

HEWES : Heisenberg-Euler Weak-Field Expansion Simulator

Generated by Doxygen 1.9.3

1 HEWES – Heisenberg-Euler Weak-Field Expansion Simulator	1
1.1 Contents	1
1.2 Preparing the Makefile	2
1.3 Short User Manual	2
1.3.1 Note on Simulation Settings	3
1.3.2 Note on Resource Occupation	4
1.3.3 Note on Output Analysis	4
1.4 Authors	5
2 Hierarchical Index	7
2.1 Class Hierarchy	7
3 Data Structure Index	9
3.1 Data Structures	9
4 File Index	11
4.1 File List	11
5 Data Structure Documentation	13
5.1 Gauss1D Class Reference	13
5.1.1 Detailed Description	14
5.1.2 Constructor & Destructor Documentation	14
5.1.2.1 Gauss1D()	14
5.1.3 Member Function Documentation	15
5.1.3.1 addToSpace()	15
5.1.4 Field Documentation	16
5.1.4.1 kx	16
5.1.4.2 ky	16
5.1.4.3 kz	16
5.1.4.4 phig	16
5.1.4.5 phix	17
5.1.4.6 phiy	17
5.1.4.7 phiz	17
5.1.4.8 px	17
5.1.4.9 py	18
5.1.4.10 pz	18
5.1.4.11 x0x	18
5.1.4.12 x0y	18
5.1.4.13 x0z	19
5.2 Gauss2D Class Reference	19
5.2.1 Detailed Description	20
5.2.2 Constructor & Destructor Documentation	20
5.2.2.1 Gauss2D()	20
5.2.3 Member Function Documentation	21

5.2.3.1 addToSpace()	21
5.2.4 Field Documentation	22
5.2.4.1 A1	22
5.2.4.2 A2	22
5.2.4.3 Amp	22
5.2.4.4 axis	23
5.2.4.5 dis	23
5.2.4.6 lambda	23
5.2.4.7 Ph0	23
5.2.4.8 PhA	24
5.2.4.9 phip	24
5.2.4.10 w0	24
5.2.4.11 zr	24
5.3 Gauss3D Class Reference	25
5.3.1 Detailed Description	26
5.3.2 Constructor & Destructor Documentation	26
5.3.2.1 Gauss3D()	26
5.3.3 Member Function Documentation	27
5.3.3.1 addToSpace()	27
5.3.4 Field Documentation	27
5.3.4.1 A1	28
5.3.4.2 A2	28
5.3.4.3 Amp	28
5.3.4.4 axis	28
5.3.4.5 dis	29
5.3.4.6 lambda	29
5.3.4.7 Ph0	29
5.3.4.8 PhA	29
5.3.4.9 phip	30
5.3.4.10 w0	30
5.3.4.11 zr	30
5.4 gaussian1D Struct Reference	30
5.4.1 Detailed Description	31
5.4.2 Field Documentation	31
5.4.2.1 k	31
5.4.2.2 p	31
5.4.2.3 phi	31
5.4.2.4 phig	32
5.4.2.5 x0	32
5.5 gaussian2D Struct Reference	32
5.5.1 Detailed Description	32
5.5.2 Field Documentation	33

5.5.2.1 amp	33
5.5.2.2 axis	33
5.5.2.3 ph0	33
5.5.2.4 phA	33
5.5.2.5 phip	33
5.5.2.6 w0	34
5.5.2.7 x0	34
5.5.2.8 zr	34
5.6 gaussian3D Struct Reference	34
5.6.1 Detailed Description	35
5.6.2 Field Documentation	35
5.6.2.1 amp	35
5.6.2.2 axis	35
5.6.2.3 ph0	35
5.6.2.4 phA	35
5.6.2.5 phip	36
5.6.2.6 w0	36
5.6.2.7 x0	36
5.6.2.8 zr	36
5.7 ICSetter Class Reference	36
5.7.1 Detailed Description	37
5.7.2 Member Function Documentation	37
5.7.2.1 add()	38
5.7.2.2 addGauss1D()	38
5.7.2.3 addGauss2D()	39
5.7.2.4 addGauss3D()	40
5.7.2.5 addPlaneWave1D()	40
5.7.2.6 addPlaneWave2D()	41
5.7.2.7 addPlaneWave3D()	42
5.7.2.8 eval()	42
5.7.3 Field Documentation	43
5.7.3.1 gauss1Ds	43
5.7.3.2 gauss2Ds	43
5.7.3.3 gauss3Ds	44
5.7.3.4 planeWaves1D	44
5.7.3.5 planeWaves2D	44
5.7.3.6 planeWaves3D	44
5.8 Lattice Class Reference	45
5.8.1 Detailed Description	46
5.8.2 Constructor & Destructor Documentation	46
5.8.2.1 Lattice()	47
5.8.3 Member Function Documentation	47

5.8.3.1 <code>get_dataPointDimension()</code>	47
5.8.3.2 <code>get_dx()</code>	48
5.8.3.3 <code>get_dy()</code>	48
5.8.3.4 <code>get_dz()</code>	48
5.8.3.5 <code>get_ghostLayerWidth()</code>	49
5.8.3.6 <code>get_stencilOrder()</code>	49
5.8.3.7 <code>get_tot_lx()</code>	50
5.8.3.8 <code>get_tot_ly()</code>	50
5.8.3.9 <code>get_tot_lz()</code>	51
5.8.3.10 <code>get_tot_noDP()</code>	51
5.8.3.11 <code>get_tot_noP()</code>	51
5.8.3.12 <code>get_tot_nx()</code>	52
5.8.3.13 <code>get_tot_ny()</code>	52
5.8.3.14 <code>get_tot_nz()</code>	52
5.8.3.15 <code>initializeCommunicator()</code>	53
5.8.3.16 <code>setDiscreteDimensions()</code>	53
5.8.3.17 <code>setPhysicalDimensions()</code>	54
5.8.4 Field Documentation	55
5.8.4.1 <code>comm</code>	55
5.8.4.2 <code>dataPointDimension</code>	55
5.8.4.3 <code>dx</code>	55
5.8.4.4 <code>dy</code>	56
5.8.4.5 <code>dz</code>	56
5.8.4.6 <code>ghostLayerWidth</code>	56
5.8.4.7 <code>my_prc</code>	56
5.8.4.8 <code>n_prc</code>	57
5.8.4.9 <code>profobj</code>	57
5.8.4.10 <code>statusFlags</code>	57
5.8.4.11 <code>stencilOrder</code>	57
5.8.4.12 <code>sunctx</code>	58
5.8.4.13 <code>tot_lx</code>	58
5.8.4.14 <code>tot_ly</code>	58
5.8.4.15 <code>tot_lz</code>	58
5.8.4.16 <code>tot_noDP</code>	59
5.8.4.17 <code>tot_noP</code>	59
5.8.4.18 <code>tot_nx</code>	59
5.8.4.19 <code>tot_ny</code>	59
5.8.4.20 <code>tot_nz</code>	60
5.9 LatticePatch Class Reference	60
5.9.1 Detailed Description	63
5.9.2 Constructor & Destructor Documentation	63
5.9.2.1 <code>LatticePatch()</code>	64

5.9.2.2 <code>~LatticePatch()</code>	64
5.9.3 Member Function Documentation	64
5.9.3.1 <code>checkFlag()</code>	65
5.9.3.2 <code>derive()</code>	66
5.9.3.3 <code>derotate()</code>	70
5.9.3.4 <code>discreteSize()</code>	72
5.9.3.5 <code>exchangeGhostCells()</code>	73
5.9.3.6 <code>generateTranslocationLookup()</code>	75
5.9.3.7 <code>getDelta()</code>	76
5.9.3.8 <code>initializeBuffers()</code>	77
5.9.3.9 <code>initializeGhostLayer()</code>	78
5.9.3.10 <code>origin()</code>	79
5.9.3.11 <code>rotateIntoEigen()</code>	80
5.9.3.12 <code>rotateToX()</code>	81
5.9.3.13 <code>rotateToY()</code>	82
5.9.3.14 <code>rotateToZ()</code>	83
5.9.4 Friends And Related Function Documentation	84
5.9.4.1 <code>generatePatchwork</code>	84
5.9.5 Field Documentation	85
5.9.5.1 <code>buffData</code>	85
5.9.5.2 <code>buffX</code>	86
5.9.5.3 <code>buffY</code>	86
5.9.5.4 <code>buffZ</code>	86
5.9.5.5 <code>du</code>	86
5.9.5.6 <code>duData</code>	87
5.9.5.7 <code>dx</code>	87
5.9.5.8 <code>dy</code>	87
5.9.5.9 <code>dz</code>	87
5.9.5.10 <code>envelopeLattice</code>	88
5.9.5.11 <code>gCLData</code>	88
5.9.5.12 <code>gCRData</code>	88
5.9.5.13 <code>ghostCellLeft</code>	88
5.9.5.14 <code>ghostCellLeftToSend</code>	89
5.9.5.15 <code>ghostCellRight</code>	89
5.9.5.16 <code>ghostCellRightToSend</code>	89
5.9.5.17 <code>ghostCells</code>	89
5.9.5.18 <code>ghostCellsToSend</code>	89
5.9.5.19 <code>ID</code>	90
5.9.5.20 <code>lgcTox</code>	90
5.9.5.21 <code>lgcToy</code>	90
5.9.5.22 <code>lgcToz</code>	90
5.9.5.23 <code>Llx</code>	91

5.9.5.24 Lly	91
5.9.5.25 Llz	91
5.9.5.26 lx	91
5.9.5.27 ly	92
5.9.5.28 lz	92
5.9.5.29 nx	92
5.9.5.30 ny	92
5.9.5.31 nz	93
5.9.5.32 rgcTox	93
5.9.5.33 rgcToy	93
5.9.5.34 rgcToz	93
5.9.5.35 statusFlags	94
5.9.5.36 u	94
5.9.5.37 uAux	94
5.9.5.38 uAuxData	94
5.9.5.39 uData	95
5.9.5.40 uTox	95
5.9.5.41 uToy	95
5.9.5.42 uToz	95
5.9.5.43 x0	96
5.9.5.44 xTou	96
5.9.5.45 y0	96
5.9.5.46 yTou	96
5.9.5.47 z0	97
5.9.5.48 zTou	97
5.10 OutputManager Class Reference	97
5.10.1 Detailed Description	98
5.10.2 Constructor & Destructor Documentation	98
5.10.2.1 OutputManager()	98
5.10.3 Member Function Documentation	98
5.10.3.1 generateOutputFolder()	99
5.10.3.2 getSimCode()	100
5.10.3.3 outUState()	100
5.10.3.4 SimCodeGenerator()	101
5.10.4 Field Documentation	102
5.10.4.1 myPrc	102
5.10.4.2 Path	102
5.10.4.3 simCode	102
5.11 PlaneWave Class Reference	103
5.11.1 Detailed Description	103
5.11.2 Field Documentation	104
5.11.2.1 kx	104

5.11.2.2 ky	104
5.11.2.3 kz	104
5.11.2.4 phix	104
5.11.2.5 phiy	105
5.11.2.6 phiz	105
5.11.2.7 px	105
5.11.2.8 py	105
5.11.2.9 pz	106
5.12 planewave Struct Reference	106
5.12.1 Detailed Description	106
5.12.2 Field Documentation	106
5.12.2.1 k	106
5.12.2.2 p	107
5.12.2.3 phi	107
5.13 PlaneWave1D Class Reference	107
5.13.1 Detailed Description	108
5.13.2 Constructor & Destructor Documentation	108
5.13.2.1 PlaneWave1D()	108
5.13.3 Member Function Documentation	109
5.13.3.1 addToSpace()	109
5.14 PlaneWave2D Class Reference	110
5.14.1 Detailed Description	111
5.14.2 Constructor & Destructor Documentation	111
5.14.2.1 PlaneWave2D()	111
5.14.3 Member Function Documentation	112
5.14.3.1 addToSpace()	112
5.15 PlaneWave3D Class Reference	112
5.15.1 Detailed Description	113
5.15.2 Constructor & Destructor Documentation	113
5.15.2.1 PlaneWave3D()	114
5.15.3 Member Function Documentation	114
5.15.3.1 addToSpace()	115
5.16 Simulation Class Reference	115
5.16.1 Detailed Description	117
5.16.2 Constructor & Destructor Documentation	117
5.16.2.1 Simulation()	117
5.16.2.2 ~Simulation()	118
5.16.3 Member Function Documentation	118
5.16.3.1 addInitialConditions()	118
5.16.3.2 addPeriodicICLayerInX()	120
5.16.3.3 addPeriodicICLayerInXY()	121
5.16.3.4 advanceToTime()	122

5.16.3.5 checkFlag()	123
5.16.3.6 checkNoFlag()	124
5.16.3.7 get_cart_comm()	125
5.16.3.8 initializeCVODEobject()	125
5.16.3.9 initializePatchwork()	127
5.16.3.10 outAllFieldData()	128
5.16.3.11 setDiscreteDimensionsOfLattice()	129
5.16.3.12 setInitialConditions()	130
5.16.3.13 setPhysicalDimensionsOfLattice()	131
5.16.3.14 start()	132
5.16.4 Field Documentation	133
5.16.4.1 ccode_mem	133
5.16.4.2 icsettings	134
5.16.4.3 lattice	134
5.16.4.4 latticePatch	134
5.16.4.5 outputManager	134
5.16.4.6 statusFlags	135
5.16.4.7 t	135
5.17 TimeEvolution Class Reference	135
5.17.1 Detailed Description	136
5.17.2 Member Function Documentation	136
5.17.2.1 f()	136
5.17.3 Field Documentation	137
5.17.3.1 c	137
5.17.3.2 TimeEvolver	137
6 File Documentation	139
6.1 README.md File Reference	139
6.2 src/DerivationStencils.cpp File Reference	139
6.2.1 Detailed Description	139
6.3 DerivationStencils.cpp	140
6.4 src/DerivationStencils.h File Reference	140
6.4.1 Detailed Description	142
6.4.2 Function Documentation	142
6.4.2.1 s10b() [1/2]	142
6.4.2.2 s10b() [2/2]	143
6.4.2.3 s10c() [1/2]	143
6.4.2.4 s10c() [2/2]	144
6.4.2.5 s10f() [1/2]	144
6.4.2.6 s10f() [2/2]	145
6.4.2.7 s11b() [1/2]	145
6.4.2.8 s11b() [2/2]	146

6.4.2.9 s11f() [1/2]	146
6.4.2.10 s11f() [2/2]	147
6.4.2.11 s12b() [1/2]	147
6.4.2.12 s12b() [2/2]	148
6.4.2.13 s12c() [1/2]	149
6.4.2.14 s12c() [2/2]	149
6.4.2.15 s12f() [1/2]	150
6.4.2.16 s12f() [2/2]	150
6.4.2.17 s13b() [1/2]	151
6.4.2.18 s13b() [2/2]	151
6.4.2.19 s13f() [1/2]	152
6.4.2.20 s13f() [2/2]	152
6.4.2.21 s1b() [1/2]	153
6.4.2.22 s1b() [2/2]	153
6.4.2.23 s1f() [1/2]	154
6.4.2.24 s1f() [2/2]	154
6.4.2.25 s2b() [1/2]	155
6.4.2.26 s2b() [2/2]	155
6.4.2.27 s2c() [1/2]	156
6.4.2.28 s2c() [2/2]	156
6.4.2.29 s2f() [1/2]	157
6.4.2.30 s2f() [2/2]	157
6.4.2.31 s3b() [1/2]	158
6.4.2.32 s3b() [2/2]	158
6.4.2.33 s3f() [1/2]	159
6.4.2.34 s3f() [2/2]	159
6.4.2.35 s4b() [1/2]	160
6.4.2.36 s4b() [2/2]	160
6.4.2.37 s4c() [1/2]	161
6.4.2.38 s4c() [2/2]	161
6.4.2.39 s4f() [1/2]	162
6.4.2.40 s4f() [2/2]	162
6.4.2.41 s5b() [1/2]	163
6.4.2.42 s5b() [2/2]	163
6.4.2.43 s5f() [1/2]	164
6.4.2.44 s5f() [2/2]	164
6.4.2.45 s6b() [1/2]	165
6.4.2.46 s6b() [2/2]	165
6.4.2.47 s6c() [1/2]	166
6.4.2.48 s6c() [2/2]	166
6.4.2.49 s6f() [1/2]	167
6.4.2.50 s6f() [2/2]	167

6.4.2.51 s7b() [1/2]	168
6.4.2.52 s7b() [2/2]	168
6.4.2.53 s7f() [1/2]	169
6.4.2.54 s7f() [2/2]	169
6.4.2.55 s8b() [1/2]	170
6.4.2.56 s8b() [2/2]	170
6.4.2.57 s8c() [1/2]	171
6.4.2.58 s8c() [2/2]	171
6.4.2.59 s8f() [1/2]	172
6.4.2.60 s8f() [2/2]	172
6.4.2.61 s9b() [1/2]	173
6.4.2.62 s9b() [2/2]	173
6.4.2.63 s9f() [1/2]	174
6.4.2.64 s9f() [2/2]	174
6.5 DerivationStencils.h	175
6.6 src/ICSetters.cpp File Reference	178
6.6.1 Detailed Description	178
6.7 ICSetters.cpp	178
6.8 src/ICSetters.h File Reference	182
6.8.1 Detailed Description	184
6.9 ICSetters.h	184
6.10 src/LatticePatch.cpp File Reference	187
6.10.1 Detailed Description	188
6.10.2 Function Documentation	188
6.10.2.1 check_retval()	188
6.10.2.2 errorKill()	189
6.10.2.3 generatePatchwork()	190
6.11 LatticePatch.cpp	192
6.12 src/LatticePatch.h File Reference	202
6.12.1 Detailed Description	203
6.12.2 Enumeration Type Documentation	203
6.12.2.1 LatticeOptions	203
6.12.2.2 LatticePatchOptions	204
6.12.3 Function Documentation	204
6.12.3.1 check_retval()	204
6.12.3.2 errorKill()	205
6.13 LatticePatch.h	206
6.14 src/main.cpp File Reference	209
6.14.1 Detailed Description	210
6.14.2 Function Documentation	210
6.14.2.1 main()	210
6.15 main.cpp	218

6.16 src/Outputters.cpp File Reference	221
6.16.1 Detailed Description	222
6.17 Outputters.cpp	222
6.18 src/Outputters.h File Reference	223
6.18.1 Detailed Description	224
6.19 Outputters.h	225
6.20 src/SimulationClass.cpp File Reference	225
6.20.1 Detailed Description	226
6.21 SimulationClass.cpp	226
6.22 src/SimulationClass.h File Reference	229
6.22.1 Detailed Description	230
6.22.2 Enumeration Type Documentation	231
6.22.2.1 SimulationOptions	231
6.23 SimulationClass.h	231
6.24 src/SimulationFunctions.cpp File Reference	232
6.24.1 Detailed Description	233
6.24.2 Function Documentation	233
6.24.2.1 Sim1D()	234
6.24.2.2 Sim2D()	237
6.24.2.3 Sim3D()	240
6.24.2.4 timer()	243
6.25 SimulationFunctions.cpp	244
6.26 src/SimulationFunctions.h File Reference	247
6.26.1 Detailed Description	248
6.26.2 Function Documentation	248
6.26.2.1 Sim1D()	248
6.26.2.2 Sim2D()	251
6.26.2.3 Sim3D()	254
6.26.2.4 timer()	257
6.27 SimulationFunctions.h	258
6.28 src/TimeEvolutionFunctions.cpp File Reference	259
6.28.1 Detailed Description	259
6.28.2 Function Documentation	259
6.28.2.1 linear1DProp()	260
6.28.2.2 linear2DProp()	261
6.28.2.3 linear3DProp()	263
6.28.2.4 nonlinear1DProp()	265
6.28.2.5 nonlinear2DProp()	270
6.28.2.6 nonlinear3DProp()	274
6.29 TimeEvolutionFunctions.cpp	278
6.30 src/TimeEvolutionFunctions.h File Reference	286
6.30.1 Detailed Description	287

6.30.2 Function Documentation	287
6.30.2.1 linear1DProp()	288
6.30.2.2 linear2DProp()	289
6.30.2.3 linear3DProp()	291
6.30.2.4 nonlinear1DProp()	293
6.30.2.5 nonlinear2DProp()	298
6.30.2.6 nonlinear3DProp()	302
6.31 TimeEvolutionFunctions.h	306
Index	307

Chapter 1

HEWES – Heisenberg-Euler Weak-Field Expansion Simulator

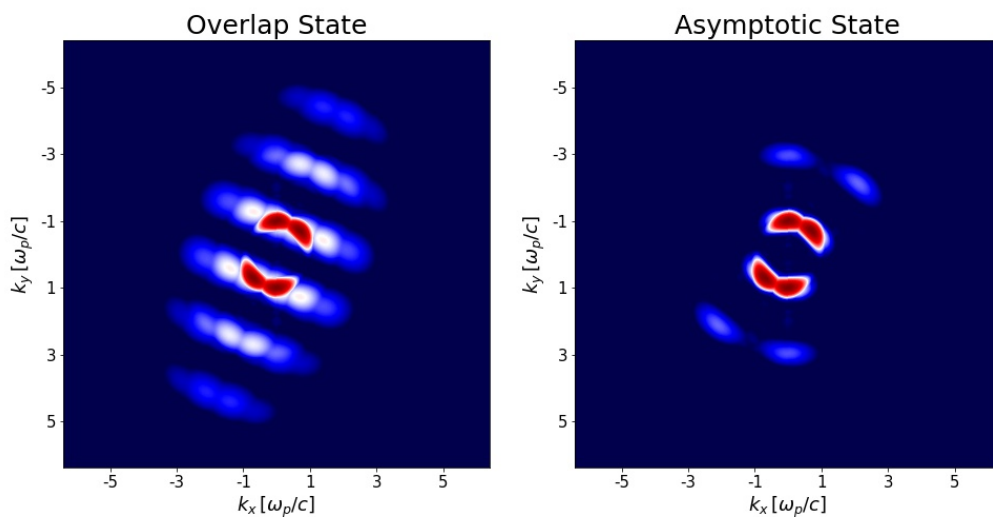


Figure 1.1 Harmonic Generation

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

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

1.1 Contents

- Preparing the Makefile

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

1.2 Preparing the Makefile

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

The `make` utility is used for building and a recent compiler version supporting OpenMP is required. Features up to the C++20 standard are used.

Additionally required software:

- An MPI implementation.
While Intel(R) MPI has mostly been used for scientific work on high-performance computing systems, the provided Makefile here is by default created for use with the *open* implementations `OpenMPI` or `MPICH`. While some useful Intel(R) processor specific optimizations and compiler options are not available with the latter, they are easier to get and set up on a personal device simply via the corresponding package manager.
- The `SUNDIALS` package with the `CVode` solver.
Version 6 is required. The code is presumably compliant with the upcoming version 7.
For the installation of `SUNDIALS`, `CMake` is required. Follow the installation guide and do not forget to enable MPI and specify the directory of the `mpicxx` wrapper for use of the MPI-based `NVECTOR_↔PARALLEL` module. Make sure to edit the `SUNDIALS` binary and library paths in the Makefile.

A minimal `Makefile` template is provided. Further compiler options might be beneficial, depending on the used system and software; e.g., higher vectorization and register usage instructions.

1.3 Short User Manual

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

- First, specify the path you want the output data to go via the variable `outputDirectory`.
- Second, decide if you want to simulate in 1D, 2D, or 3D and uncomment only that full section.
You can then specify
 - the relative and absolute integration tolerances of the `CVode` solver.
Recommended values are between $1e-12$ and $1e-18$.
 - the order of accuracy of the numerical scheme via the stencil order.
You can choose an integer in the range 1-13.
 - the physical side lengths of the grid in meters.
 - the number of lattice points per dimension.
 - the slicing of the lattice into patches (only for 2D and 3D simulations, automatic in 1D) – this determines the number of patches and therefore the required distinct processing units for MPI.
The total number of processes is given by the product of patches in any dimension.
Note: In the 3D case you better insure that every patch is cubic in terms of lattice points. This is decisive for computational efficiency.

- whether to have periodic or vanishing boundary values (currently has to be chosen periodic).
- whether you want to simulate on top of the linear vacuum only 4-photon processes (1), 6-photon processes (2), both (3), or none (0) – the linear Maxwell case.
- the total time of the simulation in units $c=1$, i.e., the distance propagated by the light waves in meters.
- the number of time steps that will be solved stepwise by CNode.
In order to keep interpolation errors small do not choose this number too small.
- the multiple of steps at which you want the data to be written to disk.
The name of the files written to the output directory is of the form `{step_number}_{process_number}`.
- which electromagnetic waveform(s) you want to propagate.
You can choose between a plane wave (not much physical content, but useful for checks) and implementations of Gaussians in 1D, 2D, and 3D. Their parameters can be tuned.
A description of the wave implementations is given in [ref.pdf](#). Note that the 3D Gaussians, as they are implemented up to now, should be propagated in the xy-plane. More waveform implementations will follow in subsequent versions of the code.

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

- Third, in the `src` directory, build the executable `Simulation` via the `make` command.
- Forth, run the simulation.
You determine the number of processes via the MPI execution command. Note that in 2D and 3D simulations this number has to coincide with the actual number of patches, as described above.
Here, the simulation would be executed distributed over four processes:

```
mpirun -np 4 ./Simulation
```
- Monitor stdout and stderr. The unique simulation identifier number (starting timestep = name of data directory), the process steps, and the used wall times per step are printed on stdout. Errors are printed on stderr.
Note: Convergence of the employed CNode solver can not be guaranteed and issues of this kind can hardly be predicted. On top, they are even system dependent. Piece of advice: Only pass decimal numbers for the grid settings and initial conditions.
CNode warnings and errors are reported on stdout and stderr.
A `config.txt` file containing the relevant part of `main.cpp` is written to the output directory in order to save the simulation settings of each particular run.

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

1.3.1 Note on Simulation Settings

You may want to start with two Gaussian pulses in 1D colliding head-on in a pump-probe setup. For this event, specify a high-frequency probe pulse with a low amplitude and a low-frequency pump pulse with a high frequency. Both frequencies should be chosen to be below a fourth of the Nyquist frequency to avoid unphysical dispersion effects. The wavelengths should neither be chosen too large (bulky wave) on a fine patchwork of narrow patches. Their communication might be problematic with too small halo layer depths. You would observe a blurring over time. The amplitudes need be below 1 – the critical field strength – for the weak-field expansion to be valid.

You can then investigate the arising of higher harmonics in frequency space via a Fourier analysis. The signals from the higher harmonics can be highlighted by subtracting the results of the same simulation in the linear Maxwell vacuum. You will be left with the nonlinear effects.

Choosing the probe pulse to be polarized with an angle to the polarization of the pump you may observe a fractional polarization flip of the probe due to their nonlinear interaction.

Decide beforehand which steps you need to be written to disk for your analysis.

Example scenarios of colliding Gaussians are preconfigured for any dimension.

1.3.2 Note on Resource Occupation

The computational load depends mostly on the grid size. The order of accuracy of the numerical scheme and CNode are rather secondary except for simulations running on many processing units, as the communication load is dependent on the stencil order.

Simulations in 1D are relatively cheap and can easily be run on a modern laptop within minutes. The output size per step is less than a megabyte.

Simulations in 2D with about one million grid points are still feasible for a personal machine but might take about an hour of time to finish. The output size per step is in the range of some dozen megabytes.

Sensible simulations in 3D require large memory resources and therefore need to be run on distributed systems. Even hundreds of cores can be kept busy for many hours or days. The output size quickly amounts to dozens of gigabytes for just a single state.

1.3.3 Note on Output Analysis

The field data are written to csv files. A `SimResults` folder is created in the chosen output directory if it does not exist and a folder named after the starting timestep of the simulation is created where the csv files are written into. The timestep filename is given in the form `yy-mm-dd_hh-MM-ss`.

There are six columns, 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.

Every process writes to its own csv file, the ending of which (after an underscore) corresponds to the process number, as described above. This is not an elegant solution, but the best portable way that also works fast. On the other hand, it requires some postprocessing to read-in the files in order. A Python `module` taking care of this is provided.

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

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

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

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

Some example Python analysis scripts can be found in the `examples`. The [first steps](#) demonstrate how the simulated data is accurately read-in from disk to numpy arrays using the provided [get field data module](#). [Harmonic generation](#) in various forms is sketched as one application showing nonlinear quantum vacuum effects. There is however *no simulation data provided* as it would make the repository size unnecessarily large.

1.4 Authors

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

Chapter 2

Hierarchical Index

2.1 Class Hierarchy

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

Gauss1D	13
Gauss2D	19
Gauss3D	25
gaussian1D	30
gaussian2D	32
gaussian3D	34
ICSetter	36
Lattice	45
LatticePatch	60
OutputManager	97
PlaneWave	103
PlaneWave1D	107
PlaneWave2D	110
PlaneWave3D	112
planewave	106
Simulation	115
TimeEvolution	135

Chapter 3

Data Structure Index

3.1 Data Structures

Here are the data structures with brief descriptions:

Gauss1D	Class for Gaussian waves in 1D	13
Gauss2D	Class for Gaussian waves in 2D	19
Gauss3D	Class for Gaussian waves in 3D	25
gaussian1D	1D Gaussian wave structure	30
gaussian2D	2D Gaussian wave structure	32
gaussian3D	3D Gaussian wave structure	34
ICSetter	ICSetter class to initialize wave types with default parameters	36
Lattice	Lattice class for the construction of the enveloping discrete simulation space	45
LatticePatch	LatticePatch class for the construction of the patches in the enveloping lattice	60
OutputManager	Output Manager class to generate and coordinate output writing to disk	97
PlaneWave	Super-class for plane waves	103
planewave	Plane wave structure	106
PlaneWave1D	Class for plane waves in 1D	107
PlaneWave2D	Class for plane waves in 2D	110
PlaneWave3D	Class for plane waves in 3D	112
Simulation	Simulation class to instantiate the whole walkthrough of a Simulation	115
TimeEvolution	Monostate TimeEvolution Class to propagate the field data in time in a given order of the HE weak-field expansion	135

Chapter 4

File Index

4.1 File List

Here is a list of all files with brief descriptions:

src/ DerivationStencils.cpp	
Empty. All definitions in the header	139
src/ DerivationStencils.h	
Definition of derivation stencils from order 1 to 13	140
src/ ICSetters.cpp	
Implementation of the plane wave and Gaussian wave packets in 1D, 2D, 3D	178
src/ ICSetters.h	
Declaration of the plane wave and Gaussian wave packets in 1D, 2D, 3D	182
src/ LatticePatch.cpp	
Costruction of the overall envelope lattice and the lattice patches	187
src/ LatticePatch.h	
Declaration of the lattice and lattice patches	202
src/ main.cpp	
Main function to configure the user's simulation settings	209
src/ Outputters.cpp	
Generation of output writing to disk	221
src/ Outputters.h	
OutputManager class to outstream simulation data	223
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)	225
src/ SimulationClass.h	
Class for the Simulation object calling all functionality: from wave settings over lattice construction, time evolution and outputs initialization of the CNode object	229
src/ SimulationFunctions.cpp	
Implementation of the complete simulation functions for 1D, 2D, and 3D, as called in the main function	232
src/ SimulationFunctions.h	
Full simulation functions for 1D, 2D, and 3D used in main.cpp	247
src/ TimeEvolutionFunctions.cpp	
Implementation of functions to propagate data vectors in time according to Maxwell's equations, and various orders in the HE weak-field expansion	259
src/ TimeEvolutionFunctions.h	
Functions to propagate data vectors in time according to Maxwell's equations, and various orders in the HE weak-field expansion	286

Chapter 5

Data Structure Documentation

5.1 Gauss1D Class Reference

class for Gaussian waves in 1D

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

Public Member Functions

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

Private Attributes

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

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

5.1.1 Detailed Description

class for Gaussian waves 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 [88](#) of file [ICSetters.h](#).

5.1.2 Constructor & Destructor Documentation

5.1.2.1 Gauss1D()

```
Gauss1D::Gauss1D (
    vector< sunrealtype > k = {1, 0, 0},
    vector< sunrealtype > p = {0, 0, 1},
    vector< sunrealtype > xo = {0, 0, 0},
    sunrealtype phig_ = 1.01,
    vector< sunrealtype > 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 122 of file `ICSetters.cpp`.

```
00124 {
00125     kx = k[0];      /** - wavevectors \f$ k_x \f$ */
00126     ky = k[1];      /** - \f$ k_y \f$ */
00127     kz = k[2];      /** - \f$ k_z \f$ normalized to \f$ 1/\lambda \f$ */
00128     px = p[0];      /** - amplitude (polarization) in x-direction */
00129     py = p[1];      /** - amplitude (polarization) in y-direction */
00130     pz = p[2];      /** - amplitude (polarization) in z-direction */
00131     phix = phi[0];   /** - phase shift in x-direction */
00132     phiy = phi[1];   /** - phase shift in y-direction */
00133     phiz = phi[2];   /** - phase shift in z-direction */
00134     phig = phig;     /** - width */
00135     x0x = xo[0];      /** - shift from origin in x-direction */
00136     x0y = xo[1];      /** - shift from origin in y-direction */
00137     x0z = xo[2];      /** - shift from origin in z-direction */
00138 }
```

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 141 of file `ICSetters.cpp`.

```
00142 {
00143     const sunrealtype wavelength =
00144         sqrt(kx * kx + ky * ky + kz * kz); /* \f$ 1/\lambda \f$ */
00145     x = x - x0x; /* x-coordinate minus shift from origin */
00146     y = y - x0y; /* y-coordinate minus shift from origin */
00147     z = z - x0z; /* z-coordinate minus shift from origin */
00148     const sunrealtype kScalarX = (kx * x + ky * y + kz * z) * 2 *
00149         numbers::pi; /* \f$ 2\pi \cdot \vec{k} \cdot \vec{x} \f$ */
00150     const sunrealtype envelopeAmp =
00151         exp(-(x * x + y * y + z * z) / phig / phig); /* enveloping Gauss shape */
00152     // Gaussian wave definition
00153     const array<sunrealtype, 3> E{
00154         {
00155             px * cos(kScalarX - phix) * envelopeAmp, /* \f$ E_x \f$ */
00156             py * cos(kScalarX - phiy) * envelopeAmp, /* \f$ E_y \f$ */
00157             pz * cos(kScalarX - phiz) * envelopeAmp}; /* \f$ E_z \f$ */
00158         // Put E-field into space
00159         pTo6Space[0] += E[0];
00160         pTo6Space[1] += E[1];
00161         pTo6Space[2] += E[2];
00162         // and B-field
00163         pTo6Space[3] += (ky * E[2] - kz * E[1]) / wavelength;
00164         pTo6Space[4] += (kz * E[0] - kx * E[2]) / wavelength;
00165         pTo6Space[5] += (kx * E[1] - ky * E[0]) / wavelength;
00166     }
```

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

5.1.4 Field Documentation

5.1.4.1 k_x

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

wavenumber k_x

Definition at line 91 of file [ICSetters.h](#).

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

5.1.4.2 k_y

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

wavenumber k_y

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

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

5.1.4.3 k_z

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

wavenumber k_z

Definition at line 95 of file [ICSetters.h](#).

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

5.1.4.4 Φ_g

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

pulse width Φ_g

Definition at line 115 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 103 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 105 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 107 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 97 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 99 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 101 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 109 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 111 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 113 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 waves in 2D

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

Public Member Functions

- [Gauss2D](#) (vector< sunrealtype > dis_={0, 0, 0}, vector< sunrealtype > axis_={1, 0, 0}, sunrealtype Amp_←_1.0l, sunrealtype phip_=0, sunrealtype w0_=1e-5, sunrealtype zr_=4e-5, sunrealtype Ph0_=2e-5, sunrealtype PhA_=0.45e-5)
construction with default parameters
- void [addToSpace](#) (sunrealtype x, sunrealtype y, sunrealtype z, sunrealtype *pTo6Space) const
function for the actual implementation in space

Private Attributes

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

5.2.1 Detailed Description

class for Gaussian waves in 2D

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

- propagation direction (subtracted distance to origin) z_g
- radial distance to propagation axis $r = \sqrt{x^2 - z_g^2}$
- $k = 2\pi/\lambda$
- 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 142 of file [ICSetters.h](#).

5.2.2 Constructor & Destructor Documentation

5.2.2.1 Gauss2D()

```
Gauss2D::Gauss2D (
    vector< sunrealtype > dis_ = {0, 0, 0},
    vector< sunrealtype > axis_ = {1, 0, 0},
    sunrealtype Amp_ = 1.01,
    sunrealtype phip_ = 0,
    sunrealtype w0_ = 1e-5,
    sunrealtype zr_ = 4e-5,
    sunrealtype Ph0_ = 2e-5,
    sunrealtype PhA_ = 0.45e-5 )
```

construction with default parameters

[Gauss2D](#) construction with

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

Definition at line 169 of file `ICSetters.cpp`.

```
00171
00172     dis = dis_;           /** - center it approaches */
00173     axis = axis_;         /** - direction form where it comes */
00174     Amp = Amp_;           /** - amplitude */
00175     phip = phip_;         /** - polarization rotation from TE-mode */
00176     w0 = w0_;            /** - taille */
00177     zr = zr_;            /** - Rayleigh length */
00178     Ph0 = Ph0_;          /** - beam center */
00179     PhA = PhA_;          /** - beam length */
00180     A1 = Amp * cos(phia); // amplitude in z-direction
00181     A2 = Amp * sin(phia); // amplitude on xy-plane
00182     lambda = numbers::pi * w0 * w0 / zr; // formula for wavelength
00183 }
```

References [A1](#), [A2](#), [Amp](#), [axis](#), [dis](#), [lambda](#), [Ph0](#), [PhA](#), [phia](#), [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 185 of file `ICSetters.cpp`.

```
00186
00187     //f$ \vec{x} = \vec{x}_0-\vec{dis} \f$ // coordinates minus distance to
00188     //origin
00189     x -= dis[0];
00190     y -= dis[1];
00191     // z-=dis[2];
00192     z = NAN;
00193     // \f$ z_g = \vec{x}\cdot\vec{e}_g \f$ projection on propagation axis
00194     const sunrealtype zg =
00195         x * axis[0] + y * axis[1]; //+z*axis[2]; // =z-z0 -> propagation
00196         //direction, minus origin
00197     // \f$ r = \sqrt{\vec{x}^2 - z_g^2} \f$ -> pythagoras of radius minus
00198     // projection on prop axis
00199     const sunrealtype r = sqrt((x * x + y * y /*+z*z*/)) -
00200         zg * zg; // radial distance to propagation axis
00201     // \f$ w(z) = w0\sqrt{1+(z_g/z_R)^2} \f$
00202     const sunrealtype wz = w0 * sqrt(1 + (zg * zg / zr / zr)); // waist at position z
00203     // \f$ g(z) = atan(z_g/z_r) \f$
00204     const sunrealtype gz = atan(zg / zr); // Gouy phase
00205     // \f$ R(z) = z_g*(1+(z_r/z_g)^2) \f$
00206     sunrealtype Rz = NAN; // beam curvature
00207     if (zg != 0)
00208         Rz = zg * (1 + (zr * zr / zg / zg));
00209     else
00210         Rz = 1e308;
00211     // wavenumber \f$ k = 2\pi/\lambda \f$
00212     const sunrealtype k = 2 * numbers::pi / lambda;
00213     // \f$ \Phi_F = kr^2/(2R(z))+g(z)-kz_g \f$
00214     const sunrealtype PhF =
00215         -k * r * r / (2 * Rz) + gz - k * zg; // to be inserted into cosine
00216     // \f$ G = \sqrt{w_0/w_z}\e^{-r/w(z)^2}\e^{(zg-Ph0)^2/PhA^2}\cos(PhF) \f$ -
00217     // CCode is a diva, no chance to remove the square in the second exponential
00218     // -> h too small
00219     const sunrealtype G2D = sqrt(w0 / wz) * exp(-r * r / wz / wz) *
00220         exp(-(zg - Ph0) * (zg - Ph0) / PhA / PhA) *
00221         cos(PhF); // gauss shape
00222     // \f$ c_\alpha = \vec{e}_x\cdot\vec{axis} \f$
00223     // projection components; do like this for CCode convergence -> otherwise
00224     // results in machine error values for non-existent field components if
00225     // axis[0] and axis[1] are given
00226     const sunrealtype ca =
```

```

00227     axis[0]; // x-component of propagation axis which is given as parameter
00228     const sunrealtype sa = sqrt(1 - ca * ca); // no z-component for 2D propagation
00229     // E-field to space: polarization in xy-plane (A2) is projection of
00230     // z-polarization (A1) on x- and y-directions
00231     pTo6Space[0] += sa * (G2D * A2);
00232     pTo6Space[1] += -ca * (G2D * A2);
00233     pTo6Space[2] += G2D * A1;
00234     // B-field -> negative derivative wrt polarization shift of E-field
00235     pTo6Space[3] += -sa * (G2D * A1);
00236     pTo6Space[4] += ca * (G2D * A1);
00237     pTo6Space[5] += G2D * A2;
00238 }

```

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 162 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 164 of file [ICSetters.h](#).

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

5.2.4.3 Amp

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

amplitude A

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

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

5.2.4.4 axis

```
vector<sunrealtype> Gauss2D::axis [private]
```

normalized propagation axis

Definition at line 147 of file [ICSetters.h](#).

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

5.2.4.5 dis

```
vector<sunrealtype> Gauss2D::dis [private]
```

distance maximum to origin

Definition at line 145 of file [ICSetters.h](#).

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

5.2.4.6 lambda

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

wavelength λ

Definition at line 166 of file [ICSetters.h](#).

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

5.2.4.7 Ph0

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

center of beam Φ_0

Definition at line 158 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 160 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 152 of file [ICSetters.h](#).

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

5.2.4.10 w0

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

taille ω_0

Definition at line 154 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 156 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 waves in 3D

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

Public Member Functions

- [Gauss3D](#) (vector< sunrealtype > dis_={0, 0, 0}, vector< sunrealtype > axis_={1, 0, 0}, sunrealtype Amp_₀=1.0l, sunrealtype phip_=0, sunrealtype w0_=1e-5, sunrealtype zr_=4e-5, sunrealtype Ph0_=2e-5, sunrealtype PhA_=0.45e-5)
construction with default parameters
- void [addToSpace](#) (sunrealtype x, sunrealtype y, sunrealtype z, sunrealtype *pTo6Space) const
function for the actual implementation in space

Private Attributes

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

5.3.1 Detailed Description

class for Gaussian waves in 3D

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

- propagation direction (subtracted distance to origin) z_g
- radial distance to propagation axis $r = \sqrt{x^2 + y^2}$
- $k = 2\pi/\lambda$
- 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 194 of file [ICSetters.h](#).

5.3.2 Constructor & Destructor Documentation

5.3.2.1 Gauss3D()

```
Gauss3D::Gauss3D (
    vector< sunrealtype > dis_ = {0, 0, 0},
    vector< sunrealtype > axis_ = {1, 0, 0},
    sunrealtype Amp_ = 1.01,
    sunrealtype phip_ = 0,
    sunrealtype w0_ = 1e-5,
    sunrealtype zr_ = 4e-5,
    sunrealtype Ph0_ = 2e-5,
    sunrealtype PhA_ = 0.45e-5 )
```

construction with default parameters

[Gauss3D](#) construction with

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

Definition at line 241 of file `ICSetters.cpp`.

```
00245                                     {
00246     dis = dis_;    /** - center it approaches */
00247     axis = axis_; /** - direction from where it comes */
00248     Amp = Amp_;   /** - amplitude */
00249     // pol=pol_;
00250     phip = phip_; /** - polarization rotation form TE-mode */
00251     w0 = w0_;     /** - taille */
00252     zr = zr_;     /** - Rayleigh length */
00253     Ph0 = Ph0_;   /** - beam center */
00254     PhA = PhA_;   /** - beam length */
00255     lambda = numbers::pi * w0 * w0 / zr;
00256     A1 = Amp * cos(phip);
00257     A2 = Amp * sin(phip);
00258 }
```

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 261 of file `ICSetters.cpp`.

```
00262                                     {
00263     x -= dis[0];
00264     y -= dis[1];
00265     z -= dis[2];
00266     const sunrealtype zg = x * axis[0] + y * axis[1] + z * axis[2];
00267     const sunrealtype r = sqrt((x * x + y * y + z * z) - zg * zg);
00268     const sunrealtype wz = w0 * sqrt(1 + (zg * zg / zr / zr));
00269     const sunrealtype gz = atan(zg / zr);
00270     sunrealtype Rz = NAN;
00271     if (zg != 0)
00272         Rz = zg * (1 + (zr * zr / zg / zg));
00273     else
00274         Rz = 1e308;
00275     const sunrealtype k = 2 * numbers::pi / lambda;
00276     const sunrealtype PhF = -k * r * r / (2 * Rz) + gz - k * zg;
00277     const sunrealtype G3D = (w0 / wz) * exp(-r * r / wz / wz) *
00278         exp(-(zg - Ph0) * (zg - Ph0) / PhA / PhA) * cos(PhF);
00279     const sunrealtype ca = axis[0];
00280     const sunrealtype sa = sqrt(1 - ca * ca);
00281     pTo6Space[0] += sa * (G3D * A2);
00282     pTo6Space[1] += -ca * (G3D * A2);
00283     pTo6Space[2] += G3D * A1;
00284     pTo6Space[3] += -sa * (G3D * A1);
00285     pTo6Space[4] += ca * (G3D * A1);
00286     pTo6Space[5] += G3D * A2;
00287 }
```

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 216 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 218 of file [ICSetters.h](#).

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

5.3.4.3 Amp

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

amplitude A

Definition at line 201 of file [ICSetters.h](#).

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

5.3.4.4 axis

```
vector<sunrealtype> Gauss3D::axis [private]
```

normalized propagation axis

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

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

5.3.4.5 dis

```
vector<sunrealtype> Gauss3D::dis [private]
```

distance maximum to origin

Definition at line 197 of file [ICSetters.h](#).

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

5.3.4.6 lambda

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

wavelength λ

Definition at line 220 of file [ICSetters.h](#).

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

5.3.4.7 Ph0

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

center of beam Φ_0

Definition at line 212 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 214 of file [ICSetters.h](#).

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

5.3.4.9 phip

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

polarization rotation from TE-mode around propagation direction

Definition at line 204 of file [ICSetters.h](#).

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

5.3.4.10 w0

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

taille ω_0

Definition at line 208 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 210 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

- vector< sunrealtype > [k](#)
- vector< sunrealtype > [p](#)
- vector< sunrealtype > [x0](#)
- sunrealtype [phig](#)
- vector< sunrealtype > [phi](#)

5.4.1 Detailed Description

1D Gaussian wave structure

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

5.4.2 Field Documentation

5.4.2.1 k

```
vector<sunrealtype> gaussian1D::k
```

wavevector (normalized to $1/\lambda$)

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

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

5.4.2.2 p

```
vector<sunrealtype> gaussian1D::p
```

amplitude & polarization vector

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

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

5.4.2.3 phi

```
vector<sunrealtype> gaussian1D::phi
```

phase shift

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

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

5.4.2.4 phig

```
sunrealtype gaussian1D::phig
```

width

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

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

5.4.2.5 x0

```
vector<sunrealtype> gaussian1D::x0
```

shift from origin

Definition at line 28 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

- vector< sunrealtype > [x0](#)
- vector< sunrealtype > [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 34 of file [SimulationFunctions.h](#).

5.5.2 Field Documentation

5.5.2.1 amp

```
sunrealtype gaussian2D::amp
```

amplitude

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

5.5.2.2 axis

```
vector<sunrealtype> gaussian2D::axis
```

direction to center

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

5.5.2.3 ph0

```
sunrealtype gaussian2D::ph0
```

beam center

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

5.5.2.4 phA

```
sunrealtype gaussian2D::phA
```

beam length

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

5.5.2.5 phip

```
sunrealtype gaussian2D::phip
```

polarization rotation

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

5.5.2.6 w0

```
sunrealtype gaussian2D::w0
```

taille

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

5.5.2.7 x0

```
vector<sunrealtype> gaussian2D::x0
```

center

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

5.5.2.8 zr

```
sunrealtype gaussian2D::zr
```

Rayleigh length

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

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

- [src/SimulationFunctions.h](#)

5.6 gaussian3D Struct Reference

3D Gaussian wave structure

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

Data Fields

- vector< sunrealtype > [x0](#)
- vector< sunrealtype > [axis](#)
- sunrealtype [amp](#)
- sunrealtype [phip](#)
- sunrealtype [w0](#)
- sunrealtype [zr](#)
- sunrealtype [ph0](#)
- sunrealtype [phA](#)

5.6.1 Detailed Description

3D Gaussian wave structure

Definition at line 46 of file [SimulationFunctions.h](#).

5.6.2 Field Documentation

5.6.2.1 amp

```
sunrealtype gaussian3D::amp
```

amplitude

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

5.6.2.2 axis

```
vector<sunrealtype> gaussian3D::axis
```

direction to center

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

5.6.2.3 ph0

```
sunrealtype gaussian3D::ph0
```

beam center

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

5.6.2.4 phA

```
sunrealtype gaussian3D::phA
```

beam length

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

5.6.2.5 phip

```
sunrealtype gaussian3D::phip
```

polarization rotation

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

5.6.2.6 w0

```
sunrealtype gaussian3D::w0
```

taille

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

5.6.2.7 x0

```
vector<sunrealtype> gaussian3D::x0
```

center

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

5.6.2.8 zr

```
sunrealtype gaussian3D::zr
```

Rayleigh length

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

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

- [src/SimulationFunctions.h](#)

5.7 ICSetter Class Reference

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

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

Public Member Functions

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

Private Attributes

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

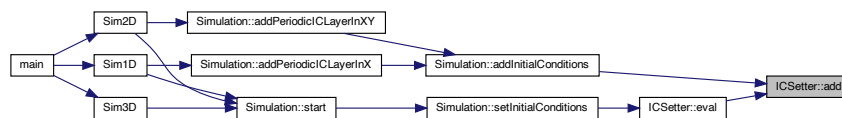
Definition at line 302 of file [ICSetters.cpp](#).

```
00303 {
00304     for (const auto wave : planeWaves1D)
00305         wave.addToSpace(x, y, z, pTo6Space);
00306     for (const auto wave : planeWaves2D)
00307         wave.addToSpace(x, y, z, pTo6Space);
00308     for (const auto wave : planeWaves3D)
00309         wave.addToSpace(x, y, z, pTo6Space);
00310     for (const auto wave : gauss1Ds)
00311         wave.addToSpace(x, y, z, pTo6Space);
00312     for (const auto wave : gauss2Ds)
00313         wave.addToSpace(x, y, z, pTo6Space);
00314     for (const auto wave : gauss3Ds)
00315         wave.addToSpace(x, y, z, pTo6Space);
00316 }
```

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 (
    vector< sunrealtype > k = {1, 0, 0},
    vector< sunrealtype > p = {0, 0, 1},
    vector< sunrealtype > xo = {0, 0, 0},
    sunrealtype phig_ = 1.01,
    vector< sunrealtype > phi = {0, 0, 0} )
```

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

Add Gaussian waves in 1D to their container vector

Definition at line 337 of file [ICSetters.cpp](#).

```
00339 {
00340     gauss1Ds.emplace_back(Gauss1D(k, p, xo, phig_, phi));
00341 }
```

References [gauss1Ds](#).

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

Here is the caller graph for this function:



5.7.2.3 addGauss2D()

```
void ICSetter::addGauss2D (
    vector< sunrealtype > dis_ = {0, 0, 0},
    vector< sunrealtype > axis_ = {1, 0, 0},
    sunrealtype Amp_ = 1.01,
    sunrealtype phip_ = 0,
    sunrealtype w0_ = 1e-5,
    sunrealtype zr_ = 4e-5,
    sunrealtype Ph0_ = 2e-5,
    sunrealtype PhA_ = 0.45e-5 )
```

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

Add Gaussian waves in 2D to their container vector

Definition at line 344 of file [ICSetters.cpp](#).

```
00346 {
00347     gauss2Ds.emplace_back(
00348         Gauss2D(dis_, axis_, Amp_, phip_, w0_, zr_, Ph0_, PhA_));
00349 }
```

References [gauss2Ds](#).

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

Here is the caller graph for this function:



5.7.2.4 addGauss3D()

```
void ICSetter::addGauss3D (
    vector< sunrealtype > dis_ = {0, 0, 0},
    vector< sunrealtype > axis_ = {1, 0, 0},
    sunrealtype Amp_ = 1.01,
    sunrealtype phip_ = 0,
    sunrealtype w0_ = 1e-5,
    sunrealtype zr_ = 4e-5,
    sunrealtype Ph0_ = 2e-5,
    sunrealtype PhA_ = 0.45e-5 )
```

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

Add Gaussian waves in 3D to their container vector

Definition at line 352 of file [ICSetters.cpp](#).

```
00354 {
00355     gauss3Ds.emplace_back(
00356         Gauss3D(dis_, axis_, Amp_, phip_, w0_, zr_, Ph0_, PhA_));
00357 }
```

References [gauss3Ds](#).

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

Here is the caller graph for this function:



5.7.2.5 addPlaneWave1D()

```
void ICSetter::addPlaneWave1D (
    vector< sunrealtype > k = {1, 0, 0},
    vector< sunrealtype > p = {0, 0, 1},
    vector< sunrealtype > 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 319 of file [ICSetters.cpp](#).

```
00320 {
00321     planeWaves1D.emplace_back(PlaneWave1D(k, p, phi));
00322 }
```

References [planeWaves1D](#).

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

Here is the caller graph for this function:



5.7.2.6 addPlaneWave2D()

```
void ICSetter::addPlaneWave2D (
    vector< sunrealtype > k = {1, 0, 0},
    vector< sunrealtype > p = {0, 0, 1},
    vector< sunrealtype > 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 325 of file `ICSetters.cpp`.

```
00326 {
00327     planeWaves2D.emplace_back(PlaneWave2D(k, p, phi));
00328 }
```

References [planeWaves2D](#).

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

Here is the caller graph for this function:



5.7.2.7 addPlaneWave3D()

```
void ICSetter::addPlaneWave3D (
    vector< sunrealtype > k = {1, 0, 0},
    vector< sunrealtype > p = {0, 0, 1},
    vector< sunrealtype > 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 331 of file [ICSetters.cpp](#).

```
00332 {
00333     planeWaves3D.emplace_back(PlaneWave3D(k, p, phi));
00334 }
```

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 add field values

Definition at line 290 of file [ICSetters.cpp](#).

```
00291 {
00292     pTo6Space[0] = 0;
00293     pTo6Space[1] = 0;
00294     pTo6Space[2] = 0;
00295     pTo6Space[3] = 0;
00296     pTo6Space[4] = 0;
00297     pTo6Space[5] = 0;
00298     add(x, y, z, pTo6Space);
00299 }
```

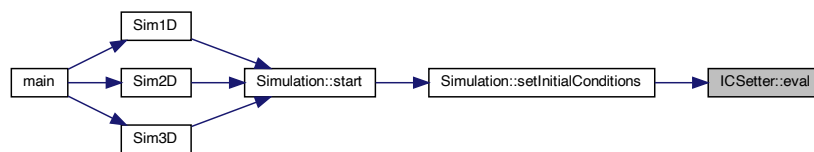
References [add\(\)](#).

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

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

container vector for Gaussian waves in 1D

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

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

5.7.3.2 gauss2Ds

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

container vector for Gaussian waves in 2D

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

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

5.7.3.3 gauss3Ds

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

container vector for Gaussian waves in 3D

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

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

5.7.3.4 planeWaves1D

```
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

```
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

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

container vector for plane waves in 3D

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

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

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

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

5.8 Lattice Class Reference

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

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

Public Member Functions

- void [initializeCommunicator](#) (const int nx, const int ny, const int nz, const bool per)
function to create and deploy the cartesian communicator
- [Lattice](#) (const int StO)
default construction
- void [setDiscreteDimensions](#) (const sunindextype _nx, const sunindextype _ny, const sunindextype _nz)
component function for resizing the discrete dimensions of the lattice
- void [setPhysicalDimensions](#) (const sunrealtype _lx, const sunrealtype _ly, const sunrealtype _lz)
component function for resizing the physical size of the lattice

- const sunrealtype & [get_tot_lx](#) () const
- const sunrealtype & [get_tot_ly](#) () const
- const sunrealtype & [get_tot_lz](#) () const
- const sunindextype & [get_tot_nx](#) () const
- const sunindextype & [get_tot_ny](#) () const
- const sunindextype & [get_tot_nz](#) () const
- const sunindextype & [get_tot_noP](#) () const
- const sunindextype & [get_tot_noDP](#) () const
- const sunrealtype & [get_dx](#) () const
- const sunrealtype & [get_dy](#) () const
- const sunrealtype & [get_dz](#) () const
- constexpr int [get_dataPointDimension](#) () const
- const int & [get_stencilOrder](#) () const
- const int & [get_ghostLayerWidth](#) () const

Data Fields

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

Private Attributes

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

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

5.8.2 Constructor & Destructor Documentation

5.8.2.1 Lattice()

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

default construction

Construct the lattice and set the stencil order.

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

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

References [statusFlags](#).

5.8.3 Member Function Documentation

5.8.3.1 get_dataPointDimension()

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

getter function

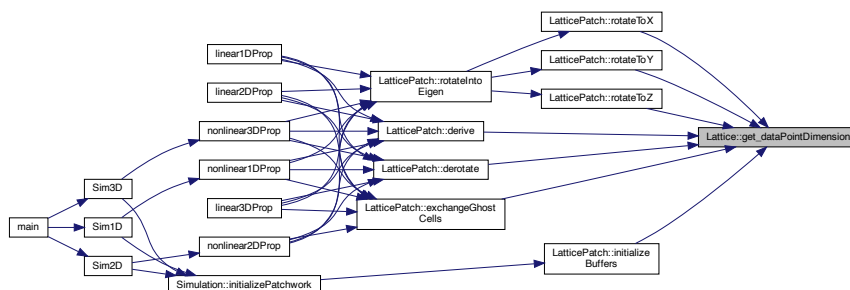
Definition at line 116 of file [LatticePatch.h](#).

```
00116     {
00117     return dataPointDimension;
00118 }
```

References [dataPointDimension](#).

Referenced by [LatticePatch::derive\(\)](#), [LatticePatch::derotate\(\)](#), [LatticePatch::exchangeGhostCells\(\)](#), [LatticePatch::initializeBuffers\(\)](#), [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 113 of file [LatticePatch.h](#).

```
00113 { return dx; }
```

References [dx](#).

5.8.3.3 `get_dy()`

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

getter function

Definition at line 114 of file [LatticePatch.h](#).

```
00114 { return dy; }
```

References [dy](#).

5.8.3.4 `get_dz()`

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

getter function

Definition at line 115 of file [LatticePatch.h](#).

```
00115 { return dz; }
```

References [dz](#).

5.8.3.5 get_ghostLayerWidth()

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

getter function

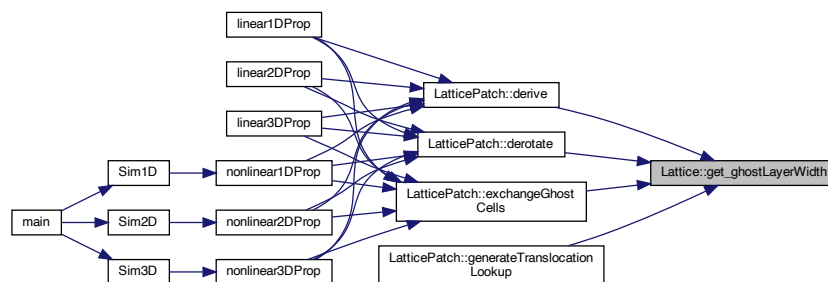
Definition at line 120 of file [LatticePatch.h](#).

```
00120 {
00121     return ghostLayerWidth;
00122 }
```

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

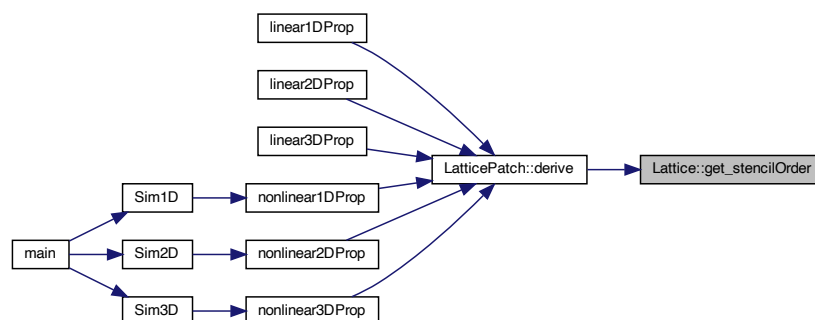
Definition at line 119 of file [LatticePatch.h](#).

```
00119 { 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

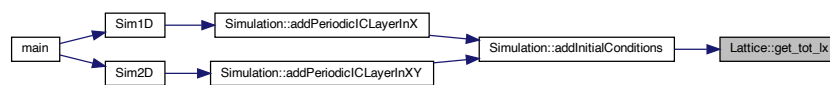
Definition at line 105 of file [LatticePatch.h](#).

```
00105 { 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 106 of file [LatticePatch.h](#).

```
00106 { 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

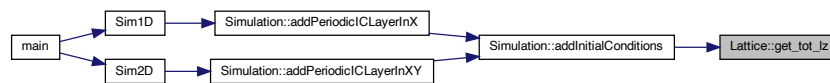
Definition at line 107 of file [LatticePatch.h](#).

```
00107 { 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 112 of file [LatticePatch.h](#).

```
00112 { 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 111 of file [LatticePatch.h](#).

```
00111 { 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 108 of file [LatticePatch.h](#).

```
00108 { 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 109 of file [LatticePatch.h](#).

```
00109 { 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 110 of file [LatticePatch.h](#).

```
00110 { return tot_nz; }
```

References [tot_nz](#).

5.8.3.15 initializeCommunicator()

```
void Lattice::initializeCommunicator (
    const int nx,
    const int ny,
    const int nz,
    const bool per )
```

function to create and deploy the cartesian communicator

Initialize the cartesian communicator.

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

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

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

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

Here is the caller graph for this function:



5.8.3.16 setDiscreteDimensions()

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

component function for resizing the discrete dimensions of the lattice

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

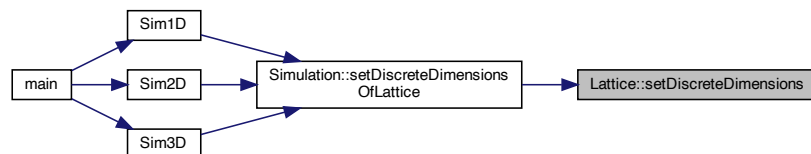
Definition at line 45 of file [LatticePatch.cpp](#).

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

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

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

Here is the caller graph for this function:



5.8.3.17 setPhysicalDimensions()

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

component function for resizing the physical size of the lattice

Set the physical size of the lattice.

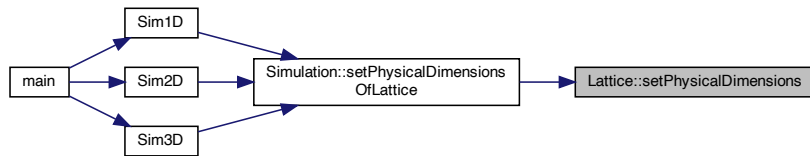
Definition at line 61 of file [LatticePatch.cpp](#).

```
00062 {
00063     tot_lx = _lx;
00064     tot_ly = _ly;
00065     tot_lz = _lz;
00066     // calculate physical distance between points
00067     dx = tot_lx / tot_nx;
00068     dy = tot_ly / tot_ny;
00069     dz = tot_lz / tot_nz;
00070     statusFlags |= FLatticeDimensionSet;
00071 }
```

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

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

Here is the caller graph for this function:



5.8.4 Field Documentation

5.8.4.1 comm

```
MPI_Comm Lattice::comm
```

personal communicator of the lattice

Definition at line 87 of file [LatticePatch.h](#).

Referenced by [LatticePatch::exchangeGhostCells\(\)](#), [Simulation::get_cart_comm\(\)](#), [initializeCommunicator\(\)](#), [Simulation::initializeCVODEobject\(\)](#), 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 65 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 69 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 71 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 73 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 77 of file [LatticePatch.h](#).

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

5.8.4.7 my_prc

```
int Lattice::my_prc
```

number of MPI process

Definition at line 85 of file [LatticePatch.h](#).

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

5.8.4.8 n_prc

```
int Lattice::n_prc
```

number of MPI processes

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

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

5.8.4.9 profobj

```
SUNProfiler Lattice::profobj
```

SUNProfiler object.

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

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

5.8.4.10 statusFlags

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

char for checking if lattice flags are set

Definition at line 79 of file [LatticePatch.h](#).

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

5.8.4.11 stencilOrder

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

stencil order

Definition at line 75 of file [LatticePatch.h](#).

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

5.8.4.12 `sunctx`

```
SUNContext Lattice::sunctx
```

SUNContext object.

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

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

5.8.4.13 `tot_lx`

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

physical size of the lattice in x-direction

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

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

5.8.4.14 `tot_ly`

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

physical size of the lattice in y-direction

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

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

5.8.4.15 `tot_lz`

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

physical size of the lattice in z-direction

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

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

5.8.4.16 tot_noDP

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

number of lattice points times data dimension of each point

Definition at line 67 of file [LatticePatch.h](#).

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

5.8.4.17 tot_noP

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

total number of lattice points

Definition at line 63 of file [LatticePatch.h](#).

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

5.8.4.18 tot_nx

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

number of points in x-direction

Definition at line 57 of file [LatticePatch.h](#).

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

5.8.4.19 tot_ny

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

number of points in y-direction

Definition at line 59 of file [LatticePatch.h](#).

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

5.8.4.20 tot_nz

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

number of points in z-direction

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

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

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

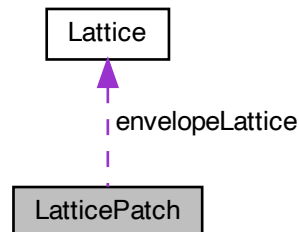
- [src/LatticePatch.h](#)
- [src/LatticePatch.cpp](#)

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
- int [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*
- 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 [initializeGhostLayer](#) ()
- initialize ghost cells for halo exchange*
- void [initializeBuffers](#) ()
- initialize buffers to save derivatives*
- void [exchangeGhostCells](#) (const int dir)
- function to exchange ghost cells in uAux for the derivative*
- void [derive](#) (const int dir)
- function to derive the centered values in uAux and save them noncentered*
- void [checkFlag](#) (unsigned int flag) const
- function to check if a flag has been set and if not abort*

Data Fields

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

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

Private Member Functions

- void [rotateToX](#) (sunrealtype *outArray, const sunrealtype *inArray, const vector< int > &lookup)
- void [rotateToY](#) (sunrealtype *outArray, const sunrealtype *inArray, const vector< int > &lookup)
- void [rotateToZ](#) (sunrealtype *outArray, const sunrealtype *inArray, const vector< int > &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
- const [Lattice](#) * [envelopeLattice](#)
pointer to the enveloping lattice
- vector< sunrealtype > [uAux](#)
- unsigned char [statusFlags](#)

- vector< int > [uTox](#)
- vector< int > [uToy](#)
- vector< int > [uToz](#)
- vector< int > [xTou](#)
- vector< int > [yTou](#)
- vector< int > [zTou](#)

- vector< sunrealtype > [buffX](#)
- vector< sunrealtype > [buffY](#)

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

- `vector< int > lgcTox`
- `vector< int > rgcTox`
- `vector< int > lgcToy`
- `vector< int > rgcToy`
- `vector< int > lgcToz`
- `vector< int > 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 139 of file [LatticePatch.h](#).

5.9.2 Constructor & Destructor Documentation

5.9.2.1 LatticePatch()

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

constructor setting up a default first lattice patch

Construct the lattice patch.

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

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

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

5.9.2.2 ~LatticePatch()

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

destructor freeing parallel vectors

Destruct the patch and thereby destroy the NVectors.

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

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

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

5.9.3 Member Function Documentation

5.9.3.1 checkFlag()

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

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

Check if all flags are set.

Definition at line 580 of file [LatticePatch.cpp](#).

