00279 // Configure the output
00280 sim.outputManager.generateOutputFolder(outputDirectory);
00281 if (!myPrc) {
00282     std::cout << "Simulation code: " << sim.outputManager.getSimCode()
00283         << std::endl;
00284 }
00285 sim.outputManager.set_outputStyle(outputStyle);
00286
00287 //MPI_Barrier(MPI_COMM_WORLD); // insure correct output
00288 //sim.outAllFieldData(0); // output of initial state
00289 // Conduct the propagation in space and time
00290 for (int step = 1; step <= number_of_steps; step++) {
00291     sim.advanceToTime(endTime / number_of_steps * step);
00292     if (step % output_step == 0) {
00293 #if defined(_MPI)
00294         MPI_Barrier(MPI_COMM_WORLD); // insure correct output
00295 #endif
00296     sim.outAllFieldData(step);
00297 }
00298 #if defined(_MPI)
00299     double tn = MPI_Wtime();
00300 #elif !defined(_MPI) && defined(_OPENMP)
00301     double tn = omp_get_wtime();
00302 #endif
00303     if (!myPrc) {
00304         std::cout << "\rStep " << step << "\t\t" << std::flush;
00305 #if defined(_MPI) || defined(_OPENMP)
00306     timer(ts, tn);
00307 #endif
00308 }
00309 }
00310 return;
00311 }
```

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:



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 &)

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 int myPrc = 0, nPrc = 1;
00032 #if defined(_MPI)
00033     double ts = MPI_Wtime();
00034     MPI_Comm_size(MPI_COMM_WORLD, &nPrc);
00035     MPI_Comm_rank(MPI_COMM_WORLD, &myPrc);
00036
00037 // Check feasibility of the patchwork decomposition
00038 if (myPrc == 0) {
00039     if (disc_dim % nPrc != 0) {
00040         errorKill("The number of lattice points must be "
00041                 "divisible by the number of processes.");
00042     }
00043 }
00044 #elif !defined(_MPI) && defined(_OPENMP)
00045     double ts = omp_get_wtime();
00046 #endif
00047
```

```

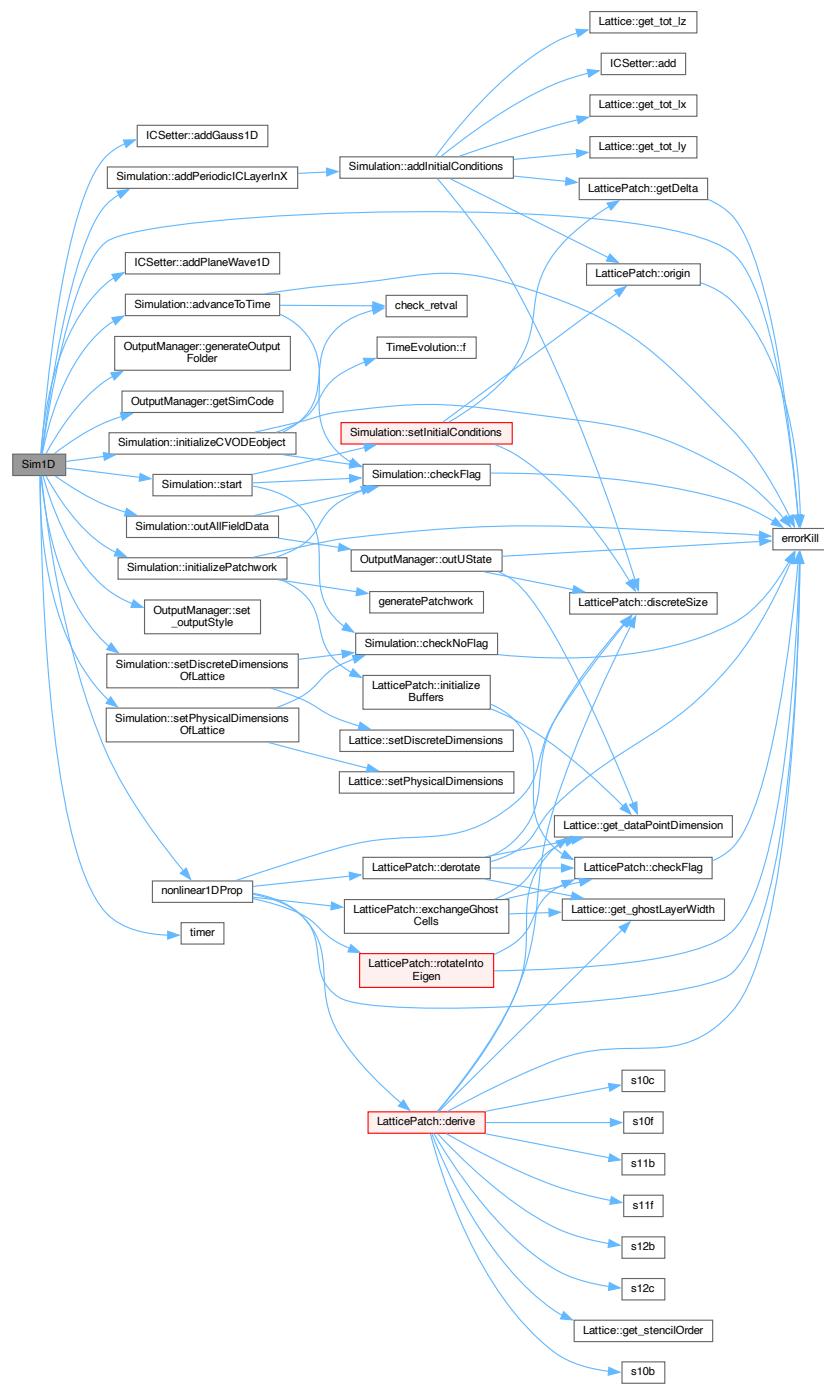
00048 // Initialize the simulation, set up the cartesian communicator
00049 std::array<int, 3> patches = {nPrc, 1, 1};
00050 Simulation sim(patches[0], patches[1], patches[2], StencilOrder, periodic);
00051
00052 // Configure the patchwork
00053 sim.setPhysicalDimensionsOfLattice(phys_dim,1,1);
00054 sim.setDiscreteDimensionsOfLattice(disc_dim,1,1);
00055 sim.initializePatchwork(patches[0], patches[1], patches[2]);
00056
00057 // Add em-waves
00058 for (const auto &gauss : gaussians)
00059     sim.icsettings.addGauss1D(gauss.k, gauss.p, gauss.x0, gauss.phig,
00060                               gauss.phi);
00061 for (const auto &plane : planes)
00062     sim.icsettings.addPlaneWave1D(plane.k, plane.p, plane.phi);
00063
00064 // Check that the patchwork is ready and set the initial conditions
00065 sim.start();
00066 sim.addPeriodicICLayerInX();
00067
00068 // Initialize CVode with abs and rel tolerances
00069 sim.initializeCVODEObject(CVodeTol[0], CVodeTol[1]);
00070
00071 // Configure the time evolution function
00072 TimeEvolution::c = interactions;
00073 TimeEvolution::TimeEvolver = nonlinear1DProp;
00074
00075 // Configure the output
00076 sim.outputManager.generateOutputFolder(outputDirectory);
00077 if (!myPrc) {
00078     std::cout << "Simulation code: " << sim.outputManager.getSimCode()
00079             << std::endl;
00080 }
00081 sim.outputManager.set_outputStyle(outputStyle);
00082
00083 //MPI_Barrier(MPI_COMM_WORLD); // insure correct output
00084 //sim.outAllFieldData(0); // output of initial state
00085 // Conduct the propagation in space and time
00086 for (int step = 1; step <= numberofSteps; step++) {
00087     sim.advanceToTime(endTime / numberofSteps * step);
00088     if (step % outputStep == 0) {
00089 #if defined(_MPI)
00090         MPI_Barrier(MPI_COMM_WORLD); // insure correct output
00091 #endif
00092         sim.outAllFieldData(step);
00093     }
00094 #if defined(_MPI)
00095     double tn = MPI_Wtime();
00096 #elif !defined(_MPI) && defined(_OPENMP)
00097     double tn = omp_get_wtime();
00098 #endif
00099     if (!myPrc) {
00100         std::cout << "\rStep " << step << "\t\t" << std::flush;
00101 #if defined(_MPI) || defined(_OPENMP)
00102         timer(ts, tn);
00103 #endif
00104     }
00105 }
00106
00107 return;
00108 }

```

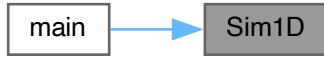
References [ICSetter::addGauss1D\(\)](#), [Simulation::addPeriodicICLayerInX\(\)](#), [ICSetter::addPlaneWave1D\(\)](#), [Simulation::advanceToTime\(\)](#), [TimeEvolution::c](#), [errorKill\(\)](#), [OutputManager::generateOutputFolder\(\)](#), [OutputManager::getSimCode\(\)](#), [Simulation::icsettings](#), [Simulation::initializeCVODEObject\(\)](#), [Simulation::initializePatchwork\(\)](#), [nonlinear1DProp\(\)](#), [Simulation::outAllFieldData\(\)](#), [Simulation::outputManager](#), [OutputManager::set_outputStyle\(\)](#), [Simulation::setDiscreteDimensionsOfLattice\(\)](#), [Simulation::setPhysicalDimensionsOfLattice\(\)](#), [Simulation::start\(\)](#), [TimeEvolution::TimeEvolver](#), and [timer\(\)](#).

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

Here is the call graph for this function:



Here is the caller 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 111 of file [SimulationFunctions.cpp](#).

```

00119
00120
00121 // MPI data
00122 int myPrc = 0, nPrc = 1; // Get process rank and number of processes
00123 #if defined(_MPI)
00124 double ts = MPI_Wtime();
00125 MPI_Comm_rank(MPI_COMM_WORLD,
00126                 &myPrc); // Return process rank, number \in [1,nPrc]
00127 MPI_Comm_size(MPI_COMM_WORLD,
00128                 &nPrc); // Return number of processes (communicator size)
00129
00130 // Check feasibility of the patchwork decomposition
00131 if (myPrc == 0) {
00132     if (nPrc != patches[0] * patches[1]) {
00133         errorKill(
00134             "The number of MPI processes must match the number of patches.");
00135     }
00136 }
00137 #elif !defined(_MPI) && defined(_OPENMP)
00138     double ts = omp_get_wtime();
00139 #endif
00140
00141 // Initialize the simulation, set up the cartesian communicator
00142 Simulation sim(patches[0], patches[1], 1, StencilOrder, periodic);
00143
00144 /* // Check that lattice communicator is unique; same as used in patchwork
00145 generation char cart_comm_name[MPI_MAX_OBJECT_NAME]; int cart_namelen;
00146 MPI_Comm_get_name(*sim.get_cart_comm(), cart_comm_name, &cart_namelen);

```

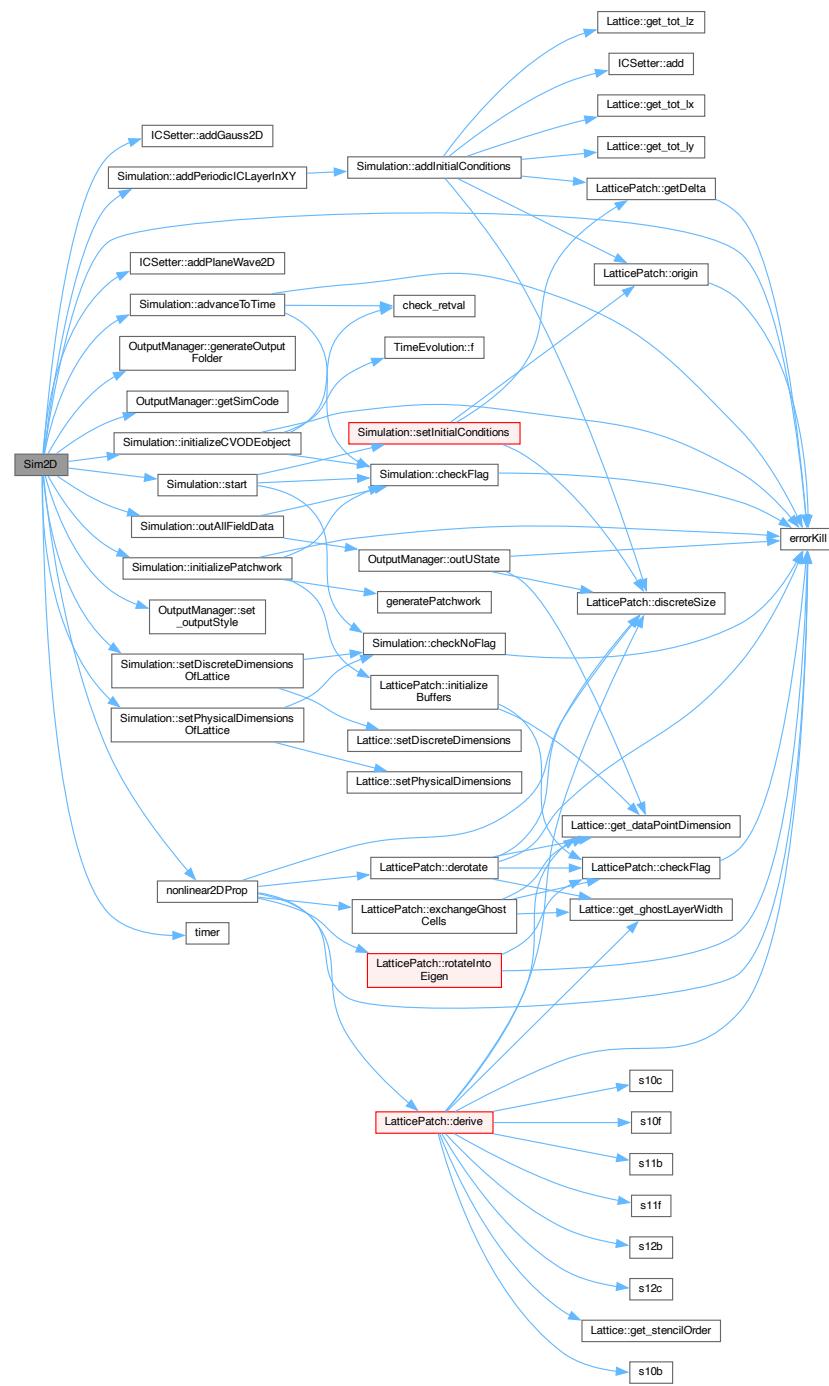
```

00147 printf("sim.get_cart_comm gives %s \n", cart_comm_name);
00148 */
00149
00150 // Configure the patchwork
00151 sim.setPhysicalDimensionsOfLattice(phys_dims[0],
00152                                     phys_dims[1],
00153                                     1); // spacing of the lattice
00154 sim.setDiscreteDimensionsOfLattice(
00155     disc_dims[0], disc_dims[1], 1); // Spacing equivalence to points
00156 sim.initializePatchwork(patches[0], patches[1], 1);
00157
00158 // Add em-waves
00159 for (const auto &gauss : gaussians)
00160     sim.icsettings.addGauss2D(gauss.x0, gauss.axis, gauss.amp, gauss.phip,
00161                               gauss.w0, gauss.zr, gauss.ph0, gauss.phA);
00162 for (const auto &plane : planes)
00163     sim.icsettings.addPlaneWave2D(plane.k, plane.p, plane.phi);
00164
00165 // Check that the patchwork is ready and set the initial conditions
00166 sim.start(); // Check if the lattice is set up, set initial field
00167           // configuration
00168 sim.addPeriodicICLayerInXY(); // insure periodicity in propagation directions
00169
00170 // Initialize CVode with rel and abs tolerances
00171 sim.initializeCVODEobject(CVodeTol[0], CVodeTol[1]);
00172
00173 // Configure the time evolution function
00174 TimeEvolution::c = interactions;
00175 TimeEvolution::TimeEvolver = nonlinear2DProp;
00176
00177 // Configure the output
00178 sim.outputManager.generateOutputFolder(outputDirectory);
00179 if (!myPrc) {
00180     std::cout << "Simulation code: " << sim.outputManager.getSimCode()
00181             << std::endl;
00182 }
00183 sim.outputManager.set_outputStyle(outputStyle);
00184
00185 //MPI_Barrier(MPI_COMM_WORLD); // insure correct output
00186 //sim.outAllFieldData(0); // output of initial state
00187 // Conduct the propagation in space and time
00188 for (int step = 1; step <= numberofSteps; step++) {
00189     sim.advanceToTime(endTime / numberofSteps * step);
00190     if (step % outputStep == 0) {
00191 #if defined(_MPI)
00192         MPI_Barrier(MPI_COMM_WORLD); // insure correct output
00193 #endif
00194         sim.outAllFieldData(step);
00195     }
00196 #if defined(_MPI)
00197     double tn = MPI_Wtime();
00198 #elif !defined(_MPI) && defined(_OPENMP)
00199     double tn = omp_get_wtime();
00200 #endif
00201     if (!myPrc) {
00202         std::cout << "\rStep " << step << "\t\t" << std::flush;
00203 #if defined(_MPI) || defined(_OPENMP)
00204         timer(ts, tn);
00205 #endif
00206     }
00207 }
00208
00209 return;
00210 }
```

References [ICSetter::addGauss2D\(\)](#), [Simulation::addPeriodicICLayerInXY\(\)](#), [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\(\)](#).

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

Here is the call graph for this function:



Here is the caller graph for this function:



6.26.2.3 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 213 of file [SimulationFunctions.cpp](#).

```

00221
00222
00223 // MPI data
00224 int myPrc = 0, nPrc = 1; // Get process rank and numer of process
00225 #if defined(_MPI)
00226 double ts = MPI_Wtime();
00227 MPI_Comm_rank(MPI_COMM_WORLD,
00228     &myPrc); // rank of the process inside the world communicator
00229 MPI_Comm_size(MPI_COMM_WORLD,
00230     &nPrc); // Size of the communicator is the number of processes
00231
00232 // Check feasibility of the patchwork decomposition
00233 if (myPrc == 0) {
00234     if (nPrc != patches[0] * patches[1] * patches[2]) {
00235         errorKill(
00236             "The number of MPI processes must match the number of patches.");
00237     }
00238     if ( ( disc_dims[0] / patches[0] != disc_dims[1] / patches[1] ) ||
00239         ( disc_dims[0] / patches[0] != disc_dims[2] / patches[2] ) ) {
00240         std::clog
00241             << "\nWarning: Patches should be cubic in terms of the lattice "
00242                 "points for the computational efficiency of larger simulations.\n";
00243     }
00244 }
00245 #elif !defined(_MPI) && defined(_OPENMP)
00246     double ts = omp_get_wtime();
00247 #endif
00248

```

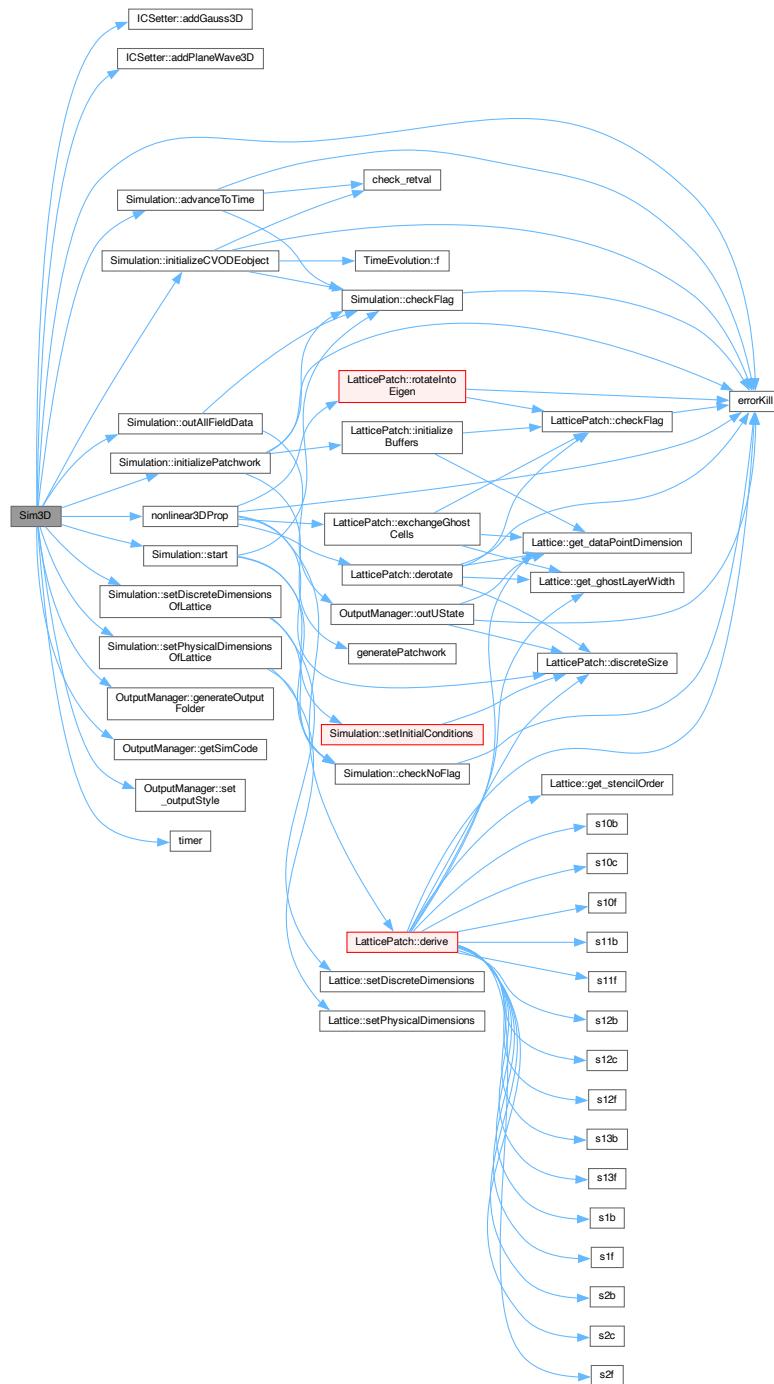
```

00249 // Initialize the simulation, set up the cartesian communicator
00250 Simulation sim(patches[0], patches[1], patches[2],
00251             StencilOrder, periodic); // Simulation object with slicing
00252
00253 // Create the SUNContext object associated with the thread of execution
00254 sim.setPhysicalDimensionsOfLattice(phys_dims[0], phys_dims[1],
00255                                         phys_dims[2]); // spacing of the box
00256 sim.setDiscreteDimensionsOfLattice(
00257     disc_dims[0], disc_dims[1],
00258     disc_dims[2]); // Spacing equivalence to points
00259 sim.initializePatchwork(patches[0], patches[1], patches[2]);
00260 //sim.initializeGhostCells();
00261
00262 // Add em-waves
00263 for (const auto &plane : planes)
00264     sim.icsettings.addPlaneWave3D(plane.k, plane.p, plane.phi);
00265 for (const auto &gauss : gaussians)
00266     sim.icsettings.addGauss3D(gauss.x0, gauss.axis, gauss.amp, gauss.phip,
00267                                gauss.w0, gauss.zr, gauss.ph0, gauss.phA);
00268
00269 // Check that the patchwork is ready and set the initial conditions
00270 sim.start();
00271
00272 // Initialize CVode with abs and rel tolerances
00273 sim.initializeCVODEobject(CVodeTol[0], CVodeTol[1]);
00274
00275 // Configure the time evolution function
00276 TimeEvolution::c = interactions;
00277 TimeEvolution::TimeEvolver = nonlinear3DProp;
00278
00279 // Configure the output
00280 sim.outputManager.generateOutputFolder(outputDirectory);
00281 if (!myPrc) {
00282     std::cout << "Simulation code: " << sim.outputManager.getSimCode()
00283             << std::endl;
00284 }
00285 sim.outputManager.set_outputStyle(outputStyle);
00286
00287 //MPI_Barrier(MPI_COMM_WORLD); // insure correct output
00288 //sim.outAllFieldData(0); // output of initial state
00289 // Conduct the propagation in space and time
00290 for (int step = 1; step <= numberofSteps; step++) {
00291     sim.advanceToTime(endTime / numberofSteps * step);
00292     if (step % outputStep == 0) {
00293 #if defined(_MPI)
00294         MPI_Barrier(MPI_COMM_WORLD); // insure correct output
00295 #endif
00296         sim.outAllFieldData(step);
00297     }
00298 #if defined(_MPI)
00299     double tn = MPI_Wtime();
00300 #elif !defined(_MPI) && defined(_OPENMP)
00301     double tn = omp_get_wtime();
00302 #endif
00303     if (!myPrc) {
00304         std::cout << "\rStep " << step << "\t\t" << std::flush;
00305 #if defined(_MPI) || defined(_OPENMP)
00306         timer(ts, tn);
00307 #endif
00308     }
00309 }
00310 return;
00311 }
```

References [ICSetter::addGauss3D\(\)](#), [ICSetter::addPlaneWave3D\(\)](#), [Simulation::advanceToTime\(\)](#), [TimeEvolution::c](#), [errorKill\(\)](#), [OutputManager::generateOutputFolder\(\)](#), [OutputManager::getSimCode\(\)](#), [Simulation::icsettings](#), [Simulation::initializeCVODEobject\(\)](#), [Simulation::initializePatchwork\(\)](#), [nonlinear3DProp\(\)](#), [Simulation::outAllFieldData\(\)](#), [Simulation::outputManager](#), [OutputManager::set_outputStyle\(\)](#), [Simulation::setDiscreteDimensionsOfLattice\(\)](#), [Simulation::setPhysicalDimensionsOfLattice\(\)](#), [Simulation::start\(\)](#), [TimeEvolution::TimeEvolver](#), and [timer\(\)](#).