```
00580                                     {
00581     if (!(statusFlags & flag)) {
00582         string errorMessage;
00583         switch (flag) {
00584             case FLatticePatchSetUp:
00585                 errorMessage = "The Lattice patch was not set up please make sure to "
00586                               "initilize a Lattice topology";
00587                 break;
00588             case TranslocationLookupSetUp:
00589                 errorMessage = "The translocation lookup tables have not been generated, "
00590                               "please be sure to run generateTranslocationLookup()";
00591                 break;
00592             case GhostLayersInitialized:
00593                 errorMessage = "The space for the ghost layers has not been allocated, "
00594                               "please be sure to run initializeGhostLayer()";
00595                 break;
00596             case BuffersInitialized:
00597                 errorMessage = "The space for the buffers has not been allocated, please "
00598                               "be sure to run initializeBuffers()";
00599                 break;
00600             default:
00601                 errorMessage = "Uppss, you've made a non-standard error, sadly I can't "
00602                               "help you there";
00603                 break;
00604         }
00605         errorKill(errorMessage);
00606     }
00607     return;
00608 }
```

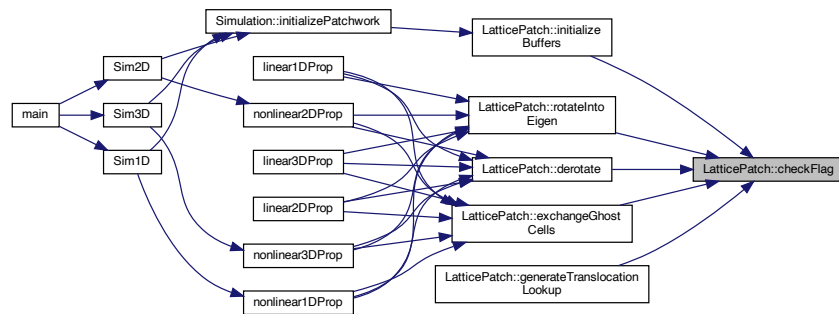
References [BuffersInitialized](#), [errorKill\(\)](#), [FLatticePatchSetUp](#), [GhostLayersInitialized](#), [statusFlags](#), and [TranslocationLookupSetUp](#).

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

```

00611                                     {
00612     // ghost layer width
00613     const int gLW = envelopeLattice->get_ghostLayerWidth();
00614     // dimensionality of data points -> 6
00615     const int dPD = envelopeLattice->get_dataPointDimension();
00616     // total width of patch in given direction including ghost layers at ends
00617     const int dirWidth = discreteSize(dir) + 2 * gLW;
00618     // width of patch only in given direction
00619     const int dirWidth0 = discreteSize(dir);
00620     // size of plane perpendicular to given dimension
00621     const int perpPlainSize = discreteSize() / discreteSize(dir);
00622     // physical distance between points in that direction
00623     sunrealtype dxi = NAN;
00624     switch (dir) {
00625     case 1:
00626         dxi = dx;
00627         break;
00628     case 2:
00629         dxi = dy;
00630         break;
00631     case 3:
00632         dxi = dz;
00633         break;
00634     default:
00635         dxi = 1;
00636         errorKill("Tried to derive in the wrong direction");
00637         break;
00638     }
00639     // Derive according to chosen stencil accuracy order (which determines also
00640     // gLW)
00641     const int order = envelopeLattice->get_stencilOrder();
00642     switch (order) {
00643     case 1:
00644         for (int i = 0; i < perpPlainSize; i++) {
00645             for (int j = (i * dirWidth + gLW + dirWidth0) * dPD;
00646                 j < (i * dirWidth + gLW + dirWidth0) * dPD; j += dPD) {
00647                 uAux[j + 0 - gLW * dPD] = slb(&uAux[j + 0]) / dxi;
00648                 uAux[j + 1 - gLW * dPD] = slb(&uAux[j + 1]) / dxi;
00649                 uAux[j + 2 - gLW * dPD] = slf(&uAux[j + 2]) / dxi;
00650                 uAux[j + 3 - gLW * dPD] = slf(&uAux[j + 3]) / dxi;
00651                 uAux[j + 4 - gLW * dPD] = slf(&uAux[j + 4]) / dxi;

```



```

00652         uAux[j + 5 - gLW * dPD] = s1f(&uAux[j + 5]) / dxi;
00653     }
00654 }
00655 break;
00656 case 2:
00657     for (int i = 0; i < perpPlainSize; i++) {
00658         for (int j = (i * dirWidth + gLW) * dPD;
00659             j < (i * dirWidth + gLW + dirWidthO) * dPD; j += dPD) {
00660             uAux[j + 0 - gLW * dPD] = s2b(&uAux[j + 0]) / dxi;
00661             uAux[j + 1 - gLW * dPD] = s2b(&uAux[j + 1]) / dxi;
00662             uAux[j + 2 - gLW * dPD] = s2f(&uAux[j + 2]) / dxi;
00663             uAux[j + 3 - gLW * dPD] = s2f(&uAux[j + 3]) / dxi;
00664             uAux[j + 4 - gLW * dPD] = s2c(&uAux[j + 4]) / dxi;
00665             uAux[j + 5 - gLW * dPD] = s2c(&uAux[j + 5]) / dxi;
00666         }
00667     }
00668     break;
00669 case 3:
00670     for (int i = 0; i < perpPlainSize; i++) {
00671         for (int j = (i * dirWidth + gLW) * dPD;
00672             j < (i * dirWidth + gLW + dirWidthO) * dPD; j += dPD) {
00673             uAux[j + 0 - gLW * dPD] = s3b(&uAux[j + 0]) / dxi;
00674             uAux[j + 1 - gLW * dPD] = s3b(&uAux[j + 1]) / dxi;
00675             uAux[j + 2 - gLW * dPD] = s3f(&uAux[j + 2]) / dxi;
00676             uAux[j + 3 - gLW * dPD] = s3f(&uAux[j + 3]) / dxi;
00677             uAux[j + 4 - gLW * dPD] = s3f(&uAux[j + 4]) / dxi;
00678             uAux[j + 5 - gLW * dPD] = s3f(&uAux[j + 5]) / dxi;
00679         }
00680     }
00681     break;
00682 case 4:
00683     for (int i = 0; i < perpPlainSize; i++) {
00684         for (int j = (i * dirWidth + gLW) * dPD;
00685             j < (i * dirWidth + gLW + dirWidthO) * dPD; j += dPD) {
00686             uAux[j + 0 - gLW * dPD] = s4b(&uAux[j + 0]) / dxi;
00687             uAux[j + 1 - gLW * dPD] = s4b(&uAux[j + 1]) / dxi;
00688             uAux[j + 2 - gLW * dPD] = s4f(&uAux[j + 2]) / dxi;
00689             uAux[j + 3 - gLW * dPD] = s4f(&uAux[j + 3]) / dxi;
00690             uAux[j + 4 - gLW * dPD] = s4c(&uAux[j + 4]) / dxi;
00691             uAux[j + 5 - gLW * dPD] = s4c(&uAux[j + 5]) / dxi;
00692         }
00693     }
00694     break;
00695 case 5:
00696     for (int i = 0; i < perpPlainSize; i++) {
00697         for (int j = (i * dirWidth + gLW) * dPD;
00698             j < (i * dirWidth + gLW + dirWidthO) * dPD; j += dPD) {
00699             uAux[j + 0 - gLW * dPD] = s5b(&uAux[j + 0]) / dxi;
00700             uAux[j + 1 - gLW * dPD] = s5b(&uAux[j + 1]) / dxi;
00701             uAux[j + 2 - gLW * dPD] = s5f(&uAux[j + 2]) / dxi;
00702             uAux[j + 3 - gLW * dPD] = s5f(&uAux[j + 3]) / dxi;
00703             uAux[j + 4 - gLW * dPD] = s5f(&uAux[j + 4]) / dxi;
00704             uAux[j + 5 - gLW * dPD] = s5f(&uAux[j + 5]) / dxi;
00705         }
00706     }
00707     break;
00708 case 6:
00709     for (int i = 0; i < perpPlainSize; i++) {
00710         for (int j = (i * dirWidth + gLW) * dPD;
00711             j < (i * dirWidth + gLW + dirWidthO) * dPD; j += dPD) {
00712             uAux[j + 0 - gLW * dPD] = s6b(&uAux[j + 0]) / dxi;
00713             uAux[j + 1 - gLW * dPD] = s6b(&uAux[j + 1]) / dxi;
00714             uAux[j + 2 - gLW * dPD] = s6f(&uAux[j + 2]) / dxi;
00715             uAux[j + 3 - gLW * dPD] = s6f(&uAux[j + 3]) / dxi;
00716             uAux[j + 4 - gLW * dPD] = s6c(&uAux[j + 4]) / dxi;
00717             uAux[j + 5 - gLW * dPD] = s6c(&uAux[j + 5]) / dxi;
00718         }
00719     }
00720     break;
00721 case 7:
00722     for (int i = 0; i < perpPlainSize; i++) {
00723         for (int j = (i * dirWidth + gLW) * dPD;
00724             j < (i * dirWidth + gLW + dirWidthO) * dPD; j += dPD) {
00725             uAux[j + 0 - gLW * dPD] = s7b(&uAux[j + 0]) / dxi;
00726             uAux[j + 1 - gLW * dPD] = s7b(&uAux[j + 1]) / dxi;
00727             uAux[j + 2 - gLW * dPD] = s7f(&uAux[j + 2]) / dxi;
00728             uAux[j + 3 - gLW * dPD] = s7f(&uAux[j + 3]) / dxi;
00729             uAux[j + 4 - gLW * dPD] = s7f(&uAux[j + 4]) / dxi;
00730             uAux[j + 5 - gLW * dPD] = s7f(&uAux[j + 5]) / dxi;
00731         }
00732     }
00733     break;
00734 case 8:
00735     for (int i = 0; i < perpPlainSize; i++) {
00736         for (int j = (i * dirWidth + gLW) * dPD;
00737             j < (i * dirWidth + gLW + dirWidthO) * dPD; j += dPD) {
00738             uAux[j + 0 - gLW * dPD] = s8b(&uAux[j + 0]) / dxi;

```

```

00739         uAux[j + 1 - gLW * dPD] = s8b(&uAux[j + 1]) / dxi;
00740         uAux[j + 2 - gLW * dPD] = s8f(&uAux[j + 2]) / dxi;
00741         uAux[j + 3 - gLW * dPD] = s8f(&uAux[j + 3]) / dxi;
00742         uAux[j + 4 - gLW * dPD] = s8c(&uAux[j + 4]) / dxi;
00743         uAux[j + 5 - gLW * dPD] = s8c(&uAux[j + 5]) / dxi;
00744     }
00745 }
00746 break;
00747 case 9:
00748     for (int i = 0; i < perpPlainSize; i++) {
00749         for (int j = (i * dirWidth + gLW) * dPD;
00750             j < (i * dirWidth + gLW + dirWidthO) * dPD; j += dPD) {
00751             uAux[j + 0 - gLW * dPD] = s9b(&uAux[j + 0]) / dxi;
00752             uAux[j + 1 - gLW * dPD] = s9b(&uAux[j + 1]) / dxi;
00753             uAux[j + 2 - gLW * dPD] = s9f(&uAux[j + 2]) / dxi;
00754             uAux[j + 3 - gLW * dPD] = s9f(&uAux[j + 3]) / dxi;
00755             uAux[j + 4 - gLW * dPD] = s9f(&uAux[j + 4]) / dxi;
00756             uAux[j + 5 - gLW * dPD] = s9f(&uAux[j + 5]) / dxi;
00757         }
00758     }
00759 break;
00760 case 10:
00761     for (int i = 0; i < perpPlainSize; i++) {
00762         for (int j = (i * dirWidth + gLW) * dPD;
00763             j < (i * dirWidth + gLW + dirWidthO) * dPD; j += dPD) {
00764             uAux[j + 0 - gLW * dPD] = s10b(&uAux[j + 0]) / dxi;
00765             uAux[j + 1 - gLW * dPD] = s10b(&uAux[j + 1]) / dxi;
00766             uAux[j + 2 - gLW * dPD] = s10f(&uAux[j + 2]) / dxi;
00767             uAux[j + 3 - gLW * dPD] = s10f(&uAux[j + 3]) / dxi;
00768             uAux[j + 4 - gLW * dPD] = s10c(&uAux[j + 4]) / dxi;
00769             uAux[j + 5 - gLW * dPD] = s10c(&uAux[j + 5]) / dxi;
00770         }
00771     }
00772 break;
00773 case 11:
00774     for (int i = 0; i < perpPlainSize; i++) {
00775         for (int j = (i * dirWidth + gLW) * dPD;
00776             j < (i * dirWidth + gLW + dirWidthO) * dPD; j += dPD) {
00777             uAux[j + 0 - gLW * dPD] = s11b(&uAux[j + 0]) / dxi;
00778             uAux[j + 1 - gLW * dPD] = s11b(&uAux[j + 1]) / dxi;
00779             uAux[j + 2 - gLW * dPD] = s11f(&uAux[j + 2]) / dxi;
00780             uAux[j + 3 - gLW * dPD] = s11f(&uAux[j + 3]) / dxi;
00781             uAux[j + 4 - gLW * dPD] = s11f(&uAux[j + 4]) / dxi;
00782             uAux[j + 5 - gLW * dPD] = s11f(&uAux[j + 5]) / dxi;
00783         }
00784     }
00785 break;
00786 case 12:
00787     for (int i = 0; i < perpPlainSize; i++) {
00788         for (int j = (i * dirWidth + gLW) * dPD;
00789             j < (i * dirWidth + gLW + dirWidthO) * dPD; j += dPD) {
00790             uAux[j + 0 - gLW * dPD] = s12b(&uAux[j + 0]) / dxi;
00791             uAux[j + 1 - gLW * dPD] = s12b(&uAux[j + 1]) / dxi;
00792             uAux[j + 2 - gLW * dPD] = s12f(&uAux[j + 2]) / dxi;
00793             uAux[j + 3 - gLW * dPD] = s12f(&uAux[j + 3]) / dxi;
00794             uAux[j + 4 - gLW * dPD] = s12c(&uAux[j + 4]) / dxi;
00795             uAux[j + 5 - gLW * dPD] = s12c(&uAux[j + 5]) / dxi;
00796         }
00797     }
00798 break;
00799 case 13:
00800     // Iterate through all points in the plane perpendicular to the given
00801     // direction
00802     for (int i = 0; i < perpPlainSize; i++) {
00803         // Iterate through the direction for each perpendicular plane point
00804         for (int j = (i * dirWidth + gLW /*to shift left by gLW below */) * dPD;
00805             j < (i * dirWidth + gLW + dirWidthO) * dPD; j += dPD) {
00806             /* Compute the stencil derivative for any of the six field components
00807              * with a ghostlayer width adjusted to the order of the finite
00808              * difference scheme */
00809             uAux[j + 0 - gLW * dPD] = s13b(&uAux[j + 0]) / dxi;
00810             uAux[j + 1 - gLW * dPD] = s13b(&uAux[j + 1]) / dxi;
00811             uAux[j + 2 - gLW * dPD] = s13f(&uAux[j + 2]) / dxi;
00812             uAux[j + 3 - gLW * dPD] = s13f(&uAux[j + 3]) / dxi;
00813             uAux[j + 4 - gLW * dPD] = s13f(&uAux[j + 4]) / dxi;
00814             uAux[j + 5 - gLW * dPD] = s13f(&uAux[j + 5]) / dxi;
00815         }
00816     }
00817 break;
00818
00819 default:
00820     errorKill("Please set an existing stencil order");
00821     break;
00822 }
00823 }

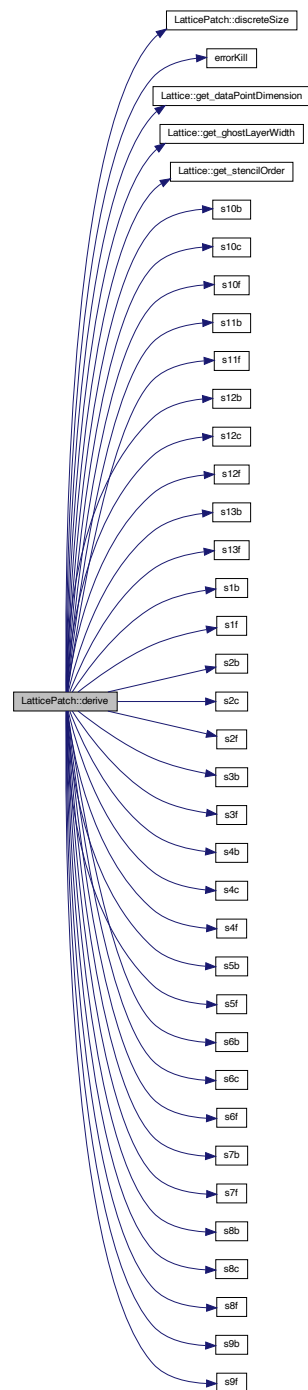
```

References [discreteSize\(\)](#), [dx](#), [dy](#), [dz](#), [envelopeLattice](#), [errorKill\(\)](#), [Lattice::get_dataPointDimension\(\)](#), [Lattice::get_ghostLayerWidth\(\)](#),

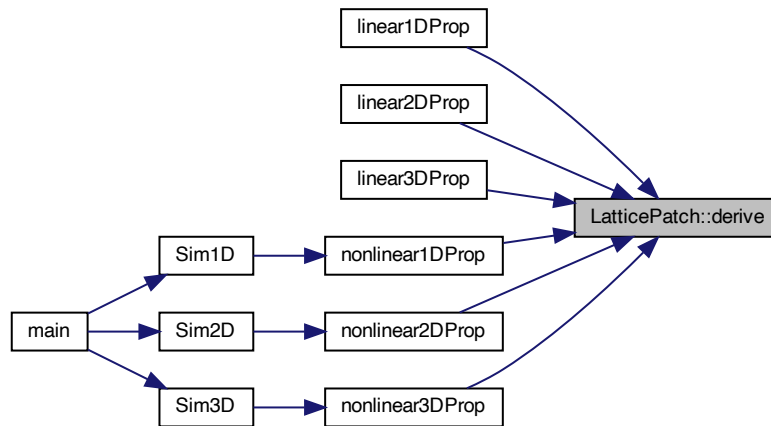
[Lattice::get_stencilOrder\(\)](#), [s10b\(\)](#), [s10c\(\)](#), [s10f\(\)](#), [s11b\(\)](#), [s11f\(\)](#), [s12b\(\)](#), [s12c\(\)](#), [s12f\(\)](#), [s13b\(\)](#), [s13f\(\)](#), [s1b\(\)](#), [s1f\(\)](#), [s2b\(\)](#), [s2c\(\)](#), [s2f\(\)](#), [s3b\(\)](#), [s3f\(\)](#), [s4b\(\)](#), [s4c\(\)](#), [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 394 of file [LatticePatch.cpp](#).

```
00394                                     {
00395     // Check that the lattice as well as the translocation lookups have been set
00396     // up;
00397     checkFlag(FLatticePatchSetUp);
00398     checkFlag(TranslocationLookupSetUp);
00399     const int dPD = envelopeLattice->get_dataPointDimension();
00400     const int gLW = envelopeLattice->get_ghostLayerWidth();
00401     const int uSize = discreteSize();
00402     int ii = 0, target = 0;
00403     switch (dir) {
00404     case 1:
00405         #pragma ivdep
00406         #pragma omp simd
00407         for (int i = 0; i < uSize; i++) {
00408             // get correct indices in u and rotation space
00409             target = dPD * i;
00410             ii = dPD * (uTox[i] - gLW);
00411             buffOut[target + 0] = uAux[5 + ii];
00412             buffOut[target + 1] = (-uAux[ii] + uAux[2 + ii]) / 2.;
00413             buffOut[target + 2] = (uAux[1 + ii] - uAux[3 + ii]) / 2.;
00414             buffOut[target + 3] = uAux[4 + ii];
00415             buffOut[target + 4] = (uAux[1 + ii] + uAux[3 + ii]) / 2.;
00416             buffOut[target + 5] = (uAux[ii] + uAux[2 + ii]) / 2.;
00417         }
00418         break;
00419     case 2:
00420         #pragma omp simd
00421         for (int i = 0; i < uSize; i++) {
00422             target = dPD * i;
```

```

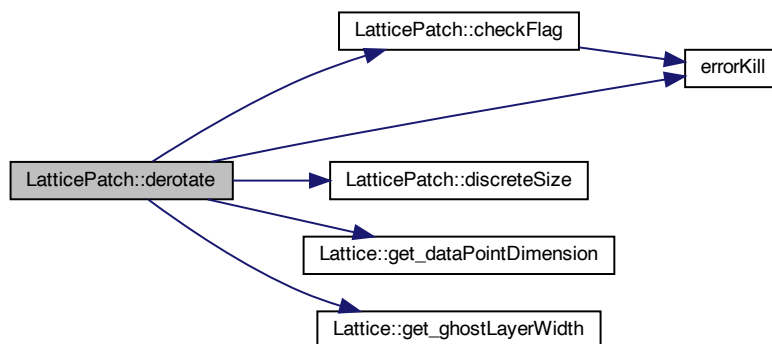
00423     ii = dPD * (uToy[i] - gLW);
00424     buffOut[target + 0] = (uAux[ii] - uAux[2 + ii]) / 2.;
00425     buffOut[target + 1] = uAux[5 + ii];
00426     buffOut[target + 2] = (-uAux[1 + ii] + uAux[3 + ii]) / 2.;
00427     buffOut[target + 3] = (uAux[1 + ii] + uAux[3 + ii]) / 2.;
00428     buffOut[target + 4] = uAux[4 + ii];
00429     buffOut[target + 5] = (uAux[ii] + uAux[2 + ii]) / 2.;
00430 }
00431 break;
00432 case 3:
00433 #pragma omp simd
00434     for (int i = 0; i < uSize; i++) {
00435         target = dPD * i;
00436         ii = dPD * (uToz[i] - gLW);
00437         buffOut[target + 0] = (-uAux[ii] + uAux[2 + ii]) / 2.;
00438         buffOut[target + 1] = (uAux[1 + ii] - uAux[3 + ii]) / 2.;
00439         buffOut[target + 2] = uAux[5 + ii];
00440         buffOut[target + 3] = (uAux[1 + ii] + uAux[3 + ii]) / 2.;
00441         buffOut[target + 4] = (uAux[ii] + uAux[2 + ii]) / 2.;
00442         buffOut[target + 5] = uAux[4 + ii];
00443     }
00444     break;
00445 default:
00446     errorKill("Tried to derotate from the wrong direction");
00447     break;
00448 }
00449 }

```

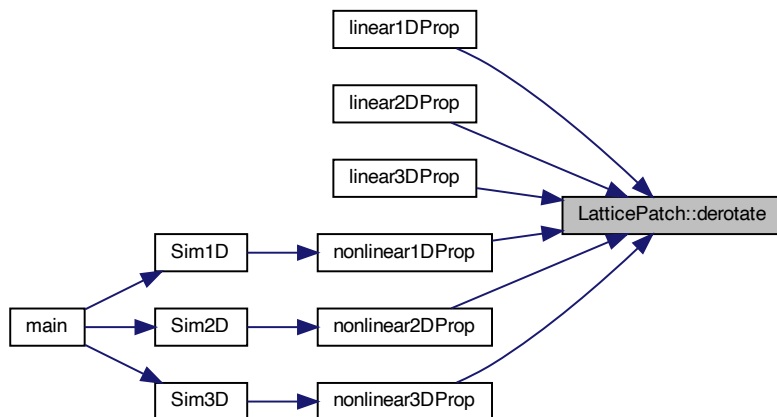
References [checkFlag\(\)](#), [discreteSize\(\)](#), [envelopeLattice](#), [errorKill\(\)](#), [FLatticePatchSetUp](#), [Lattice::get_dataPointDimension\(\)](#), [Lattice::get_ghostLayerWidth\(\)](#), [TranslocationLookupSetUp](#), [uAux](#), [uTox](#), [uToy](#), and [uToz](#).

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()

```
int 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

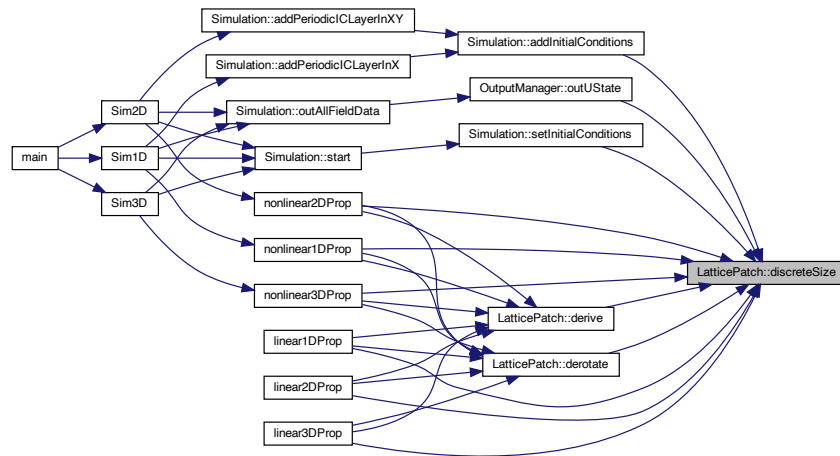
Definition at line 183 of file [LatticePatch.cpp](#).

```
00183 {
00184     switch (dir) {
00185     case 0:
00186         return nx * ny * nz;
00187     case 1:
00188         return nx;
00189     case 2:
00190         return ny;
00191     case 3:
00192         return nz;
00193     // case 4: return uAux.size(); // for debugging
00194     default:
00195         return -1;
00196     }
00197 }
```

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 in uAux for the derivative

Perform the ghost cell exchange in a specified direction.

Definition at line 467 of file [LatticePatch.cpp](#).

```

00467
00468 // Check that the lattice has been set up
00469 checkFlag(FLatticeDimensionSet);
00470 checkFlag(FLatticePatchSetUp);
00471 // Variables to per dimension calculate the halo indices, and distance to
00472 // other side halo boundary
00473 int mx = 1, my = 1, mz = 1, distToRight = 1;
00474 const int gLW = envelopeLattice->get_ghostLayerWidth();
00475 // In the chosen direction m is set to ghost layer width while the others
00476 // remain to form the plane
00477 switch (dir) {
00478 case 1:
00479     mx = gLW;
00480     my = ny;
00481     mz = nz;
00482     distToRight = (nx - gLW);
00483     break;
00484 case 2:
00485     mx = nx;
00486     my = gLW;
00487     mz = nz;
00488     distToRight = nx * (ny - gLW);
00489     break;
00490 case 3:
00491     mx = nx;
00492     my = ny;
00493     mz = gLW;
00494     distToRight = nx * ny * (nz - gLW);
00495     break;
00496 }
00497 // total number of exchanged points
00498 const int dPD = envelopeLattice->get_dataPointDimension();
00499 const int exchangeSize = mx * my * mz * dPD;
00500 // provide size of the halos for ghost cells

```

```

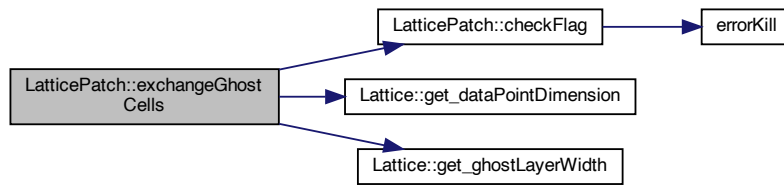
00501 ghostCellLeft.resize(exchangeSize);
00502 ghostCellRight.resize(ghostCellLeft.size());
00503 ghostCellLeftToSend.resize(ghostCellLeft.size());
00504 ghostCellRightToSend.resize(ghostCellLeft.size());
00505 gCLData = &ghostCellLeft[0];
00506 gCRData = &ghostCellRight[0];
00507 statusFlags |= GhostLayersInitialized;
00508
00509 // Initialize running index li for the halo buffers, and index ui of uData for
00510 // data transfer
00511 int li = 0, ui = 0;
00512
00513 for (int iz = 0; iz < mz; iz++) {
00514     for (int iy = 0; iy < my; iy++) {
00515         // uData vector start index of halo data to be transferred
00516         // with each z-step add the whole xy-plane and with y-step the x-range ->
00517         // iterate all x-ranges
00518         ui = (iz * nx * ny + iy * nx) * dPD;
00519         // copy left halo data from uData to buffer, transfer size is given by
00520         // x-length (not x-range) perhaps faster but more fragile C lib copy
00521         // operation (contained in cstring header)
00522         /*
00523         memcpy(&ghostCellLeftToSend[li],
00524             &uData[ui],
00525             sizeof(sunrealtype)*mx*dPD);
00526         // increase ui by the distance to vis-a-vis boundary and copy right halo
00527         data to buffer ui+=distToRight*dPD; memcpy(&ghostCellRightToSend[li],
00528             &uData[ui],
00529             sizeof(sunrealtype)*mx*dPD);
00530         */
00531         // perhaps more safe but slower copy operation (contained in algorithm
00532         // header) performance highly system dependent
00533         copy(&uData[ui], &uData[ui + mx * dPD], &ghostCellLeftToSend[li]);
00534         ui += distToRight * dPD;
00535         copy(&uData[ui], &uData[ui + mx * dPD], &ghostCellRightToSend[li]);
00536
00537         // increase halo index by transferred items per y-iteration step
00538         // (x-length)
00539         li += mx * dPD;
00540     }
00541 }
00542
00543 /* Send and receive the data to and from neighboring latticePatches */
00544 // Adjust direction to cartesian communicator
00545 int dim = 2; // default for dir==1
00546 if (dir == 2) {
00547     dim = 1;
00548 } else if (dir == 3) {
00549     dim = 0;
00550 }
00551 MPI_Request requests[2];
00552 int rank_source = 0, rank_dest = 0;
00553 MPI_Cart_shift(envelopeLattice->comm, dim, -1, &rank_source,
00554     &rank_dest); // s.t. rank_dest is left & v.v.
00555
00556 // nonblocking Isend/Irecv
00557 /*
00558 MPI_Isend(&ghostCellLeftToSend[0], exchangeSize, MPI_SUNREALTYPE, rank_dest,
00559     1, envelopeLattice->comm, &requests[0]); MPI_Irecv(&ghostCellRight[0],
00560     exchangeSize, MPI_SUNREALTYPE, rank_source, 1, envelopeLattice->comm,
00561     &requests[0]); MPI_Isend(&ghostCellRightToSend[0], exchangeSize,
00562     MPI_SUNREALTYPE, rank_source, 2, envelopeLattice->comm, &requests[1]);
00563 MPI_Irecv(&ghostCellLeft[0], exchangeSize, MPI_SUNREALTYPE, rank_dest, 2,
00564     envelopeLattice->comm, &requests[1]);
00565
00566 MPI_Waitall(2, requests, MPI_STATUS_IGNORE);
00567 */
00568
00569 // blocking Sendrecv:
00570
00571 MPI_Sendrecv(&ghostCellLeftToSend[0], exchangeSize, MPI_SUNREALTYPE,
00572     rank_dest, 1, &ghostCellRight[0], exchangeSize, MPI_SUNREALTYPE,
00573     rank_source, 1, envelopeLattice->comm, MPI_STATUS_IGNORE);
00574 MPI_Sendrecv(&ghostCellRightToSend[0], exchangeSize, MPI_SUNREALTYPE,
00575     rank_source, 2, &ghostCellLeft[0], exchangeSize, MPI_SUNREALTYPE,
00576     rank_dest, 2, envelopeLattice->comm, MPI_STATUS_IGNORE);
00577 }

```

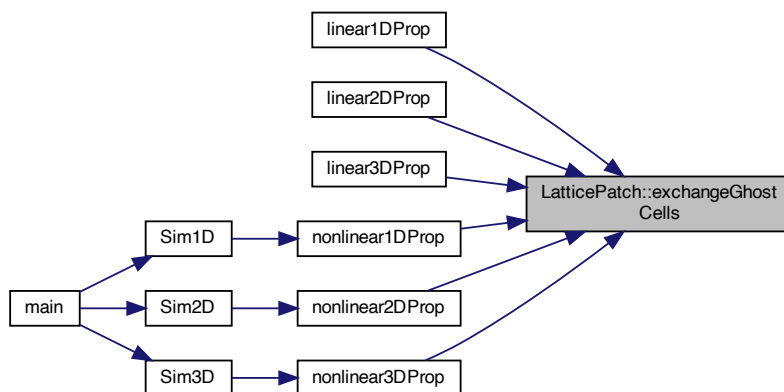
References [checkFlag\(\)](#), [Lattice::comm](#), [envelopeLattice](#), [FLatticeDimensionSet](#), [FLatticePatchSetUp](#), [gCLData](#), [gCRData](#), [Lattice::get_dataPointDimension\(\)](#), [Lattice::get_ghostLayerWidth\(\)](#), [ghostCellLeft](#), [ghostCellLeftToSend](#), [ghostCellRight](#), [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

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

```

00233 {
00234     // Check that the lattice has been set up
00235     checkFlag(FLatticeDimensionSet);
00236     // lenghts for auxilliary layers, including ghost layers
00237     const int gLW = envelopeLattice->get_ghostLayerWidth();
00238     const int mx = nx + 2 * gLW;
00239     const int my = ny + 2 * gLW;
00240     const int mz = nz + 2 * gLW;
00241     // sizes for lookup vectors
00242     // generate u->uAux
00243     uTox.resize(nx * ny * nz);
00244     uToy.resize(nx * ny * nz);
  
```

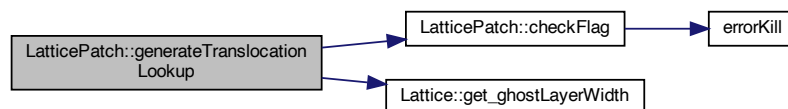
```

00245   uToz.resize(nx * ny * nz);
00246   // generate uAux->u with length including halo
00247   xTou.resize(mx * ny * nz);
00248   yTou.resize(nx * my * nz);
00249   zTou.resize(nx * ny * mz);
00250   // variables for cartesian position in the 3D discrete lattice
00251   int px = 0, py = 0, pz = 0;
00252   for (int i = 0; i < uToy.size(); i++) { // loop over all points in the patch
00253       // calculate cartesian coordinates
00254       px = i % nx;
00255       py = (i / nx) % ny;
00256       pz = (i / nx) / ny;
00257       // fill lookups extended by halos (useful for y and z direction)
00258       uTox[i] = (px + gLW) + py * mx +
00259               pz * mx * ny; // unroll (de-flatten) cartesian dimension
00260       xTou[px + py * mx + pz * mx * ny] =
00261           i; // match cartesian point to u location
00262       uToy[i] = (py + gLW) + pz * my + px * my * nz;
00263       yTou[py + pz * my + px * my * nz] = i;
00264       uToz[i] = (pz + gLW) + px * mz + py * mz * nx;
00265       zTou[pz + px * mz + py * mz * nx] = i;
00266   }
00267   // same for ghost layer lookup tables
00268   lgcTox.resize(gLW * ny * nz);
00269   rgcTox.resize(gLW * ny * nz);
00270   for (int i = 0; i < lgcTox.size(); i++) {
00271       px = i % gLW;
00272       py = (i / gLW) % ny;
00273       pz = (i / gLW) / ny;
00274       lgcTox[i] = px + py * mx + pz * mx * ny;
00275       rgcTox[i] = px + nx + gLW + py * mx + pz * mx * ny;
00276   }
00277   lgcToy.resize(gLW * nx * nz);
00278   rgcToy.resize(gLW * nx * nz);
00279   for (int i = 0; i < lgcToy.size(); i++) {
00280       px = i % nx;
00281       py = (i / nx) % gLW;
00282       pz = (i / nx) / gLW;
00283       lgcToy[i] = py + pz * my + px * my * nz;
00284       rgcToy[i] = py + ny + gLW + pz * my + px * my * nz;
00285   }
00286   lgcToz.resize(gLW * nx * ny);
00287   rgcToz.resize(gLW * nx * ny);
00288   for (int i = 0; i < lgcToz.size(); i++) {
00289       px = i % nx;
00290       py = (i / nx) % ny;
00291       pz = (i / nx) / ny;
00292       lgcToz[i] = pz + px * mz + py * mz * nx;
00293       rgcToz[i] = pz + nz + gLW + px * mz + py * mz * nx;
00294   }
00295   statusFlags |= TranslocationLookupSetUp;
00296 }

```

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

```
00215                                     {
00216     switch (dir) {
00217     case 1:
00218         return dx;
00219     case 2:
00220         return dy;
00221     case 3:
00222         return dz;
00223     default:
00224         errorKill(
00225             "LatticePatch::getDelta function called with wrong dir parameter");
00226         return -1;
00227     }
00228 }
```

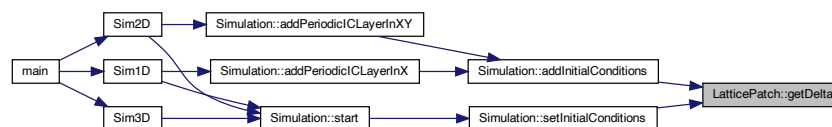
References [dx](#), [dy](#), [dz](#), and [errorKill\(\)](#).

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

```
00452                                     {
00453     // Check that the lattice has been set up
```

```

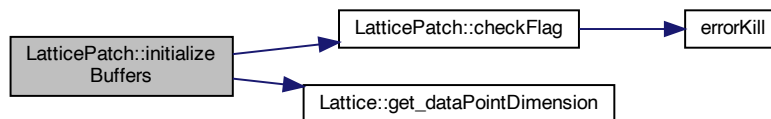
00454     checkFlag(FLatticeDimensionSet);
00455     const int dPD = envelopeLattice->get_dataPointDimension();
00456     buffX.resize(nx * ny * nz * dPD);
00457     buffY.resize(nx * ny * nz * dPD);
00458     buffZ.resize(nx * ny * nz * dPD);
00459     // Set pointers used for propagation functions
00460     buffData[0] = &buffX[0];
00461     buffData[1] = &buffY[0];
00462     buffData[2] = &buffZ[0];
00463     statusFlags |= BuffersInitialized;
00464 }

```

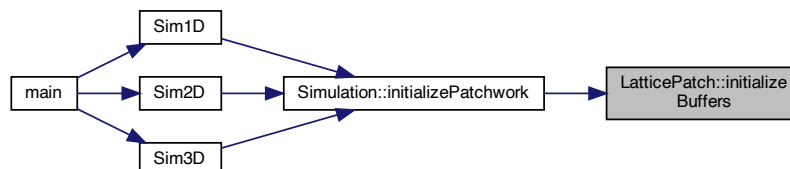
References [buffData](#), [BuffersInitialized](#), [buffX](#), [buffY](#), [buffZ](#), [checkFlag\(\)](#), [envelopeLattice](#), [FLatticeDimensionSet](#), [Lattice::get_dataPointDimension\(\)](#), [nx](#), [ny](#), [nz](#), and [statusFlags](#).

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

Here is the call graph for this function:



Here is the caller graph for this function:



5.9.3.9 initializeGhostLayer()

```
void LatticePatch::initializeGhostLayer ( )
```

initialize ghost cells for halo exchange

5.9.3.10 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 200 of file [LatticePatch.cpp](#).

```

00200                                     {
00201     switch (dir) {
00202     case 1:
00203         return x0;
00204     case 2:
00205         return y0;
00206     case 3:
00207         return z0;
00208     default:
00209         errorKill("LatticePatch::origin function called with wrong dir parameter");
00210         return -1;
00211     }
00212 }

```

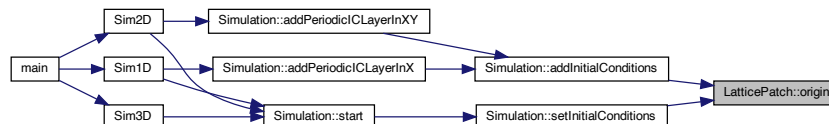
References [errorKill\(\)](#), [x0](#), [y0](#), and [z0](#).

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

Here is the call graph for this function:



Here is the caller graph for this function:



5.9.3.11 rotateIntoEigen()

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

function to rotate u into Z-matrix eigenraum

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

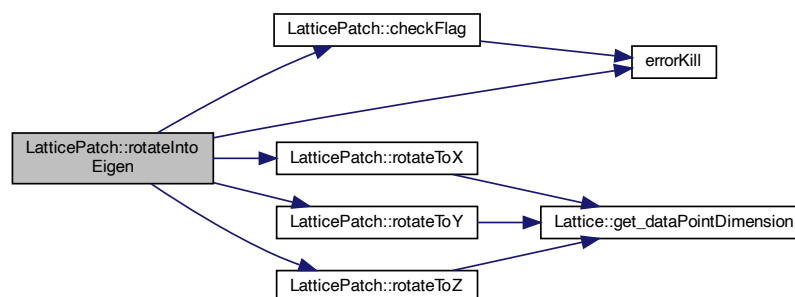
Definition at line 301 of file [LatticePatch.cpp](#).

```
00301 {
00302     // Check that the lattice, ghost layers as well as the translocation lookups
00303     // have been set up;
00304     checkFlag(FLatticePatchSetUp);
00305     checkFlag(TranslocationLookupSetUp);
00306     checkFlag(GhostLayersInitialized); // this check is only after call to
00307                                         // exchange ghost cells
00308     switch (dir) {
00309     case 1:
00310         rotateToX(uAuxData, gCLData, lgcTox);
00311         rotateToX(uAuxData, uData, uTox);
00312         rotateToX(uAuxData, gCRData, rgcTox);
00313         break;
00314     case 2:
00315         rotateToY(uAuxData, gCLData, lgcToy);
00316         rotateToY(uAuxData, uData, uToy);
00317         rotateToY(uAuxData, gCRData, rgcToy);
00318         break;
00319     case 3:
00320         rotateToZ(uAuxData, gCLData, lgcToz);
00321         rotateToZ(uAuxData, uData, uToz);
00322         rotateToZ(uAuxData, gCRData, rgcToz);
00323         break;
00324     default:
00325         errorKill("Tried to rotate into the wrong direction");
00326         break;
00327     }
00328 }
```

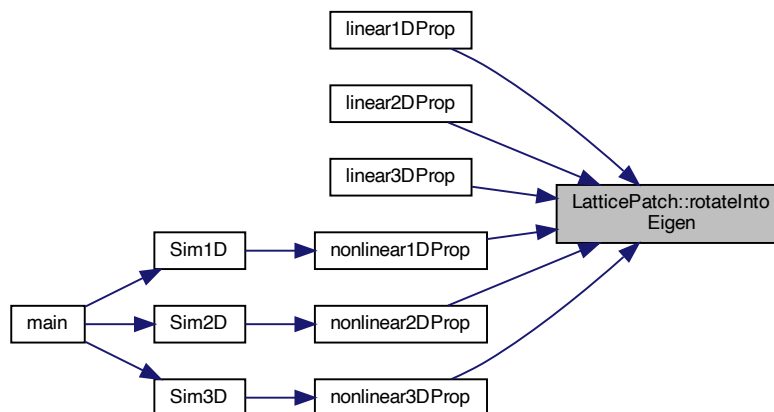
References [checkFlag\(\)](#), [errorKill\(\)](#), [FLatticePatchSetUp](#), [gCLData](#), [gCRData](#), [GhostLayersInitialized](#), [lgcTox](#), [lgcToy](#), [lgcToz](#), [rgcTox](#), [rgcToy](#), [rgcToz](#), [rotateToX\(\)](#), [rotateToY\(\)](#), [rotateToZ\(\)](#), [TranslocationLookupSetUp](#), [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:



5.9.3.12 rotateToX()

```

void LatticePatch::rotateToX (
    sunrealtype * outArray,
    const sunrealtype * inArray,
    const vector< int > & 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 332 of file [LatticePatch.cpp](#).

```

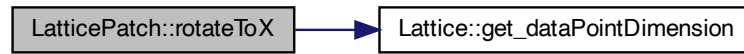
00334 {
00335     int ii = 0, target = 0;
00336     #pragma ivdep
00337     #pragma omp simd // safelen(6)
00338     for (int i = 0; i < lookup.size(); i++) {
00339         // get correct u-vector and spatial indices along previously defined lookup
00340         // tables
00341         target = envelopeLattice->get_dataPointDimension() * lookup[i];
00342         ii = envelopeLattice->get_dataPointDimension() * i;
00343         outArray[target + 0] = -inArray[1 + ii] + inArray[5 + ii];
00344         outArray[target + 1] = inArray[2 + ii] + inArray[4 + ii];
00345         outArray[target + 2] = inArray[1 + ii] + inArray[5 + ii];
00346         outArray[target + 3] = -inArray[2 + ii] + inArray[4 + ii];
00347         outArray[target + 4] = inArray[3 + ii];
00348         outArray[target + 5] = inArray[ii];
00349     }
00350 }

```

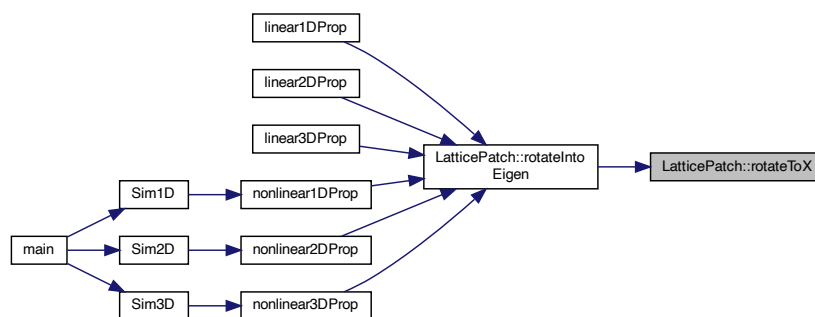
References [envelopeLattice](#), and [Lattice::get_dataPointDimension\(\)](#).

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

Here is the call graph for this function:



Here is the caller graph for this function:



5.9.3.13 rotateToY()

```

void LatticePatch::rotateToY (
    sunrealtype * outArray,
    const sunrealtype * inArray,
    const vector< int > & 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 354 of file [LatticePatch.cpp](#).

```

00356                                     {
00357     int ii = 0, target = 0;
00358     #pragma ivdep
00359     #pragma omp simd
00360     for (int i = 0; i < lookup.size(); i++) {
00361         target = envelopeLattice->get_dataPointDimension() * lookup[i];
00362         ii = envelopeLattice->get_dataPointDimension() * i;
00363         outArray[target + 0] = inArray[ii] + inArray[5 + ii];
00364         outArray[target + 1] = -inArray[2 + ii] + inArray[3 + ii];
00365         outArray[target + 2] = -inArray[ii] + inArray[5 + ii];
00366         outArray[target + 3] = inArray[2 + ii] + inArray[3 + ii];
00367         outArray[target + 4] = inArray[4 + ii];
00368         outArray[target + 5] = inArray[1 + ii];
00369     }
00370 }
  
```

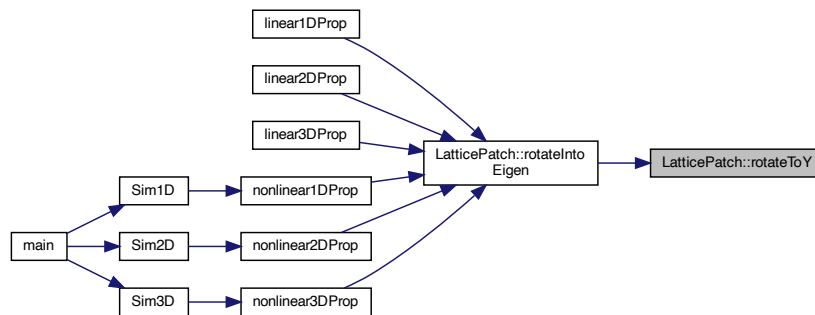
References [envelopeLattice](#), and [Lattice::get_dataPointDimension\(\)](#).

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

Here is the call graph for this function:



Here is the caller graph for this function:



5.9.3.14 rotateToZ()

```

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

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

Definition at line 374 of file [LatticePatch.cpp](#).

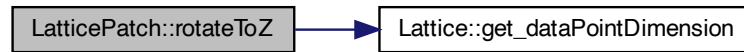
```

00376                                     {
00377     int ii = 0, target = 0;
00378     #pragma ivdep
00379     #pragma omp simd
00380     for (int i = 0; i < lookup.size(); i++) {
00381         target = envelopeLattice->get_dataPointDimension() * lookup[i];
00382         ii = envelopeLattice->get_dataPointDimension() * i;
00383         outArray[target + 0] = -inArray[ii] + inArray[4 + ii];
00384         outArray[target + 1] = inArray[1 + ii] + inArray[3 + ii];
00385         outArray[target + 2] = inArray[ii] + inArray[4 + ii];
00386         outArray[target + 3] = -inArray[1 + ii] + inArray[3 + ii];
00387         outArray[target + 4] = inArray[5 + ii];
00388         outArray[target + 5] = inArray[2 + ii];
00389     }
00390 }
  
```

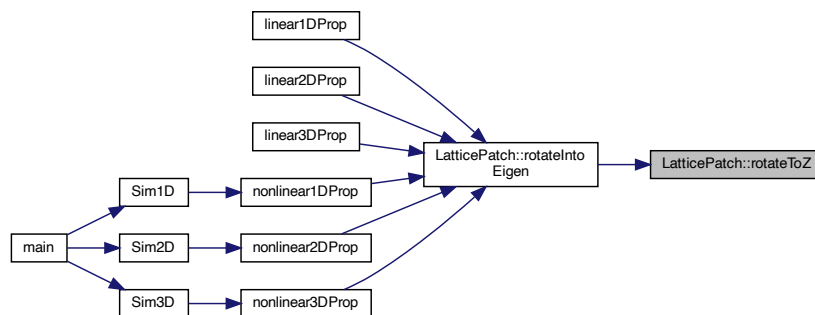
References [envelopeLattice](#), and [Lattice::get_dataPointDimension\(\)](#).

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

Here is the call graph for this function:



Here is the caller graph for this function:



5.9.4 Friends And Related Function Documentation

5.9.4.1 generatePatchwork

```

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

friend function for creating the patchwork slicing of the overall lattice

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

```

00110 {
00111     // Retrieve the ghost layer depth
00112     const int gLW = envelopeLattice.get_ghostLayerWidth();
00113     // Retrieve the data point dimension
00114     const int dPD = envelopeLattice.get_dataPointDimension();
00115     // MPI process/patch
00116     const int my_prc = envelopeLattice.my_prc;
00117     // Determine thicknes of the slice
  
```

```

00118     const sunindextype tot_NOXP = envelopeLattice.get_tot_nx(); // total points of lattice
00119     const sunindextype tot_NOYP = envelopeLattice.get_tot_ny();
00120     const sunindextype tot_NOZP = envelopeLattice.get_tot_nz();
00121     // position of the patch in the lattice of patches - process associated to
00122     // position
00123     const sunindextype LIx = my_prc % DLx;
00124     const sunindextype LIy = (my_prc / DLx) % DLy;
00125     const sunindextype LIz = (my_prc / DLx) / DLy;
00126     // Determine the number of points in the patch and first absolute points in
00127     // each dimension
00128     const sunindextype local_NOXP = tot_NOXP / DLx;
00129     const sunindextype local_NOYP = tot_NOYP / DLy;
00130     const sunindextype local_NOZP = tot_NOZP / DLz;
00131     // absolute positions of the first point in each dimension
00132     const sunindextype firstXPoint = local_NOXP * LIx;
00133     const sunindextype firstYPoint = local_NOYP * LIy;
00134     const sunindextype firstZPoint = local_NOZP * LIz;
00135     // total number of points in the patch
00136     const sunindextype local_NODP = dPD * local_NOXP * local_NOYP * local_NOZP;
00137
00138     // Set patch up with above derived quantities
00139     patchToMold.dx = envelopeLattice.get_dx();
00140     patchToMold.dy = envelopeLattice.get_dy();
00141     patchToMold.dz = envelopeLattice.get_dz();
00142     patchToMold.x0 = firstXPoint * patchToMold.dx;
00143     patchToMold.y0 = firstYPoint * patchToMold.dy;
00144     patchToMold.z0 = firstZPoint * patchToMold.dz;
00145     patchToMold.LIx = LIx;
00146     patchToMold.LIy = LIy;
00147     patchToMold.LIz = LIz;
00148     patchToMold.nx = local_NOXP;
00149     patchToMold.ny = local_NOYP;
00150     patchToMold.nz = local_NOZP;
00151     patchToMold.lx = patchToMold.nx * patchToMold.dx;
00152     patchToMold.ly = patchToMold.ny * patchToMold.dy;
00153     patchToMold.lz = patchToMold.nz * patchToMold.dz;
00154     /* Create and allocate memory for parallel vectors with defined local and
00155     * global lengths *
00156     * (-> CCode problem sizes Nlocal and N)
00157     * for field data and temporal derivatives and set extra pointers to them */
00158     patchToMold.u =
00159         N_VNew_Parallel(envelopeLattice.comm, local_NODP,
00160             envelopeLattice.get_tot_nodp(), envelopeLattice.sunctx);
00161     patchToMold.uData = NV_DATA_P(patchToMold.u);
00162     patchToMold.du =
00163         N_VNew_Parallel(envelopeLattice.comm, local_NODP,
00164             envelopeLattice.get_tot_nodp(), envelopeLattice.sunctx);
00165     patchToMold.duData = NV_DATA_P(patchToMold.du);
00166     // Allocate space for auxiliary uAux so that the lattice and all possible
00167     // directions of ghost layers fit
00168     const int s1 = patchToMold.nx, s2 = patchToMold.ny, s3 = patchToMold.nz;
00169     const int s_min = min(s1, min(s2, s3));
00170     patchToMold.uAux.resize(s1 * s2 * s3 / s_min * (s_min + 2 * gLW) * dPD);
00171     patchToMold.uAuxData = &patchToMold.uAux[0];
00172     patchToMold.envelopeLattice = &envelopeLattice;
00173     // Set patch "name" to process number -> only for debugging
00174     // patchToMold.ID=my_prc;
00175     // set flag
00176     patchToMold.statusFlags = FLatticePatchSetUp;
00177     patchToMold.generateTranslocationLookup();
00178     return 0;
00179 }

```

5.9.5 Field Documentation

5.9.5.1 buffData

array<sunrealtype *, 3> LatticePatch::buffData

pointer to spatial derivative data buffers

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

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

5.9.5.2 buffX

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

buffer to save spatial derivative values

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

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

5.9.5.3 buffY

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

buffer to save spatial derivative values

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

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

5.9.5.4 buffZ

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

buffer to save spatial derivative values

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

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

5.9.5.5 du

```
N_Vector LatticePatch::du
```

N_Vector for saving temporal derivatives of the field data.

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

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

5.9.5.6 duData

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

pointer to time-derivative data

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

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

5.9.5.7 dx

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

physical distance between lattice points in x-direction

Definition at line 166 of file [LatticePatch.h](#).

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

5.9.5.8 dy

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

physical distance between lattice points in y-direction

Definition at line 168 of file [LatticePatch.h](#).

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

5.9.5.9 dz

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

physical distance between lattice points in z-direction

Definition at line 170 of file [LatticePatch.h](#).

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

5.9.5.10 envelopeLattice

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

pointer to the enveloping lattice

Definition at line 172 of file [LatticePatch.h](#).

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

5.9.5.11 gCLData

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

pointer to halo data

Definition at line 220 of file [LatticePatch.h](#).

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

5.9.5.12 gCRData

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

pointer to halo data

Definition at line 220 of file [LatticePatch.h](#).

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

5.9.5.13 ghostCellLeft

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

buffer for passing ghost cell data

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

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

5.9.5.14 ghostCellLeftToSend

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

buffer for passing ghost cell data

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

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

5.9.5.15 ghostCellRight

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

buffer for passing ghost cell data

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

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

5.9.5.16 ghostCellRightToSend

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

buffer for passing ghost cell data

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

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

5.9.5.17 ghostCells

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

buffer for passing ghost cell data

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

5.9.5.18 ghostCellsToSend

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

buffer for passing ghost cell data

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

5.9.5.19 ID

```
int LatticePatch::ID
```

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

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

5.9.5.20 lgcTox

```
vector<int> LatticePatch::lgcTox [private]
```

ghost cell translocation lookup table

Definition at line 190 of file [LatticePatch.h](#).

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

5.9.5.21 lgcToy

```
vector<int> LatticePatch::lgcToy [private]
```

ghost cell translocation lookup table

Definition at line 190 of file [LatticePatch.h](#).

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

5.9.5.22 lgcToz

```
vector<int> LatticePatch::lgcToz [private]
```

ghost cell translocation lookup table

Definition at line 190 of file [LatticePatch.h](#).

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

5.9.5.23 Llx

```
sunindextype LatticePatch::Llx [private]
```

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

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

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

5.9.5.24 Lly

```
sunindextype LatticePatch::Lly [private]
```

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

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

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

5.9.5.25 Llz

```
sunindextype LatticePatch::Llz [private]
```

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

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

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

5.9.5.26 lx

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

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

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

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

5.9.5.27 ly

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

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

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

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

5.9.5.28 lz

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

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

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

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

5.9.5.29 nx

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

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

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

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

5.9.5.30 ny

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

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

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

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

5.9.5.31 nz

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

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

Definition at line 164 of file [LatticePatch.h](#).

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

5.9.5.32 rgcTox

```
vector<int> LatticePatch::rgcTox [private]
```

ghost cell translocation lookup table

Definition at line 190 of file [LatticePatch.h](#).

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

5.9.5.33 rgcToy

```
vector<int> LatticePatch::rgcToy [private]
```

ghost cell translocation lookup table

Definition at line 190 of file [LatticePatch.h](#).

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

5.9.5.34 rgcToz

```
vector<int> LatticePatch::rgcToz [private]
```

ghost cell translocation lookup table

Definition at line 190 of file [LatticePatch.h](#).

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

5.9.5.35 statusFlags

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

char for checking flags

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

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

5.9.5.36 u

```
N_Vector LatticePatch::u
```

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

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

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

5.9.5.37 uAux

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

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

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

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

5.9.5.38 uAuxData

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

pointer to auxiliary data vector

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

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

5.9.5.39 uData

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

pointer to field data

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

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

5.9.5.40 uTox

```
vector<int> LatticePatch::uTox [private]
```

translocation lookup table

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

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

5.9.5.41 uToy

```
vector<int> LatticePatch::uToy [private]
```

translocation lookup table

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

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

5.9.5.42 uToz

```
vector<int> LatticePatch::uToz [private]
```

translocation lookup table

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

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

5.9.5.43 x0

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

origin of the patch in physical space; x-coordinate

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

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

5.9.5.44 xTou

```
vector<int> LatticePatch::xTou [private]
```

translocation lookup table

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

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

5.9.5.45 y0

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

origin of the patch in physical space; y-coordinate

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

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

5.9.5.46 yTou

```
vector<int> LatticePatch::yTou [private]
```

translocation lookup table

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

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

5.9.5.47 z0

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

origin of the patch in physical space; z-coordinate

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

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

5.9.5.48 zTou

```
vector<int> LatticePatch::zTou [private]
```

translocation lookup table

Definition at line 175 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 string &dir)
function that creates folder to save simulation info
- void [outUState](#) (const int &state, const [LatticePatch](#) &latticePatch)
output function for the whole lattice
- string [getSimCode](#) ()
simCode getter function

Static Private Member Functions

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

Private Attributes

- string [simCode](#)
variable to save the SimCode generated at execution
- string [Path](#)
variable for the path to the output folder
- int [myPrc](#)
process ID

5.10.1 Detailed Description

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

Definition at line 27 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 9 of file [Outputters.cpp](#).

```
00009      {
00010      simCode = SimCodeGenerator();
00011      MPI_Comm_rank (MPI_COMM_WORLD, &myPrc);
00012  }
```

References [myPrc](#), [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 string & dir )
```

function that creates folder to save simulation info

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

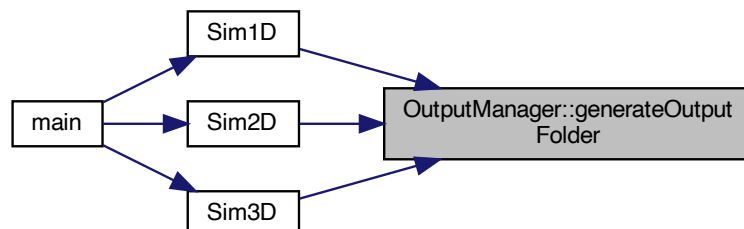
Definition at line 40 of file [Outputters.cpp](#).

```
00040 {
00041     // Do this only once for the first process
00042     if (myPrc == 0) {
00043         if (!fs::is_directory(dir))
00044             fs::create_directory(dir);
00045         if (!fs::is_directory(dir + "/SimResults"))
00046             fs::create_directory(dir + "/SimResults");
00047         if (!fs::is_directory(dir + "/SimResults/" + simCode))
00048             fs::create_directory(dir + "/SimResults/" + simCode);
00049     }
00050     // path variable for the output generation
00051     Path = dir + "/SimResults/" + simCode + "/";
00052
00053     ifstream fin("main.cpp");
00054     ofstream fout(Path + "config.txt");
00055     string line;
00056     int begin=1000;
00057     for (int i = 1; !fin.eof(); i++) {
00058         getline(fin, line);
00059         if (line.starts_with("    //----- B")) {
00060             begin=i;
00061         }
00062         if (i < begin) {
00063             continue;
00064         }
00065         fout << line << endl;
00066         if (line.starts_with("    //----- E")) {
00067             break;
00068         }
00069     }
00070
00071     return;
00072 }
```

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

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

Here is the caller graph for this function:



5.10.3.2 getSimCode()

```
string OutputManager::getSimCode ( )
```

simCode getter function

Return the date+time simulation identifier for logging.

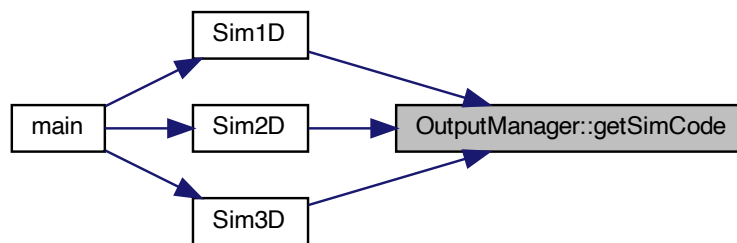
Definition at line 99 of file [Outputters.cpp](#).

```
00099 { 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 LatticePatch & latticePatch )
```

output function for the whole lattice

Write the field data to a csv file from each process (patch) with the field data into the simCode directory. The state (simulation step) denotes the prefix and the suffix after an underscore is given by the process/patch number

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

```
00078                                     {
00079     ofstream ofs;
00080     ofs.open(Path + to_string(state) + "_" + to_string(myPrc) + ".csv");
00081     // Set precision, number of digits for the values
00082     ofs << setprecision(numeric_limits<sunrealttype>::digits10);
00083
00084     // Walk through each lattice point
00085     for (int i = 0; i < latticePatch.discreteSize() * 6; i += 6) {
00086         // Six columns to contain the field data: Ex,Ey,Ez,Bx,By,Bz
00087         ofs << latticePatch.uData[i + 0] << "," << latticePatch.uData[i + 1] << "," <<
00088             << latticePatch.uData[i + 2] << "," << latticePatch.uData[i + 3] << "," <<
00089             << latticePatch.uData[i + 4] << "," << latticePatch.uData[i + 5]
00090         << endl;
00091     }
```

```

00092
00093     ofs.close();
00094
00095     return;
00096 }

```

References [LatticePatch::discreteSize\(\)](#), [myPrc](#), [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 SimCodeGenerator()

```
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 16 of file [Outputters.cpp](#).

```

00016     {
00017     const chrono::time_point<chrono::system_clock> now{
00018         chrono::system_clock::now()};
00019     const chrono::year_month_day ymd{chrono::floor<chrono::days>(now)};
00020     const auto tod = now - chrono::floor<chrono::days>(now);
00021     const chrono::hh_mm_ss hms{tod};
00022
00023     stringstream temp;
00024     temp << setfill('0') << setw(2)
00025         << static_cast<int>(ymd.year() - chrono::years(2000)) << "-"
00026         << setfill('0') << setw(2) << static_cast<unsigned>(ymd.month()) << "-"
00027         << setfill('0') << setw(2) << static_cast<unsigned>(ymd.day()) << "_"
00028         << setfill('0') << setw(2) << hms.hours().count() << "-" << setfill('0')
00029         << setw(2) << hms.minutes().count() << "-" << setfill('0') << setw(2)
00030         << hms.seconds().count();

```

```

00031  //« "_" « hms.subseconds().count(); // subseconds render the filename too
00032  //large
00033  return temp.str();
00034 }

```

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

Here is the caller graph for this function:



5.10.4 Field Documentation

5.10.4.1 myPrc

```
int OutputManager::myPrc [private]
```

process ID

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

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

5.10.4.2 Path

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

variable for the path to the output folder

Definition at line 34 of file [Outputters.h](#).

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

5.10.4.3 simCode

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

variable to save the SimCode generated at execution

Definition at line 32 of file [Outputters.h](#).

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

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

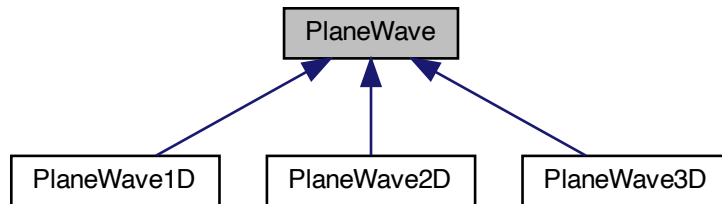
- [src/Outputters.h](#)
- [src/Outputters.cpp](#)

5.11 PlaneWave Class Reference

super-class for plane waves

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

Inheritance diagram for PlaneWave:



Protected Attributes

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

5.11.1 Detailed Description

super-class for plane waves

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

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

5.11.2 Field Documentation

5.11.2.1 k_x

`sunrealtype PlaneWave::kx [protected]`

wavenumber k_x

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

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

5.11.2.2 k_y

`sunrealtype PlaneWave::ky [protected]`

wavenumber k_y

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

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

5.11.2.3 k_z

`sunrealtype PlaneWave::kz [protected]`

wavenumber k_z

Definition at line 32 of file [ICSetters.h](#).

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

5.11.2.4 ϕ_x

`sunrealtype PlaneWave::phix [protected]`

phase shift in x-direction, ϕ_x

Definition at line 40 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 42 of file [ICSetters.h](#).

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

5.11.2.6 phiz

sunrealtype PlaneWave::phiz [protected]

phase shift in z-direction, ϕ_z

Definition at line 44 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 34 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 36 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 38 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

- vector< sunrealtype > [k](#)
- vector< sunrealtype > [p](#)
- vector< sunrealtype > [phi](#)

5.12.1 Detailed Description

plane wave structure

Definition at line 18 of file [SimulationFunctions.h](#).

5.12.2 Field Documentation

5.12.2.1 k

`vector<sunrealtype> planewave::k`

wavevector (normalized to $1/\lambda$)

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

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

5.12.2.2 p

```
vector<sunrealtype> planewave::p
```

amplitude & polarization vector

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

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

5.12.2.3 phi

```
vector<sunrealtype> planewave::phi
```

phase shift

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

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

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

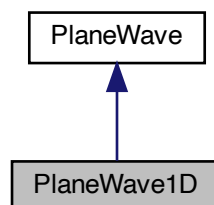
- [src/SimulationFunctions.h](#)

5.13 PlaneWave1D Class Reference

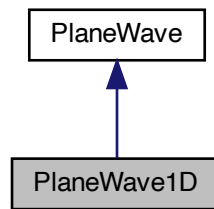
class for plane waves in 1D

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

Inheritance diagram for PlaneWave1D:



Collaboration diagram for PlaneWave1D:



Public Member Functions

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

Additional Inherited Members

5.13.1 Detailed Description

class for plane waves in 1D

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

5.13.2 Constructor & Destructor Documentation

5.13.2.1 PlaneWave1D()

```

PlaneWave1D::PlaneWave1D (
    vector< sunrealtype > k = {1, 0, 0},
    vector< sunrealtype > p = {0, 0, 1},
    vector< sunrealtype > 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 12 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](#).

```
00028 {
00029     const sunrealtype wavelength =
00030         sqrt(kx * kx + ky * ky + kz * kz); /** \f$ 1/\lambda \f$ */
00031     const sunrealtype kScalarX = (kx * x + ky * y + kz * z) * 2 *
00032         numbers::pi; /** \f$ 2\pi \cdot \vec{k} \cdot \vec{x} \f$ */
00033     // Plane wave definition
00034     const array<sunrealtype, 3> E{{ /** E-field vector */
00035         px * cos(kScalarX - phix), /** \f$ E_x \f$ */
00036         py * cos(kScalarX - phiy), /** \f$ E_y \f$ */
00037         pz * cos(kScalarX - phiz)}}; /** \f$ E_z \f$ */
00038     // Put E-field into space
00039     pTo6Space[0] += E[0];
00040     pTo6Space[1] += E[1];
00041     pTo6Space[2] += E[2];
00042     // and B-field
00043     pTo6Space[3] += (ky * E[2] - kz * E[1]) / wavelength;
```

```
00044   pTo6Space[4] += (kz * E[0] - kx * E[2]) / wavelength;  
00045   pTo6Space[5] += (kx * E[1] - ky * E[0]) / wavelength;  
00046 }
```

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:

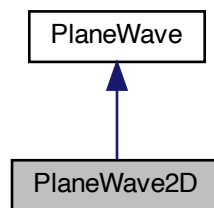
- [src/ICSetters.h](#)
- [src/ICSetters.cpp](#)

5.14 PlaneWave2D Class Reference

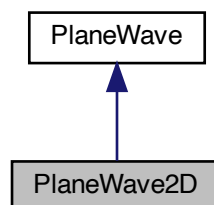
class for plane waves in 2D

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

Inheritance diagram for PlaneWave2D:



Collaboration diagram for PlaneWave2D:



Public Member Functions

- [PlaneWave2D](#) (vector< sunrealtype > k={1, 0, 0}, vector< sunrealtype > p={0, 0, 1}, vector< sunrealtype > phi={0, 0, 0})
construction with default parameters
- void [addToSpace](#) (sunrealtype x, sunrealtype y, sunrealtype z, sunrealtype *pTo6Space) const
function for the actual implementation in the lattice

Additional Inherited Members

5.14.1 Detailed Description

class for plane waves in 2D

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

5.14.2 Constructor & Destructor Documentation

5.14.2.1 PlaneWave2D()

```
PlaneWave2D::PlaneWave2D (
    vector< sunrealtype > k = {1, 0, 0},
    vector< sunrealtype > p = {0, 0, 1},
    vector< sunrealtype > 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 49 of file [ICSetters.cpp](#).

```
00050 {
00051     kx = k[0]; /** - wavevectors \f$ k_x \f$ */
00052     ky = k[1]; /** - \f$ k_y \f$ */
00053     kz = k[2]; /** - \f$ k_z \f$ normalized to \f$ 1/\lambda \f$ */
00054     // Amplitude bug: lower by factor 9
00055     px = p[0] / 9; /** - amplitude (polarization) in x-direction \f$ p_x \f$ */
00056     py = p[1] / 9; /** - amplitude (polarization) in y-direction \f$ p_y \f$ */
00057     pz = p[2] / 9; /** - amplitude (polarization) in z-direction \f$ p_z \f$ */
00058     phix = phi[0]; /** - phase shift in x-direction \f$ \phi_x \f$ */
00059     phiy = phi[1]; /** - phase shift in y-direction \f$ \phi_y \f$ */
00060     phiz = phi[2]; /** - phase shift in z-direction \f$ \phi_z \f$ */
00061 }
```

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 64 of file [ICSetters.cpp](#).

```
00065                                     {
00066     const sunrealtype wavelength =
00067         sqrt(kx * kx + ky * ky + kz * kz); /* \f$ 1/\lambda \f$ */
00068     const sunrealtype kScalarX = (kx * x + ky * y + kz * z) * 2 *
00069         numbers::pi; /* \f$ 2\pi \ \vec{k} \cdot \vec{x} \f$ */
00070     // Plane wave definition
00071     const array<sunrealtype, 3> E{{ /* E-field vector */
00072         px * cos(kScalarX - phix), /* \f$ E_x \f$ */
00073         py * cos(kScalarX - phiy), /* \f$ E_y \f$ */
00074         pz * cos(kScalarX - phiz)}}; /* \f$ E_z \f$ */
00075     // Put E-field into space
00076     pTo6Space[0] += E[0];
00077     pTo6Space[1] += E[1];
00078     pTo6Space[2] += E[2];
00079     // and B-field
00080     pTo6Space[3] += (ky * E[2] - kz * E[1]) / wavelength;
00081     pTo6Space[4] += (kz * E[0] - kx * E[2]) / wavelength;
00082     pTo6Space[5] += (kx * E[1] - ky * E[0]) / wavelength;
00083 }
```

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:

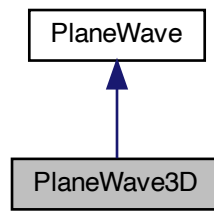
- [src/ICSetters.h](#)
- [src/ICSetters.cpp](#)

5.15 PlaneWave3D Class Reference

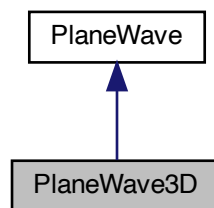
class for plane waves in 3D

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

Inheritance diagram for PlaneWave3D:



Collaboration diagram for PlaneWave3D:



Public Member Functions

- [PlaneWave3D](#) (vector< sunrealtype > k={1, 0, 0}, vector< sunrealtype > p={0, 0, 1}, vector< sunrealtype > phi={0, 0, 0})
construction with default parameters
- void [addToSpace](#) (sunrealtype x, sunrealtype y, sunrealtype z, sunrealtype *pTo6Space) const
function for the actual implementation in space

Additional Inherited Members

5.15.1 Detailed Description

class for plane waves in 3D

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

5.15.2 Constructor & Destructor Documentation

5.15.2.1 PlaneWave3D()

```
PlaneWave3D::PlaneWave3D (
    vector< sunrealtype > k = {1, 0, 0},
    vector< sunrealtype > p = {0, 0, 1},
    vector< sunrealtype > 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 86 of file [ICSetters.cpp](#).

```
00087 {
00088     kx = k[0];      /** - wavevectors \f$ k_x \f$ */
00089     ky = k[1];      /** - \f$ k_y \f$ */
00090     kz = k[2];      /** - \f$ k_z \f$ normalized to \f$ 1/\lambda \f$ */
00091     px = p[0];      /** - amplitude (polarization) in x-direction \f$ p_x \f$ */
00092     py = p[1];      /** - amplitude (polarization) in y-direction \f$ p_y \f$ */
00093     pz = p[2];      /** - amplitude (polarization) in z-direction \f$ p_z \f$ */
00094     phix = phi[0];  /** - phase shift in x-direction \f$ \phi_x \f$ */
00095     phiy = phi[1];  /** - phase shift in y-direction \f$ \phi_y \f$ */
00096     phiz = phi[2];  /** - phase shift in z-direction \f$ \phi_z \f$ */
00097 }
```

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 100 of file [ICSetters.cpp](#).

```
00101                                     {
00102     const sunrealtype wavelength =
00103         sqrt(kx * kx + ky * ky + kz * kz); /* \f$ 1/\lambda \f$ */
00104     const sunrealtype kScalarX = (kx * x + ky * y + kz * z) * 2 *
00105         numbers::pi; /* \f$ 2\pi \cdot \vec{k} \cdot \vec{x} \f$ */
00106     // Plane wave definition
00107     const array<sunrealtype, 3> E{ /* E-field vector \f$ \vec{E} \f$ */
00108         px * cos(kScalarX - phix), /* \f$ E_x \f$ */
00109         py * cos(kScalarX - phiy), /* \f$ E_y \f$ */
00110         pz * cos(kScalarX - phiz) }; /* \f$ E_z \f$ */
00111     // Put E-field into space
00112     pTo6Space[0] += E[0];
00113     pTo6Space[1] += E[1];
00114     pTo6Space[2] += E[2];
00115     // and B-field
00116     pTo6Space[3] += (ky * E[2] - kz * E[1]) / wavelength;
00117     pTo6Space[4] += (kz * E[0] - kx * E[2]) / wavelength;
00118     pTo6Space[5] += (kx * E[1] - ky * E[0]) / wavelength;
00119 }
```

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:

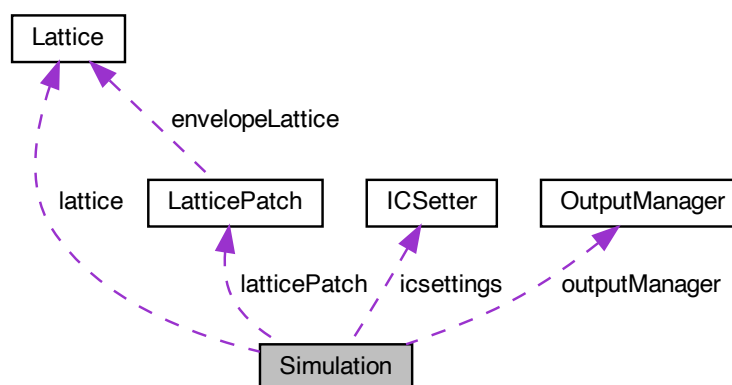
- [src/ICSetters.h](#)
- [src/ICSetters.cpp](#)

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 CNode memory and Sundials context
- MPI_Comm * [get_cart_comm](#) ()
Reference to the cartesian communicator of the lattice -> for debugging.
- void [setDiscreteDimensionsOfLattice](#) (const sunindextype _tot_nx, const sunindextype _tot_ny, const sunindextype _tot_nz)
function to set discrete dimensions of the lattice
- void [setPhysicalDimensionsOfLattice](#) (const sunrealtype lx, const sunrealtype ly, const sunrealtype lz)
function to set physical dimensions of the lattice
- void [initializePatchwork](#) (const int nx, const int ny, const int nz)
function to initialize the Patchwork
- void [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 int xm, const int ym, const int 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 generate Output of the whole field at a given time
- void [checkFlag](#) (unsigned int flag) const
function to check that a flag has been set and if not print an error
- void [checkNoFlag](#) (unsigned int flag) const
function to check that if flag has not been set and if print an error

Data Fields

- [ICSetter icsettings](#)
IC Setter object.
- [OutputManager outputManager](#)
Output Manager object.
- void * [cnode_mem](#)
Pointer to CNode memory object – public to avoid cross library errors.

Private Attributes

- [Lattice](#) `lattice`
Lattice object.
- [LatticePatch](#) `latticePatch`
LatticePatch object.
- `sunrealtype t`
current time of the simulation
- `unsigned char` [statusFlags](#)
char for checking simulation flags

5.16.1 Detailed Description

[Simulation](#) class to instantiate the whole walkthrough of a [Simulation](#).

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

5.16.2 Constructor & Destructor Documentation

5.16.2.1 Simulation()

```
Simulation::Simulation (
    const int nx,
    const int ny,
    const int nz,
    const int StencilOrder,
    const bool periodicity )
```

constructor function for the creation of the cartesian communicator

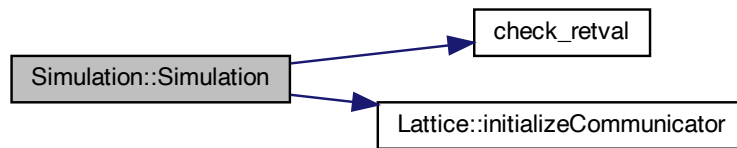
Along with the simulation object, create the cartesian communicator and SUNContext object

Definition at line 14 of file [SimulationClass.cpp](#).

```
00015                                     :
00016     lattice(StencilOrder){
00017     statusFlags = 0;
00018     t = 0;
00019     // Initialize the cartesian communicator
00020     lattice.initializeCommunicator(nx, ny, nz, periodicity);
00021
00022     // Create the SUNContext object associated with the thread of execution
00023     int retval = 0;
00024     retval = SUNContext_Create(&lattice.comm, &lattice.sunctx);
00025     if (check_retval(&retval, "SUNContext_Create", 1, lattice.my_prc))
00026         MPI_Abort(lattice.comm, 1);
00027     // if (flag != CV_SUCCESS) { printf("SUNContext_Create failed, flag=%d.\n",
00028     // flag);
00029     //     MPI_Abort(lattice.comm, 1); }
00030 }
```

References [check_retval\(\)](#), [Lattice::comm](#), [Lattice::initializeCommunicator\(\)](#), [lattice](#), [Lattice::my_prc](#), [statusFlags](#), [Lattice::sunctx](#), and [t](#).

Here is the call graph for this function:



5.16.2.2 ~Simulation()

```
Simulation::~~Simulation ( )
```

destructor function freeing CCode memory and Sundials context

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

Definition at line 34 of file [SimulationClass.cpp](#).

```

00034 {
00035     // Free solver memory
00036     if (statusFlags & CcodeObjectSetUp) {
00037         // PrintFinalStats(ccode_mem); // TODO write this function as in ccodes
00038         // cvAdvDiff_bnd.c SUNDIALS_MARK_FUNCTION_END(lattice.profobj);
00039         CcodeFree(&ccode_mem);
00040         SUNContext_Free(&lattice.sunctx);
00041     }
00042 }
```

References [ccode_mem](#), [CcodeObjectSetUp](#), [lattice](#), [statusFlags](#), and [Lattice::sunctx](#).

5.16.3 Member Function Documentation

5.16.3.1 addInitialConditions()

```

void Simulation::addInitialConditions (
    const int xm,
    const int ym,
    const int zm = 0 )
```

functions to add initial periodic field configurations

Use parameters to add periodic IC layers.

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

```

00186 {
00187     const sunrealtype dx = latticePatch.getDelta(1);
00188     const sunrealtype dy = latticePatch.getDelta(2);
```

```

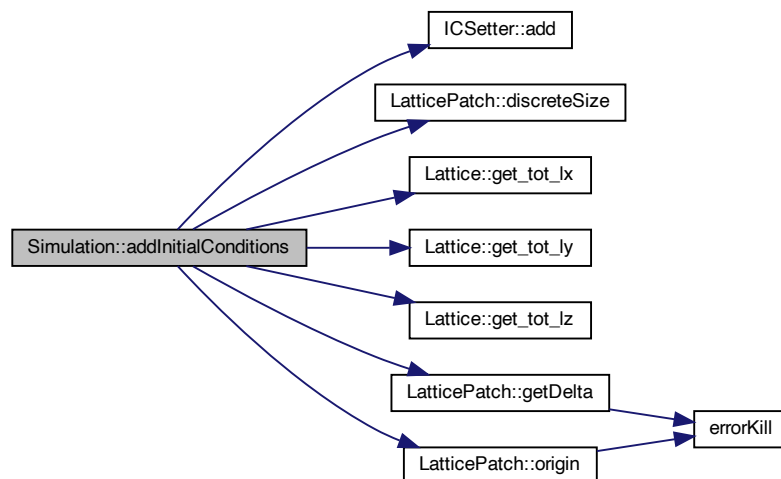
00189  const sunrealtype dz = latticePatch.getDelta(3);
00190  const int nx = latticePatch.discreteSize(1);
00191  const int ny = latticePatch.discreteSize(2);
00192  // Correct for demanded displacement, rest as for setInitialConditions
00193  const sunrealtype x0 = latticePatch.origin(1) + xm*lattice.get_tot_lx();
00194  const sunrealtype y0 = latticePatch.origin(2) + ym*lattice.get_tot_ly();
00195  const sunrealtype z0 = latticePatch.origin(3) + zm*lattice.get_tot_lz();
00196  int px = 0, py = 0, pz = 0;
00197  for (int i = 0; i < latticePatch.discreteSize() * 6; i += 6) {
00198      px = (i / 6) % nx;
00199      py = ((i / 6) / nx) % ny;
00200      pz = ((i / 6) / nx) / ny;
00201      icsettings.add(static_cast<sunrealtype>(px) * dx + x0,
00202                    static_cast<sunrealtype>(py) * dy + y0,
00203                    static_cast<sunrealtype>(pz) * dz + z0, &latticePatch.uData[i]);
00204  }
00205  return;
00206 }

```

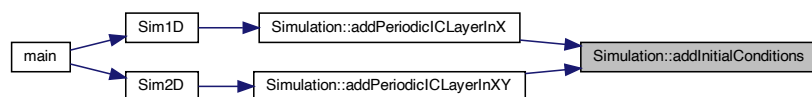
References [ICSetter::add\(\)](#), [LatticePatch::discreteSize\(\)](#), [Lattice::get_tot_lx\(\)](#), [Lattice::get_tot_ly\(\)](#), [Lattice::get_tot_lz\(\)](#), [LatticePatch::getDelta\(\)](#), [icsettings](#), [lattice](#), [latticePatch](#), [LatticePatch::origin\(\)](#), and [LatticePatch::uData](#).

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.

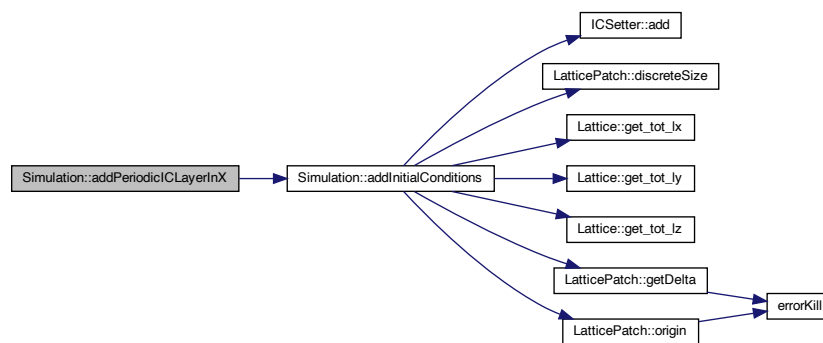
Definition at line 209 of file [SimulationClass.cpp](#).

```
00209 {
00210     addInitialConditions(-1, 0, 0);
00211     addInitialConditions(1, 0, 0);
00212     return;
00213 }
```

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.

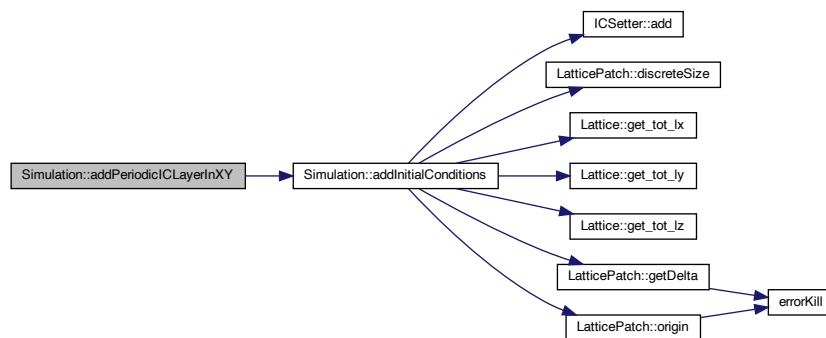
Definition at line 216 of file [SimulationClass.cpp](#).

```
00216 {
00217     addInitialConditions(-1, -1, 0);
00218     addInitialConditions(-1, 0, 0);
00219     addInitialConditions(-1, 1, 0);
00220     addInitialConditions(0, 1, 0);
00221     addInitialConditions(0, -1, 0);
00222     addInitialConditions(1, -1, 0);
00223     addInitialConditions(1, 0, 0);
00224     addInitialConditions(1, 1, 0);
00225     return;
00226 }
```

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.

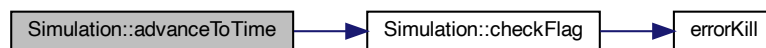
Definition at line 229 of file [SimulationClass.cpp](#).

```
00229 {
00230     checkFlag(SimulationStarted);
00231     int flag = 0;
00232     flag = CCode(cvode_mem, tEnd, latticePatch.u, &t,
00233                 CV_NORMAL); // CV_NORMAL: internal steps to reach tEnd, then
00234                             // interpolate to return latticePatch.u, return time
00235                             // reached by the solver as t
00236     if (flag != CV_SUCCESS)
00237         printf("CCode failed, flag=%d.\n", flag);
00238 }
```

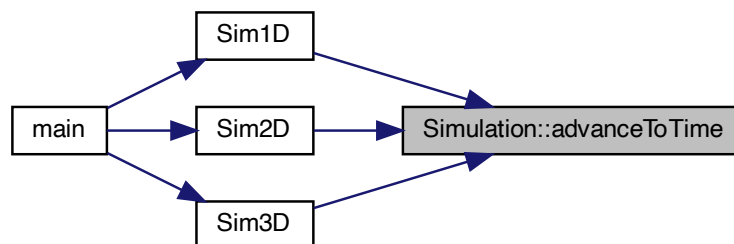
References [checkFlag\(\)](#), [cvode_mem](#), [latticePatch](#), [SimulationStarted](#), [t](#), and [LatticePatch::u](#).

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 that a flag has been set and if not print an error

Check the presence configuration flags.

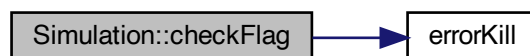
Definition at line 247 of file [SimulationClass.cpp](#).

```
00247 {
00248     if (!(statusFlags & flag)) {
00249         string errorMessage;
00250         switch (flag) {
00251             case LatticeDiscreteSetUp:
00252                 errorMessage = "The discrete size of the Simulation has not been set up";
00253                 break;
00254             case LatticePhysicalSetUp:
00255                 errorMessage = "The physical size of the Simulation has not been set up";
00256                 break;
00257             case LatticePatchworkSetUp:
00258                 errorMessage = "The patchwork for the Simulation has not been set up";
00259                 break;
00260             case CvodeObjectSetUp:
00261                 errorMessage = "The CVODE object has not been initialized";
00262                 break;
00263             case SimulationStarted:
00264                 errorMessage = "The Simulation has not been started";
00265                 break;
00266             default:
00267                 errorMessage = "Uppss, you've made a non-standard error, sadly I can't "
00268                               "help you there";
00269                 break;
00270         }
00271         errorKill(errorMessage);
00272     }
00273     return;
00274 }
```

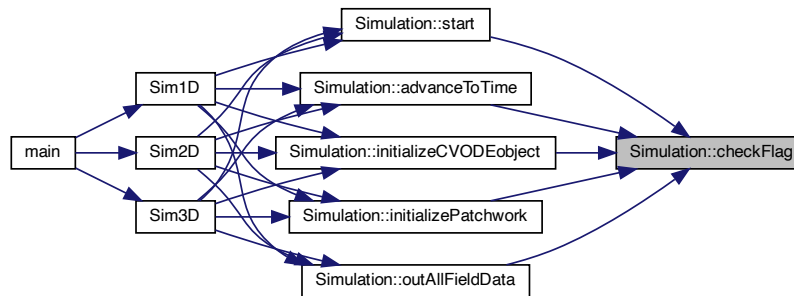
References [CvodeObjectSetUp](#), [errorKill\(\)](#), [LatticeDiscreteSetUp](#), [LatticePatchworkSetUp](#), [LatticePhysicalSetUp](#), [SimulationStarted](#), and [statusFlags](#).

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 that if flag has not been set and if print an error

Check the absence of configuration flags.

Definition at line 277 of file [SimulationClass.cpp](#).

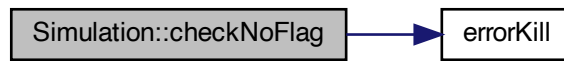
```

00277     {
00278     if ((statusFlags & flag)) {
00279         string errorMessage;
00280         switch (flag) {
00281         case LatticeDiscreteSetUp:
00282             errorMessage =
00283                 "The discrete size of the Simulation has already been set up";
00284             break;
00285         case LatticePhysicalSetUp:
00286             errorMessage =
00287                 "The physical size of the Simulation has already been set up";
00288             break;
00289         case LatticePatchworkSetUp:
00290             errorMessage = "The patchwork for the Simulation has already been set up";
00291             break;
00292         case CvodeObjectSetUp:
00293             errorMessage = "The CVODE object has already been initialized";
00294             break;
00295         case SimulationStarted:
00296             errorMessage = "The simulation has already started, some changes are no "
00297                 "longer possible";
00298             break;
00299         default:
00300             errorMessage = "Uppss, you've made a non-standard error, sadly I can't "
00301                 "help you there";
00302             break;
00303         }
00304         errorKill(errorMessage);
00305     }
00306     return;
00307 }
```

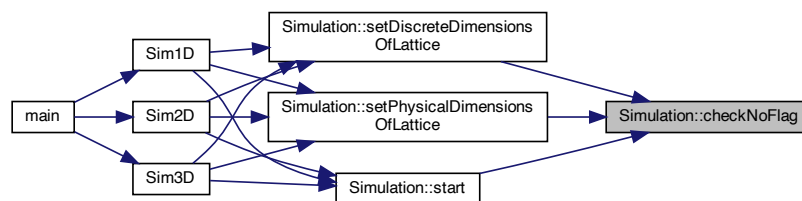
References [CvodeObjectSetUp](#), [errorKill\(\)](#), [LatticeDiscreteSetUp](#), [LatticePatchworkSetUp](#), [LatticePhysicalSetUp](#), [SimulationStarted](#), and [statusFlags](#).

Referenced by [setDiscreteDimensionsOfLattice\(\)](#), [setPhysicalDimensionsOfLattice\(\)](#), and [start\(\)](#).

Here is the call graph for this function:



Here is the caller graph for this function:



5.16.3.7 get_cart_comm()

```
MPI_Comm * Simulation::get_cart_comm ( ) [inline]
```

Reference to the cartesian communicator of the lattice -> for debugging.

Definition at line 63 of file [SimulationClass.h](#).

```
00063 { return &lattice.comm; };
```

References [Lattice::comm](#), and [lattice](#).

5.16.3.8 initializeCVODEobject()

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

function to initialize the CVODE object with all requirements

Configure CVODE.

Definition at line 74 of file [SimulationClass.cpp](#).

```
00075 {
00076     checkFlag(SimulationStarted);
```

```

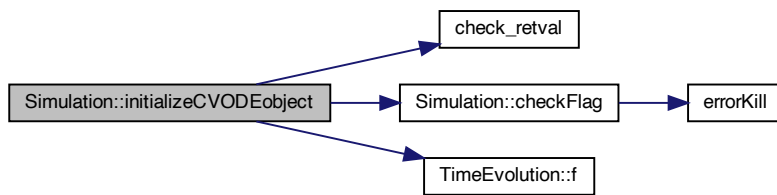
00077
00078 // CCode settings return value
00079 int retval = 0;
00080
00081 // Set the profiler
00082 retval = SUNContext_GetProfiler(lattice.sunctx, &lattice.profbj);
00083 if (check_retval(&retval, "SUNContext_GetProfiler", 1, lattice.my_prc))
00084     MPI_Abort(lattice.comm, 1);
00085 // if (flag != CV_SUCCESS) { printf("SUNContext_GetProfiler failed,
00086 // flag=%d.\n", flag);
00087 //     MPI_Abort(lattice.comm, 1); }
00088
00089 // SUNDIALS_MARK_FUNCTION_BEGIN(profbj);
00090
00091 // Create CCode object - returns a pointer to the ccode memory structure
00092 // with Adams method (Adams-Moulton formula) solver chosen for non-stiff ODE
00093 ccode_mem = CCodeCreate(CV_ADAMS, lattice.sunctx);
00094
00095 // Specify user data and attach it to the main ccode memory block
00096 retval = CCodeSetUserData(
00097     ccode_mem,
00098     &latticePatch); // patch contains the user data as used in CVRhsFn
00099 if (check_retval(&retval, "CCodeSetUserData", 1, lattice.my_prc))
00100     MPI_Abort(lattice.comm, 1);
00101 // if (flag != CV_SUCCESS) { printf("CCodeSetUserData failed, flag=%d.\n",
00102 // flag);
00103 //     MPI_Abort(lattice.comm, 1); }
00104
00105 // Initialize CCode solver -> can only be called after start of simulation to
00106 // have data ready Provide required problem and solution specifications,
00107 // allocate internal memory, and initialize ccode
00108 retval = CCodeInit(ccode_mem, TimeEvolution::f, 0,
00109     latticePatch.u); // allocate memory, CVRhsFn f, t_i=0, u
00110 // contains the initial values
00111 if (check_retval(&retval, "CCodeInit", 1, lattice.my_prc))
00112     MPI_Abort(lattice.comm, 1);
00113 // if (flag != CV_SUCCESS) { printf("CCodeInit failed, flag=%d.\n", flag);
00114 //     MPI_Abort(lattice.comm, 1); }
00115
00116 // Create fixed point nonlinear solver object (suitable for non-stiff ODE) and
00117 // attach it to CCode
00118 SUNNonlinearSolver NLS =
00119     SUNNonlinSol_FixedPoint(latticePatch.u, 0, lattice.sunctx);
00120 retval = CCodeSetNonlinearSolver(ccode_mem, NLS);
00121 if (check_retval(&retval, "CCodeSetNonlinearSolver", 1, lattice.my_prc))
00122     MPI_Abort(lattice.comm, 1);
00123 // if (flag != CV_SUCCESS) { printf("CCodeSetNonlinearSolver failed,
00124 // flag=%d.\n", flag);
00125 //     MPI_Abort(lattice.comm, 1); }
00126
00127 // Specify the maximum number of steps to be taken by the solver in its
00128 // attempt to reach the next output time
00129 retval = CCodeSetMaxNumSteps(ccode_mem, 10000);
00130 if (check_retval(&retval, "CCodeSetMaxNumSteps", 1, lattice.my_prc))
00131     MPI_Abort(lattice.comm, 1);
00132 // if (flag != CV_SUCCESS) { printf("CCodeSetMaxNumSteps failed, flag=%d.\n",
00133 // flag);
00134 //     MPI_Abort(lattice.comm, 1); }
00135
00136 // Specify integration tolerances - a scalar relative tolerance and scalar
00137 // absolute tolerance
00138 retval = CCodeSStolerances(ccode_mem, reltol, abstol);
00139 if (check_retval(&retval, "CCodeSStolerances", 1, lattice.my_prc))
00140     MPI_Abort(lattice.comm, 1);
00141 // if (flag != CV_SUCCESS) { printf("CCodeSStolerances failed, flag=%d.\n",
00142 // flag);
00143 //     MPI_Abort(lattice.comm, 1); }
00144
00145 statusFlags |= CcodeObjectSetUp;
00146 }

```

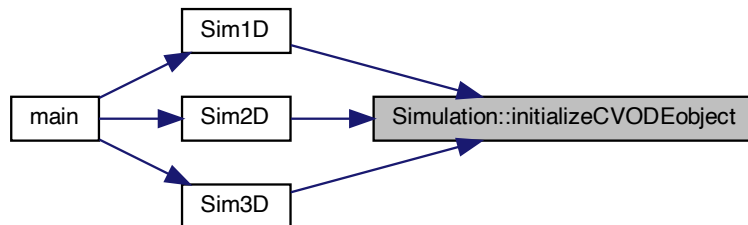
References [check_retval\(\)](#), [checkFlag\(\)](#), [Lattice::comm](#), [ccode_mem](#), [CcodeObjectSetUp](#), [TimeEvolution::f\(\)](#), [lattice](#), [latticePatch](#), [Lattice::my_prc](#), [Lattice::profbj](#), [SimulationStarted](#), [statusFlags](#), [Lattice::sunctx](#), and [LatticePatch::u](#).

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

Here is the call graph for this function:



Here is the caller graph for this function:



5.16.3.9 initializePatchwork()

```

void Simulation::initializePatchwork (
    const int nx,
    const int ny,
    const int nz )
  
```

function to initialize the Patchwork

Check that the lattice dimensions are set up and generate the patchwork.

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

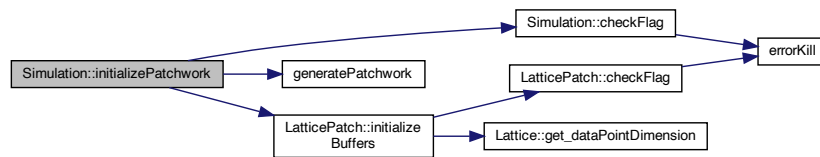
```

00062     {
00063         checkFlag(LatticeDiscreteSetUp);
00064         checkFlag(LatticePhysicalSetUp);
00065
00066         // Generate the patchwork
00067         generatePatchwork(lattice, latticePatch, nx, ny, nz);
00068         latticePatch.initializeBuffers();
00069
00070         statusFlags |= LatticePatchworkSetUp;
00071     }
  
```

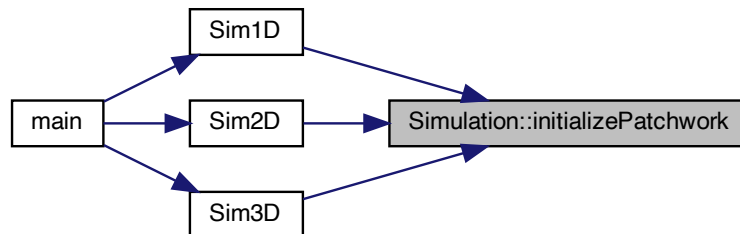
References [checkFlag\(\)](#), [generatePatchwork\(\)](#), [LatticePatch::initializeBuffers\(\)](#), [lattice](#), [LatticeDiscreteSetUp](#), [latticePatch](#), [LatticePatchworkSetUp](#), [LatticePhysicalSetUp](#), and [statusFlags](#).

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

Here is the call graph for this function:



Here is the caller graph for this function:



5.16.3.10 outAllFieldData()

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

function to generate Output of the whole field at a given time

Write specified simulations steps to disk.

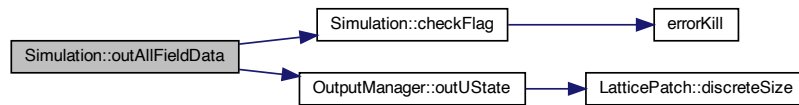
Definition at line 241 of file [SimulationClass.cpp](#).

```
00241 {
00242     checkFlag(SimulationStarted);
00243     outputManager.outUState(state, latticePatch);
00244 }
```

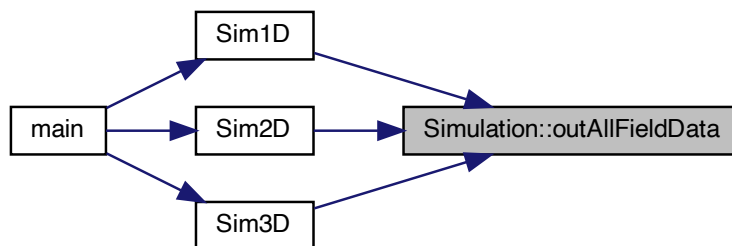
References [checkFlag\(\)](#), [latticePatch](#), [outputManager](#), [OutputManager::outUState\(\)](#), and [SimulationStarted](#).

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

Here is the call graph for this function:



Here is the caller graph for this function:



5.16.3.11 setDiscreteDimensionsOfLattice()

```

void Simulation::setDiscreteDimensionsOfLattice (
    const sunindextype _tot_nx,
    const sunindextype _tot_ny,
    const sunindextype _tot_nz )
  
```

function to set discrete dimensions of the lattice

Set the discrete dimensions, the number of points per dimension.

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

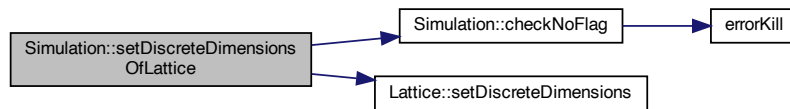
```

00046                                     {
00047     checkNoFlag(LatticePatchworkSetUp);
00048     lattice.setDiscreteDimensions(nx, ny, nz);
00049     statusFlags |= LatticeDiscreteSetUp;
00050 }
  
```

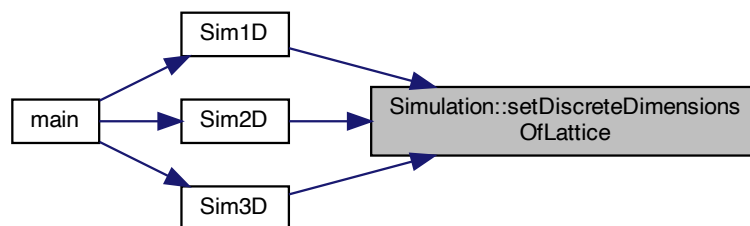
References [checkNoFlag\(\)](#), [lattice](#), [LatticeDiscreteSetUp](#), [LatticePatchworkSetUp](#), [Lattice::setDiscreteDimensions\(\)](#), and [statusFlags](#).

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

Here is the call graph for this function:



Here is the caller graph for this function:



5.16.3.12 setInitialConditions()

```
void Simulation::setInitialConditions ( )
```

functions to set the initial field configuration onto the lattice

Set initial conditions: Fill the lattice points with the initial field values

Definition at line 161 of file [SimulationClass.cpp](#).

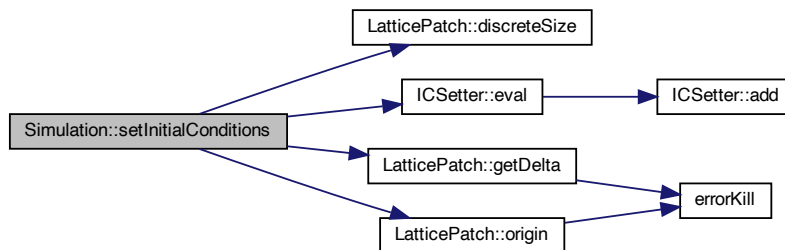
```

00161     {
00162     const sunrealtype dx = latticePatch.getDelta(1);
00163     const sunrealtype dy = latticePatch.getDelta(2);
00164     const sunrealtype dz = latticePatch.getDelta(3);
00165     const int nx = latticePatch.discreteSize(1);
00166     const int ny = latticePatch.discreteSize(2);
00167     const sunrealtype x0 = latticePatch.origin(1);
00168     const sunrealtype y0 = latticePatch.origin(2);
00169     const sunrealtype z0 = latticePatch.origin(3);
00170     int px = 0, py = 0, pz = 0;
00171     // space coordinates
00172     for (int i = 0; i < latticePatch.discreteSize() * 6; i += 6) {
00173         px = (i / 6) % nx;
00174         py = ((i / 6) / nx) % ny;
00175         pz = ((i / 6) / nx) / ny;
00176         // Call the 'eval' function to fill the lattice points with the field data
00177         icsettings.eval(static_cast<sunrealtype>(px) * dx + x0,
00178                        static_cast<sunrealtype>(py) * dy + y0,
00179                        static_cast<sunrealtype>(pz) * dz + z0, &latticePatch.uData[i]);
00180     }
00181     return;
00182 }
  
```

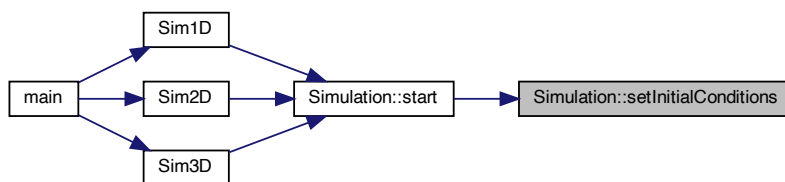

References [LatticePatch::discreteSize\(\)](#), [ICSetter::eval\(\)](#), [LatticePatch::getDelta\(\)](#), [icsettings](#), [latticePatch](#), [LatticePatch::origin\(\)](#), and [LatticePatch::uData](#).

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

Here is the call graph for this function:



Here is the caller graph for this function:



5.16.3.13 setPhysicalDimensionsOfLattice()

```

void Simulation::setPhysicalDimensionsOfLattice (
    const sunrealtype lx,
    const sunrealtype ly,
    const sunrealtype lz )
  
```

function to set physical dimensions of the lattice

Set the physical dimensions with lengths in micro meters.

Definition at line 53 of file [SimulationClass.cpp](#).

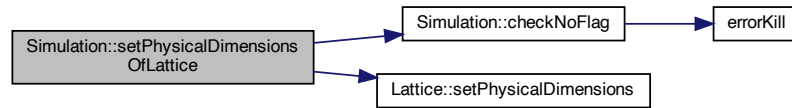
```

00054 {
00055     checkNoFlag(LatticePatchworkSetUp);
00056     lattice.setPhysicalDimensions(lx, ly, lz);
00057     statusFlags |= LatticePhysicalSetUp;
00058 }
  
```

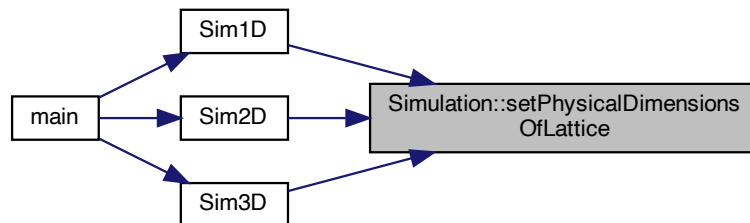
References [checkNoFlag\(\)](#), [lattice](#), [LatticePatchworkSetUp](#), [LatticePhysicalSetUp](#), [Lattice::setPhysicalDimensions\(\)](#), and [statusFlags](#).

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

Here is the call graph for this function:



Here is the caller graph for this function:



5.16.3.14 start()

```
void Simulation::start ( )
```

function to start the simulation for time iteration

Check if the lattice patchwork is set up and set the initial conditions.

Definition at line 149 of file [SimulationClass.cpp](#).

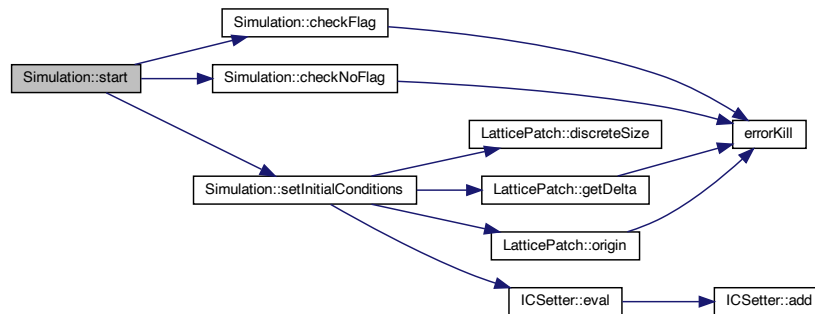
```

00149         {
00150     checkFlag(LatticeDiscreteSetUp);
00151     checkFlag(LatticePhysicalSetUp);
00152     checkFlag(LatticePatchworkSetUp);
00153     checkNoFlag(SimulationStarted);
00154     checkNoFlag(CvodeObjectSetUp);
00155     setInitialConditions();
00156     statusFlags |= SimulationStarted;
00157 }
  
```

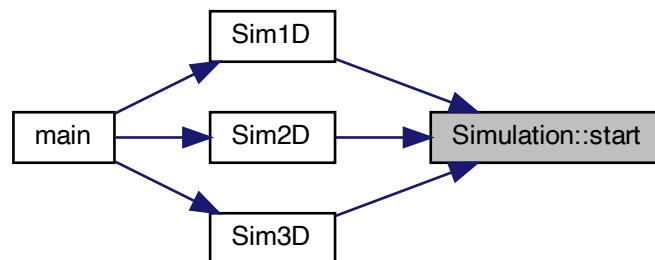
References [checkFlag\(\)](#), [checkNoFlag\(\)](#), [CvodeObjectSetUp](#), [LatticeDiscreteSetUp](#), [LatticePatchworkSetUp](#), [LatticePhysicalSetUp](#), [setInitialConditions\(\)](#), [SimulationStarted](#), and [statusFlags](#).

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 ccode_mem

```
void* Simulation::ccode_mem
```

Pointer to CCode memory object – public to avoid cross library errors.

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

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

5.16.4.2 icsettings

`ICSetter` `Simulation::icsettings`

IC Setter object.

Definition at line 52 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 42 of file [SimulationClass.h](#).

Referenced by [addInitialConditions\(\)](#), [get_cart_comm\(\)](#), [initializeCVODEobject\(\)](#), [initializePatchwork\(\)](#), [setDiscreteDimensionsOfLattice\(\)](#), [setPhysicalDimensionsOfLattice\(\)](#), [Simulation\(\)](#), and [~Simulation\(\)](#).

5.16.4.4 latticePatch

`LatticePatch` `Simulation::latticePatch` [private]

`LatticePatch` object.

Definition at line 44 of file [SimulationClass.h](#).

Referenced by [addInitialConditions\(\)](#), [advanceToTime\(\)](#), [initializeCVODEobject\(\)](#), [initializePatchwork\(\)](#), [outAllFieldData\(\)](#), and [setInitialConditions\(\)](#).

5.16.4.5 outputManager

`OutputManager` `Simulation::outputManager`

Output Manager object.

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

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

5.16.4.6 statusFlags

```
unsigned char Simulation::statusFlags [private]
```

char for checking simulation flags

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

Referenced by [checkFlag\(\)](#), [checkNoFlag\(\)](#), [initializeCVODEobject\(\)](#), [initializePatchwork\(\)](#), [setDiscreteDimensionsOfLattice\(\)](#), [setPhysicalDimensionsOfLattice\(\)](#), [Simulation\(\)](#), [start\(\)](#), and [~Simulation\(\)](#).

5.16.4.7 t

```
sunrealtype Simulation::t [private]
```

current time of the simulation

Definition at line 46 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 17 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)

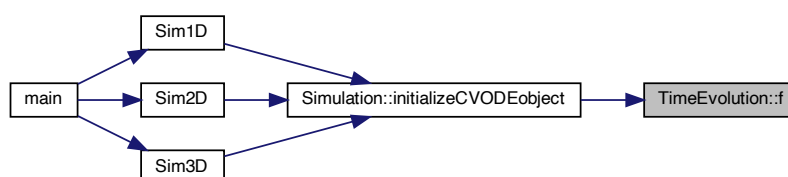
Definition at line 13 of file [TimeEvolutionFunctions.cpp](#).

```
00013 {
00014     // Set recover pointer to provided lattice patch where the data resides
00015     LatticePatch *data = nullptr;
00016     data = static_cast<LatticePatch *>(data_loc);
00017
00018     // pointers for update circle
00019     sunrealtype *udata = nullptr, *dudata = nullptr;
00020     sunrealtype *originaluData = nullptr, *originalduData = nullptr;
00021
00022     // Access NVECTOR_PARALLEL argument data with pointers
00023     udata = NV_DATA_P(u);
00024     dudata = NV_DATA_P(udot);
00025
00026     // Store original data location of the patch
00027     originaluData = data->uData;
00028     originalduData = data->duData;
00029     // Point patch data to arguments of f
00030     data->uData = udata;
00031     data->duData = dudata;
00032
00033     // Time-evolve these arguments (the field data) with specific propagator below
00034     TimeEvolver(data, u, udot, c);
00035
00036     // Refer patch data back to original location
00037     data->uData = originaluData;
00038     data->duData = originalduData;
00039
00040     return (0);
00041 }
```

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 20 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 23 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

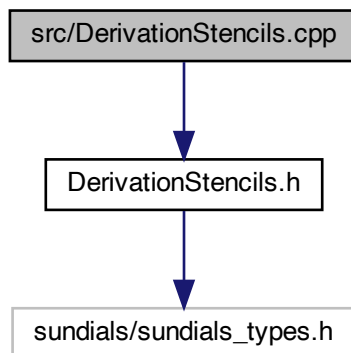
6.1 README.md File Reference

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.](#)

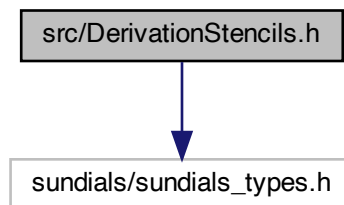
```
00001 ///////////////////////////////////////////////////////////////////
00002 /// @file DerivationStencils.cpp
00003 /// @brief Empty. All definitions in the header.
00004 ///////////////////////////////////////////////////////////////////
00005 #include "DerivationStencils.h"
```

6.4 src/DerivationStencils.h File Reference

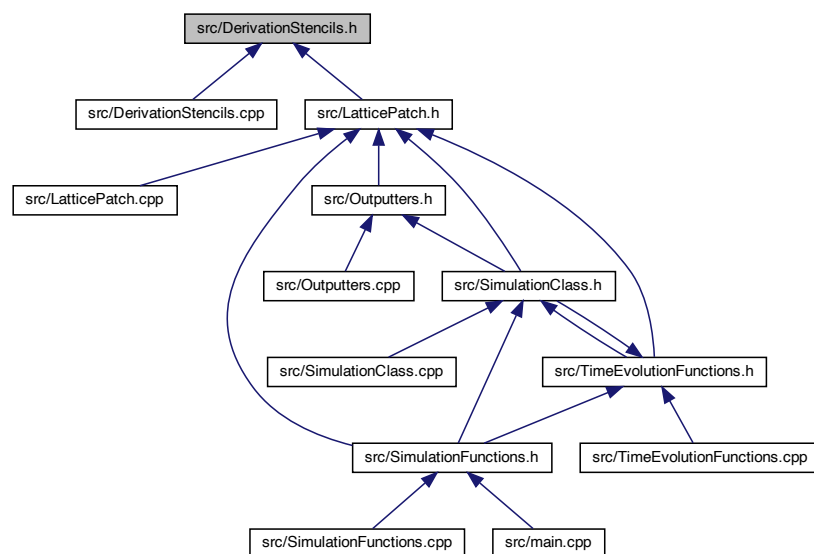
Definition of derivation stencils from order 1 to 13.

```
#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 *udata, int nx)
- sunrealtype [s1b](#) (sunrealtype *udata, int nx)
- sunrealtype [s2f](#) (const sunrealtype *udata, int nx)
- sunrealtype [s2c](#) (const sunrealtype *udata, int nx)
- sunrealtype [s2b](#) (const sunrealtype *udata, int nx)
- sunrealtype [s3f](#) (const sunrealtype *udata, int nx)
- sunrealtype [s3b](#) (sunrealtype *udata, int nx)
- sunrealtype [s4f](#) (const sunrealtype *udata, int nx)
- sunrealtype [s4c](#) (const sunrealtype *udata, int nx)
- sunrealtype [s4b](#) (const sunrealtype *udata, int nx)
- sunrealtype [s5f](#) (const sunrealtype *udata, int nx)
- sunrealtype [s5b](#) (sunrealtype *udata, int nx)
- sunrealtype [s6f](#) (const sunrealtype *udata, int nx)
- sunrealtype [s6c](#) (const sunrealtype *udata, int nx)
- sunrealtype [s6b](#) (const sunrealtype *udata, int nx)
- sunrealtype [s7f](#) (const sunrealtype *udata, int nx)
- sunrealtype [s7b](#) (sunrealtype *udata, int nx)
- sunrealtype [s8f](#) (const sunrealtype *udata, int nx)
- sunrealtype [s8c](#) (const sunrealtype *udata, int nx)
- sunrealtype [s8b](#) (const sunrealtype *udata, int nx)
- sunrealtype [s9f](#) (const sunrealtype *udata, int nx)
- sunrealtype [s9b](#) (sunrealtype *udata, int nx)
- sunrealtype [s10f](#) (const sunrealtype *udata, int nx)
- sunrealtype [s10c](#) (const sunrealtype *udata, int nx)
- sunrealtype [s10b](#) (const sunrealtype *udata, int nx)
- sunrealtype [s11f](#) (const sunrealtype *udata, int nx)
- sunrealtype [s11b](#) (sunrealtype *udata, int nx)
- sunrealtype [s12f](#) (const sunrealtype *udata, int nx)
- sunrealtype [s12c](#) (const sunrealtype *udata, int nx)
- sunrealtype [s12b](#) (const sunrealtype *udata, int nx)
- sunrealtype [s13f](#) (const sunrealtype *udata, int nx)
- sunrealtype [s13b](#) (sunrealtype *udata, int nx)
- sunrealtype [s1f](#) (sunrealtype *udata)
- sunrealtype [s1b](#) (sunrealtype *udata)
- sunrealtype [s2f](#) (sunrealtype *udata)
- sunrealtype [s2c](#) (sunrealtype *udata)
- sunrealtype [s2b](#) (sunrealtype *udata)
- sunrealtype [s3f](#) (sunrealtype *udata)
- sunrealtype [s3b](#) (sunrealtype *udata)
- sunrealtype [s4f](#) (sunrealtype *udata)
- sunrealtype [s4c](#) (sunrealtype *udata)
- sunrealtype [s4b](#) (sunrealtype *udata)
- sunrealtype [s5f](#) (sunrealtype *udata)
- sunrealtype [s5b](#) (sunrealtype *udata)
- sunrealtype [s6f](#) (sunrealtype *udata)
- sunrealtype [s6c](#) (sunrealtype *udata)
- sunrealtype [s6b](#) (sunrealtype *udata)
- sunrealtype [s7f](#) (sunrealtype *udata)
- sunrealtype [s7b](#) (sunrealtype *udata)
- sunrealtype [s8f](#) (sunrealtype *udata)
- sunrealtype [s8c](#) (sunrealtype *udata)
- sunrealtype [s8b](#) (sunrealtype *udata)
- sunrealtype [s9f](#) (sunrealtype *udata)

- sunrealtype [s9b](#) (sunrealtype *udata)
- sunrealtype [s10f](#) (sunrealtype *udata)
- sunrealtype [s10c](#) (sunrealtype *udata)
- sunrealtype [s10b](#) (sunrealtype *udata)
- sunrealtype [s11f](#) (sunrealtype *udata)
- sunrealtype [s11b](#) (sunrealtype *udata)
- sunrealtype [s12f](#) (sunrealtype *udata)
- sunrealtype [s12c](#) (sunrealtype *udata)
- sunrealtype [s12b](#) (sunrealtype *udata)
- sunrealtype [s13f](#) (sunrealtype *udata)
- sunrealtype [s13b](#) (sunrealtype *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]

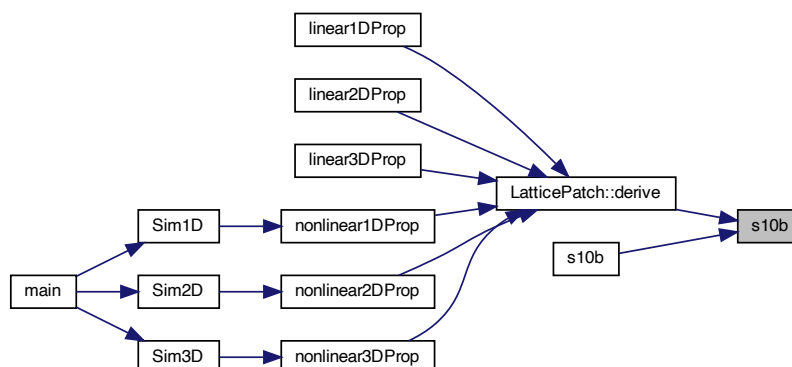
```
sunrealtype s10b (
    const sunrealtype * udata,
    int nx ) [inline]
```

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

```
00146 {
00147     return 1.0 / 840.0 * udata[-4 * nx] - 1.0 / 63.0 * udata[-3 * nx] +
00148           3.0 / 28.0 * udata[-2 * nx] - 4.0 / 7.0 * udata[-1 * nx] -
00149           11.0 / 30.0 * udata[0] + 6.0 / 5.0 * udata[1 * nx] -
00150           1.0 / 2.0 * udata[2 * nx] + 4.0 / 21.0 * udata[3 * nx] -
00151           3.0 / 56.0 * udata[4 * nx] + 1.0 / 105.0 * udata[5 * nx] -
00152           1.0 / 1260.0 * udata[6 * nx];
00153 }
```

Referenced by [LatticePatch::derive\(\)](#), and [s10b\(\)](#).

Here is the caller graph for this function:



6.4.2.2 s10b() [2/2]

```
sunrealtype s10b (
    sunrealtype * udata ) [inline]
```

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

```
00243 { return s10b(udata, 6); }
```

References [s10b\(\)](#).

Here is the call graph for this function:



6.4.2.3 s10c() [1/2]

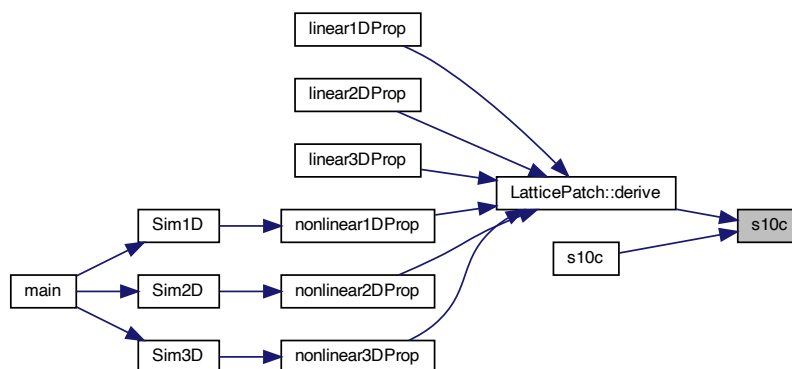
```
sunrealtype s10c (
    const sunrealtype * udata,
    int nx ) [inline]
```

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

```
00139 {
00140     return -1.0 / 1260.0 * udata[-5 * nx] + 5.0 / 504.0 * udata[-4 * nx] -
00141            5.0 / 84.0 * udata[-3 * nx] + 5.0 / 21.0 * udata[-2 * nx] -
00142            5.0 / 6.0 * udata[-1 * nx] + 0 + 5.0 / 6.0 * udata[1 * nx] -
00143            5.0 / 21.0 * udata[2 * nx] + 5.0 / 84.0 * udata[3 * nx] -
00144            5.0 / 504.0 * udata[4 * nx] + 1.0 / 1260.0 * udata[5 * nx];
00145 }
```

Referenced by [LatticePatch::derive\(\)](#), and [s10c\(\)](#).

Here is the caller graph for this function:



6.4.2.4 s10c() [2/2]

```
sunrealtype s10c (
    sunrealtype * udata ) [inline]
```

Definition at line 242 of file [DerivationStencils.h](#).

```
00242 { return s10c(udata, 6); }
```

References [s10c\(\)](#).

Here is the call graph for this function:



6.4.2.5 s10f() [1/2]

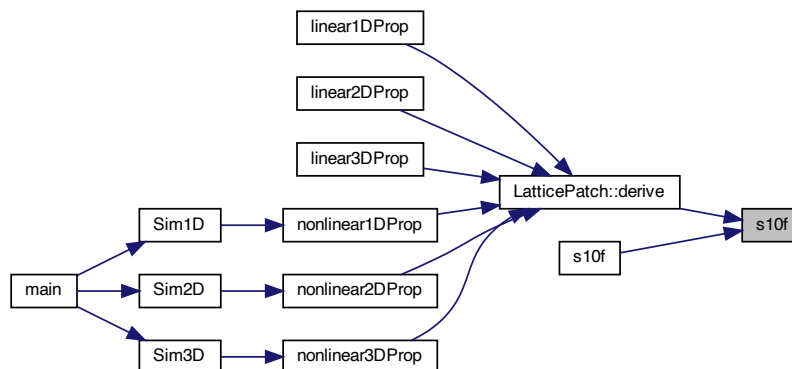
```
sunrealtype s10f (
    const sunrealtype * udata,
    int nx ) [inline]
```

Definition at line 131 of file [DerivationStencils.h](#).

```
00131 {
00132     return 1.0 / 1260.0 * udata[-6 * nx] - 1.0 / 105.0 * udata[-5 * nx] +
00133           3.0 / 56.0 * udata[-4 * nx] - 4.0 / 21.0 * udata[-3 * nx] +
00134           1.0 / 2.0 * udata[-2 * nx] - 6.0 / 5.0 * udata[-1 * nx] +
00135           11.0 / 30.0 * udata[0] + 4.0 / 7.0 * udata[1 * nx] -
00136           3.0 / 28.0 * udata[2 * nx] + 1.0 / 63.0 * udata[3 * nx] -
00137           1.0 / 840.0 * udata[4 * nx];
00138 }
```

Referenced by [LatticePatch::derive\(\)](#), and [s10f\(\)](#).

Here is the caller graph for this function:



6.4.2.6 s10f() [2/2]

```
sunrealtype s10f (  
    sunrealtype * udata ) [inline]
```

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

```
00241 { return s10f(udata, 6); }
```

References [s10f\(\)](#).

Here is the call graph for this function:



6.4.2.7 s11b() [1/2]

```
sunrealtype s11b (  
    sunrealtype * udata ) [inline]
```

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

```
00245 { return s11b(udata, 6); }
```

References [s11b\(\)](#).

Here is the call graph for this function:



6.4.2.8 s11b() [2/2]

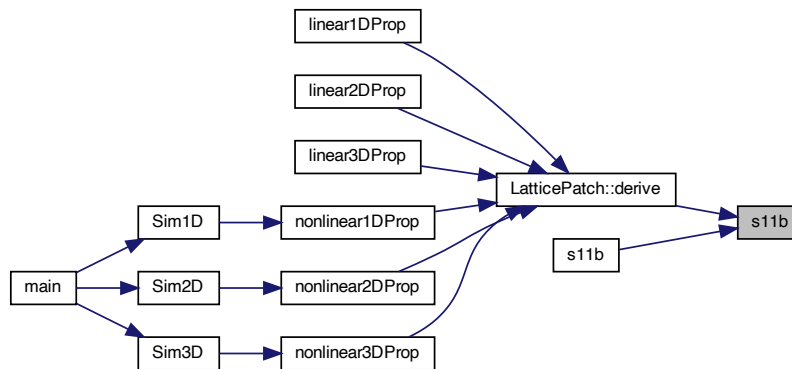
```
sunrealtype s11b (
    sunrealtype * udata,
    int nx ) [inline]
```

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

```
00162     {
00163     return -1.0 / 2310.0 * udata[-5 * nx] + 1.0 / 168.0 * udata[-4 * nx] -
00164           5.0 / 126.0 * udata[-3 * nx] + 5.0 / 28.0 * udata[-2 * nx] -
00165           5.0 / 7.0 * udata[-1 * nx] - 1.0 / 6.0 * udata[0] + udata[1 * nx] -
00166           5.0 / 14.0 * udata[2 * nx] + 5.0 / 42.0 * udata[3 * nx] -
00167           5.0 / 168.0 * udata[4 * nx] + 1.0 / 210.0 * udata[5 * nx] -
00168           1.0 / 2772.0 * udata[6 * nx];
00169 }
```

Referenced by [LatticePatch::derive\(\)](#), and [s11b\(\)](#).