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

Here is the call graph for this function:



Here is the caller graph for this function:



6.26.2.4 timer()

```
void timer (
    double & t1,
    double & t2 ) [inline]
```

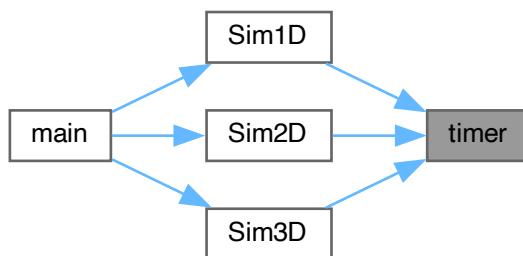
timer function

Calculate and print the elapsed wall 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<sunrealtyp, 3> k;    /**< wavevector (normalized to \f$ 1/\lambda \f$) */
00021     std::array<sunrealtyp, 3> p;    /**< amplitude & polarization vector */
00022     std::array<sunrealtyp, 3> phi;  /**< phase shift */
00023 };
00024
00025 /// 1D Gaussian wave structure
00026 struct gaussian1D {
00027     std::array<sunrealtyp, 3> k;    /**< wavevector (normalized to \f$ 1/\lambda \f$) */
00028     std::array<sunrealtyp, 3> p;    /**< amplitude & polarization vector */
00029     std::array<sunrealtyp, 3> x0;   /**< shift from origin */
00030     sunrealtyp phig;            /**< width */
00031     std::array<sunrealtyp, 3> phi;  /**< phase shift */
00032 };
00033
00034 /// 2D Gaussian wave structure
00035 struct gaussian2D {
00036     std::array<sunrealtyp, 3> x0;   /**< center */
00037     std::array<sunrealtyp, 3> axis; /**< direction from where it comes */
00038     sunrealtyp amp;              /**< amplitude */
00039     sunrealtyp phip;             /**< polarization rotation */
00040     sunrealtyp w0;               /**< taille */
00041     sunrealtyp zr;               /**< Rayleigh length */
00042     sunrealtyp ph0;              /**< beam center */
00043     sunrealtyp phA;              /**< beam length */
00044 };
00045
00046 /// 3D Gaussian wave structure
00047 struct gaussian3D {
00048     std::array<sunrealtyp, 3> x0;   /**< center */
00049     std::array<sunrealtyp, 3> axis; /**< direction from where it comes */
00050     sunrealtyp amp;              /**< amplitude */
00051     sunrealtyp phip;             /**< polarization rotation */
00052     sunrealtyp w0;               /**< taille */
00053     sunrealtyp zr;               /**< Rayleigh length */
00054     sunrealtyp ph0;              /**< beam center */
00055     sunrealtyp phA;              /**< beam length */
00056 };
00057
00058 /***** simulation function declarations *****/
00059
00060 /// complete 1D Simulation function
00061 void Sim1D(const std::array<sunrealtyp,2>, const int, const sunrealtyp,
00062             const sunindextyp, const bool, int *, const sunrealtyp, 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<sunrealtyp,2>, const int,
00068             const std::array<sunrealtyp,2>,
00069             const std::array<sunindextyp,2>, const std::array<int,2>,
00070             const bool, int *,
00071             const sunrealtyp, 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<sunrealtyp,2>, const int,
00076             const std::array<sunrealtyp,3>,
00077             const std::array<sunindextyp,3>, const std::array<int,3>,
00078             const bool, int *,
00079             const sunrealtyp, const int, const std::string,
00080             const int, const char,
00081             const std::vector<planewave> &, const std::vector<gaussian3D> &);
00082

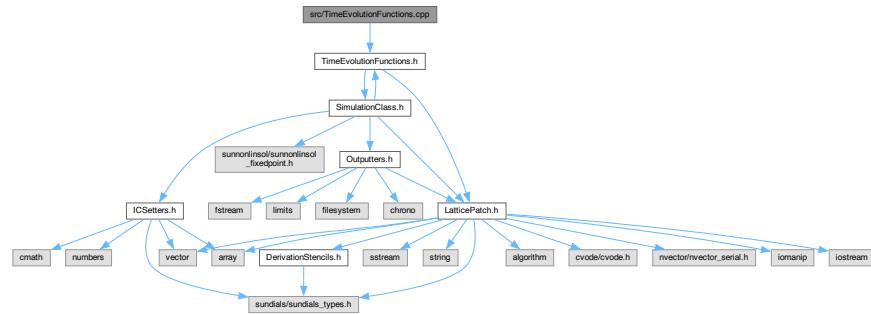
```

```
00083 /// 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;

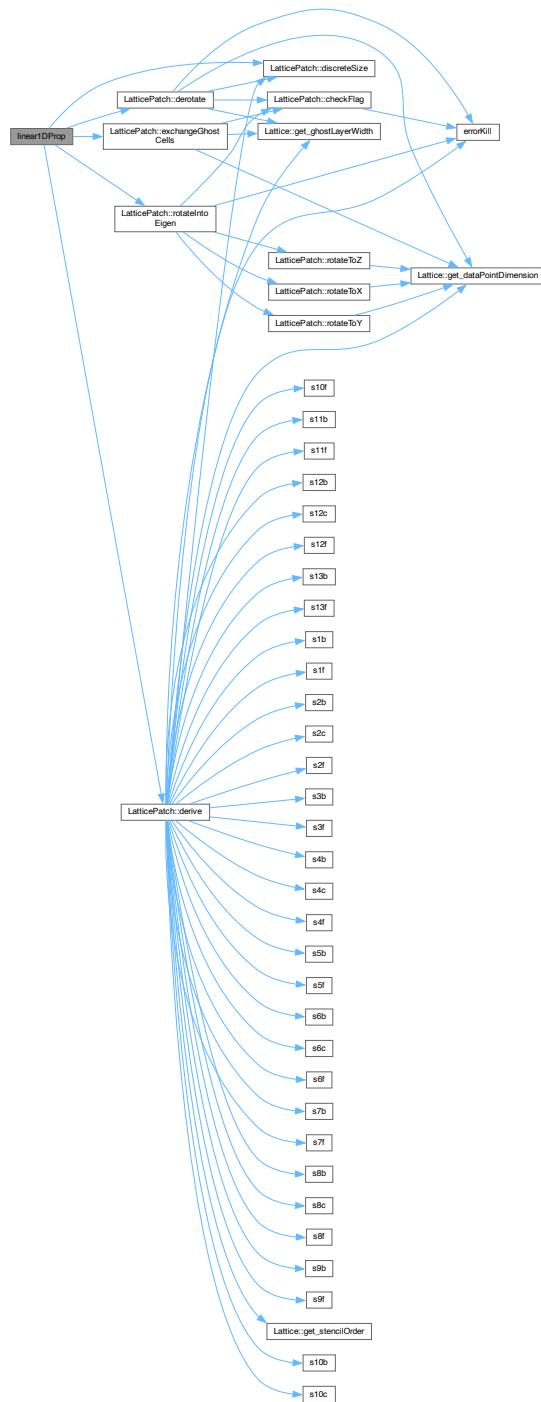
Maxwell propagation function for 1D – only for reference.

Definition at line 56 of file [TimeEvolutionFunctions.cpp](#).

```
00056
00057
00058     // pointers to temporal and spatial derivative data
00059     sunrealtype *duData = data->duData;
00060     sunrealtype *dxData = data->buffData[1 - 1];
00061
00062     // sequence along any dimension according to the scheme:
00063     data->exchangeGhostCells(1); // -> exchange halos
00064     data->rotateIntoEigen(
00065         1); // -> rotate all data to prepare derivative operation
00066     data->derive(1); // -> perform derivative approximation operation on it
00067     data->derotate(
00068         1, dxData); // -> derotate derived data for ensuing time-evolution
00069
00070     const sunindextype totalNP = data->discreteSize();
00071     sunindextype pp = 0;
00072     for (sunindextype i = 0; i < totalNP; i++) {
00073         pp = i * 6;
00074         /*
00075             simple vacuum Maxwell equations for the temporal derivatives using the
00076             spatial derivative only in x-direction without polarization or
00077             magnetization terms
00078         */
00079         duData[pp + 0] = 0;
00080         duData[pp + 1] = -dxData[pp + 5];
00081         duData[pp + 2] = dxData[pp + 4];
00082         duData[pp + 3] = 0;
00083         duData[pp + 4] = dxData[pp + 2];
00084         duData[pp + 5] = -dxData[pp + 1];
00085     }
00086 }
```

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

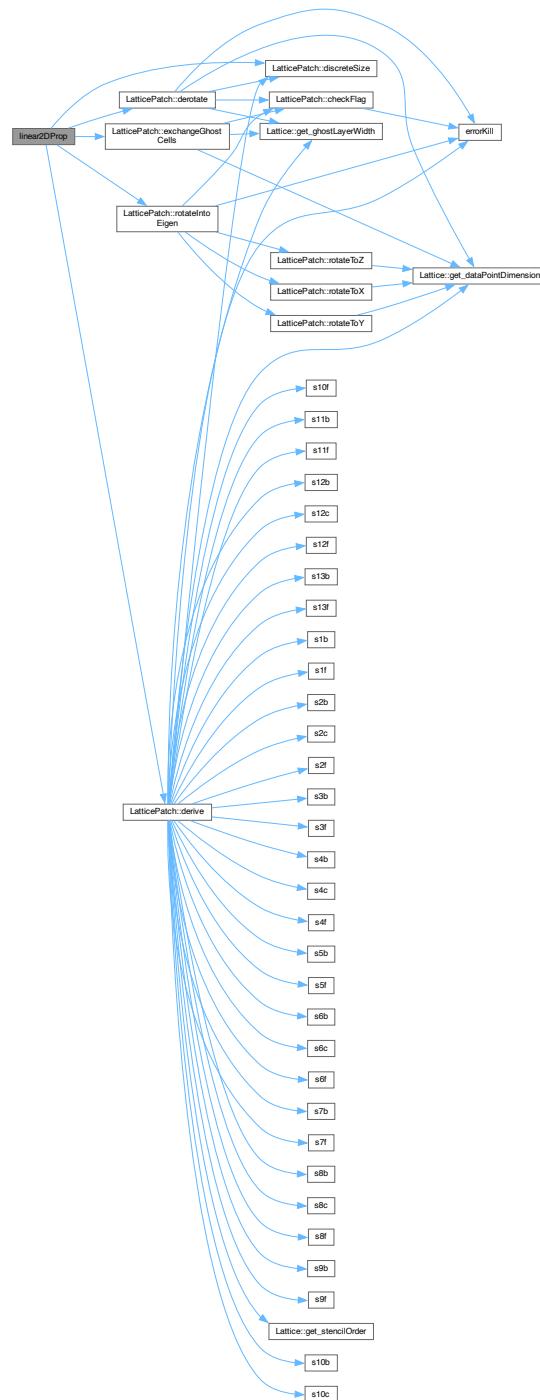
Maxwell propagation function for 2D – only for reference.

Definition at line 312 of file [TimeEvolutionFunctions.cpp](#).

```
00312
00313
00314     sunrealtype *duData = data->duData;
00315     sunrealtype *dxData = data->buffData[1 - 1];
00316     sunrealtype *dyData = data->buffData[2 - 1];
00317
00318     data->exchangeGhostCells(1);
00319     data->rotateIntoEigen(1);
00320     data->derive(1);
00321     data->derotate(1, dxData);
00322     data->exchangeGhostCells(2);
00323     data->rotateIntoEigen(2);
00324     data->derive(2);
00325     data->derotate(2, dyData);
00326
00327     const sunindextype totalNP = data->discreteSize();
00328     sunindextype pp = 0;
00329     for (sunindextype i = 0; i < totalNP; i++) {
00330         pp = i * 6;
00331         duData[pp + 0] = dyData[pp + 5];
00332         duData[pp + 1] = -dxData[pp + 5];
00333         duData[pp + 2] = -dyData[pp + 3] + dxData[pp + 4];
00334         duData[pp + 3] = -dyData[pp + 2];
00335         duData[pp + 4] = dxData[pp + 2];
00336         duData[pp + 5] = dyData[pp + 0] - dxData[pp + 1];
00337     }
00338 }
```

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

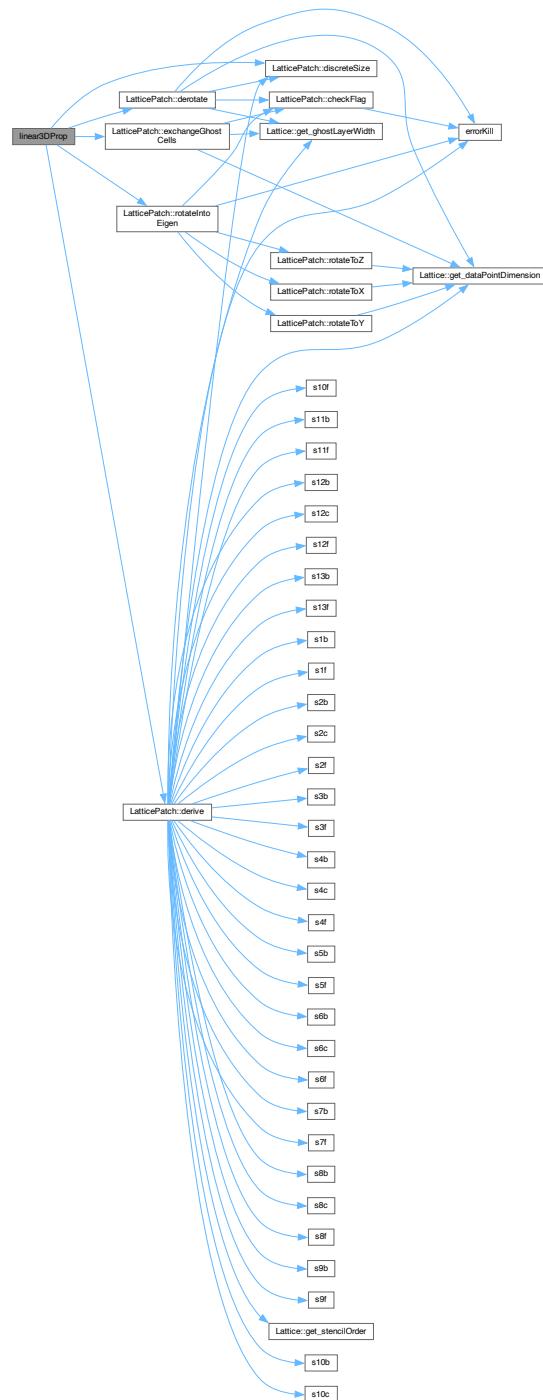
Maxwell propagation function for 3D – only for reference.

Definition at line 533 of file [TimeEvolutionFunctions.cpp](#).

```
00533
00534
00535     sunrealtype *duData = data->duData;
00536     sunrealtype *dxData = data->buffData[1 - 1];
00537     sunrealtype *dyData = data->buffData[2 - 1];
00538     sunrealtype *dzData = data->buffData[3 - 1];
00539
00540     data->exchangeGhostCells(1);
00541     data->rotateIntoEigen(1);
00542     data->derive(1);
00543     data->derotate(1, dxData);
00544     data->exchangeGhostCells(2);
00545     data->rotateIntoEigen(2);
00546     data->derive(2);
00547     data->derotate(2, dyData);
00548     data->exchangeGhostCells(3);
00549     data->rotateIntoEigen(3);
00550     data->derive(3);
00551     data->derotate(3, dzData);
00552
00553     const sunindextype totalNP = data->discreteSize();
00554     sunindextype pp = 0;
00555     for (sunindextype i = 0; i < totalNP; i++) {
00556         pp = i * 6;
00557         duData[pp + 0] = dyData[pp + 5] - dzData[pp + 4];
00558         duData[pp + 1] = dzData[pp + 3] - dxData[pp + 5];
00559         duData[pp + 2] = dxData[pp + 4] - dyData[pp + 3];
00560         duData[pp + 3] = -dyData[pp + 2] + dzData[pp + 1];
00561         duData[pp + 4] = -dzData[pp + 0] + dxData[pp + 2];
00562         duData[pp + 5] = -dxData[pp + 1] + dyData[pp + 0];
00563     }
00564 }
```

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 89 of file [TimeEvolutionFunctions.cpp](#).

```
00089
00090
00091 // NVector pointers to provided field values and their temp. derivatives
00092 #if defined(_MPI)
00093 #if !defined(_OPENMP)
00094     sunrealtype *udata = NV_DATA_P(u),
00095             *dudata = NV_DATA_P(udot);
00096 #elif defined(_OPENMP)
00097     sunrealtype *udata = N_VGetArrayPointer_MPIPlusX(u),
00098             *dudata = N_VGetArrayPointer_MPIPlusX(udot);
00099 #endif
00100 #elif defined(_OPENMP)
00101     sunrealtype *udata = NV_DATA_OMP(u),
00102             *dudata = NV_DATA_OMP(udot);
00103 #else
00104     sunrealtype *udata = NV_DATA_S(u),
00105             *dudata = NV_DATA_S(udot);
00106 #endif
00107
00108 // pointer to spatial derivatives via patch data
00109 sunrealtype *dxData = data->buffData[1 - 1];
00110
00111 // same sequence as in the linear case
00112 data->exchangeGhostCells(1);
00113 data->rotateIntoEigen(1);
00114 data->derive(1);
00115 data->derotate(1, dxData);
00116
00117 /*
00118 F and G are nonzero in the nonlinear case,
00119 polarization and magnetization derivatives
00120 w.r.t. E- and B-field go into the e.o.m.
00121 */
00122 static sunrealtype f, g; // em field invariants F, G
00123 // derivatives of HE Lagrangian w.r.t. field invariants
00124 static sunrealtype lf, lff, lfg, lg, lgg;
00125 // matrix to hold derivatives of polarization and magnetization
00126 static std::array<sunrealtype, 21> JMM;
00127 // array to hold E^2 and B^2 components
00128 static std::array<sunrealtype, 6> Quad;
00129 // array to hold intermediate temp. derivatives of E and B
00130 static std::array<sunrealtype, 6> h;
00131 // determinant needed for explicit matrix inversion
00132 static sunrealtype detC = nan("0x12345");
00133
00134 // number of points in the patch
00135 const sunindextype totalNP = data->discreteSize();
00136 #pragma omp parallel for default(none) \
00137 private(JMM, Quad, h, detC) \
00138 shared(totalNP, c, f, g, lf, lff, lfg, lg, lgg, udata, dudata, dxData) \
00139 schedule(static)
00140 for (sunindextype pp = 0; pp < totalNP * 6;
00141     pp += 6) { // loop over all 6dim points in the patch
00142     // em field Lorentz invariants F and G
00143     f = 0.5 * ((Quad[0] = udata[pp] * udata[pp]) +
00144                 (Quad[1] = udata[pp + 1] * udata[pp + 1]) +
00145                 (Quad[2] = udata[pp + 2] * udata[pp + 2]) -
00146                 (Quad[3] = udata[pp + 3] * udata[pp + 3]) -
00147                 (Quad[4] = udata[pp + 4] * udata[pp + 4]) -
00148                 (Quad[5] = udata[pp + 5] * udata[pp + 5]));
00149     g = udata[pp] * udata[pp + 3] + udata[pp + 1] * udata[pp + 4] +
00150         udata[pp + 2] * udata[pp + 5];
00151     // process/expansion order and corresponding derivative values of L
00152     // w.r.t. F, G
00153     switch (*c) {
00154     case 0: // linear Maxwell vacuum
00155         lf = 0;
00156         lff = 0;
00157         lfg = 0;
00158         lg = 0;
00159         lgg = 0;
00160         break;
00161     case 1: // only 4-photon processes
00162         lf = 0.000206527095658582755255648 * f;
```

```

00163     lff = 0.000206527095658582755255648;
00164     lfg = 0;
00165     lg = 0.0003614224174025198216973841 * g;
00166     lgg = 0.0003614224174025198216973841;
00167     break;
00168 case 2: // only 6-photon processes
00169     lf = 0.000354046449700427580438254 * f * f +
00170         0.000191775160254398272737387 * g * g;
00171     lff = 0.0007080928994008551608765075 * f;
00172     lfg = 0.0003835503205087965454747749 * g;
00173     lg = 0.0003835503205087965454747749 * f * g;
00174     lgg = 0.0003835503205087965454747749 * f;
00175     break;
00176 case 3: // 4- and 6-photon processes
00177     lf = (0.000206527095658582755255648 + 0.000354046449700427580438254 * f) *
00178         f +
00179         0.000191775160254398272737387 * g * g;
00180     lff = 0.000206527095658582755255648 + 0.0007080928994008551608765075 * f;
00181     lfg = 0.0003835503205087965454747749 * g;
00182     lg = (0.0003614224174025198216973841 +
00183         0.0003835503205087965454747749 * f) *
00184         g;
00185     lgg = 0.0003614224174025198216973841 + 0.0003835503205087965454747749 * f;
00186     break;
00187 default:
00188     errorKill(
00189         "You need to specify a correct order in the weak-field expansion.");
00190 }
00191
00192 // derivatives of polarization and magnetization w.r.t. E and B
00193 // Jpx(Ex)
00194 JMM[0] = lf + lff * Quad[0] +
00195     udata[3 + pp] * (2 * lfg * udata[pp] + lgg * udata[3 + pp]);
00196 // Jpx(Ey)
00197 JMM[1] =
00198     lff * udata[pp] * udata[1 + pp] + lfg * udata[1 + pp] * udata[3 + pp] +
00199     lfg * udata[pp] * udata[4 + pp] + lgg * udata[3 + pp] * udata[4 + pp];
00200 // Jpy(Ey)
00201 JMM[2] = lf + lff * Quad[1] +
00202     udata[4 + pp] * (2 * lfg * udata[1 + pp] + lgg * udata[4 + pp]);
00203 // Jpx(Ez) = Jpz(Ex)
00204 JMM[3] =
00205     lff * udata[pp] * udata[2 + pp] + lfg * udata[2 + pp] * udata[3 + pp] +
00206     lfg * udata[pp] * udata[5 + pp] + lgg * udata[3 + pp] * udata[5 + pp];
00207 // Jpy(Ez) = Jpz(Ey)
00208 JMM[4] = lff * udata[1 + pp] * udata[2 + pp] +
00209     lfg * udata[2 + pp] * udata[4 + pp] +
00210     lfg * udata[1 + pp] * udata[5 + pp] +
00211     lgg * udata[4 + pp] * udata[5 + pp];
00212 // Jpz(Ez)
00213 JMM[5] = lf + lff * Quad[2] +
00214     udata[5 + pp] * (2 * lfg * udata[2 + pp] + lgg * udata[5 + pp]);
00215 // Jpx(Bx) = Jmx(Ex)
00216 JMM[6] = lg + lfg * (Quad[0] - Quad[3 + 0]) +
00217     (-lff + lgg) * udata[pp] * udata[3 + pp];
00218 // Jpy(Bx) = Jmx(Ey)
00219 JMM[7] = -(udata[3 + pp] * (lff * udata[1 + pp] + lfg * udata[4 + pp])) +
00220     udata[pp] * (lfg * udata[1 + pp] + lgg * udata[4 + pp]);
00221 // Jpz(Bx) = Jmx(Ez)
00222 JMM[8] = -(udata[3 + pp] * (lff * udata[2 + pp] + lfg * udata[5 + pp])) +
00223     udata[pp] * (lfg * udata[2 + pp] + lgg * udata[5 + pp]);
00224 // Jmx(Bx)
00225 JMM[9] = -lf + lgg * Quad[0] +
00226     udata[3 + pp] * (-2 * lfg * udata[pp] + lff * udata[3 + pp]);
00227 // Jpx(By) = Jmy(Ex)
00228 JMM[10] = udata[1 + pp] * (lfg * udata[pp] + lgg * udata[3 + pp]) -
00229     (lff * udata[pp] + lfg * udata[3 + pp]) * udata[4 + pp];
00230 // Jpy(By) = Jmy(Ey)
00231 JMM[11] = lg + lfg * (Quad[1] - Quad[4 + 0]) +
00232     (-lff + lgg) * udata[1 + pp] * udata[4 + pp];
00233 // Jpz(By) = Jmy(Ez)
00234 JMM[12] = -(udata[4 + pp] * (lff * udata[2 + pp] + lfg * udata[5 + pp])) +
00235     udata[1 + pp] * (lfg * udata[2 + pp] + lgg * udata[5 + pp]);
00236 // Jmx(By) = Jmy(Bx)
00237 JMM[13] = lgg * udata[pp] * udata[1 + pp] +
00238     lff * udata[3 + pp] * udata[4 + pp] -
00239     lfg * (udata[1 + pp] * udata[3 + pp] + udata[pp] * udata[4 + pp]);
00240 // Jmy(By)
00241 JMM[14] = -lf + lgg * Quad[1] +
00242     udata[4 + pp] * (-2 * lfg * udata[1 + pp] + lff * udata[4 + pp]);
00243 // Jmz(Ex) = Jpx(Bz)
00244 JMM[15] = udata[2 + pp] * (lfg * udata[pp] + lgg * udata[3 + pp]) -
00245     (lff * udata[pp] + lfg * udata[3 + pp]) * udata[5 + pp];
00246 // Jmz(Ey) = Jpy(Bz)
00247 JMM[16] = udata[2 + pp] * (lfg * udata[1 + pp] + lgg * udata[4 + pp]) -
00248     (lff * udata[1 + pp] + lfg * udata[4 + pp]) * udata[5 + pp];
00249 // Jpz(Bz) = Jmz(Ez)

```

```

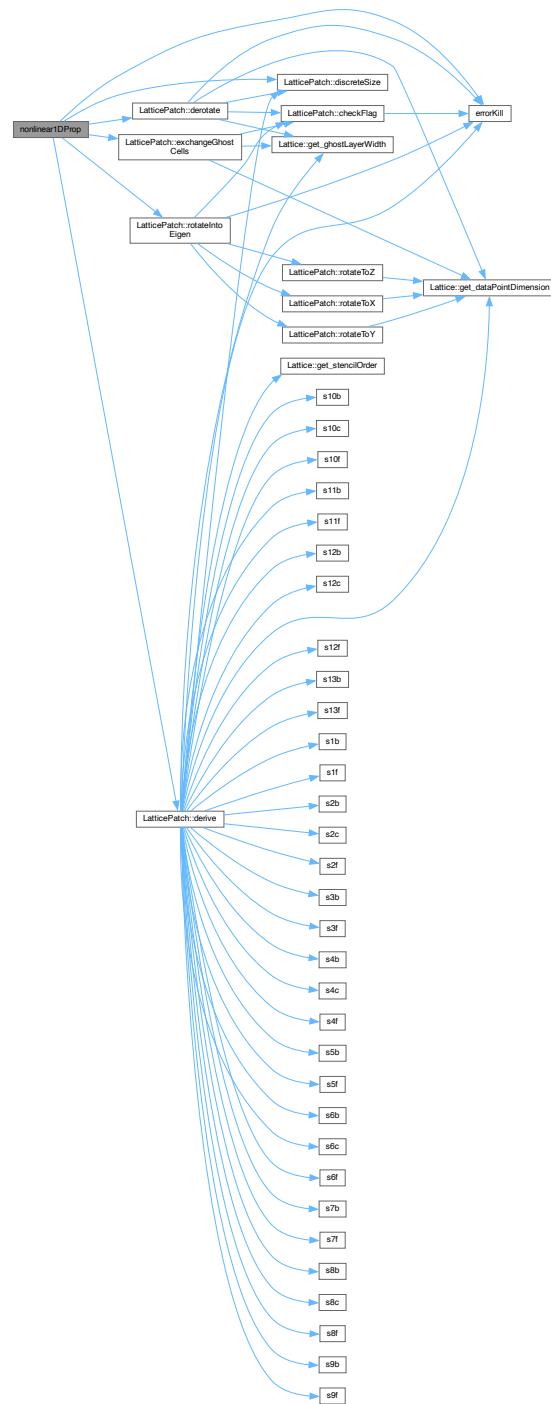
00250     JMM[17] = lgg + lfg * (Quad[2] - Quad[5 + 0]) +
00251         (-lff + lgg) * udata[2 + pp] * udata[5 + pp];
00252 // Jmz(Bx) = Jmx(Bz)
00253     JMM[18] = lgg * udata[pp] * udata[2 + pp] +
00254         lff * udata[3 + pp] * udata[5 + pp] -
00255         lfg * (udata[2 + pp] * udata[3 + pp] + udata[pp] * udata[5 + pp]);
00256 // Jmy(Bz) = Jmz(By)
00257     JMM[19] =
00258         lgg * udata[1 + pp] * udata[2 + pp] +
00259         lff * udata[4 + pp] * udata[5 + pp] -
00260         lfg * (udata[2 + pp] * udata[4 + pp] + udata[1 + pp] * udata[5 + pp]);
00261 // Jmz(Bz)
00262     JMM[20] = -lff + lgg * Quad[2] +
00263         udata[5 + pp] * (-2 * lfg * udata[2 + pp] + lff * udata[5 + pp]);
00264
00265 // apply Z
00266 // top block: -QJm(E)*E, Q-QJm(B)*B
00267     h[0] = 0;
00268     h[1] = dxData[pp] * JMM[15] + dxData[1 + pp] * JMM[16] +
00269         dxData[2 + pp] * JMM[17] + dxData[3 + pp] * JMM[18] +
00270         dxData[4 + pp] * JMM[19] + dxData[5 + pp] * (-1 + JMM[20]);
00271     h[2] = -(dxData[pp] * JMM[10]) - dxData[1 + pp] * JMM[11] -
00272         dxData[2 + pp] * JMM[12] - dxData[3 + pp] * JMM[13] +
00273         dxData[4 + pp] * (1 - JMM[14]) - dxData[5 + pp] * JMM[19];
00274 // bottom blocks: -Q*E
00275     h[3] = 0;
00276     h[4] = dxData[2 + pp];
00277     h[5] = -dxData[1 + pp];
00278 // (1+A)^-1 applies only to E components
00279 // -Jp(B)*B
00280     h[0] -= h[3] * JMM[6] + h[4] * JMM[10] + h[5] * JMM[15];
00281     h[1] -= h[3] * JMM[7] + h[4] * JMM[11] + h[5] * JMM[16];
00282     h[2] -= h[3] * JMM[8] + h[4] * JMM[12] + h[5] * JMM[17];
00283 // apply C^-1 explicitly, with C=1+Jp(E)
00284     dudata[pp + 0] =
00285         h[2] * (-JMM[3] * (1 + JMM[2])) + JMM[1] * JMM[4]) +
00286         h[1] * (JMM[3] * JMM[4] - JMM[1] * (1 + JMM[5])) +
00287         h[0] * (1 - JMM[4] * JMM[4] + JMM[5] + JMM[2] * (1 + JMM[5]));
00288     dudata[pp + 1] =
00289         h[2] * (JMM[3] * JMM[1] - (1 + JMM[0]) * JMM[4]) +
00290         h[1] * (1 - JMM[3] * JMM[3] + JMM[5] + JMM[0] * (1 + JMM[5])) +
00291         h[0] * (JMM[4] * JMM[3] - JMM[1] * (1 + JMM[5]));
00292     dudata[pp + 2] =
00293         h[2] * (1 - JMM[1] * JMM[1] + JMM[2] + JMM[0] * (1 + JMM[2])) +
00294         h[1] * (JMM[1] * JMM[3] - (1 + JMM[0]) * JMM[4]) +
00295         h[0] * (((1 + JMM[2]) * JMM[3]) + JMM[1] * JMM[4]);
00296     detC = // determinant of C
00297         -((1 + JMM[2]) * (-1 + JMM[3] * JMM[3])) +
00298         ((JMM[3] * JMM[1] - JMM[4]) * JMM[4] + JMM[5] + JMM[2] * JMM[5] +
00299             JMM[0] * (1 + JMM[2] - JMM[4] * JMM[4] + (1 + JMM[2]) * JMM[5]) -
00300             JMM[1] * (-JMM[4] * JMM[3]) + JMM[1] * (1 + JMM[5]));
00301     dudata[pp + 0] /= detC;
00302     dudata[pp + 1] /= detC;
00303     dudata[pp + 2] /= detC;
00304     dudata[pp + 3] = h[3];
00305     dudata[pp + 4] = h[4];
00306     dudata[pp + 5] = h[5];
00307 }
00308 return;
00309 }

```

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 341 of file [TimeEvolutionFunctions.cpp](#).

```

00341
00342
00343 #if defined(_MPI)
00344 #if !defined(_OPENMP)
00345     sunrealtype *udata = NV_DATA_P(u),
00346             *dudata = NV_DATA_P(udot);
00347 #elif defined(_OPENMP)
00348     sunrealtype *udata = N_VGetArrayPointer_MPIPlusX(u),
00349             *dudata = N_VGetArrayPointer_MPIPlusX(udot);
00350 #endif
00351 #elif defined(_OPENMP)
00352     sunrealtype *udata = NV_DATA_OMP(u),
00353             *dudata = NV_DATA_OMP(udot);
00354 #else
00355     sunrealtype *udata = NV_DATA_S(u),
00356             *dudata = NV_DATA_S(udot);
00357 #endif
00358
00359     sunrealtype *dxData = data->buffData[1 - 1];
00360     sunrealtype *dyData = data->buffData[2 - 1];
00361
00362     data->exchangeGhostCells(1);
00363     data->rotateIntoEigen(1);
00364     data->derive(1);
00365     data->derotate(1, dxData);
00366     data->exchangeGhostCells(2);
00367     data->rotateIntoEigen(2);
00368     data->derive(2);
00369     data->derotate(2, dyData);
00370
00371     static sunrealtype f, g;
00372     static sunrealtype lf, lff, lfg, lg, lgg;
00373     static std::array<sunrealtype, 21> JMM;
00374     static std::array<sunrealtype, 6> Quad;
00375     static std::array<sunrealtype, 6> h;
00376     static sunrealtype detC;
00377
00378     const sunindextype totalNP = data->discreteSize();
00379     #pragma omp parallel for default(none) \
00380     private(JMM, Quad, h, detC) \
00381     shared(totalNP, c, f, g, lf, lff, lfg, lg, lgg, udata, dudata, \
00382           dxData, dyData) \
00383     schedule(static)
  
```

```

00384     for (sunindextype pp = 0; pp < totalNP * 6; pp += 6) {
00385         f = 0.5 * ((Quad[0] = udata[pp] * udata[pp]) +
00386                     (Quad[1] = udata[pp + 1] * udata[pp + 1]) +
00387                     (Quad[2] = udata[pp + 2] * udata[pp + 2]) -
00388                     (Quad[3] = udata[pp + 3] * udata[pp + 3]) -
00389                     (Quad[4] = udata[pp + 4] * udata[pp + 4]) -
00390                     (Quad[5] = udata[pp + 5] * udata[pp + 5]));
00391         g = udata[pp] * udata[pp + 3] + udata[pp + 1] * udata[pp + 4] +
00392             udata[pp + 2] * udata[pp + 5];
00393         switch (*c) {
00394             case 0:
00395                 lf = 0;
00396                 lff = 0;
00397                 lfg = 0;
00398                 lg = 0;
00399                 lgg = 0;
00400                 break;
00401             case 1:
00402                 lf = 0.000206527095658582755255648 * f;
00403                 lff = 0.000206527095658582755255648;
00404                 lfg = 0;
00405                 lg = 0.0003614224174025198216973841 * g;
00406                 lgg = 0.0003614224174025198216973841;
00407                 break;
00408             case 2:
00409                 lf = 0.000354046449700427580438254 * f * f +
00410                     0.000191775160254398272737387 * g * g;
00411                 lff = 0.0007080928994008551608765075 * f;
00412                 lfg = 0.0003835503205087965454747749 * g;
00413                 lg = 0.0003835503205087965454747749 * f * g;
00414                 lgg = 0.0003835503205087965454747749 * f;
00415                 break;
00416             case 3:
00417                 lf = (0.000206527095658582755255648 + 0.000354046449700427580438254 * f) *
00418                     f +
00419                     0.000191775160254398272737387 * g * g;
00420                 lff = 0.000206527095658582755255648 + 0.000708092899400855160876508 * f;
00421                 lfg = 0.0003835503205087965454747749 * g;
00422                 lg = (0.000361422417402519821697384 + 0.000383550320508796545474775 * f) *
00423                     g;
00424                 lgg = 0.000361422417402519821697384 + 0.000383550320508796545474775 * f;
00425                 break;
00426             default:
00427                 errorKill(
00428                     "You need to specify a correct order in the weak-field expansion.");
00429     }
00430
00431     JMM[0] = lf + lff * Quad[0] +
00432             udata[3 + pp] * (2 * lfg * udata[pp] + lgg * udata[3 + pp]);
00433     JMM[1] =
00434         lff * udata[pp] * udata[1 + pp] + lfg * udata[1 + pp] * udata[3 + pp] +
00435         lfg * udata[pp] * udata[4 + pp] + lgg * udata[3 + pp] * udata[4 + pp];
00436     JMM[2] = lf + lff * Quad[1] +
00437         udata[4 + pp] * (2 * lfg * udata[1 + pp] + lgg * udata[4 + pp]);
00438     JMM[3] =
00439         lff * udata[pp] * udata[2 + pp] + lfg * udata[2 + pp] * udata[3 + pp] +
00440         lfg * udata[pp] * udata[5 + pp] + lgg * udata[3 + pp] * udata[5 + pp];
00441     JMM[4] = lff * udata[1 + pp] * udata[2 + pp] +
00442         lfg * udata[2 + pp] * udata[4 + pp] +
00443         lfg * udata[1 + pp] * udata[5 + pp] +
00444         lgg * udata[4 + pp] * udata[5 + pp];
00445     JMM[5] = lf + lff * Quad[2] +
00446         udata[5 + pp] * (2 * lfg * udata[2 + pp] + lgg * udata[5 + pp]);
00447     JMM[6] = lg + lfg * (Quad[0] - Quad[3 + 0]) +
00448         (-lff + lgg) * udata[pp] * udata[3 + pp];
00449     JMM[7] = -(udata[3 + pp] * (lff * udata[1 + pp] + lfg * udata[4 + pp])) +
00450         udata[pp] * (lfg * udata[1 + pp] + lgg * udata[4 + pp]);
00451     JMM[8] = -(udata[3 + pp] * (lff * udata[2 + pp] + lfg * udata[5 + pp])) +
00452         udata[pp] * (lfg * udata[2 + pp] + lgg * udata[5 + pp]);
00453     JMM[9] = -lf + lfg * Quad[2] +
00454         udata[3 + pp] * (-2 * lfg * udata[pp] + lff * udata[3 + pp]);
00455     JMM[10] = udata[1 + pp] * (lfg * udata[pp] + lgg * udata[3 + pp]) -
00456         (lff * udata[pp] + lfg * udata[3 + pp]) * udata[4 + pp];
00457     JMM[11] = lg + lfg * (Quad[1] - Quad[4 + 0]) +
00458         (-lff + lgg) * udata[1 + pp] * udata[4 + pp];
00459     JMM[12] = -(udata[4 + pp] * (lff * udata[2 + pp] + lfg * udata[5 + pp])) +
00460         udata[1 + pp] * (lfg * udata[2 + pp] + lgg * udata[5 + pp]);
00461     JMM[13] = lgg * udata[pp] * udata[1 + pp] +
00462         lff * udata[3 + pp] * udata[4 + pp] -
00463         lfg * (udata[1 + pp] * udata[3 + pp] + udata[pp] * udata[4 + pp]);
00464     JMM[14] = -lf + lgg * Quad[1] +
00465         udata[4 + pp] * (-2 * lfg * udata[1 + pp] + lff * udata[4 + pp]);
00466     JMM[15] = udata[2 + pp] * (lfg * udata[pp] + lgg * udata[3 + pp]) -
00467         (lff * udata[pp] + lfg * udata[3 + pp]) * udata[5 + pp];
00468     JMM[16] = udata[2 + pp] * (lfg * udata[1 + pp] + lgg * udata[4 + pp]) -
00469         (lff * udata[1 + pp] + lfg * udata[4 + pp]) * udata[5 + pp];
00470     JMM[17] = lg + lfg * (Quad[2] - Quad[5 + 0]) +

```

```

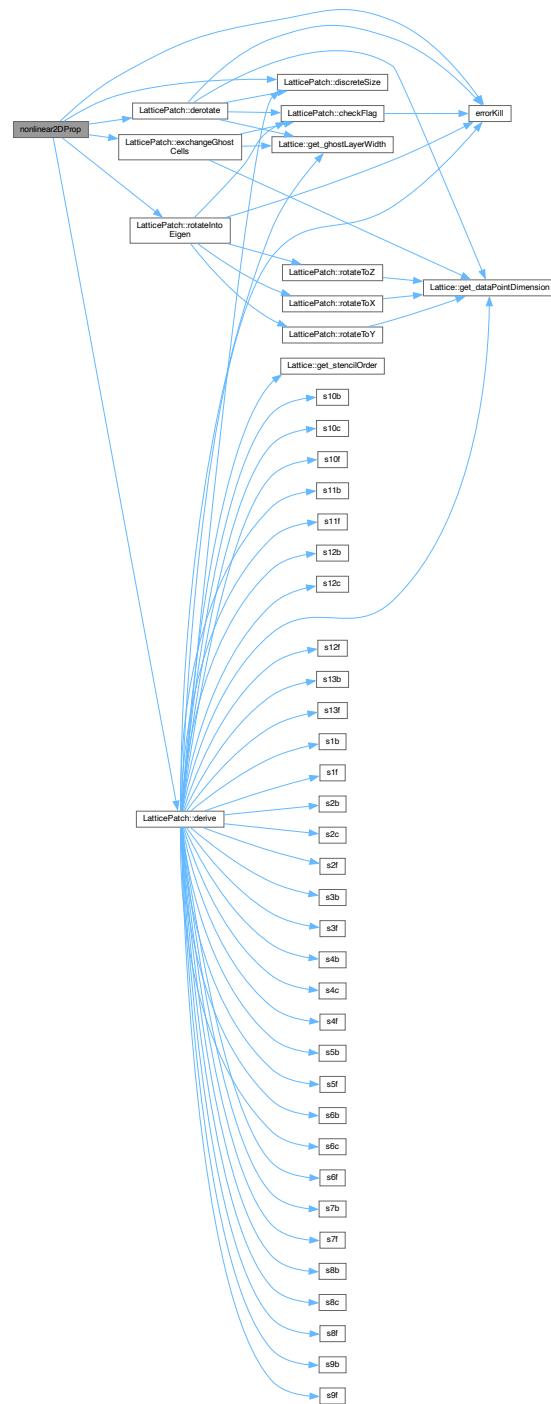
00471      (-lff + lgg) * udata[2 + pp] * udata[5 + pp];
00472 JMM[18] = lgg * udata[pp] * udata[2 + pp] +
00473     lff * udata[3 + pp] * udata[5 + pp] -
00474     lfg * (udata[2 + pp] * udata[3 + pp] + udata[pp] * udata[5 + pp]);
00475 JMM[19] =
00476     lgg * udata[1 + pp] * udata[2 + pp] +
00477     lff * udata[4 + pp] * udata[5 + pp] -
00478     lfg * (udata[2 + pp] * udata[4 + pp] + udata[1 + pp] * udata[5 + pp]);
00479 JMM[20] = -lf + lgg * Quad[2] +
00480     udata[5 + pp] * (-2 * lfg * udata[2 + pp] + lff * udata[5 + pp]);
00481
00482 h[0] = 0;
00483 h[1] = dxData[pp] * JMM[15] + dxData[1 + pp] * JMM[16] +
00484     dxData[2 + pp] * JMM[17] + dxData[3 + pp] * JMM[18] +
00485     dxData[4 + pp] * JMM[19] + dxData[5 + pp] * (-1 + JMM[20]);
00486 h[2] = -(dxData[pp] * JMM[10]) - dxData[1 + pp] * JMM[11] -
00487     dxData[2 + pp] * JMM[12] - dxData[3 + pp] * JMM[13] +
00488     dxData[4 + pp] * (1 - JMM[14]) - dxData[5 + pp] * JMM[19];
00489 h[3] = 0;
00490 h[4] = dxData[2 + pp];
00491 h[5] = -dxData[1 + pp];
00492 h[0] += -(dyData[pp] * JMM[15]) - dyData[1 + pp] * JMM[16] -
00493     dyData[2 + pp] * JMM[17] - dyData[3 + pp] * JMM[18] -
00494     dyData[4 + pp] * JMM[19] + dyData[5 + pp] * (1 - JMM[20]);
00495 h[1] += 0;
00496 h[2] += dyData[pp] * JMM[6] + dyData[1 + pp] * JMM[7] +
00497     dyData[2 + pp] * JMM[8] + dyData[3 + pp] * (-1 + JMM[9]) +
00498     dyData[4 + pp] * JMM[13] + dyData[5 + pp] * JMM[18];
00499 h[3] += -dyData[2 + pp];
00500 h[4] += 0;
00501 h[5] += dyData[pp];
00502 h[0] -= h[3] * JMM[6] + h[4] * JMM[10] + h[5] * JMM[15];
00503 h[1] -= h[3] * JMM[7] + h[4] * JMM[11] + h[5] * JMM[16];
00504 h[2] -= h[3] * JMM[8] + h[4] * JMM[12] + h[5] * JMM[17];
00505 dudata[pp + 0] =
00506     h[2] * (-JMM[3] * (1 + JMM[2])) + JMM[1] * JMM[4]) +
00507     h[1] * (JMM[3] * JMM[4] - JMM[1] * (1 + JMM[5])) +
00508     h[0] * (1 - JMM[4] * JMM[4] + JMM[5] + JMM[2] * (1 + JMM[5]));
00509 dudata[pp + 1] =
00510     h[2] * (JMM[3] * JMM[1] - (1 + JMM[0]) * JMM[4]) +
00511     h[1] * (1 - JMM[3] * JMM[3] + JMM[5] + JMM[0] * (1 + JMM[5])) +
00512     h[0] * (JMM[4] * JMM[3] - JMM[1] * (1 + JMM[5]));
00513 dudata[pp + 2] =
00514     h[2] * (1 - JMM[1] * JMM[1] + JMM[2] + JMM[0] * (1 + JMM[2])) +
00515     h[1] * (JMM[1] * JMM[3] - (1 + JMM[0]) * JMM[4]) +
00516     h[0] * (((1 + JMM[2]) * JMM[3]) + JMM[1] * JMM[4]);
00517 detC =
00518     -((1 + JMM[2]) * (-1 + JMM[3] * JMM[3])) +
00519     (JMM[3] * JMM[1] - JMM[4]) * JMM[4] + JMM[5] + JMM[2] * JMM[5] +
00520     JMM[0] * (1 + JMM[2] - JMM[4] * JMM[4] + (1 + JMM[2]) * JMM[5]) -
00521     JMM[1] * (-JMM[4] * JMM[3]) + JMM[1] * (1 + JMM[5]));
00522 dudata[pp + 0] /= detC;
00523 dudata[pp + 1] /= detC;
00524 dudata[pp + 2] /= detC;
00525 dudata[pp + 3] = h[3];
00526 dudata[pp + 4] = h[4];
00527 dudata[pp + 5] = h[5];
00528 }
00529 return;
00530 }

```

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 567 of file [TimeEvolutionFunctions.cpp](#).

```
00567
00568
00569 #if defined(_MPI)
00570 #if !defined(_OPENMP)
00571     sunrealtype *udata = NV_DATA_P(u),
00572                 *dudata = NV_DATA_P(udot);
00573 #elif defined(_OPENMP)
00574     sunrealtype *udata = N_VGetArrayPointer_MPIPlusX(u),
00575                 *dudata = N_VGetArrayPointer_MPIPlusX(udot);
00576 #endif
00577 #elif defined(_OPENMP)
00578     sunrealtype *udata = NV_DATA_OMP(u),
00579                 *dudata = NV_DATA_OMP(udot);
00580 #else
00581     sunrealtype *udata = NV_DATA_S(u),
00582                 *dudata = NV_DATA_S(udot);
00583 #endif
00584
00585     sunrealtype *dxData = data->buffData[1 - 1];
00586     sunrealtype *dyData = data->buffData[2 - 1];
00587     sunrealtype *dzData = data->buffData[3 - 1];
00588
00589     data->exchangeGhostCells(1);
00590     data->rotateIntoEigen(1);
00591     data->derive(1);
00592     data->derotate(1,dxData);
00593     data->exchangeGhostCells(2);
00594     data->rotateIntoEigen(2);
00595     data->derive(2);
00596     data->derotate(2,dyData);
00597     data->exchangeGhostCells(3);
00598     data->rotateIntoEigen(3);
00599     data->derive(3);
00600     data->derotate(3,dzData);
00601
00602     static sunrealtype f, g;
00603     static sunrealtype lff, lfg, lg, lgg;
00604     static std::array<sunrealtype, 21> JMM;
00605     static std::array<sunrealtype, 6> Quad;
00606     static std::array<sunrealtype, 6> h;
00607     static sunrealtype detC = nan("0x12345");
00608
00609     const sunindextype totalNP = data->discreteSize();
```

```

00610 #pragma omp parallel for default(none) \
00611 private(JMM, Quad, h, detC) \
00612 shared(totalNP, c, f, g, lff, lfg, lg, lgg, udata, dudata, \
00613     dxData, dyData, dzData) \
00614 schedule(static)
00615 for (sunindextype pp = 0; pp < totalNP * 6; pp += 6) {
00616     f = 0.5 * ((Quad[0] = udata[pp] * udata[pp]) +
00617                 (Quad[1] = udata[pp + 1] * udata[pp + 1]) +
00618                 (Quad[2] = udata[pp + 2] * udata[pp + 2]) -
00619                 (Quad[3] = udata[pp + 3] * udata[pp + 3]) -
00620                 (Quad[4] = udata[pp + 4] * udata[pp + 4]) -
00621                 (Quad[5] = udata[pp + 5] * udata[pp + 5]));
00622     g = udata[pp] * udata[pp + 3] + udata[pp + 1] * udata[pp + 4] +
00623         udata[pp + 2] * udata[pp + 5];
00624     switch (*c) {
00625     case 0:
00626         lf = 0;
00627         lff = 0;
00628         lfg = 0;
00629         lg = 0;
00630         lgg = 0;
00631         break;
00632     case 1:
00633         lf = 0.000206527095658582755255648 * f;
00634         lff = 0.000206527095658582755255648;
00635         lfg = 0;
00636         lg = 0.0003614224174025198216973841 * g;
00637         lgg = 0.0003614224174025198216973841;
00638         break;
00639     case 2:
00640         lf = 0.000354046449700427580438254 * f * f +
00641             0.000191775160254398272737387 * g * g;
00642         lff = 0.0007080928994008551608765075 * f;
00643         lfg = 0.0003835503205087965454747749 * g;
00644         lg = 0.0003835503205087965454747749 * f * g;
00645         lgg = 0.0003835503205087965454747749 * f;
00646         break;
00647     case 3:
00648         lf = (0.000206527095658582755255648 + 0.000354046449700427580438254 * f) *
00649             f +
00650             0.000191775160254398272737387 * g * g;
00651         lff = 0.000206527095658582755255648 + 0.000708092899400855160876508 * f;
00652         lfg = 0.0003835503205087965454747749 * g;
00653         lg = (0.000361422417402519821697384 + 0.000383550320508796545474775 * f) *
00654             g;
00655         lgg = 0.000361422417402519821697384 + 0.000383550320508796545474775 * f;
00656         break;
00657     default:
00658         errorKill(
00659             "You need to specify a correct order in the weak-field expansion.");
00660     }
00661
00662 JMM[0] = lf + lff * Quad[0] +
00663     udata[3 + pp] * (2 * lfg * udata[pp] + lgg * udata[3 + pp]);
00664 JMM[1] =
00665     lff * udata[pp] * udata[1 + pp] + lfg * udata[1 + pp] * udata[3 + pp] +
00666     lfg * udata[pp] * udata[4 + pp] + lgg * udata[3 + pp] * udata[4 + pp];
00667 JMM[2] = lf + lff * Quad[1] +
00668     udata[4 + pp] * (2 * lfg * udata[1 + pp] + lgg * udata[4 + pp]);
00669 JMM[3] =
00670     lff * udata[pp] * udata[2 + pp] + lfg * udata[2 + pp] * udata[3 + pp] +
00671     lfg * udata[pp] * udata[5 + pp] + lgg * udata[3 + pp] * udata[5 + pp];
00672 JMM[4] = lff * udata[1 + pp] * udata[2 + pp] +
00673     lfg * udata[2 + pp] * udata[4 + pp] +
00674     lfg * udata[1 + pp] * udata[5 + pp] +
00675     lgg * udata[4 + pp] * udata[5 + pp];
00676 JMM[5] = lf + lff * Quad[2] +
00677     udata[5 + pp] * (2 * lfg * udata[2 + pp] + lgg * udata[5 + pp]);
00678 JMM[6] = lg + lfg * (Quad[0] - Quad[3 + 0]) +
00679     (-lff + lgg) * udata[pp] * udata[3 + pp];
00680 JMM[7] = -(udata[3 + pp] * (lff * udata[1 + pp] + lfg * udata[4 + pp])) +
00681     udata[pp] * (lfg * udata[1 + pp] + lgg * udata[4 + pp]);
00682 JMM[8] = -(udata[3 + pp] * (lff * udata[2 + pp] + lfg * udata[5 + pp])) +
00683     udata[pp] * (lfg * udata[2 + pp] + lgg * udata[5 + pp]);
00684 JMM[9] = -lf + lgg * Quad[0] +
00685     udata[3 + pp] * (-2 * lfg * udata[pp] + lff * udata[3 + pp]);
00686 JMM[10] = udata[1 + pp] * (lfg * udata[pp] + lgg * udata[3 + pp]) -
00687     (lff * udata[pp] + lfg * udata[3 + pp]) * udata[4 + pp];
00688 JMM[11] = lg + lfg * (Quad[1] - Quad[4 + 0]) +
00689     (-lff + lgg) * udata[1 + pp] * udata[4 + pp];
00690 JMM[12] = -(udata[4 + pp] * (lff * udata[2 + pp] + lfg * udata[5 + pp])) +
00691     udata[1 + pp] * (lfg * udata[2 + pp] + lgg * udata[5 + pp]);
00692 JMM[13] = lgg * udata[pp] * udata[1 + pp] +
00693     lff * udata[3 + pp] * udata[4 + pp] -
00694     lfg * (udata[1 + pp] * udata[3 + pp] + udata[pp] * udata[4 + pp]);
00695 JMM[14] = -lf + lgg * Quad[1] +
00696     udata[4 + pp] * (-2 * lfg * udata[1 + pp] + lff * udata[4 + pp]);