Here is the caller graph for this function:



6.4.2.9 s11f() [1/2]

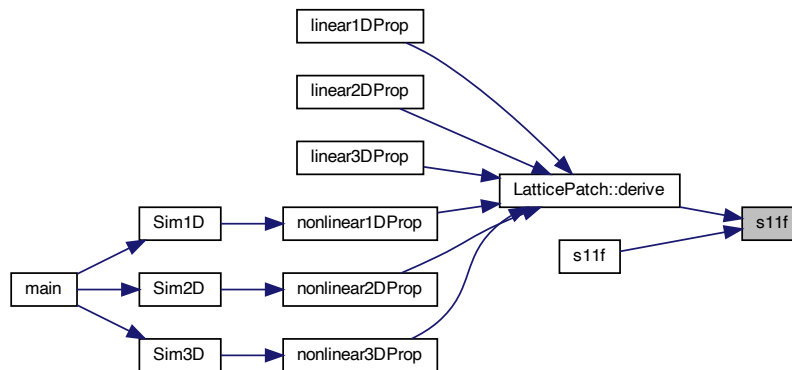
```
sunrealtype s11f (
    const sunrealtype * udata,
    int nx ) [inline]
```

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

```
00154     {
00155     return 1.0 / 2772.0 * udata[-6 * nx] - 1.0 / 210.0 * udata[-5 * nx] +
00156           5.0 / 168.0 * udata[-4 * nx] - 5.0 / 42.0 * udata[-3 * nx] +
00157           5.0 / 14.0 * udata[-2 * nx] - 1.0 / 1.0 * udata[-1 * nx] +
00158           1.0 / 6.0 * udata[0] + 5.0 / 7.0 * udata[1 * nx] -
00159           5.0 / 28.0 * udata[2 * nx] + 5.0 / 126.0 * udata[3 * nx] -
00160           1.0 / 168.0 * udata[4 * nx] + 1.0 / 2310.0 * udata[5 * nx];
00161 }
```

Referenced by [LatticePatch::derive\(\)](#), and [s11f\(\)](#).

Here is the caller graph for this function:



6.4.2.10 s11f() [2/2]

```

sunrealtype s11f (
    sunrealtype * udata ) [inline]

```

Definition at line 244 of file [DerivationStencils.h](#).

```

00244 { return s11f(udata, 6); }

```

References [s11f\(\)](#).

Here is the call graph for this function:



6.4.2.11 s12b() [1/2]

```

sunrealtype s12b (
    const sunrealtype * udata,
    int nx ) [inline]

```

Definition at line 187 of file [DerivationStencils.h](#).

```

00187

```

```

{

```

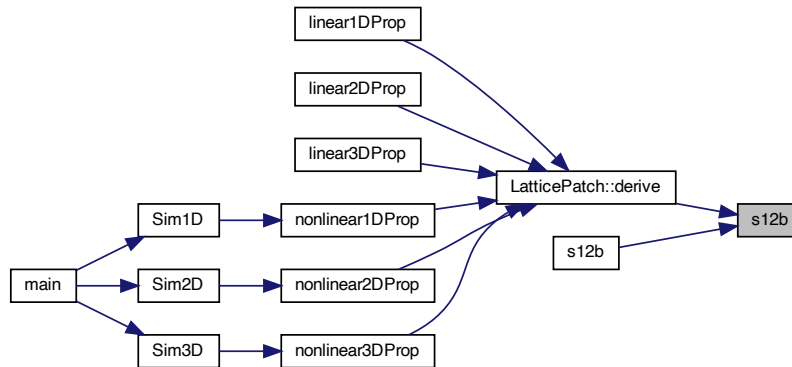
```

00188     return -1.0 / 3960.0 * udata[-5 * nx] + 1.0 / 264.0 * udata[-4 * nx] -
00189           1.0 / 36.0 * udata[-3 * nx] + 5.0 / 36.0 * udata[-2 * nx] -
00190           5.0 / 8.0 * udata[-1 * nx] - 13.0 / 42.0 * udata[0] +
00191           7.0 / 6.0 * udata[1 * nx] - 1.0 / 2.0 * udata[2 * nx] +
00192           5.0 / 24.0 * udata[3 * nx] - 5.0 / 72.0 * udata[4 * nx] +
00193           1.0 / 60.0 * udata[5 * nx] - 1.0 / 396.0 * udata[6 * nx] +
00194           1.0 / 5544.0 * udata[7 * nx];
00195 }

```

Referenced by [LatticePatch::derive\(\)](#), and [s12b\(\)](#).

Here is the caller graph for this function:



6.4.2.12 `s12b()` [2/2]

```

sunrealtype s12b (
    sunrealtype * udata ) [inline]

```

Definition at line 248 of file [DerivationStencils.h](#).

```

00248 { return s12b(udata, 6); }

```

References [s12b\(\)](#).

Here is the call graph for this function:



6.4.2.13 s12c() [1/2]

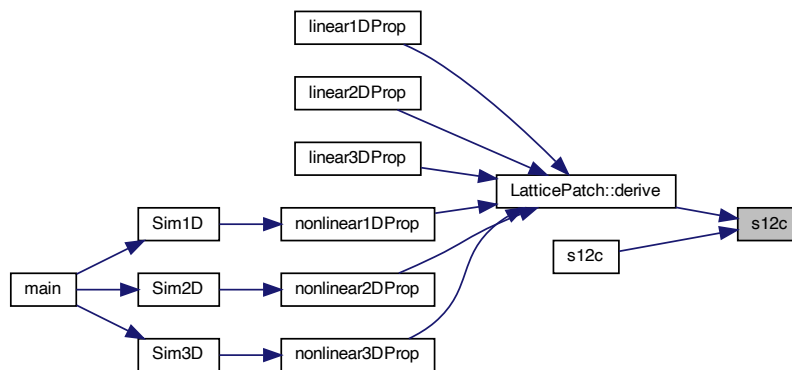
```
sunrealtype s12c (
    const sunrealtype * udata,
    int nx ) [inline]
```

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

```
00179 {
00180     return 1.0 / 5544.0 * udata[-6 * nx] - 1.0 / 385.0 * udata[-5 * nx] +
00181           1.0 / 56.0 * udata[-4 * nx] - 5.0 / 63.0 * udata[-3 * nx] +
00182           15.0 / 56.0 * udata[-2 * nx] - 6.0 / 7.0 * udata[-1 * nx] + 0 +
00183           6.0 / 7.0 * udata[1 * nx] - 15.0 / 56.0 * udata[2 * nx] +
00184           5.0 / 63.0 * udata[3 * nx] - 1.0 / 56.0 * udata[4 * nx] +
00185           1.0 / 385.0 * udata[5 * nx] - 1.0 / 5544.0 * udata[6 * nx];
00186 }
```

Referenced by [LatticePatch::derive\(\)](#), and [s12c\(\)](#).

Here is the caller graph for this function:

**6.4.2.14 s12c()** [2/2]

```
sunrealtype s12c (
    sunrealtype * udata ) [inline]
```

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

```
00247 { return s12c(udata, 6); }
```

References [s12c\(\)](#).

Here is the call graph for this function:



6.4.2.15 s12f() [1/2]

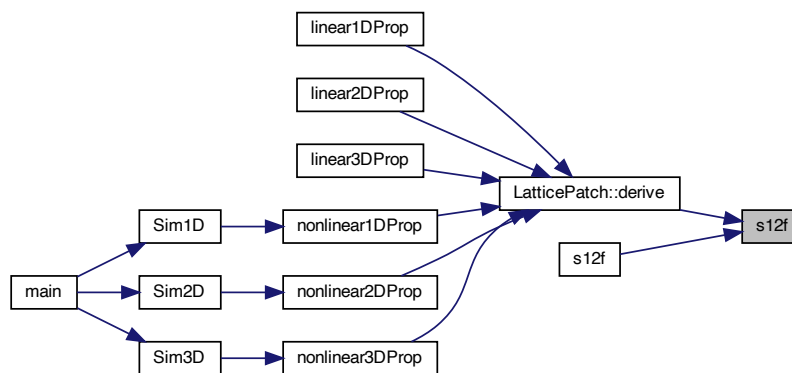
```
sunrealtype s12f (
    const sunrealtype * udata,
    int nx ) [inline]
```

Definition at line 170 of file [DerivationStencils.h](#).

```
00170 {
00171     return -1.0 / 5544.0 * udata[-7 * nx] + 1.0 / 396.0 * udata[-6 * nx] -
00172           1.0 / 60.0 * udata[-5 * nx] + 5.0 / 72.0 * udata[-4 * nx] -
00173           5.0 / 24.0 * udata[-3 * nx] + 1.0 / 2.0 * udata[-2 * nx] -
00174           7.0 / 6.0 * udata[-1 * nx] + 13.0 / 42.0 * udata[0] +
00175           5.0 / 8.0 * udata[1 * nx] - 5.0 / 36.0 * udata[2 * nx] +
00176           1.0 / 36.0 * udata[3 * nx] - 1.0 / 264.0 * udata[4 * nx] +
00177           1.0 / 3960.0 * udata[5 * nx];
00178 }
```

Referenced by [LatticePatch::derive\(\)](#), and [s12f\(\)](#).

Here is the caller graph for this function:



6.4.2.16 s12f() [2/2]

```
sunrealtype s12f (
    sunrealtype * udata ) [inline]
```

Definition at line 246 of file [DerivationStencils.h](#).

```
00246 { return s12f(udata, 6); }
```

References [s12f\(\)](#).

Here is the call graph for this function:



6.4.2.17 s13b() [1/2]

```
sunrealtype s13b (
    sunrealtype * udata ) [inline]
```

Definition at line 250 of file [DerivationStencils.h](#).

```
00250 { return s13b(udata, 6); }
```

References [s13b\(\)](#).

Here is the call graph for this function:

**6.4.2.18 s13b()** [2/2]

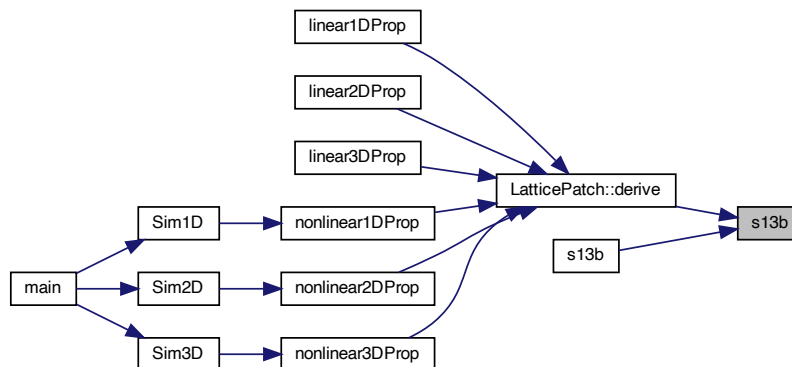
```
sunrealtype s13b (
    sunrealtype * udata,
    int nx ) [inline]
```

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

```
00205 {
00206     return 1.0 / 10296.0 * udata[-6 * nx] - 1.0 / 660.0 * udata[-5 * nx] +
00207            1.0 / 88.0 * udata[-4 * nx] - 1.0 / 18.0 * udata[-3 * nx] +
00208            5.0 / 24.0 * udata[-2 * nx] - 3.0 / 4.0 * udata[-1 * nx] -
00209            1.0 / 7.0 * udata[0] + udata[1 * nx] - 3.0 / 8.0 * udata[2 * nx] +
00210            5.0 / 36.0 * udata[3 * nx] - 1.0 / 24.0 * udata[4 * nx] +
00211            1.0 / 110.0 * udata[5 * nx] - 1.0 / 792.0 * udata[6 * nx] +
00212            1.0 / 12012.0 * udata[7 * nx];
00213 }
```

Referenced by [LatticePatch::derive\(\)](#), and [s13b\(\)](#).

Here is the caller graph for this function:



6.4.2.19 s13f() [1/2]

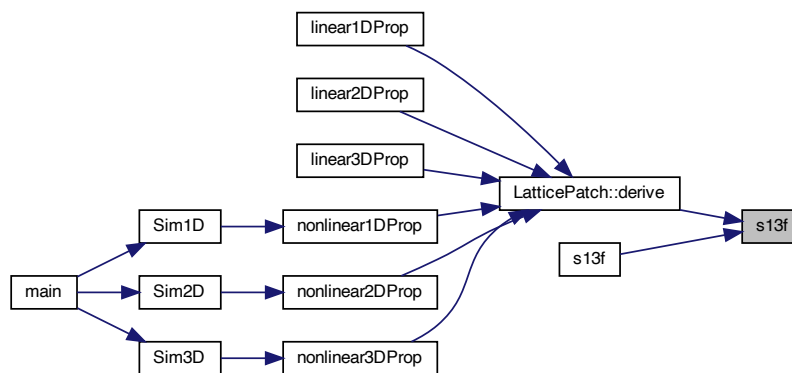
```
sunrealtype s13f (
    const sunrealtype * udata,
    int nx ) [inline]
```

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

```
00196 {
00197     return -1.0 / 12012.0 * udata[-7 * nx] + 1.0 / 792.0 * udata[-6 * nx] -
00198            1.0 / 110.0 * udata[-5 * nx] + 1.0 / 24.0 * udata[-4 * nx] -
00199            5.0 / 36.0 * udata[-3 * nx] + 3.0 / 8.0 * udata[-2 * nx] -
00200            1.0 / 1.0 * udata[-1 * nx] + 1.0 / 7.0 * udata[0] +
00201            3.0 / 4.0 * udata[1 * nx] - 5.0 / 24.0 * udata[2 * nx] +
00202            1.0 / 18.0 * udata[3 * nx] - 1.0 / 88.0 * udata[4 * nx] +
00203            1.0 / 660.0 * udata[5 * nx] - 1.0 / 10296.0 * udata[6 * nx];
00204 }
```

Referenced by [LatticePatch::derive\(\)](#), and [s13f\(\)](#).

Here is the caller graph for this function:



6.4.2.20 s13f() [2/2]

```
sunrealtype s13f (
    sunrealtype * udata ) [inline]
```

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

```
00249 { return s13f(udata, 6); }
```

References [s13f\(\)](#).

Here is the call graph for this function:



6.4.2.21 s1b() [1/2]

```
sunrealtype s1b (
    sunrealtype * udata ) [inline]
```

Definition at line 220 of file [DerivationStencils.h](#).

```
00220 { return s1b(udata, 6); }
```

References [s1b\(\)](#).

Here is the call graph for this function:

**6.4.2.22 s1b()** [2/2]

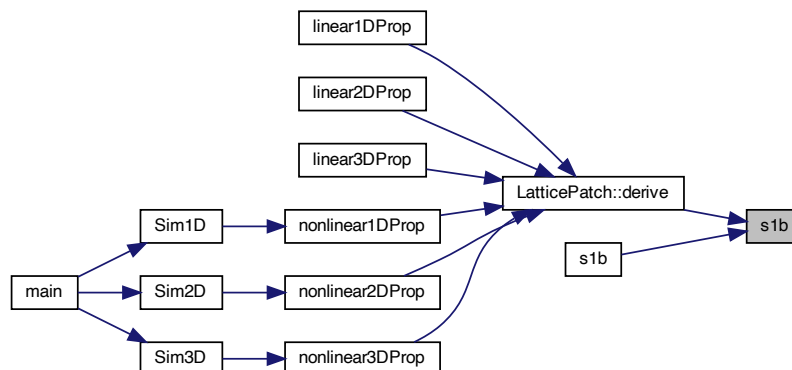
```
sunrealtype s1b (
    sunrealtype * udata,
    int nx ) [inline]
```

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

```
00021 {
00022     return -1.0 / 1.0 * udata[0] + udata[1 * nx];
00023 }
```

Referenced by [LatticePatch::derive\(\)](#), and [s1b\(\)](#).

Here is the caller graph for this function:



6.4.2.23 s1f() [1/2]

```
sunrealtype s1f (
    sunrealtype * udata ) [inline]
```

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

```
00219 { return s1f(udata, 6); }
```

References [s1f\(\)](#).

Here is the call graph for this function:

**6.4.2.24 s1f()** [2/2]

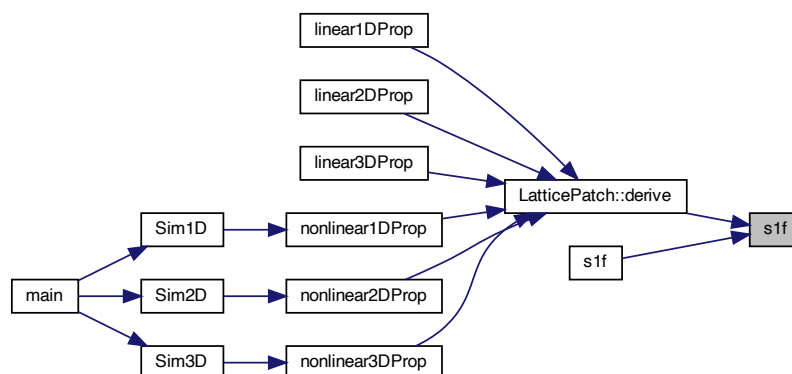
```
sunrealtype s1f (
    sunrealtype * udata,
    int nx ) [inline]
```

Definition at line 17 of file [DerivationStencils.h](#).

```
00017 {
00018     return -1.0 / 1.0 * udata[-1 * nx] + udata[0];
00019 }
```

Referenced by [LatticePatch::derive\(\)](#), and [s1f\(\)](#).

Here is the caller graph for this function:



6.4.2.25 s2b() [1/2]

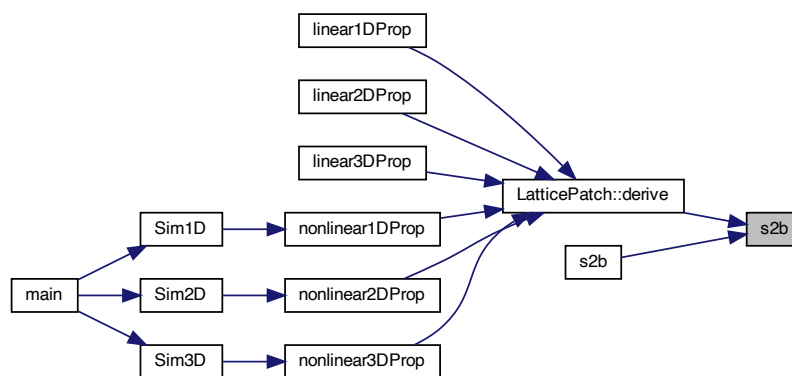
```
sunrealtype s2b (
    const sunrealtype * udata,
    int nx ) [inline]
```

Definition at line 32 of file [DerivationStencils.h](#).

```
00032 {
00033     return -3.0 / 2.0 * udata[0] + 2.0 / 1.0 * udata[1 * nx] -
00034           1.0 / 2.0 * udata[2 * nx];
00035 }
```

Referenced by [LatticePatch::derive\(\)](#), and [s2b\(\)](#).

Here is the caller graph for this function:



6.4.2.26 s2b() [2/2]

```
sunrealtype s2b (
    sunrealtype * udata ) [inline]
```

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

```
00223 { return s2b(udata, 6); }
```

References [s2b\(\)](#).

Here is the call graph for this function:



6.4.2.27 s2c() [1/2]

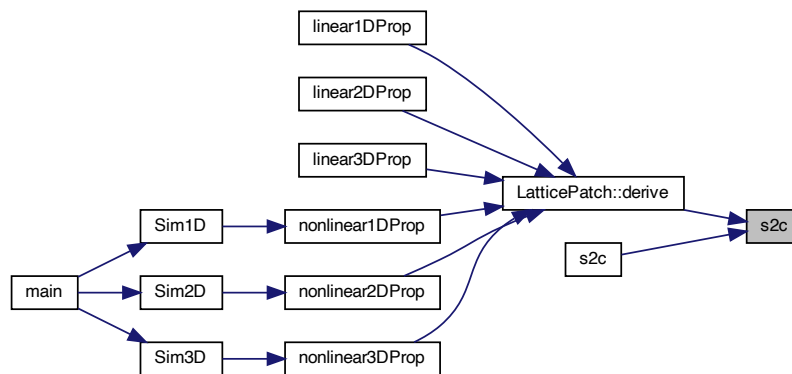
```
sunrealtype s2c (
    const sunrealtype * udata,
    int nx ) [inline]
```

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

```
00029 {
00030     return -1.0 / 2.0 * udata[-1 * nx] + 0 + 1.0 / 2.0 * udata[1 * nx];
00031 }
```

Referenced by [LatticePatch::derive\(\)](#), and [s2c\(\)](#).

Here is the caller graph for this function:



6.4.2.28 s2c() [2/2]

```
sunrealtype s2c (
    sunrealtype * udata ) [inline]
```

Definition at line 222 of file [DerivationStencils.h](#).

```
00222 { return s2c(udata, 6); }
```

References [s2c\(\)](#).

Here is the call graph for this function:



6.4.2.29 s2f() [1/2]

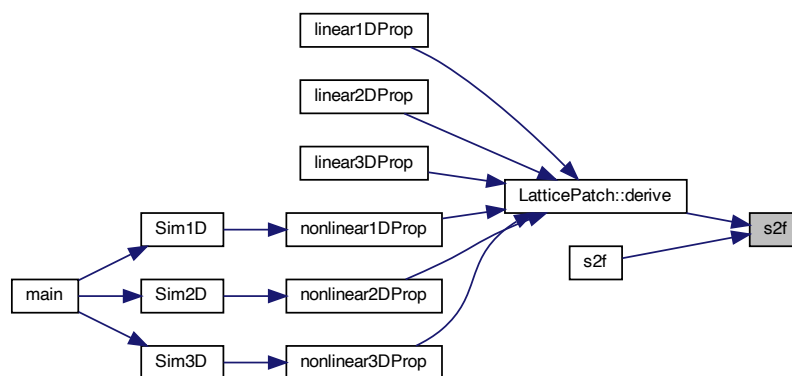
```
sunrealtype s2f (
    const sunrealtype * udata,
    int nx ) [inline]
```

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

```
00025 {
00026     return 1.0 / 2.0 * udata[-2 * nx] - 2.0 / 1.0 * udata[-1 * nx] +
00027           3.0 / 2.0 * udata[0];
00028 }
```

Referenced by [LatticePatch::derive\(\)](#), and [s2f\(\)](#).

Here is the caller graph for this function:

**6.4.2.30 s2f()** [2/2]

```
sunrealtype s2f (
    sunrealtype * udata ) [inline]
```

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

```
00221 { return s2f(udata, 6); }
```

References [s2f\(\)](#).

Here is the call graph for this function:



6.4.2.31 s3b() [1/2]

```
sunrealtype s3b (
    sunrealtype * udata ) [inline]
```

Definition at line 225 of file [DerivationStencils.h](#).

```
00225 { return s3b(udata, 6); }
```

References [s3b\(\)](#).

Here is the call graph for this function:



6.4.2.32 s3b() [2/2]

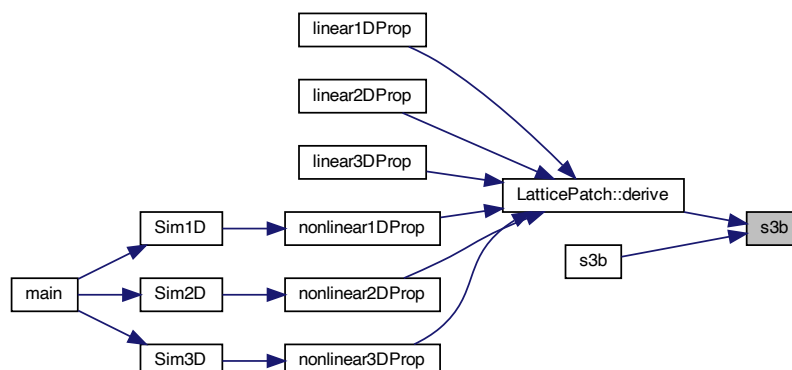
```
sunrealtype s3b (
    sunrealtype * udata,
    int nx ) [inline]
```

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

```
00040 {
00041     return -1.0 / 3.0 * udata[-1 * nx] - 1.0 / 2.0 * udata[0] + udata[1 * nx] -
00042            1.0 / 6.0 * udata[2 * nx];
00043 }
```

Referenced by [LatticePatch::derive\(\)](#), and [s3b\(\)](#).

Here is the caller graph for this function:



6.4.2.33 s3f() [1/2]

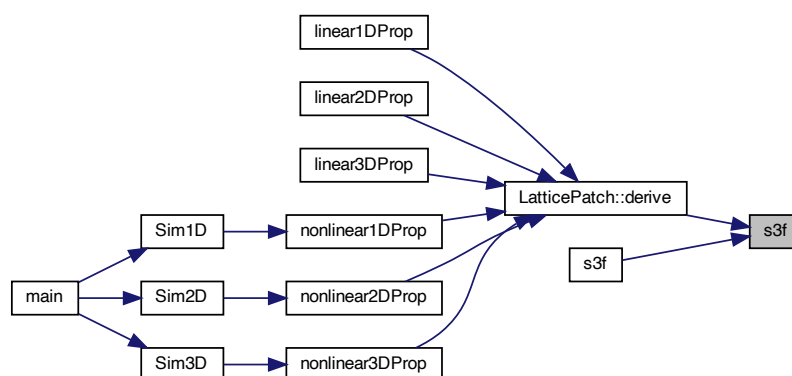
```
sunrealtype s3f (
    const sunrealtype * udata,
    int nx ) [inline]
```

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

```
00036 {
00037     return 1.0 / 6.0 * udata[-2 * nx] - 1.0 / 1.0 * udata[-1 * nx] +
00038           1.0 / 2.0 * udata[0] + 1.0 / 3.0 * udata[1 * nx];
00039 }
```

Referenced by [LatticePatch::derive\(\)](#), and [s3f\(\)](#).

Here is the caller graph for this function:

**6.4.2.34 s3f()** [2/2]

```
sunrealtype s3f (
    sunrealtype * udata ) [inline]
```

Definition at line 224 of file [DerivationStencils.h](#).

```
00224 { return s3f(udata, 6); }
```

References [s3f\(\)](#).

Here is the call graph for this function:



6.4.2.35 `s4b()` [1/2]

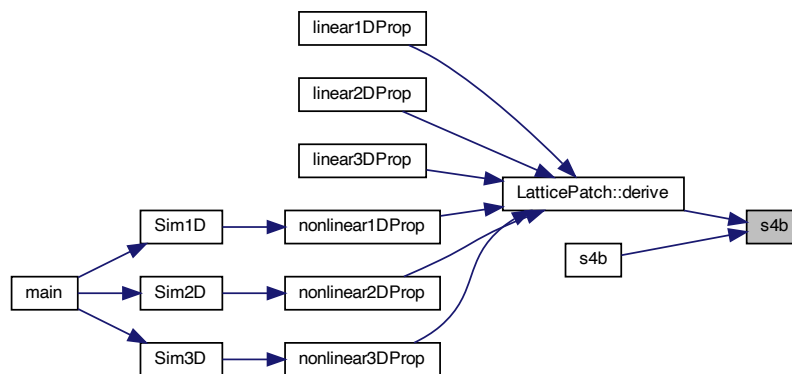
```
sunrealtype s4b (
    const sunrealtype * udata,
    int nx ) [inline]
```

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

```
00053 {
00054     return -1.0 / 4.0 * udata[-1 * nx] - 5.0 / 6.0 * udata[0] +
00055           3.0 / 2.0 * udata[1 * nx] - 1.0 / 2.0 * udata[2 * nx] +
00056           1.0 / 12.0 * udata[3 * nx];
00057 }
```

Referenced by [LatticePatch::derive\(\)](#), and [s4b\(\)](#).

Here is the caller graph for this function:



6.4.2.36 `s4b()` [2/2]

```
sunrealtype s4b (
    sunrealtype * udata ) [inline]
```

Definition at line 228 of file [DerivationStencils.h](#).

```
00228 { return s4b(udata, 6); }
```

References [s4b\(\)](#).

Here is the call graph for this function:



6.4.2.37 s4c() [1/2]

```

sunrealtype s4c (
    const sunrealtype * udata,
    int nx ) [inline]

```

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

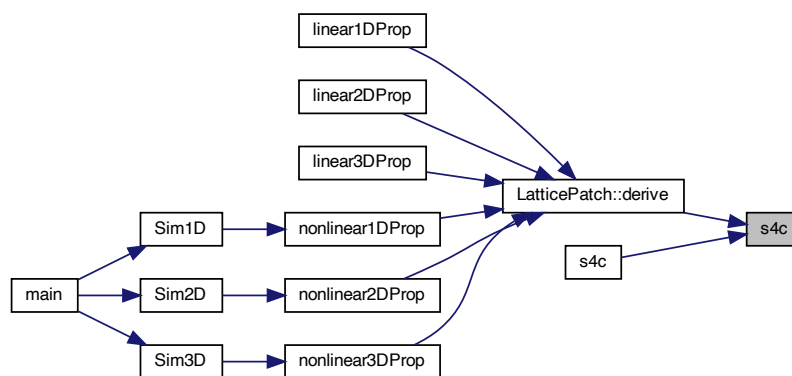
```

00049 {
00050     return 1.0 / 12.0 * udata[-2 * nx] - 2.0 / 3.0 * udata[-1 * nx] + 0 +
00051           2.0 / 3.0 * udata[1 * nx] - 1.0 / 12.0 * udata[2 * nx];
00052 }

```

Referenced by [LatticePatch::derive\(\)](#), and [s4c\(\)](#).

Here is the caller graph for this function:

**6.4.2.38 s4c()** [2/2]

```

sunrealtype s4c (
    sunrealtype * udata ) [inline]

```

Definition at line 227 of file [DerivationStencils.h](#).

```

00227 { return s4c(udata, 6); }

```

References [s4c\(\)](#).

Here is the call graph for this function:



6.4.2.39 s4f() [1/2]

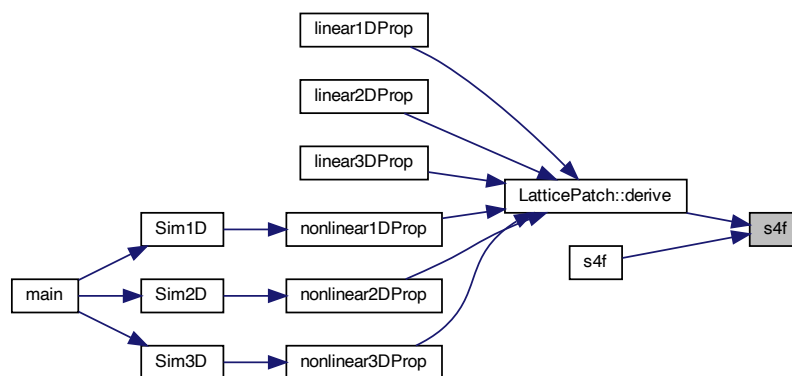
```
sunrealtype s4f (
    const sunrealtype * udata,
    int nx ) [inline]
```

Definition at line 44 of file [DerivationStencils.h](#).

```
00044 {
00045     return -1.0 / 12.0 * udata[-3 * nx] + 1.0 / 2.0 * udata[-2 * nx] -
00046           3.0 / 2.0 * udata[-1 * nx] + 5.0 / 6.0 * udata[0] +
00047           1.0 / 4.0 * udata[1 * nx];
00048 }
```

Referenced by [LatticePatch::derive\(\)](#), and [s4f\(\)](#).

Here is the caller graph for this function:



6.4.2.40 s4f() [2/2]

```
sunrealtype s4f (
    sunrealtype * udata ) [inline]
```

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

```
00226 { return s4f(udata, 6); }
```

References [s4f\(\)](#).

Here is the call graph for this function:



6.4.2.41 s5b() [1/2]

```
sunrealtype s5b (
    sunrealtype * udata ) [inline]
```

Definition at line 230 of file [DerivationStencils.h](#).

```
00230 { return s5b(udata, 6); }
```

References [s5b\(\)](#).

Here is the call graph for this function:

**6.4.2.42 s5b()** [2/2]

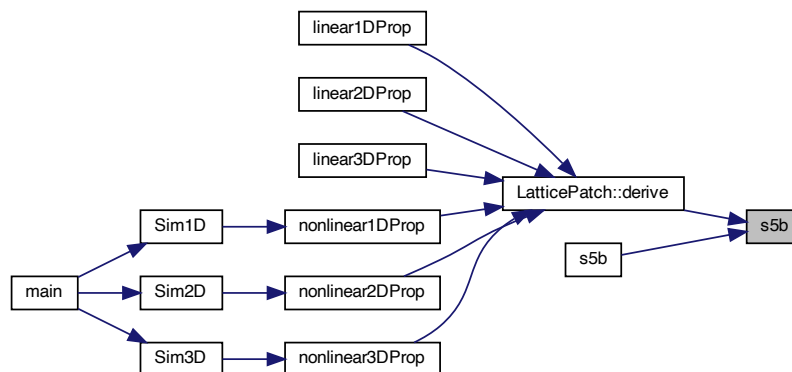
```
sunrealtype s5b (
    sunrealtype * udata,
    int nx ) [inline]
```

Definition at line 63 of file [DerivationStencils.h](#).

```
00063 {
00064     return 1.0 / 20.0 * udata[-2 * nx] - 1.0 / 2.0 * udata[-1 * nx] -
00065           1.0 / 3.0 * udata[0] + udata[1 * nx] - 1.0 / 4.0 * udata[2 * nx] +
00066           1.0 / 30.0 * udata[3 * nx];
00067 }
```

Referenced by [LatticePatch::derive\(\)](#), and [s5b\(\)](#).

Here is the caller graph for this function:



6.4.2.43 s5f() [1/2]

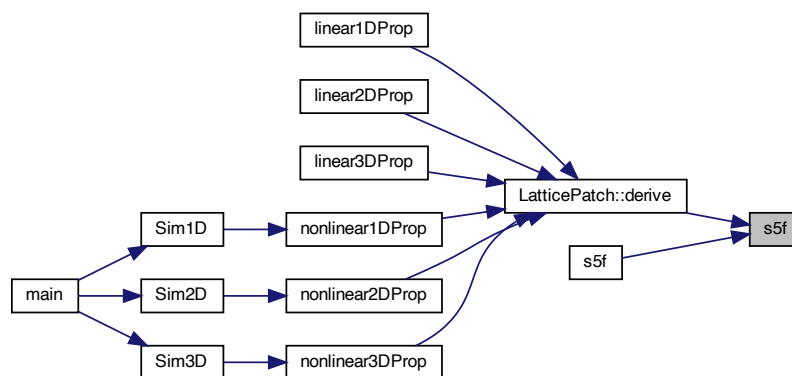
```
sunrealtype s5f (
    const sunrealtype * udata,
    int nx ) [inline]
```

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

```
00058 {
00059     return -1.0 / 30.0 * udata[-3 * nx] + 1.0 / 4.0 * udata[-2 * nx] -
00060           1.0 / 1.0 * udata[-1 * nx] + 1.0 / 3.0 * udata[0] +
00061           1.0 / 2.0 * udata[1 * nx] - 1.0 / 20.0 * udata[2 * nx];
00062 }
```

Referenced by [LatticePatch::derive\(\)](#), and [s5f\(\)](#).

Here is the caller graph for this function:



6.4.2.44 s5f() [2/2]

```
sunrealtype s5f (
    sunrealtype * udata ) [inline]
```

Definition at line 229 of file [DerivationStencils.h](#).

```
00229 { return s5f(udata, 6); }
```

References [s5f\(\)](#).

Here is the call graph for this function:



6.4.2.45 s6b() [1/2]

```

sunrealtype s6b (
    const sunrealtype * udata,
    int nx ) [inline]

```

Definition at line 79 of file [DerivationStencils.h](#).

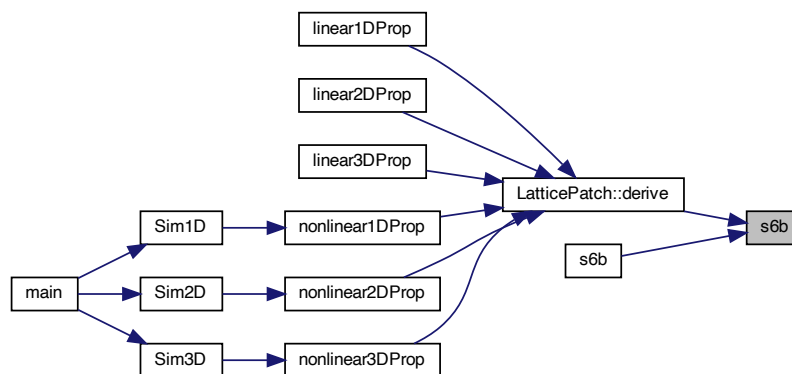
```

00079 {
00080     return 1.0 / 30.0 * udata[-2 * nx] - 2.0 / 5.0 * udata[-1 * nx] -
00081           7.0 / 12.0 * udata[0] + 4.0 / 3.0 * udata[1 * nx] -
00082           1.0 / 2.0 * udata[2 * nx] + 2.0 / 15.0 * udata[3 * nx] -
00083           1.0 / 60.0 * udata[4 * nx];
00084 }

```

Referenced by [LatticePatch::derive\(\)](#), and [s6b\(\)](#).

Here is the caller graph for this function:

**6.4.2.46 s6b()** [2/2]

```

sunrealtype s6b (
    sunrealtype * udata ) [inline]

```

Definition at line 233 of file [DerivationStencils.h](#).

```

00233 { return s6b(udata, 6); }

```

References [s6b\(\)](#).

Here is the call graph for this function:



6.4.2.47 s6c() [1/2]

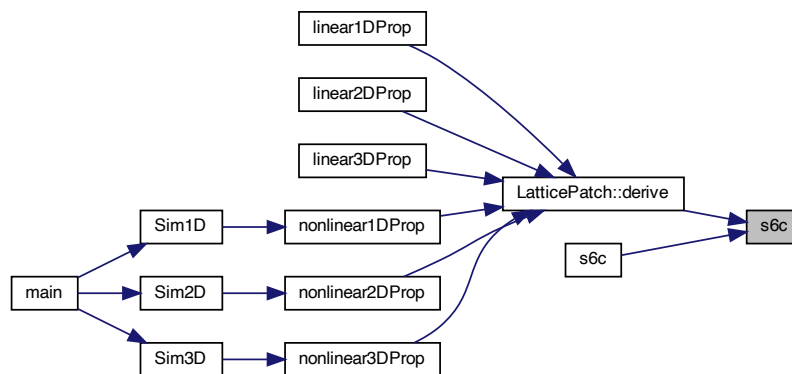
```
sunrealtype s6c (
    const sunrealtype * udata,
    int nx ) [inline]
```

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

```
00074 {
00075     return -1.0 / 60.0 * udata[-3 * nx] + 3.0 / 20.0 * udata[-2 * nx] -
00076           3.0 / 4.0 * udata[-1 * nx] + 0 + 3.0 / 4.0 * udata[1 * nx] -
00077           3.0 / 20.0 * udata[2 * nx] + 1.0 / 60.0 * udata[3 * nx];
00078 }
```

Referenced by [LatticePatch::derive\(\)](#), and [s6c\(\)](#).

Here is the caller graph for this function:



6.4.2.48 s6c() [2/2]

```
sunrealtype s6c (
    sunrealtype * udata ) [inline]
```

Definition at line 232 of file [DerivationStencils.h](#).

```
00232 { return s6c(udata, 6); }
```

References [s6c\(\)](#).

Here is the call graph for this function:



6.4.2.49 s6f() [1/2]

```

sunrealtype s6f (
    const sunrealtype * udata,
    int nx ) [inline]

```

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

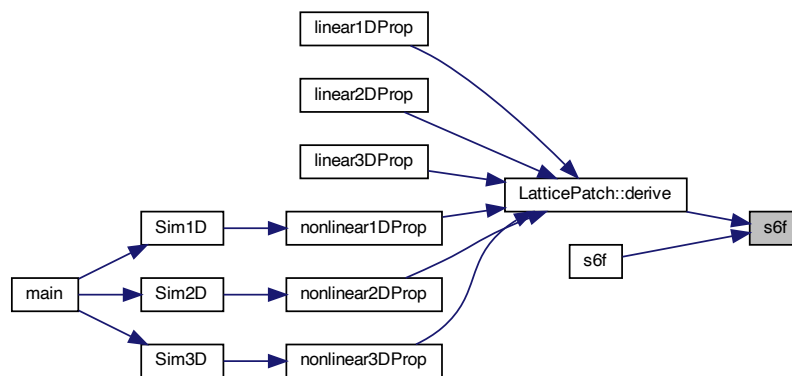
```

00068
00069     return 1.0 / 60.0 * udata[-4 * nx] - 2.0 / 15.0 * udata[-3 * nx] +
00070           1.0 / 2.0 * udata[-2 * nx] - 4.0 / 3.0 * udata[-1 * nx] +
00071           7.0 / 12.0 * udata[0] + 2.0 / 5.0 * udata[1 * nx] -
00072           1.0 / 30.0 * udata[2 * nx];
00073 }

```

Referenced by [LatticePatch::derive\(\)](#), and [s6f\(\)](#).

Here is the caller graph for this function:

**6.4.2.50 s6f()** [2/2]

```

sunrealtype s6f (
    sunrealtype * udata ) [inline]

```

Definition at line 231 of file [DerivationStencils.h](#).

```

00231 { return s6f(udata, 6); }

```

References [s6f\(\)](#).

Here is the call graph for this function:



6.4.2.51 s7b() [1/2]

```
sunrealtype s7b (
    sunrealtype * udata ) [inline]
```

Definition at line 235 of file [DerivationStencils.h](#).

```
00235 { return s7b(udata, 6); }
```

References [s7b\(\)](#).

Here is the call graph for this function:



6.4.2.52 s7b() [2/2]

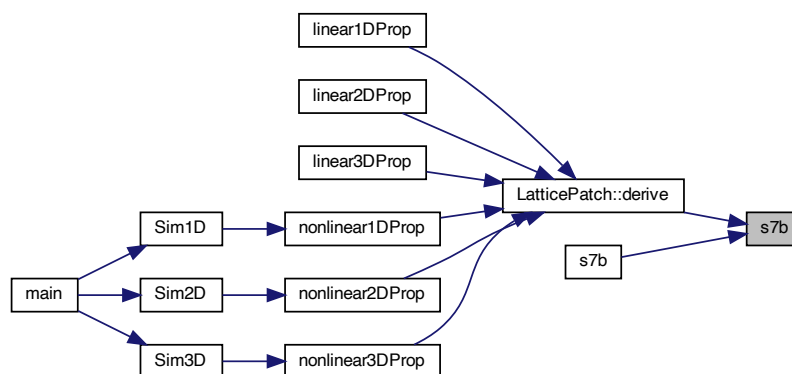
```
sunrealtype s7b (
    sunrealtype * udata,
    int nx ) [inline]
```

Definition at line 91 of file [DerivationStencils.h](#).

```
00091 {
00092     return -1.0 / 105.0 * udata[-3 * nx] + 1.0 / 10.0 * udata[-2 * nx] -
00093            3.0 / 5.0 * udata[-1 * nx] - 1.0 / 4.0 * udata[0] + udata[1 * nx] -
00094            3.0 / 10.0 * udata[2 * nx] + 1.0 / 15.0 * udata[3 * nx] -
00095            1.0 / 140.0 * udata[4 * nx];
00096 }
```

Referenced by [LatticePatch::derive\(\)](#), and [s7b\(\)](#).

Here is the caller graph for this function:



6.4.2.53 s7f() [1/2]

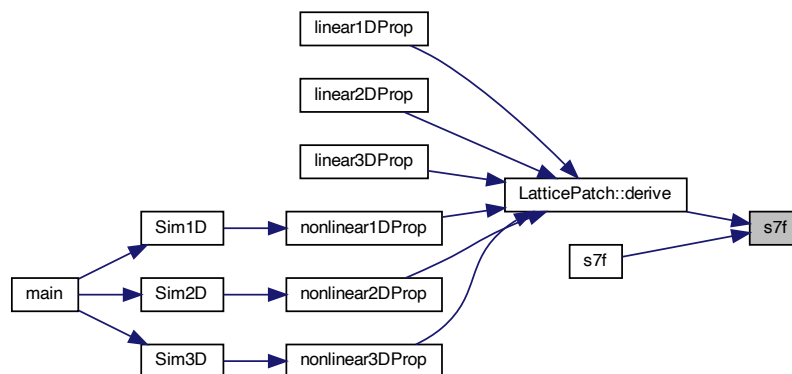
```
sunrealtype s7f (
    const sunrealtype * udata,
    int nx ) [inline]
```

Definition at line 85 of file [DerivationStencils.h](#).

```
00085 {
00086     return 1.0 / 140.0 * udata[-4 * nx] - 1.0 / 15.0 * udata[-3 * nx] +
00087           3.0 / 10.0 * udata[-2 * nx] - 1.0 / 1.0 * udata[-1 * nx] +
00088           1.0 / 4.0 * udata[0] + 3.0 / 5.0 * udata[1 * nx] -
00089           1.0 / 10.0 * udata[2 * nx] + 1.0 / 105.0 * udata[3 * nx];
00090 }
```

Referenced by [LatticePatch::derive\(\)](#), and [s7f\(\)](#).

Here is the caller graph for this function:

**6.4.2.54 s7f()** [2/2]

```
sunrealtype s7f (
    sunrealtype * udata ) [inline]
```

Definition at line 234 of file [DerivationStencils.h](#).

```
00234 { return s7f(udata, 6); }
```

References [s7f\(\)](#).

Here is the call graph for this function:



6.4.2.55 s8b() [1/2]

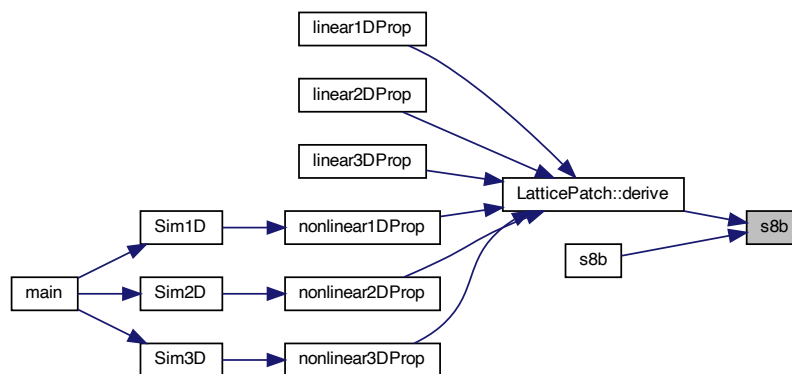
```
sunrealtype s8b (
    const sunrealtype * udata,
    int nx ) [inline]
```

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

```
00110 {
00111     return -1.0 / 168.0 * udata[-3 * nx] + 1.0 / 14.0 * udata[-2 * nx] -
00112            1.0 / 2.0 * udata[-1 * nx] - 9.0 / 20.0 * udata[0] +
00113            5.0 / 4.0 * udata[1 * nx] - 1.0 / 2.0 * udata[2 * nx] +
00114            1.0 / 6.0 * udata[3 * nx] - 1.0 / 28.0 * udata[4 * nx] +
00115            1.0 / 280.0 * udata[5 * nx];
00116 }
```

Referenced by [LatticePatch::derive\(\)](#), and [s8b\(\)](#).

Here is the caller graph for this function:



6.4.2.56 s8b() [2/2]

```
sunrealtype s8b (
    sunrealtype * udata ) [inline]
```

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

```
00238 { return s8b(udata, 6); }
```

References [s8b\(\)](#).

Here is the call graph for this function:



6.4.2.57 s8c() [1/2]

```

sunrealtype s8c (
    const sunrealtype * udata,
    int nx ) [inline]

```

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

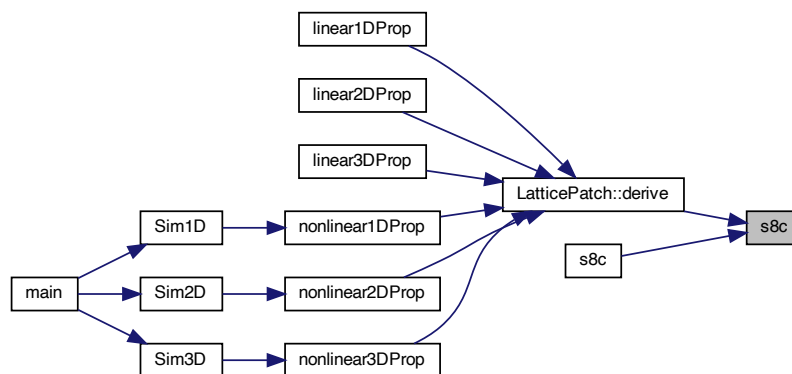
```

00104 {
00105     return 1.0 / 280.0 * udata[-4 * nx] - 4.0 / 105.0 * udata[-3 * nx] +
00106           1.0 / 5.0 * udata[-2 * nx] - 4.0 / 5.0 * udata[-1 * nx] + 0 +
00107           4.0 / 5.0 * udata[1 * nx] - 1.0 / 5.0 * udata[2 * nx] +
00108           4.0 / 105.0 * udata[3 * nx] - 1.0 / 280.0 * udata[4 * nx];
00109 }

```

Referenced by [LatticePatch::derive\(\)](#), and [s8c\(\)](#).

Here is the caller graph for this function:

**6.4.2.58 s8c()** [2/2]

```

sunrealtype s8c (
    sunrealtype * udata ) [inline]

```

Definition at line 237 of file [DerivationStencils.h](#).

```

00237 { return s8c(udata, 6); }

```

References [s8c\(\)](#).

Here is the call graph for this function:



6.4.2.59 s8f() [1/2]

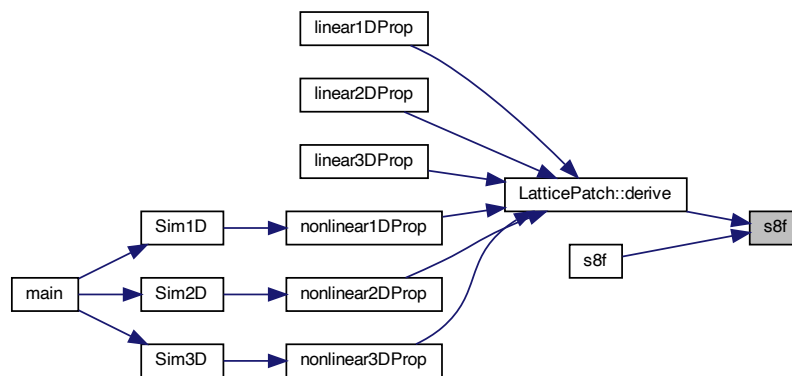
```
sunrealtype s8f (
    const sunrealtype * udata,
    int nx ) [inline]
```

Definition at line 97 of file [DerivationStencils.h](#).

```
00097 {
00098     return -1.0 / 280.0 * udata[-5 * nx] + 1.0 / 28.0 * udata[-4 * nx] -
00099            1.0 / 6.0 * udata[-3 * nx] + 1.0 / 2.0 * udata[-2 * nx] -
00100            5.0 / 4.0 * udata[-1 * nx] + 9.0 / 20.0 * udata[0] +
00101            1.0 / 2.0 * udata[1 * nx] - 1.0 / 14.0 * udata[2 * nx] +
00102            1.0 / 168.0 * udata[3 * nx];
00103 }
```

Referenced by [LatticePatch::derive\(\)](#), and [s8f\(\)](#).

Here is the caller graph for this function:



6.4.2.60 s8f() [2/2]

```
sunrealtype s8f (
    sunrealtype * udata ) [inline]
```

Definition at line 236 of file [DerivationStencils.h](#).

```
00236 { return s8f(udata, 6); }
```

References [s8f\(\)](#).

Here is the call graph for this function:



6.4.2.61 s9b() [1/2]

```
sunrealtype s9b (
    sunrealtype * udata ) [inline]
```

Definition at line 240 of file [DerivationStencils.h](#).

```
00240 { return s9b(udata, 6); }
```

References [s9b\(\)](#).

Here is the call graph for this function:

**6.4.2.62 s9b()** [2/2]

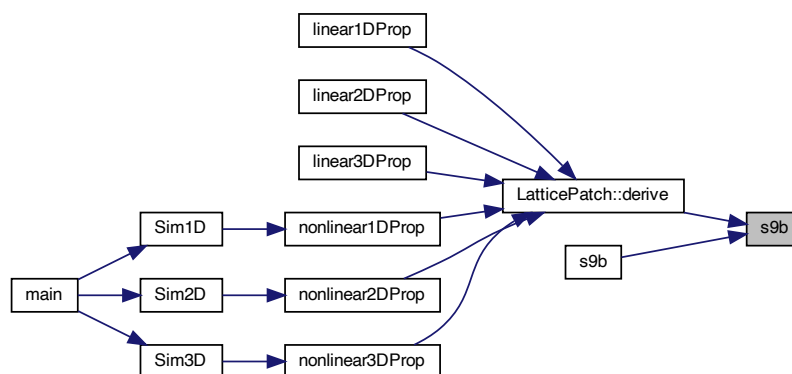
```
sunrealtype s9b (
    sunrealtype * udata,
    int nx ) [inline]
```

Definition at line 124 of file [DerivationStencils.h](#).

```
00124 {
00125     return 1.0 / 504.0 * udata[-4 * nx] - 1.0 / 42.0 * udata[-3 * nx] +
00126            1.0 / 7.0 * udata[-2 * nx] - 2.0 / 3.0 * udata[-1 * nx] -
00127            1.0 / 5.0 * udata[0] + udata[1 * nx] - 1.0 / 3.0 * udata[2 * nx] +
00128            2.0 / 21.0 * udata[3 * nx] - 1.0 / 56.0 * udata[4 * nx] +
00129            1.0 / 630.0 * udata[5 * nx];
00130 }
```

Referenced by [LatticePatch::derive\(\)](#), and [s9b\(\)](#).

Here is the caller graph for this function:



6.4.2.63 s9f() [1/2]

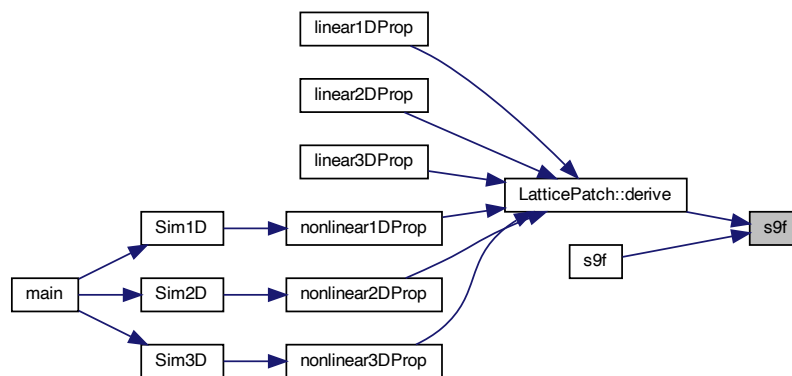
```
sunrealtype s9f (
    const sunrealtype * udata,
    int nx ) [inline]
```

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

```
00117 {
00118     return -1.0 / 630.0 * udata[-5 * nx] + 1.0 / 56.0 * udata[-4 * nx] -
00119            2.0 / 21.0 * udata[-3 * nx] + 1.0 / 3.0 * udata[-2 * nx] -
00120            1.0 / 1.0 * udata[-1 * nx] + 1.0 / 5.0 * udata[0] +
00121            2.0 / 3.0 * udata[1 * nx] - 1.0 / 7.0 * udata[2 * nx] +
00122            1.0 / 42.0 * udata[3 * nx] - 1.0 / 504.0 * udata[4 * nx];
00123 }
```

Referenced by [LatticePatch::derive\(\)](#), and [s9f\(\)](#).

Here is the caller graph for this function:



6.4.2.64 s9f() [2/2]

```
sunrealtype s9f (
    sunrealtype * udata ) [inline]
```

Definition at line 239 of file [DerivationStencils.h](#).

```
00239 { return s9f(udata, 6); }
```

References [s9f\(\)](#).