```

```

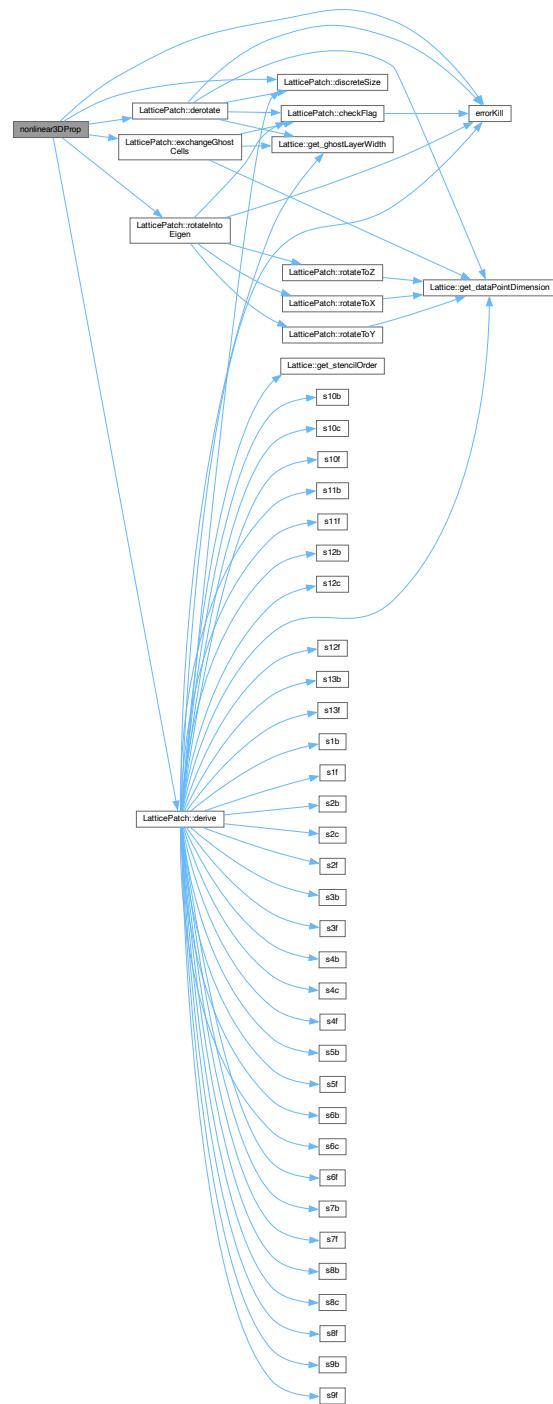
00697     JMM[15] = udata[2 + pp] * (lfg * udata[pp] + lgg * udata[3 + pp]) -
00698             (lff * udata[pp] + lfg * udata[3 + pp]) * udata[5 + pp];
00699     JMM[16] = udata[2 + pp] * (lfg * udata[1 + pp] + lgg * udata[4 + pp]) -
00700             (lff * udata[1 + pp] + lfg * udata[4 + pp]) * udata[5 + pp];
00701     JMM[17] = lg + lfg * (Quad[2] - Quad[5 + 0]) +
00702             (-lff + lgg) * udata[2 + pp] * udata[5 + pp];
00703     JMM[18] = lgg * udata[pp] * udata[2 + pp] +
00704             lff * udata[3 + pp] * udata[5 + pp] -
00705             lfg * (udata[2 + pp] * udata[3 + pp] + udata[pp] * udata[5 + pp]);
00706     JMM[19] =
00707         lgg * udata[1 + pp] * udata[2 + pp] +
00708         lff * udata[4 + pp] * udata[5 + pp] -
00709         lfg * (udata[2 + pp] * udata[4 + pp] + udata[1 + pp] * udata[5 + pp]);
00710     JMM[20] = -lf + lgg * Quad[2] +
00711             udata[5 + pp] * (-2 * lfg * udata[2 + pp] + lff * udata[5 + pp]);
00712
00713     h[0] = 0;
00714     h[1] = dxData[pp] * JMM[15] + dxData[1 + pp] * JMM[16] +
00715             dxData[2 + pp] * JMM[17] + dxData[3 + pp] * JMM[18] +
00716             dxData[4 + pp] * JMM[19] + dxData[5 + pp] * (-1 + JMM[20]);
00717     h[2] = -(dxData[pp] * JMM[10]) - dxData[1 + pp] * JMM[11] -
00718             dxData[2 + pp] * JMM[12] - dxData[3 + pp] * JMM[13] +
00719             dxData[4 + pp] * (1 - JMM[14]) - dxData[5 + pp] * JMM[19];
00720     h[3] = 0;
00721     h[4] = dxData[2 + pp];
00722     h[5] = -dxData[1 + pp];
00723     h[0] += -(dyData[pp] * JMM[15]) - dyData[1 + pp] * JMM[16] -
00724             dyData[2 + pp] * JMM[17] - dyData[3 + pp] * JMM[18] -
00725             dyData[4 + pp] * JMM[19] + dyData[5 + pp] * (1 - JMM[20]);
00726     h[1] += 0;
00727     h[2] += dyData[pp] * JMM[6] + dyData[1 + pp] * JMM[7] +
00728             dyData[2 + pp] * JMM[8] + dyData[3 + pp] * (-1 + JMM[9]) +
00729             dyData[4 + pp] * JMM[13] + dyData[5 + pp] * JMM[18];
00730     h[3] += -dyData[2 + pp];
00731     h[4] += 0;
00732     h[5] += dyData[pp];
00733     h[0] += dzData[pp] * JMM[10] + dzData[1 + pp] * JMM[11] +
00734             dzData[2 + pp] * JMM[12] + dzData[3 + pp] * JMM[13] +
00735             dzData[4 + pp] * (-1 + JMM[14]) + dzData[5 + pp] * JMM[19];
00736     h[1] += -(dzData[pp] * JMM[6]) - dzData[1 + pp] * JMM[7] -
00737             dzData[2 + pp] * JMM[8] + dzData[3 + pp] * (1 - JMM[9]) -
00738             dzData[4 + pp] * JMM[13] - dzData[5 + pp] * JMM[18];
00739     h[2] += 0;
00740     h[3] += dzData[1 + pp];
00741     h[4] += -dzData[pp];
00742     h[5] += 0;
00743     h[0] -= h[3] * JMM[6] + h[4] * JMM[10] + h[5] * JMM[15];
00744     h[1] -= h[3] * JMM[7] + h[4] * JMM[11] + h[5] * JMM[16];
00745     h[2] -= h[3] * JMM[8] + h[4] * JMM[12] + h[5] * JMM[17];
00746     dudata[pp + 0] =
00747         h[2] * (-JMM[3] * (1 + JMM[2])) + JMM[1] * JMM[4] +
00748         h[1] * (JMM[3] * JMM[4] - JMM[1] * (1 + JMM[5])) +
00749         h[0] * (1 - JMM[4] * JMM[4] + JMM[5] + JMM[2] * (1 + JMM[5]));
00750     dudata[pp + 1] =
00751         h[2] * (JMM[3] * JMM[1] - (1 + JMM[0]) * JMM[4]) +
00752         h[1] * (1 - JMM[3] * JMM[3] + JMM[5] + JMM[0] * (1 + JMM[5])) +
00753         h[0] * (JMM[4] * JMM[3] - JMM[1] * (1 + JMM[5]));
00754     dudata[pp + 2] =
00755         h[2] * (1 - JMM[1] * JMM[1] + JMM[2] + JMM[0] * (1 + JMM[2])) +
00756         h[1] * (JMM[1] * JMM[3] - (1 + JMM[0]) * JMM[4]) +
00757         h[0] * ((-1 + JMM[2]) * JMM[3] + JMM[1] * JMM[4]);
00758     detC =
00759         -((1 + JMM[2]) * (-1 + JMM[3] * JMM[3])) +
00760         (JMM[3] * JMM[1] - JMM[4]) * JMM[4] + JMM[5] + JMM[2] * JMM[5] +
00761         JMM[0] * (1 + JMM[2] - JMM[4] * JMM[4] + (1 + JMM[2]) * JMM[5]) -
00762         JMM[1] * ((-JMM[4] * JMM[3]) + JMM[1] * (1 + JMM[5]));
00763     dudata[pp + 0] /= detC;
00764     dudata[pp + 1] /= detC;
00765     dudata[pp + 2] /= detC;
00766     dudata[pp + 3] = h[3];
00767     dudata[pp + 4] = h[4];
00768     dudata[pp + 5] = h[5];
00769 }
00770 return;
00771 }

```

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 /// CCode 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. derivatievees with NVector pointers
00018 #if defined(_MPI)
00019 #if !defined(_OPENMP)
00020     unrealtype *udata = NV_DATA_P(u),
00021             *dudata = NV_DATA_P(udot);
00022 #elif defined(_OPENMP)
00023     unrealtype *udata = N_VGetArrayPointer_MPIPlusX(u),
00024             *dudata = N_VGetArrayPointer_MPIPlusX(udot);
00025 #endif
00026 #elif defined(_OPENMP)
00027     unrealtype *udata = NV_DATA_OMP(u),
00028             *dudata = NV_DATA_OMP(udot);
00029 #else
00030     unrealtype *udata = NV_DATA_S(u),
00031             *dudata = NV_DATA_S(udot);
00032 #endif
00033
00034     // Store original data location of the patch
00035     unrealtype *originaluData = data->uData,
00036             *originalduData = data->duData;
00037
00038     // Point patch data to arguments of f
00039     data->uData = udata;
00040     data->duData = dudata;
00041
00042     // Time-evolve these arguments (the field data) with specific propagator below
00043     TimeEvolver(data, u, udot, c);
00044
00045     // Refer patch data back to original location
00046     data->uData = originaluData;
00047     data->duData = originalduData;
00048
00049     return (0);
00050 }
00051
00052 /// only under-the-hood-callable Maxwell propagation in 1D;
00053 /// unused parameters 2-4 for compliance with CVRhsFn - field data is here
00054 /// accessed implicitly via user data (lattice patch);
00055 /// same effect as the respective nonlinear function without nonlinear terms
00056 void linear1DProp(LatticePatch *data, N_Vector u, N_Vector udot, int *c) {
00057
00058     // pointers to temporal and spatial derivative data
00059     unrealtype *duData = data->duData;
00060     unrealtype *dxData = data->buffData[1 - 1];
00061

```

```

00062 // sequence along any dimension according to the scheme:
00063 data->exchangeGhostCells(1); // -> exchange halos
00064 data->rotateIntoEigen(
00065     1); // -> rotate all data to prepare derivative operation
00066 data->derive(1); // -> perform derivative approximation operation on it
00067 data->derotate(
00068     1, dxData); // -> derotate derived data for ensuing time-evolution
00069
00070 const sunindextype totalNP = data->discreteSize();
00071 sunindextype pp = 0;
00072 for (sunindextype i = 0; i < totalNP; i++) {
00073     pp = i * 6;
00074     /*
00075         simple vacuum Maxwell equations for the temporal derivatives using the
00076         spatial derivative only in x-direction without polarization or
00077         magnetization terms
00078     */
00079     duData[pp + 0] = 0;
00080     duData[pp + 1] = -dxData[pp + 5];
00081     duData[pp + 2] = dxData[pp + 4];
00082     duData[pp + 3] = 0;
00083     duData[pp + 4] = dxData[pp + 2];
00084     duData[pp + 5] = -dxData[pp + 1];
00085 }
00086 }
00087
00088 /// nonlinear 1D HE propagation
00089 void nonlinear1DProp(LatticePatch *data, N_Vector u, N_Vector udot, int *c) {
00090
00091     // NVector pointers to provided field values and their temp. derivatives
00092 #if defined(_MPI)
00093 #if !defined(_OPENMP)
00094     sunrealtype *udata = NV_DATA_P(u),
00095             *dudata = NV_DATA_P(udot);
00096 #elif defined(_OPENMP)
00097     sunrealtype *udata = N_VGetArrayPointer_MPIPlusX(u),
00098             *dudata = N_VGetArrayPointer_MPIPlusX(udot);
00099 #endif
00100 #elif defined(_OPENMP)
00101     sunrealtype *udata = NV_DATA_OMP(u),
00102             *dudata = NV_DATA_OMP(udot);
00103 #else
00104     sunrealtype *udata = NV_DATA_S(u),
00105             *dudata = NV_DATA_S(udot);
00106 #endif
00107
00108     // pointer to spatial derivatives via patch data
00109     sunrealtype *dxData = data->buffData[1 - 1];
00110
00111     // same sequence as in the linear case
00112     data->exchangeGhostCells(1);
00113     data->rotateIntoEigen(1);
00114     data->derive(1);
00115     data->derotate(1, dxData);
00116
00117     /*
00118         F and G are nonzero in the nonlinear case,
00119         polarization and magnetization derivatives
00120         w.r.t. E- and B-field go into the e.o.m.
00121     */
00122     static sunrealtype f, g; // em field invariants F, G
00123     // derivatives of HE Lagrangian w.r.t. field invariants
00124     static sunrealtype lf, lff, lfg, lg, lgg;
00125     // matrix to hold derivatives of polarization and magnetization
00126     static std::array<sunrealtype, 21> JMM;
00127     // array to hold E^2 and B^2 components
00128     static std::array<sunrealtype, 6> Quad;
00129     // array to hold intermediate temp. derivatives of E and B
00130     static std::array<sunrealtype, 6> h;
00131     // determinant needed for explicit matrix inversion
00132     static sunrealtype detC = nan("0x12345");
00133
00134     // number of points in the patch
00135     const sunindextype totalNP = data->discreteSize();
00136     #pragma omp parallel for default(none) \
00137     private(JMM, Quad, h, detC) \
00138     shared(totalNP, c, f, g, lf, lff, lfg, lg, lgg, udata, dudata, dxData) \
00139     schedule(static)
00140     for (sunindextype pp = 0; pp < totalNP * 6;
00141         pp += 6) { // loop over all 6dim points in the patch
00142         // em field Lorentz invariants F and G
00143         f = 0.5 * ((Quad[0] = udata[pp] * udata[pp]) +
00144                     (Quad[1] = udata[pp + 1] * udata[pp + 1])) +
00145                     (Quad[2] = udata[pp + 2] * udata[pp + 2]) -
00146                     (Quad[3] = udata[pp + 3] * udata[pp + 3]) -
00147                     (Quad[4] = udata[pp + 4] * udata[pp + 4]) -
00148                     (Quad[5] = udata[pp + 5] * udata[pp + 5]));

```

```

00149     g = udata[pp] * udata[pp + 3] + udata[pp + 1] * udata[pp + 4] +
00150         udata[pp + 2] * udata[pp + 5];
00151 // process/expansion order and corresponding derivative values of L
00152 // w.r.t. F, G
00153 switch (*c) {
00154 case 0: // linear Maxwell vacuum
00155     lf = 0;
00156     lff = 0;
00157     lfg = 0;
00158     lg = 0;
00159     lgg = 0;
00160     break;
00161 case 1: // only 4-photon processes
00162     lf = 0.000206527095658582755255648 * f;
00163     lff = 0.000206527095658582755255648;
00164     lfg = 0;
00165     lg = 0.0003614224174025198216973841 * g;
00166     lgg = 0.0003614224174025198216973841;
00167     break;
00168 case 2: // only 6-photon processes
00169     lf = 0.000354046449700427580438254 * f * f +
00170         0.000191775160254398272737387 * g * g;
00171     lff = 0.0007080928994008551608765075 * f;
00172     lfg = 0.0003835503205087965454747749 * g;
00173     lg = 0.0003835503205087965454747749 * f * g;
00174     lgg = 0.0003835503205087965454747749 * f;
00175     break;
00176 case 3: // 4- and 6-photon processes
00177     lf = (0.000206527095658582755255648 + 0.000354046449700427580438254 * f) *
00178         f +
00179         0.000191775160254398272737387 * g * g;
00180     lff = 0.000206527095658582755255648 + 0.0007080928994008551608765075 * f;
00181     lfg = 0.0003835503205087965454747749 * g;
00182     lg = (0.0003614224174025198216973841 +
00183         0.0003835503205087965454747749 * f) *
00184         g;
00185     lgg = 0.0003614224174025198216973841 + 0.0003835503205087965454747749 * f;
00186     break;
00187 default:
00188     errorKill(
00189         "You need to specify a correct order in the weak-field expansion.");
00190 }
00191
00192 // derivatives of polarization and magnetization w.r.t. E and B
00193 // Jpx(Ex)
00194 JMM[0] = lf + lff * Quad[0] +
00195     udata[3 + pp] * (2 * lfg * udata[pp] + lgg * udata[3 + pp]);
00196 // Jpx(Ey)
00197 JMM[1] =
00198     lff * udata[pp] * udata[1 + pp] + lfg * udata[1 + pp] * udata[3 + pp] +
00199     lfg * udata[pp] * udata[4 + pp] + lgg * udata[3 + pp] * udata[4 + pp];
00200 // Jpy(Ey)
00201 JMM[2] = lf + lff * Quad[1] +
00202     udata[4 + pp] * (2 * lfg * udata[1 + pp] + lgg * udata[4 + pp]);
00203 // Jpx(Ez) = Jpz(Ex)
00204 JMM[3] =
00205     lff * udata[pp] * udata[2 + pp] + lfg * udata[2 + pp] * udata[3 + pp] +
00206     lfg * udata[pp] * udata[5 + pp] + lgg * udata[3 + pp] * udata[5 + pp];
00207 // Jpy(Ez) = Jpz(Ey)
00208 JMM[4] = lff * udata[1 + pp] * udata[2 + pp] +
00209     lfg * udata[2 + pp] * udata[4 + pp] +
00210     lfg * udata[1 + pp] * udata[5 + pp] +
00211     lgg * udata[4 + pp] * udata[5 + pp];
00212 // Jpz(Ez)
00213 JMM[5] = lf + lff * Quad[2] +
00214     udata[5 + pp] * (2 * lfg * udata[2 + pp] + lgg * udata[5 + pp]);
00215 // Jpx(Bx) = Jmx(Ex)
00216 JMM[6] = lg + lfg * (Quad[0] - Quad[3 + 0]) +
00217     (-lff + lgg) * udata[pp] * udata[3 + pp];
00218 // Jpy(Bx) = Jmx(Ey)
00219 JMM[7] = -(udata[3 + pp] * (lff * udata[1 + pp] + lfg * udata[4 + pp])) +
00220     udata[pp] * (lfg * udata[1 + pp] + lgg * udata[4 + pp]);
00221 // Jpz(Bx) = Jmx(Ez)
00222 JMM[8] = -(udata[3 + pp] * (lff * udata[2 + pp] + lfg * udata[5 + pp])) +
00223     udata[pp] * (lfg * udata[2 + pp] + lgg * udata[5 + pp]);
00224 // Jmx(Bx)
00225 JMM[9] = -lf + lgg * Quad[0] +
00226     udata[3 + pp] * (-2 * lfg * udata[pp] + lff * udata[3 + pp]);
00227 // Jpx(By) = Jmy(Ex)
00228 JMM[10] = udata[1 + pp] * (lfg * udata[pp] + lgg * udata[3 + pp]) -
00229     (lf * udata[pp] + lfg * udata[3 + pp]) * udata[4 + pp];
00230 // Jpy(By) = Jmy(Ey)
00231 JMM[11] = lg + lfg * (Quad[1] - Quad[4 + 0]) +
00232     (-lff + lgg) * udata[1 + pp] * udata[4 + pp];
00233 // Jpz(By) = Jmy(Ez)
00234 JMM[12] = -(udata[4 + pp] * (lff * udata[2 + pp] + lfg * udata[5 + pp])) +
00235     udata[1 + pp] * (lfg * udata[2 + pp] + lgg * udata[5 + pp]);

```

```

00236 // Jmx(By) = Jmy(Bx)
00237 JMM[13] = lgg * udata[pp] * udata[1 + pp] +
00238     lff * udata[3 + pp] * udata[4 + pp] -
00239     lfg * (udata[1 + pp] * udata[3 + pp] + udata[pp] * udata[4 + pp]);
00240 // Jmy(By)
00241 JMM[14] = -lf + lgg * Quad[1] +
00242     udata[4 + pp] * (-2 * lfg * udata[1 + pp] + lff * udata[4 + pp]);
00243 // Jmz(Ex) = Jpx(Bz)
00244 JMM[15] = udata[2 + pp] * (lfg * udata[pp] + lgg * udata[3 + pp]) -
00245     (lff * udata[pp] + lfg * udata[3 + pp]) * udata[5 + pp];
00246 // Jmz(Ey) = Jpy(Bz)
00247 JMM[16] = udata[2 + pp] * (lfg * udata[1 + pp] + lgg * udata[4 + pp]) -
00248     (lff * udata[1 + pp] + lfg * udata[4 + pp]) * udata[5 + pp];
00249 // Jpz(Bz) = Jmz(Ez)
00250 JMM[17] = lg + lfg * (Quad[2] - Quad[5 + 0]) +
00251     (-lff + lgg) * udata[2 + pp] * udata[5 + pp];
00252 // Jmz(Bx) = Jmx(Bz)
00253 JMM[18] = lgg * udata[pp] * udata[2 + pp] +
00254     lff * udata[3 + pp] * udata[5 + pp] -
00255     lfg * (udata[2 + pp] * udata[3 + pp] + udata[pp] * udata[5 + pp]);
00256 // Jmy(Bz) = Jmz(By)
00257 JMM[19] =
00258     lgg * udata[1 + pp] * udata[2 + pp] +
00259     lff * udata[4 + pp] * udata[5 + pp] -
00260     lfg * (udata[2 + pp] * udata[4 + pp] + udata[1 + pp] * udata[5 + pp]);
00261 // Jmz(Bz)
00262 JMM[20] = -lf + lgg * Quad[2] +
00263     udata[5 + pp] * (-2 * lfg * udata[2 + pp] + lff * udata[5 + pp]);
00264
00265 // apply Z
00266 // top block: -QJm(E)*E, Q-QJm(B)*B
00267 h[0] = 0;
00268 h[1] = dxData[pp] * JMM[15] + dxData[1 + pp] * JMM[16] +
00269     dxData[2 + pp] * JMM[17] + dxData[3 + pp] * JMM[18] +
00270     dxData[4 + pp] * JMM[19] + dxData[5 + pp] * (-1 + JMM[20]);
00271 h[2] = -(dxData[pp] * JMM[10]) - dxData[1 + pp] * JMM[11] -
00272     dxData[2 + pp] * JMM[12] - dxData[3 + pp] * JMM[13] +
00273     dxData[4 + pp] * (1 - JMM[14]) - dxData[5 + pp] * JMM[19];
00274 // bottom blocks: -Q*B
00275 h[3] = 0;
00276 h[4] = dxData[2 + pp];
00277 h[5] = -dxData[1 + pp];
00278 // (1+A)^-1 applies only to E components
00279 // -Jp(B)*B
00280 h[0] -= h[3] * JMM[6] + h[4] * JMM[10] + h[5] * JMM[15];
00281 h[1] -= h[3] * JMM[7] + h[4] * JMM[11] + h[5] * JMM[16];
00282 h[2] -= h[3] * JMM[8] + h[4] * JMM[12] + h[5] * JMM[17];
00283 // apply C^-1 explicitly, with C=1+Jp(E)
00284 dudata[pp + 0] =
00285     h[2] * (-JMM[3] * (1 + JMM[2])) + JMM[1] * JMM[4]) +
00286     h[1] * (JMM[3] * JMM[4] - JMM[1] * (1 + JMM[5])) +
00287     h[0] * (1 - JMM[4] * JMM[4] + JMM[5] + JMM[2] * (1 + JMM[5]));
00288 dudata[pp + 1] =
00289     h[2] * (JMM[3] * JMM[1] - (1 + JMM[0]) * JMM[4]) +
00290     h[1] * (1 - JMM[3] * JMM[3] + JMM[5] + JMM[0] * (1 + JMM[5])) +
00291     h[0] * (JMM[4] * JMM[3] - JMM[1] * (1 + JMM[5]));
00292 dudata[pp + 2] =
00293     h[2] * (1 - JMM[1] * JMM[1] + JMM[2] + JMM[0] * (1 + JMM[2])) +
00294     h[1] * (JMM[1] * JMM[3] - (1 + JMM[0]) * JMM[4]) +
00295     h[0] * ((-1 + JMM[2]) * JMM[3]) + JMM[1] * JMM[4]);
00296 detC = // determinant of C
00297     -((1 + JMM[2]) * (-1 + JMM[3] * JMM[3])) +
00298     (JMM[3] * JMM[1] - JMM[4]) * JMM[4] + JMM[5] + JMM[2] * JMM[5] +
00299     JMM[0] * (1 + JMM[2] - JMM[4] * JMM[4] + (1 + JMM[2]) * JMM[5]) -
00300     JMM[1] * ((-JMM[4] * JMM[3]) + JMM[1] * (1 + JMM[5]));
00301 dudata[pp + 0] /= detC;
00302 dudata[pp + 1] /= detC;
00303 dudata[pp + 2] /= detC;
00304 dudata[pp + 3] = h[3];
00305 dudata[pp + 4] = h[4];
00306 dudata[pp + 5] = h[5];
00307 }
00308 return;
00309 }
00310
00311 // only under-the-hood-callable Maxwell propagation in 2D
00312 void linear2DProp(LatticePatch *data, N_Vector u, N_Vector udot, int *c) {
00313
00314     sunrealtype *duData = data->duData;
00315     sunrealtype *dxData = data->buffData[1 - 1];
00316     sunrealtype *dyData = data->buffData[2 - 1];
00317
00318     data->exchangeGhostCells(1);
00319     data->rotateIntoEigen(1);
00320     data->derive(1);
00321     data->derotate(1, dxData);
00322     data->exchangeGhostCells(2);

```