Here is the call 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 // Include Guard
00007 #ifndef DERIVATIONSTENCILS
00008 #define DERIVATIONSTENCILS
00009
00010 #include <sundials/sundials_types.h> /* definition of type sunrealtype */
00011
00012 ////////////////////////////////////////////////////////////////////
00013 // Stencils with variable nx -> data point dimension //
00014 ////////////////////////////////////////////////////////////////////
00015
00016 // Downwind (forward) differentiating
00017 inline sunrealtype s1f(sunrealtype *udata, int nx) {
00018     return -1.0 / 1.0 * udata[-1 * nx] + udata[0];
00019 }
00020 // Upwind (backward) differentiating
00021 inline sunrealtype s1b(sunrealtype *udata, int nx) {
00022     return -1.0 / 1.0 * udata[0] + udata[1 * nx];
00023 }
00024
00025 inline sunrealtype s2f(const sunrealtype *udata, int nx) {
00026     return 1.0 / 2.0 * udata[-2 * nx] - 2.0 / 1.0 * udata[-1 * nx] +
00027         3.0 / 2.0 * udata[0];
00028 }
00029 inline sunrealtype s2c(const sunrealtype *udata, int nx) {
00030     return -1.0 / 2.0 * udata[-1 * nx] + 0 + 1.0 / 2.0 * udata[1 * nx];
00031 }
00032 inline sunrealtype s2b(const sunrealtype *udata, int nx) {
00033     return -3.0 / 2.0 * udata[0] + 2.0 / 1.0 * udata[1 * nx] -
00034         1.0 / 2.0 * udata[2 * nx];
00035 }
00036 inline sunrealtype s3f(const sunrealtype *udata, int nx) {
00037     return 1.0 / 6.0 * udata[-2 * nx] - 1.0 / 1.0 * udata[-1 * nx] +
00038         1.0 / 2.0 * udata[0] + 1.0 / 3.0 * udata[1 * nx];
00039 }
00040 inline sunrealtype s3b(const sunrealtype *udata, int nx) {
00041     return -1.0 / 3.0 * udata[-1 * nx] - 1.0 / 2.0 * udata[0] + udata[1 * nx] -
00042         1.0 / 6.0 * udata[2 * nx];
00043 }
00044 inline sunrealtype s4f(const sunrealtype *udata, int nx) {
00045     return -1.0 / 12.0 * udata[-3 * nx] + 1.0 / 2.0 * udata[-2 * nx] -
00046         3.0 / 2.0 * udata[-1 * nx] + 5.0 / 6.0 * udata[0] +
00047         1.0 / 4.0 * udata[1 * nx];
00048 }
00049 inline sunrealtype s4c(const sunrealtype *udata, int nx) {
00050     return 1.0 / 12.0 * udata[-2 * nx] - 2.0 / 3.0 * udata[-1 * nx] + 0 +
00051         2.0 / 3.0 * udata[1 * nx] - 1.0 / 12.0 * udata[2 * nx];
00052 }
00053 inline sunrealtype s4b(const sunrealtype *udata, int nx) {
00054     return -1.0 / 4.0 * udata[-1 * nx] - 5.0 / 6.0 * udata[0] +
00055         3.0 / 2.0 * udata[1 * nx] - 1.0 / 2.0 * udata[2 * nx] +
00056         1.0 / 12.0 * udata[3 * nx];
00057 }
00058 inline sunrealtype s5f(const sunrealtype *udata, int nx) {
00059     return -1.0 / 30.0 * udata[-3 * nx] + 1.0 / 4.0 * udata[-2 * nx] -
00060         1.0 / 1.0 * udata[-1 * nx] + 1.0 / 3.0 * udata[0] +
00061         1.0 / 2.0 * udata[1 * nx] - 1.0 / 20.0 * udata[2 * nx];
00062 }
00063 inline sunrealtype s5b(const sunrealtype *udata, int nx) {
00064     return 1.0 / 20.0 * udata[-2 * nx] - 1.0 / 2.0 * udata[-1 * nx] -
00065         1.0 / 3.0 * udata[0] + udata[1 * nx] - 1.0 / 4.0 * udata[2 * nx] +
00066         1.0 / 30.0 * udata[3 * nx];
00067 }
00068 inline sunrealtype s6f(const sunrealtype *udata, int nx) {
00069     return 1.0 / 60.0 * udata[-4 * nx] - 2.0 / 15.0 * udata[-3 * nx] +
00070         1.0 / 2.0 * udata[-2 * nx] - 4.0 / 3.0 * udata[-1 * nx] +
00071         7.0 / 12.0 * udata[0] + 2.0 / 5.0 * udata[1 * nx] -
00072         1.0 / 30.0 * udata[2 * nx];
00073 }
00074 inline sunrealtype s6c(const sunrealtype *udata, int nx) {
00075     return -1.0 / 60.0 * udata[-3 * nx] + 3.0 / 20.0 * udata[-2 * nx] -
00076         3.0 / 4.0 * udata[-1 * nx] + 0 + 3.0 / 4.0 * udata[1 * nx] -
00077         3.0 / 20.0 * udata[2 * nx] + 1.0 / 60.0 * udata[3 * nx];
00078 }
00079 inline sunrealtype s6b(const sunrealtype *udata, int nx) {
00080     return 1.0 / 30.0 * udata[-2 * nx] - 2.0 / 5.0 * udata[-1 * nx] -
00081         7.0 / 12.0 * udata[0] + 4.0 / 3.0 * udata[1 * nx] -
00082         1.0 / 2.0 * udata[2 * nx] + 2.0 / 15.0 * udata[3 * nx] -

```

```

00083         1.0 / 60.0 * udata[4 * nx];
00084     }
00085     inline sunrealtype s7f(const sunrealtype *udata, int nx) {
00086         return 1.0 / 140.0 * udata[-4 * nx] - 1.0 / 15.0 * udata[-3 * nx] +
00087             3.0 / 10.0 * udata[-2 * nx] - 1.0 / 1.0 * udata[-1 * nx] +
00088             1.0 / 4.0 * udata[0] + 3.0 / 5.0 * udata[1 * nx] -
00089             1.0 / 10.0 * udata[2 * nx] + 1.0 / 105.0 * udata[3 * nx];
00090     }
00091     inline sunrealtype s7b(sunrealtype *udata, int nx) {
00092         return -1.0 / 105.0 * udata[-3 * nx] + 1.0 / 10.0 * udata[-2 * nx] -
00093             3.0 / 5.0 * udata[-1 * nx] - 1.0 / 4.0 * udata[0] + udata[1 * nx] -
00094             3.0 / 10.0 * udata[2 * nx] + 1.0 / 15.0 * udata[3 * nx] -
00095             1.0 / 140.0 * udata[4 * nx];
00096     }
00097     inline sunrealtype s8f(const sunrealtype *udata, int nx) {
00098         return -1.0 / 280.0 * udata[-5 * nx] + 1.0 / 28.0 * udata[-4 * nx] -
00099             1.0 / 6.0 * udata[-3 * nx] + 1.0 / 2.0 * udata[-2 * nx] -
00100             5.0 / 4.0 * udata[-1 * nx] + 9.0 / 20.0 * udata[0] +
00101             1.0 / 2.0 * udata[1 * nx] - 1.0 / 14.0 * udata[2 * nx] +
00102             1.0 / 168.0 * udata[3 * nx];
00103     }
00104     inline sunrealtype s8c(const sunrealtype *udata, int nx) {
00105         return 1.0 / 280.0 * udata[-4 * nx] - 4.0 / 105.0 * udata[-3 * nx] +
00106             1.0 / 5.0 * udata[-2 * nx] - 4.0 / 5.0 * udata[-1 * nx] + 0 +
00107             4.0 / 5.0 * udata[1 * nx] - 1.0 / 5.0 * udata[2 * nx] +
00108             4.0 / 105.0 * udata[3 * nx] - 1.0 / 280.0 * udata[4 * nx];
00109     }
00110     inline sunrealtype s8b(const sunrealtype *udata, int nx) {
00111         return -1.0 / 168.0 * udata[-3 * nx] + 1.0 / 14.0 * udata[-2 * nx] -
00112             1.0 / 2.0 * udata[-1 * nx] - 9.0 / 20.0 * udata[0] +
00113             5.0 / 4.0 * udata[1 * nx] - 1.0 / 2.0 * udata[2 * nx] +
00114             1.0 / 6.0 * udata[3 * nx] - 1.0 / 28.0 * udata[4 * nx] +
00115             1.0 / 280.0 * udata[5 * nx];
00116     }
00117     inline sunrealtype s9f(const sunrealtype *udata, int nx) {
00118         return -1.0 / 630.0 * udata[-5 * nx] + 1.0 / 56.0 * udata[-4 * nx] -
00119             2.0 / 21.0 * udata[-3 * nx] + 1.0 / 3.0 * udata[-2 * nx] -
00120             1.0 / 1.0 * udata[-1 * nx] + 1.0 / 5.0 * udata[0] +
00121             2.0 / 3.0 * udata[1 * nx] - 1.0 / 7.0 * udata[2 * nx] +
00122             1.0 / 42.0 * udata[3 * nx] - 1.0 / 504.0 * udata[4 * nx];
00123     }
00124     inline sunrealtype s9b(sunrealtype *udata, int nx) {
00125         return 1.0 / 504.0 * udata[-4 * nx] - 1.0 / 42.0 * udata[-3 * nx] +
00126             1.0 / 7.0 * udata[-2 * nx] - 2.0 / 3.0 * udata[-1 * nx] -
00127             1.0 / 5.0 * udata[0] + udata[1 * nx] - 1.0 / 3.0 * udata[2 * nx] +
00128             2.0 / 21.0 * udata[3 * nx] - 1.0 / 56.0 * udata[4 * nx] +
00129             1.0 / 630.0 * udata[5 * nx];
00130     }
00131     inline sunrealtype s10f(const sunrealtype *udata, int nx) {
00132         return 1.0 / 1260.0 * udata[-6 * nx] - 1.0 / 105.0 * udata[-5 * nx] +
00133             3.0 / 56.0 * udata[-4 * nx] - 4.0 / 21.0 * udata[-3 * nx] +
00134             1.0 / 2.0 * udata[-2 * nx] - 6.0 / 5.0 * udata[-1 * nx] +
00135             11.0 / 30.0 * udata[0] + 4.0 / 7.0 * udata[1 * nx] -
00136             3.0 / 28.0 * udata[2 * nx] + 1.0 / 63.0 * udata[3 * nx] -
00137             1.0 / 840.0 * udata[4 * nx];
00138     }
00139     inline sunrealtype s10c(const sunrealtype *udata, int nx) {
00140         return -1.0 / 1260.0 * udata[-5 * nx] + 5.0 / 504.0 * udata[-4 * nx] -
00141             5.0 / 84.0 * udata[-3 * nx] + 5.0 / 21.0 * udata[-2 * nx] -
00142             5.0 / 6.0 * udata[-1 * nx] + 0 + 5.0 / 6.0 * udata[1 * nx] -
00143             5.0 / 21.0 * udata[2 * nx] + 5.0 / 84.0 * udata[3 * nx] -
00144             5.0 / 504.0 * udata[4 * nx] + 1.0 / 1260.0 * udata[5 * nx];
00145     }
00146     inline sunrealtype s10b(const sunrealtype *udata, int nx) {
00147         return 1.0 / 840.0 * udata[-4 * nx] - 1.0 / 63.0 * udata[-3 * nx] +
00148             3.0 / 28.0 * udata[-2 * nx] - 4.0 / 7.0 * udata[-1 * nx] -
00149             11.0 / 30.0 * udata[0] + 6.0 / 5.0 * udata[1 * nx] -
00150             1.0 / 2.0 * udata[2 * nx] + 4.0 / 21.0 * udata[3 * nx] -
00151             3.0 / 56.0 * udata[4 * nx] + 1.0 / 105.0 * udata[5 * nx] -
00152             1.0 / 1260.0 * udata[6 * nx];
00153     }
00154     inline sunrealtype s11f(const sunrealtype *udata, int nx) {
00155         return 1.0 / 2772.0 * udata[-6 * nx] - 1.0 / 210.0 * udata[-5 * nx] +
00156             5.0 / 168.0 * udata[-4 * nx] - 5.0 / 42.0 * udata[-3 * nx] +
00157             5.0 / 14.0 * udata[-2 * nx] - 1.0 / 1.0 * udata[-1 * nx] +
00158             1.0 / 6.0 * udata[0] + 5.0 / 7.0 * udata[1 * nx] -
00159             5.0 / 28.0 * udata[2 * nx] + 5.0 / 126.0 * udata[3 * nx] -
00160             1.0 / 168.0 * udata[4 * nx] + 1.0 / 2310.0 * udata[5 * nx];
00161     }
00162     inline sunrealtype s11b(sunrealtype *udata, int nx) {
00163         return -1.0 / 2310.0 * udata[-5 * nx] + 1.0 / 168.0 * udata[-4 * nx] -
00164             5.0 / 126.0 * udata[-3 * nx] + 5.0 / 28.0 * udata[-2 * nx] -
00165             5.0 / 7.0 * udata[-1 * nx] - 1.0 / 6.0 * udata[0] + udata[1 * nx] -
00166             5.0 / 14.0 * udata[2 * nx] + 5.0 / 42.0 * udata[3 * nx] -
00167             5.0 / 168.0 * udata[4 * nx] + 1.0 / 210.0 * udata[5 * nx] -
00168             1.0 / 2772.0 * udata[6 * nx];
00169     }

```

```

00170 inline sunrealtype s12f(const sunrealtype *udata, int nx) {
00171     return -1.0 / 5544.0 * udata[-7 * nx] + 1.0 / 396.0 * udata[-6 * nx] -
00172         1.0 / 60.0 * udata[-5 * nx] + 5.0 / 72.0 * udata[-4 * nx] -
00173         5.0 / 24.0 * udata[-3 * nx] + 1.0 / 2.0 * udata[-2 * nx] -
00174         7.0 / 6.0 * udata[-1 * nx] + 13.0 / 42.0 * udata[0] +
00175         5.0 / 8.0 * udata[1 * nx] - 5.0 / 36.0 * udata[2 * nx] +
00176         1.0 / 36.0 * udata[3 * nx] - 1.0 / 264.0 * udata[4 * nx] +
00177         1.0 / 3960.0 * udata[5 * nx];
00178 }
00179 inline sunrealtype s12c(const sunrealtype *udata, int nx) {
00180     return 1.0 / 5544.0 * udata[-6 * nx] - 1.0 / 385.0 * udata[-5 * nx] +
00181         1.0 / 56.0 * udata[-4 * nx] - 5.0 / 63.0 * udata[-3 * nx] +
00182         15.0 / 56.0 * udata[-2 * nx] - 6.0 / 7.0 * udata[-1 * nx] + 0 +
00183         6.0 / 7.0 * udata[1 * nx] - 15.0 / 56.0 * udata[2 * nx] +
00184         5.0 / 63.0 * udata[3 * nx] - 1.0 / 56.0 * udata[4 * nx] +
00185         1.0 / 385.0 * udata[5 * nx] - 1.0 / 5544.0 * udata[6 * nx];
00186 }
00187 inline sunrealtype s12b(const sunrealtype *udata, int nx) {
00188     return -1.0 / 3960.0 * udata[-5 * nx] + 1.0 / 264.0 * udata[-4 * nx] -
00189         1.0 / 36.0 * udata[-3 * nx] + 5.0 / 36.0 * udata[-2 * nx] -
00190         5.0 / 36.0 * udata[-1 * nx] - 13.0 / 42.0 * udata[0] +
00191         7.0 / 6.0 * udata[1 * nx] - 1.0 / 2.0 * udata[2 * nx] +
00192         5.0 / 24.0 * udata[3 * nx] - 5.0 / 72.0 * udata[4 * nx] +
00193         1.0 / 60.0 * udata[5 * nx] - 1.0 / 396.0 * udata[6 * nx] +
00194         1.0 / 5544.0 * udata[7 * nx];
00195 }
00196 inline sunrealtype s13f(const sunrealtype *udata, int nx) {
00197     return -1.0 / 12012.0 * udata[-7 * nx] + 1.0 / 792.0 * udata[-6 * nx] -
00198         1.0 / 110.0 * udata[-5 * nx] + 1.0 / 24.0 * udata[-4 * nx] -
00199         5.0 / 36.0 * udata[-3 * nx] + 3.0 / 8.0 * udata[-2 * nx] -
00200         1.0 / 1.0 * udata[-1 * nx] + 1.0 / 7.0 * udata[0] +
00201         3.0 / 4.0 * udata[1 * nx] - 5.0 / 24.0 * udata[2 * nx] +
00202         1.0 / 18.0 * udata[3 * nx] - 1.0 / 88.0 * udata[4 * nx] +
00203         1.0 / 660.0 * udata[5 * nx] - 1.0 / 10296.0 * udata[6 * nx];
00204 }
00205 inline sunrealtype s13b(const sunrealtype *udata, int nx) {
00206     return 1.0 / 10296.0 * udata[-6 * nx] - 1.0 / 660.0 * udata[-5 * nx] +
00207         1.0 / 88.0 * udata[-4 * nx] - 1.0 / 18.0 * udata[-3 * nx] +
00208         5.0 / 24.0 * udata[-2 * nx] - 3.0 / 4.0 * udata[-1 * nx] -
00209         1.0 / 7.0 * udata[0] + udata[1 * nx] - 3.0 / 8.0 * udata[2 * nx] +
00210         5.0 / 36.0 * udata[3 * nx] - 1.0 / 24.0 * udata[4 * nx] +
00211         1.0 / 110.0 * udata[5 * nx] - 1.0 / 792.0 * udata[6 * nx] +
00212         1.0 / 12012.0 * udata[7 * nx];
00213 }
00214
00215 //////////////////////////////////////////////////
00216 // Stencils with nx fixed to 6//
00217 //////////////////////////////////////////////////
00218
00219 inline sunrealtype s1f(sunrealtype *udata) { return s1f(udata, 6); }
00220 inline sunrealtype s1b(sunrealtype *udata) { return s1b(udata, 6); }
00221 inline sunrealtype s2f(sunrealtype *udata) { return s2f(udata, 6); }
00222 inline sunrealtype s2c(sunrealtype *udata) { return s2c(udata, 6); }
00223 inline sunrealtype s2b(sunrealtype *udata) { return s2b(udata, 6); }
00224 inline sunrealtype s3f(sunrealtype *udata) { return s3f(udata, 6); }
00225 inline sunrealtype s3b(sunrealtype *udata) { return s3b(udata, 6); }
00226 inline sunrealtype s4f(sunrealtype *udata) { return s4f(udata, 6); }
00227 inline sunrealtype s4c(sunrealtype *udata) { return s4c(udata, 6); }
00228 inline sunrealtype s4b(sunrealtype *udata) { return s4b(udata, 6); }
00229 inline sunrealtype s5f(sunrealtype *udata) { return s5f(udata, 6); }
00230 inline sunrealtype s5b(sunrealtype *udata) { return s5b(udata, 6); }
00231 inline sunrealtype s6f(sunrealtype *udata) { return s6f(udata, 6); }
00232 inline sunrealtype s6c(sunrealtype *udata) { return s6c(udata, 6); }
00233 inline sunrealtype s6b(sunrealtype *udata) { return s6b(udata, 6); }
00234 inline sunrealtype s7f(sunrealtype *udata) { return s7f(udata, 6); }
00235 inline sunrealtype s7b(sunrealtype *udata) { return s7b(udata, 6); }
00236 inline sunrealtype s8f(sunrealtype *udata) { return s8f(udata, 6); }
00237 inline sunrealtype s8c(sunrealtype *udata) { return s8c(udata, 6); }
00238 inline sunrealtype s8b(sunrealtype *udata) { return s8b(udata, 6); }
00239 inline sunrealtype s9f(sunrealtype *udata) { return s9f(udata, 6); }
00240 inline sunrealtype s9b(sunrealtype *udata) { return s9b(udata, 6); }
00241 inline sunrealtype s10f(sunrealtype *udata) { return s10f(udata, 6); }
00242 inline sunrealtype s10c(sunrealtype *udata) { return s10c(udata, 6); }
00243 inline sunrealtype s10b(sunrealtype *udata) { return s10b(udata, 6); }
00244 inline sunrealtype s11f(sunrealtype *udata) { return s11f(udata, 6); }
00245 inline sunrealtype s11b(sunrealtype *udata) { return s11b(udata, 6); }
00246 inline sunrealtype s12f(sunrealtype *udata) { return s12f(udata, 6); }
00247 inline sunrealtype s12c(sunrealtype *udata) { return s12c(udata, 6); }
00248 inline sunrealtype s12b(sunrealtype *udata) { return s12b(udata, 6); }
00249 inline sunrealtype s13f(sunrealtype *udata) { return s13f(udata, 6); }
00250 inline sunrealtype s13b(sunrealtype *udata) { return s13b(udata, 6); }
00251
00252 // End of Includeguard
00253 #endif

```

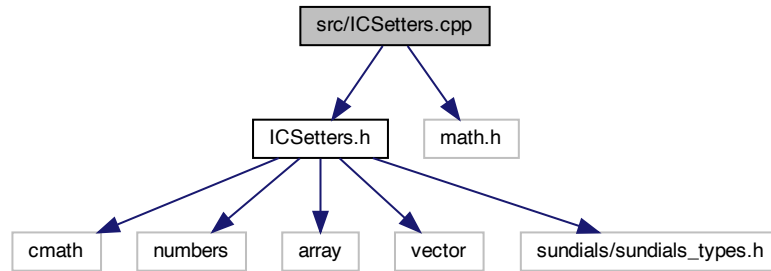
6.6 src/ICSetters.cpp File Reference

Implementation of the plane wave and Gaussian wave packets in 1D, 2D, 3D.

```
#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 in 1D, 2D, 3D.

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 in 1D, 2D,
00004 /// 3D
00005 //////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
00006
00007 #include "ICSetters.h"
00008
00009 #include <math.h>
00010
00011 /** PlaneWave1D construction with */
00012 PlaneWave1D::PlaneWave1D(vector<sunrealtype> k, vector<sunrealtype> p,
00013                          vector<sunrealtype> 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, const sunrealtype z,
00028                             sunrealtype *pTo6Space) const {
00029     const sunrealtype wavelength =
00030         sqrt(kx * kx + ky * ky + kz * kz); /** \f$ 1/\lambda \f$ */
00031     const sunrealtype kScalarX = (kx * x + ky * y + kz * z) * 2 *
  
```



```

00032             numbers::pi; /* \f$ 2\pi \ \vec{k} \cdot \vec{x} \f$ */
00033 // Plane wave definition
00034 const array<sunrealtype, 3> E{{ /* E-field vector */
00035     px * cos(kScalarX - phix), /* \f$ E_x \f$ */
00036     py * cos(kScalarX - phiy), /* \f$ E_y \f$ */
00037     pz * cos(kScalarX - phiz)}}; /* \f$ E_z \f$ */
00038 // Put E-field into space
00039 pTo6Space[0] += E[0];
00040 pTo6Space[1] += E[1];
00041 pTo6Space[2] += E[2];
00042 // and B-field
00043 pTo6Space[3] += (ky * E[2] - kz * E[1]) / wavelength;
00044 pTo6Space[4] += (kz * E[0] - kx * E[2]) / wavelength;
00045 pTo6Space[5] += (kx * E[1] - ky * E[0]) / wavelength;
00046 }
00047
00048 /** PlaneWave2D construction with */
00049 PlaneWave2D::PlaneWave2D(vector<sunrealtype> k, vector<sunrealtype> p,
00050     vector<sunrealtype> phi) {
00051     kx = k[0]; /** - wavevectors \f$ k_x \f$ */
00052     ky = k[1]; /** - \f$ k_y \f$ */
00053     kz = k[2]; /** - \f$ k_z \f$ normalized to \f$ 1/\lambda \f$ */
00054 // Amplitude bug: lower by factor 9
00055 px = p[0] / 9; /** - amplitude (polarization) in x-direction \f$ p_x \f$ */
00056 py = p[1] / 9; /** - amplitude (polarization) in y-direction \f$ p_y \f$ */
00057 pz = p[2] / 9; /** - amplitude (polarization) in z-direction \f$ p_z \f$ */
00058 phix = phi[0]; /** - phase shift in x-direction \f$ \phi_x \f$ */
00059 phiy = phi[1]; /** - phase shift in y-direction \f$ \phi_y \f$ */
00060 phiz = phi[2]; /** - phase shift in z-direction \f$ \phi_z \f$ */
00061 }
00062
00063 /** PlaneWave2D implementation in space */
00064 void PlaneWave2D::addToSpace(const sunrealtype x, const sunrealtype y, const sunrealtype z,
00065     sunrealtype *pTo6Space) const {
00066     const sunrealtype wavelength =
00067         sqrt(kx * kx + ky * ky + kz * kz); /* \f$ 1/\lambda \f$ */
00068     const sunrealtype kScalarX = (kx * x + ky * y + kz * z) * 2 *
00069         numbers::pi; /* \f$ 2\pi \ \vec{k} \cdot \vec{x} \f$ */
00070 // Plane wave definition
00071 const array<sunrealtype, 3> E{{ /* E-field vector */
00072     px * cos(kScalarX - phix), /* \f$ E_x \f$ */
00073     py * cos(kScalarX - phiy), /* \f$ E_y \f$ */
00074     pz * cos(kScalarX - phiz)}}; /* \f$ E_z \f$ */
00075 // Put E-field into space
00076 pTo6Space[0] += E[0];
00077 pTo6Space[1] += E[1];
00078 pTo6Space[2] += E[2];
00079 // and B-field
00080 pTo6Space[3] += (ky * E[2] - kz * E[1]) / wavelength;
00081 pTo6Space[4] += (kz * E[0] - kx * E[2]) / wavelength;
00082 pTo6Space[5] += (kx * E[1] - ky * E[0]) / wavelength;
00083 }
00084
00085 /** PlaneWave3D construction with */
00086 PlaneWave3D::PlaneWave3D(vector<sunrealtype> k, vector<sunrealtype> p,
00087     vector<sunrealtype> phi) {
00088     kx = k[0]; /** - wavevectors \f$ k_x \f$ */
00089     ky = k[1]; /** - \f$ k_y \f$ */
00090     kz = k[2]; /** - \f$ k_z \f$ normalized to \f$ 1/\lambda \f$ */
00091 px = p[0]; /** - amplitude (polarization) in x-direction \f$ p_x \f$ */
00092 py = p[1]; /** - amplitude (polarization) in y-direction \f$ p_y \f$ */
00093 pz = p[2]; /** - amplitude (polarization) in z-direction \f$ p_z \f$ */
00094 phix = phi[0]; /** - phase shift in x-direction \f$ \phi_x \f$ */
00095 phiy = phi[1]; /** - phase shift in y-direction \f$ \phi_y \f$ */
00096 phiz = phi[2]; /** - phase shift in z-direction \f$ \phi_z \f$ */
00097 }
00098
00099 /** PlaneWave3D implementation in space */
00100 void PlaneWave3D::addToSpace(sunrealtype x, sunrealtype y, sunrealtype z,
00101     sunrealtype *pTo6Space) const {
00102     const sunrealtype wavelength =
00103         sqrt(kx * kx + ky * ky + kz * kz); /* \f$ 1/\lambda \f$ */
00104     const sunrealtype kScalarX = (kx * x + ky * y + kz * z) * 2 *
00105         numbers::pi; /* \f$ 2\pi \ \vec{k} \cdot \vec{x} \f$ */
00106 // Plane wave definition
00107 const array<sunrealtype, 3> E{{ /* E-field vector \f$ \vec{E} \f$ */
00108     px * cos(kScalarX - phix), /* \f$ E_x \f$ */
00109     py * cos(kScalarX - phiy), /* \f$ E_y \f$ */
00110     pz * cos(kScalarX - phiz)}}; /* \f$ E_z \f$ */
00111 // Put E-field into space
00112 pTo6Space[0] += E[0];
00113 pTo6Space[1] += E[1];
00114 pTo6Space[2] += E[2];
00115 // and B-field
00116 pTo6Space[3] += (ky * E[2] - kz * E[1]) / wavelength;
00117 pTo6Space[4] += (kz * E[0] - kx * E[2]) / wavelength;
00118 pTo6Space[5] += (kx * E[1] - ky * E[0]) / wavelength;

```

```

00119 }
00120
00121 /** Gauss1D construction with */
00122 Gauss1D::Gauss1D(vector<sunrealtype> k, vector<sunrealtype> p,
00123                 vector<sunrealtype> xo, sunrealtype phig_,
00124                 vector<sunrealtype> phi) {
00125     kx = k[0]; /** - wavevectors \f$ k_x \f$ */
00126     ky = k[1]; /** - \f$ k_y \f$ */
00127     kz = k[2]; /** - \f$ k_z \f$ normalized to \f$ 1/\lambda \f$ */
00128     px = p[0]; /** - amplitude (polarization) in x-direction */
00129     py = p[1]; /** - amplitude (polarization) in y-direction */
00130     pz = p[2]; /** - amplitude (polarization) in z-direction */
00131     phix = phi[0]; /** - phase shift in x-direction */
00132     phiy = phi[1]; /** - phase shift in y-direction */
00133     phiz = phi[2]; /** - phase shift in z-direction */
00134     phig = phig_; /** - width */
00135     x0x = xo[0]; /** - shift from origin in x-direction */
00136     x0y = xo[1]; /** - shift from origin in y-direction */
00137     x0z = xo[2]; /** - shift from origin in z-direction */
00138 }
00139
00140 /** Gauss1D implementation in space */
00141 void Gauss1D::addToSpace(sunrealtype x, sunrealtype y, sunrealtype z,
00142                        sunrealtype *pTo6Space) const {
00143     const sunrealtype wavelength =
00144         sqrt(kx * kx + ky * ky + kz * kz); /** \f$ 1/\lambda \f$ */
00145     x = x - x0x; /** x-coordinate minus shift from origin */
00146     y = y - x0y; /** y-coordinate minus shift from origin */
00147     z = z - x0z; /** z-coordinate minus shift from origin */
00148     const sunrealtype kScalarX = (kx * x + ky * y + kz * z) * 2 *
00149         numbers::pi; /** \f$ 2\pi \cdot \vec{k} \cdot \vec{x} \f$ */
00150     const sunrealtype envelopeAmp =
00151         exp(-(x * x + y * y + z * z) / phig / phig); /** enveloping Gauss shape */
00152     /** Gaussian wave definition
00153     const array<sunrealtype, 3> E{
00154         {
00155             px * cos(kScalarX - phix) * envelopeAmp, /** \f$ E_x \f$ */
00156             py * cos(kScalarX - phiy) * envelopeAmp, /** \f$ E_y \f$ */
00157             pz * cos(kScalarX - phiz) * envelopeAmp}; /** \f$ E_z \f$ */
00158     /** Put E-field into space
00159     pTo6Space[0] += E[0];
00160     pTo6Space[1] += E[1];
00161     pTo6Space[2] += E[2];
00162     /** and B-field
00163     pTo6Space[3] += (ky * E[2] - kz * E[1]) / wavelength;
00164     pTo6Space[4] += (kz * E[0] - kx * E[2]) / wavelength;
00165     pTo6Space[5] += (kx * E[1] - ky * E[0]) / wavelength;
00166 }
00167
00168 /** Gauss2D construction with */
00169 Gauss2D::Gauss2D(vector<sunrealtype> dis_, vector<sunrealtype> axis_,
00170                 sunrealtype Amp_, sunrealtype phip_, sunrealtype w0_,
00171                 sunrealtype zr_, sunrealtype Ph0_, sunrealtype PhA_) {
00172     dis = dis_; /** - center it approaches */
00173     axis = axis_; /** - direction form where it comes */
00174     Amp = Amp_; /** - amplitude */
00175     phip = phip_; /** - polarization rotation from TE-mode */
00176     w0 = w0_; /** - taille */
00177     zr = zr_; /** - Rayleigh length */
00178     Ph0 = Ph0_; /** - beam center */
00179     PhA = PhA_; /** - beam length */
00180     A1 = Amp * cos(phip); /** amplitude in z-direction
00181     A2 = Amp * sin(phip); /** amplitude on xy-plane
00182     lambda = numbers::pi * w0 * w0 / zr; /** formula for wavelength
00183 }
00184
00185 void Gauss2D::addToSpace(sunrealtype x, sunrealtype y, sunrealtype z,
00186                        sunrealtype *pTo6Space) const {
00187     /** \f$ \vec{x} = \vec{x}_0 - \vec{dis} \f$ // coordinates minus distance to
00188     /** origin
00189     x -= dis[0];
00190     y -= dis[1];
00191     /** z -= dis[2];
00192     z = NAN;
00193     /** \f$ z_g = \vec{x} \cdot \vec{e}_g \f$ projection on propagation axis
00194     const sunrealtype zg =
00195         x * axis[0] + y * axis[1]; /** +z*axis[2]; // =z-z0 -> propagation
00196         /** direction, minus origin
00197     /** \f$ r = \sqrt{\vec{x}^2 - z_g^2} \f$ -> pythagoras of radius minus
00198     /** projection on prop axis
00199     const sunrealtype r = sqrt((x * x + y * y /** +z*z) -
00200         zg * zg); /** radial distance to propagation axis
00201     /** \f$ w(z) = w0 \sqrt{1 + (z_g/z_R)^2} \f$
00202     const sunrealtype wz = w0 * sqrt(1 + (zg * zg / zr / zr)); /** waist at position z
00203     /** \f$ g(z) = \text{atan}(z_g/z_r) \f$
00204     const sunrealtype gz = atan(zg / zr); /** Gouy phase
00205     /** \f$ R(z) = z_g * (1 + (z_r/z_g)^2) \f$

```

```

00206     sunrealtype Rz = NAN; // beam curvature
00207     if (zg != 0)
00208         Rz = zg * (1 + (zr * zr / zg / zg));
00209     else
00210         Rz = 1e308;
00211     // wavenumber \f$ k = 2\pi/\lambda \f$
00212     const sunrealtype k = 2 * numbers::pi / lambda;
00213     // \f$ \Phi_F = kr^2/(2*R(z))+g(z)-kz_g \f$
00214     const sunrealtype PhF =
00215         -k * r * r / (2 * Rz) + gz - k * zg; // to be inserted into cosine
00216     // \f$ G = \sqrt{w_0/w_z}\{e^{-(r/w(z))^2}\}e^{\{zg-Ph_0\}^2/PhA^2}\cos(PhF) \f$ -
00217     // CNode is a diva, no chance to remove the square in the second exponential
00218     // -> h too small
00219     const sunrealtype G2D = sqrt(w0 / wz) * exp(-r * r / wz / wz) *
00220         exp(-(zg - Ph0) * (zg - Ph0) / PhA / PhA) *
00221         cos(PhF); // gauss shape
00222     // \f$ c_\alpha = \vec{e}_x \cdot \vec{axis} \f$
00223     // projection components; do like this for CNode convergence -> otherwise
00224     // results in machine error values for non-existent field components if
00225     // axis[0] and axis[1] are given
00226     const sunrealtype ca =
00227         axis[0]; // x-component of propagation axis which is given as parameter
00228     const sunrealtype sa = sqrt(1 - ca * ca); // no z-component for 2D propagation
00229     // E-field to space; polarization in xy-plane (A2) is projection of
00230     // z-polarization (A1) on x- and y-directions
00231     pTo6Space[0] += sa * (G2D * A2);
00232     pTo6Space[1] += -ca * (G2D * A2);
00233     pTo6Space[2] += G2D * A1;
00234     // B-field -> negative derivative wrt polarization shift of E-field
00235     pTo6Space[3] += -sa * (G2D * A1);
00236     pTo6Space[4] += ca * (G2D * A1);
00237     pTo6Space[5] += G2D * A2;
00238 }
00239
00240 /** Gauss3D construction with */
00241 Gauss3D::Gauss3D(vector<sunrealtype> dis_, vector<sunrealtype> axis_,
00242     sunrealtype Amp_,
00243     // vector<sunrealtype> pol_,
00244     sunrealtype phip_, sunrealtype w0_, sunrealtype zr_,
00245     sunrealtype Ph0_, sunrealtype PhA_) {
00246     dis = dis_; /** - center it approaches */
00247     axis = axis_; /** - direction from where it comes */
00248     Amp = Amp_; /** - amplitude */
00249     // pol=pol_;
00250     phip = phip_; /** - polarization rotation form TE-mode */
00251     w0 = w0_; /** - taille */
00252     zr = zr_; /** - Rayleigh length */
00253     Ph0 = Ph0_; /** - beam center */
00254     PhA = PhA_; /** - beam length */
00255     lambda = numbers::pi * w0 * w0 / zr;
00256     A1 = Amp * cos(phia);
00257     A2 = Amp * sin(phia);
00258 }
00259
00260 /** Gauss3D implementation in space */
00261 void Gauss3D::addToSpace(sunrealtype x, sunrealtype y, sunrealtype z,
00262     sunrealtype *pTo6Space) const {
00263     x -= dis[0];
00264     y -= dis[1];
00265     z -= dis[2];
00266     const sunrealtype zg = x * axis[0] + y * axis[1] + z * axis[2];
00267     const sunrealtype r = sqrt((x * x + y * y + z * z) - zg * zg);
00268     const sunrealtype wz = w0 * sqrt(1 + (zg * zg / zr / zr));
00269     const sunrealtype gz = atan(zg / zr);
00270     sunrealtype Rz = NAN;
00271     if (zg != 0)
00272         Rz = zg * (1 + (zr * zr / zg / zg));
00273     else
00274         Rz = 1e308;
00275     const sunrealtype k = 2 * numbers::pi / lambda;
00276     const sunrealtype PhF = -k * r * r / (2 * Rz) + gz - k * zg;
00277     const sunrealtype G3D = (w0 / wz) * exp(-r * r / wz / wz) *
00278         exp(-(zg - Ph0) * (zg - Ph0) / PhA / PhA) * cos(PhF);
00279     const sunrealtype ca = axis[0];
00280     const sunrealtype sa = sqrt(1 - ca * ca);
00281     pTo6Space[0] += sa * (G3D * A2);
00282     pTo6Space[1] += -ca * (G3D * A2);
00283     pTo6Space[2] += G3D * A1;
00284     pTo6Space[3] += -sa * (G3D * A1);
00285     pTo6Space[4] += ca * (G3D * A1);
00286     pTo6Space[5] += G3D * A2;
00287 }
00288
00289 /** Evaluate lattice point values to zero and add field values */
00290 void ICSetter::eval(sunrealtype x, sunrealtype y, sunrealtype z,
00291     sunrealtype *pTo6Space) {
00292     pTo6Space[0] = 0;

```

```

00293 pTo6Space[1] = 0;
00294 pTo6Space[2] = 0;
00295 pTo6Space[3] = 0;
00296 pTo6Space[4] = 0;
00297 pTo6Space[5] = 0;
00298 add(x, y, z, pTo6Space);
00299 }
00300
00301 /** Add all initial field values to the lattice space */
00302 void ICSetter::add(sunrealtype x, unrealtype y, unrealtype z,
00303                  unrealtype *pTo6Space) {
00304     for (const auto wave : planeWaves1D)
00305         wave.addToSpace(x, y, z, pTo6Space);
00306     for (const auto wave : planeWaves2D)
00307         wave.addToSpace(x, y, z, pTo6Space);
00308     for (const auto wave : planeWaves3D)
00309         wave.addToSpace(x, y, z, pTo6Space);
00310     for (const auto wave : gauss1Ds)
00311         wave.addToSpace(x, y, z, pTo6Space);
00312     for (const auto wave : gauss2Ds)
00313         wave.addToSpace(x, y, z, pTo6Space);
00314     for (const auto wave : gauss3Ds)
00315         wave.addToSpace(x, y, z, pTo6Space);
00316 }
00317
00318 /** Add plane waves in 1D to their container vector */
00319 void ICSetter::addPlaneWave1D(vector<unrealtype> k, vector<unrealtype> p,
00320                               vector<unrealtype> phi) {
00321     planeWaves1D.emplace_back(PlaneWave1D(k, p, phi));
00322 }
00323
00324 /** Add plane waves in 2D to their container vector */
00325 void ICSetter::addPlaneWave2D(vector<unrealtype> k, vector<unrealtype> p,
00326                               vector<unrealtype> phi) {
00327     planeWaves2D.emplace_back(PlaneWave2D(k, p, phi));
00328 }
00329
00330 /** Add plane waves in 3D to their container vector */
00331 void ICSetter::addPlaneWave3D(vector<unrealtype> k, vector<unrealtype> p,
00332                               vector<unrealtype> phi) {
00333     planeWaves3D.emplace_back(PlaneWave3D(k, p, phi));
00334 }
00335
00336 /** Add Gaussian waves in 1D to their container vector */
00337 void ICSetter::addGauss1D(vector<unrealtype> k, vector<unrealtype> p,
00338                           vector<unrealtype> xo, unrealtype phig_,
00339                           vector<unrealtype> phi) {
00340     gauss1Ds.emplace_back(Gauss1D(k, p, xo, phig_, phi));
00341 }
00342
00343 /** Add Gaussian waves in 2D to their container vector */
00344 void ICSetter::addGauss2D(vector<unrealtype> dis_, vector<unrealtype> axis_,
00345                           unrealtype Amp_, unrealtype phip_, unrealtype w0_,
00346                           unrealtype zr_, unrealtype Ph0_, unrealtype PhA_) {
00347     gauss2Ds.emplace_back(
00348         Gauss2D(dis_, axis_, Amp_, phip_, w0_, zr_, Ph0_, PhA_));
00349 }
00350
00351 /** Add Gaussian waves in 3D to their container vector */
00352 void ICSetter::addGauss3D(vector<unrealtype> dis_, vector<unrealtype> axis_,
00353                           unrealtype Amp_, unrealtype phip_, unrealtype w0_,
00354                           unrealtype zr_, unrealtype Ph0_, unrealtype PhA_) {
00355     gauss3Ds.emplace_back(
00356         Gauss3D(dis_, axis_, Amp_, phip_, w0_, zr_, Ph0_, PhA_));
00357 }

```

6.8 src/ICSetters.h File Reference

Declaration of the plane wave and Gaussian wave packets in 1D, 2D, 3D.

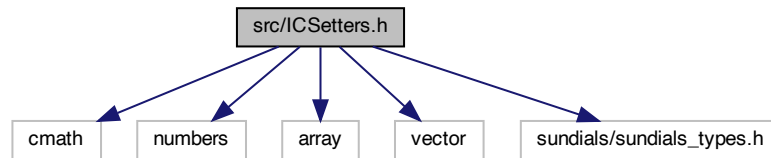
```

#include <cmath>
#include <numbers>
#include <array>
#include <vector>

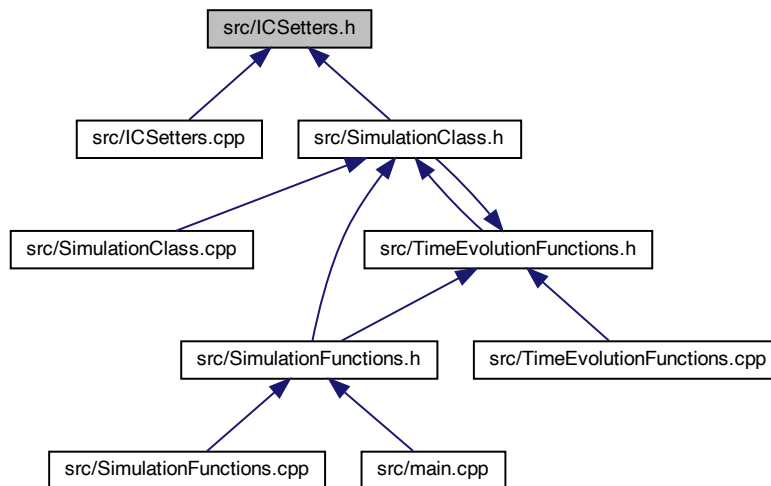
```

```
#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 waves in 1D
- class [Gauss2D](#)
class for Gaussian waves in 2D
- class [Gauss3D](#)
class for Gaussian waves 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 in 1D, 2D, 3D.

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 in 1D, 2D, 3D
00004 ///////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
00005
00006 // Include Guard
00007 #ifndef ICSETTERS
00008 #define ICSETTERS
00009
00010 // math, constants, vector, and array
00011 #include <cmath>
00012 // #include <mathimf.h>
00013 #include <numbers>
00014 #include <array>
00015 #include <vector>
00016
00017 #include <sundials/sundials_types.h> /* definition of type sunrealtype */
00018
00019 using namespace std;
00020
00021 /** @brief super-class for plane waves
00022  *
00023  * They are given in the form  $\vec{E} = \vec{E}_0 \cos(\vec{k} \cdot \vec{x} - \vec{\phi})$ 
00024  *  $\vec{\phi} = \vec{\phi}_x \hat{x} + \vec{\phi}_y \hat{y} + \vec{\phi}_z \hat{z}$ 
00025  */
00026 class PlaneWave {
00027 protected:
00028     /// wavenumber  $k_x$ 
00029     sunrealtype kx;
00030     /// wavenumber  $k_y$ 
00031     sunrealtype ky;
00032     /// wavenumber  $k_z$ 
00033     sunrealtype kz;
00034     /// polarization & amplitude in x-direction,  $p_x$ 
00035     sunrealtype px;
00036     /// polarization & amplitude in y-direction,  $p_y$ 
00037     sunrealtype py;
00038     /// polarization & amplitude in z-direction,  $p_z$ 
00039     sunrealtype pz;
00040     /// phase shift in x-direction,  $\phi_x$ 
00041     sunrealtype phix;
00042     /// phase shift in y-direction,  $\phi_y$ 
00043     sunrealtype phiy;
00044     /// phase shift in z-direction,  $\phi_z$ 
00045     sunrealtype phiz;
00046 };
00047
00048 /** @brief class for plane waves in 1D */
00049 class PlaneWave1D : public PlaneWave {
00050 public:
00051     /// construction with default parameters
00052     PlaneWave1D(vector<sunrealtype> k = {1, 0, 0},
00053                vector<sunrealtype> p = {0, 0, 1},
00054                vector<sunrealtype> phi = {0, 0, 0});
00055     /// function for the actual implementation in the lattice
00056     void addToSpace(sunrealtype x, sunrealtype y, sunrealtype z,
00057                    sunrealtype *pTo6Space) const;
00058 };
00059
00060 /** @brief class for plane waves in 2D */
00061 class PlaneWave2D : public PlaneWave {
00062 public:
00063     /// construction with default parameters
00064     PlaneWave2D(vector<sunrealtype> k = {1, 0, 0},
00065                vector<sunrealtype> p = {0, 0, 1},
00066                vector<sunrealtype> phi = {0, 0, 0});
00067     /// function for the actual implementation in the lattice
00068     void addToSpace(sunrealtype x, sunrealtype y, sunrealtype z,
00069                    sunrealtype *pTo6Space) const;
00070 };
00071
```

```

00070
00071 /** @brief class for plane waves in 3D */
00072 class PlaneWave3D : public PlaneWave {
00073 public:
00074     /// construction with default parameters
00075     PlaneWave3D(vector<sunrealtype> k = {1, 0, 0},
00076                 vector<sunrealtype> p = {0, 0, 1},
00077                 vector<sunrealtype> phi = {0, 0, 0});
00078     /// function for the actual implementation in space
00079     void addToSpace(sunrealtype x, sunrealtype y, sunrealtype z,
00080                     sunrealtype *pTo6Space) const;
00081 };
00082
00083 /** @brief class for Gaussian waves in 1D
00084 *
00085 * They are given in the form  $\vec{E} = \vec{p} \exp \left( -(\vec{x} - \vec{x}_0)^2 / \Phi_g^2 \right) \cos(\vec{k} \cdot \vec{x})$ 
00086 *  $-\left( \vec{x} - \vec{x}_0 \right)^2 / \Phi_g^2$ 
00087 */
00088 class Gauss1D {
00089 private:
00090     /// wavenumber  $k_x$ 
00091     sunrealtype kx;
00092     /// wavenumber  $k_y$ 
00093     sunrealtype ky;
00094     /// wavenumber  $k_z$ 
00095     sunrealtype kz;
00096     /// polarization & amplitude in x-direction,  $p_x$ 
00097     sunrealtype px;
00098     /// polarization & amplitude in y-direction,  $p_y$ 
00099     sunrealtype py;
00100     /// polarization & amplitude in z-direction,  $p_z$ 
00101     sunrealtype pz;
00102     /// phase shift in x-direction,  $\phi_x$ 
00103     sunrealtype phix;
00104     /// phase shift in y-direction,  $\phi_y$ 
00105     sunrealtype phiy;
00106     /// phase shift in z-direction,  $\phi_z$ 
00107     sunrealtype phiz;
00108     /// center of pulse in x-direction,  $x_0$ 
00109     sunrealtype x0;
00110     /// center of pulse in y-direction,  $y_0$ 
00111     sunrealtype y0;
00112     /// center of pulse in z-direction,  $z_0$ 
00113     sunrealtype z0;
00114     /// pulse width  $\Phi_g$ 
00115     sunrealtype phig;
00116 public:
00117     /// construction with default parameters
00118     Gauss1D(vector<sunrealtype> k = {1, 0, 0}, vector<sunrealtype> p = {0, 0, 1},
00119             vector<sunrealtype> xo = {0, 0, 0}, sunrealtype phig_ = 1.01,
00120             vector<sunrealtype> phi = {0, 0, 0});
00121     /// function for the actual implementation in space
00122     void addToSpace(sunrealtype x, sunrealtype y, sunrealtype z,
00123                     sunrealtype *pTo6Space) const;
00124 public:
00125 };
00126
00127 /** @brief class for Gaussian waves in 2D
00128 *
00129 * They are given in the form
00130 *  $\vec{E} = A \vec{\epsilon} \sqrt{\frac{\omega_0}{\omega(z)}} \exp \left( -\left( \frac{z_g - \Phi_0}{\Phi_A} \right)^2 \right) \cos \left( \frac{k}{r^2} R(z) + g(z) - k \frac{z_g}{r} \right)$ 
00131 *  $-\left( \frac{z_g - \Phi_0}{\Phi_A} \right)^2$ 
00132 *  $-\cos \left( \frac{k}{r^2} R(z) + g(z) - k \frac{z_g}{r} \right)$ 
00133 *  $-\cos \left( \frac{k}{r^2} R(z) + g(z) - k \frac{z_g}{r} \right)$ 
00134 *  $-\cos \left( \frac{k}{r^2} R(z) + g(z) - k \frac{z_g}{r} \right)$ 
00135 *  $-\cos \left( \frac{k}{r^2} R(z) + g(z) - k \frac{z_g}{r} \right)$ 
00136 *  $-\cos \left( \frac{k}{r^2} R(z) + g(z) - k \frac{z_g}{r} \right)$ 
00137 *  $-\cos \left( \frac{k}{r^2} R(z) + g(z) - k \frac{z_g}{r} \right)$ 
00138 *  $-\cos \left( \frac{k}{r^2} R(z) + g(z) - k \frac{z_g}{r} \right)$ 
00139 *  $-\cos \left( \frac{k}{r^2} R(z) + g(z) - k \frac{z_g}{r} \right)$ 
00140 *  $-\cos \left( \frac{k}{r^2} R(z) + g(z) - k \frac{z_g}{r} \right)$ 
00141 *  $-\cos \left( \frac{k}{r^2} R(z) + g(z) - k \frac{z_g}{r} \right)$ 
00142 *  $-\cos \left( \frac{k}{r^2} R(z) + g(z) - k \frac{z_g}{r} \right)$ 
00143 *  $-\cos \left( \frac{k}{r^2} R(z) + g(z) - k \frac{z_g}{r} \right)$ 
00144 *  $-\cos \left( \frac{k}{r^2} R(z) + g(z) - k \frac{z_g}{r} \right)$ 
00145 *  $-\cos \left( \frac{k}{r^2} R(z) + g(z) - k \frac{z_g}{r} \right)$ 
00146 *  $-\cos \left( \frac{k}{r^2} R(z) + g(z) - k \frac{z_g}{r} \right)$ 
00147 *  $-\cos \left( \frac{k}{r^2} R(z) + g(z) - k \frac{z_g}{r} \right)$ 
00148 *  $-\cos \left( \frac{k}{r^2} R(z) + g(z) - k \frac{z_g}{r} \right)$ 
00149 *  $-\cos \left( \frac{k}{r^2} R(z) + g(z) - k \frac{z_g}{r} \right)$ 
00150 *  $-\cos \left( \frac{k}{r^2} R(z) + g(z) - k \frac{z_g}{r} \right)$ 
00151 *  $-\cos \left( \frac{k}{r^2} R(z) + g(z) - k \frac{z_g}{r} \right)$ 
00152 *  $-\cos \left( \frac{k}{r^2} R(z) + g(z) - k \frac{z_g}{r} \right)$ 
00153 *  $-\cos \left( \frac{k}{r^2} R(z) + g(z) - k \frac{z_g}{r} \right)$ 
00154 *  $-\cos \left( \frac{k}{r^2} R(z) + g(z) - k \frac{z_g}{r} \right)$ 
00155 *  $-\cos \left( \frac{k}{r^2} R(z) + g(z) - k \frac{z_g}{r} \right)$ 
00156 *  $-\cos \left( \frac{k}{r^2} R(z) + g(z) - k \frac{z_g}{r} \right)$ 

```

```

00157     /// center of beam \f$ \Phi_0 \f$
00158     sunrealtype Ph0;
00159     /// length of beam \f$ \Phi_A \f$
00160     sunrealtype PhA;
00161     /// amplitude projection on TE-mode
00162     sunrealtype A1;
00163     /// amplitude projection on xy-plane
00164     sunrealtype A2;
00165     /// wavelength \f$ \lambda \f$
00166     sunrealtype lambda;
00167
00168 public:
00169     /// construction with default parameters
00170     Gauss2D(vector<sunrealtype> dis_ = {0, 0, 0},
00171            vector<sunrealtype> axis_ = {1, 0, 0}, sunrealtype Amp_ = 1.01,
00172            sunrealtype phip_ = 0, sunrealtype w0_ = 1e-5, sunrealtype zr_ = 4e-5,
00173            sunrealtype Ph0_ = 2e-5, sunrealtype PhA_ = 0.45e-5);
00174     /// function for the actual implementation in space
00175     void addToSpace(sunrealtype x, sunrealtype y, sunrealtype z,
00176                    sunrealtype *pTo6Space) const;
00177
00178 public:
00179 };
00180
00181 /** @brief class for Gaussian waves in 3D
00182  *
00183  * They are given in the form
00184  * 
$$\vec{E} = A \vec{\epsilon} \frac{\omega_0}{\omega(z)} \exp\left(-\frac{(z_g - \Phi_0)/\Phi_A^2}{2R(z)} + g(z) - k \frac{z_g}{R(z)}\right)$$

00185  * with
00186  * 
$$k = \frac{2\pi}{\lambda}$$

00187  * - propagation direction (subtracted distance to origin)  $z_g$ 
00188  * - radial distance to propagation axis  $r = \sqrt{x^2 + z_g^2}$ 
00189  * -  $k = 2\pi / \lambda$ 
00190  * - waist at position  $z$ ,  $\omega(z) = w_0 \sqrt{1 + (z_g/z_R)^2}$ 
00191  * - Gouy phase  $g(z) = \tan^{-1}(z_g/z_R)$ 
00192  * - beam curvature  $R(z) = z_g \sqrt{1 + (z_r/z_g)^2}$ 
00193  * obtained via the chosen parameters */
00194 class Gauss3D {
00195 private:
00196     /// distance maximum to origin
00197     vector<sunrealtype> dis;
00198     /// normalized propagation axis
00199     vector<sunrealtype> axis;
00200     /// amplitude \f$ A \f$
00201     sunrealtype Amp;
00202     /// polarization rotation from TE-mode around propagation direction
00203     /// that determines \f$ \vec{\epsilon} \f$ above
00204     sunrealtype phip;
00205     /// polarization
00206     vector<sunrealtype> pol;
00207     /// taille \f$ \omega_0 \f$
00208     sunrealtype w0;
00209     /// Rayleigh length \f$ z_R = \pi \omega_0^2 / \lambda \f$
00210     sunrealtype zr;
00211     /// center of beam \f$ \Phi_0 \f$
00212     sunrealtype Ph0;
00213     /// length of beam \f$ \Phi_A \f$
00214     sunrealtype PhA;
00215     /// amplitude projection on TE-mode (z-axis)
00216     sunrealtype A1;
00217     /// amplitude projection on xy-plane
00218     sunrealtype A2;
00219     /// wavelength \f$ \lambda \f$
00220     sunrealtype lambda;
00221
00222 public:
00223     /// construction with default parameters
00224     Gauss3D(vector<sunrealtype> dis_ = {0, 0, 0},
00225            vector<sunrealtype> axis_ = {1, 0, 0}, sunrealtype Amp_ = 1.01,
00226            sunrealtype phip_ = 0,
00227            sunrealtype pol_ = {0,0,1},
00228            sunrealtype w0_ = 1e-5, sunrealtype zr_ = 4e-5,
00229            sunrealtype Ph0_ = 2e-5, sunrealtype PhA_ = 0.45e-5);
00230     /// function for the actual implementation in space
00231     void addToSpace(sunrealtype x, sunrealtype y, sunrealtype z,
00232                    sunrealtype *pTo6Space) const;
00233
00234 public:
00235 };
00236
00237 /** @brief ICSetter class to initialize wave types with default parameters */
00238 class ICSetter {
00239 private:
00240     /// container vector for plane waves in 1D
00241     vector<PlaneWave1D> planeWaves1D;
00242     /// container vector for plane waves in 2D
00243     vector<PlaneWave2D> planeWaves2D;

```



```

00244  /// container vector for plane waves in 3D
00245  vector<PlaneWave3D> planeWaves3D;
00246  /// container vector for Gaussian waves in 1D
00247  vector<Gauss1D> gauss1Ds;
00248  /// container vector for Gaussian waves in 2D
00249  vector<Gauss2D> gauss2Ds;
00250  /// container vector for Gaussian waves in 3D
00251  vector<Gauss3D> gauss3Ds;
00252
00253 public:
00254  /// function to set all coordinates to zero and then 'add' the field values
00255  void eval(sunrealtype x, 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(sunrealtype x, unrealtype y, unrealtype z, unrealtype *pTo6Space);
00260  /// function to add plane waves in 1D to their container vector
00261  void addPlaneWave1D(vector<unrealtype> k = {1, 0, 0},
00262                    vector<unrealtype> p = {0, 0, 1},
00263                    vector<unrealtype> phi = {0, 0, 0});
00264  /// function to add plane waves in 2D to their container vector
00265  void addPlaneWave2D(vector<unrealtype> k = {1, 0, 0},
00266                    vector<unrealtype> p = {0, 0, 1},
00267                    vector<unrealtype> phi = {0, 0, 0});
00268  /// function to add plane waves in 3D to their container vector
00269  void addPlaneWave3D(vector<unrealtype> k = {1, 0, 0},
00270                    vector<unrealtype> p = {0, 0, 1},
00271                    vector<unrealtype> phi = {0, 0, 0});
00272  /// function to add Gaussian waves in 1D to their container vector
00273  void addGauss1D(vector<unrealtype> k = {1, 0, 0},
00274                vector<unrealtype> p = {0, 0, 1},
00275                vector<unrealtype> xo = {0, 0, 0}, unrealtype phig_ = 1.01,
00276                vector<unrealtype> phi = {0, 0, 0});
00277  /// function to add Gaussian waves in 2D to their container vector
00278  void addGauss2D(vector<unrealtype> dis_ = {0, 0, 0},
00279                vector<unrealtype> axis_ = {1, 0, 0},
00280                unrealtype Amp_ = 1.01, unrealtype phip_ = 0,
00281                unrealtype w0_ = 1e-5, unrealtype zr_ = 4e-5,
00282                unrealtype Ph0_ = 2e-5, unrealtype PhA_ = 0.45e-5);
00283  /// function to add Gaussian waves in 3D to their container vector
00284  void addGauss3D(vector<unrealtype> dis_ = {0, 0, 0},
00285                vector<unrealtype> axis_ = {1, 0, 0},
00286                unrealtype Amp_ = 1.01, unrealtype phip_ = 0,
00287                unrealtype w0_ = 1e-5, unrealtype zr_ = 4e-5,
00288                unrealtype Ph0_ = 2e-5, unrealtype PhA_ = 0.45e-5);
00289 };
00290
00291 // End of Includeguard
00292 #endif

```

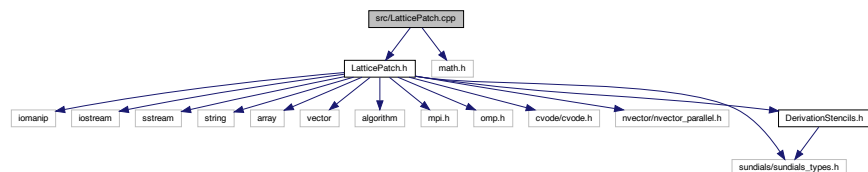
6.10 src/LatticePatch.cpp File Reference

Costruction of the overall envelope lattice and the lattice patches.

```
#include "LatticePatch.h"
```

```
#include <math.h>
```

Include dependency graph for LatticePatch.cpp:



Functions

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

Set up the patchwork.

- void [errorKill](#) (const string &errorMessage)

Print a specific error message to stdout.

- int [check_retval](#) (void *returnvalue, const char *funcname, int opt, int id)

6.10.1 Detailed Description

Costruction of the overall envelope lattice and the lattice patches.

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

6.10.2 Function Documentation

6.10.2.1 check_retval()

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

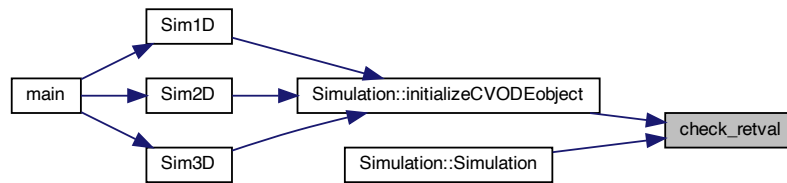
Check function return value. From CCode examples. opt == 0 means SUNDIALS function allocates memory so check if returned NULL pointer opt == 1 means SUNDIALS function returns an integer value so check if retval < 0 opt == 2 means function allocates memory so check if returned NULL pointer

Definition at line 841 of file [LatticePatch.cpp](#).

```
00841                                     {
00842     int *retval = nullptr;
00843
00844     /* Check if SUNDIALS function returned NULL pointer - no memory allocated */
00845     if (opt == 0 && returnvalue == nullptr) {
00846         fprintf(stderr,
00847             "\nSUNDIALS_ERROR(%d): %s() failed - returned NULL pointer\n\n", id,
00848             funcname);
00849         return (1);
00850     }
00851
00852     /* Check if retval < 0 */
00853     else if (opt == 1) {
00854         retval = (int *)returnvalue;
00855         if (*retval < 0) {
00856             fprintf(stderr, "\nSUNDIALS_ERROR(%d): %s() failed with retval = %d\n\n",
00857                 id, funcname, *retval);
00858             return (1);
00859         }
00860     }
00861
00862     /* Check if function returned NULL pointer - no memory allocated */
00863     else if (opt == 2 && returnvalue == nullptr) {
00864         fprintf(stderr,
00865             "\nMEMORY_ERROR(%d): %s() failed - returned NULL pointer\n\n", id,
00866             funcname);
00867         return (1);
00868     }
00869
00870     return (0);
00871 }
```

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

Here is the caller graph for this function:



6.10.2.2 errorKill()

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

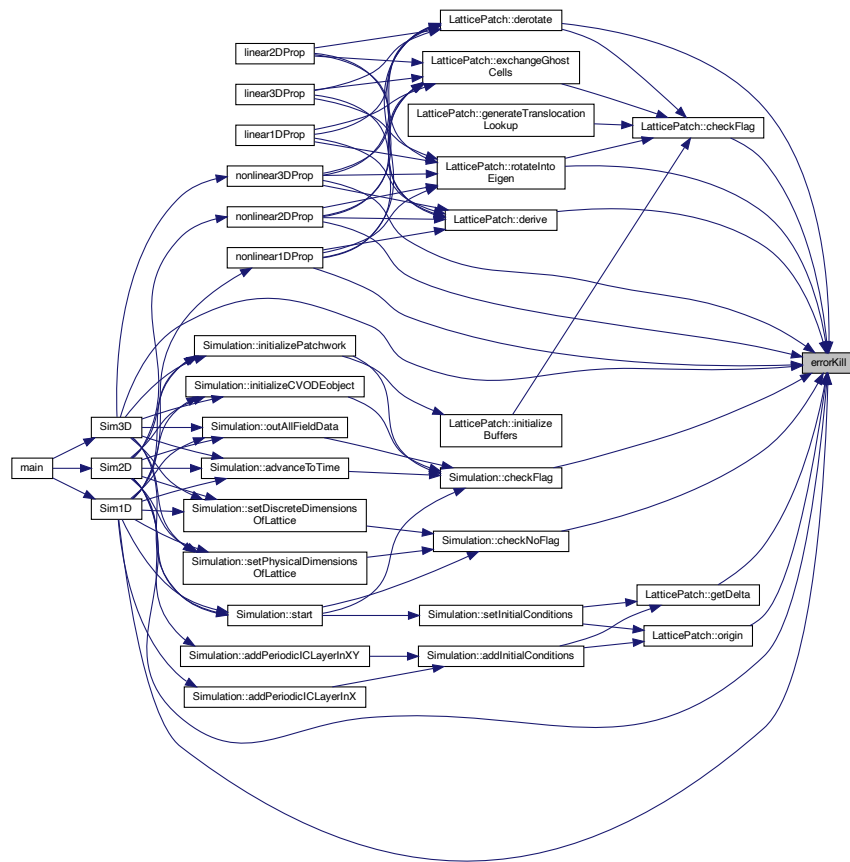
Print a specific error message to stdout.

Definition at line 828 of file [LatticePatch.cpp](#).

```
00828 {
00829     cerr << endl << "Error: " << errorMessage << " Aborting..." << endl;
00830     MPI_Abort (MPI_COMM_WORLD, 1);
00831     return;
00832 }
```

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

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

```
00110 {
00111     // Retrieve the ghost layer depth
00112     const int gLW = envelopeLattice.get_ghostLayerWidth();
00113     // Retrieve the data point dimension
00114     const int dPD = envelopeLattice.get_dataPointDimension();
00115     // MPI process/patch
00116     const int my_prc = envelopeLattice.my_prc;
00117     // Determine thickness of the slice
00118     const sunindextype tot_NOXP = envelopeLattice.get_tot_nx(); // total points of lattice
00119     const sunindextype tot_NOYP = envelopeLattice.get_tot_ny();
```

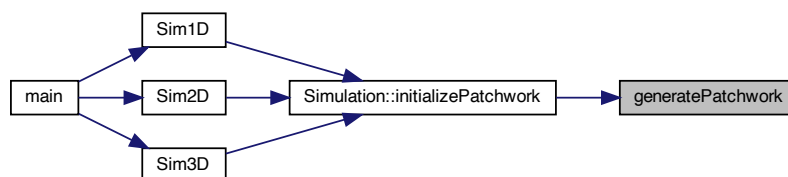
```

00120  const sunindextype tot_NOZP = envelopeLattice.get_tot_nz();
00121  // position of the patch in the lattice of patches - process associated to
00122  // position
00123  const sunindextype LIx = my_prc % DLx;
00124  const sunindextype LIy = (my_prc / DLx) % DLy;
00125  const sunindextype LIz = (my_prc / DLx) / DLy;
00126  // Determine the number of points in the patch and first absolute points in
00127  // each dimension
00128  const sunindextype local_NOXP = tot_NOXP / DLx;
00129  const sunindextype local_NOYP = tot_NOYP / DLy;
00130  const sunindextype local_NOZP = tot_NOZP / DLz;
00131  // absolute positions of the first point in each dimension
00132  const sunindextype firstXPoint = local_NOXP * LIx;
00133  const sunindextype firstYPoint = local_NOYP * LIy;
00134  const sunindextype firstZPoint = local_NOZP * LIz;
00135  // total number of points in the patch
00136  const sunindextype local_NODP = dPD * local_NOXP * local_NOYP * local_NOZP;
00137
00138  // Set patch up with above derived quantities
00139  patchToMold.dx = envelopeLattice.get_dx();
00140  patchToMold.dy = envelopeLattice.get_dy();
00141  patchToMold.dz = envelopeLattice.get_dz();
00142  patchToMold.x0 = firstXPoint * patchToMold.dx;
00143  patchToMold.y0 = firstYPoint * patchToMold.dy;
00144  patchToMold.z0 = firstZPoint * patchToMold.dz;
00145  patchToMold.LIx = LIx;
00146  patchToMold.LIy = LIy;
00147  patchToMold.LIz = LIz;
00148  patchToMold.nx = local_NOXP;
00149  patchToMold.ny = local_NOYP;
00150  patchToMold.nz = local_NOZP;
00151  patchToMold.lx = patchToMold.nx * patchToMold.dx;
00152  patchToMold.ly = patchToMold.ny * patchToMold.dy;
00153  patchToMold.lz = patchToMold.nz * patchToMold.dz;
00154  /* Create and allocate memory for parallel vectors with defined local and
00155   * global lenghts *
00156   * (-> CNode problem sizes Nlocal and N)
00157   * for field data and temporal derivatives and set extra pointers to them */
00158  patchToMold.u =
00159      N_VNew_Parallel(envelopeLattice.comm, local_NODP,
00160                     envelopeLattice.get_tot_nodp(), envelopeLattice.sunctx);
00161  patchToMold.uData = NV_DATA_P(patchToMold.u);
00162  patchToMold.du =
00163      N_VNew_Parallel(envelopeLattice.comm, local_NODP,
00164                     envelopeLattice.get_tot_nodp(), envelopeLattice.sunctx);
00165  patchToMold.duData = NV_DATA_P(patchToMold.du);
00166  // Allocate space for auxiliary uAux so that the lattice and all possible
00167  // directions of ghost layers fit
00168  const int s1 = patchToMold.nx, s2 = patchToMold.ny, s3 = patchToMold.nz;
00169  const int s_min = min(s1, min(s2, s3));
00170  patchToMold.uAux.resize(s1 * s2 * s3 / s_min * (s_min + 2 * gLW) * dPD);
00171  patchToMold.uAuxData = &patchToMold.uAux[0];
00172  patchToMold.envelopeLattice = &envelopeLattice;
00173  // Set patch "name" to process number -> only for debugging
00174  // patchToMold.ID=my_prc;
00175  // set flag
00176  patchToMold.statusFlags = FLatticePatchSetUp;
00177  patchToMold.generateTranslocationLookup();
00178  return 0;
00179 }

```

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 Costruction of the overall envelope lattice and the lattice patches
00004 //////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
00005
00006 #include "LatticePatch.h"
00007
00008 #include <math.h>
00009
00010 //////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
00011 /// Implementation of Lattice component functions ///
00012 //////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
00013
00014 /// Initialize the cartesian communicator
00015 void Lattice::initializeCommunicator(const int nx, const int ny,
00016     const int nz, const bool per) {
00017     const int dims[3] = {nz, ny, nx};
00018     const int periods[3] = {static_cast<int>(per), static_cast<int>(per),
00019         static_cast<int>(per)};
00020     // Create the cartesian communicator for MPI_COMM_WORLD
00021     MPI_Cart_create(MPI_COMM_WORLD, 3, dims, periods, 1, &comm);
00022     // Set MPI variables of the lattice
00023     MPI_Comm_size(comm, &(n_prc));
00024     MPI_Comm_rank(comm, &(my_prc));
00025     // Associate name to the communicator to identify it -> for debugging and
00026     // nicer error messages
00027     constexpr char lattice_comm_name[] = "Lattice";
00028     MPI_Comm_set_name(comm, lattice_comm_name);
00029
00030     // Test if process naming is the same for both communicators
00031     /*
00032     int MYPRC;
00033     MPI_Comm_rank(MPI_COMM_WORLD, &MYPRC);
00034     cout<<"\r"<<my_prc<<"\t"<<MYPRC<<endl;
00035     */
00036 }
00037
00038 /// Construct the lattice and set the stencil order
00039 Lattice::Lattice(const int StO) : stencilOrder(StO),
00040     ghostLayerWidth(StO/2+1) {
00041     statusFlags = 0;
00042 }
00043
00044 /// Set the number of points in each dimension of the lattice
00045 void Lattice::setDiscreteDimensions(const sunindextype _nx,
00046     const sunindextype _ny, const sunindextype _nz) {
00047     // copy the given data for number of points
00048     tot_nx = _nx;
00049     tot_ny = _ny;
00050     tot_nz = _nz;
00051     // compute the resulting number of points and datapoints
00052     tot_noP = tot_nx * tot_ny * tot_nz;
00053     tot_noDP = dataPointDimension * tot_noP;
00054     // compute the new Delta, the physical resolution
00055     dx = tot_lx / tot_nx;
00056     dy = tot_ly / tot_ny;
00057     dz = tot_lz / tot_nz;
00058 }
00059
00060 /// Set the physical size of the lattice
00061 void Lattice::setPhysicalDimensions(const sunrealtype _lx,
00062     const sunrealtype _ly, const sunrealtype _lz) {
00063     tot_lx = _lx;
00064     tot_ly = _ly;
00065     tot_lz = _lz;
00066     // calculate physical distance between points
00067     dx = tot_lx / tot_nx;
00068     dy = tot_ly / tot_ny;
00069     dz = tot_lz / tot_nz;
00070     statusFlags |= FLatticeDimensionSet;
00071 }
00072
00073 //////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
00074 /// Implementation of LatticePatch component functions ///
00075 //////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
00076
00077 /// Construct the lattice patch
00078 LatticePatch::LatticePatch() {
00079     // set default origin coordinates to (0,0,0)
00080     x0 = y0 = z0 = 0;
00081     // set default position in Lattice-Patchwork to (0,0,0)
00082     Llx = Lly = Llz = 0;

```

```

00083 // set default physical length for lattice patch to (0,0,0)
00084 lx = ly = lz = 0;
00085 // set default discrete length for lattice patch to (0,1,1)
00086 /* This is done in this manner as even in 1D simulations require a 1 point
00087 * width */
00088 nx = 0;
00089 ny = nz = 1;
00090
00091 // u is not initialized as it wouldn't make any sense before the dimensions
00092 // are set idem for the enveloping lattice
00093
00094 // set default statusFlags to non set
00095 statusFlags = 0;
00096 }
00097
00098 /// Destruct the patch and thereby destroy the NVectors
00099 LatticePatch::~LatticePatch() {
00100 // Deallocate memory for solution vector
00101 if (statusFlags & FLatticePatchSetUp) {
00102 // Destroy data vectors
00103 N_VDestroy_Parallel(u);
00104 N_VDestroy_Parallel(du);
00105 }
00106 }
00107
00108 /// Set up the patchwork
00109 int generatePatchwork(const Lattice &envelopeLattice, LatticePatch &patchToMold,
00110 const int DLx, const int DLy, const int DLz) {
00111 // Retrieve the ghost layer depth
00112 const int gLW = envelopeLattice.get_ghostLayerWidth();
00113 // Retrieve the data point dimension
00114 const int dPD = envelopeLattice.get_dataPointDimension();
00115 // MPI process/patch
00116 const int my_prc = envelopeLattice.my_prc;
00117 // Determine thickness of the slice
00118 const sunindextype tot_NOXP = envelopeLattice.get_tot_nx(); // total points of lattice
00119 const sunindextype tot_NOYP = envelopeLattice.get_tot_ny();
00120 const sunindextype tot_NOZP = envelopeLattice.get_tot_nz();
00121 // position of the patch in the lattice of patches - process associated to
00122 // position
00123 const sunindextype LIx = my_prc % DLx;
00124 const sunindextype LIy = (my_prc / DLx) % DLy;
00125 const sunindextype LIz = (my_prc / DLx) / DLy;
00126 // Determine the number of points in the patch and first absolute points in
00127 // each dimension
00128 const sunindextype local_NOXP = tot_NOXP / DLx;
00129 const sunindextype local_NOYP = tot_NOYP / DLy;
00130 const sunindextype local_NOZP = tot_NOZP / DLz;
00131 // absolute positions of the first point in each dimension
00132 const sunindextype firstXPoint = local_NOXP * LIx;
00133 const sunindextype firstYPoint = local_NOYP * LIy;
00134 const sunindextype firstZPoint = local_NOZP * LIz;
00135 // total number of points in the patch
00136 const sunindextype local_NODP = dPD * local_NOXP * local_NOYP * local_NOZP;
00137
00138 // Set patch up with above derived quantities
00139 patchToMold.dx = envelopeLattice.get_dx();
00140 patchToMold.dy = envelopeLattice.get_dy();
00141 patchToMold.dz = envelopeLattice.get_dz();
00142 patchToMold.x0 = firstXPoint * patchToMold.dx;
00143 patchToMold.y0 = firstYPoint * patchToMold.dy;
00144 patchToMold.z0 = firstZPoint * patchToMold.dz;
00145 patchToMold.LIx = LIx;
00146 patchToMold.LIy = LIy;
00147 patchToMold.LIz = LIz;
00148 patchToMold.nx = local_NOXP;
00149 patchToMold.ny = local_NOYP;
00150 patchToMold.nz = local_NOZP;
00151 patchToMold.lx = patchToMold.nx * patchToMold.dx;
00152 patchToMold.ly = patchToMold.ny * patchToMold.dy;
00153 patchToMold.lz = patchToMold.nz * patchToMold.dz;
00154 /* Create and allocate memory for parallel vectors with defined local and
00155 * global lengths *
00156 * (-> CCode problem sizes Nlocal and N)
00157 * for field data and temporal derivatives and set extra pointers to them */
00158 patchToMold.u =
00159 N_VNew_Parallel(envelopeLattice.comm, local_NODP,
00160 envelopeLattice.get_tot_nodp(), envelopeLattice.sunctx);
00161 patchToMold.uData = NV_DATA_P(patchToMold.u);
00162 patchToMold.du =
00163 N_VNew_Parallel(envelopeLattice.comm, local_NODP,
00164 envelopeLattice.get_tot_nodp(), envelopeLattice.sunctx);
00165 patchToMold.duData = NV_DATA_P(patchToMold.du);
00166 // Allocate space for auxiliary uAux so that the lattice and all possible
00167 // directions of ghost layers fit
00168 const int s1 = patchToMold.nx, s2 = patchToMold.ny, s3 = patchToMold.nz;
00169 const int s_min = min(s1, min(s2, s3));

```

```

00170 patchToMold.uAux.resize(s1 * s2 * s3 / s_min * (s_min + 2 * gLW) * dPD);
00171 patchToMold.uAuxData = &patchToMold.uAux[0];
00172 patchToMold.envelopeLattice = &envelopeLattice;
00173 // Set patch "name" to process number -> only for debugging
00174 // patchToMold.ID=my_prc;
00175 // set flag
00176 patchToMold.statusFlags = FLatticePatchSetUp;
00177 patchToMold.generateTranslocationLookup();
00178 return 0;
00179 }
00180
00181 /// Return the discrete size of the patch: number of lattice patch points in
00182 /// specified dimension
00183 int LatticePatch::discreteSize(int dir) const {
00184     switch (dir) {
00185         case 0:
00186             return nx * ny * nz;
00187         case 1:
00188             return nx;
00189         case 2:
00190             return ny;
00191         case 3:
00192             return nz;
00193         // case 4: return uAux.size(); // for debugging
00194         default:
00195             return -1;
00196     }
00197 }
00198
00199 /// Return the physical origin of the patch in a dimension
00200 sunrealtype LatticePatch::origin(const int dir) const {
00201     switch (dir) {
00202         case 1:
00203             return x0;
00204         case 2:
00205             return y0;
00206         case 3:
00207             return z0;
00208         default:
00209             errorKill("LatticePatch::origin function called with wrong dir parameter");
00210             return -1;
00211     }
00212 }
00213
00214 /// Return the distance between points in the patch in a dimension
00215 sunrealtype LatticePatch::getDelta(const int dir) const {
00216     switch (dir) {
00217         case 1:
00218             return dx;
00219         case 2:
00220             return dy;
00221         case 3:
00222             return dz;
00223         default:
00224             errorKill(
00225                 "LatticePatch::getDelta function called with wrong dir parameter");
00226             return -1;
00227     }
00228 }
00229
00230 /** To avoid cache misses:
00231  * create vectors to translate u vector into space coordinates and vice versa
00232  * and same for left and right ghost layers to space */
00233 void LatticePatch::generateTranslocationLookup() {
00234     // Check that the lattice has been set up
00235     checkFlag(FLatticeDimensionSet);
00236     // lengths for auxilliary layers, including ghost layers
00237     const int gLW = envelopeLattice->get_ghostLayerWidth();
00238     const int mx = nx + 2 * gLW;
00239     const int my = ny + 2 * gLW;
00240     const int mz = nz + 2 * gLW;
00241     // sizes for lookup vectors
00242     // generate u->uAux
00243     uTox.resize(nx * ny * nz);
00244     uToy.resize(nx * ny * nz);
00245     uToz.resize(nx * ny * nz);
00246     // generate uAux->u with length including halo
00247     xTou.resize(mx * ny * nz);
00248     yTou.resize(nx * my * nz);
00249     zTou.resize(nx * ny * mz);
00250     // variables for cartesian position in the 3D discrete lattice
00251     int px = 0, py = 0, pz = 0;
00252     for (int i = 0; i < uToy.size(); i++) { // loop over all points in the patch
00253         // calculate cartesian coordinates
00254         px = i % nx;
00255         py = (i / nx) % ny;
00256         pz = (i / nx) / ny;

```



```

00257 // fill lookups extended by halos (useful for y and z direction)
00258 uTox[i] = (px + gLW) + py * mx +
00259           pz * mx * ny; // unroll (de-flatten) cartesian dimension
00260 xTou[px + py * mx + pz * mx * ny] =
00261     i; // match cartesian point to u location
00262 uToy[i] = (py + gLW) + pz * my + px * my * nz;
00263 yTou[py + pz * my + px * my * nz] = i;
00264 uToz[i] = (pz + gLW) + px * mz + py * mz * nx;
00265 zTou[pz + px * mz + py * mz * nx] = i;
00266 }
00267 // same for ghost layer lookup tables
00268 lgcTox.resize(gLW * ny * nz);
00269 rgcTox.resize(gLW * ny * nz);
00270 for (int i = 0; i < lgcTox.size(); i++) {
00271     px = i % gLW;
00272     py = (i / gLW) % ny;
00273     pz = (i / gLW) / ny;
00274     lgcTox[i] = px + py * mx + pz * mx * ny;
00275     rgcTox[i] = px + nx + gLW + py * mx + pz * mx * ny;
00276 }
00277 lgcToy.resize(gLW * nx * nz);
00278 rgcToy.resize(gLW * nx * nz);
00279 for (int i = 0; i < lgcToy.size(); i++) {
00280     px = i % nx;
00281     py = (i / nx) % gLW;
00282     pz = (i / nx) / gLW;
00283     lgcToy[i] = py + pz * my + px * my * nz;
00284     rgcToy[i] = py + ny + gLW + pz * my + px * my * nz;
00285 }
00286 lgcToz.resize(gLW * nx * ny);
00287 rgcToz.resize(gLW * nx * ny);
00288 for (int i = 0; i < lgcToz.size(); i++) {
00289     px = i % nx;
00290     py = (i / nx) % ny;
00291     pz = (i / nx) / ny;
00292     lgcToz[i] = pz + px * mz + py * mz * nx;
00293     rgcToz[i] = pz + nz + gLW + px * mz + py * mz * nx;
00294 }
00295 statusFlags |= TranslocationLookupSetUp;
00296 }
00297
00298 /** Rotate into eigenraum along R matrices of paper using below rotation
00299 * functions
00300 * -> uAuxData gets the rotated left-halo-, inner-patch-, right-halo-data */
00301 void LatticePatch::rotateIntoEigen(const int dir) {
00302     // Check that the lattice, ghost layers as well as the translocation lookups
00303     // have been set up;
00304     checkFlag(FLatticePatchSetUp);
00305     checkFlag(TranslocationLookupSetUp);
00306     checkFlag(GhostLayersInitialized); // this check is only after call to
00307                                         // exchange ghost cells
00308     switch (dir) {
00309     case 1:
00310         rotateToX(uAuxData, gCLData, lgcTox);
00311         rotateToX(uAuxData, uData, uTox);
00312         rotateToX(uAuxData, gCRData, rgcTox);
00313         break;
00314     case 2:
00315         rotateToY(uAuxData, gCLData, lgcToy);
00316         rotateToY(uAuxData, uData, uToy);
00317         rotateToY(uAuxData, gCRData, rgcToy);
00318         break;
00319     case 3:
00320         rotateToZ(uAuxData, gCLData, lgcToz);
00321         rotateToZ(uAuxData, uData, uToz);
00322         rotateToZ(uAuxData, gCRData, rgcToz);
00323         break;
00324     default:
00325         errorKill("Tried to rotate into the wrong direction");
00326         break;
00327     }
00328 }
00329
00330 /// Rotate halo and inner-patch data vectors with rotation matrix Rx into
00331 /// eigenspace of Z matrix and write to auxiliary vector
00332 inline void LatticePatch::rotateToX(sunrealtype *outArray,
00333                                     const unrealtype *inArray,
00334                                     const vector<int> &lookup) {
00335     int ii = 0, target = 0;
00336     #pragma ivdep
00337     #pragma omp simd // safelen(6)
00338     for (int i = 0; i < lookup.size(); i++) {
00339         // get correct u-vector and spatial indices along previously defined lookup
00340         // tables
00341         target = envelopeLattice->get_dataPointDimension() * lookup[i];
00342         ii = envelopeLattice->get_dataPointDimension() * i;
00343         outArray[target + 0] = -inArray[1 + ii] + inArray[5 + ii];

```

```

00344     outArray[target + 1] = inArray[2 + ii] + inArray[4 + ii];
00345     outArray[target + 2] = inArray[1 + ii] + inArray[5 + ii];
00346     outArray[target + 3] = -inArray[2 + ii] + inArray[4 + ii];
00347     outArray[target + 4] = inArray[3 + ii];
00348     outArray[target + 5] = inArray[ii];
00349 }
00350 }
00351
00352 /// Rotate halo and inner-patch data vectors with rotation matrix Ry into
00353 /// eigenspace of Z matrix and write to auxiliary vector
00354 inline void LatticePatch::rotateToY(sunrealtype *outArray,
00355                                     const unrealtype *inArray,
00356                                     const vector<int> &lookup) {
00357     int ii = 0, target = 0;
00358     #pragma ivdep
00359     #pragma omp simd
00360     for (int i = 0; i < lookup.size(); i++) {
00361         target = envelopeLattice->get_dataPointDimension() * lookup[i];
00362         ii = envelopeLattice->get_dataPointDimension() * i;
00363         outArray[target + 0] = inArray[ii] + inArray[5 + ii];
00364         outArray[target + 1] = -inArray[2 + ii] + inArray[3 + ii];
00365         outArray[target + 2] = -inArray[ii] + inArray[5 + ii];
00366         outArray[target + 3] = inArray[2 + ii] + inArray[3 + ii];
00367         outArray[target + 4] = inArray[4 + ii];
00368         outArray[target + 5] = inArray[1 + ii];
00369     }
00370 }
00371
00372 /// Rotate halo and inner-patch data vectors with rotation matrix Rz into
00373 /// eigenspace of Z matrix and write to auxiliary vector
00374 inline void LatticePatch::rotateToZ(sunrealtype *outArray,
00375                                     const unrealtype *inArray,
00376                                     const vector<int> &lookup) {
00377     int ii = 0, target = 0;
00378     #pragma ivdep
00379     #pragma omp simd
00380     for (int i = 0; i < lookup.size(); i++) {
00381         target = envelopeLattice->get_dataPointDimension() * lookup[i];
00382         ii = envelopeLattice->get_dataPointDimension() * i;
00383         outArray[target + 0] = -inArray[ii] + inArray[4 + ii];
00384         outArray[target + 1] = inArray[1 + ii] + inArray[3 + ii];
00385         outArray[target + 2] = inArray[ii] + inArray[4 + ii];
00386         outArray[target + 3] = -inArray[1 + ii] + inArray[3 + ii];
00387         outArray[target + 4] = inArray[5 + ii];
00388         outArray[target + 5] = inArray[2 + ii];
00389     }
00390 }
00391
00392 /// Derotate uAux with transposed rotation matrices and write to derivative
00393 /// buffer - normalization is done here by the factor 1/2
00394 void LatticePatch::derotate(int dir, unrealtype *buffOut) {
00395     // Check that the lattice as well as the translocation lookups have been set
00396     // up;
00397     checkFlag(FLatticePatchSetUp);
00398     checkFlag(TranslocationLookupSetUp);
00399     const int dPD = envelopeLattice->get_dataPointDimension();
00400     const int gLW = envelopeLattice->get_ghostLayerWidth();
00401     const int uSize = discreteSize();
00402     int ii = 0, target = 0;
00403     switch (dir) {
00404     case 1:
00405         #pragma ivdep
00406         #pragma omp simd
00407         for (int i = 0; i < uSize; i++) {
00408             // get correct indices in u and rotation space
00409             target = dPD * i;
00410             ii = dPD * (uToX[i] - gLW);
00411             buffOut[target + 0] = uAux[5 + ii];
00412             buffOut[target + 1] = (-uAux[ii] + uAux[2 + ii]) / 2.;
00413             buffOut[target + 2] = (uAux[1 + ii] - uAux[3 + ii]) / 2.;
00414             buffOut[target + 3] = uAux[4 + ii];
00415             buffOut[target + 4] = (uAux[1 + ii] + uAux[3 + ii]) / 2.;
00416             buffOut[target + 5] = (uAux[ii] + uAux[2 + ii]) / 2.;
00417         }
00418         break;
00419     case 2:
00420         #pragma omp simd
00421         for (int i = 0; i < uSize; i++) {
00422             target = dPD * i;
00423             ii = dPD * (uToY[i] - gLW);
00424             buffOut[target + 0] = (uAux[ii] - uAux[2 + ii]) / 2.;
00425             buffOut[target + 1] = uAux[5 + ii];
00426             buffOut[target + 2] = (-uAux[1 + ii] + uAux[3 + ii]) / 2.;
00427             buffOut[target + 3] = (uAux[1 + ii] + uAux[3 + ii]) / 2.;
00428             buffOut[target + 4] = uAux[4 + ii];
00429             buffOut[target + 5] = (uAux[ii] + uAux[2 + ii]) / 2.;
00430         }

```

```

00431     break;
00432     case 3:
00433 #pragma omp simd
00434     for (int i = 0; i < uSize; i++) {
00435         target = dPD * i;
00436         ii = dPD * (uToz[i] - gLW);
00437         buffOut[target + 0] = (-uAux[ii] + uAux[2 + ii]) / 2.;
00438         buffOut[target + 1] = (uAux[1 + ii] - uAux[3 + ii]) / 2.;
00439         buffOut[target + 2] = uAux[5 + ii];
00440         buffOut[target + 3] = (uAux[1 + ii] + uAux[3 + ii]) / 2.;
00441         buffOut[target + 4] = (uAux[ii] + uAux[2 + ii]) / 2.;
00442         buffOut[target + 5] = uAux[4 + ii];
00443     }
00444     break;
00445     default:
00446         errorKill("Tried to derotate from the wrong direction");
00447         break;
00448 }
00449 }
00450
00451 /// Create buffers to save derivative values, optimizing computational load
00452 void LatticePatch::initializeBuffers() {
00453     // Check that the lattice has been set up
00454     checkFlag(FLatticeDimensionSet);
00455     const int dPD = envelopeLattice->get_dataPointDimension();
00456     buffX.resize(nx * ny * nz * dPD);
00457     buffY.resize(nx * ny * nz * dPD);
00458     buffZ.resize(nx * ny * nz * dPD);
00459     // Set pointers used for propagation functions
00460     buffData[0] = &buffX[0];
00461     buffData[1] = &buffY[0];
00462     buffData[2] = &buffZ[0];
00463     statusFlags |= BuffersInitialized;
00464 }
00465
00466 /// Perform the ghost cell exchange in a specified direction
00467 void LatticePatch::exchangeGhostCells(const int dir) {
00468     // Check that the lattice has been set up
00469     checkFlag(FLatticeDimensionSet);
00470     checkFlag(FLatticePatchSetUp);
00471     // Variables to per dimension calculate the halo indices, and distance to
00472     // other side halo boundary
00473     int mx = 1, my = 1, mz = 1, distToRight = 1;
00474     const int gLW = envelopeLattice->get_ghostLayerWidth();
00475     // In the chosen direction m is set to ghost layer width while the others
00476     // remain to form the plane
00477     switch (dir) {
00478     case 1:
00479         mx = gLW;
00480         my = ny;
00481         mz = nz;
00482         distToRight = (nx - gLW);
00483         break;
00484     case 2:
00485         mx = nx;
00486         my = gLW;
00487         mz = nz;
00488         distToRight = nx * (ny - gLW);
00489         break;
00490     case 3:
00491         mx = nx;
00492         my = ny;
00493         mz = gLW;
00494         distToRight = nx * ny * (nz - gLW);
00495         break;
00496     }
00497     // total number of exchanged points
00498     const int dPD = envelopeLattice->get_dataPointDimension();
00499     const int exchangeSize = mx * my * mz * dPD;
00500     // provide size of the halos for ghost cells
00501     ghostCellLeft.resize(exchangeSize);
00502     ghostCellRight.resize(ghostCellLeft.size());
00503     ghostCellLeftToSend.resize(ghostCellLeft.size());
00504     ghostCellRightToSend.resize(ghostCellLeft.size());
00505     gCLData = &ghostCellLeft[0];
00506     gCRData = &ghostCellRight[0];
00507     statusFlags |= GhostLayersInitialized;
00508
00509     // Initialize running index li for the halo buffers, and index ui of uData for
00510     // data transfer
00511     int li = 0, ui = 0;
00512
00513     for (int iz = 0; iz < mz; iz++) {
00514         for (int iy = 0; iy < my; iy++) {
00515             // uData vector start index of halo data to be transferred
00516             // with each z-step add the whole xy-plane and with y-step the x-range ->
00517             // iterate all x-ranges

```

```

00518     ui = (iz * nx * ny + iy * nx) * dPD;
00519     // copy left halo data from uData to buffer, transfer size is given by
00520     // x-length (not x-range) perhaps faster but more fragile C lib copy
00521     // operation (contained in cstring header)
00522     /*
00523     memcpy(&ghostCellLeftToSend[li],
00524            &uData[ui],
00525            sizeof(sunrealtype)*mx*dPD);
00526     // increase ui by the distance to vis-a-vis boundary and copy right halo
00527     data to buffer ui+=distToRight*dPD; memcpy(&ghostCellRightToSend[li],
00528            &uData[ui],
00529            sizeof(sunrealtype)*mx*dPD);
00530     */
00531     // perhaps more safe but slower copy operation (contained in algorithm
00532     // header) performance highly system dependent
00533     copy(&uData[ui], &uData[ui + mx * dPD], &ghostCellLeftToSend[li]);
00534     ui += distToRight * dPD;
00535     copy(&uData[ui], &uData[ui + mx * dPD], &ghostCellRightToSend[li]);
00536
00537     // increase halo index by transferred items per y-iteration step
00538     // (x-length)
00539     li += mx * dPD;
00540 }
00541 }
00542
00543 /* Send and receive the data to and from neighboring latticePatches */
00544 // Adjust direction to cartesian communicator
00545 int dim = 2; // default for dir==1
00546 if (dir == 2) {
00547     dim = 1;
00548 } else if (dir == 3) {
00549     dim = 0;
00550 }
00551 MPI_Request requests[2];
00552 int rank_source = 0, rank_dest = 0;
00553 MPI_Cart_shift(envelopeLattice->comm, dim, -1, &rank_source,
00554               &rank_dest); // s.t. rank_dest is left & v.v.
00555
00556 // nonblocking Isend/Irecv
00557 /*
00558 MPI_Isend(&ghostCellLeftToSend[0], exchangeSize, MPI_SUNREALTYPE, rank_dest,
00559 1, envelopeLattice->comm, &requests[0]); MPI_Irecv(&ghostCellRight[0],
00560 exchangeSize, MPI_SUNREALTYPE, rank_source, 1, envelopeLattice->comm,
00561 &requests[0]); MPI_Isend(&ghostCellRightToSend[0], exchangeSize,
00562 MPI_SUNREALTYPE, rank_source, 2, envelopeLattice->comm, &requests[1]);
00563 MPI_Irecv(&ghostCellLeft[0], exchangeSize, MPI_SUNREALTYPE, rank_dest, 2,
00564 envelopeLattice->comm, &requests[1]);
00565
00566 MPI_Waitall(2, requests, MPI_STATUS_IGNORE);
00567 */
00568
00569 // blocking Sendrecv:
00570
00571 MPI_Sendrecv(&ghostCellLeftToSend[0], exchangeSize, MPI_SUNREALTYPE,
00572             rank_dest, 1, &ghostCellRight[0], exchangeSize, MPI_SUNREALTYPE,
00573             rank_source, 1, envelopeLattice->comm, MPI_STATUS_IGNORE);
00574 MPI_Sendrecv(&ghostCellRightToSend[0], exchangeSize, MPI_SUNREALTYPE,
00575             rank_source, 2, &ghostCellLeft[0], exchangeSize, MPI_SUNREALTYPE,
00576             rank_dest, 2, envelopeLattice->comm, MPI_STATUS_IGNORE);
00577 }
00578
00579 /// Check if all flags are set
00580 void LatticePatch::checkFlag(unsigned int flag) const {
00581     if (!(statusFlags & flag)) {
00582         string errorMessage;
00583         switch (flag) {
00584             case FLatticePatchSetUp:
00585                 errorMessage = "The Lattice patch was not set up please make sure to "
00586                               "initilize a Lattice topology";
00587                 break;
00588             case TranslocationLookupSetUp:
00589                 errorMessage = "The translocation lookup tables have not been generated, "
00590                               "please be sure to run generateTranslocationLookup()";
00591                 break;
00592             case GhostLayersInitialized:
00593                 errorMessage = "The space for the ghost layers has not been allocated, "
00594                               "please be sure to run initializeGhostLayer()";
00595                 break;
00596             case BuffersInitialized:
00597                 errorMessage = "The space for the buffers has not been allocated, please "
00598                               "be sure to run initializeBuffers()";
00599                 break;
00600             default:
00601                 errorMessage = "Uppss, you've made a non-standard error, sadly I can't "
00602                               "help you there";
00603                 break;
00604         }
00605     }

```

```

00605     errorKill(errorMessage);
00606 }
00607 return;
00608 }
00609
00610 /// Calculate derivatives in the patch (uAux) in the specified direction
00611 void LatticePatch::derive(const int dir) {
00612     // ghost layer width
00613     const int gLW = envelopeLattice->get_ghostLayerWidth();
00614     // dimensionality of data points -> 6
00615     const int dPD = envelopeLattice->get_dataPointDimension();
00616     // total width of patch in given direction including ghost layers at ends
00617     const int dirWidth = discreteSize(dir) + 2 * gLW;
00618     // width of patch only in given direction
00619     const int dirWidthO = discreteSize(dir);
00620     // size of plane perpendicular to given dimension
00621     const int perpPlainSize = discreteSize() / discreteSize(dir);
00622     // physical distance between points in that direction
00623     sunrealtype dxi = NAN;
00624     switch (dir) {
00625     case 1:
00626         dxi = dx;
00627         break;
00628     case 2:
00629         dxi = dy;
00630         break;
00631     case 3:
00632         dxi = dz;
00633         break;
00634     default:
00635         dxi = 1;
00636         errorKill("Tried to derive in the wrong direction");
00637         break;
00638     }
00639     // Derive according to chosen stencil accuracy order (which determines also
00640     // gLW)
00641     const int order = envelopeLattice->get_stencilOrder();
00642     switch (order) {
00643     case 1:
00644         for (int i = 0; i < perpPlainSize; i++) {
00645             for (int j = (i * dirWidth + gLW) * dPD;
00646                  j < (i * dirWidth + gLW + dirWidthO) * dPD; j += dPD) {
00647                 uAux[j + 0 - gLW * dPD] = s1b(&uAux[j + 0]) / dxi;
00648                 uAux[j + 1 - gLW * dPD] = s1b(&uAux[j + 1]) / dxi;
00649                 uAux[j + 2 - gLW * dPD] = s1f(&uAux[j + 2]) / dxi;
00650                 uAux[j + 3 - gLW * dPD] = s1f(&uAux[j + 3]) / dxi;
00651                 uAux[j + 4 - gLW * dPD] = s1f(&uAux[j + 4]) / dxi;
00652                 uAux[j + 5 - gLW * dPD] = s1f(&uAux[j + 5]) / dxi;
00653             }
00654         }
00655         break;
00656     case 2:
00657         for (int i = 0; i < perpPlainSize; i++) {
00658             for (int j = (i * dirWidth + gLW) * dPD;
00659                  j < (i * dirWidth + gLW + dirWidthO) * dPD; j += dPD) {
00660                 uAux[j + 0 - gLW * dPD] = s2b(&uAux[j + 0]) / dxi;
00661                 uAux[j + 1 - gLW * dPD] = s2b(&uAux[j + 1]) / dxi;
00662                 uAux[j + 2 - gLW * dPD] = s2f(&uAux[j + 2]) / dxi;
00663                 uAux[j + 3 - gLW * dPD] = s2f(&uAux[j + 3]) / dxi;
00664                 uAux[j + 4 - gLW * dPD] = s2c(&uAux[j + 4]) / dxi;
00665                 uAux[j + 5 - gLW * dPD] = s2c(&uAux[j + 5]) / dxi;
00666             }
00667         }
00668         break;
00669     case 3:
00670         for (int i = 0; i < perpPlainSize; i++) {
00671             for (int j = (i * dirWidth + gLW) * dPD;
00672                  j < (i * dirWidth + gLW + dirWidthO) * dPD; j += dPD) {
00673                 uAux[j + 0 - gLW * dPD] = s3b(&uAux[j + 0]) / dxi;
00674                 uAux[j + 1 - gLW * dPD] = s3b(&uAux[j + 1]) / dxi;
00675                 uAux[j + 2 - gLW * dPD] = s3f(&uAux[j + 2]) / dxi;
00676                 uAux[j + 3 - gLW * dPD] = s3f(&uAux[j + 3]) / dxi;
00677                 uAux[j + 4 - gLW * dPD] = s3f(&uAux[j + 4]) / dxi;
00678                 uAux[j + 5 - gLW * dPD] = s3f(&uAux[j + 5]) / dxi;
00679             }
00680         }
00681         break;
00682     case 4:
00683         for (int i = 0; i < perpPlainSize; i++) {
00684             for (int j = (i * dirWidth + gLW) * dPD;
00685                  j < (i * dirWidth + gLW + dirWidthO) * dPD; j += dPD) {
00686                 uAux[j + 0 - gLW * dPD] = s4b(&uAux[j + 0]) / dxi;
00687                 uAux[j + 1 - gLW * dPD] = s4b(&uAux[j + 1]) / dxi;
00688                 uAux[j + 2 - gLW * dPD] = s4f(&uAux[j + 2]) / dxi;
00689                 uAux[j + 3 - gLW * dPD] = s4f(&uAux[j + 3]) / dxi;
00690                 uAux[j + 4 - gLW * dPD] = s4c(&uAux[j + 4]) / dxi;
00691                 uAux[j + 5 - gLW * dPD] = s4c(&uAux[j + 5]) / dxi;

```

```

00692     }
00693 }
00694 break;
00695 case 5:
00696     for (int i = 0; i < perpPlainSize; i++) {
00697         for (int j = (i * dirWidth + gLW) * dPD;
00698             j < (i * dirWidth + gLW + dirWidthO) * dPD; j += dPD) {
00699             uAux[j + 0 - gLW * dPD] = s5b(&uAux[j + 0]) / dxi;
00700             uAux[j + 1 - gLW * dPD] = s5b(&uAux[j + 1]) / dxi;
00701             uAux[j + 2 - gLW * dPD] = s5f(&uAux[j + 2]) / dxi;
00702             uAux[j + 3 - gLW * dPD] = s5f(&uAux[j + 3]) / dxi;
00703             uAux[j + 4 - gLW * dPD] = s5f(&uAux[j + 4]) / dxi;
00704             uAux[j + 5 - gLW * dPD] = s5f(&uAux[j + 5]) / dxi;
00705         }
00706     }
00707     break;
00708 case 6:
00709     for (int i = 0; i < perpPlainSize; i++) {
00710         for (int j = (i * dirWidth + gLW) * dPD;
00711             j < (i * dirWidth + gLW + dirWidthO) * dPD; j += dPD) {
00712             uAux[j + 0 - gLW * dPD] = s6b(&uAux[j + 0]) / dxi;
00713             uAux[j + 1 - gLW * dPD] = s6b(&uAux[j + 1]) / dxi;
00714             uAux[j + 2 - gLW * dPD] = s6f(&uAux[j + 2]) / dxi;
00715             uAux[j + 3 - gLW * dPD] = s6f(&uAux[j + 3]) / dxi;
00716             uAux[j + 4 - gLW * dPD] = s6c(&uAux[j + 4]) / dxi;
00717             uAux[j + 5 - gLW * dPD] = s6c(&uAux[j + 5]) / dxi;
00718         }
00719     }
00720     break;
00721 case 7:
00722     for (int i = 0; i < perpPlainSize; i++) {
00723         for (int j = (i * dirWidth + gLW) * dPD;
00724             j < (i * dirWidth + gLW + dirWidthO) * dPD; j += dPD) {
00725             uAux[j + 0 - gLW * dPD] = s7b(&uAux[j + 0]) / dxi;
00726             uAux[j + 1 - gLW * dPD] = s7b(&uAux[j + 1]) / dxi;
00727             uAux[j + 2 - gLW * dPD] = s7f(&uAux[j + 2]) / dxi;
00728             uAux[j + 3 - gLW * dPD] = s7f(&uAux[j + 3]) / dxi;
00729             uAux[j + 4 - gLW * dPD] = s7f(&uAux[j + 4]) / dxi;
00730             uAux[j + 5 - gLW * dPD] = s7f(&uAux[j + 5]) / dxi;
00731         }
00732     }
00733     break;
00734 case 8:
00735     for (int i = 0; i < perpPlainSize; i++) {
00736         for (int j = (i * dirWidth + gLW) * dPD;
00737             j < (i * dirWidth + gLW + dirWidthO) * dPD; j += dPD) {
00738             uAux[j + 0 - gLW * dPD] = s8b(&uAux[j + 0]) / dxi;
00739             uAux[j + 1 - gLW * dPD] = s8b(&uAux[j + 1]) / dxi;
00740             uAux[j + 2 - gLW * dPD] = s8f(&uAux[j + 2]) / dxi;
00741             uAux[j + 3 - gLW * dPD] = s8f(&uAux[j + 3]) / dxi;
00742             uAux[j + 4 - gLW * dPD] = s8c(&uAux[j + 4]) / dxi;
00743             uAux[j + 5 - gLW * dPD] = s8c(&uAux[j + 5]) / dxi;
00744         }
00745     }
00746     break;
00747 case 9:
00748     for (int i = 0; i < perpPlainSize; i++) {
00749         for (int j = (i * dirWidth + gLW) * dPD;
00750             j < (i * dirWidth + gLW + dirWidthO) * dPD; j += dPD) {
00751             uAux[j + 0 - gLW * dPD] = s9b(&uAux[j + 0]) / dxi;
00752             uAux[j + 1 - gLW * dPD] = s9b(&uAux[j + 1]) / dxi;
00753             uAux[j + 2 - gLW * dPD] = s9f(&uAux[j + 2]) / dxi;
00754             uAux[j + 3 - gLW * dPD] = s9f(&uAux[j + 3]) / dxi;
00755             uAux[j + 4 - gLW * dPD] = s9f(&uAux[j + 4]) / dxi;
00756             uAux[j + 5 - gLW * dPD] = s9f(&uAux[j + 5]) / dxi;
00757         }
00758     }
00759     break;
00760 case 10:
00761     for (int i = 0; i < perpPlainSize; i++) {
00762         for (int j = (i * dirWidth + gLW) * dPD;
00763             j < (i * dirWidth + gLW + dirWidthO) * dPD; j += dPD) {
00764             uAux[j + 0 - gLW * dPD] = s10b(&uAux[j + 0]) / dxi;
00765             uAux[j + 1 - gLW * dPD] = s10b(&uAux[j + 1]) / dxi;
00766             uAux[j + 2 - gLW * dPD] = s10f(&uAux[j + 2]) / dxi;
00767             uAux[j + 3 - gLW * dPD] = s10f(&uAux[j + 3]) / dxi;
00768             uAux[j + 4 - gLW * dPD] = s10c(&uAux[j + 4]) / dxi;
00769             uAux[j + 5 - gLW * dPD] = s10c(&uAux[j + 5]) / dxi;
00770         }
00771     }
00772     break;
00773 case 11:
00774     for (int i = 0; i < perpPlainSize; i++) {
00775         for (int j = (i * dirWidth + gLW) * dPD;
00776             j < (i * dirWidth + gLW + dirWidthO) * dPD; j += dPD) {
00777             uAux[j + 0 - gLW * dPD] = s11b(&uAux[j + 0]) / dxi;
00778             uAux[j + 1 - gLW * dPD] = s11b(&uAux[j + 1]) / dxi;

```

```

00779         uAux[j + 2 - gLW * dPD] = s11f(&uAux[j + 2]) / dxi;
00780         uAux[j + 3 - gLW * dPD] = s11f(&uAux[j + 3]) / dxi;
00781         uAux[j + 4 - gLW * dPD] = s11f(&uAux[j + 4]) / dxi;
00782         uAux[j + 5 - gLW * dPD] = s11f(&uAux[j + 5]) / dxi;
00783     }
00784 }
00785 break;
00786 case 12:
00787     for (int i = 0; i < perpPlainSize; i++) {
00788         for (int j = (i * dirWidth + gLW) * dPD;
00789             j < (i * dirWidth + gLW + dirWidth0) * dPD; j += dPD) {
00790             uAux[j + 0 - gLW * dPD] = s12b(&uAux[j + 0]) / dxi;
00791             uAux[j + 1 - gLW * dPD] = s12b(&uAux[j + 1]) / dxi;
00792             uAux[j + 2 - gLW * dPD] = s12f(&uAux[j + 2]) / dxi;
00793             uAux[j + 3 - gLW * dPD] = s12f(&uAux[j + 3]) / dxi;
00794             uAux[j + 4 - gLW * dPD] = s12c(&uAux[j + 4]) / dxi;
00795             uAux[j + 5 - gLW * dPD] = s12c(&uAux[j + 5]) / dxi;
00796         }
00797     }
00798 break;
00799 case 13:
00800     // Iterate through all points in the plane perpendicular to the given
00801     // direction
00802     for (int i = 0; i < perpPlainSize; i++) {
00803         // Iterate through the direction for each perpendicular plane point
00804         for (int j = (i * dirWidth + gLW /*to shift left by gLW below */) * dPD;
00805             j < (i * dirWidth + gLW + dirWidth0) * dPD; j += dPD) {
00806             /* Compute the stencil derivative for any of the six field components
00807              * with a ghostlayer width adjusted to the order of the finite
00808              * difference scheme */
00809             uAux[j + 0 - gLW * dPD] = s13b(&uAux[j + 0]) / dxi;
00810             uAux[j + 1 - gLW * dPD] = s13b(&uAux[j + 1]) / dxi;
00811             uAux[j + 2 - gLW * dPD] = s13f(&uAux[j + 2]) / dxi;
00812             uAux[j + 3 - gLW * dPD] = s13f(&uAux[j + 3]) / dxi;
00813             uAux[j + 4 - gLW * dPD] = s13f(&uAux[j + 4]) / dxi;
00814             uAux[j + 5 - gLW * dPD] = s13f(&uAux[j + 5]) / dxi;
00815         }
00816     }
00817 break;
00818 default:
00819     errorKill("Please set an existing stencil order");
00820 break;
00821 }
00822 }
00823 }
00824
00825 // Helper functions //
00826
00827 // Print a specific error message to stdout
00828 void errorKill(const string & errorMessage) {
00829     cerr << endl << "Error: " << errorMessage << " Aborting..." << endl;
00830     MPI_Abort(MPI_COMM_WORLD, 1);
00831     return;
00832 }
00833
00834 /** Check function return value. From CCode examples.
00835     opt == 0 means SUNDIALS function allocates memory so check if
00836     returned NULL pointer
00837     opt == 1 means SUNDIALS function returns an integer value so check if
00838     retval < 0
00839     opt == 2 means function allocates memory so check if returned
00840     NULL pointer */
00841 int check_retval(void *returnvalue, const char *funcname, int opt, int id) {
00842     int *retval = nullptr;
00843
00844     /* Check if SUNDIALS function returned NULL pointer - no memory allocated */
00845     if (opt == 0 && returnvalue == nullptr) {
00846         fprintf(stderr,
00847             "\nSUNDIALS_ERROR(%d): %s() failed - returned NULL pointer\n\n", id,
00848             funcname);
00849         return (1);
00850     }
00851
00852     /* Check if retval < 0 */
00853     else if (opt == 1) {
00854         retval = (int *)returnvalue;
00855         if (*retval < 0) {
00856             fprintf(stderr, "\nSUNDIALS_ERROR(%d): %s() failed with retval = %d\n\n",
00857                 id, funcname, *retval);
00858             return (1);
00859         }
00860     }
00861
00862     /* Check if function returned NULL pointer - no memory allocated */
00863     else if (opt == 2 && returnvalue == nullptr) {
00864         fprintf(stderr,
00865             "\nMEMORY_ERROR(%d): %s() failed - returned NULL pointer\n\n", id,

```

```

00866         funcname);
00867     return (1);
00868 }
00869
00870 return (0);
00871 }

```

6.12 src/LatticePatch.h File Reference

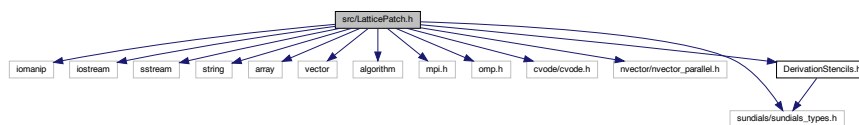
Declaration of the lattice and lattice patches.

```

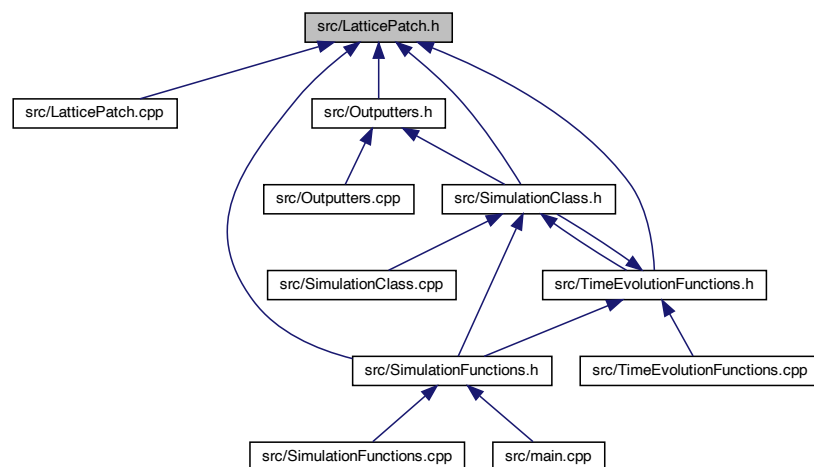
#include <iomanip>
#include <iostream>
#include <sstream>
#include <string>
#include <array>
#include <vector>
#include <algorithm>
#include <mpi.h>
#include <omp.h>
#include <cvode/cvode.h>
#include <nvector/nvector_parallel.h>
#include <sundials/sundials_types.h>
#include "DerivationStencils.h"

```

Include dependency graph for LatticePatch.h:



This graph shows which files directly or indirectly include this file:



Data Structures

- class [Lattice](#)
[Lattice](#) class for the construction of the enveloping discrete simulation space.
- class [LatticePatch](#)
[LatticePatch](#) class for the construction of the patches in the enveloping lattice.

Enumerations

- enum [LatticeOptions](#) { [FLatticeDimensionSet](#) = 0x01 }
 - enum [LatticePatchOptions](#) { [FLatticePatchSetUp](#) = 0x01 , [TranslocationLookupSetUp](#) = 0x02 , [GhostLayersInitialized](#) = 0x04 , [BuffersInitialized](#) = 0x08 }
- lattice patch construction checking flags*

Functions

- void [errorKill](#) (const string &errorMessage)
Print a specific error message to stdout.
- int [check_retval](#) (void *returnvalue, const char *funcname, int opt, int id)

6.12.1 Detailed Description

Declaration of the lattice and lattice patches.

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

6.12.2 Enumeration Type Documentation

6.12.2.1 LatticeOptions

enum [LatticeOptions](#)

Enumerator

FLatticeDimensionSet	
--------------------------------------	--

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

```
00037     {
00038         FLatticeDimensionSet = 0x01, // 1
00039         /*OPT_B = 0x02, // 2
00040         OPT_C = 0x04, // 4
00041         OPT_D = 0x08, // 8
00042         OPT_E = 0x10, // 16
00043         OPT_F = 0x20,*/ // 32
00044     };
```

6.12.2.2 LatticePatchOptions

enum [LatticePatchOptions](#)

lattice patch construction checking flags

Enumerator

FLatticePatchSetUp	
TranslocationLookupSetUp	
GhostLayersInitialized	
BuffersInitialized	

Definition at line 127 of file [LatticePatch.h](#).

```
00127     {
00128     FLatticePatchSetUp = 0x01,
00129     TranslocationLookupSetUp = 0x02,
00130     GhostLayersInitialized = 0x04,
00131     BuffersInitialized = 0x08
00132     /*OPT_D = 0x08,
00133     OPT_E = 0x10,
00134     OPT_F = 0x20,*/
00135 };
```

6.12.3 Function Documentation

6.12.3.1 check_retval()

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

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

Definition at line 841 of file [LatticePatch.cpp](#).

```
00841     {
00842     int *retval = nullptr;
00843
00844     /* Check if SUNDIALS function returned NULL pointer - no memory allocated */
00845     if (opt == 0 && returnvalue == nullptr) {
00846         fprintf(stderr,
00847             "\nSUNDIALS_ERROR(%d): %s() failed - returned NULL pointer\n\n", id,
00848             funcname);
00849         return (1);
00850     }
00851
00852     /* Check if retval < 0 */
00853     else if (opt == 1) {
00854         retval = (int *)returnvalue;
00855         if (*retval < 0) {
00856             fprintf(stderr, "\nSUNDIALS_ERROR(%d): %s() failed with retval = %d\n\n",
00857                 id, funcname, *retval);
00858             return (1);
00859         }
00860     }
00861 }
```

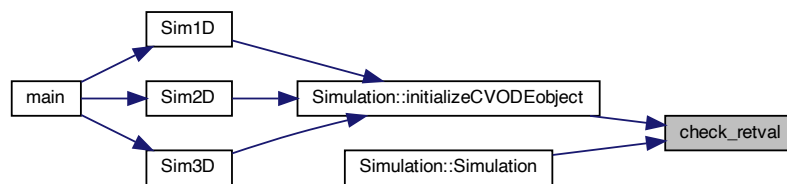
```

00862  /* Check if function returned NULL pointer - no memory allocated */
00863  else if (opt == 2 && returnvalue == nullptr) {
00864      fprintf(stderr,
00865              "\nMEMORY_ERROR(%d): %s() failed - returned NULL pointer\n\n", id,
00866              funcname);
00867      return (1);
00868  }
00869
00870  return (0);
00871 }

```

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

Here is the caller graph for this function:



6.12.3.2 errorKill()

```

void errorKill (
    const string & errorMessage )

```

Print a specific error message to stdout.

Definition at line 828 of file [LatticePatch.cpp](#).

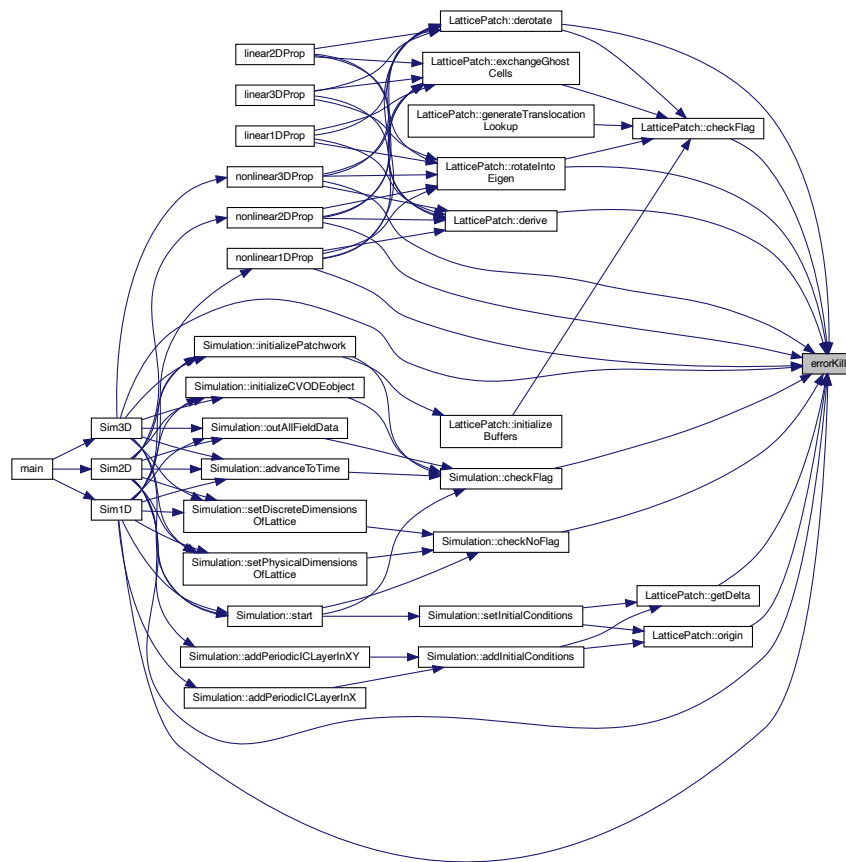
```

00828  {
00829      cerr << endl << "Error: " << errorMessage << " Aborting..." << endl;
00830      MPI_Abort(MPI_COMM_WORLD, 1);
00831      return;
00832  }

```

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

Here is the caller graph for this function:



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 // Include Guard
00007 #ifndef LATTICEPATCH
00008 #define LATTICEPATCH
00009
00010 // IO
00011 #include <iomanip>
00012 #include <iostream>
00013 #include <sstream>
00014
00015 // string, container, algorithm
00016 #include <string>
00017 // #include <string_view>
00018 #include <array>
00019 #include <vector>
00020 #include <algorithm>
00021
00022 // MPI & OpenMP
00023 #include <mpi.h>
00024 #include <omp.h>
00025
00026 // Sundials
00027 #include <cvode/cvode.h> /* prototypes for CVODE fcts. */
00028 #include <nvector/nvector_parallel.h> /* definition of N_Vector and macros */
00029 #include <sundials/sundials_types.h> /* definition of type sunrealtype */

```

```

00030
00031 // stencils
00032 #include "DerivationStencils.h"
00033
00034 using namespace std;
00035
00036 // lattice construction checking flags
00037 enum LatticeOptions {
00038     FLatticeDimensionSet = 0x01, // 1
00039     /*OPT_B = 0x02, // 2
00040     OPT_C = 0x04, // 4
00041     OPT_D = 0x08, // 8
00042     OPT_E = 0x10, // 16
00043     OPT_F = 0x20,*/ // 32
00044 };
00045
00046 /** @brief Lattice class for the construction of the enveloping discrete
00047  * simulation space */
00048 class Lattice {
00049 private:
00050     /// physical size of the lattice in x-direction
00051     sunrealtype tot_lx;
00052     /// physical size of the lattice in y-direction
00053     sunrealtype tot_ly;
00054     /// physical size of the lattice in z-direction
00055     sunrealtype tot_lz;
00056     /// number of points in x-direction
00057     sunindextype tot_nx;
00058     /// number of points in y-direction
00059     sunindextype tot_ny;
00060     /// number of points in z-direction
00061     sunindextype tot_nz;
00062     /// total number of lattice points
00063     sunindextype tot_noP;
00064     /// dimension of each data point -> set once and for all
00065     static constexpr int dataPointDimension = 6;
00066     /// number of lattice points times data dimension of each point
00067     sunindextype tot_noDP;
00068     /// physical distance between lattice points in x-direction
00069     sunrealtype dx;
00070     /// physical distance between lattice points in y-direction
00071     sunrealtype dy;
00072     /// physical distance between lattice points in z-direction
00073     sunrealtype dz;
00074     /// stencil order
00075     const int stencilOrder;
00076     /// required width of ghost layers (depends on the stencil order)
00077     const int ghostLayerWidth;
00078     /// char for checking if lattice flags are set
00079     unsigned char statusFlags;
00080
00081 public:
00082     /// number of MPI processes
00083     int n_prc;
00084     /// number of MPI process
00085     int my_prc;
00086     /// personal communicator of the lattice
00087     MPI_Comm comm;
00088     /// function to create and deploy the cartesian communicator
00089     void initializeCommunicator(const int nx, const int ny,
00090                               const int nz, const bool per);
00091     /// default construction
00092     Lattice(const int StO);
00093     /// SUNContext object
00094     SUNContext suncctx;
00095     /// SUNProfiler object
00096     SUNProfiler profobj;
00097     /// component function for resizing the discrete dimensions of the lattice
00098     void setDiscreteDimensions(const sunindextype _nx,
00099                               const sunindextype _ny, const sunindextype _nz);
00100     /// component function for resizing the physical size of the lattice
00101     void setPhysicalDimensions(const sunrealtype _lx,
00102                               const sunrealtype _ly, const sunrealtype _lz);
00103     ///@{
00104     /** getter function */
00105     [[nodiscard]] const sunrealtype &get_tot_lx() const { return tot_lx; }
00106     [[nodiscard]] const sunrealtype &get_tot_ly() const { return tot_ly; }
00107     [[nodiscard]] const sunrealtype &get_tot_lz() const { return tot_lz; }
00108     [[nodiscard]] const sunindextype &get_tot_nx() const { return tot_nx; }
00109     [[nodiscard]] const sunindextype &get_tot_ny() const { return tot_ny; }
00110     [[nodiscard]] const sunindextype &get_tot_nz() const { return tot_nz; }
00111     [[nodiscard]] const sunindextype &get_tot_noP() const { return tot_noP; }
00112     [[nodiscard]] const sunindextype &get_tot_noDP() const { return tot_noDP; }
00113     [[nodiscard]] const sunrealtype &get_dx() const { return dx; }
00114     [[nodiscard]] const sunrealtype &get_dy() const { return dy; }
00115     [[nodiscard]] const sunrealtype &get_dz() const { return dz; }
00116     [[nodiscard]] constexpr int get_dataPointDimension() const {

```

```

00117     return dataPointDimension;
00118 }
00119 [[nodiscard]] const int &get_stencilOrder() const { return stencilOrder; }
00120 [[nodiscard]] const int &get_ghostLayerWidth() const {
00121     return ghostLayerWidth;
00122 }
00123 ///@}
00124 };
00125
00126 /// lattice patch construction checking flags
00127 enum LatticePatchOptions {
00128     FLatticePatchSetUp = 0x01,
00129     TranslocationLookupSetUp = 0x02,
00130     GhostLayersInitialized = 0x04,
00131     BuffersInitialized = 0x08
00132     /*OPT_D = 0x08,
00133     OPT_E = 0x10,
00134     OPT_F = 0x20,*/
00135 };
00136
00137 /** @brief LatticePatch class for the construction of the patches in the
00138  * enveloping lattice */
00139 class LatticePatch {
00140 private:
00141     /// origin of the patch in physical space; x-coordinate
00142     sunrealtype x0;
00143     /// origin of the patch in physical space; y-coordinate
00144     sunrealtype y0;
00145     /// origin of the patch in physical space; z-coordinate
00146     sunrealtype z0;
00147     /// inner position of lattice-patch in the lattice patchwork; x-points
00148     sunindextype Llx;
00149     /// inner position of lattice-patch in the lattice patchwork; y-points
00150     sunindextype Lly;
00151     /// inner position of lattice-patch in the lattice patchwork; z-points
00152     sunindextype Llz;
00153     /// physical size of the lattice-patch in the x-dimension
00154     sunrealtype lx;
00155     /// physical size of the lattice-patch in the y-dimension
00156     sunrealtype ly;
00157     /// physical size of the lattice-patch in the z-dimension
00158     sunrealtype lz;
00159     /// number of points in the lattice patch in the x-dimension
00160     sunindextype nx;
00161     /// number of points in the lattice patch in the y-dimension
00162     sunindextype ny;
00163     /// number of points in the lattice patch in the z-dimension
00164     sunindextype nz;
00165     /// physical distance between lattice points in x-direction
00166     sunrealtype dx;
00167     /// physical distance between lattice points in y-direction
00168     sunrealtype dy;
00169     /// physical distance between lattice points in z-direction
00170     sunrealtype dz;
00171     /// pointer to the enveloping lattice
00172     const Lattice *envelopeLattice;
00173     ///@{
00174     /** translocation lookup table */
00175     vector<int> uTox, uToy, uToz, xTou, yTou, zTou;
00176     ///@}
00177     /** aid (auxilliarily) vector including ghost cells to compute the derivatives
00178     vector<sunrealtype> uAux;
00179     ///@{
00180     /** buffer to save spatial derivative values */
00181     vector<sunrealtype> buffX, buffY, buffZ;
00182     ///@}
00183     ///@{
00184     /** buffer for passing ghost cell data */
00185     vector<sunrealtype> ghostCellLeft, ghostCellRight, ghostCellLeftToSend,
00186     ghostCellRightToSend, ghostCellsToSend, ghostCells;
00187     ///@}
00188     ///@{
00189     /** ghost cell translocation lookup table */
00190     vector<int> lgcTox, rgcTox, lgcToy, rgcToy, lgcToz, rgcToz;
00191     ///@}
00192     /** char for checking flags */
00193     unsigned char statusFlags;
00194     ///@{
00195     /** rotate and translocate an input array according to a lookup into an output
00196     * array */
00197     inline void rotateToX(sunrealtype *outArray, const sunrealtype *inArray,
00198     const vector<int> &lookup);
00199     inline void rotateToY(sunrealtype *outArray, const sunrealtype *inArray,
00200     const vector<int> &lookup);
00201     inline void rotateToZ(sunrealtype *outArray, const sunrealtype *inArray,
00202     const vector<int> &lookup);
00203     ///@}

```

```

00204 public:
00205     /// ID of the LatticePatch, corresponds to process number
00206     // (required solely for debugging)
00207     int ID;
00208     /// N_Vector for saving field components u=(E,B) in lattice points
00209     N_Vector u;
00210     /// N_Vector for saving temporal derivatives of the field data
00211     N_Vector du;
00212     /// pointer to field data
00213     sunrealtype *uData;
00214     /// pointer to auxiliary data vector
00215     sunrealtype *uAuxData;
00216     /// pointer to time-derivative data
00217     sunrealtype *duData;
00218     ///@{
00219     /** pointer to halo data */
00220     sunrealtype *gCLData, *gCRData;
00221     ///@}
00222     /// pointer to spatial derivative data buffers
00223     array<sunrealtype *, 3> buffData;
00224     /// constructor setting up a default first lattice patch
00225     LatticePatch();
00226     /// destructor freeing parallel vectors
00227     ~LatticePatch();
00228     /// friend function for creating the patchwork slicing of the overall lattice
00229     friend int generatePatchwork(const Lattice &envelopeLattice,
00230                                LatticePatch &patchToMold, const int DLx,
00231                                const int DLy, const int DLz);
00232     /// function to get the discrete size of the LatticePatch
00233     // (0 direction corresponds to total)
00234     int discreteSize(int dir=0) const;
00235     /// function to get the origin of the patch
00236     sunrealtype origin(const int dir) const;
00237     /// function to get distance between points
00238     sunrealtype getDelta(const int dir) const;
00239     /// function to fill out the lookup tables
00240     // for translocation and de-translocation of data point
00241     void generateTranslocationLookup();
00242     /// function to rotate u into Z-matrix eigenraum
00243     // and make it the primary lattice direction of dir
00244     void rotateIntoEigen(const int dir);
00245     /// function to derotate uAux into dudata lattice direction of x
00246     void derotate(int dir, sunrealtype *buffOut);
00247     /// initialize ghost cells for halo exchange
00248     void initializeGhostLayer();
00249     /// initialize buffers to save derivatives
00250     void initializeBuffers();
00251     /// function to exchange ghost cells in uAux for the derivative
00252     void exchangeGhostCells(const int dir);
00253     /// function to derive the centered values in uAux and save them noncentered
00254     void derive(const int dir);
00255     /// function to check if a flag has been set and if not abort
00256     void checkFlag(unsigned int flag) const;
00257 };
00258
00259 // helper function for error messages
00260 void errorKill(const string & errorMessage);
00261
00262 // helper function to check for CCode success
00263 int check_retval(void *returnvalue, const char *funcname, int opt, int id);
00264
00265 // End of Includeguard
00266 #endif

```

6.14 src/main.cpp File Reference

Main function to configure the user's simulation settings.

- included processes of the weak-field expansion, see [README.md](#)
- physical total simulation time
- discrete time steps
- output step multiples

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 CNode 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

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

A 3D simulation with specified

- relative and absolute tolerances of the CNode 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
- periodic or non-periodic boundaries
- processes of the weak-field expansion, see [README.md](#)
- physical total simulation time
- discrete time steps
- output step multiples

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

- 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 8 of file [main.cpp](#).