```

00323     data->rotateIntoEigen(2);
00324     data->derive(2);
00325     data->derotate(2, dyData);
00326
00327     const sunindextype totalNP = data->discreteSize();
00328     sunindextype pp = 0;
00329     for (sunindextype i = 0; i < totalNP; i++) {
00330         pp = i * 6;
00331         duData[pp + 0] = dyData[pp + 5];
00332         duData[pp + 1] = -dxData[pp + 5];
00333         duData[pp + 2] = -dyData[pp + 3] + dxData[pp + 4];
00334         duData[pp + 3] = -dyData[pp + 2];
00335         duData[pp + 4] = dxData[pp + 2];
00336         duData[pp + 5] = dyData[pp + 0] - dxData[pp + 1];
00337     }
00338 }
00339
00340 ///////////////////////////////////////////////////////////////////
00341 void nonlinear2DProp(LatticePatch *data, N_Vec u, N_Vec udot, int *c) {
00342
00343 #if defined(_MPI)
00344 #if !defined(_OPENMP)
00345     sunrealtype *udata = NV_DATA_P(u),
00346                 *dudata = NV_DATA_P(udot);
00347 #elif defined(_OPENMP)
00348     sunrealtype *udata = N_VGetArrayPointer_MPIPlusX(u),
00349                 *dudata = N_VGetArrayPointer_MPIPlusX(udot);
00350 #endif
00351 #elif defined(_OPENMP)
00352     sunrealtype *udata = NV_DATA_OMP(u),
00353                 *dudata = NV_DATA_OMP(udot);
00354 #else
00355     sunrealtype *udata = NV_DATA_S(u),
00356                 *dudata = NV_DATA_S(udot);
00357 #endif
00358
00359     sunrealtype *dxData = data->buffData[1 - 1];
00360     sunrealtype *dyData = data->buffData[2 - 1];
00361
00362     data->exchangeGhostCells(1);
00363     data->rotateIntoEigen(1);
00364     data->derive(1);
00365     data->derotate(1, dxData);
00366     data->exchangeGhostCells(2);
00367     data->rotateIntoEigen(2);
00368     data->derive(2);
00369     data->derotate(2, dyData);
00370
00371     static sunrealtype f, g;
00372     static sunrealtype lf, lff, lfg, lg, lgg;
00373     static std::array<sunrealtype, 21> JMM;
00374     static std::array<sunrealtype, 6> Quad;
00375     static std::array<sunrealtype, 6> h;
00376     static sunrealtype detC;
00377
00378     const sunindextype totalNP = data->discreteSize();
00379 #pragma omp parallel for default(none) \
00380 private(JMM, Quad, h, detC) \
00381 shared(totalNP, c, f, g, lf, lff, lfg, lg, lgg, udata, dudata, \
00382         dxData, dyData) \
00383 schedule(static)
00384     for (sunindextype pp = 0; pp < totalNP * 6; pp += 6) {
00385         f = 0.5 * ((Quad[0] = udata[pp] * udata[pp]) +
00386                     (Quad[1] = udata[pp + 1] * udata[pp + 1]) +
00387                     (Quad[2] = udata[pp + 2] * udata[pp + 2]) -
00388                     (Quad[3] = udata[pp + 3] * udata[pp + 3]) -
00389                     (Quad[4] = udata[pp + 4] * udata[pp + 4]) -
00390                     (Quad[5] = udata[pp + 5] * udata[pp + 5]));
00391         g = udata[pp] * udata[pp + 3] + udata[pp + 1] * udata[pp + 4] +
00392             udata[pp + 2] * udata[pp + 5];
00393         switch (*c) {
00394             case 0:
00395                 lf = 0;
00396                 lff = 0;
00397                 lfg = 0;
00398                 lg = 0;
00399                 lgg = 0;
00400                 break;
00401             case 1:
00402                 lf = 0.000206527095658582755255648 * f;
00403                 lff = 0.000206527095658582755255648;
00404                 lfg = 0;
00405                 lg = 0.0003614224174025198216973841 * g;
00406                 lgg = 0.0003614224174025198216973841;
00407                 break;
00408             case 2:
00409                 lf = 0.000354046449700427580438254 * f * f +

```

```

00410      0.000191775160254398272737387 * g * g;
00411      lff = 0.0007080928994008551608765075 * f;
00412      lfg = 0.0003835503205087965454747749 * g;
00413      lg = 0.0003835503205087965454747749 * f * g;
00414      lgg = 0.0003835503205087965454747749 * f;
00415      break;
00416  case 3:
00417      lf = (0.000206527095658582755255648 + 0.000354046449700427580438254 * f) *
00418      f +
00419      0.000191775160254398272737387 * g * g;
00420      lff = 0.000206527095658582755255648 + 0.000708092899400855160876508 * f;
00421      lfg = 0.0003835503205087965454747749 * g;
00422      lg = (0.000361422417402519821697384 + 0.000383550320508796545474775 * f) *
00423      g;
00424      lgg = 0.000361422417402519821697384 + 0.000383550320508796545474775 * f;
00425      break;
00426  default:
00427      errorKill(
00428          "You need to specify a correct order in the weak-field expansion.");
00429  }
00430
00431  JMM[0] = lf + lff * Quad[0] +
00432      udata[3 + pp] * (2 * lfg * udata[pp] + lgg * udata[3 + pp]);
00433  JMM[1] =
00434      lff * udata[pp] * udata[1 + pp] + lfg * udata[1 + pp] * udata[3 + pp] +
00435      lfg * udata[pp] * udata[4 + pp] + lgg * udata[3 + pp] * udata[4 + pp];
00436  JMM[2] = lf + lff * Quad[1] +
00437      udata[4 + pp] * (2 * lfg * udata[1 + pp] + lgg * udata[4 + pp]);
00438  JMM[3] =
00439      lff * udata[pp] * udata[2 + pp] + lfg * udata[2 + pp] * udata[3 + pp] +
00440      lfg * udata[pp] * udata[5 + pp] + lgg * udata[3 + pp] * udata[5 + pp];
00441  JMM[4] = lff * udata[1 + pp] * udata[2 + pp] +
00442      lfg * udata[2 + pp] * udata[4 + pp] +
00443      lfg * udata[1 + pp] * udata[5 + pp] +
00444      lgg * udata[4 + pp] * udata[5 + pp];
00445  JMM[5] = lf + lff * Quad[2] +
00446      udata[5 + pp] * (2 * lfg * udata[2 + pp] + lgg * udata[5 + pp]);
00447  JMM[6] = lg + lfg * (Quad[0] - Quad[3 + 0]) +
00448      (-lff + lgg) * udata[pp] * udata[3 + pp];
00449  JMM[7] = -(udata[3 + pp] * (lff * udata[1 + pp] + lfg * udata[4 + pp])) +
00450      udata[pp] * (lfg * udata[1 + pp] + lgg * udata[4 + pp]);
00451  JMM[8] = -(udata[3 + pp] * (lff * udata[2 + pp] + lfg * udata[5 + pp])) +
00452      udata[pp] * (lfg * udata[2 + pp] + lgg * udata[5 + pp]);
00453  JMM[9] = -lf + lgg * Quad[0] +
00454      udata[3 + pp] * (-2 * lfg * udata[pp] + lff * udata[3 + pp]);
00455  JMM[10] = udata[1 + pp] * (lfg * udata[pp] + lgg * udata[3 + pp]) -
00456      (lff * udata[pp] + lfg * udata[3 + pp]) * udata[4 + pp];
00457  JMM[11] = lg + lfg * (Quad[1] - Quad[4 + 0]) +
00458      (-lff + lgg) * udata[1 + pp] * udata[4 + pp];
00459  JMM[12] = -(udata[4 + pp] * (lff * udata[2 + pp] + lfg * udata[5 + pp])) +
00460      udata[1 + pp] * (lfg * udata[2 + pp] + lgg * udata[5 + pp]);
00461  JMM[13] = lgg * udata[pp] * udata[1 + pp] +
00462      lff * udata[3 + pp] * udata[4 + pp] -
00463      lfg * (udata[1 + pp] * udata[3 + pp] + udata[pp] * udata[4 + pp]);
00464  JMM[14] = -lf + lgg * Quad[1] +
00465      udata[4 + pp] * (-2 * lfg * udata[1 + pp] + lff * udata[4 + pp]);
00466  JMM[15] = udata[2 + pp] * (lfg * udata[pp] + lgg * udata[3 + pp]) -
00467      (lff * udata[pp] + lfg * udata[3 + pp]) * udata[5 + pp];
00468  JMM[16] = udata[2 + pp] * (lfg * udata[1 + pp] + lgg * udata[4 + pp]) -
00469      (lff * udata[1 + pp] + lfg * udata[4 + pp]) * udata[5 + pp];
00470  JMM[17] = lg + lfg * (Quad[2] - Quad[5 + 0]) +
00471      (-lff + lgg) * udata[2 + pp] * udata[5 + pp];
00472  JMM[18] = lgg * udata[pp] * udata[2 + pp] -
00473      lff * udata[3 + pp] * udata[5 + pp] -
00474      lfg * (udata[2 + pp] * udata[3 + pp] + udata[pp] * udata[5 + pp]);
00475  JMM[19] =
00476      lgg * udata[1 + pp] * udata[2 + pp] +
00477      lff * udata[4 + pp] * udata[5 + pp] -
00478      lfg * (udata[2 + pp] * udata[4 + pp] + udata[1 + pp] * udata[5 + pp]);
00479  JMM[20] = -lf + lgg * Quad[2] +
00480      udata[5 + pp] * (-2 * lfg * udata[2 + pp] + lff * udata[5 + pp]);
00481
00482  h[0] = 0;
00483  h[1] = dxData[pp] * JMM[15] + dxData[1 + pp] * JMM[16] +
00484      dxData[2 + pp] * JMM[17] + dxData[3 + pp] * JMM[18] +
00485      dxData[4 + pp] * JMM[19] + dxData[5 + pp] * (-1 + JMM[20]);
00486  h[2] = -(dxData[pp] * JMM[10]) - dxData[1 + pp] * JMM[11] -
00487      dxData[2 + pp] * JMM[12] - dxData[3 + pp] * JMM[13] +
00488      dxData[4 + pp] * (1 - JMM[14]) - dxData[5 + pp] * JMM[19];
00489  h[3] = 0;
00490  h[4] = dxData[2 + pp];
00491  h[5] = -dxData[1 + pp];
00492  h[0] += -(dyData[pp] * JMM[15]) - dyData[1 + pp] * JMM[16] -
00493      dyData[2 + pp] * JMM[17] - dyData[3 + pp] * JMM[18] -
00494      dyData[4 + pp] * JMM[19] + dyData[5 + pp] * (1 - JMM[20]);
00495  h[1] += 0;
00496  h[2] += dyData[pp] * JMM[6] + dyData[1 + pp] * JMM[7] +

```

```

00497     dyData[2 + pp] * JMM[8] + dyData[3 + pp] * (-1 + JMM[9]) +
00498     dyData[4 + pp] * JMM[13] + dyData[5 + pp] * JMM[18];
00499     h[3] += -dyData[2 + pp];
00500     h[4] += 0;
00501     h[5] += dyData[pp];
00502     h[0] -= h[3] * JMM[6] + h[4] * JMM[10] + h[5] * JMM[15];
00503     h[1] -= h[3] * JMM[7] + h[4] * JMM[11] + h[5] * JMM[16];
00504     h[2] -= h[3] * JMM[8] + h[4] * JMM[12] + h[5] * JMM[17];
00505     dudata[pp + 0] =
00506     h[2] * (-JMM[3] * (1 + JMM[2])) + JMM[1] * JMM[4]) +
00507     h[1] * (JMM[3] * JMM[4] - JMM[1] * (1 + JMM[5])) +
00508     h[0] * (1 - JMM[4] * JMM[4] + JMM[5] + JMM[2] * (1 + JMM[5]));
00509     dudata[pp + 1] =
00510     h[2] * (JMM[3] * JMM[1] - (1 + JMM[0]) * JMM[4]) +
00511     h[1] * (1 - JMM[3] * JMM[3] + JMM[5] + JMM[0] * (1 + JMM[5])) +
00512     h[0] * (JMM[4] * JMM[3] - JMM[1] * (1 + JMM[5]));
00513     dudata[pp + 2] =
00514     h[2] * (1 - JMM[1] * JMM[1] + JMM[2] + JMM[0] * (1 + JMM[2])) +
00515     h[1] * (JMM[1] * JMM[3] - (1 + JMM[0]) * JMM[4]) +
00516     h[0] * (((1 + JMM[2]) * JMM[3]) + JMM[1] * JMM[4]);
00517     detC =
00518     -((1 + JMM[2]) * (-1 + JMM[3] * JMM[3])) +
00519     (JMM[3] * JMM[1] - JMM[4]) * JMM[4] + JMM[5] + JMM[2] * JMM[5] +
00520     JMM[0] * (1 + JMM[2] - JMM[4] * JMM[4] + (1 + JMM[2]) * JMM[5]) -
00521     JMM[1] * (-((JMM[4] * JMM[3]) + JMM[1] * (1 + JMM[5])));
00522     dudata[pp + 0] /= detC;
00523     dudata[pp + 1] /= detC;
00524     dudata[pp + 2] /= detC;
00525     dudata[pp + 3] = h[3];
00526     dudata[pp + 4] = h[4];
00527     dudata[pp + 5] = h[5];
00528 }
00529 return;
00530 }
00531
00532 // only under-the-hood-callable Maxwell propagation in 3D
00533 void linear3DProp(LatticePatch *data, N_Vector u, N_Vector udot, int *c) {
00534
00535     sunrealtype *duData = data->duData;
00536     sunrealtype *dxData = data->buffData[1 - 1];
00537     sunrealtype *dyData = data->buffData[2 - 1];
00538     sunrealtype *dzData = data->buffData[3 - 1];
00539
00540     data->exchangeGhostCells(1);
00541     data->rotateIntoEigen(1);
00542     data->derive(1);
00543     data->derotate(1, dxData);
00544     data->exchangeGhostCells(2);
00545     data->rotateIntoEigen(2);
00546     data->derive(2);
00547     data->derotate(2, dyData);
00548     data->exchangeGhostCells(3);
00549     data->rotateIntoEigen(3);
00550     data->derive(3);
00551     data->derotate(3, dzData);
00552
00553     const sunindextype totalNP = data->discreteSize();
00554     sunindextype pp = 0;
00555     for (sunindextype i = 0; i < totalNP; i++) {
00556         pp = i * 6;
00557         duData[pp + 0] = dyData[pp + 5] - dzData[pp + 4];
00558         duData[pp + 1] = dzData[pp + 3] - dxData[pp + 5];
00559         duData[pp + 2] = dxData[pp + 4] - dyData[pp + 3];
00560         duData[pp + 3] = -dyData[pp + 2] + dzData[pp + 1];
00561         duData[pp + 4] = -dzData[pp + 0] + dxData[pp + 2];
00562         duData[pp + 5] = -dxData[pp + 1] + dyData[pp + 0];
00563     }
00564 }
00565
00566 // nonlinear 3D HE propagation
00567 void nonlinear3DProp(LatticePatch *data, N_Vector u, N_Vector udot, int *c) {
00568
00569 #if defined(_MPI)
00570 #if !defined(_OPENMP)
00571     sunrealtype *udata = NV_DATA_P(u),
00572             *dudata = NV_DATA_P(udot);
00573 #elif defined(_OPENMP)
00574     sunrealtype *udata = N_VGetArrayPointer_MPIPlusX(u),
00575             *dudata = N_VGetArrayPointer_MPIPlusX(udot);
00576 #endif
00577 #elif defined(_OPENMP)
00578     sunrealtype *udata = NV_DATA_OMP(u),
00579             *dudata = NV_DATA_OMP(udot);
00580 #else
00581     sunrealtype *udata = NV_DATA_S(u),
00582             *dudata = NV_DATA_S(udot);
00583 #endif

```

```

00584
00585     sunrealtype *dxData = data->buffData[1 - 1];
00586     sunrealtype *dyData = data->buffData[2 - 1];
00587     sunrealtype *dzData = data->buffData[3 - 1];
00588
00589     data->exchangeGhostCells(1);
00590     data->rotateIntoEigen(1);
00591     data->derive(1);
00592     data->derotate(1, dxData);
00593     data->exchangeGhostCells(2);
00594     data->rotateIntoEigen(2);
00595     data->derive(2);
00596     data->derotate(2, dyData);
00597     data->exchangeGhostCells(3);
00598     data->rotateIntoEigen(3);
00599     data->derive(3);
00600     data->derotate(3, dzData);
00601
00602     static sunrealtype f, g;
00603     static sunrealtype lf, lff, lfg, lg, lgg;
00604     static std::array<sunrealtype, 21> JMM;
00605     static std::array<sunrealtype, 6> Quad;
00606     static std::array<sunrealtype, 6> h;
00607     static sunrealtype detC = nan("0x12345");
00608
00609     const sunindextype totalNP = data->discreteSize();
00610     #pragma omp parallel for default(none) \
00611     private(JMM, Quad, h, detC) \
00612     shared(totalNP, c, f, g, lf, lff, lfg, lg, lgg, udata, dudata, \
00613           dxData, dyData, dzData) \
00614     schedule(static)
00615     for (sunindextype pp = 0; pp < totalNP * 6; pp += 6) {
00616         f = 0.5 * ((Quad[0] = udata[pp] * udata[pp]) +
00617                     (Quad[1] = udata[pp + 1] * udata[pp + 1])) +
00618                     (Quad[2] = udata[pp + 2] * udata[pp + 2]) -
00619                     (Quad[3] = udata[pp + 3] * udata[pp + 3]) -
00620                     (Quad[4] = udata[pp + 4] * udata[pp + 4]) -
00621                     (Quad[5] = udata[pp + 5] * udata[pp + 5]));
00622         g = udata[pp] * udata[pp + 3] + udata[pp + 1] * udata[pp + 4] +
00623             udata[pp + 2] * udata[pp + 5];
00624         switch (*c) {
00625             case 0:
00626                 lf = 0;
00627                 lff = 0;
00628                 lfg = 0;
00629                 lg = 0;
00630                 lgg = 0;
00631                 break;
00632             case 1:
00633                 lf = 0.000206527095658582755255648 * f;
00634                 lff = 0.000206527095658582755255648;
00635                 lfg = 0;
00636                 lg = 0.0003614224174025198216973841 * g;
00637                 lgg = 0.0003614224174025198216973841;
00638                 break;
00639             case 2:
00640                 lf = 0.000354046449700427580438254 * f * f +
00641                     0.000191775160254398272737387 * g * g;
00642                 lff = 0.0007080928994008551608765075 * f;
00643                 lfg = 0.0003835503205087965454747749 * g;
00644                 lg = 0.0003835503205087965454747749 * f * g;
00645                 lgg = 0.0003835503205087965454747749 * f;
00646                 break;
00647             case 3:
00648                 lf = (0.000206527095658582755255648 + 0.000354046449700427580438254 * f) *
00649                     f +
00650                     0.000191775160254398272737387 * g * g;
00651                 lff = 0.000206527095658582755255648 + 0.000708092899400855160876508 * f;
00652                 lfg = 0.0003835503205087965454747749 * g;
00653                 lg = (0.000361422417402519821697384 + 0.000383550320508796545474775 * f) *
00654                     g;
00655                 lgg = 0.000361422417402519821697384 + 0.000383550320508796545474775 * f;
00656                 break;
00657             default:
00658                 errorKill(
00659                     "You need to specify a correct order in the weak-field expansion.");
00660             }
00661
00662     JMM[0] = lf + lff * Quad[0] +
00663             udata[3 + pp] * (2 * lfg * udata[pp] + lgg * udata[3 + pp]);
00664     JMM[1] =
00665         lff * udata[pp] * udata[1 + pp] + lfg * udata[1 + pp] * udata[3 + pp] +
00666         lfg * udata[pp] * udata[4 + pp] + lgg * udata[3 + pp] * udata[4 + pp];
00667     JMM[2] = lf + lff * Quad[1] +
00668             udata[4 + pp] * (2 * lfg * udata[1 + pp] + lgg * udata[4 + pp]);
00669     JMM[3] =
00670         lff * udata[pp] * udata[2 + pp] + lfg * udata[2 + pp] * udata[3 + pp] +

```

```

00671     lfg * udata[pp] * udata[5 + pp] + lgg * udata[3 + pp] * udata[5 + pp];
00672 JMM[4] = lff * udata[1 + pp] * udata[2 + pp] +
00673     lfg * udata[2 + pp] * udata[4 + pp] +
00674     lfg * udata[1 + pp] * udata[5 + pp] +
00675     lgg * udata[4 + pp] * udata[5 + pp];
00676 JMM[5] = lf + lff * Quad[2] +
00677     udata[5 + pp] * (2 * lfg * udata[2 + pp] + lgg * udata[5 + pp]);
00678 JMM[6] = lg + lfg * (Quad[0] - Quad[3 + 0]) +
00679     (-lff + lgg) * udata[pp] * udata[3 + pp];
00680 JMM[7] = -(udata[3 + pp] * (lff * udata[1 + pp] + lfg * udata[4 + pp])) +
00681     udata[pp] * (lfg * udata[1 + pp] + lgg * udata[4 + pp]);
00682 JMM[8] = -(udata[3 + pp] * (lff * udata[2 + pp] + lfg * udata[5 + pp])) +
00683     udata[pp] * (lfg * udata[2 + pp] + lgg * udata[5 + pp]);
00684 JMM[9] = -lf + lgg * Quad[0] +
00685     udata[3 + pp] * (-2 * lfg * udata[pp] + lff * udata[3 + pp]);
00686 JMM[10] = udata[1 + pp] * (lfg * udata[pp] + lgg * udata[3 + pp]) -
00687     (lff * udata[pp] + lfg * udata[3 + pp]) * udata[4 + pp];
00688 JMM[11] = lg + lfg * (Quad[1] - Quad[4 + 0]) +
00689     (-lff + lgg) * udata[1 + pp] * udata[4 + pp];
00690 JMM[12] = -(udata[4 + pp] * (lff * udata[2 + pp] + lfg * udata[5 + pp])) +
00691     udata[1 + pp] * (lfg * udata[2 + pp] + lgg * udata[5 + pp]);
00692 JMM[13] = lgg * udata[pp] * udata[1 + pp] +
00693     lff * udata[3 + pp] * udata[4 + pp] -
00694     lfg * (udata[1 + pp] * udata[3 + pp] + udata[pp] * udata[4 + pp]);
00695 JMM[14] = -lf + lgg * Quad[1] +
00696     udata[4 + pp] * (-2 * lfg * udata[1 + pp] + lff * udata[4 + pp]);
00697 JMM[15] = udata[2 + pp] * (lfg * udata[pp] + lgg * udata[3 + pp]) -
00698     (lff * udata[pp] + lfg * udata[3 + pp]) * udata[5 + pp];
00699 JMM[16] = udata[2 + pp] * (lfg * udata[1 + pp] + lgg * udata[4 + pp]) -
00700     (lff * udata[1 + pp] + lfg * udata[4 + pp]) * udata[5 + pp];
00701 JMM[17] = lg + lfg * (Quad[2] - Quad[5 + 0]) +
00702     (-lff + lgg) * udata[2 + pp] * udata[5 + pp];
00703 JMM[18] = lgg * udata[pp] * udata[2 + pp] +
00704     lff * udata[3 + pp] * udata[5 + pp] -
00705     lfg * (udata[2 + pp] * udata[3 + pp] + udata[pp] * udata[5 + pp]);
00706 JMM[19] =
00707     lgg * udata[1 + pp] * udata[2 + pp] +
00708     lff * udata[4 + pp] * udata[5 + pp] -
00709     lfg * (udata[2 + pp] * udata[4 + pp] + udata[1 + pp] * udata[5 + pp]);
00710 JMM[20] = -lf + lgg * Quad[2] +
00711     udata[5 + pp] * (-2 * lfg * udata[2 + pp] + lff * udata[5 + pp]);
00712
00713 h[0] = 0;
00714 h[1] = dxData[pp] * JMM[15] + dxData[1 + pp] * JMM[16] +
00715     dxData[2 + pp] * JMM[17] + dxData[3 + pp] * JMM[18] +
00716     dxData[4 + pp] * JMM[19] + dxData[5 + pp] * (-1 + JMM[20]);
00717 h[2] = -(dxData[pp] * JMM[10]) - dxData[1 + pp] * JMM[11] -
00718     dxData[2 + pp] * JMM[12] - dxData[3 + pp] * JMM[13] +
00719     dxData[4 + pp] * (1 - JMM[14]) - dxData[5 + pp] * JMM[19];
00720 h[3] = 0;
00721 h[4] = dxData[2 + pp];
00722 h[5] = -dxData[1 + pp];
00723 h[0] += -(dyData[pp] * JMM[15]) - dyData[1 + pp] * JMM[16] -
00724     dyData[2 + pp] * JMM[17] - dyData[3 + pp] * JMM[18] -
00725     dyData[4 + pp] * JMM[19] + dyData[5 + pp] * (1 - JMM[20]);
00726 h[1] += 0;
00727 h[2] += dyData[pp] * JMM[6] + dyData[1 + pp] * JMM[7] +
00728     dyData[2 + pp] * JMM[8] + dyData[3 + pp] * (-1 + JMM[9]) +
00729     dyData[4 + pp] * JMM[13] + dyData[5 + pp] * JMM[18];
00730 h[3] += -dyData[2 + pp];
00731 h[4] += 0;
00732 h[5] += dyData[pp];
00733 h[0] += dzData[pp] * JMM[10] + dzData[1 + pp] * JMM[11] +
00734     dzData[2 + pp] * JMM[12] + dzData[3 + pp] * JMM[13] +
00735     dzData[4 + pp] * (-1 + JMM[14]) + dzData[5 + pp] * JMM[19];
00736 h[1] += -(dzData[pp] * JMM[6]) - dzData[1 + pp] * JMM[7] -
00737     dzData[2 + pp] * JMM[8] + dzData[3 + pp] * (1 - JMM[9]) -
00738     dzData[4 + pp] * JMM[13] - dzData[5 + pp] * JMM[18];
00739 h[2] += 0;
00740 h[3] += dzData[1 + pp];
00741 h[4] += -dzData[pp];
00742 h[5] += 0;
00743 h[0] -= h[3] * JMM[6] + h[4] * JMM[10] + h[5] * JMM[15];
00744 h[1] -= h[3] * JMM[7] + h[4] * JMM[11] + h[5] * JMM[16];
00745 h[2] -= h[3] * JMM[8] + h[4] * JMM[12] + h[5] * JMM[17];
00746 dudata[pp + 0] =
00747     h[2] * (-JMM[3] * (1 + JMM[2])) + JMM[1] * JMM[4]) +
00748     h[1] * (JMM[3] * JMM[4] - JMM[1] * (1 + JMM[5])) +
00749     h[0] * (1 - JMM[4] * JMM[4] + JMM[5] + JMM[2] * (1 + JMM[5]));
00750 dudata[pp + 1] =
00751     h[2] * (JMM[3] * JMM[1] - (1 + JMM[0]) * JMM[4]) +
00752     h[1] * (1 - JMM[3] * JMM[3] + JMM[5] + JMM[0] * (1 + JMM[5])) +
00753     h[0] * (JMM[4] * JMM[3] - JMM[1] * (1 + JMM[5]));
00754 dudata[pp + 2] =
00755     h[2] * (1 - JMM[1] * JMM[1] + JMM[2] + JMM[0] * (1 + JMM[2])) +
00756     h[1] * (JMM[1] * JMM[3] - (1 + JMM[0]) * JMM[4]) +
00757     h[0] * (-((1 + JMM[2]) * JMM[3]) + JMM[1] * JMM[4]);

```