```
00009 {
00010     // Initialize MPI environment
00011     MPI_Init (&argc, &argv);
00012     MPI_Comm comm = MPI_COMM_WORLD;
00013     // Prepare MPI for Master-only threading
00014     //int provided;
00015     //MPI_Init_thread(&argc, &argv, MPI_THREAD_FUNNELED, &provided);
00016
00017     int rank = 0;
00018     MPI_Comm_rank(comm,&rank);
00019     double ti=MPI_Wtime(); // Overall start time
00020
00021     /** Determine the output directory.
00022      * A "SimResults" folder will be created if non-existent
00023      * with a subdirectory named in the identifier format
00024      * "yy-mm-dd_hh-MM-ss" that contains the csv files */
00025     constexpr auto outputDirectory = "/path/to/directory/";
00026
00027
00028     //----- BEGIN OF CONFIGURATION -----//
00029
00030     //////////////// - 1D - //////////////////////
00031     /** A 1D simulation with specified */
00032
00033     /**/ Specify your settings here /**/
00034     constexpr array <sunrealtype,2> CNodeTolerances={1.0e-16,1.0e-16}; /// - relative and absolute
00035     tolerances of the CNode solver
00036     constexpr int StencilOrder=13; /// - accuracy order of the
00037     stencils in the range 1-13
00038     constexpr sunrealtype physical_sidelength=300e-6; /// - physical length of the
00039     lattice in meters
00040     constexpr sunindextype latticepoints=6e3; /// - number of lattice points
00041     constexpr bool periodic=true; /// - periodic or vanishing
00042     boundary values
00043     int processOrder=3; /// - included processes of the
00044     weak-field expansion, see README.md
00045     constexpr sunrealtype simulationTime=100.0e-6l; /// - physical total
00046     simulation time
00047     constexpr int numberOfSteps=100; /// - discrete time steps
00048     constexpr int outputStep=100; /// - output step multiples
00049 }
```

```

00044     /// Add electromagnetic waves.
00045     planewave plane1;           /// A plane wave with
00046     plane1.k = {1e5,0,0};      /// - wavevector (normalized to \f$ 1/\lambda \f$)
00047     plane1.p = {0,0,0.1};      /// - amplitude/polarization
00048     plane1.phi = {0,0,0};      /// - phase shift
00049     planewave plane2;          /// Another plane wave with
00050     plane2.k = {-1e6,0,0};      /// - wavevector (normalized to \f$ 1/\lambda \f$)
00051     plane2.p = {0,0,0.5};      /// - amplitude/polarization
00052     plane2.phi = {0,0,0};      /// - phase shift
00053     // Do not comment out this vector, even if no plane wave is used. But if, emplace used plane
waves.
00054     vector<planewave> planewaves;
00055     //planewaves.emplace_back(plane1);
00056     //planewaves.emplace_back(plane2);
00057
00058     gaussian1D gauss1;          /// A Gaussian wave with
00059     gauss1.k = {1.0e6,0,0};     /// - wavevector (normalized to \f$ 1/\lambda \f$)
00060     gauss1.p = {0,0,0.1};      /// - polarization/amplitude
00061     gauss1.x0 = {100e-6,0,0};   /// - shift from origin
00062     gauss1.phig = 5e-6;         /// - width
00063     gauss1.phi = {0,0,0};      /// - phase shift
00064     gaussian1D gauss2;          /// Another Gaussian with
00065     gauss2.k = {-0.2e6,0,0};    /// - wavevector (normalized to \f$ 1/\lambda \f$)
00066     gauss2.p = {0,0,0.5};      /// - polarization/amplitude
00067     gauss2.x0 = {200e-6,0,0};   /// - shift from origin
00068     gauss2.phig = 15e-6;        /// - width
00069     gauss2.phi = {0,0,0};      /// - phase shift
00070     // Do not comment out this vector, even if no Gaussian wave is used. But if, emplace used Gaussian
waves.
00071     vector<gaussian1D> Gaussians1D;
00072     Gaussians1D.emplace_back(gauss1);
00073     Gaussians1D.emplace_back(gauss2);
00074
00075     /// Do not change this below ///
00076     int *interactions = &processOrder;
00077     Sim1D(CVodeTolerances,StencilOrder,physical_sidelength,latticepoints,
00078           periodic,interactions,simulationTime,numberOfSteps,
00079           outputDirectory,outputStep,
00080           planewaves,Gaussians1D);
00081
00082     //////////////////////////////////////
00083
00084     ////////////////////////////////////// - 2D - //////////////////////////////////////
00085     /** A 2D simulation with specified */
00086
00087     /// Specify your settings here ///
00088     constexpr array<sunrealtype,2> CVodeTolerances={1.0e-12,1.0e-12}; /// - relative and absolute
00089     tolerances of the CVode solver
00090     constexpr int StencilOrder=13; /// - accuracy order of the
00091     stencils in the range 1-13
00092     constexpr array<sunrealtype,2> physical_sidelengths={80e-6,80e-6}; /// - physical length of the
00093     lattice in the given dimensions in meters
00094     constexpr array<sunindextype,2> latticepoints_per_dim={800,800}; /// - number of lattice points
00095     per dimension
00096     constexpr array<int,2> patches_per_dim={2,2}; /// - slicing of discrete
00097     dimensions into patches
00098     constexpr bool periodic=true; /// - periodic or vanishing
00099     boundary values
00100     int processOrder=3; /// - included processes of the
00101     weak-field expansion, see README.md
00102     constexpr sunrealtype simulationTime=40e-6l; /// - physical total simulation
00103     time
00104     constexpr int numberOfSteps=100; /// - discrete time steps
00105     constexpr int outputStep=100; /// - output step multiples
00106
00107     /// Add electromagnetic waves.
00108     planewave plane1;           /// A plane wave with
00109     plane1.k = {1e5,0,0};      /// - wavevector (normalized to \f$ 1/\lambda \f$)
00110     plane1.p = {0,0,0.1};      /// - amplitude/polarization
00111     plane1.phi = {0,0,0};      /// - phase shift
00112     planewave plane2;          /// Another plane wave with
00113     plane2.k = {-1e6,0,0};      /// - wavevector
00114     plane2.p = {0,0,0.5};      /// - amplitude/polarization
00115     plane2.phi = {0,0,0};      /// - phase shift
00116     // Do not comment out this vector, even if no plane wave is used. But if, emplace used plane
waves.
00117     vector<planewave> planewaves;
00118     //planewaves.emplace_back(plane1);
00119     //planewaves.emplace_back(plane2);
00120
00121     gaussian2D gauss1;          /// A Gaussian wave with
00122     gauss1.x0 = {40e-6,40e-6};  /// - center it approaches
00123     gauss1.axis = {1,0};        /// - normalized direction _from_ which the wave approaches the
00124     center
00125     gauss1.amp = 0.5;           /// - amplitude
00126     gauss1.phip = 2*atan(0);     /// - polarization rotation from TE-mode (z-axis)

```

```

00119     gauss1.w0 = 2.3e-6;           /// - taille
00120     gauss1.zr = 16.619e-6;        /// - Rayleigh length
00121     /// the wavelength is determined by the relation \f$ \lambda = \pi w_0^2 / z_R \f$
00122     gauss1.ph0 = 2e-5;           /// - beam center
00123     gauss1.phA = 0.45e-5;        /// - beam length
00124     gaussian2D gauss2;           /// Another Gaussian wave with
00125     gauss2.x0 = {40e-6, 40e-6};   /// - center it approaches
00126     gauss2.axis = {-0.7071, 0.7071}; /// - normalized direction from which the wave approaches the
center
00127     gauss2.amp = 0.5;             /// - amplitude
00128     gauss2.phip = 2*atan(0);      /// - polarization rotation fom TE-mode (z-axis)
00129     gauss2.w0 = 2.3e-6;          /// - taille
00130     gauss2.zr = 16.619e-6;        /// - Rayleigh length
00131     gauss2.ph0 = 2e-5;           /// - beam center
00132     gauss2.phA = 0.45e-5;        /// - beam length
00133     // Do not comment out this vector, even if no Gaussian wave is used. But if, emplace used Gaussian
waves.
00134     vector<gaussian2D> Gaussians2D;
00135     Gaussians2D.emplace_back(gauss1);
00136     Gaussians2D.emplace_back(gauss2);
00137
00138     /// Do not change this below ///
00139     static_assert(latticepoints_per_dim[0]*patches_per_dim[0]==0 &&
00140         latticepoints_per_dim[1]*patches_per_dim[1]==0,
00141         "The number of lattice points in each dimension must be "
00142         "divisible by the number of patches in that direction.");
00143     int * interactions = &processOrder;
00144     Sim2D(CVodeTolerances, StencilOrder, physical_sidelengths,
00145         latticepoints_per_dim, patches_per_dim, periodic, interactions,
00146         simulationTime, numberOfSteps, outputDirectory, outputStep,
00147         planewaves, Gaussians2D);
00148
00149     //////////////////////////////////////
00150
00151     ////////////////////////////////////// - 3D - //////////////////////////////////////
00152     /** A 3D simulation with specified */
00153
00154     /// Specify your settings here ///
00155     constexpr array<sunrealtype, 2> CVodeTolerances={1.0e-12, 1.0e-12};           /// - relative and
absolute tolerances of the CVode solver
00156     constexpr int StencilOrder=4;           /// - accuracy order of
the stencils in the range 1-13
00157     constexpr array<sunrealtype, 3> physical_sidelengths={80e-6, 80e-6, 20e-6}; /// - physical dimensions
in meters
00158     constexpr array<sunindextype, 3> latticepoints_per_dim={800, 800, 200};      /// - number of lattice
points in any dimension
00159     constexpr array<int, 3> patches_per_dim= {8, 8, 2};           /// - slicing of discrete
dimensions into patches
00160     constexpr bool periodic=false;           /// - perodic or
non-periodic boundaries
00161     int processOrder=3;           /// - processes of the
weak-field expansion, see README.md
00162     constexpr sunrealtype simulationTime=20e-6;           /// - physical total
simulation time
00163     constexpr int numberOfSteps=50;           /// - discrete time steps
00164     constexpr int outputStep=50;           /// - output step
00165
00166     multiples
00167     /// Add electromagnetic waves.
00168     planewave plane1;           /// A plane wave with
00169     plane1.k = {1e5, 0, 0};      /// - wavevector (normalized to \f$ 1/\lambda \f$)
00170     plane1.p = {0, 0, 0.1};      /// - amplitude/polarization
00171     plane1.phi = {0, 0, 0};      /// - phase shift
00172     planewave plane2;           /// Another plane wave with
00173     plane2.k = {-1e6, 0, 0};      /// - wavevector (normalized to \f$ 1/\lambda \f$)
00174     plane2.p = {0, 0, 0.5};      /// - amplitude/polarization
00175     plane2.phi = {0, 0, 0};      /// - phase shift
00176     // Do not comment out this vector, even if no plane wave is used. But if, emplace used plane
waves.
00177     vector<planewave> planewaves;
00178     //planewaves.emplace_back(plane1);
00179     //planewaves.emplace_back(plane2);
00180
00181     gaussian3D gauss1;           /// A Gaussian wave with
00182     gauss1.x0 = {40e-6, 40e-6, 10e-6}; /// - center it approaches
00183     gauss1.axis = {1, 0, 0};      /// - normalized direction _from_ which the wave approaches
the center
00184     gauss1.amp = 0.05;           /// - amplitude
00185     gauss1.phip = 2*atan(0);      /// - polarization rotation from TE-mode (z-axis)
00186     gauss1.w0 = 3.5e-6;          /// - taille
00187     gauss1.zr = 19.242e-6;        /// - Rayleigh length
00188     /// the wavelength is determined by the relation \f$ \lambda = \pi w_0^2 / z_R \f$
00189     gauss1.ph0 = 2e-5;           /// - beam center
00190     gauss1.phA = 0.45e-5;        /// - beam length
00191     gaussian3D gauss2;           /// Another Gaussian wave with
00192     gauss2.x0 = {40e-6, 40e-6, 10e-6}; /// - center it approaches

```

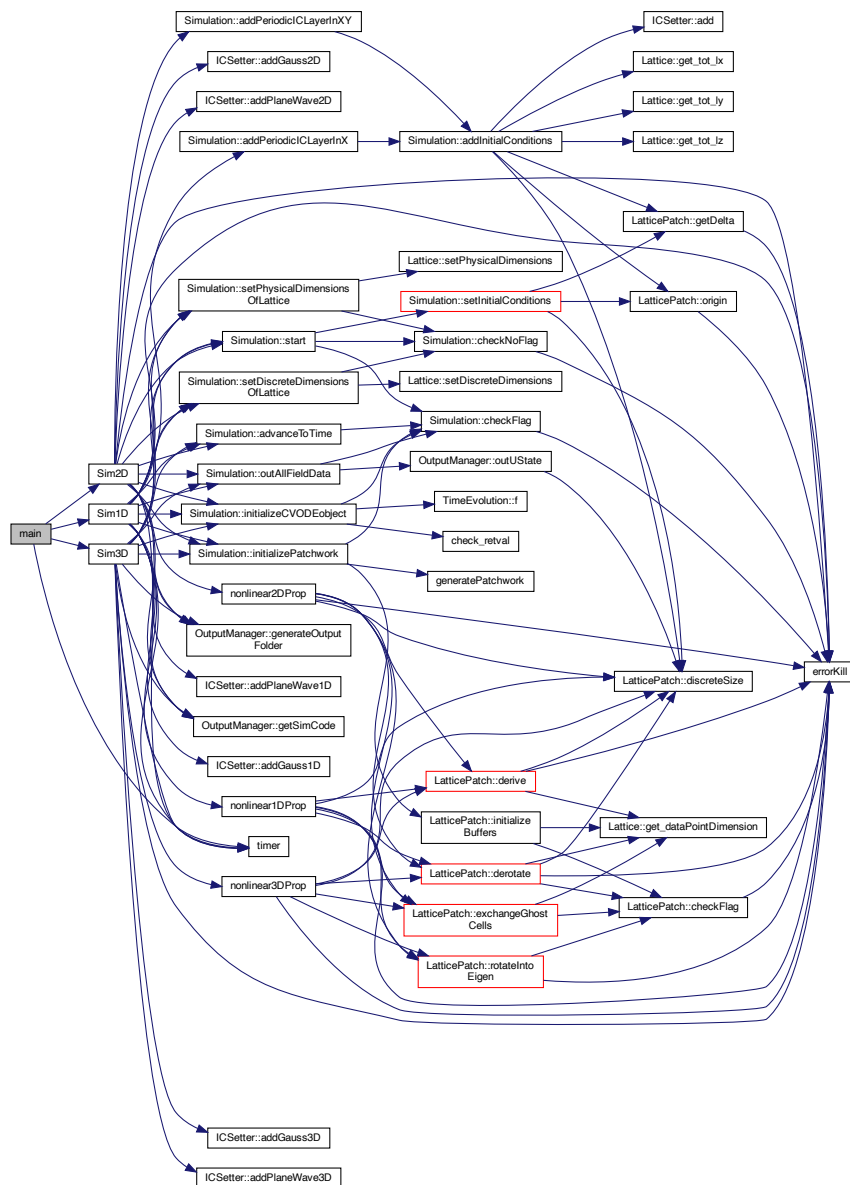
```

00192     gauss2.axis = {0,1,0};           /// - normalized direction from which the wave approaches the
center
00193     gauss2.amp = 0.05;               /// - amplitude
00194     gauss2.phip = 2*atan(0);         /// - polarization rotation from TE-mode (z-axis)
00195     gauss2.w0 = 3.5e-6;             /// - taille
00196     gauss2.zr = 19.242e-6;          /// - Rayleigh length
00197     gauss2.ph0 = 2e-5;              /// - beam center
00198     gauss2.phA = 0.45e-5;           /// - beam length
00199     // Do not comment out this vector, even if no Gaussian wave is used. But if, emplace used Gaussian
waves.
00200     vector<gaussian3D> Gaussians3D;
00201     Gaussians3D.emplace_back(gauss1);
00202     Gaussians3D.emplace_back(gauss2);
00203
00204     /// Do not change this below ///
00205     static_assert(latticepoints_per_dim[0]*patches_per_dim[0]==0 &&
00206         latticepoints_per_dim[1]*patches_per_dim[1]==0 &&
00207         latticepoints_per_dim[2]*patches_per_dim[2]==0,
00208         "The number of lattice points in each dimension must be "
00209         "divisible by the number of patches in that direction.");
00210     int *interactions = &processOrder;
00211     Sim3D(CVodeTolerances,StencilOrder,physical_sidelengths,
00212         latticepoints_per_dim,patches_per_dim,periodic,interactions,
00213         simulationTime,numberOfSteps,outputDirectory,outputStep,
00214         planewaves,Gaussians3D);
00215
00216     //////////////////////////////////////
00217
00218     //----- END OF CONFIGURATION -----//
00219
00220     double tf=MPI_Wtime(); // Overall finish time
00221     if(rank==0) {cout<<endl; timer(ti,tf);} // Print the elapsed time
00222
00223     // Finalize MPI environment
00224     MPI_Finalize();
00225
00226     return 0;
00227 }

```

References [planewave::k](#), [gaussian1D::k](#), [planewave::p](#), [gaussian1D::p](#), [planewave::phi](#), [gaussian1D::phi](#), [gaussian1D::phig](#), [Sim1D\(\)](#), [Sim2D\(\)](#), [Sim3D\(\)](#), [timer\(\)](#), and [gaussian1D::x0](#).

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
00008 int main(int argc, char *argv[])
00009 {
00010     // Initialize MPI environment
00011     MPI_Init (&argc, &argv);
00012     MPI_Comm comm = MPI_COMM_WORLD;
00013     // Prepare MPI for Master-only threading
00014     //int provided;
```



```

00015 //MPI_Init_thread(&argc, &argv, MPI_THREAD_FUNNELED, &provided);
00016
00017 int rank = 0;
00018 MPI_Comm_rank(comm, &rank);
00019 double ti=MPI_Wtime(); // Overall start time
00020
00021 /** Determine the output directory.
00022  * A "SimResults" folder will be created if non-existent
00023  * with a subdirectory named in the identifier format
00024  * "yy-mm-dd_hh-MM-ss" that contains the csv files */
00025 constexpr auto outputDirectory = "/path/to/directory/";
00026
00027
00028 //----- BEGIN OF CONFIGURATION -----//
00029
00030 //////////////// - 1D - ////////////////
00031 /** A 1D simulation with specified */
00032
00033 ///// Specify your settings here ////
00034 constexpr array<sunrealtype,2> CNodeTolerances={1.0e-16,1.0e-16}; /// - relative and absolute
tolerances of the CNode solver
00035 constexpr int StencilOrder=13; /// - accuracy order of the
stencils in the range 1-13
00036 constexpr sunrealtype physical_sidelength=300e-6; /// - physical length of the
lattice in meters
00037 constexpr sunindextype latticepoints=6e3; /// - number of lattice points
00038 constexpr bool periodic=true; /// - periodic or vanishing
boundary values
00039 int processOrder=3; /// - included processes of the
weak-field expansion, see README.md
00040 constexpr sunrealtype simulationTime=100.0e-6l; /// - physical total
simulation time
00041 constexpr int numberOfSteps=100; /// - discrete time steps
00042 constexpr int outputStep=100; /// - output step multiples
00043
00044 ///// Add electromagnetic waves.
00045 planewave plane1; /// A plane wave with
00046 plane1.k = {1e5,0,0}; /// - wavevector (normalized to \f$ 1/\lambda \f$)
00047 plane1.p = {0,0,0.1}; /// - amplitude/polarization
00048 plane1.phi = {0,0,0}; /// - phase shift
00049 planewave plane2; /// Another plane wave with
00050 plane2.k = {-1e6,0,0}; /// - wavevector (normalized to \f$ 1/\lambda \f$)
00051 plane2.p = {0,0,0.5}; /// - amplitude/polarization
00052 plane2.phi = {0,0,0}; /// - phase shift
00053 // Do not comment out this vector, even if no plane wave is used. But if, emplace used plane
waves.
00054 vector<planewave> planewaves;
00055 //planewaves.emplace_back(plane1);
00056 //planewaves.emplace_back(plane2);
00057
00058 gaussian1D gauss1; /// A Gaussian wave with
00059 gauss1.k = {1.0e6,0,0}; /// - wavevector (normalized to \f$ 1/\lambda \f$)
00060 gauss1.p = {0,0,0.1}; /// - polarization/amplitude
00061 gauss1.x0 = {100e-6,0,0}; /// - shift from origin
00062 gauss1.phig = 5e-6; /// - width
00063 gauss1.phi = {0,0,0}; /// - phase shift
00064 gaussian1D gauss2; /// Another Gaussian with
00065 gauss2.k = {-0.2e6,0,0}; /// - wavevector (normalized to \f$ 1/\lambda \f$)
00066 gauss2.p = {0,0,0.5}; /// - polarization/amplitude
00067 gauss2.x0 = {200e-6,0,0}; /// - shift from origin
00068 gauss2.phig = 15e-6; /// - width
00069 gauss2.phi = {0,0,0}; /// - phase shift
00070 // Do not comment out this vector, even if no Gaussian wave is used. But if, emplace used Gaussian
waves.
00071 vector<gaussian1D> Gaussians1D;
00072 Gaussians1D.emplace_back(gauss1);
00073 Gaussians1D.emplace_back(gauss2);
00074
00075 ///// Do not change this below ////
00076 int *interactions = &processOrder;
00077 Sim1D(CNodeTolerances, StencilOrder, physical_sidelength, latticepoints,
periodic, interactions, simulationTime, numberOfSteps,
00078 outputDirectory, outputStep,
00079 planewaves, Gaussians1D);
00080
00081
00082 ////////////////
00083
00084 //////////////// - 2D - ////////////////
00085 /** A 2D simulation with specified */
00086
00087 ///// Specify your settings here ////
00088 constexpr array<sunrealtype,2> CNodeTolerances={1.0e-12,1.0e-12}; /// - relative and absolute
tolerances of the CNode solver
00089 constexpr int StencilOrder=13; /// - accuracy order of the
stencils in the range 1-13
00090 constexpr array<sunrealtype,2> physical_sidelengths={80e-6,80e-6}; /// - physical length of the

```

```

lattice in the given dimensions in meters
00092     constexpr array<sunindextype,2> latticepoints_per_dim={800,800};    /// - number of lattice points
per dimension
00093     constexpr array<int,2> patches_per_dim={2,2};                      /// - slicing of discrete
dimensions into patches
00094     constexpr bool periodic=true;                                       /// - periodic or vanishing
boundary values
00095     int processOrder=3;                                                 /// - included processes of the
weak-field expansion, see README.md
00096     constexpr sunrealtype simulationTime=40e-6l;                       /// - physical total simulation
time
00097     constexpr int numberOfSteps=100;                                    /// - discrete time steps
00098     constexpr int outputStep=100;                                       /// - output step multiples
00099
00100     /// Add electromagnetic waves.
00101     planewave plane1;                                                    /// A plane wave with
00102     plane1.k = {1e5,0,0};                                                /// - wavevector (normalized to  $\frac{1}{\lambda}$ )
00103     plane1.p = {0,0,0.1};                                                /// - amplitude/polarization
00104     plane1.phi = {0,0,0};                                                /// - phase shift
00105     planewave plane2;                                                    /// Another plane wave with
00106     plane2.k = {-1e6,0,0};                                                /// - wavevector
00107     plane2.p = {0,0,0.5};                                                /// - amplitude/polarization
00108     plane2.phi = {0,0,0};                                                /// - phase shift
00109     // Do not comment out this vector, even if no plane wave is used. But if, emplace used plane
waves.
00110     vector<planewave> planewaves;
00111     //planewaves.emplace_back(plane1);
00112     //planewaves.emplace_back(plane2);
00113
00114     gaussian2D gauss1;                                                    /// A Gaussian wave with
00115     gauss1.x0 = {40e-6,40e-6};                                           /// - center it approaches
00116     gauss1.axis = {1,0};                                                  /// - normalized direction _from_ which the wave approaches the
center
00117     gauss1.amp = 0.5;                                                     /// - amplitude
00118     gauss1.phip = 2*atan(0);                                              /// - polarization rotation from TE-mode (z-axis)
00119     gauss1.w0 = 2.3e-6;                                                  /// - taille
00120     gauss1.zr = 16.619e-6;                                                /// - Rayleigh length
00121     /// the wavelength is determined by the relation  $\frac{1}{\lambda} = \frac{\pi w_0^2}{z_R}$ 
00122     gauss1.ph0 = 2e-5;                                                    /// - beam center
00123     gauss1.phA = 0.45e-5;                                                /// - beam length
00124     gaussian2D gauss2;                                                    /// Another Gaussian wave with
00125     gauss2.x0 = {40e-6,40e-6};                                           /// - center it approaches
00126     gauss2.axis = {-0.7071,0.7071};                                       /// - normalized direction from which the wave approaches the
center
00127     gauss2.amp = 0.5;                                                     /// - amplitude
00128     gauss2.phip = 2*atan(0);                                              /// - polarization rotation fom TE-mode (z-axis)
00129     gauss2.w0 = 2.3e-6;                                                  /// - taille
00130     gauss2.zr = 16.619e-6;                                                /// - Rayleigh length
00131     gauss2.ph0 = 2e-5;                                                    /// - beam center
00132     gauss2.phA = 0.45e-5;                                                /// - beam length
00133     // Do not comment out this vector, even if no Gaussian wave is used. But if, emplace used Gaussian
waves.
00134     vector<gaussian2D> Gaussians2D;
00135     Gaussians2D.emplace_back(gauss1);
00136     Gaussians2D.emplace_back(gauss2);
00137
00138     /// Do not change this below ///
00139     static_assert(latticepoints_per_dim[0]*patches_per_dim[0]==0 &&
00140         latticepoints_per_dim[1]*patches_per_dim[1]==0,
00141         "The number of lattice points in each dimension must be "
00142         "divisible by the number of patches in that direction.");
00143     int * interactions = &processOrder;
00144     Sim2D(CVodeTolerances,StencilOrder,physical_sidelengths,
00145         latticepoints_per_dim,patches_per_dim,periodic,interactions,
00146         simulationTime,numberOfSteps,outputDirectory,outputStep,
00147         planewaves,Gaussians2D);
00148
00149     //////////////////////////////////////
00150
00151
00152     ////////////////////////////////////// - 3D - //////////////////////////////////////
00153     /** A 3D simulation with specified */
00154
00155     /// Specify your settings here ///
00156     constexpr array<sunrealtype,2> CVodeTolerances={1.0e-12,1.0e-12};    /// - relative and
absolute tolerances of the CVode solver
00157     constexpr int StencilOrder=4;                                         /// - accuracy order of
the stencils in the range 1-13
00158     constexpr array<sunrealtype,3> physical_sidelengths={80e-6,80e-6,20e-6}; /// - physical dimensions
in meters
00159     constexpr array<sunindextype,3> latticepoints_per_dim={800,800,200}; /// - number of lattice
points in any dimension
00160     constexpr array<int,3> patches_per_dim= {8,8,2};                    /// - slicing of discrete
dimensions into patches
00161     constexpr bool periodic=false;                                       /// - perodic or
non-periodic boundaries
00162     int processOrder=3;                                                  /// - processes of the

```

```

weak-field expansion, see README.md
00163     constexpr sunrealtype simulationTime=20e-6;           /// - physical total
simulation time
00164     constexpr int numberOfSteps=50;                       /// - discrete time steps
00165     constexpr int outputStep=50;                           /// - output step
multiples
00166     /// Add electromagnetic waves.
00167     planewave plane1;                                       /// A plane wave with
00168     plane1.k = {1e5,0,0};                                   /// - wavevector (normalized to \f$ 1/\lambda \f$)
00169     plane1.p = {0,0,0.1};                                   /// - amplitude/polarization
00170     plane1.phi = {0,0,0};                                   /// - phase shift
00171     planewave plane2;                                       /// Another plane wave with
00172     plane2.k = {-1e6,0,0};                                  /// - wavevector (normalized to \f$ 1/\lambda \f$)
00173     plane2.p = {0,0,0.5};                                   /// - amplitude/polarization
00174     plane2.phi = {0,0,0};                                   /// - phase shift
00175     // Do not comment out this vector, even if no plane wave is used. But if, emplace used plane
waves.
00176     vector<planewave> planewaves;
00177     //planewaves.emplace_back(plane1);
00178     //planewaves.emplace_back(plane2);
00179
00180     gaussian3D gauss1;                                       /// A Gaussian wave with
00181     gauss1.x0 = {40e-6,40e-6,10e-6};                       /// - center it approaches
00182     gauss1.axis = {1,0,0};                                   /// - normalized direction _from_ which the wave approaches
the center
00183     gauss1.amp = 0.05;                                       /// - amplitude
00184     gauss1.phip = 2*atan(0);                                 /// - polarization rotation from TE-mode (z-axis)
00185     gauss1.w0 = 3.5e-6;                                     /// - taille
00186     gauss1.zr = 19.242e-6;                                   /// - Rayleigh length
00187     /// the wavelength is determined by the relation \f$ \lambda = \pi w_0^2 / z_R \f$
00188     gauss1.ph0 = 2e-5;                                       /// - beam center
00189     gauss1.phA = 0.45e-5;                                    /// - beam length
00190     gaussian3D gauss2;                                       /// Another Gaussian wave with
00191     gauss2.x0 = {40e-6,40e-6,10e-6};                       /// - center it approaches
00192     gauss2.axis = {0,1,0};                                   /// - normalized direction from which the wave approaches the
center
00193     gauss2.amp = 0.05;                                       /// - amplitude
00194     gauss2.phip = 2*atan(0);                                 /// - polarization rotation from TE-mode (z-axis)
00195     gauss2.w0 = 3.5e-6;                                     /// - taille
00196     gauss2.zr = 19.242e-6;                                   /// - Rayleigh length
00197     gauss2.ph0 = 2e-5;                                       /// - beam center
00198     gauss2.phA = 0.45e-5;                                    /// - beam length
00199     // Do not comment out this vector, even if no Gaussian wave is used. But if, emplace used Gaussian
waves.
00200     vector<gaussian3D> Gaussians3D;
00201     Gaussians3D.emplace_back(gauss1);
00202     Gaussians3D.emplace_back(gauss2);
00203
00204     /// Do not change this below ///
00205     static_assert(latticepoints_per_dim[0]*patches_per_dim[0]==0 &&
00206         latticepoints_per_dim[1]*patches_per_dim[1]==0 &&
00207         latticepoints_per_dim[2]*patches_per_dim[2]==0,
00208         "The number of lattice points in each dimension must be "
00209         "divisible by the number of patches in that direction.");
00210     int *interactions = &processOrder;
00211     Sim3D(CVodeTolerances,StencilOrder,physical_sidelengths,
00212         latticepoints_per_dim,patches_per_dim,periodic,interactions,
00213         simulationTime,numberOfSteps,outputDirectory,outputStep,
00214         planewaves,Gaussians3D);
00215
00216     //////////////////////////////////////
00217     //----- END OF CONFIGURATION -----//
00218
00219     double tf=MPI_Wtime(); // Overall finish time
00220     if(rank==0) {cout<<endl; timer(ti,tf);} // Print the elapsed time
00221
00222     // Finalize MPI environment
00223     MPI_Finalize();
00224
00225     return 0;
00226 }
00227 }

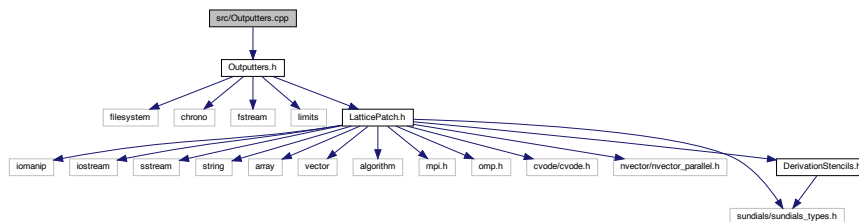
```

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 /// Directly generate the simCode at construction
00009 OutputManager::OutputManager() {
00010     simCode = SimCodeGenerator();
00011     MPI_Comm_rank(MPI_COMM_WORLD, &myPrc);
00012 }
00013
00014 /// Generate the identifier number reverse from year to minute in the format
00015 /// yy-mm-dd_hh-MM-ss
00016 string OutputManager::SimCodeGenerator() {
00017     const chrono::time_point<chrono::system_clock> now{
00018         chrono::system_clock::now()};
00019     const chrono::year_month_day ymd{chrono::floor<chrono::days>(now)};
00020     const auto tod = now - chrono::floor<chrono::days>(now);
00021     const chrono::hh_mm_ss hms{tod};
00022
00023     stringstream temp;
00024     temp << setfill('0') << setw(2)
00025         << static_cast<int>(ymd.year() - chrono::years(2000)) << "-"
00026         << setfill('0') << setw(2) << static_cast<unsigned>(ymd.month()) << "-"
00027         << setfill('0') << setw(2) << static_cast<unsigned>(ymd.day()) << "-"
00028         << setfill('0') << setw(2) << hms.hours().count() << "-" << setfill('0')
00029         << setw(2) << hms.minutes().count() << "-" << setfill('0') << setw(2)
00030         << hms.seconds().count();
00031     ///< " " << hms.subseconds().count(); // subseconds render the filename too
00032     //large
00033     return temp.str();
00034 }
00035
00036 /** Generate the folder to save the data to by one process:
00037  * In the given directory it creates a directory "SimResults" and a directory
00038  * with the simCode. The relevant part of the main file is written to a
00039  * "config.txt" file in that directory to log the settings. */
00040 void OutputManager::generateOutputFolder(const string &dir) {
00041     // Do this only once for the first process
00042     if (myPrc == 0) {
00043         if (!fs::is_directory(dir))

```

```

00044     fs::create_directory(dir);
00045     if (!fs::is_directory(dir + "/SimResults"))
00046         fs::create_directory(dir + "/SimResults");
00047     if (!fs::is_directory(dir + "/SimResults/" + simCode))
00048         fs::create_directory(dir + "/SimResults/" + simCode);
00049 }
00050 // path variable for the output generation
00051 Path = dir + "/SimResults/" + simCode + "/";
00052
00053 ifstream fin("main.cpp");
00054 ofstream fout(Path + "config.txt");
00055 string line;
00056 int begin=1000;
00057 for (int i = 1; !fin.eof(); i++) {
00058     getline(fin, line);
00059     if (line.starts_with("    //----- B")) {
00060         begin=i;
00061     }
00062     if (i < begin) {
00063         continue;
00064     }
00065     fout << line << endl;
00066     if (line.starts_with("    //----- E")) {
00067         break;
00068     }
00069 }
00070
00071 return;
00072 }
00073
00074 /** Write the field data to a csv file from each process (patch) with the field
00075 * data into the simCode directory. The state (simulation step) denotes the
00076 * prefix and the suffix after an underscore is given by the process/patch
00077 * number */
00078 void OutputManager::outUState(const int &state, const LatticePatch &latticePatch) {
00079     ofstream ofs;
00080     ofs.open(Path + to_string(state) + "_" + to_string(myPrc) + ".csv");
00081     // Set precision, number of digits for the values
00082     ofs << setprecision(numeric_limits<sunrealtype>::digits10);
00083
00084     // Walk through each lattice point
00085     for (int i = 0; i < latticePatch.discreteSize() * 6; i += 6) {
00086         // Six columns to contain the field data: Ex,Ey,Ez,Bx,By,Bz
00087         ofs << latticePatch.uData[i + 0] << "," << latticePatch.uData[i + 1] << ","
00088             << latticePatch.uData[i + 2] << "," << latticePatch.uData[i + 3] << ","
00089             << latticePatch.uData[i + 4] << "," << latticePatch.uData[i + 5]
00090             << endl;
00091     }
00092
00093     ofs.close();
00094
00095     return;
00096 }
00097
00098 /// Return the date+time simulation identifier for logging
00099 string OutputManager::getSimCode() { return simCode; }

```

6.18 src/Outputters.h File Reference

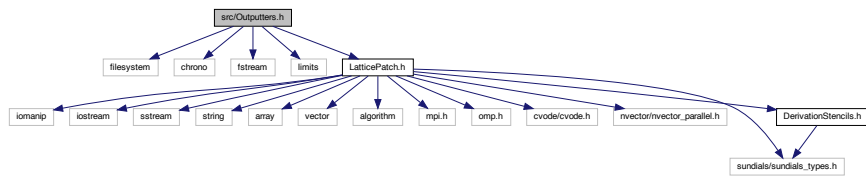
[OutputManager](#) class to outstream simulation data.

```

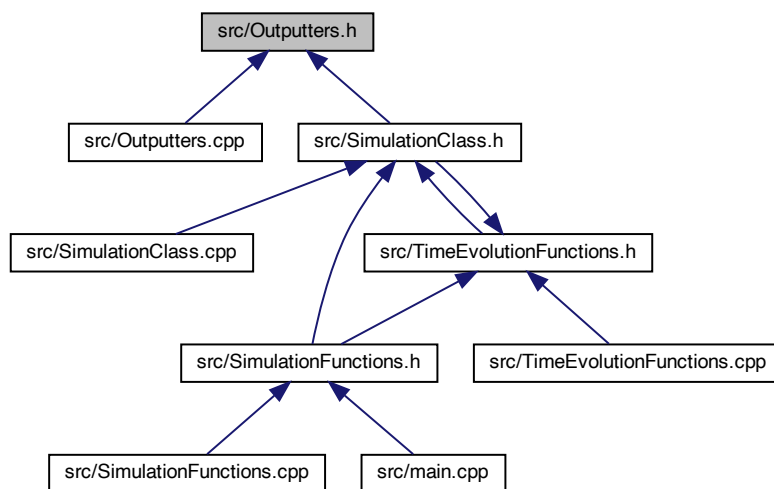
#include <filesystem>
#include <chrono>
#include <fstream>
#include <limits>
#include "LatticePatch.h"

```

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 // Include Guard
00007 #ifndef OUTPUTTERS
00008 #define OUTPUTTERS
00009
00010 // perform operations on the filesystem
00011 #include <filesystem>
00012
00013 // output controlling with limits and timestep
00014 #include <chrono>
00015 #include <fstream>
00016 #include <limits>
00017
00018 // project subfile header
00019 #include "LatticePatch.h"
00020
00021 using namespace std;
00022 namespace fs = std::filesystem;
00023 namespace chrono = std::chrono;
00024
00025 /** @brief Output Manager class to generate and coordinate output writing to
00026  * disk */
00027 class OutputManager {
00028 private:
00029     /// function to create the Code of the Simulations
00030     static string SimCodeGenerator();
00031     /// variable to save the SimCode generated at execution
00032     string simCode;
00033     /// variable for the path to the output folder
00034     string Path;
00035     /// process ID
00036     int myPrc;
00037
00038 public:
00039     /// default constructor
00040     OutputManager();
00041     /// function that creates folder to save simulation info
00042     void generateOutputFolder(const string &dir);
00043     /// output function for the whole lattice
00044     void outUState(const int &state, const LatticePatch &latticePatch);
00045     /// simCode getter function
00046     string getSimCode();
00047 };
00048
00049 // End of Includeguard
00050 #endif

```

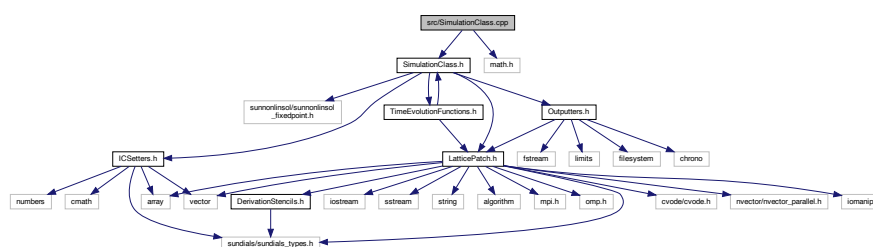
6.20 src/SimulationClass.cpp File Reference

Interface to the whole [Simulation](#) procedure: from wave settings over lattice construction, time evolution and outputs (also all relevant CVODE steps are performed here)

```
#include "SimulationClass.h"
```

```
#include <math.h>
```

Include dependency graph for SimulationClass.cpp:



6.20.1 Detailed Description

Interface to the whole [Simulation](#) procedure: from wave settings over lattice construction, time evolution and outputs (also all relevant CVODE steps are performed here)

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

6.21 SimulationClass.cpp

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

```
00001 ///////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
00002 /// @file SimulationClass.cpp
00003 /// @brief Interface to the whole Simulation procedure:
00004 /// from wave settings over lattice construction, time evolution and outputs
00005 /// (also all relevant CVODE steps are performed here)
00006 ///////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
00007
00008 #include "SimulationClass.h"
00009
00010 #include <math.h>
00011
00012 /// Along with the simulation object, create the cartesian communicator and
00013 /// SUNContext object
00014 Simulation::Simulation(const int nx, const int ny, const int nz,
00015                      const int StencilOrder, const bool periodicity) :
00016     lattice(StencilOrder){
00017     statusFlags = 0;
00018     t = 0;
00019     // Initialize the cartesian communicator
00020     lattice.initializeCommunicator(nx, ny, nz, periodicity);
00021
00022     // Create the SUNContext object associated with the thread of execution
00023     int retval = 0;
00024     retval = SUNContext_Create(&lattice.comm, &lattice.sunctx);
00025     if (check_retval(&retval, "SUNContext_Create", 1, lattice.my_prc))
00026         MPI_Abort(lattice.comm, 1);
00027     // if (flag != CV_SUCCESS) { printf("SUNContext_Create failed, flag=%d.\n",
00028     // flag);
00029     //     MPI_Abort(lattice.comm, 1); }
00030 }
00031
00032 /// Free the CCode solver memory and Sundials context object with the finish of
00033 /// the simulation
00034 Simulation::~Simulation() {
00035     // Free solver memory
00036     if (statusFlags & CcodeObjectSetUp) {
00037         // PrintFinalStats(cvode_mem); // TODO write this function as in cvodes
00038         // cvAdvDiff_bnd.c SUNDIALS_MARK_FUNCTION_END(lattice.profbj);
00039         CVodeFree(&cvode_mem);
00040         SUNContext_Free(&lattice.sunctx);
00041     }
00042 }
00043
00044 /// Set the discrete dimensions, the number of points per dimension
00045 void Simulation::setDiscreteDimensionsOfLattice(const sunindextype nx,
00046         const sunindextype ny, const sunindextype nz) {
00047     checkNoFlag(LatticePatchworkSetUp);
00048     lattice.setDiscreteDimensions(nx, ny, nz);
00049     statusFlags |= LatticeDiscreteSetUp;
00050 }
00051
00052 /// Set the physical dimensions with lengths in micro meters
00053 void Simulation::setPhysicalDimensionsOfLattice(const sunrealtype lx,
00054         const sunrealtype ly, const sunrealtype lz) {
00055     checkNoFlag(LatticePatchworkSetUp);
00056     lattice.setPhysicalDimensions(lx, ly, lz);
00057     statusFlags |= LatticePhysicalSetUp;
00058 }
00059
00060 /// Check that the lattice dimensions are set up and generate the patchwork
00061 void Simulation::initializePatchwork(const int nx, const int ny,
00062         const int nz) {
00063     checkFlag(LatticeDiscreteSetUp);
00064     checkFlag(LatticePhysicalSetUp);
00065
00066     // Generate the patchwork
00067     generatePatchwork(lattice, latticePatch, nx, ny, nz);
00068     latticePatch.initializeBuffers();

```



```

00069
00070     statusFlags |= LatticePatchworkSetUp;
00071 }
00072
00073 /// Configure CVOICE
00074 void Simulation::initializeCVOICEObject(const sunrealtype reitol,
00075     const sunrealtype abstol) {
00076     checkFlag(SimulationStarted);
00077
00078     // CVOICE settings return value
00079     int retval = 0;
00080
00081     // Set the profiler
00082     retval = SUNContext_GetProfiler(lattice.sunctx, &lattice.profobj);
00083     if (check_retval(&retval, "SUNContext_GetProfiler", 1, lattice.my_prc))
00084         MPI_Abort(lattice.comm, 1);
00085     // if (flag != CV_SUCCESS) { printf("SUNContext_GetProfiler failed,
00086     // flag=%d.\n", flag);
00087     //     MPI_Abort(lattice.comm, 1); }
00088
00089     // SUNDIALS_MARK_FUNCTION_BEGIN(profobj);
00090
00091     // Create CVOICE object - returns a pointer to the cvoice memory structure
00092     // with Adams method (Adams-Moulton formula) solver chosen for non-stiff ODE
00093     cvoice_mem = CVOICECreate(CV_ADAMS, lattice.sunctx);
00094
00095     // Specify user data and attach it to the main cvoice memory block
00096     retval = CVOICESetUserData(
00097         cvoice_mem,
00098         &latticePatch); // patch contains the user data as used in CVRhsFn
00099     if (check_retval(&retval, "CVOICESetUserData", 1, lattice.my_prc))
00100         MPI_Abort(lattice.comm, 1);
00101     // if (flag != CV_SUCCESS) { printf("CVOICESetUserData failed, flag=%d.\n",
00102     // flag);
00103     //     MPI_Abort(lattice.comm, 1); }
00104
00105     // Initialize CVOICE solver -> can only be called after start of simulation to
00106     // have data ready Provide required problem and solution specifications,
00107     // allocate internal memory, and initialize cvoice
00108     retval = CVOICEInit(cvoice_mem, TimeEvolution::f, 0,
00109         latticePatch.u); // allocate memory, CVRhsFn f, t_i=0, u
00110         // contains the initial values
00111     if (check_retval(&retval, "CVOICEInit", 1, lattice.my_prc))
00112         MPI_Abort(lattice.comm, 1);
00113     // if (flag != CV_SUCCESS) { printf("CVOICEInit failed, flag=%d.\n", flag);
00114     //     MPI_Abort(lattice.comm, 1); }
00115
00116     // Create fixed point nonlinear solver object (suitable for non-stiff ODE) and
00117     // attach it to CVOICE
00118     SUNNonlinearSolver NLS =
00119         SUNNonlinSol_FixedPoint(latticePatch.u, 0, lattice.sunctx);
00120     retval = CVOICESetNonlinearSolver(cvoice_mem, NLS);
00121     if (check_retval(&retval, "CVOICESetNonlinearSolver", 1, lattice.my_prc))
00122         MPI_Abort(lattice.comm, 1);
00123     // if (flag != CV_SUCCESS) {printf("CVOICESetNonlinearSolver failed,
00124     // flag=%d.\n", flag);
00125     //     MPI_Abort(lattice.comm, 1); }
00126
00127     // Specify the maximum number of steps to be taken by the solver in its
00128     // attempt to reach the next output time
00129     retval = CVOICESetMaxNumSteps(cvoice_mem, 10000);
00130     if (check_retval(&retval, "CVOICESetMaxNumSteps", 1, lattice.my_prc))
00131         MPI_Abort(lattice.comm, 1);
00132     // if (flag != CV_SUCCESS) { printf("CVOICESetMaxNumSteps failed, flag=%d.\n",
00133     // flag);
00134     //     MPI_Abort(lattice.comm, 1); }
00135
00136     // Specify integration tolerances - a scalar relative tolerance and scalar
00137     // absolute tolerance
00138     retval = CVOICEStolerances(cvoice_mem, reitol, abstol);
00139     if (check_retval(&retval, "CVOICEStolerances", 1, lattice.my_prc))
00140         MPI_Abort(lattice.comm, 1);
00141     // if (flag != CV_SUCCESS) { printf("CVOICEStolerances failed, flag=%d.\n",
00142     // flag);
00143     //     MPI_Abort(lattice.comm, 1); }
00144
00145     statusFlags |= CVOICEObjectSetUp;
00146 }
00147
00148 /// Check if the lattice patchwork is set up and set the initial conditions
00149 void Simulation::start() {
00150     checkFlag(LatticeDiscreteSetUp);
00151     checkFlag(LatticePhysicalSetUp);
00152     checkFlag(LatticePatchworkSetUp);
00153     checkNoFlag(SimulationStarted);
00154     checkNoFlag(CVOICEObjectSetUp);
00155     setInitialConditions();

```

```

00156     statusFlags |= SimulationStarted;
00157 }
00158
00159 /// Set initial conditions: Fill the lattice points with the initial field
00160 /// values
00161 void Simulation::setInitialConditions() {
00162     const sunrealtype dx = latticePatch.getDelta(1);
00163     const sunrealtype dy = latticePatch.getDelta(2);
00164     const sunrealtype dz = latticePatch.getDelta(3);
00165     const int nx = latticePatch.discreteSize(1);
00166     const int ny = latticePatch.discreteSize(2);
00167     const sunrealtype x0 = latticePatch.origin(1);
00168     const sunrealtype y0 = latticePatch.origin(2);
00169     const sunrealtype z0 = latticePatch.origin(3);
00170     int px = 0, py = 0, pz = 0;
00171     // space coordinates
00172     for (int i = 0; i < latticePatch.discreteSize() * 6; i += 6) {
00173         px = (i / 6) % nx;
00174         py = ((i / 6) / nx) % ny;
00175         pz = ((i / 6) / nx) / ny;
00176         // Call the 'eval' function to fill the lattice points with the field data
00177         icsettings.eval(static_cast<sunrealtype>(px) * dx + x0,
00178             static_cast<sunrealtype>(py) * dy + y0,
00179             static_cast<sunrealtype>(pz) * dz + z0, &latticePatch.uData[i]);
00180     }
00181     return;
00182 }
00183
00184 /// Use parameters to add periodic IC layers
00185 void Simulation::addInitialConditions(const int xm, const int ym,
00186     const int zm /* zm=0 always */) {
00187     const sunrealtype dx = latticePatch.getDelta(1);
00188     const sunrealtype dy = latticePatch.getDelta(2);
00189     const sunrealtype dz = latticePatch.getDelta(3);
00190     const int nx = latticePatch.discreteSize(1);
00191     const int ny = latticePatch.discreteSize(2);
00192     // Correct for demanded displacement, rest as for setInitialConditions
00193     const sunrealtype x0 = latticePatch.origin(1) + xm*lattice.get_tot_lx();
00194     const sunrealtype y0 = latticePatch.origin(2) + ym*lattice.get_tot_ly();
00195     const sunrealtype z0 = latticePatch.origin(3) + zm*lattice.get_tot_lz();
00196     int px = 0, py = 0, pz = 0;
00197     for (int i = 0; i < latticePatch.discreteSize() * 6; i += 6) {
00198         px = (i / 6) % nx;
00199         py = ((i / 6) / nx) % ny;
00200         pz = ((i / 6) / nx) / ny;
00201         icsettings.add(static_cast<sunrealtype>(px) * dx + x0,
00202             static_cast<sunrealtype>(py) * dy + y0,
00203             static_cast<sunrealtype>(pz) * dz + z0, &latticePatch.uData[i]);
00204     }
00205     return;
00206 }
00207
00208 /// Add initial conditions in one dimension
00209 void Simulation::addPeriodicICLayerInX() {
00210     addInitialConditions(-1, 0, 0);
00211     addInitialConditions(1, 0, 0);
00212     return;
00213 }
00214
00215 /// Add initial conditions in two dimensions
00216 void Simulation::addPeriodicICLayerInXY() {
00217     addInitialConditions(-1, -1, 0);
00218     addInitialConditions(-1, 0, 0);
00219     addInitialConditions(-1, 1, 0);
00220     addInitialConditions(0, 1, 0);
00221     addInitialConditions(0, -1, 0);
00222     addInitialConditions(1, -1, 0);
00223     addInitialConditions(1, 0, 0);
00224     addInitialConditions(1, 1, 0);
00225     return;
00226 }
00227
00228 /// Advance the solution in time - integrate the ODE over an interval t
00229 void Simulation::advanceToTime(const sunrealtype &tEnd) {
00230     checkFlag(SimulationStarted);
00231     int flag = 0;
00232     flag = CNode(cvode_mem, tEnd, latticePatch.u, &t,
00233         CV_NORMAL); // CV_NORMAL: internal steps to reach tEnd, then
00234                     // interpolate to return latticePatch.u, return time
00235                     // reached by the solver as t
00236     if (flag != CV_SUCCESS)
00237         printf("CNode failed, flag=%d.\n", flag);
00238 }
00239
00240 /// Write specified simulations steps to disk
00241 void Simulation::outAllFieldData(const int &state) {
00242     checkFlag(SimulationStarted);

```

```

00243     outputManager.outUState(state, latticePatch);
00244 }
00245
00246 /// Check the presence configuration flags
00247 void Simulation::checkFlag(unsigned int flag) const {
00248     if (!(statusFlags & flag)) {
00249         string errorMessage;
00250         switch (flag) {
00251             case LatticeDiscreteSetUp:
00252                 errorMessage = "The discrete size of the Simulation has not been set up";
00253                 break;
00254             case LatticePhysicalSetUp:
00255                 errorMessage = "The physical size of the Simulation has not been set up";
00256                 break;
00257             case LatticePatchworkSetUp:
00258                 errorMessage = "The patchwork for the Simulation has not been set up";
00259                 break;
00260             case CvodeObjectSetUp:
00261                 errorMessage = "The CVODE object has not been initialized";
00262                 break;
00263             case SimulationStarted:
00264                 errorMessage = "The Simulation has not been started";
00265                 break;
00266             default:
00267                 errorMessage = "Uppss, you've made a non-standard error, sadly I can't "
00268                     "help you there";
00269                 break;
00270         }
00271         errorKill(errorMessage);
00272     }
00273     return;
00274 }
00275
00276 /// Check the absence of configuration flags
00277 void Simulation::checkNoFlag(unsigned int flag) const {
00278     if ((statusFlags & flag)) {
00279         string errorMessage;
00280         switch (flag) {
00281             case LatticeDiscreteSetUp:
00282                 errorMessage =
00283                     "The discrete size of the Simulation has already been set up";
00284                 break;
00285             case LatticePhysicalSetUp:
00286                 errorMessage =
00287                     "The physical size of the Simulation has already been set up";
00288                 break;
00289             case LatticePatchworkSetUp:
00290                 errorMessage = "The patchwork for the Simulation has already been set up";
00291                 break;
00292             case CvodeObjectSetUp:
00293                 errorMessage = "The CVODE object has already been initialized";
00294                 break;
00295             case SimulationStarted:
00296                 errorMessage = "The simulation has already started, some changes are no "
00297                     "longer possible";
00298                 break;
00299             default:
00300                 errorMessage = "Uppss, you've made a non-standard error, sadly I can't "
00301                     "help you there";
00302                 break;
00303         }
00304         errorKill(errorMessage);
00305     }
00306     return;
00307 }

```

6.22 src/SimulationClass.h File Reference

Class for the [Simulation](#) object calling all functionality: from wave settings over lattice construction, time evolution and outputs initialization of the CCode object.

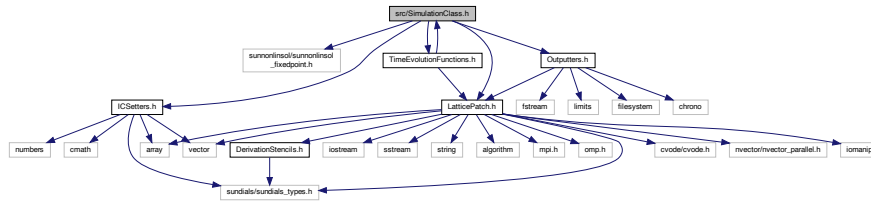
```

#include "sunnonlinsol/sunnonlinsol_fixedpoint.h"
#include "ICSetters.h"
#include "LatticePatch.h"
#include "Outputters.h"

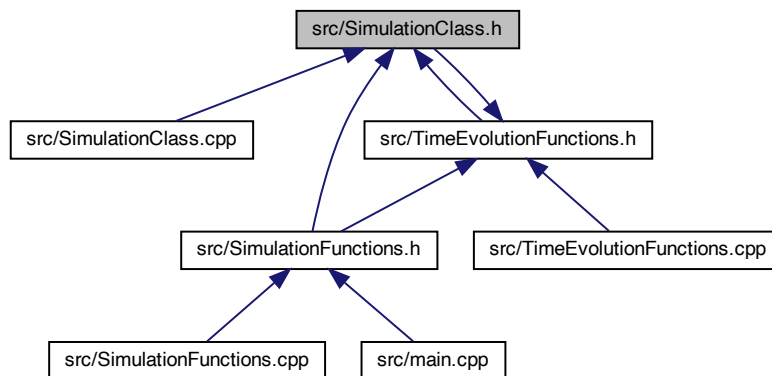
```

```
#include "TimeEvolutionFunctions.h"
```

Include dependency graph for SimulationClass.h:



This graph shows which files directly or indirectly include this file:



Data Structures

- class [Simulation](#)
[Simulation](#) class to instantiate the whole walkthrough of a [Simulation](#).

Enumerations

- enum [SimulationOptions](#) {
[LatticeDiscreteSetUp](#) = 0x01 , [LatticePhysicalSetUp](#) = 0x02 , [LatticePatchworkSetUp](#) = 0x04 ,
[CvodeObjectSetUp](#) = 0x08 ,
[SimulationStarted](#) = 0x10 }
simulation checking flags

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 CNode object.

Definition in file [SimulationClass.h](#).

6.22.2 Enumeration Type Documentation

6.22.2.1 SimulationOptions

enum [SimulationOptions](#)

simulation checking flags

Enumerator

LatticeDiscreteSetUp	
LatticePhysicalSetUp	
LatticePatchworkSetUp	
CvodeObjectSetUp	
SimulationStarted	

Definition at line 24 of file [SimulationClass.h](#).

```

00024 {
00025     LatticeDiscreteSetUp = 0x01,
00026     LatticePhysicalSetUp = 0x02,
00027     LatticePatchworkSetUp = 0x04, // not used anymore
00028     CvodeObjectSetUp = 0x08,
00029     SimulationStarted = 0x10
00030     /*OPT_B = 0x02,
00031     OPT_C = 0x04,
00032     OPT_D = 0x08,
00033     OPT_E = 0x10,
00034     OPT_F = 0x20,*/
00035 };

```

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 CCode object
00006 ///////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
00007
00008 // Include Guard
00009 #ifndef SIMULATIONCLASS
00010 #define SIMULATIONCLASS
00011
00012 /* access to the fixed point SUNNonlinearSolver */
00013 #include "sunnonlinsol/sunnonlinsol_fixedpoint.h"
00014
00015 // project subfile headers
00016 #include "ICSetters.h"
00017 #include "LatticePatch.h"
00018 #include "Outputters.h"
00019 #include "TimeEvolutionFunctions.h"
00020
00021 using namespace std;
00022
00023 /// simulation checking flags
00024 enum SimulationOptions {
00025     LatticeDiscreteSetUp = 0x01,
00026     LatticePhysicalSetUp = 0x02,
00027     LatticePatchworkSetUp = 0x04, // not used anymore
00028     CvodeObjectSetUp = 0x08,
00029     SimulationStarted = 0x10
00030     /*OPT_B = 0x02,

```

```

00031     OPT_C = 0x04,
00032     OPT_D = 0x08,
00033     OPT_E = 0x10,
00034     OPT_F = 0x20,*/
00035 };
00036
00037 /** @brief Simulation class to instantiate the whole walkthrough of a Simulation
00038 */
00039 class Simulation {
00040 private:
00041     /// Lattice object
00042     Lattice lattice;
00043     /// LatticePatch object
00044     LatticePatch latticePatch;
00045     /// current time of the simulation
00046     sunrealtype t;
00047     /// char for checking simulation flags
00048     unsigned char statusFlags;
00049
00050 public:
00051     /// IC Setter object
00052     ICSetter icSettings;
00053     /// Output Manager object
00054     OutputManager outputManager;
00055     /// Pointer to CNode memory object - public to avoid cross library errors
00056     void *cnode_mem;
00057     /// constructor function for the creation of the cartesian communicator
00058     Simulation(const int nx, const int ny, const int nz, const int StencilOrder,
00059               const bool periodicity);
00060     /// destructor function freeing CNode memory and Sundials context
00061     ~Simulation();
00062     /// Reference to the cartesian communicator of the lattice -> for debugging
00063     MPI_Comm *get_cart_comm() { return &lattice.comm; };
00064     /// function to set discrete dimensions of the lattice
00065     void setDiscreteDimensionsOfLattice(const sunindextype _tot_nx,
00066                                         const sunindextype _tot_ny, const sunindextype _tot_nz);
00067     /// function to set physical dimensions of the lattice
00068     void setPhysicalDimensionsOfLattice(const sunrealtype lx, const sunrealtype ly,
00069                                         const sunrealtype lz);
00070     /// function to initialize the Patchwork
00071     void initializePatchwork(const int nx, const int ny, const int nz);
00072     /// function to initialize the CVODE object with all requirements
00073     void initializeCVODEobject(const sunrealtype reltol,
00074                                const sunrealtype abstol);
00075     /// function to start the simulation for time iteration
00076     void start();
00077     /// functions to set the initial field configuration onto the lattice
00078     void setInitialConditions();
00079     /// functions to add initial periodic field configurations
00080     void addInitialConditions(const int xm, const int ym, const int zm = 0);
00081     /// function to add a periodic IC Layer in one dimension
00082     void addPeriodicICLayerInX();
00083     /// function to add periodic IC Layers in two dimensions
00084     void addPeriodicICLayerInXY();
00085     /// function to advance solution in time with CVODE
00086     void advanceToTime(const sunrealtype &stEnd);
00087     /// function to generate Output of the whole field at a given time
00088     void outAllFieldData(const int &state);
00089     /// function to check that a flag has been set and if not print an error
00090     // message and cause an abort on all ranks
00091     void checkFlag(unsigned int flag) const;
00092     /// function to check that if flag has not been set and if print an error
00093     // message and cause an abort on all ranks
00094     void checkNoFlag(unsigned int flag) const;
00095 };
00096
00097 // End of Includeguard
00098 #endif

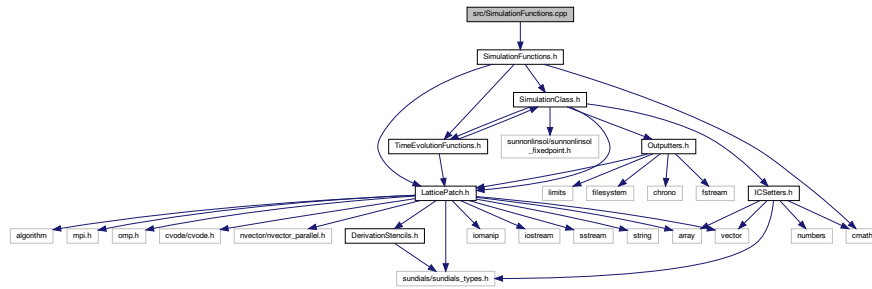
```

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)
- void [Sim1D](#) (const array< sunrealtype, 2 > CNodeTol, const int StencilOrder, const sunrealtype phys_dim, const sunindextype disc_dim, const bool periodic, int *interactions, const sunrealtype endTime, const int numberOfSteps, const string outputDirectory, const int outputStep, const vector< [planewave](#) > &planes, const vector< [gaussian1D](#) > &gaussians)
complete 1D Simulation function
- void [Sim2D](#) (const array< sunrealtype, 2 > CNodeTol, int const StencilOrder, const array< sunrealtype, 2 > phys_dims, const array< sunindextype, 2 > disc_dims, const array< int, 2 > patches, const bool periodic, int *interactions, const sunrealtype endTime, const int numberOfSteps, const string outputDirectory, const int outputStep, const vector< [planewave](#) > &planes, const vector< [gaussian2D](#) > &gaussians)
complete 2D Simulation function
- void [Sim3D](#) (const array< sunrealtype, 2 > CNodeTol, const int StencilOrder, const array< sunrealtype, 3 > phys_dims, const array< sunindextype, 3 > disc_dims, const array< int, 3 > patches, const bool periodic, int *interactions, const sunrealtype endTime, const int numberOfSteps, const string outputDirectory, const int outputStep, const vector< [planewave](#) > &planes, const 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 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 string outputDirectory,
    const int outputStep,
    const vector< planewave > & planes,
    const vector< gaussian1D > & gaussians )
```

complete 1D [Simulation](#) function

Conduct the complete 1D simulation process

Definition at line 23 of file [SimulationFunctions.cpp](#).

```
00029     {
00030
00031     // MPI data
00032     int myPrc = 0, nprc = 0;
00033     MPI_Comm_size(MPI_COMM_WORLD, &nprc);
00034     MPI_Comm_rank(MPI_COMM_WORLD, &myPrc);
00035
00036     // Check feasibility of the patchwork decomposition
00037     if (myPrc == 0) {
00038         if (disc_dim % nprc != 0) {
00039             errorKill("The number of lattice points must be "
00040                     "divisible by the number of processes.");
00041         }
00042     }
00043
00044     // Initialize the simulation, set up the cartesian communicator
00045     array<int, 3> patches = {nprc, 1, 1};
00046     Simulation sim(patches[0], patches[1], patches[2], StencilOrder, periodic);
00047
00048     // Configure the patchwork
00049     sim.setPhysicalDimensionsOfLattice(phys_dim,1,1);
00050     sim.setDiscreteDimensionsOfLattice(disc_dim,1,1);
00051     sim.initializePatchwork(patches[0], patches[1], patches[2]);
00052
00053     // Add em-waves
00054     for (const auto gauss : gaussians)
00055         sim.icsettings.addGauss1D(gauss.k, gauss.p, gauss.x0, gauss.phig,
00056                                   gauss.phi);
00057     for (const auto plane : planes)
00058         sim.icsettings.addPlaneWave1D(plane.k, plane.p, plane.phi);
00059
00060     // Check that the patchwork is ready and set the initial conditions
00061     sim.start();
00062     sim.addPeriodicICLayerInX();
00063
00064     // Initialize CVode with abs and rel tolerances
00065     sim.initializeCVODEObject(CVodeTol[0], CVodeTol[1]);
00066
00067     // Configure the time evolution function
00068     TimeEvolution::c = interactions;
00069     TimeEvolution::TimeEvolver = nonlinear1DProp;
00070
00071     // Configure the output
00072     sim.outputManager.generateOutputFolder(outputDirectory);
00073     if (!myPrc) {
00074         cout << "Simulation code: " << sim.outputManager.getSimCode() << endl;
00075     }
00076
00077     // Conduct the propagation in space and time
00078     double ts = MPI_Wtime();
00079     for (int step = 1; step <= numberOfSteps; step++) {
00080         sim.advanceToTime(endTime / numberOfSteps * step);
00081         if (step % outputStep == 0) {
00082             sim.outAllFieldData(step);
00083         }
00084     }
```



```
00084     double tn = MPI_Wtime();
00085     if (!myProc) {
00086         cout << "\rStep " << step << "\t\t" << flush;
00087         timer(ts, tn);
00088     }
00089 }
00090
00091 return;
00092 }
```

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](#), [Simulation::setDiscreteDimensionsOfLattice\(\)](#), [Simulation::setPhysicalDimensions\(\)](#), [Simulation::start\(\)](#), [TimeEvolution::TimeEvolver](#), and [timer\(\)](#).