```

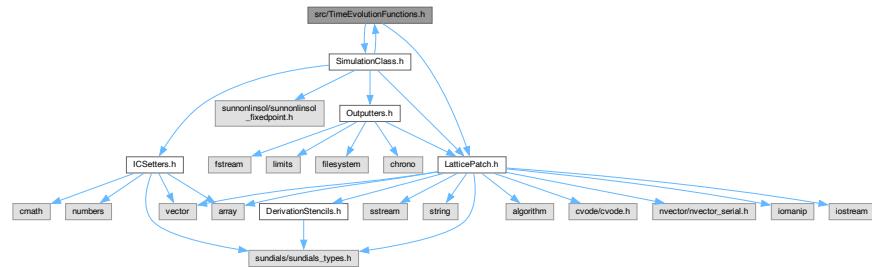
00758     detC =
00759         -((1 + JMM[2]) * (-1 + JMM[3] * JMM[3])) +
00760         (JMM[3] * JMM[1] - JMM[4]) * JMM[4] + JMM[5] + JMM[2] * JMM[5] +
00761         JMM[0] * (1 + JMM[2] - JMM[4] * JMM[4] + (1 + JMM[2]) * JMM[5]) -
00762         JMM[1] * (- (JMM[4] * JMM[3]) + JMM[1] * (1 + JMM[5]));
00763     dudata[pp + 0] /= detC;
00764     dudata[pp + 1] /= detC;
00765     dudata[pp + 2] /= detC;
00766     dudata[pp + 3] = h[3];
00767     dudata[pp + 4] = h[4];
00768     dudata[pp + 5] = h[5];
00769 }
00770 return;
00771 }

```

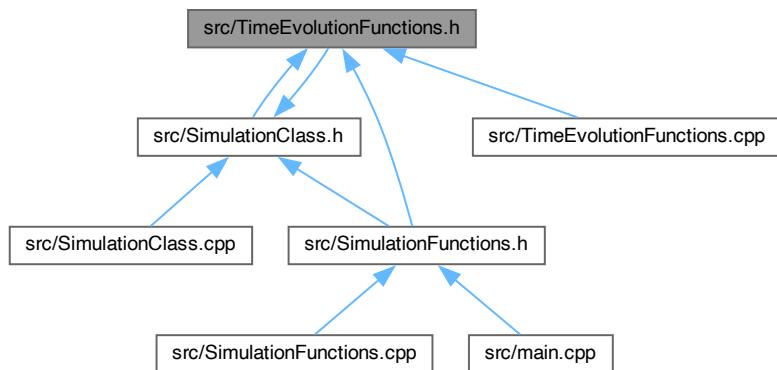
6.30 src/TimeEvolutionFunctions.h File Reference

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.

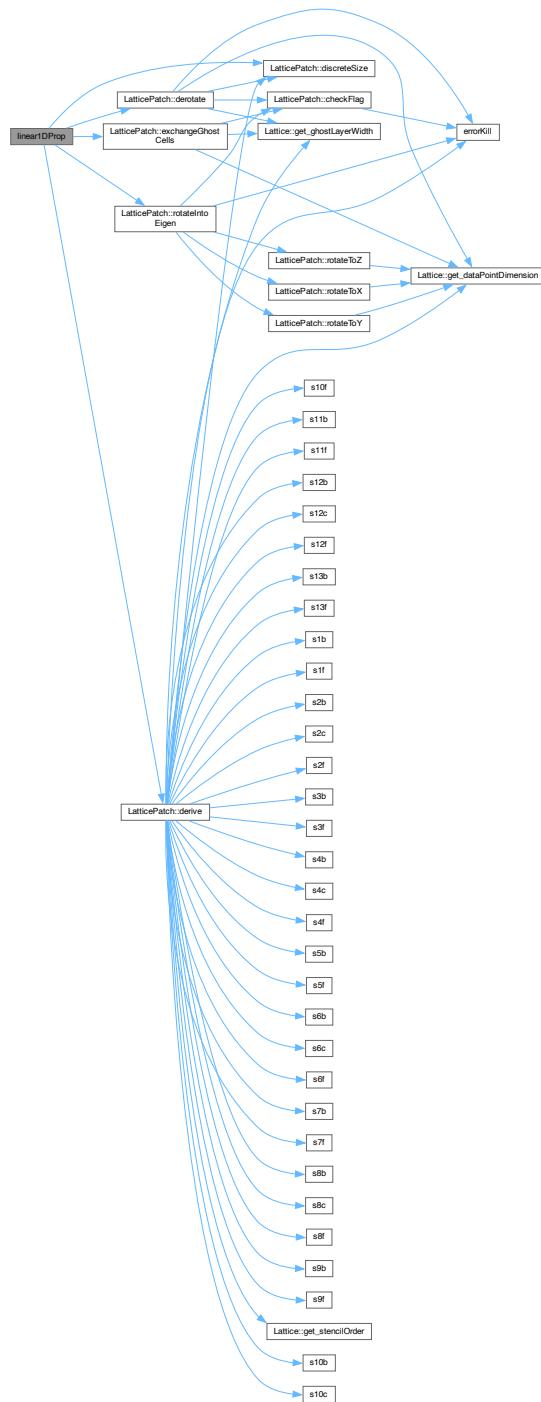
Definition at line 56 of file [TimeEvolutionFunctions.cpp](#).

```
00056
00057
00058 // pointers to temporal and spatial derivative data
00059 sunrealtype *duData = data->duData;
00060 sunrealtype *dxData = data->buffData[1 - 1];
00061
00062 // sequence along any dimension according to the scheme:
00063 data->exchangeGhostCells(1); // -> exchange halos
00064 data->rotateIntoEigen(
00065     1); // -> rotate all data to prepare derivative operation
00066 data->derive(1); // -> perform derivative approximation operation on it
00067 data->derotate(
00068     1, dxData); // -> derotate derived data for ensuing time-evolution
00069
00070 const sunindextype totalNP = data->discreteSize();
00071 sunindextype pp = 0;
00072 for (sunindextype i = 0; i < totalNP; i++) {
00073     pp = i * 6;
```

```
00074  /*
00075   simple vacuum Maxwell equations for the temporal derivatives using the
00076   spatial derivative only in x-direction without polarization or
00077   magnetization terms
00078 */
00079 duData[pp + 0] = 0;
00080 duData[pp + 1] = -dxData[pp + 5];
00081 duData[pp + 2] = dxData[pp + 4];
00082 duData[pp + 3] = 0;
00083 duData[pp + 4] = dxData[pp + 2];
00084 duData[pp + 5] = -dxData[pp + 1];
00085 }
00086 }
```

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.

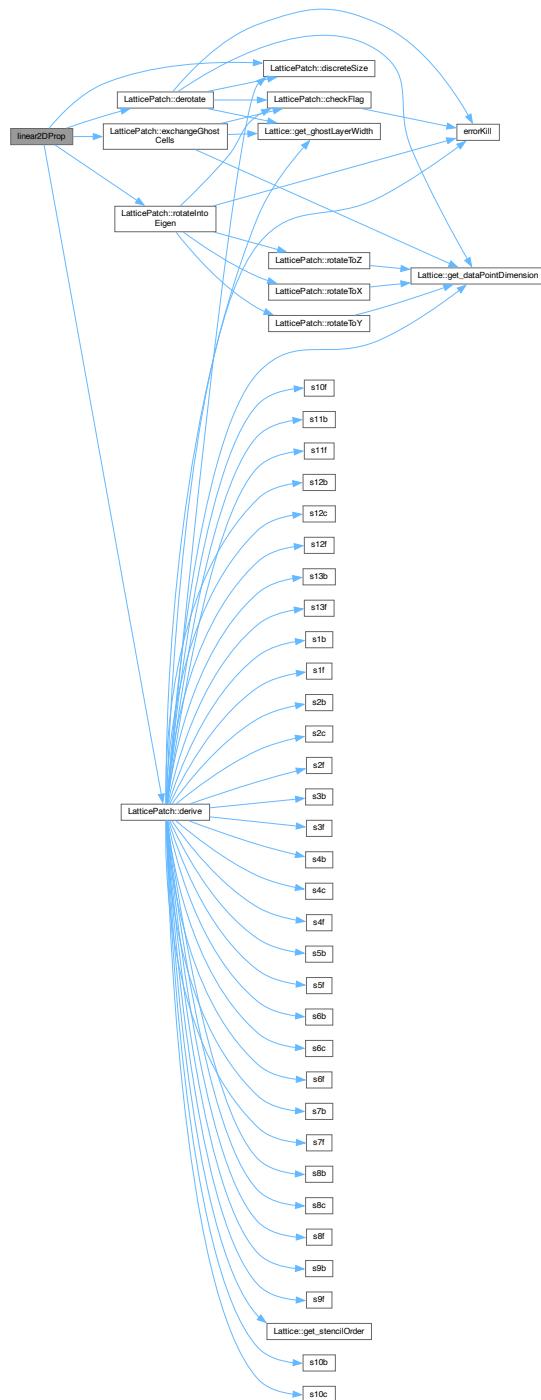
Maxwell propagation function for 2D – only for reference.

Definition at line 312 of file [TimeEvolutionFunctions.cpp](#).

```
00312  
00313  
00314     sunrealtype *duData = data->duData;  
00315     sunrealtype *dxData = data->buffData[1 - 1];  
00316     sunrealtype *dyData = data->buffData[2 - 1];  
00317  
00318     data->exchangeGhostCells(1);  
00319     data->rotateIntoEigen(1);  
00320     data->derive(1);  
00321     data->derotate(1, dxData);  
00322     data->exchangeGhostCells(2);  
00323     data->rotateIntoEigen(2);  
00324     data->derive(2);  
00325     data->derotate(2, dyData);  
00326  
00327     const sunindextype totalNP = data->discreteSize();  
00328     sunindextype pp = 0;  
00329     for (sunindextype i = 0; i < totalNP; i++) {  
00330         pp = i * 6;  
00331         duData[pp + 0] = dyData[pp + 5];  
00332         duData[pp + 1] = -dxData[pp + 5];  
00333         duData[pp + 2] = -dyData[pp + 3] + dxData[pp + 4];  
00334         duData[pp + 3] = -dyData[pp + 2];  
00335         duData[pp + 4] = dxData[pp + 2];  
00336         duData[pp + 5] = dyData[pp + 0] - dxData[pp + 1];  
00337     }  
00338 }
```

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.

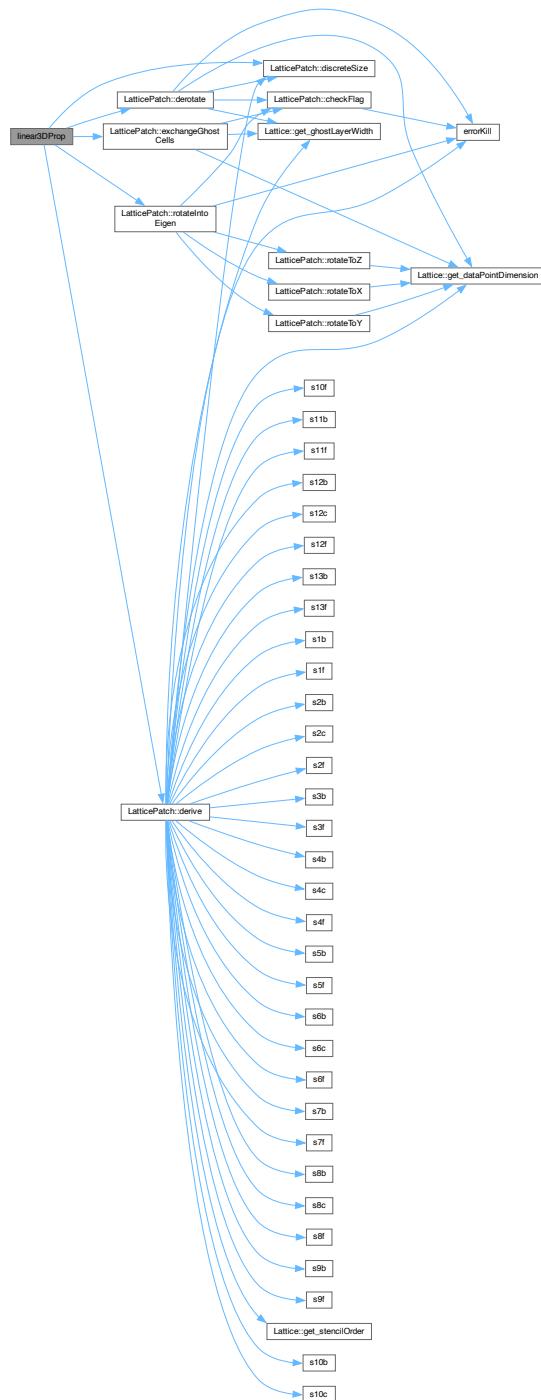
Maxwell propagation function for 3D – only for reference.

Definition at line 533 of file [TimeEvolutionFunctions.cpp](#).

```
00533  
00534  
00535 sunrealtyp *duData = data->duData;  
00536 sunrealtyp *dxData = data->buffData[1 - 1];  
00537 sunrealtyp *dyData = data->buffData[2 - 1];  
00538 sunrealtyp *dzData = data->buffData[3 - 1];  
00539  
00540 data->exchangeGhostCells(1);  
00541 data->rotateIntoEigen(1);  
00542 data->derive(1);  
00543 data->derotate(1, dxData);  
00544 data->exchangeGhostCells(2);  
00545 data->rotateIntoEigen(2);  
00546 data->derive(2);  
00547 data->derotate(2, dyData);  
00548 data->exchangeGhostCells(3);  
00549 data->rotateIntoEigen(3);  
00550 data->derive(3);  
00551 data->derotate(3, dzData);  
00552  
00553 const sunindextype totalNP = data->discreteSize();  
00554 sunindextype pp = 0;  
00555 for (sunindextype i = 0; i < totalNP; i++) {  
00556     pp = i * 6;  
00557     duData[pp + 0] = dyData[pp + 5] - dzData[pp + 4];  
00558     duData[pp + 1] = dzData[pp + 3] - dxData[pp + 5];  
00559     duData[pp + 2] = dxData[pp + 4] - dyData[pp + 3];  
00560     duData[pp + 3] = -dyData[pp + 2] + dzData[pp + 1];  
00561     duData[pp + 4] = -dzData[pp + 0] + dxData[pp + 2];  
00562     duData[pp + 5] = -dxData[pp + 1] + dyData[pp + 0];  
00563 }  
00564 }
```

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 89 of file TimeEvolutionFunctions.cpp.

```
00089
00090
00091 // NVector pointers to provided field values and their temp. derivatives
00092 #if defined(_MPI)
00093 #if !defined(_OPENMP)
00094     sunrealtype *udata = NV_DATA_P(u),
00095             *dudata = NV_DATA_P(udot);
00096 #elif defined(_OPENMP)
00097     sunrealtype *udata = N_VGetArrayPointer_MPIPlusX(u),
00098             *dudata = N_VGetArrayPointer_MPIPlusX(udot);
00099 #endif
00100 #elif defined(_OPENMP)
00101     sunrealtype *udata = NV_DATA_OMP(u),
00102             *dudata = NV_DATA_OMP(udot);
00103 #else
00104     sunrealtype *udata = NV_DATA_S(u),
00105             *dudata = NV_DATA_S(udot);
00106 #endif
00107
00108 // pointer to spatial derivatives via patch data
00109 sunrealtype *dxData = data->buffData[1 - 1];
00110
00111 // same sequence as in the linear case
00112 data->exchangeGhostCells(1);
00113 data->rotateIntoEigen(1);
00114 data->derive(1);
00115 data->derotate(1, dxData);
00116
00117 /*
00118 F and G are nonzero in the nonlinear case,
00119 polarization and magnetization derivatives
00120 w.r.t. E- and B-field go into the e.o.m.
00121 */
00122 static sunrealtype f, g; // em field invariants F, G
00123 // derivatives of HE Lagrangian w.r.t. field invariants
00124 static sunrealtype lf, lff, lfg, lg, lgg;
00125 // matrix to hold derivatives of polarization and magnetization
00126 static std::array<sunrealtype, 21> JMM;
00127 // array to hold E^2 and B^2 components
00128 static std::array<sunrealtype, 6> Quad;
00129 // array to hold intermediate temp. derivatives of E and B
00130 static std::array<sunrealtype, 6> h;
00131 // determinant needed for explicit matrix inversion
00132 static sunrealtype detC = nan("0x12345");
00133
00134 // number of points in the patch
00135 const sunindextype totalNP = data->discreteSize();
00136 #pragma omp parallel for default(none) \
00137 private(JMM, Quad, h, detC) \
00138 shared(totalNP, c, f, g, lf, lff, lfg, lg, lgg, udata, dudata, dxData) \
00139 schedule(static)
00140 for (sunindextype pp = 0; pp < totalNP * 6;
00141     pp += 6) { // loop over all 6dim points in the patch
00142     // em field Lorentz invariants F and G
00143     f = 0.5 * ((Quad[0] = udata[pp] * udata[pp]) +
00144                 (Quad[1] = udata[pp + 1] * udata[pp + 1]) +
00145                 (Quad[2] = udata[pp + 2] * udata[pp + 2]) -
00146                 (Quad[3] = udata[pp + 3] * udata[pp + 3]) -
00147                 (Quad[4] = udata[pp + 4] * udata[pp + 4]) -
00148                 (Quad[5] = udata[pp + 5] * udata[pp + 5]));
00149     g = udata[pp] * udata[pp + 3] + udata[pp + 1] * udata[pp + 4] +
00150         udata[pp + 2] * udata[pp + 5];
00151     // process/expansion order and corresponding derivative values of L
00152     // w.r.t. F, G
00153     switch (*c) {
00154     case 0: // linear Maxwell vacuum
00155         lf = 0;
00156         lff = 0;
00157         lfg = 0;
00158         lg = 0;
00159         lgg = 0;
00160         break;
00161     case 1: // only 4-photon processes
00162         lf = 0.000206527095658582755255648 * f;
```

```

00163     lff = 0.000206527095658582755255648;
00164     lfg = 0;
00165     lg = 0.0003614224174025198216973841 * g;
00166     lgg = 0.0003614224174025198216973841;
00167     break;
00168 case 2: // only 6-photon processes
00169     lf = 0.000354046449700427580438254 * f * f +
00170         0.000191775160254398272737387 * g * g;
00171     lff = 0.0007080928994008551608765075 * f;
00172     lfg = 0.0003835503205087965454747749 * g;
00173     lg = 0.0003835503205087965454747749 * f * g;
00174     lgg = 0.0003835503205087965454747749 * f;
00175     break;
00176 case 3: // 4- and 6-photon processes
00177     lf = (0.000206527095658582755255648 + 0.000354046449700427580438254 * f) *
00178         f +
00179         0.000191775160254398272737387 * g * g;
00180     lff = 0.000206527095658582755255648 + 0.0007080928994008551608765075 * f;
00181     lfg = 0.0003835503205087965454747749 * g;
00182     lg = (0.0003614224174025198216973841 +
00183         0.0003835503205087965454747749 * f) *
00184         g;
00185     lgg = 0.0003614224174025198216973841 + 0.0003835503205087965454747749 * f;
00186     break;
00187 default:
00188     errorKill(
00189         "You need to specify a correct order in the weak-field expansion.");
00190 }
00191
00192 // derivatives of polarization and magnetization w.r.t. E and B
00193 // Jpx(Ex)
00194 JMM[0] = lf + lff * Quad[0] +
00195     udata[3 + pp] * (2 * lfg * udata[pp] + lgg * udata[3 + pp]);
00196 // Jpx(Ey)
00197 JMM[1] =
00198     lff * udata[pp] * udata[1 + pp] + lfg * udata[1 + pp] * udata[3 + pp] +
00199     lfg * udata[pp] * udata[4 + pp] + lgg * udata[3 + pp] * udata[4 + pp];
00200 // Jpy(Ey)
00201 JMM[2] = lf + lff * Quad[1] +
00202     udata[4 + pp] * (2 * lfg * udata[1 + pp] + lgg * udata[4 + pp]);
00203 // Jpx(Ez) = Jpz(Ex)
00204 JMM[3] =
00205     lff * udata[pp] * udata[2 + pp] + lfg * udata[2 + pp] * udata[3 + pp] +
00206     lfg * udata[pp] * udata[5 + pp] + lgg * udata[3 + pp] * udata[5 + pp];
00207 // Jpy(Ez) = Jpz(Ey)
00208 JMM[4] = lff * udata[1 + pp] * udata[2 + pp] +
00209     lfg * udata[2 + pp] * udata[4 + pp] +
00210     lfg * udata[1 + pp] * udata[5 + pp] +
00211     lgg * udata[4 + pp] * udata[5 + pp];
00212 // Jpz(Ez)
00213 JMM[5] = lf + lff * Quad[2] +
00214     udata[5 + pp] * (2 * lfg * udata[2 + pp] + lgg * udata[5 + pp]);
00215 // Jpx(Bx) = Jmx(Ex)
00216 JMM[6] = lg + lfg * (Quad[0] - Quad[3 + 0]) +
00217     (-lff + lgg) * udata[pp] * udata[3 + pp];
00218 // Jpy(Bx) = Jmx(Ey)
00219 JMM[7] = -(udata[3 + pp] * (lff * udata[1 + pp] + lfg * udata[4 + pp])) +
00220     udata[pp] * (lfg * udata[1 + pp] + lgg * udata[4 + pp]);
00221 // Jpz(Bx) = Jmx(Ez)
00222 JMM[8] = -(udata[3 + pp] * (lff * udata[2 + pp] + lfg * udata[5 + pp])) +
00223     udata[pp] * (lfg * udata[2 + pp] + lgg * udata[5 + pp]);
00224 // Jmx(Bx)
00225 JMM[9] = -lf + lgg * Quad[0] +
00226     udata[3 + pp] * (-2 * lfg * udata[pp] + lff * udata[3 + pp]);
00227 // Jpx(By) = Jmy(Ex)
00228 JMM[10] = udata[1 + pp] * (lfg * udata[pp] + lgg * udata[3 + pp]) -
00229     (lff * udata[pp] + lfg * udata[3 + pp]) * udata[4 + pp];
00230 // Jpy(By) = Jmy(Ey)
00231 JMM[11] = lg + lfg * (Quad[1] - Quad[4 + 0]) +
00232     (-lff + lgg) * udata[1 + pp] * udata[4 + pp];
00233 // Jpz(By) = Jmy(Ez)
00234 JMM[12] = -(udata[4 + pp] * (lff * udata[2 + pp] + lfg * udata[5 + pp])) +
00235     udata[1 + pp] * (lfg * udata[2 + pp] + lgg * udata[5 + pp]);
00236 // Jmx(By) = Jmy(Bx)
00237 JMM[13] = lgg * udata[pp] * udata[1 + pp] +
00238     lff * udata[3 + pp] * udata[4 + pp] -
00239     lfg * (udata[1 + pp] * udata[3 + pp] + udata[pp] * udata[4 + pp]);
00240 // Jmy(By)
00241 JMM[14] = -lf + lgg * Quad[1] +
00242     udata[4 + pp] * (-2 * lfg * udata[1 + pp] + lff * udata[4 + pp]);
00243 // Jmz(Ex) = Jpx(Bz)
00244 JMM[15] = udata[2 + pp] * (lfg * udata[pp] + lgg * udata[3 + pp]) -
00245     (lff * udata[pp] + lfg * udata[3 + pp]) * udata[5 + pp];
00246 // Jmz(Ey) = Jpy(Bz)
00247 JMM[16] = udata[2 + pp] * (lfg * udata[1 + pp] + lgg * udata[4 + pp]) -
00248     (lff * udata[1 + pp] + lfg * udata[4 + pp]) * udata[5 + pp];
00249 // Jpz(Bz) = Jmz(Ez)

```

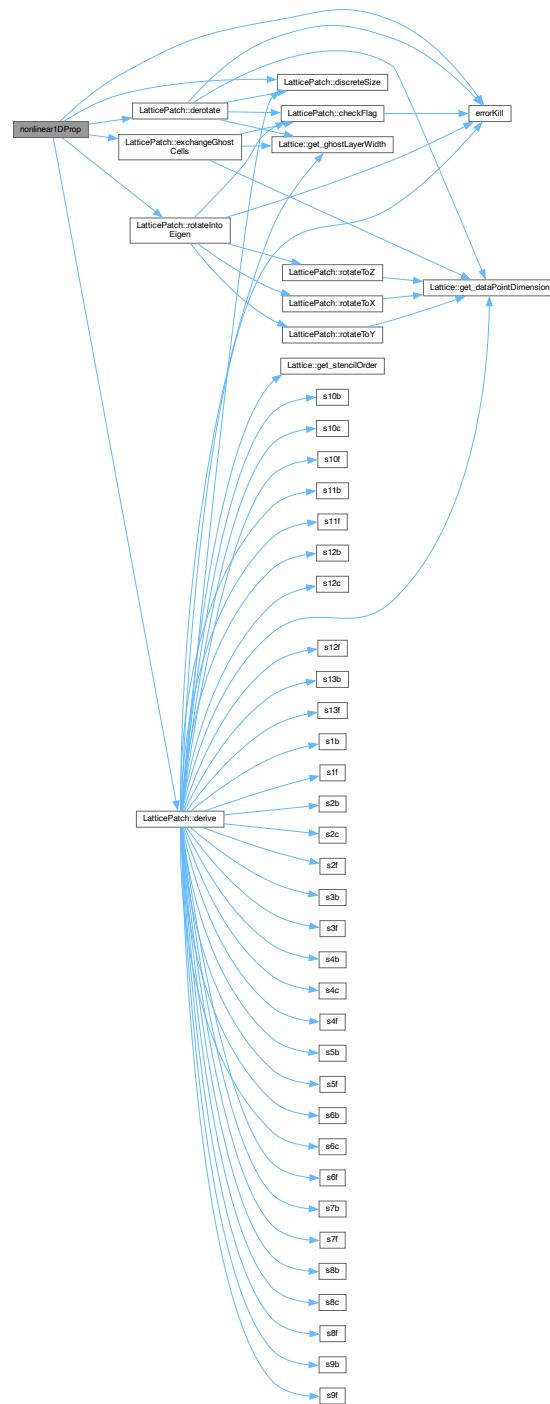
```

00250     JMM[17] = lg + lfg * (Quad[2] - Quad[5 + 0]) +
00251         (-lff + lgg) * udata[2 + pp] * udata[5 + pp];
00252 // Jmz(Bx) = Jmx(Bz)
00253     JMM[18] = lgg * udata[pp] * udata[2 + pp] +
00254         lff * udata[3 + pp] * udata[5 + pp] -
00255         lfg * (udata[2 + pp] * udata[3 + pp] + udata[pp] * udata[5 + pp]);
00256 // Jmy(Bz) = Jmz(By)
00257     JMM[19] =
00258         lgg * udata[1 + pp] * udata[2 + pp] +
00259         lff * udata[4 + pp] * udata[5 + pp] -
00260         lfg * (udata[2 + pp] * udata[4 + pp] + udata[1 + pp] * udata[5 + pp]);
00261 // Jmz(Bz)
00262     JMM[20] = -lf + lgg * Quad[2] +
00263         udata[5 + pp] * (-2 * lfg * udata[2 + pp] + lff * udata[5 + pp]);
00264
00265 // apply Z
00266 // top block: -QJm(E)*E, Q-QJm(B)*B
00267     h[0] = 0;
00268     h[1] = dxData[pp] * JMM[15] + dxData[1 + pp] * JMM[16] +
00269         dxData[2 + pp] * JMM[17] + dxData[3 + pp] * JMM[18] +
00270         dxData[4 + pp] * JMM[19] + dxData[5 + pp] * (-1 + JMM[20]);
00271     h[2] = -(dxData[pp] * JMM[10]) - dxData[1 + pp] * JMM[11] -
00272         dxData[2 + pp] * JMM[12] - dxData[3 + pp] * JMM[13] +
00273         dxData[4 + pp] * (1 - JMM[14]) - dxData[5 + pp] * JMM[19];
00274 // bottom blocks: -Q*E
00275     h[3] = 0;
00276     h[4] = dxData[2 + pp];
00277     h[5] = -dxData[1 + pp];
00278 // (1+A)^-1 applies only to E components
00279 // -Jp(B)*B
00280     h[0] -= h[3] * JMM[6] + h[4] * JMM[10] + h[5] * JMM[15];
00281     h[1] -= h[3] * JMM[7] + h[4] * JMM[11] + h[5] * JMM[16];
00282     h[2] -= h[3] * JMM[8] + h[4] * JMM[12] + h[5] * JMM[17];
00283 // apply C^-1 explicitly, with C=1+Jp(E)
00284     dudata[pp + 0] =
00285         h[2] * (-JMM[3] * (1 + JMM[2])) + JMM[1] * JMM[4]) +
00286         h[1] * (JMM[3] * JMM[4] - JMM[1] * (1 + JMM[5])) +
00287         h[0] * (1 - JMM[4] * JMM[4] + JMM[5] + JMM[2] * (1 + JMM[5]));
00288     dudata[pp + 1] =
00289         h[2] * (JMM[3] * JMM[1] - (1 + JMM[0]) * JMM[4]) +
00290         h[1] * (1 - JMM[3] * JMM[3] + JMM[5] + JMM[0] * (1 + JMM[5])) +
00291         h[0] * (JMM[4] * JMM[3] - JMM[1] * (1 + JMM[5]));
00292     dudata[pp + 2] =
00293         h[2] * (1 - JMM[1] * JMM[1] + JMM[2] + JMM[0] * (1 + JMM[2])) +
00294         h[1] * (JMM[1] * JMM[3] - (1 + JMM[0]) * JMM[4]) +
00295         h[0] * (((1 + JMM[2]) * JMM[3]) + JMM[1] * JMM[4]);
00296     detC = // determinant of C
00297         -((1 + JMM[2]) * (-1 + JMM[3] * JMM[3])) +
00298         (JMM[3] * JMM[1] - JMM[4]) * JMM[4] + JMM[5] + JMM[2] * JMM[5] +
00299         JMM[0] * (1 + JMM[2] - JMM[4] * JMM[4] + (1 + JMM[2]) * JMM[5]) -
00300         JMM[1] * (-JMM[4] * JMM[3]) + JMM[1] * (1 + JMM[5]));
00301     dudata[pp + 0] /= detC;
00302     dudata[pp + 1] /= detC;
00303     dudata[pp + 2] /= detC;
00304     dudata[pp + 3] = h[3];
00305     dudata[pp + 4] = h[4];
00306     dudata[pp + 5] = h[5];
00307 }
00308 return;
00309 }
```

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 341 of file [TimeEvolutionFunctions.cpp](#).

```
00341
00342
00343 #if defined(_MPI)
00344 #if !defined(_OPENMP)
00345     sunrealtype *udata = NV_DATA_P(u),
00346             *dudata = NV_DATA_P(udot);
00347 #elif defined(_OPENMP)
00348     sunrealtype *udata = N_VGetArrayPointer_MPIPlusX(u),
00349             *dudata = N_VGetArrayPointer_MPIPlusX(udot);
00350 #endif
00351 #elif defined(_OPENMP)
00352     sunrealtype *udata = NV_DATA_OMP(u),
00353             *dudata = NV_DATA_OMP(udot);
00354 #else
00355     sunrealtype *udata = NV_DATA_S(u),
00356             *dudata = NV_DATA_S(udot);
00357 #endif
00358
00359     sunrealtype *dxData = data->buffData[1 - 1];
00360     sunrealtype *dyData = data->buffData[2 - 1];
00361
00362     data->exchangeGhostCells(1);
00363     data->rotateIntoEigen(1);
00364     data->derive(1);
00365     data->derotate(1, dxData);
00366     data->exchangeGhostCells(2);
00367     data->rotateIntoEigen(2);
00368     data->derive(2);
00369     data->derotate(2, dyData);
00370
00371     static sunrealtype f, g;
00372     static sunrealtype lf, lff, lfg, lg, lgg;
00373     static std::array<sunrealtype, 21> JMM;
00374     static std::array<sunrealtype, 6> Quad;
00375     static std::array<sunrealtype, 6> h;
00376     static sunrealtype detC;
00377
00378     const sunindextype totalNP = data->discreteSize();
00379     #pragma omp parallel for default(none) \
00380     private(JMM, Quad, h, detC) \
00381     shared(totalNP, c, f, g, lf, lff, lfg, lg, lgg, udata, dudata, \
00382           dxData, dyData) \
00383     schedule(static)
```

```

00384     for (sunindextype pp = 0; pp < totalNP * 6; pp += 6) {
00385         f = 0.5 * ((Quad[0] = udata[pp] * udata[pp]) +
00386                     (Quad[1] = udata[pp + 1] * udata[pp + 1]) +
00387                     (Quad[2] = udata[pp + 2] * udata[pp + 2]) -
00388                     (Quad[3] = udata[pp + 3] * udata[pp + 3]) -
00389                     (Quad[4] = udata[pp + 4] * udata[pp + 4]) -
00390                     (Quad[5] = udata[pp + 5] * udata[pp + 5]));
00391         g = udata[pp] * udata[pp + 3] + udata[pp + 1] * udata[pp + 4] +
00392             udata[pp + 2] * udata[pp + 5];
00393         switch (*c) {
00394             case 0:
00395                 lf = 0;
00396                 lff = 0;
00397                 lfg = 0;
00398                 lg = 0;
00399                 lgg = 0;
00400                 break;
00401             case 1:
00402                 lf = 0.000206527095658582755255648 * f;
00403                 lff = 0.000206527095658582755255648;
00404                 lfg = 0;
00405                 lg = 0.0003614224174025198216973841 * g;
00406                 lgg = 0.0003614224174025198216973841;
00407                 break;
00408             case 2:
00409                 lf = 0.000354046449700427580438254 * f * f +
00410                     0.000191775160254398272737387 * g * g;
00411                 lff = 0.0007080928994008551608765075 * f;
00412                 lfg = 0.0003835503205087965454747749 * g;
00413                 lg = 0.0003835503205087965454747749 * f * g;
00414                 lgg = 0.0003835503205087965454747749 * f;
00415                 break;
00416             case 3:
00417                 lf = (0.000206527095658582755255648 + 0.000354046449700427580438254 * f) *
00418                     f +
00419                     0.000191775160254398272737387 * g * g;
00420                 lff = 0.000206527095658582755255648 + 0.000708092899400855160876508 * f;
00421                 lfg = 0.0003835503205087965454747749 * g;
00422                 lg = (0.000361422417402519821697384 + 0.000383550320508796545474775 * f) *
00423                     g;
00424                 lgg = 0.000361422417402519821697384 + 0.000383550320508796545474775 * f;
00425                 break;
00426             default:
00427                 errorKill(
00428                     "You need to specify a correct order in the weak-field expansion.");
00429     }
00430
00431     JMM[0] = lf + lff * Quad[0] +
00432             udata[3 + pp] * (2 * lfg * udata[pp] + lgg * udata[3 + pp]);
00433     JMM[1] =
00434         lff * udata[pp] * udata[1 + pp] + lfg * udata[1 + pp] * udata[3 + pp] +
00435         lfg * udata[pp] * udata[4 + pp] + lgg * udata[3 + pp] * udata[4 + pp];
00436     JMM[2] = lf + lff * Quad[1] +
00437         udata[4 + pp] * (2 * lfg * udata[1 + pp] + lgg * udata[4 + pp]);
00438     JMM[3] =
00439         lff * udata[pp] * udata[2 + pp] + lfg * udata[2 + pp] * udata[3 + pp] +
00440         lfg * udata[pp] * udata[5 + pp] + lgg * udata[3 + pp] * udata[5 + pp];
00441     JMM[4] = lff * udata[1 + pp] * udata[2 + pp] +
00442         lfg * udata[2 + pp] * udata[4 + pp] +
00443         lfg * udata[1 + pp] * udata[5 + pp] +
00444         lgg * udata[4 + pp] * udata[5 + pp];
00445     JMM[5] = lf + lff * Quad[2] +
00446         udata[5 + pp] * (2 * lfg * udata[2 + pp] + lgg * udata[5 + pp]);
00447     JMM[6] = lg + lfg * (Quad[0] - Quad[3 + 0]) +
00448         (-lff + lgg) * udata[pp] * udata[3 + pp];
00449     JMM[7] = -(udata[3 + pp] * (lff * udata[1 + pp] + lfg * udata[4 + pp])) +
00450         udata[pp] * (lfg * udata[1 + pp] + lgg * udata[4 + pp]);
00451     JMM[8] = -(udata[3 + pp] * (lff * udata[2 + pp] + lfg * udata[5 + pp])) +
00452         udata[pp] * (lfg * udata[2 + pp] + lgg * udata[5 + pp]);
00453     JMM[9] = -lf + lfg * Quad[0] +
00454         udata[3 + pp] * (-2 * lfg * udata[pp] + lff * udata[3 + pp]);
00455     JMM[10] = udata[1 + pp] * (lfg * udata[pp] + lgg * udata[3 + pp]) -
00456         (lff * udata[pp] + lfg * udata[3 + pp]) * udata[4 + pp];
00457     JMM[11] = lg + lfg * (Quad[1] - Quad[4 + 0]) +
00458         (-lff + lgg) * udata[1 + pp] * udata[4 + pp];
00459     JMM[12] = -(udata[4 + pp] * (lff * udata[2 + pp] + lfg * udata[5 + pp])) +
00460         udata[1 + pp] * (lfg * udata[2 + pp] + lgg * udata[5 + pp]);
00461     JMM[13] = lgg * udata[pp] * udata[1 + pp] +
00462         lff * udata[3 + pp] * udata[4 + pp] -
00463         lfg * (udata[1 + pp] * udata[3 + pp] + udata[pp] * udata[4 + pp]);
00464     JMM[14] = -lf + lgg * Quad[1] +
00465         udata[4 + pp] * (-2 * lfg * udata[1 + pp] + lff * udata[4 + pp]);
00466     JMM[15] = udata[2 + pp] * (lfg * udata[pp] + lgg * udata[3 + pp]) -
00467         (lff * udata[pp] + lfg * udata[3 + pp]) * udata[5 + pp];
00468     JMM[16] = udata[2 + pp] * (lfg * udata[1 + pp] + lgg * udata[4 + pp]) -
00469         (lff * udata[1 + pp] + lfg * udata[4 + pp]) * udata[5 + pp];
00470     JMM[17] = lg + lfg * (Quad[2] - Quad[5 + 0]) +

```

```

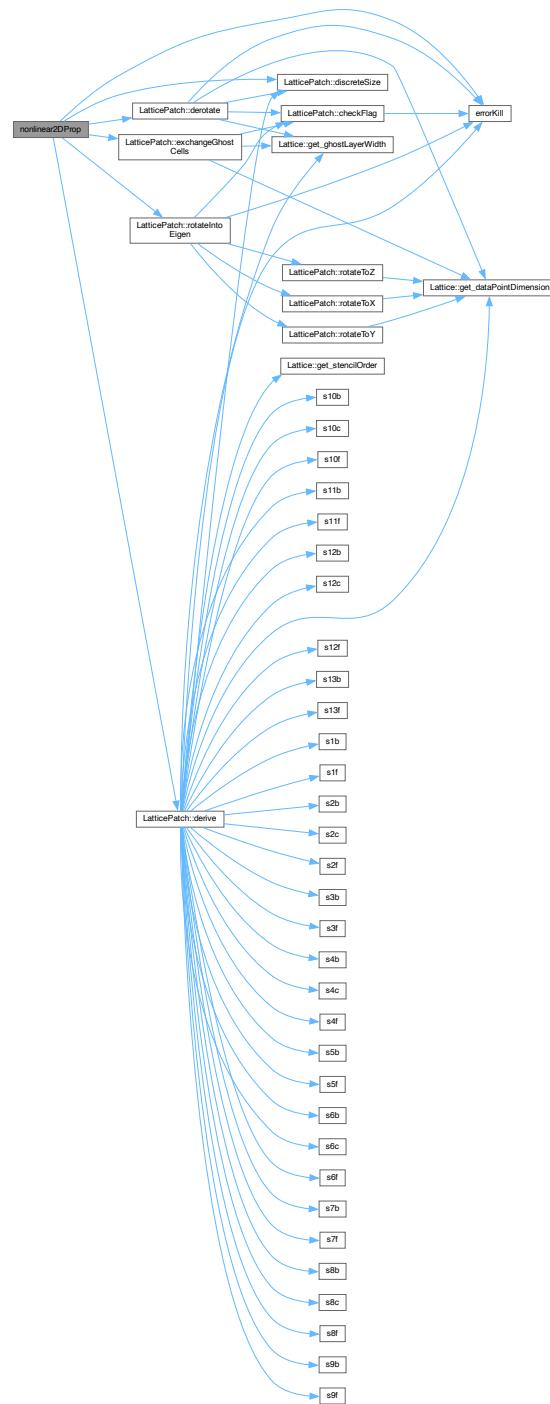
00471     (-lff + lgg) * udata[2 + pp] * udata[5 + pp];
00472 JMM[18] = lgg * udata[pp] * udata[2 + pp] +
00473     lff * udata[3 + pp] * udata[5 + pp] -
00474     lfg * (udata[2 + pp] * udata[3 + pp] + udata[pp] * udata[5 + pp]);
00475 JMM[19] =
00476     lgg * udata[1 + pp] * udata[2 + pp] +
00477     lff * udata[4 + pp] * udata[5 + pp] -
00478     lfg * (udata[2 + pp] * udata[4 + pp] + udata[1 + pp] * udata[5 + pp]);
00479 JMM[20] = -lf + lgg * Quad[2] +
00480     udata[5 + pp] * (-2 * lfg * udata[2 + pp] + lff * udata[5 + pp]);
00481
00482 h[0] = 0;
00483 h[1] = dxData[pp] * JMM[15] + dxData[1 + pp] * JMM[16] +
00484     dxData[2 + pp] * JMM[17] + dxData[3 + pp] * JMM[18] +
00485     dxData[4 + pp] * JMM[19] + dxData[5 + pp] * (-1 + JMM[20]);
00486 h[2] = -(dxData[pp] * JMM[10]) - dxData[1 + pp] * JMM[11] -
00487     dxData[2 + pp] * JMM[12] - dxData[3 + pp] * JMM[13] +
00488     dxData[4 + pp] * (1 - JMM[14]) - dxData[5 + pp] * JMM[19];
00489 h[3] = 0;
00490 h[4] = dxData[2 + pp];
00491 h[5] = -dxData[1 + pp];
00492 h[0] += -(dyData[pp] * JMM[15]) - dyData[1 + pp] * JMM[16] -
00493     dyData[2 + pp] * JMM[17] - dyData[3 + pp] * JMM[18] -
00494     dyData[4 + pp] * JMM[19] + dyData[5 + pp] * (1 - JMM[20]);
00495 h[1] += 0;
00496 h[2] += dyData[pp] * JMM[6] + dyData[1 + pp] * JMM[7] +
00497     dyData[2 + pp] * JMM[8] + dyData[3 + pp] * (-1 + JMM[9]) +
00498     dyData[4 + pp] * JMM[13] + dyData[5 + pp] * JMM[18];
00499 h[3] += -dyData[2 + pp];
00500 h[4] += 0;
00501 h[5] += dyData[pp];
00502 h[0] -= h[3] * JMM[6] + h[4] * JMM[10] + h[5] * JMM[15];
00503 h[1] -= h[3] * JMM[7] + h[4] * JMM[11] + h[5] * JMM[16];
00504 h[2] -= h[3] * JMM[8] + h[4] * JMM[12] + h[5] * JMM[17];
00505 dudata[pp + 0] =
00506     h[2] * (-JMM[3] * (1 + JMM[2])) + JMM[1] * JMM[4]) +
00507     h[1] * (JMM[3] * JMM[4] - JMM[1] * (1 + JMM[5])) +
00508     h[0] * (1 - JMM[4] * JMM[4] + JMM[5] + JMM[2] * (1 + JMM[5]));
00509 dudata[pp + 1] =
00510     h[2] * (JMM[3] * JMM[1] - (1 + JMM[0]) * JMM[4]) +
00511     h[1] * (1 - JMM[3] * JMM[3] + JMM[5] + JMM[0] * (1 + JMM[5])) +
00512     h[0] * (JMM[4] * JMM[3] - JMM[1] * (1 + JMM[5]));
00513 dudata[pp + 2] =
00514     h[2] * (1 - JMM[1] * JMM[1] + JMM[2] + JMM[0] * (1 + JMM[2])) +
00515     h[1] * (JMM[1] * JMM[3] - (1 + JMM[0]) * JMM[4]) +
00516     h[0] * (((1 + JMM[2]) * JMM[3]) + JMM[1] * JMM[4]);
00517 detC =
00518     -((1 + JMM[2]) * (-1 + JMM[3] * JMM[3])) +
00519     (JMM[3] * JMM[1] - JMM[4]) * JMM[4] + JMM[5] + JMM[2] * JMM[5] +
00520     JMM[0] * (1 + JMM[2] - JMM[4] * JMM[4] + (1 + JMM[2]) * JMM[5]) -
00521     JMM[1] * (-JMM[4] * JMM[3]) + JMM[1] * (1 + JMM[5]));
00522 dudata[pp + 0] /= detC;
00523 dudata[pp + 1] /= detC;
00524 dudata[pp + 2] /= detC;
00525 dudata[pp + 3] = h[3];
00526 dudata[pp + 4] = h[4];
00527 dudata[pp + 5] = h[5];
00528 }
00529 return;
00530 }

```

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 567 of file [TimeEvolutionFunctions.cpp](#).

```
00567
00568
00569 #if defined(_MPI)
00570 #if !defined(_OPENMP)
00571     sunrealtype *udata = NV_DATA_P(u),
00572                 *dudata = NV_DATA_P(udot);
00573 #elif defined(_OPENMP)
00574     sunrealtype *udata = N_VGetArrayPointer_MPIPlusX(u),
00575                 *dudata = N_VGetArrayPointer_MPIPlusX(udot);
00576 #endif
00577 #elif defined(_OPENMP)
00578     sunrealtype *udata = NV_DATA_OMP(u),
00579                 *dudata = NV_DATA_OMP(udot);
00580 #else
00581     sunrealtype *udata = NV_DATA_S(u),
00582                 *dudata = NV_DATA_S(udot);
00583 #endif
00584
00585     sunrealtype *dxData = data->buffData[1 - 1];
00586     sunrealtype *dyData = data->buffData[2 - 1];
00587     sunrealtype *dzData = data->buffData[3 - 1];
00588
00589     data->exchangeGhostCells(1);
00590     data->rotateIntoEigen(1);
00591     data->derive(1);
00592     data->derotate(1,dxData);
00593     data->exchangeGhostCells(2);
00594     data->rotateIntoEigen(2);
00595     data->derive(2);
00596     data->derotate(2,dyData);
00597     data->exchangeGhostCells(3);
00598     data->rotateIntoEigen(3);
00599     data->derive(3);
00600     data->derotate(3,dzData);
00601
00602     static sunrealtype f, g;
00603     static sunrealtype lff, lfg, lg, lgg;
00604     static std::array<sunrealtype, 21> JMM;
00605     static std::array<sunrealtype, 6> Quad;
00606     static std::array<sunrealtype, 6> h;
00607     static sunrealtype detC = nan("0x12345");
00608
00609     const sunindextype totalNP = data->discreteSize();
```

```

00610 #pragma omp parallel for default(none) \
00611 private(JMM, Quad, h, detC) \
00612 shared(totalNP, c, f, g, lf, lff, lfg, lg, lgg, udata, dudata, \
00613     dxData, dyData, dzData) \
00614 schedule(static)
00615 for (sunindextype pp = 0; pp < totalNP * 6; pp += 6) {
00616     f = 0.5 * ((Quad[0] = udata[pp] * udata[pp]) +
00617                 (Quad[1] = udata[pp + 1] * udata[pp + 1]) +
00618                 (Quad[2] = udata[pp + 2] * udata[pp + 2]) -
00619                 (Quad[3] = udata[pp + 3] * udata[pp + 3]) -
00620                 (Quad[4] = udata[pp + 4] * udata[pp + 4]) -
00621                 (Quad[5] = udata[pp + 5] * udata[pp + 5]));
00622     g = udata[pp] * udata[pp + 3] + udata[pp + 1] * udata[pp + 4] +
00623         udata[pp + 2] * udata[pp + 5];
00624     switch (*c) {
00625     case 0:
00626         lf = 0;
00627         lff = 0;
00628         lfg = 0;
00629         lg = 0;
00630         lgg = 0;
00631         break;
00632     case 1:
00633         lf = 0.000206527095658582755255648 * f;
00634         lff = 0.000206527095658582755255648;
00635         lfg = 0;
00636         lg = 0.0003614224174025198216973841 * g;
00637         lgg = 0.0003614224174025198216973841;
00638         break;
00639     case 2:
00640         lf = 0.000354046449700427580438254 * f * f +
00641             0.000191775160254398272737387 * g * g;
00642         lff = 0.0007080928994008551608765075 * f;
00643         lfg = 0.0003835503205087965454747749 * g;
00644         lg = 0.0003835503205087965454747749 * f * g;
00645         lgg = 0.0003835503205087965454747749 * f;
00646         break;
00647     case 3:
00648         lf = (0.000206527095658582755255648 + 0.000354046449700427580438254 * f) *
00649             f +
00650             0.000191775160254398272737387 * g * g;
00651         lff = 0.000206527095658582755255648 + 0.000708092899400855160876508 * f;
00652         lfg = 0.0003835503205087965454747749 * g;
00653         lg = (0.000361422417402519821697384 + 0.000383550320508796545474775 * f) *
00654             g;
00655         lgg = 0.000361422417402519821697384 + 0.000383550320508796545474775 * f;
00656         break;
00657     default:
00658         errorKill(
00659             "You need to specify a correct order in the weak-field expansion.");
00660     }
00661
00662 JMM[0] = lf + lff * Quad[0] +
00663     udata[3 + pp] * (2 * lfg * udata[pp] + lgg * udata[3 + pp]);
00664 JMM[1] =
00665     lff * udata[pp] * udata[1 + pp] + lfg * udata[1 + pp] * udata[3 + pp] +
00666     lfg * udata[pp] * udata[4 + pp] + lgg * udata[3 + pp] * udata[4 + pp];
00667 JMM[2] = lf + lff * Quad[1] +
00668     udata[4 + pp] * (2 * lfg * udata[1 + pp] + lgg * udata[4 + pp]);
00669 JMM[3] =
00670     lff * udata[pp] * udata[2 + pp] + lfg * udata[2 + pp] * udata[3 + pp] +
00671     lfg * udata[pp] * udata[5 + pp] + lgg * udata[3 + pp] * udata[5 + pp];
00672 JMM[4] = lff * udata[1 + pp] * udata[2 + pp] +
00673     lfg * udata[2 + pp] * udata[4 + pp] +
00674     lfg * udata[1 + pp] * udata[5 + pp] +
00675     lgg * udata[4 + pp] * udata[5 + pp];
00676 JMM[5] = lf + lff * Quad[2] +
00677     udata[5 + pp] * (2 * lfg * udata[2 + pp] + lgg * udata[5 + pp]);
00678 JMM[6] = lg + lfg * (Quad[0] - Quad[3 + 0]) +
00679     (-lff + lgg) * udata[pp] * udata[3 + pp];
00680 JMM[7] = -(udata[3 + pp] * (lff * udata[1 + pp] + lfg * udata[4 + pp])) +
00681     udata[pp] * (lfg * udata[1 + pp] + lgg * udata[4 + pp]);
00682 JMM[8] = -(udata[3 + pp] * (lff * udata[2 + pp] + lfg * udata[5 + pp])) +
00683     udata[pp] * (lfg * udata[2 + pp] + lgg * udata[5 + pp]);
00684 JMM[9] = -lf + lgg * Quad[0] +
00685     udata[3 + pp] * (-2 * lfg * udata[pp] + lff * udata[3 + pp]);
00686 JMM[10] = udata[1 + pp] * (lfg * udata[pp] + lgg * udata[3 + pp]) -
00687     (lff * udata[pp] + lfg * udata[3 + pp]) * udata[4 + pp];
00688 JMM[11] = lg + lfg * (Quad[1] - Quad[4 + 0]) +
00689     (-lff + lgg) * udata[1 + pp] * udata[4 + pp];
00690 JMM[12] = -(udata[4 + pp] * (lff * udata[2 + pp] + lfg * udata[5 + pp])) +
00691     udata[1 + pp] * (lfg * udata[2 + pp] + lgg * udata[5 + pp]);
00692 JMM[13] = lgg * udata[pp] * udata[1 + pp] +
00693     lff * udata[3 + pp] * udata[4 + pp] -
00694     lfg * (udata[1 + pp] * udata[3 + pp] + udata[pp] * udata[4 + pp]);
00695 JMM[14] = -lf + lgg * Quad[1] +
00696     udata[4 + pp] * (-2 * lfg * udata[1 + pp] + lff * udata[4 + pp]);

```

```

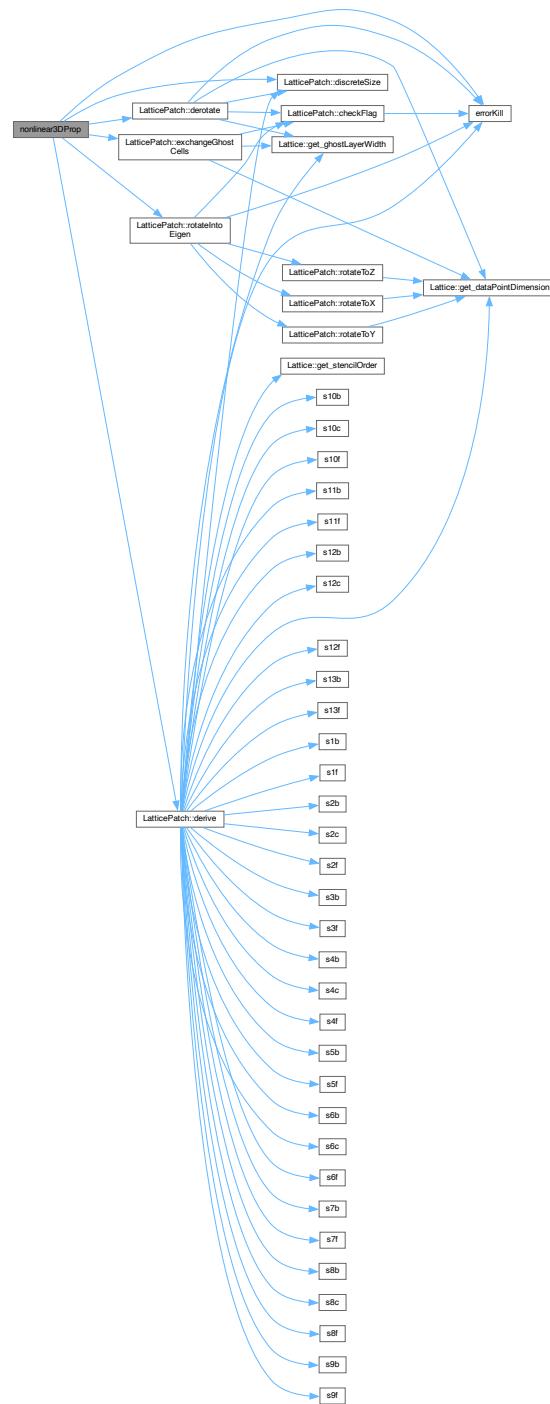
00697     JMM[15] = udata[2 + pp] * (lfg * udata[pp] + lgg * udata[3 + pp]) -
00698             (lff * udata[pp] + lfg * udata[3 + pp]) * udata[5 + pp];
00699     JMM[16] = udata[2 + pp] * (lfg * udata[1 + pp] + lgg * udata[4 + pp]) -
00700             (lff * udata[1 + pp] + lfg * udata[4 + pp]) * udata[5 + pp];
00701     JMM[17] = lg + lfg * (Quad[2] - Quad[5 + 0]) +
00702             (-lff + lgg) * udata[2 + pp] * udata[5 + pp];
00703     JMM[18] = lgg * udata[pp] * udata[2 + pp] +
00704             lff * udata[3 + pp] * udata[5 + pp] -
00705             lfg * (udata[2 + pp] * udata[3 + pp] + udata[pp] * udata[5 + pp]);
00706     JMM[19] =
00707         lgg * udata[1 + pp] * udata[2 + pp] +
00708         lff * udata[4 + pp] * udata[5 + pp] -
00709         lfg * (udata[2 + pp] * udata[4 + pp] + udata[1 + pp] * udata[5 + pp]);
00710     JMM[20] = -lf + lgg * Quad[2] +
00711             udata[5 + pp] * (-2 * lfg * udata[2 + pp] + lff * udata[5 + pp]);
00712
00713     h[0] = 0;
00714     h[1] = dxData[pp] * JMM[15] + dxData[1 + pp] * JMM[16] +
00715             dxData[2 + pp] * JMM[17] + dxData[3 + pp] * JMM[18] +
00716             dxData[4 + pp] * JMM[19] + dxData[5 + pp] * (-1 + JMM[20]);
00717     h[2] = -(dxData[pp] * JMM[10]) - dxData[1 + pp] * JMM[11] -
00718             dxData[2 + pp] * JMM[12] - dxData[3 + pp] * JMM[13] +
00719             dxData[4 + pp] * (1 - JMM[14]) - dxData[5 + pp] * JMM[19];
00720     h[3] = 0;
00721     h[4] = dxData[2 + pp];
00722     h[5] = -dxData[1 + pp];
00723     h[0] += -(dyData[pp] * JMM[15]) - dyData[1 + pp] * JMM[16] -
00724             dyData[2 + pp] * JMM[17] - dyData[3 + pp] * JMM[18] -
00725             dyData[4 + pp] * JMM[19] + dyData[5 + pp] * (1 - JMM[20]);
00726     h[1] += 0;
00727     h[2] += dyData[pp] * JMM[6] + dyData[1 + pp] * JMM[7] +
00728             dyData[2 + pp] * JMM[8] + dyData[3 + pp] * (-1 + JMM[9]) +
00729             dyData[4 + pp] * JMM[13] + dyData[5 + pp] * JMM[18];
00730     h[3] += -dyData[2 + pp];
00731     h[4] += 0;
00732     h[5] += dyData[pp];
00733     h[0] += dzData[pp] * JMM[10] + dzData[1 + pp] * JMM[11] +
00734             dzData[2 + pp] * JMM[12] + dzData[3 + pp] * JMM[13] +
00735             dzData[4 + pp] * (-1 + JMM[14]) + dzData[5 + pp] * JMM[19];
00736     h[1] += -(dzData[pp] * JMM[6]) - dzData[1 + pp] * JMM[7] -
00737             dzData[2 + pp] * JMM[8] + dzData[3 + pp] * (1 - JMM[9]) -
00738             dzData[4 + pp] * JMM[13] - dzData[5 + pp] * JMM[18];
00739     h[2] += 0;
00740     h[3] += dzData[1 + pp];
00741     h[4] += -dzData[pp];
00742     h[5] += 0;
00743     h[0] -= h[3] * JMM[6] + h[4] * JMM[10] + h[5] * JMM[15];
00744     h[1] -= h[3] * JMM[7] + h[4] * JMM[11] + h[5] * JMM[16];
00745     h[2] -= h[3] * JMM[8] + h[4] * JMM[12] + h[5] * JMM[17];
00746     dudata[pp + 0] =
00747         h[2] * (-JMM[3] * (1 + JMM[2])) + JMM[1] * JMM[4] +
00748         h[1] * (JMM[3] * JMM[4] - JMM[1] * (1 + JMM[5])) +
00749         h[0] * (1 - JMM[4] * JMM[4] + JMM[5] + JMM[2] * (1 + JMM[5]));
00750     dudata[pp + 1] =
00751         h[2] * (JMM[3] * JMM[1] - (1 + JMM[0]) * JMM[4]) +
00752         h[1] * (1 - JMM[3] * JMM[3] + JMM[5] + JMM[0] * (1 + JMM[5])) +
00753         h[0] * (JMM[4] * JMM[3] - JMM[1] * (1 + JMM[5]));
00754     dudata[pp + 2] =
00755         h[2] * (1 - JMM[1] * JMM[1] + JMM[2] + JMM[0] * (1 + JMM[2])) +
00756         h[1] * (JMM[1] * JMM[3] - (1 + JMM[0]) * JMM[4]) +
00757         h[0] * ((-1 + JMM[2]) * JMM[3] + JMM[1] * JMM[4]);
00758     detC =
00759         -((1 + JMM[2]) * (-1 + JMM[3] * JMM[3])) +
00760         (JMM[3] * JMM[1] - JMM[4]) * JMM[4] + JMM[5] + JMM[2] * JMM[5] +
00761         JMM[0] * (1 + JMM[2]) - JMM[4] * JMM[4] + (1 + JMM[2]) * JMM[5]) -
00762         JMM[1] * ((-JMM[4] * JMM[3]) + JMM[1] * (1 + JMM[5]));
00763     dudata[pp + 0] /= detC;
00764     dudata[pp + 1] /= detC;
00765     dudata[pp + 2] /= detC;
00766     dudata[pp + 3] = h[3];
00767     dudata[pp + 4] = h[4];
00768     dudata[pp + 5] = h[5];
00769 }
00770 return;
00771 }

```

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, 61

~Simulation
 Simulation, 120

A1
 Gauss2D, 20
 Gauss3D, 25

A2
 Gauss2D, 20
 Gauss3D, 26

add
 ICSetter, 36

addGauss1D
 ICSetter, 37

addGauss2D
 ICSetter, 37

addGauss3D
 ICSetter, 38

addInitialConditions
 Simulation, 120

addPeriodicCLayerInX
 Simulation, 121

addPeriodicCLayerInXY
 Simulation, 122

addPlaneWave1D
 ICSetter, 39

addPlaneWave2D
 ICSetter, 39

addPlaneWave3D
 ICSetter, 40

addToSpace
 Gauss1D, 13
 Gauss2D, 19
 Gauss3D, 25
 PlaneWave1D, 110
 PlaneWave2D, 113
 PlaneWave3D, 116

advanceToTime
 Simulation, 123

Amp
 Gauss2D, 20
 Gauss3D, 26

amp
 gaussian2D, 31
 gaussian3D, 33

axis
 Gauss2D, 20
 Gauss3D, 26
 gaussian2D, 31

 gaussian3D, 33

buffData
 LatticePatch, 84

BuffersInitialized
 LatticePatch.h, 210

buffX
 LatticePatch, 84

buffY
 LatticePatch, 84

buffZ
 LatticePatch, 84

c
 TimeEvolution, 139

check_retval
 LatticePatch.cpp, 190
 LatticePatch.h, 208

checkFlag
 LatticePatch, 62
 Simulation, 124

checkNoFlag
 Simulation, 126

comm
 Lattice, 52

cvode_mem
 Simulation, 135

CvodeObjectSetUp
 SimulationClass.h, 235

dataPointDimension
 Lattice, 52

DerivationStencils.h
 s10b, 144
 s10c, 145
 s10f, 146
 s11b, 147
 s11f, 148
 s12b, 149
 s12c, 150
 s12f, 151
 s13b, 152
 s13f, 153
 s1b, 154
 s1f, 155
 s2b, 156
 s2c, 157
 s2f, 158
 s3b, 159
 s3f, 160

s4b, 161
 s4c, 162
 s4f, 163
 s5b, 164
 s5f, 165
 s6b, 166
 s6c, 167
 s6f, 168
 s7b, 169
 s7f, 170
 s8b, 171
 s8c, 172
 s8f, 173
 s9b, 174
 s9f, 175
 derive
 LatticePatch, 63
 derotate
 LatticePatch, 68
 dis
 Gauss2D, 21
 Gauss3D, 26
 discreteSize
 LatticePatch, 70
 du
 LatticePatch, 85
 duData
 LatticePatch, 85
 duLocal
 LatticePatch, 85
 dx
 Lattice, 52
 LatticePatch, 85
 dy
 Lattice, 53
 LatticePatch, 86
 dz
 Lattice, 53
 LatticePatch, 86
 envelopeLattice
 LatticePatch, 86
 errorKill
 LatticePatch.cpp, 191
 LatticePatch.h, 208
 eval
 ICSetter, 40
 exchangeGhostCells
 LatticePatch, 71
 f
 TimeEvolution, 138
 FLatticeDimensionSet
 LatticePatch.h, 210
 FLatticePatchSetUp
 LatticePatch.h, 210
 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, 42
 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, 42
 gaussian1D, 28
 k, 29
 p, 29
 phi, 29
 phig, 29
 x0, 30
 gaussian2D, 30
 amp, 31
 axis, 31
 ph0, 31

phA, 31
phip, 31
w0, 32
x0, 32
zr, 32
gaussian3D, 33
amp, 33
axis, 33
ph0, 33
phA, 34
phip, 34
w0, 34
x0, 34
zr, 35
gCLData
 LatticePatch, 86
gCRData
 LatticePatch, 87
generateOutputFolder
 OutputManager, 97
generatePatchwork
 LatticePatch, 82
 LatticePatch.cpp, 192
generateTranslocationLookup
 LatticePatch, 73
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, 47
get_tot_ly
 Lattice, 48
get_tot_lz
 Lattice, 48
get_tot_noDP
 Lattice, 49
get_tot_noP
 Lattice, 49
get_tot_nx
 Lattice, 49
get_tot_ny
 Lattice, 50
get_tot_nz
 Lattice, 50
getDelta
 LatticePatch, 75
getSimCode
 OutputManager, 98
ghostCellLeft
 LatticePatch, 87
ghostCellLeftToSend
 LatticePatch, 87
ghostCellRight
 LatticePatch, 87
ghostCellRightToSend
 LatticePatch, 88
ghostCells
 LatticePatch, 88
ghostCellsToSend
 LatticePatch, 88
GhostLayersInitialized
 LatticePatch.h, 210
ghostLayerWidth
 Lattice, 53
ICSetter, 35
 add, 36
 addGauss1D, 37
 addGauss2D, 37
 addGauss3D, 38
 addPlaneWave1D, 39
 addPlaneWave2D, 39
 addPlaneWave3D, 40
 eval, 40
 gauss1Ds, 41
 gauss2Ds, 42
 gauss3Ds, 42
 planeWaves1D, 42
 planeWaves2D, 42
 planeWaves3D, 43
icsettings
 Simulation, 135
ID
 LatticePatch, 88
initializeBuffers
 LatticePatch, 76
initializeCVODEobject
 Simulation, 127
initializePatchwork
 Simulation, 129
k
 gaussian1D, 29
 planewave, 107
kx
 Gauss1D, 14
 PlaneWave, 104
ky
 Gauss1D, 14
 PlaneWave, 104
kz
 Gauss1D, 14
 PlaneWave, 105
lambda
 Gauss2D, 21
 Gauss3D, 27
 Lattice, 43

comm, 52
 dataPointDimension, 52
 dx, 52
 dy, 53
 dz, 53
 get_dataPointDimension, 45
 get_dx, 45
 get_dy, 46
 get_dz, 46
 get_ghostLayerWidth, 46
 get_stencilOrder, 47
 get_tot_lx, 47
 get_tot_ly, 48
 get_tot_lz, 48
 get_tot_noDP, 49
 get_tot_noP, 49
 get_tot_nx, 49
 get_tot_ny, 50
 get_tot_nz, 50
 ghostLayerWidth, 53
 Lattice, 45
 my_prc, 53
 n_prc, 54
 setDiscreteDimensions, 50
 setPhysicalDimensions, 51
 statusFlags, 54
 stencilOrder, 54
 sunctx, 54
 tot_lx, 55
 tot_ly, 55
 tot_lz, 55
 tot_noDP, 55
 tot_noP, 56
 tot_nx, 56
 tot_ny, 56
 tot_nz, 56
 lattice
 Simulation, 136
 LatticeDiscreteSetUp
 SimulationClass.h, 236
 LatticePatch, 57
 ~LatticePatch, 61
 buffData, 84
 buffX, 84
 buffY, 84
 buffZ, 84
 checkFlag, 62
 derive, 63
 derotate, 68
 discreteSize, 70
 du, 85
 duData, 85
 duLocal, 85
 dx, 85
 dy, 86
 dz, 86
 envelopeLattice, 86
 exchangeGhostCells, 71
 gCLData, 86
 gCRData, 87
 generatePatchwork, 82
 generateTranslocationLookup, 73
 getDelta, 75
 ghostCellLeft, 87
 ghostCellLeftToSend, 87
 ghostCellRight, 87
 ghostCellRightToSend, 88
 ghostCells, 88
 ghostCellsToSend, 88
 ID, 88
 initializeBuffers, 76
 LatticePatch, 60
 lgcTox, 88
 lgcToy, 89
 lgcToz, 89
 lIx, 89
 lLy, 89
 lLz, 90
 lx, 90
 ly, 90
 lz, 90
 nx, 91
 ny, 91
 nz, 91
 origin, 76
 rgcTox, 91
 rgcToy, 92
 rgcToz, 92
 rotateIntoEigen, 77
 rotateToX, 79
 rotateToY, 80
 rotateToZ, 81
 statusFlags, 92
 u, 92
 uAux, 93
 uAuxData, 93
 uData, 93
 uLocal, 93
 uTox, 94
 uToy, 94
 uToz, 94
 x0, 94
 xTou, 95
 y0, 95
 yTou, 95
 z0, 95
 zTou, 96
 latticePatch
 Simulation, 136
 LatticePatch.cpp
 check_retval, 190
 errorKill, 191
 generatePatchwork, 192
 LatticePatch.h
 BuffersInitialized, 210
 check_retval, 208

errorKill, 208
FLatticeDimensionSet, 210
FLatticePatchSetUp, 210
GhostLayersInitialized, 210
TranslocationLookupSetUp, 210
LatticePatchworkSetUp
 SimulationClass.h, 236
LatticePhysicalSetUp
 SimulationClass.h, 236
lgcTox
 LatticePatch, 88
lgcToy
 LatticePatch, 89
lgcToz
 LatticePatch, 89
linear1DProp
 TimeEvolutionFunctions.cpp, 265
 TimeEvolutionFunctions.h, 294
linear2DProp
 TimeEvolutionFunctions.cpp, 267
 TimeEvolutionFunctions.h, 296
linear3DProp
 TimeEvolutionFunctions.cpp, 269
 TimeEvolutionFunctions.h, 298
lIx
 LatticePatch, 89
lLy
 LatticePatch, 89
lLz
 LatticePatch, 90
lx
 LatticePatch, 90
ly
 LatticePatch, 90
lz
 LatticePatch, 90
main
 main.cpp, 215
main.cpp
 main, 215
my_prc
 Lattice, 53
n_prc
 Lattice, 54
NLS
 Simulation, 136
nonlinear1DProp
 TimeEvolutionFunctions.cpp, 271
 TimeEvolutionFunctions.h, 300
nonlinear2DProp
 TimeEvolutionFunctions.cpp, 276
 TimeEvolutionFunctions.h, 305
nonlinear3DProp
 TimeEvolutionFunctions.cpp, 280
 TimeEvolutionFunctions.h, 309
nx
 LatticePatch, 91
ny
 LatticePatch, 91
nz
 LatticePatch, 91
origin
 LatticePatch, 76
outAllFieldData
 Simulation, 130
OutputManager, 96
 generateOutputFolder, 97
 getSimCode, 98
 OutputManager, 97
 outputStyle, 102
 outUState, 99
 Path, 102
 set_outputStyle, 101
 simCode, 103
 SimCodeGenerator, 101
outputManager
 Simulation, 136
outputStyle
 OutputManager, 102
outUState
 OutputManager, 99
p
 gaussian1D, 29
 planewave, 107
Path
 OutputManager, 102
Ph0
 Gauss2D, 21
 Gauss3D, 27
ph0
 gaussian2D, 31
 gaussian3D, 33
PhA
 Gauss2D, 21
 Gauss3D, 27
phA
 gaussian2D, 31
 gaussian3D, 34
phi
 gaussian1D, 29
 planewave, 107
phig
 Gauss1D, 14
 gaussian1D, 29
phip
 Gauss2D, 22
 Gauss3D, 27
 gaussian2D, 31
 gaussian3D, 34
phix
 Gauss1D, 14
 PlaneWave, 105
phiy
 Gauss1D, 15

PlaneWave, 105
phiz
 Gauss1D, 15
 PlaneWave, 105
PlaneWave, 103
 kx, 104
 ky, 104
 kz, 105
 phix, 105
 phiy, 105
 phiz, 105
 px, 106
 py, 106
 pz, 106
planewave, 107
 k, 107
 p, 107
 phi, 107
PlaneWave1D, 108
 addToSpace, 110
 PlaneWave1D, 109
PlaneWave2D, 111
 addToSpace, 113
 PlaneWave2D, 113
PlaneWave3D, 114
 addToSpace, 116
 PlaneWave3D, 116
planeWaves1D
 ICSetter, 42
planeWaves2D
 ICSetter, 42
planeWaves3D
 ICSetter, 43
px
 Gauss1D, 15
 PlaneWave, 106
py
 Gauss1D, 15
 PlaneWave, 106
pz
 Gauss1D, 16
 PlaneWave, 106
README.md, 141
rgcTox
 LatticePatch, 91
rgcToy
 LatticePatch, 92
rgcToz
 LatticePatch, 92
rotateIntoEigen
 LatticePatch, 77
rotateToX
 LatticePatch, 79
rotateToY
 LatticePatch, 80
rotateToZ
 LatticePatch, 81
s10b
 DerivationStencils.h, 144
s10c
 DerivationStencils.h, 145
s10f
 DerivationStencils.h, 146
s11b
 DerivationStencils.h, 147
s11f
 DerivationStencils.h, 148
s12b
 DerivationStencils.h, 149
s12c
 DerivationStencils.h, 150
s12f
 DerivationStencils.h, 151
s13b
 DerivationStencils.h, 152
s13f
 DerivationStencils.h, 153
s1b
 DerivationStencils.h, 154
s1f
 DerivationStencils.h, 155
s2b
 DerivationStencils.h, 156
s2c
 DerivationStencils.h, 157
s2f
 DerivationStencils.h, 158
s3b
 DerivationStencils.h, 159
s3f
 DerivationStencils.h, 160
s4b
 DerivationStencils.h, 161
s4c
 DerivationStencils.h, 162
s4f
 DerivationStencils.h, 163
s5b
 DerivationStencils.h, 164
s5f
 DerivationStencils.h, 165
s6b
 DerivationStencils.h, 166
s6c
 DerivationStencils.h, 167
s6f
 DerivationStencils.h, 168
s7b
 DerivationStencils.h, 169
s7f
 DerivationStencils.h, 170
s8b
 DerivationStencils.h, 171
s8c
 DerivationStencils.h, 172

s8f
 DerivationStencils.h, 173

s9b
 DerivationStencils.h, 174

s9f
 DerivationStencils.h, 175

set_outputStyle
 OutputManager, 101

setDiscreteDimensions
 Lattice, 50

setDiscreteDimensionsOfLattice
 Simulation, 131

setInitialConditions
 Simulation, 132

setPhysicalDimensions
 Lattice, 51

setPhysicalDimensionsOfLattice
 Simulation, 133

Sim1D
 SimulationFunctions.cpp, 239
 SimulationFunctions.h, 254

Sim2D
 SimulationFunctions.cpp, 242
 SimulationFunctions.h, 257

Sim3D
 SimulationFunctions.cpp, 245
 SimulationFunctions.h, 260

simCode
 OutputManager, 103

SimCodeGenerator
 OutputManager, 101

Simulation, 117
 ~Simulation, 120
 addInitialConditions, 120
 addPeriodicCLayerInX, 121
 addPeriodicCLayerInXY, 122
 advanceToTime, 123
 checkFlag, 124
 checkNoFlag, 126
 cvode_mem, 135
 icsettings, 135
 initializeCVODEobject, 127
 initializePatchwork, 129
 lattice, 136
 latticePatch, 136
 NLS, 136
 outAllFieldData, 130
 outputManager, 136
 setDiscreteDimensionsOfLattice, 131
 setInitialConditions, 132
 setPhysicalDimensionsOfLattice, 133
 Simulation, 119
 start, 134
 statusFlags, 137
 t, 137

SimulationClass.h
 CvodeObjectSetUp, 235
 LatticeDiscreteSetUp, 236

LatticePatchworkSetUp, 236
LatticePhysicalSetUp, 236
SimulationStarted, 236

SimulationFunctions.cpp
 Sim1D, 239
 Sim2D, 242
 Sim3D, 245
 timer, 248

SimulationFunctions.h
 Sim1D, 254
 Sim2D, 257
 Sim3D, 260
 timer, 263

SimulationStarted
 SimulationClass.h, 236

src/DerivationStencils.cpp, 141, 142

src/DerivationStencils.h, 142, 176

src/ICSetters.cpp, 180

src/ICSetters.h, 185, 186

src/LatticePatch.cpp, 189, 194

src/LatticePatch.h, 206, 211

src/main.cpp, 214, 222

src/Outputters.cpp, 226

src/Outputters.h, 228, 229

src/SimulationClass.cpp, 230

src/SimulationClass.h, 234, 237

src/SimulationFunctions.cpp, 238, 249

src/SimulationFunctions.h, 252, 264

src/TimeEvolutionFunctions.cpp, 265, 284

src/TimeEvolutionFunctions.h, 293, 313

start
 Simulation, 134

statusFlags
 Lattice, 54
 LatticePatch, 92
 Simulation, 137

stencilOrder
 Lattice, 54

sunctx
 Lattice, 54

t
 Simulation, 137

TimeEvolution, 137
 c, 139
 f, 138
 TimeEvolver, 139

TimeEvolutionFunctions.cpp
 linear1DProp, 265
 linear2DProp, 267
 linear3DProp, 269
 nonlinear1DProp, 271
 nonlinear2DProp, 276
 nonlinear3DProp, 280

TimeEvolutionFunctions.h
 linear1DProp, 294
 linear2DProp, 296
 linear3DProp, 298
 nonlinear1DProp, 300

nonlinear2DProp, 305
 nonlinear3DProp, 309
TimeEvolver
 TimeEvolution, 139
timer
 SimulationFunctions.cpp, 248
 SimulationFunctions.h, 263
tot_lx
 Lattice, 55
tot_ly
 Lattice, 55
tot_lz
 Lattice, 55
tot_noDP
 Lattice, 55
tot_noP
 Lattice, 56
tot_nx
 Lattice, 56
tot_ny
 Lattice, 56
tot_nz
 Lattice, 56
TranslocationLookupSetUp
 LatticePatch.h, 210

u
 LatticePatch, 92
uAux
 LatticePatch, 93
uAuxData
 LatticePatch, 93
uData
 LatticePatch, 93
uLocal
 LatticePatch, 93
uTox
 LatticePatch, 94
uToy
 LatticePatch, 94
uToz
 LatticePatch, 94

w0
 Gauss2D, 22
 Gauss3D, 28
 gaussian2D, 32
 gaussian3D, 34

x0
 gaussian1D, 30
 gaussian2D, 32
 gaussian3D, 34
 LatticePatch, 94

x0x
 Gauss1D, 16

x0y
 Gauss1D, 16

x0z

Gauss1D, 16
xTou
 LatticePatch, 95

y0
 LatticePatch, 95

yTou
 LatticePatch, 95

z0
 LatticePatch, 95

zr
 Gauss2D, 22
 Gauss3D, 28
 gaussian2D, 32
 gaussian3D, 35

zTou
 LatticePatch, 96