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

6.24.2.2 Sim2D()

```
void Sim2D (
    const array< sunrealtype, 2 > CNodeTol,
    int const StencilOrder,
    const array< sunrealtype, 2 > phys_dims,
    const array< sunindextype, 2 > disc_dims,
    const array< int, 2 > patches,
    const bool periodic,
    int * interactions,
    const sunrealtype endTime,
    const int numberOfSteps,
    const string outputDirectory,
    const int outputStep,
    const vector< planewave > & planes,
    const vector< gaussian2D > & gaussians )
```

complete 2D [Simulation](#) function

Conduct the complete 2D simulation process

Definition at line 95 of file [SimulationFunctions.cpp](#).

```
00101 {
00102
00103 // MPI data
00104 int myPrc = 0, nprc = 0; // Get process rank and number of processes
00105 MPI_Comm_rank(MPI_COMM_WORLD,
00106               &myPrc); // Return process rank, number \in [1,nprc]
00107 MPI_Comm_size(MPI_COMM_WORLD,
00108               &nprc); // Return number of processes (communicator size)
00109
00110 // Check feasibility of the patchwork decomposition
00111 if (myPrc == 0) {
00112     if (nprc != patches[0] * patches[1]) {
00113         errorKill(
00114             "The number of MPI processes must match the number of patches.");
00115     }
00116 }
00117
00118 // Initialize the simulation, set up the cartesian communicator
00119 Simulation sim(patches[0], patches[1], 1, StencilOrder, periodic);
00120
00121 // Configure the patchwork
00122 sim.setPhysicalDimensionsOfLattice(phys_dims[0],
00123                                   phys_dims[1],
00124                                   1); // spacing of the lattice
00125 sim.setDiscreteDimensionsOfLattice(
00126     disc_dims[0], disc_dims[1], 1); // Spacing equivalence to points
00127 sim.initializePatchwork(patches[0], patches[1], 1);
00128
00129 // Add em-waves
00130 for (const auto gauss : gaussians)
00131     sim.icsettings.addGauss2D(gauss.x0, gauss.axis, gauss.amp, gauss.phip,
00132                              gauss.w0, gauss.zr, gauss.ph0, gauss.phA);
00133 for (const auto plane : planes)
00134     sim.icsettings.addPlaneWave2D(plane.k, plane.p, plane.phi);
00135
00136 // Check that the patchwork is ready and set the initial conditions
00137 sim.start(); // Check if the lattice is set up, set initial field
00138             // configuration
00139 sim.addPeriodicICLayerInXY(); // insure periodicity in propagation directions
00140
00141 // Initialize CNode with rel and abs tolerances
00142 sim.initializeCNodeObject(CNodeTol[0], CNodeTol[1]);
00143
00144 // Configure the time evolution function
00145 TimeEvolution::c = interactions;
00146 TimeEvolution::TimeEvolver = nonlinear2DProp;
00147
00148 // Configure the output
00149 sim.outputManager.generateOutputFolder(outputDirectory);
```

```

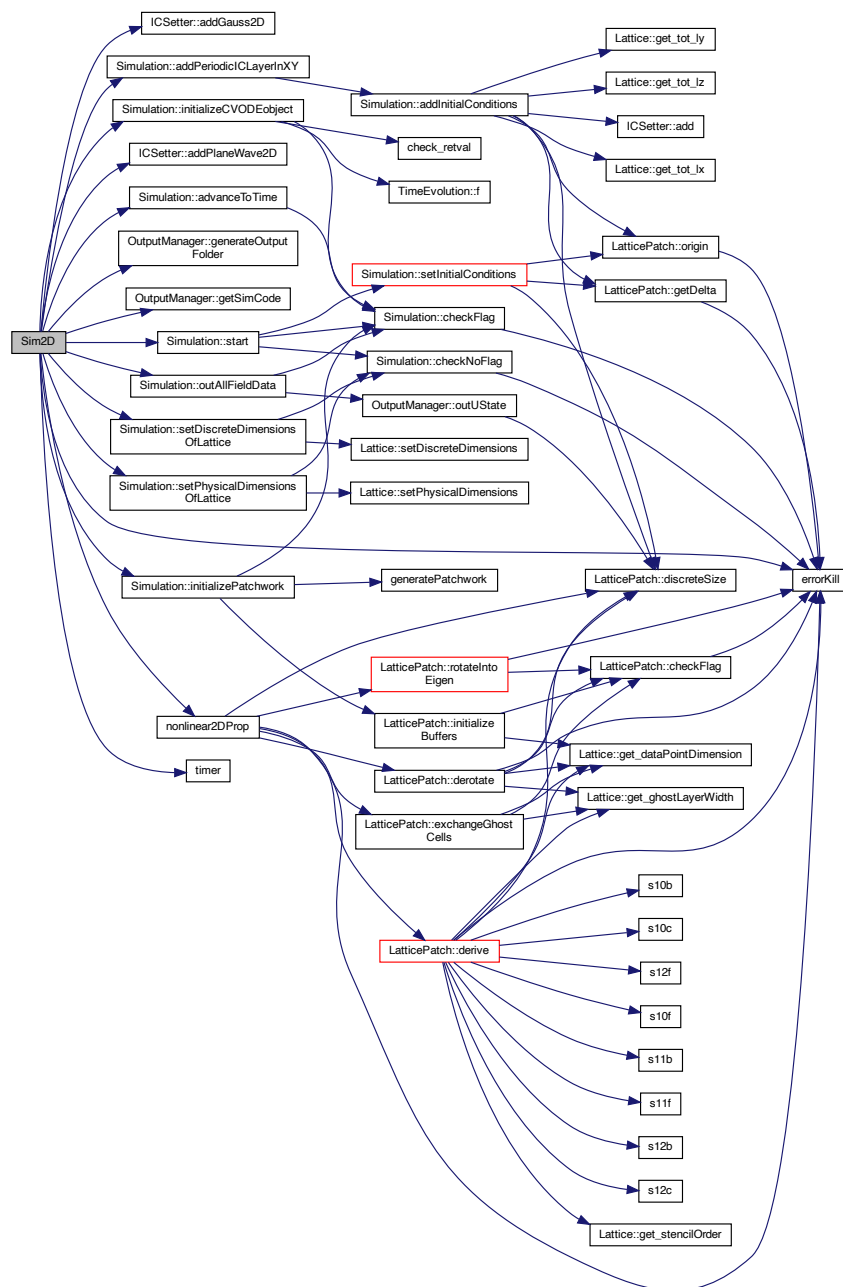
00150  if (!myPrc) {
00151      cout << "Simulation code: " << sim.outputManager.getSimCode() << endl;
00152  }
00153  double ts = MPI_Wtime();
00154
00155  // Conduct the propagation in space and time
00156  for (int step = 1; step <= numberOfSteps; step++) {
00157      sim.advanceToTime(endTime / numberOfSteps * step);
00158      if (step % outputStep == 0) {
00159          sim.outAllFieldData(step);
00160      }
00161      double tn = MPI_Wtime();
00162      if (!myPrc) {
00163          cout << "\rStep " << step << "\t\t" << flush;
00164          timer(ts, tn);
00165      }
00166  }
00167
00168  return;
00169 }

```

References [ICSetter::addGauss2D\(\)](#), [Simulation::addPeriodicCLayerInXY\(\)](#), [ICSetter::addPlaneWave2D\(\)](#), [Simulation::advanceToTime\(\)](#), [TimeEvolution::c](#), [errorKill\(\)](#), [OutputManager::generateOutputFolder\(\)](#), [OutputManager::getSimCode\(\)](#), [Simulation::icsettings](#), [Simulation::initializeCVODEobject\(\)](#), [Simulation::initializePatchwork\(\)](#), [nonlinear2DProp\(\)](#), [Simulation::outAllFieldData\(\)](#), [Simulation::outputManager](#), [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 array< sunrealtype, 2 > CNodeTol,
    const int StencilOrder,
    const array< sunrealtype, 3 > phys_dims,
    const array< sunindextype, 3 > disc_dims,
    const array< int, 3 > patches,
    const bool periodic,
    int * interactions,
    const sunrealtype endTime,
    const int numberOfSteps,
    const string outputDirectory,
    const int outputStep,
    const vector< planewave > & planes,
    const vector< gaussian3D > & gaussians )
```

complete 3D [Simulation](#) function

Conduct the complete 3D simulation process

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

```
00178 {
00179
00180 // MPI data
00181 int myPrc = 0, nprc = 0; // Get process rank and number of process
00182 MPI_Comm_rank(MPI_COMM_WORLD,
00183               &myPrc); // rank of the process inside the world communicator
00184 MPI_Comm_size(MPI_COMM_WORLD,
00185               &nprc); // Size of the communicator is the number of processes
00186
00187 // Check feasibility of the patchwork decomposition
00188 if (myPrc == 0) {
00189     if (nprc != patches[0] * patches[1] * patches[2]) {
00190         errorKill(
00191             "The number of MPI processes must match the number of patches.");
00192     }
00193     if (disc_dims[0] / patches[0] != disc_dims[1] / patches[1] |
00194         disc_dims[0] / patches[0] != disc_dims[2] / patches[2]) {
00195         clog
00196             « "\nWarning: Patches should be cubic in terms of the lattice "
00197             "points for the computational efficiency of larger simulations.\n";
00198     }
00199 }
00200
00201 // Initialize the simulation, set up the cartesian communicator
00202 Simulation sim(patches[0], patches[1], patches[2],
00203               StencilOrder, periodic); // Simulation object with slicing
00204
00205 // Create the SUNContext object associated with the thread of execution
00206 sim.setPhysicalDimensionsOfLattice(phys_dims[0], phys_dims[1],
00207                                   phys_dims[2]); // spacing of the box
00208 sim.setDiscreteDimensionsOfLattice(
00209     disc_dims[0], disc_dims[1],
00210     disc_dims[2]); // Spacing equivalence to points
00211 sim.initializePatchwork(patches[0], patches[1], patches[2]);
00212
00213 // Add em-waves
00214 for (const auto plane : planes)
00215     sim.icsettings.addPlaneWave3D(plane.k, plane.p, plane.phi);
00216 for (const auto gauss : gaussians)
00217     sim.icsettings.addGauss3D(gauss.x0, gauss.axis, gauss.amp, gauss.phip,
00218                               gauss.w0, gauss.zr, gauss.ph0, gauss.phA);
00219
00220 // Check that the patchwork is ready and set the initial conditions
00221 sim.start();
00222
00223 // Initialize CNode with abs and rel tolerances
00224 sim.initializeCNodeObject(CNodeTol[0], CNodeTol[1]);
00225
00226 // Configure the time evolution function
```

```

00227     TimeEvolution::c = interactions;
00228     TimeEvolution::TimeEvolver = nonlinear3DProp;
00229
00230     // Configure the output
00231     sim.outputManager.generateOutputFolder(outputDirectory);
00232     if (!myPrc) {
00233         cout << "Simulation code: " << sim.outputManager.getSimCode() << endl;
00234     }
00235     double ts = MPI_Wtime();
00236
00237     // Conduct the propagation in space and time
00238     for (int step = 1; step <= numberOfSteps; step++) {
00239         sim.advanceToTime(endTime / numberOfSteps * step);
00240         if (step % outputStep == 0) {
00241             sim.outAllFieldData(step);
00242         }
00243         double tn = MPI_Wtime();
00244         if (!myPrc) {
00245             cout << "\rStep " << step << "\t\t" << flush;
00246             timer(ts, tn);
00247         }
00248     }
00249     return;
00250 }

```

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](#), [Simulation::setDiscreteDimensionsOfLattice\(\)](#), [Simulation::setPhysicalDimensionsOfLattice\(\)](#), [Simulation::start\(\)](#), [TimeEvolution::TimeEvolver](#), and [timer\(\)](#).

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

Here is the caller graph for this function:



6.24.2.4 timer()

```
void timer (  
    double & t1,  
    double & t2 )
```

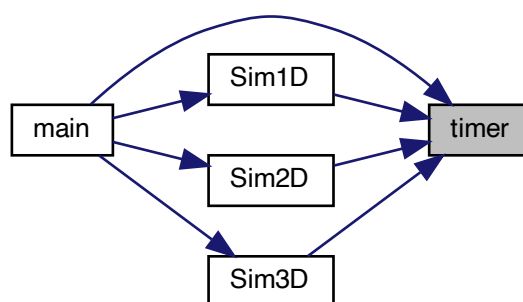
Calculate and print the total simulation time

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

```
00012     {  
00013     printf("Elapsed time:  %fs\n", (t2 - t1));  
00014 }
```

Referenced by [main\(\)](#), [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 using namespace std;
00010
00011 /** Calculate and print the total simulation time */
00012 void timer(double &t1, double &t2) {
00013     printf("Elapsed time: %fs\n", (t2 - t1));
00014 }
00015
00016 // Instantiate and preliminarily initialize the time evolver
00017 // non-const statics to be defined in actual simulation process
00018 int *TimeEvolution::c = nullptr;
00019 void (*TimeEvolution::TimeEvolver)(LatticePatch *, N_Vector, N_Vector,
00020                                     int *) = nonlinear1DProp;
00021
00022 /** Conduct the complete 1D simulation process */
00023 void Sim1D(const array<sunrealtype,2> CNodeTol, const int StencilOrder,
00024            const sunrealtype phys_dim, const sunindextype disc_dim,
00025            const bool periodic, int *interactions,
00026            const sunrealtype endTime, const int numberOfSteps,
00027            const string outputDirectory, const int outputStep,
00028            const vector<planewave> &planes,
00029            const vector<gaussian1D> &gaussians) {
00030
00031     // MPI data
00032     int myPrc = 0, nprc = 0;
00033     MPI_Comm_size(MPI_COMM_WORLD, &nprc);
00034     MPI_Comm_rank(MPI_COMM_WORLD, &myPrc);
00035
00036     // Check feasibility of the patchwork decomposition
00037     if (myPrc == 0) {
00038         if (disc_dim % nprc != 0) {
00039             errorKill("The number of lattice points must be "
00040                     "divisible by the number of processes.");
00041         }
00042     }
00043
00044     // Initialize the simulation, set up the cartesian communicator
00045     array<int, 3> patches = {nprc, 1, 1};
00046     Simulation sim(patches[0], patches[1], patches[2], StencilOrder, periodic);
00047
00048     // Configure the patchwork
00049     sim.setPhysicalDimensionsOfLattice(phys_dim, 1, 1);
00050     sim.setDiscreteDimensionsOfLattice(disc_dim, 1, 1);
00051     sim.initializePatchwork(patches[0], patches[1], patches[2]);
00052
00053     // Add em-waves
00054     for (const auto gauss : gaussians)
00055         sim.icsettings.addGauss1D(gauss.k, gauss.p, gauss.x0, gauss.phig,
00056                                   gauss.phi);
00057     for (const auto plane : planes)
00058         sim.icsettings.addPlaneWave1D(plane.k, plane.p, plane.phi);
00059
00060     // Check that the patchwork is ready and set the initial conditions
00061     sim.start();
00062     sim.addPeriodicICLayerInX();
00063
00064     // Initialize CNode with abs and rel tolerances
00065     sim.initializeCNodeObject(CNodeTol[0], CNodeTol[1]);
00066
00067     // Configure the time evolution function
00068     TimeEvolution::c = interactions;
00069     TimeEvolution::TimeEvolver = nonlinear1DProp;
00070
00071     // Configure the output
00072     sim.outputManager.generateOutputFolder(outputDirectory);
00073     if (!myPrc) {
00074         cout << "Simulation code: " << sim.outputManager.getSimCode() << endl;
00075     }
00076
00077     // Conduct the propagation in space and time
00078     double ts = MPI_Wtime();
00079     for (int step = 1; step <= numberOfSteps; step++) {
00080         sim.advanceToTime(endTime / numberOfSteps * step);
00081         if (step % outputStep == 0) {
00082             sim.outAllFieldData(step);

```

```

00083     }
00084     double tn = MPI_Wtime();
00085     if (!myPrc) {
00086         cout << "\rStep " << step << "\t\t" << flush;
00087         timer(ts, tn);
00088     }
00089 }
00090
00091 return;
00092 }
00093
00094 /** Conduct the complete 2D simulation process */
00095 void Sim2D(const array<sunrealtype,2> CNodeTol, int const StencilOrder,
00096           const array<sunrealtype,2> phys_dims, const array<sunindextype,2> disc_dims,
00097           const array<int,2> patches, const bool periodic, int *interactions,
00098           const sunrealtype endTime, const int numberOfSteps,
00099           const string outputDirectory, const int outputStep,
00100           const vector<planewave> &planes,
00101           const vector<gaussian2D> &gaussians) {
00102
00103     // MPI data
00104     int myPrc = 0, nprc = 0; // Get process rank and number of processes
00105     MPI_Comm_rank(MPI_COMM_WORLD,
00106                  &myPrc); // Return process rank, number \in [1,nprc]
00107     MPI_Comm_size(MPI_COMM_WORLD,
00108                  &nprc); // Return number of processes (communicator size)
00109
00110     // Check feasibility of the patchwork decomposition
00111     if (myPrc == 0) {
00112         if (nprc != patches[0] * patches[1]) {
00113             errorKill(
00114                 "The number of MPI processes must match the number of patches.");
00115         }
00116     }
00117
00118     // Initialize the simulation, set up the cartesian communicator
00119     Simulation sim(patches[0], patches[1], 1, StencilOrder, periodic);
00120
00121     // Configure the patchwork
00122     sim.setPhysicalDimensionsOfLattice(phys_dims[0],
00123                                       phys_dims[1],
00124                                       1); // spacing of the lattice
00125     sim.setDiscreteDimensionsOfLattice(
00126         disc_dims[0], disc_dims[1], 1); // Spacing equivalence to points
00127     sim.initializePatchwork(patches[0], patches[1], 1);
00128
00129     // Add em-waves
00130     for (const auto gauss : gaussians)
00131         sim.icsettings.addGauss2D(gauss.x0, gauss.axis, gauss.amp, gauss.phip,
00132                                   gauss.w0, gauss.zr, gauss.ph0, gauss.phA);
00133     for (const auto plane : planes)
00134         sim.icsettings.addPlaneWave2D(plane.k, plane.p, plane.phi);
00135
00136     // Check that the patchwork is ready and set the initial conditions
00137     sim.start(); // Check if the lattice is set up, set initial field
00138                 // configuration
00139     sim.addPeriodicICLayerInXY(); // insure periodicity in propagation directions
00140
00141     // Initialize CNode with rel and abs tolerances
00142     sim.initializeCNodeObject(CNodeTol[0], CNodeTol[1]);
00143
00144     // Configure the time evolution function
00145     TimeEvolution::c = interactions;
00146     TimeEvolution::TimeEvolver = nonlinear2DProp;
00147
00148     // Configure the output
00149     sim.outputManager.generateOutputFolder(outputDirectory);
00150     if (!myPrc) {
00151         cout << "Simulation code: " << sim.outputManager.getSimCode() << endl;
00152     }
00153     double ts = MPI_Wtime();
00154
00155     // Conduct the propagation in space and time
00156     for (int step = 1; step <= numberOfSteps; step++) {
00157         sim.advanceToTime(endTime / numberOfSteps * step);
00158         if (step % outputStep == 0) {
00159             sim.outAllFieldData(step);
00160         }
00161         double tn = MPI_Wtime();
00162         if (!myPrc) {
00163             cout << "\rStep " << step << "\t\t" << flush;
00164             timer(ts, tn);
00165         }
00166     }
00167
00168     return;
00169 }

```

```

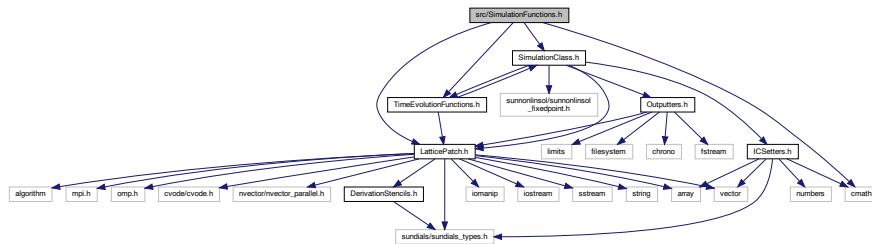
00170
00171 /** Conduct the complete 3D simulation process */
00172 void Sim3D(const array<sunrealttype,2> CNodeTol, const int StencilOrder,
00173           const array<sunrealttype,3> phys_dims,
00174           const array<sunindextype,3> disc_dims, const array<int,3> patches,
00175           const bool periodic, int *interactions, const sunrealttype endTime,
00176           const int numberOfSteps, const string outputDirectory,
00177           const int outputStep, const vector<planewave> &planes,
00178           const vector<gaussian3D> &gaussians) {
00179
00180     // MPI data
00181     int myPrc = 0, nprc = 0; // Get process rank and number of process
00182     MPI_Comm_rank(MPI_COMM_WORLD,
00183                  &myPrc); // rank of the process inside the world communicator
00184     MPI_Comm_size(MPI_COMM_WORLD,
00185                  &nprc); // Size of the communicator is the number of processes
00186
00187     // Check feasibility of the patchwork decomposition
00188     if (myPrc == 0) {
00189         if (nprc != patches[0] * patches[1] * patches[2]) {
00190             errorKill(
00191                 "The number of MPI processes must match the number of patches.");
00192         }
00193         if (disc_dims[0] / patches[0] != disc_dims[1] / patches[1] |
00194             disc_dims[0] / patches[0] != disc_dims[2] / patches[2]) {
00195             clog
00196                 « "\nWarning: Patches should be cubic in terms of the lattice "
00197                 "points for the computational efficiency of larger simulations.\n";
00198         }
00199     }
00200
00201     // Initialize the simulation, set up the cartesian communicator
00202     Simulation sim(patches[0], patches[1], patches[2],
00203                  StencilOrder, periodic); // Simulation object with slicing
00204
00205     // Create the SUNContext object associated with the thread of execution
00206     sim.setPhysicalDimensionsOfLattice(phys_dims[0], phys_dims[1],
00207                                       phys_dims[2]); // spacing of the box
00208     sim.setDiscreteDimensionsOfLattice(
00209         disc_dims[0], disc_dims[1],
00210         disc_dims[2]); // Spacing equivalence to points
00211     sim.initializePatchwork(patches[0], patches[1], patches[2]);
00212
00213     // Add em-waves
00214     for (const auto plane : planes)
00215         sim.icsettings.addPlaneWave3D(plane.k, plane.p, plane.phi);
00216     for (const auto gauss : gaussians)
00217         sim.icsettings.addGauss3D(gauss.x0, gauss.axis, gauss.amp, gauss.phip,
00218                                   gauss.w0, gauss.zr, gauss.ph0, gauss.phA);
00219
00220     // Check that the patchwork is ready and set the initial conditions
00221     sim.start();
00222
00223     // Initialize CNode with abs and rel tolerances
00224     sim.initializeCNodeObject(CNodeTol[0], CNodeTol[1]);
00225
00226     // Configure the time evolution function
00227     TimeEvolution::c = interactions;
00228     TimeEvolution::TimeEvolver = nonlinear3DProp;
00229
00230     // Configure the output
00231     sim.outputManager.generateOutputFolder(outputDirectory);
00232     if (!myPrc) {
00233         cout << "Simulation code: " << sim.outputManager.getSimCode() << endl;
00234     }
00235     double ts = MPI_Wtime();
00236
00237     // Conduct the propagation in space and time
00238     for (int step = 1; step <= numberOfSteps; step++) {
00239         sim.advanceToTime(endTime / numberOfSteps * step);
00240         if (step % outputStep == 0) {
00241             sim.outAllFieldData(step);
00242         }
00243         double tn = MPI_Wtime();
00244         if (!myPrc) {
00245             cout << "\rStep " << step << "\t\t" << flush;
00246             timer(ts, tn);
00247         }
00248     }
00249     return;
00250 }

```

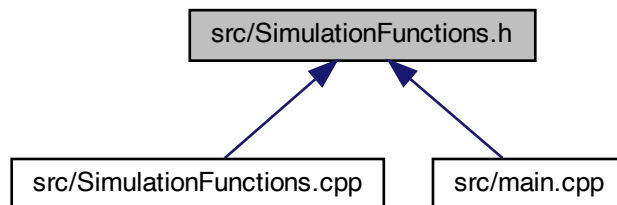
6.26 src/SimulationFunctions.h File Reference

Full simulation functions for 1D, 2D, and 3D used in [main.cpp](#).

```
#include <cmath>
#include "LatticePatch.h"
#include "SimulationClass.h"
#include "TimeEvolutionFunctions.h"
Include dependency graph for SimulationFunctions.h:
```



This graph shows which files directly or indirectly include this file:



Data Structures

- struct [planewave](#)
plane wave structure
- struct [gaussian1D](#)
1D Gaussian wave structure
- struct [gaussian2D](#)
2D Gaussian wave structure
- struct [gaussian3D](#)
3D Gaussian wave structure

Functions

- void [Sim1D](#) (const array< sunrealtype, 2 >, const int, const sunrealtype, const sunindextype, const bool, int *, const sunrealtype, const int, const string, const int, const vector< [planewave](#) > &, const vector< [gaussian1D](#) > &)
complete 1D Simulation function
- void [Sim2D](#) (const array< sunrealtype, 2 >, const int, const array< sunrealtype, 2 >, const array< sunindextype, 2 >, const array< int, 2 >, const bool, int *, const sunrealtype, const int, const string, const int, const vector< [planewave](#) > &, const vector< [gaussian2D](#) > &)
complete 2D Simulation function
- void [Sim3D](#) (const array< sunrealtype, 2 >, const int, const array< sunrealtype, 3 >, const array< sunindextype, 3 >, const array< int, 3 >, const bool, int *, const sunrealtype, const int, const string, const int, const vector< [planewave](#) > &, const vector< [gaussian3D](#) > &)
complete 3D Simulation function
- void [timer](#) (double &, double &)

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 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 string outputDirectory,
    const int outputStep,
    const vector< planewave > & planes,
    const vector< gaussian1D > & gaussians )
```

complete 1D [Simulation](#) function

Conduct the complete 1D simulation process

Definition at line 23 of file [SimulationFunctions.cpp](#).

```
00029 {
00030
00031     // MPI data
00032     int myPrc = 0, nprc = 0;
00033     MPI_Comm_size(MPI_COMM_WORLD, &nprc);
00034     MPI_Comm_rank(MPI_COMM_WORLD, &myPrc);
00035
00036     // Check feasibility of the patchwork decomposition
```

```

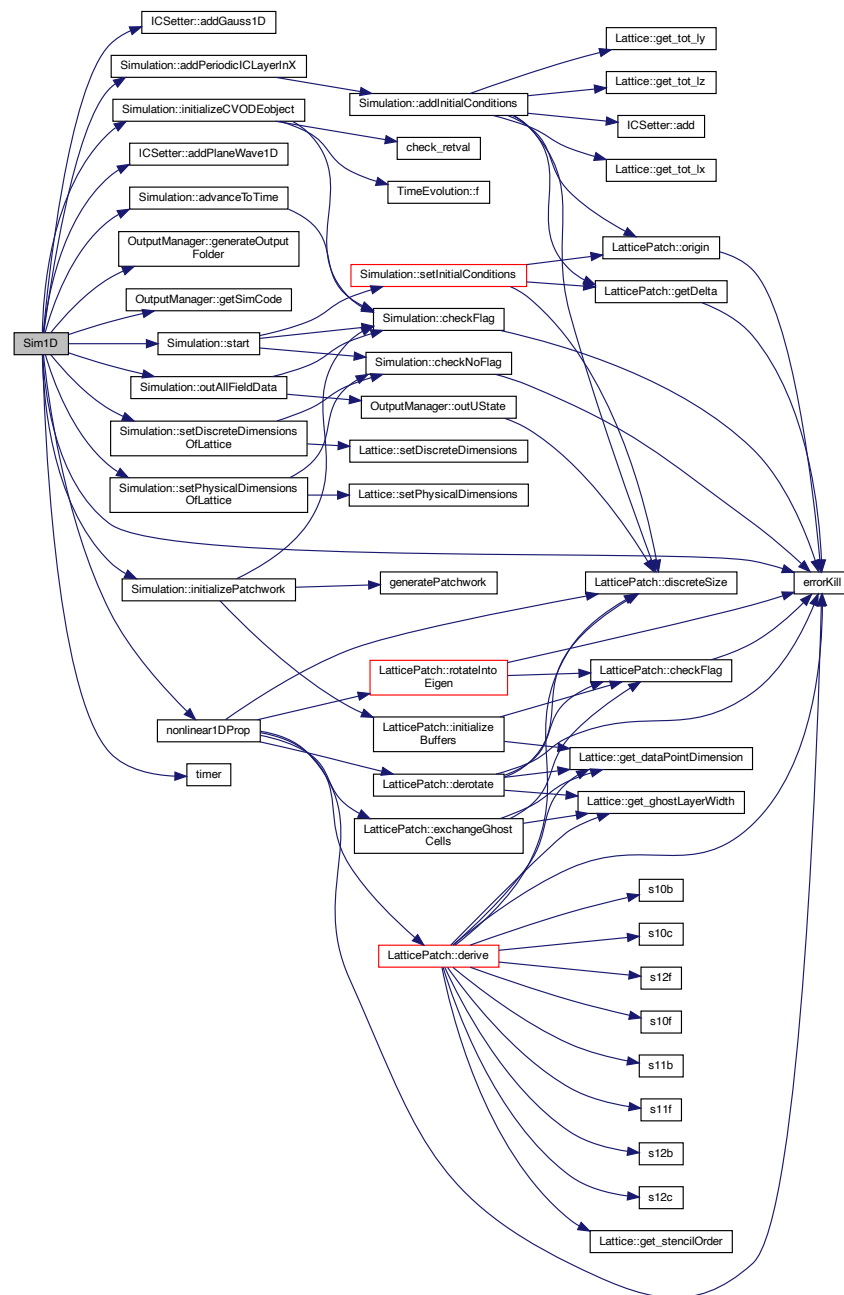
00037     if (myPrc == 0) {
00038         if (disc_dim % nprc != 0) {
00039             errorKill("The number of lattice points must be "
00040                 "divisible by the number of processes.");
00041         }
00042     }
00043
00044     // Initialize the simulation, set up the cartesian communicator
00045     array<int, 3> patches = {nprc, 1, 1};
00046     Simulation sim(patches[0], patches[1], patches[2], StencilOrder, periodic);
00047
00048     // Configure the patchwork
00049     sim.setPhysicalDimensionsOfLattice(phys_dim, 1, 1);
00050     sim.setDiscreteDimensionsOfLattice(disc_dim, 1, 1);
00051     sim.initializePatchwork(patches[0], patches[1], patches[2]);
00052
00053     // Add em-waves
00054     for (const auto gauss : gaussians)
00055         sim.icsettings.addGauss1D(gauss.k, gauss.p, gauss.x0, gauss.phig,
00056             gauss.phi);
00057     for (const auto plane : planes)
00058         sim.icsettings.addPlaneWave1D(plane.k, plane.p, plane.phi);
00059
00060     // Check that the patchwork is ready and set the initial conditions
00061     sim.start();
00062     sim.addPeriodicICLayerInX();
00063
00064     // Initialize CNode with abs and rel tolerances
00065     sim.initializeCNodeObject(CNodeTol[0], CNodeTol[1]);
00066
00067     // Configure the time evolution function
00068     TimeEvolution::c = interactions;
00069     TimeEvolution::TimeEvolver = nonlinear1DProp;
00070
00071     // Configure the output
00072     sim.outputManager.generateOutputFolder(outputDirectory);
00073     if (!myPrc) {
00074         cout << "Simulation code: " << sim.outputManager.getSimCode() << endl;
00075     }
00076
00077     // Conduct the propagation in space and time
00078     double ts = MPI_Wtime();
00079     for (int step = 1; step <= numberOfSteps; step++) {
00080         sim.advanceToTime(endTime / numberOfSteps * step);
00081         if (step % outputStep == 0) {
00082             sim.outAllFieldData(step);
00083         }
00084         double tn = MPI_Wtime();
00085         if (!myPrc) {
00086             cout << "\rStep " << step << "\t\t" << flush;
00087             timer(ts, tn);
00088         }
00089     }
00090
00091     return;
00092 }

```

References [ICSetter::addGauss1D\(\)](#), [Simulation::addPeriodicICLayerInX\(\)](#), [ICSetter::addPlaneWave1D\(\)](#), [Simulation::advanceToTime\(\)](#), [TimeEvolution::c](#), [errorKill\(\)](#), [OutputManager::generateOutputFolder\(\)](#), [OutputManager::getSimCode\(\)](#), [Simulation::icsettings](#), [Simulation::initializeCNodeObject\(\)](#), [Simulation::initializePatchwork\(\)](#), [nonlinear1DProp\(\)](#), [Simulation::outAllFieldData\(\)](#), [Simulation::outputManager](#), [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 array< sunrealtype, 2 > CNodeTol,
    int const StencilOrder,
    const array< sunrealtype, 2 > phys_dims,
    const array< sunindextype, 2 > disc_dims,
    const array< int, 2 > patches,
    const bool periodic,
    int * interactions,
    const sunrealtype endTime,
    const int numberOfSteps,
    const string outputDirectory,
    const int outputStep,
    const vector< planewave > & planes,
    const vector< gaussian2D > & gaussians )
```

complete 2D [Simulation](#) function

Conduct the complete 2D simulation process

Definition at line 95 of file [SimulationFunctions.cpp](#).

```
00101 {
00102
00103 // MPI data
00104 int myPrc = 0, nprc = 0; // Get process rank and number of processes
00105 MPI_Comm_rank(MPI_COMM_WORLD,
00106               &myPrc); // Return process rank, number \in [1,nprc]
00107 MPI_Comm_size(MPI_COMM_WORLD,
00108               &nprc); // Return number of processes (communicator size)
00109
00110 // Check feasibility of the patchwork decomposition
00111 if (myPrc == 0) {
00112     if (nprc != patches[0] * patches[1]) {
00113         errorKill(
00114             "The number of MPI processes must match the number of patches.");
00115     }
00116 }
00117
00118 // Initialize the simulation, set up the cartesian communicator
00119 Simulation sim(patches[0], patches[1], 1, StencilOrder, periodic);
00120
00121 // Configure the patchwork
00122 sim.setPhysicalDimensionsOfLattice(phys_dims[0],
00123                                   phys_dims[1],
00124                                   1); // spacing of the lattice
00125 sim.setDiscreteDimensionsOfLattice(
00126     disc_dims[0], disc_dims[1], 1); // Spacing equivalence to points
00127 sim.initializePatchwork(patches[0], patches[1], 1);
00128
00129 // Add em-waves
00130 for (const auto gauss : gaussians)
00131     sim.icsettings.addGauss2D(gauss.x0, gauss.axis, gauss.amp, gauss.phip,
00132                              gauss.w0, gauss.zr, gauss.ph0, gauss.phA);
00133 for (const auto plane : planes)
00134     sim.icsettings.addPlaneWave2D(plane.k, plane.p, plane.phi);
00135
00136 // Check that the patchwork is ready and set the initial conditions
00137 sim.start(); // Check if the lattice is set up, set initial field
00138             // configuration
00139 sim.addPeriodicICLayerInXY(); // insure periodicity in propagation directions
00140
00141 // Initialize CNode with rel and abs tolerances
00142 sim.initializeCNodeObject(CNodeTol[0], CNodeTol[1]);
00143
00144 // Configure the time evolution function
00145 TimeEvolution::c = interactions;
00146 TimeEvolution::TimeEvolver = nonlinear2DProp;
00147
00148 // Configure the output
00149 sim.outputManager.generateOutputFolder(outputDirectory);
```

```

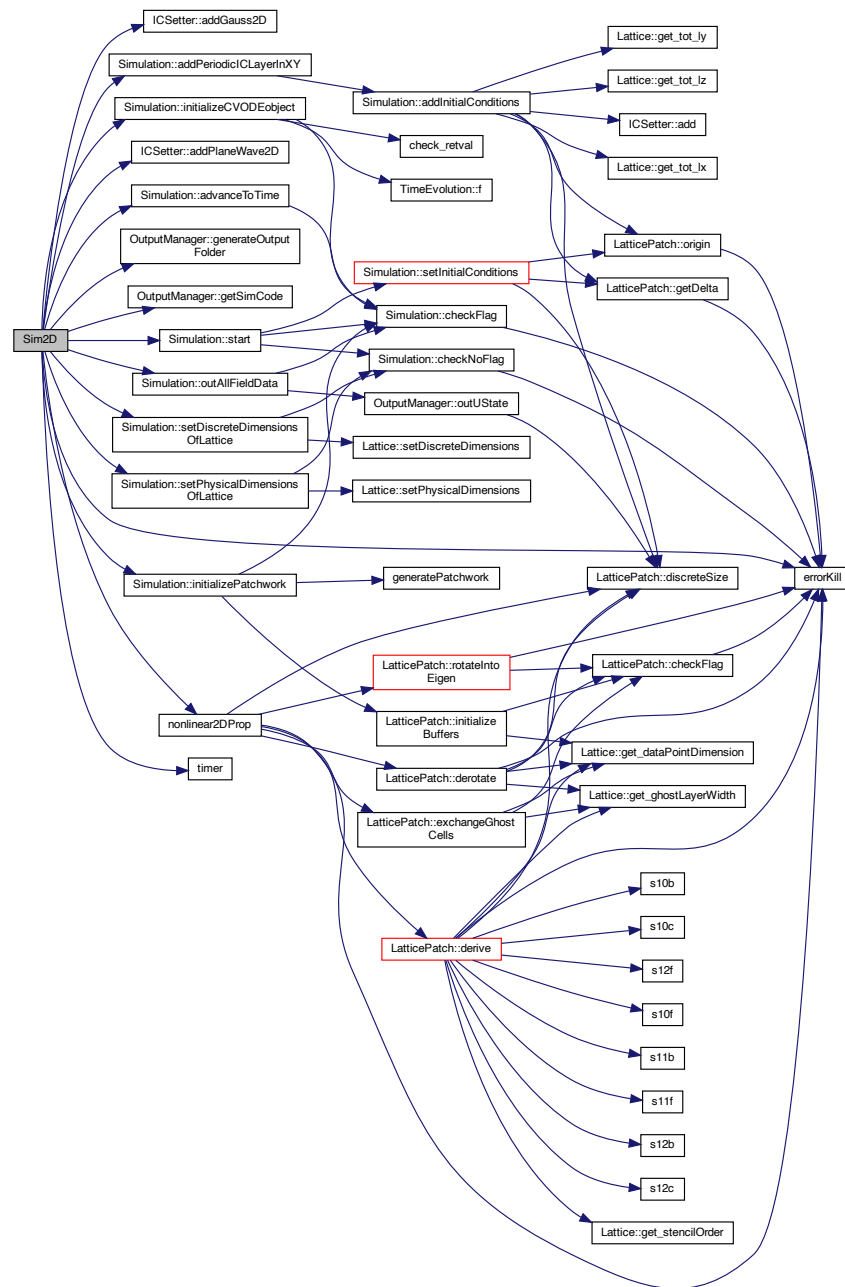
00150  if (!myPrc) {
00151      cout << "Simulation code: " << sim.outputManager.getSimCode() << endl;
00152  }
00153  double ts = MPI_Wtime();
00154
00155  // Conduct the propagation in space and time
00156  for (int step = 1; step <= numberOfSteps; step++) {
00157      sim.advanceToTime(endTime / numberOfSteps * step);
00158      if (step % outputStep == 0) {
00159          sim.outAllFieldData(step);
00160      }
00161      double tn = MPI_Wtime();
00162      if (!myPrc) {
00163          cout << "\rStep " << step << "\t\t" << flush;
00164          timer(ts, tn);
00165      }
00166  }
00167
00168  return;
00169 }

```

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

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

Here is the caller graph for this function:



```
graph LR; main --> Sim2D
```

6.26.2.3 Sim3D()

```
void Sim3D (
    const array< sunrealtype, 2 > CNodeTol,
    const int StencilOrder,
    const array< sunrealtype, 3 > phys_dims,
    const array< sunindextype, 3 > disc_dims,
    const array< int, 3 > patches,
    const bool periodic,
    int * interactions,
    const sunrealtype endTime,
    const int numberOfSteps,
    const string outputDirectory,
    const int outputStep,
    const vector< planewave > & planes,
    const vector< gaussian3D > & gaussians )
```

complete 3D [Simulation](#) function

Conduct the complete 3D simulation process

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

```
00178 {
00179
00180 // MPI data
00181 int myPrc = 0, nprc = 0; // Get process rank and number of process
00182 MPI_Comm_rank(MPI_COMM_WORLD,
00183               &myPrc); // rank of the process inside the world communicator
00184 MPI_Comm_size(MPI_COMM_WORLD,
00185               &nprc); // Size of the communicator is the number of processes
00186
00187 // Check feasibility of the patchwork decomposition
00188 if (myPrc == 0) {
00189     if (nprc != patches[0] * patches[1] * patches[2]) {
00190         errorKill(
00191             "The number of MPI processes must match the number of patches.");
00192     }
00193     if (disc_dims[0] / patches[0] != disc_dims[1] / patches[1] |
00194         disc_dims[0] / patches[0] != disc_dims[2] / patches[2]) {
00195         clog
00196             « "\nWarning: Patches should be cubic in terms of the lattice "
00197             "points for the computational efficiency of larger simulations.\n";
00198     }
00199 }
00200
00201 // Initialize the simulation, set up the cartesian communicator
00202 Simulation sim(patches[0], patches[1], patches[2],
00203               StencilOrder, periodic); // Simulation object with slicing
00204
00205 // Create the SUNContext object associated with the thread of execution
00206 sim.setPhysicalDimensionsOfLattice(phys_dims[0], phys_dims[1],
00207                                   phys_dims[2]); // spacing of the box
00208 sim.setDiscreteDimensionsOfLattice(
00209     disc_dims[0], disc_dims[1],
00210     disc_dims[2]); // Spacing equivalence to points
00211 sim.initializePatchwork(patches[0], patches[1], patches[2]);
00212
00213 // Add em-waves
00214 for (const auto plane : planes)
00215     sim.icsettings.addPlaneWave3D(plane.k, plane.p, plane.phi);
00216 for (const auto gauss : gaussians)
00217     sim.icsettings.addGauss3D(gauss.x0, gauss.axis, gauss.amp, gauss.phip,
00218                               gauss.w0, gauss.zr, gauss.ph0, gauss.phA);
00219
00220 // Check that the patchwork is ready and set the initial conditions
00221 sim.start();
00222
00223 // Initialize CNode with abs and rel tolerances
00224 sim.initializeCNodeObject(CNodeTol[0], CNodeTol[1]);
00225
00226 // Configure the time evolution function
```

```

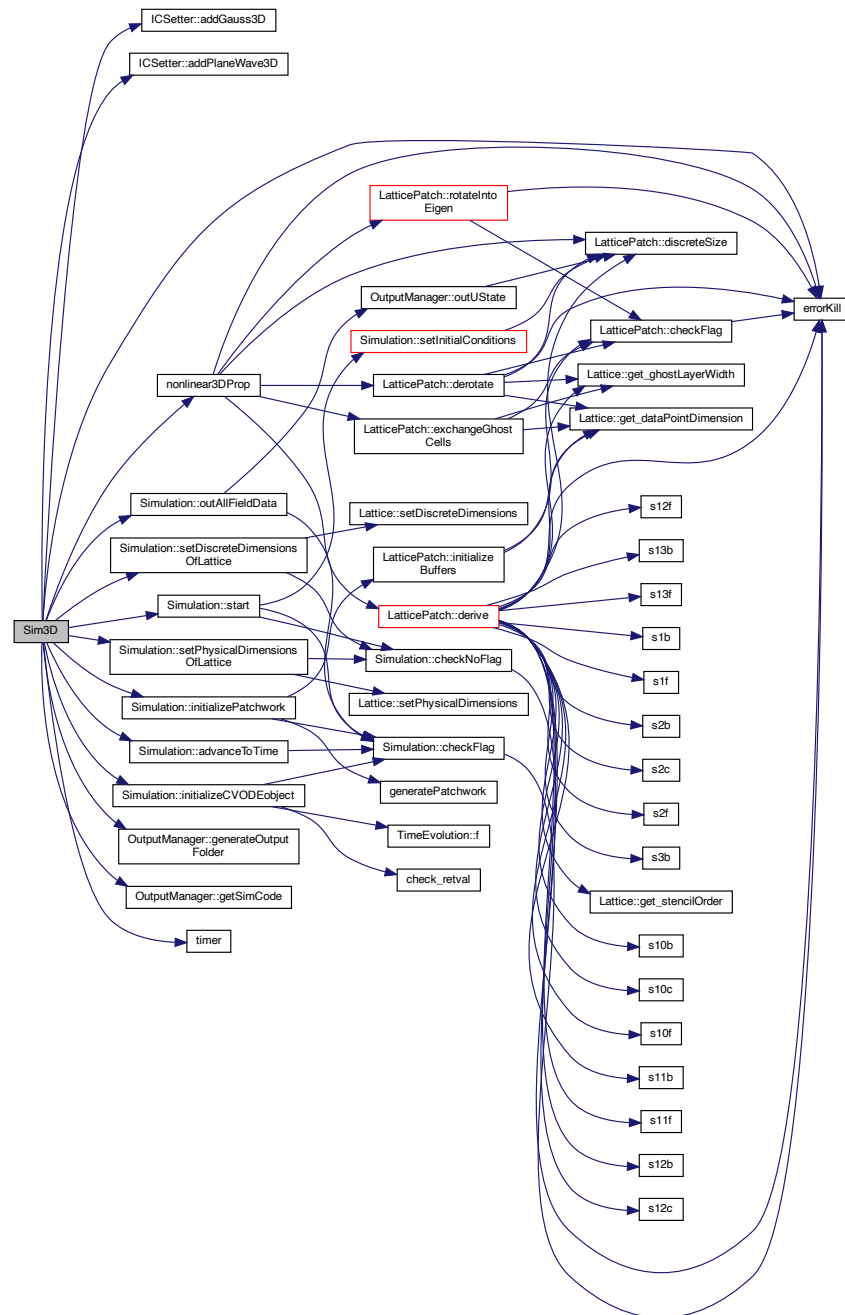
00227   TimeEvolution::c = interactions;
00228   TimeEvolution::TimeEvolver = nonlinear3DProp;
00229
00230   // Configure the output
00231   sim.outputManager.generateOutputFolder(outputDirectory);
00232   if (!myPrc) {
00233       cout << "Simulation code: " << sim.outputManager.getSimCode() << endl;
00234   }
00235   double ts = MPI_Wtime();
00236
00237   // Conduct the propagation in space and time
00238   for (int step = 1; step <= numberOfSteps; step++) {
00239       sim.advanceToTime(endTime / numberOfSteps * step);
00240       if (step % outputStep == 0) {
00241           sim.outAllFieldData(step);
00242       }
00243       double tn = MPI_Wtime();
00244       if (!myPrc) {
00245           cout << "\rStep " << step << "\t\t" << flush;
00246           timer(ts, tn);
00247       }
00248   }
00249   return;
00250 }

```

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](#), [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 )
```

MPI timer function

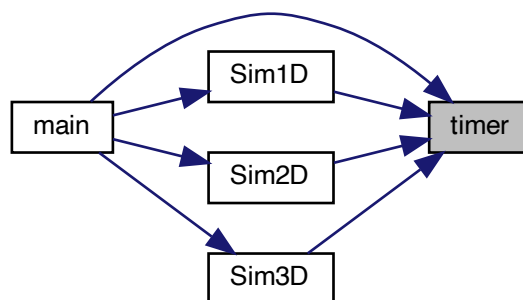
Calculate and print the total simulation time

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

```
00012     {  
00013     printf("Elapsed time:  %fs\n", (t2 - t1));  
00014 }
```

Referenced by [main\(\)](#), [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 // math
00007 #include <cmath>
00008 // #include <mathimf.h>
00009
00010 // project subfile headers
00011 #include "LatticePatch.h"
00012 #include "SimulationClass.h"
00013 #include "TimeEvolutionFunctions.h"
00014
00015 /***** EM-wave structures *****/
00016
00017 /// plane wave structure
00018 struct planewave {
00019     vector<sunrealtype> k; /**< wavevector (normalized to \f$ 1/\lambda \f$) */
00020     vector<sunrealtype> p; /**< amplitde & polarization vector */
00021     vector<sunrealtype> phi; /**< phase shift */
00022 };
00023
00024 /// 1D Gaussian wave structure
00025 struct gaussian1D {
00026     vector<sunrealtype> k; /**< wavevector (normalized to \f$ 1/\lambda \f$) */
00027     vector<sunrealtype> p; /**< amplitude & polarization vector */
00028     vector<sunrealtype> x0; /**< shift from origin */
00029     sunrealtype phig; /**< width */
00030     vector<sunrealtype> phi; /**< phase shift */
00031 };
00032
00033 /// 2D Gaussian wave structure
00034 struct gaussian2D {
00035     vector<sunrealtype> x0; /**< center */
00036     vector<sunrealtype> axis; /**< direction to center */
00037     sunrealtype amp; /**< amplitude */
00038     sunrealtype phip; /**< polarization rotation */
00039     sunrealtype w0; /**< taille */
00040     sunrealtype zr; /**< Rayleigh length */
00041     sunrealtype ph0; /**< beam center */
00042     sunrealtype phA; /**< beam length */
00043 };
00044
00045 /// 3D Gaussian wave structure
00046 struct gaussian3D {
00047     vector<sunrealtype> x0; /**< center */
00048     vector<sunrealtype> axis; /**< direction to center */
00049     sunrealtype amp; /**< amplitude */
00050     sunrealtype phip; /**< polarization rotation */
00051     sunrealtype w0; /**< taille */
00052     sunrealtype zr; /**< Rayleigh length */
00053     sunrealtype ph0; /**< beam center */
00054     sunrealtype phA; /**< beam length */
00055 };
00056
00057 /***** simulation function declarations *****/
00058
00059 /// complete 1D Simulation function
00060 void Sim1D(const array<sunrealtype,2>, const int, const sunrealtype,
00061           const sunindextype, const bool, int *, const sunrealtype, const int,
00062           const string, const int, const vector<planewave> &,
00063           const vector<gaussian1D> &);
00064 /// complete 2D Simulation function
00065 void Sim2D(const array<sunrealtype,2>, const int, const array<sunrealtype,2>,
00066           const array<sunindextype,2>, const array<int,2>, const bool, int *,
00067           const sunrealtype, const int, const string, const int,
00068           const vector<planewave> &, const vector<gaussian2D> &);
00069 /// complete 3D Simulation function
00070 void Sim3D(const array<sunrealtype,2>, const int, const array<sunrealtype,3>,
00071           const array<sunindextype,3>, const array<int,3>, const bool, int *,
00072           const sunrealtype, const int, const string, const int,
00073           const vector<planewave> &, const vector<gaussian3D> &);
00074
00075 /** MPI timer function */
00076 void timer(double &, double &);
```

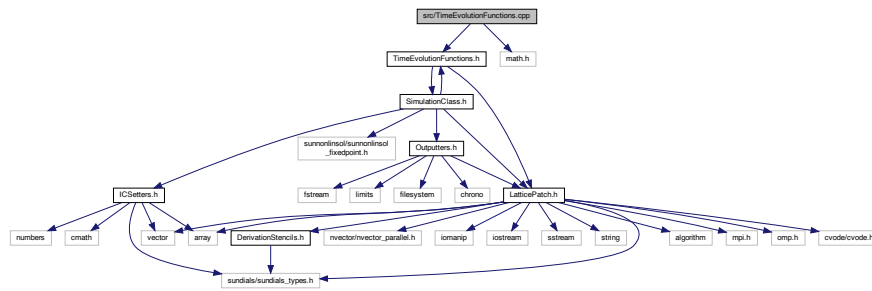

6.28 src/TimeEvolutionFunctions.cpp File Reference

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 <math.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

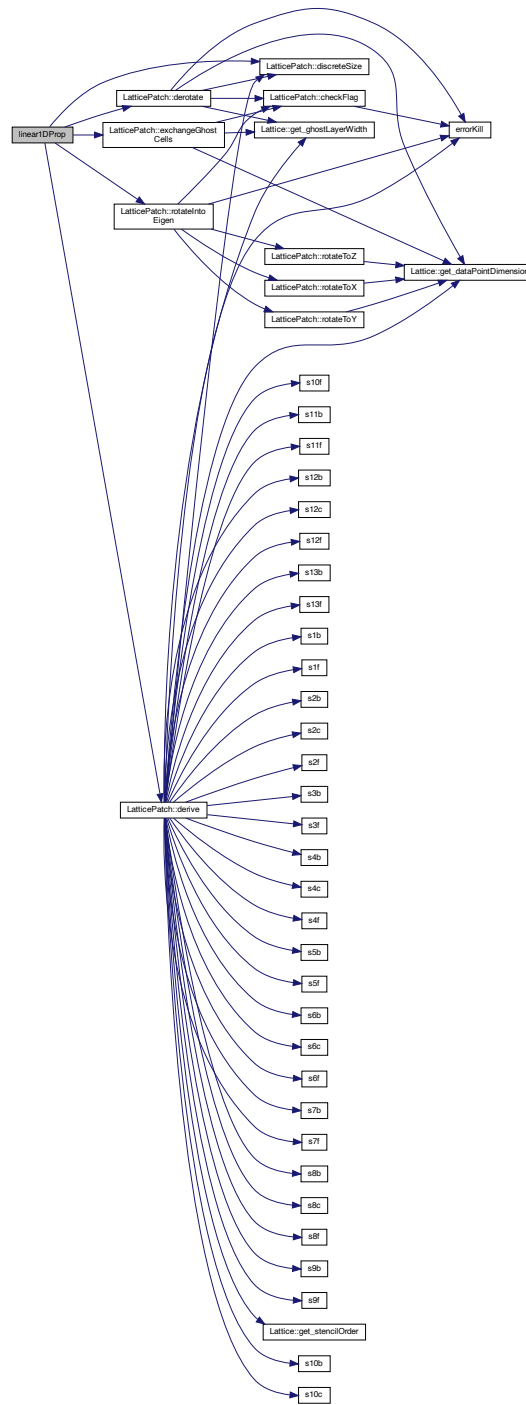
Maxwell propagation function for 1D – only for reference.

Definition at line 46 of file [TimeEvolutionFunctions.cpp](#).

```
00046 {
00047
00048 // pointers to temporal and spatial derivative data
00049 sunrealtype *duData = data->duData;
00050 sunrealtype *dxData = data->buffData[1 - 1];
00051
00052 // sequence along any dimension:
00053 data->exchangeGhostCells(1); // exchange halos
00054 data->rotateIntoEigen(
00055     1); // -> rotate all data to prepare derivative operation
00056 data->derive(1); // -> perform derivative on it
00057 data->derotate(
00058     1, dxData); // -> derotate derivative data to x-space for further use
00059
00060 int totalNP = data->discreteSize();
00061 int pp = 0;
00062 for (int i = 0; i < totalNP; i++) {
00063     pp = i * 6;
00064     /*
00065     simple vacuum Maxwell equations for spatial derivative only in x-direction
00066     temporal derivative is approximated by spatial derivative according to the
00067     numerical scheme with Jacobi=0 -> no polarization or magnetization terms
00068     */
00069     duData[pp + 0] = 0;
00070     duData[pp + 1] = -dxData[pp + 5];
00071     duData[pp + 2] = dxData[pp + 4];
00072     duData[pp + 3] = 0;
00073     duData[pp + 4] = dxData[pp + 2];
00074     duData[pp + 5] = -dxData[pp + 1];
00075 }
00076 }
```

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

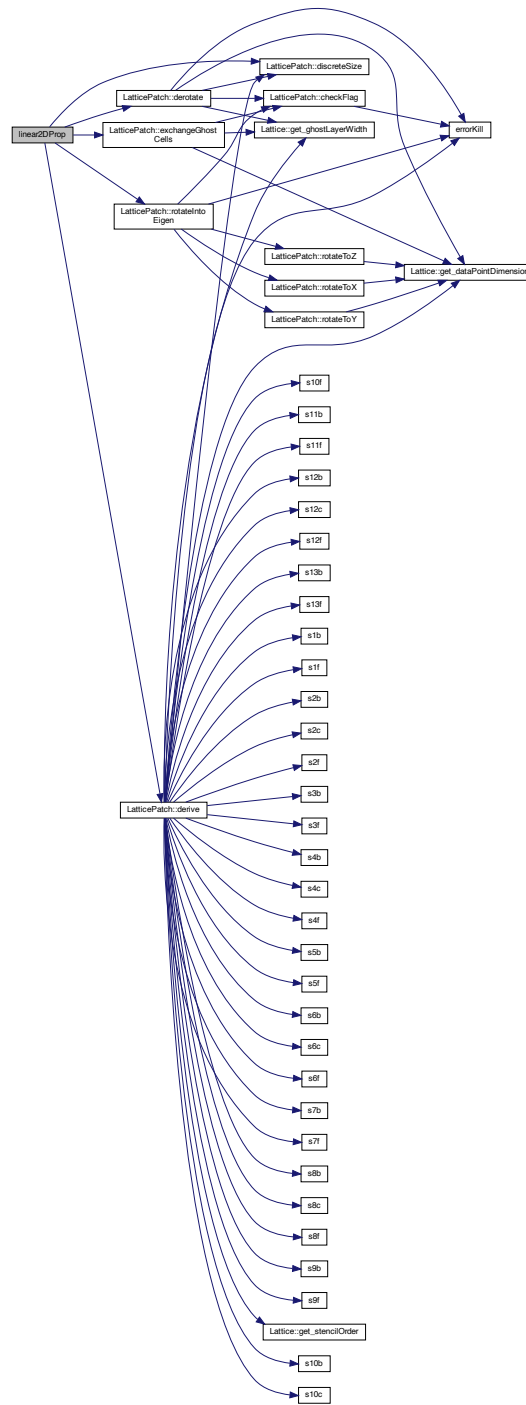
```

00265
00266
00267     sunrealtype *duData = data->duData;
00268     sunrealtype *dxData = data->buffData[1 - 1];
00269     sunrealtype *dyData = data->buffData[2 - 1];
00270
00271     data->exchangeGhostCells(1);
00272     data->rotateIntoEigen(1);
00273     data->derive(1);
00274     data->derotate(1, dxData);
00275     data->exchangeGhostCells(2);
00276     data->rotateIntoEigen(2);
00277     data->derive(2);
00278     data->derotate(2, dyData);
00279
00280     int totalNP = data->discreteSize();
00281     int pp = 0;
00282     for (int i = 0; i < totalNP; i++) {
00283         pp = i * 6;
00284         duData[pp + 0] = dyData[pp + 5];
00285         duData[pp + 1] = -dxData[pp + 5];
00286         duData[pp + 2] = -dyData[pp + 3] + dxData[pp + 4];
00287         duData[pp + 3] = -dyData[pp + 2];
00288         duData[pp + 4] = dxData[pp + 2];
00289         duData[pp + 5] = dyData[pp + 0] - dxData[pp + 1];
00290     }
00291 }

```

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

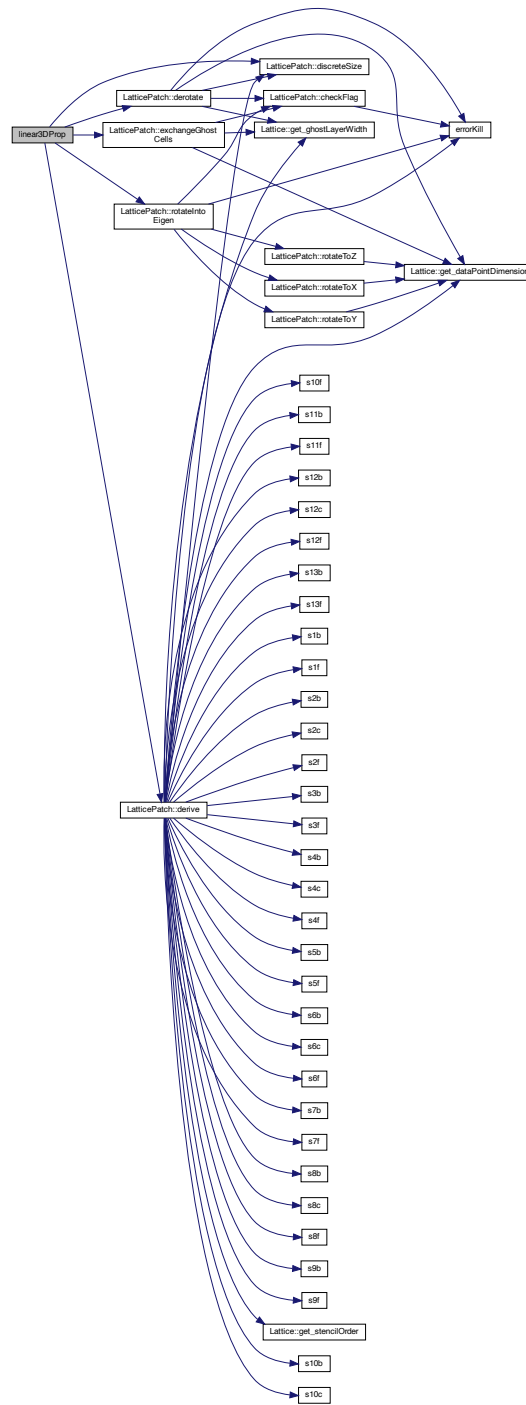
```

00476                                     {
00477
00478     sunrealtype *duData = data->duData;
00479     sunrealtype *dxData = data->buffData[1 - 1];
00480     sunrealtype *dyData = data->buffData[2 - 1];
00481     sunrealtype *dzData = data->buffData[3 - 1];
00482
00483     data->exchangeGhostCells(1);
00484     data->rotateIntoEigen(1);
00485     data->derive(1);
00486     data->derotate(1, dxData);
00487     data->exchangeGhostCells(2);
00488     data->rotateIntoEigen(2);
00489     data->derive(2);
00490     data->derotate(2, dyData);
00491     data->exchangeGhostCells(3);
00492     data->rotateIntoEigen(3);
00493     data->derive(3);
00494     data->derotate(3, dzData);
00495
00496     int totalNP = data->discreteSize();
00497     int pp = 0;
00498     for (int i = 0; i < totalNP; i++) {
00499         pp = i * 6;
00500         duData[pp + 0] = dyData[pp + 5] - dzData[pp + 4];
00501         duData[pp + 1] = dzData[pp + 3] - dxData[pp + 5];
00502         duData[pp + 2] = dxData[pp + 4] - dyData[pp + 3];
00503         duData[pp + 3] = -dyData[pp + 2] + dzData[pp + 1];
00504         duData[pp + 4] = -dzData[pp + 0] + dxData[pp + 2];
00505         duData[pp + 5] = -dxData[pp + 1] + dyData[pp + 0];
00506     }
00507 }

```

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. Calculation of the Jacobi matrix

Definition at line 79 of file [TimeEvolutionFunctions.cpp](#).

```

00079
00080
00081 // pointer to spatial derivative data sufficient, temporal derivative data
00082 // provided with udot
00083 sunrealtype *dxData = data->buffData[1 - 1];
00084
00085 // same sequence as in the linear case
00086 data->exchangeGhostCells(1);
00087 data->rotateIntoEigen(1);
00088 data->derive(1);
00089 data->derotate(1, dxData);
00090
00091 /*
00092 F and G are nonzero in the nonlinear case,
00093 polarization and magnetization contributions in Jacobi matrix style
00094 with derivatives of polarization and magnetization
00095 w.r.t. E- and B-field
00096 */
00097 sunrealtype f = NAN, g = NAN; // em field invariants F, G
00098 sunrealtype lff = NAN, lfg = NAN, lgg = NAN, lg = NAN,
00099 lgg = NAN; // derivatives of Lagrangian w.r.t. field invariants
00100 array<sunrealtype, 36> JMM; // Jacobi matrix
00101 array<sunrealtype, 6> Quad; // array to hold E^2 and B^2 components
00102 array<sunrealtype, 6> h; // holding temporal derivatives of E and B components
00103 // before operating (1+Z)^-1
00104 sunrealtype pseudoDenom = NAN; // needed for inversion of 1+Z
00105 sunrealtype *udata = nullptr,
00106 *dudata = nullptr; // pointers to data and temp. derivative data
00107 udata = NV_DATA_P(u);
00108 dudata = NV_DATA_P(udot);
00109 int totalNP = data->discreteSize(); // number of points in the patch
00110 for (int pp = 0; pp < totalNP * 6;
00111      pp += 6) { // loops through all 6dim points in the patch
00112     // for(int ppB=0;ppB<totalNP*6;ppB+=6*6){
00113     // for(int ppB;ppB<min(totalNP*6,ppB+6*6);ppB+=6){
00114     /// Calculation of the Jacobi matrix
00115     // 1. Calculate F and G
00116     f = 0.5 * ((Quad[0] = udata[pp] * udata[pp]) +
00117               (Quad[1] = udata[pp + 1] * udata[pp + 1]) +
00118               (Quad[2] = udata[pp + 2] * udata[pp + 2]) -
00119               (Quad[3] = udata[pp + 3] * udata[pp + 3]) -
00120               (Quad[4] = udata[pp + 4] * udata[pp + 4]) -
00121               (Quad[5] = udata[pp + 5] * udata[pp + 5]));
00122     g = udata[pp] * udata[pp + 3] + udata[pp + 1] * udata[pp + 4] +
00123         udata[pp + 2] * udata[pp + 5];
00124     // 2. Choose process/expansion order and assign derivative values of L
00125     // w.r.t. F, G
00126     switch (*c) {
00127     case 0:
00128         lff = 0;
00129         lfg = 0;
00130         lfg = 0;
00131         lg = 0;
00132         lgg = 0;
00133         break;
00134     case 2:
00135         lff = 0.000354046449700427580438254 * f * f +
00136             0.000191775160254398272737387 * g * g;
00137         lff = 0.0007080928994008551608765075 * f;
00138         lfg = 0.0003835503205087965454747749 * g;
00139         lg = 0.0003835503205087965454747749 * f * g;
00140         lgg = 0.0003835503205087965454747749 * f;
00141         break;
00142     case 1:
00143         lff = 0.000206527095658582755255648 * f;
00144         lff = 0.000206527095658582755255648;
00145         lfg = 0;
00146         lg = 0.0003614224174025198216973841 * g;
00147         lgg = 0.0003614224174025198216973841;
00148         break;
00149     case 3:
00150         lff = (0.000206527095658582755255648 + 0.000354046449700427580438254 * f) *
00151             f +
00152             0.000191775160254398272737387 * g * g;

```



```

00153     lff = 0.000206527095658582755255648 + 0.0007080928994008551608765075 * f;
00154     lfg = 0.0003835503205087965454747749 * g;
00155     lg = (0.0003614224174025198216973841 +
00156           0.0003835503205087965454747749 * f) *
00157         g;
00158     lgg = 0.0003614224174025198216973841 + 0.0003835503205087965454747749 * f;
00159     break;
00160     default:
00161         errorKill(
00162             "You need to specify a correct order in the weak-field expansion.");
00163     }
00164     // 3. Assign Jacobi components
00165     JMM[0] = lf + lff * Quad[0] +
00166           udata[3 + pp] * (2 * lfg * udata[pp] + lgg * udata[3 + pp]);
00167     JMM[6] =
00168         lff * udata[pp] * udata[1 + pp] + lfg * udata[1 + pp] * udata[3 + pp] +
00169         lfg * udata[pp] * udata[4 + pp] + lgg * udata[3 + pp] * udata[4 + pp];
00170     JMM[7] = lf + lff * Quad[1] +
00171           udata[4 + pp] * (2 * lfg * udata[1 + pp] + lgg * udata[4 + pp]);
00172     JMM[12] =
00173         lff * udata[pp] * udata[2 + pp] + lfg * udata[2 + pp] * udata[3 + pp] +
00174         lfg * udata[pp] * udata[5 + pp] + lgg * udata[3 + pp] * udata[5 + pp];
00175     JMM[13] = lff * udata[1 + pp] * udata[2 + pp] +
00176           lfg * udata[2 + pp] * udata[4 + pp] +
00177           lfg * udata[1 + pp] * udata[5 + pp] +
00178           lgg * udata[4 + pp] * udata[5 + pp];
00179     JMM[14] = lf + lff * Quad[2] +
00180           udata[5 + pp] * (2 * lfg * udata[2 + pp] + lgg * udata[5 + pp]);
00181     JMM[18] = lg + lfg * (Quad[0] - Quad[3 + 0]) +
00182           (-lff + lgg) * udata[pp] * udata[3 + pp];
00183     JMM[19] = -(udata[3 + pp] * (lff * udata[1 + pp] + lfg * udata[4 + pp])) +
00184           udata[pp] * (lfg * udata[1 + pp] + lgg * udata[4 + pp]);
00185     JMM[20] = -(udata[3 + pp] * (lff * udata[2 + pp] + lfg * udata[5 + pp])) +
00186           udata[pp] * (lfg * udata[2 + pp] + lgg * udata[5 + pp]);
00187     JMM[21] = -lf + lgg * Quad[0] +
00188           udata[3 + pp] * (-2 * lfg * udata[pp] + lff * udata[3 + pp]);
00189     JMM[24] = udata[1 + pp] * (lfg * udata[pp] + lgg * udata[3 + pp]) -
00190           (lff * udata[pp] + lfg * udata[3 + pp]) * udata[4 + pp];
00191     JMM[25] = lg + lfg * (Quad[1] - Quad[4 + 0]) +
00192           (-lff + lgg) * udata[1 + pp] * udata[4 + pp];
00193     JMM[26] = -(udata[4 + pp] * (lff * udata[2 + pp] + lfg * udata[5 + pp])) +
00194           udata[1 + pp] * (lfg * udata[2 + pp] + lgg * udata[5 + pp]);
00195     JMM[27] = lgg * udata[pp] * udata[1 + pp] +
00196           lff * udata[3 + pp] * udata[4 + pp] -
00197           lfg * (udata[1 + pp] * udata[3 + pp] + udata[pp] * udata[4 + pp]);
00198     JMM[28] = -lf + lgg * Quad[1] +
00199           udata[4 + pp] * (-2 * lfg * udata[1 + pp] + lff * udata[4 + pp]);
00200     JMM[30] = udata[2 + pp] * (lfg * udata[pp] + lgg * udata[3 + pp]) -
00201           (lff * udata[pp] + lfg * udata[3 + pp]) * udata[5 + pp];
00202     JMM[31] = udata[2 + pp] * (lfg * udata[1 + pp] + lgg * udata[4 + pp]) -
00203           (lff * udata[1 + pp] + lfg * udata[4 + pp]) * udata[5 + pp];
00204     JMM[32] = lg + lfg * (Quad[2] - Quad[5 + 0]) +
00205           (-lff + lgg) * udata[2 + pp] * udata[5 + pp];
00206     JMM[33] = lgg * udata[pp] * udata[2 + pp] +
00207           lff * udata[3 + pp] * udata[5 + pp] -
00208           lfg * (udata[2 + pp] * udata[3 + pp] + udata[pp] * udata[5 + pp]);
00209     JMM[34] =
00210         lgg * udata[1 + pp] * udata[2 + pp] +
00211         lff * udata[4 + pp] * udata[5 + pp] -
00212         lfg * (udata[2 + pp] * udata[4 + pp] + udata[1 + pp] * udata[5 + pp]);
00213     JMM[35] = -lf + lgg * Quad[2] +
00214           udata[5 + pp] * (-2 * lfg * udata[2 + pp] + lff * udata[5 + pp]);
00215     for (int i = 0; i < 6; i++) {
00216         for (int j = i + 1; j < 6; j++) {
00217             JMM[i * 6 + j] = JMM[j * 6 + i];
00218         }
00219     }
00220     // 4. Final values for temporal derivatives of field values
00221     h[0] = 0;
00222     h[1] = dxData[pp] * JMM[30] + dxData[1 + pp] * JMM[31] +
00223           dxData[2 + pp] * JMM[32] + dxData[3 + pp] * JMM[33] +
00224           dxData[4 + pp] * JMM[34] + dxData[5 + pp] * (-1 + JMM[35]);
00225     h[2] = -(dxData[pp] * JMM[24]) - dxData[1 + pp] * JMM[25] -
00226           dxData[2 + pp] * JMM[26] - dxData[3 + pp] * JMM[27] +
00227           dxData[4 + pp] * (1 - JMM[28]) - dxData[5 + pp] * JMM[29];
00228     h[3] = 0;
00229     h[4] = dxData[2 + pp];
00230     h[5] = -dxData[1 + pp];
00231     h[0] -= h[3] * JMM[3] + h[4] * JMM[4] + h[5] * JMM[5];
00232     h[1] -= h[3] * JMM[9] + h[4] * JMM[10] + h[5] * JMM[11];
00233     h[2] -= h[3] * JMM[15] + h[4] * JMM[16] + h[5] * JMM[17];
00234     // (1+2)^-1 applies only to E components
00235     dudata[pp + 0] =
00236         h[2] * (-JMM[2] * (1 + JMM[7])) + JMM[1] * JMM[8] +
00237         h[1] * (JMM[2] * JMM[13] - JMM[1] * (1 + JMM[14])) +
00238         h[0] * (1 - JMM[8] * JMM[13] + JMM[14] + JMM[7] * (1 + JMM[14]));
00239     dudata[pp + 1] =

```

```

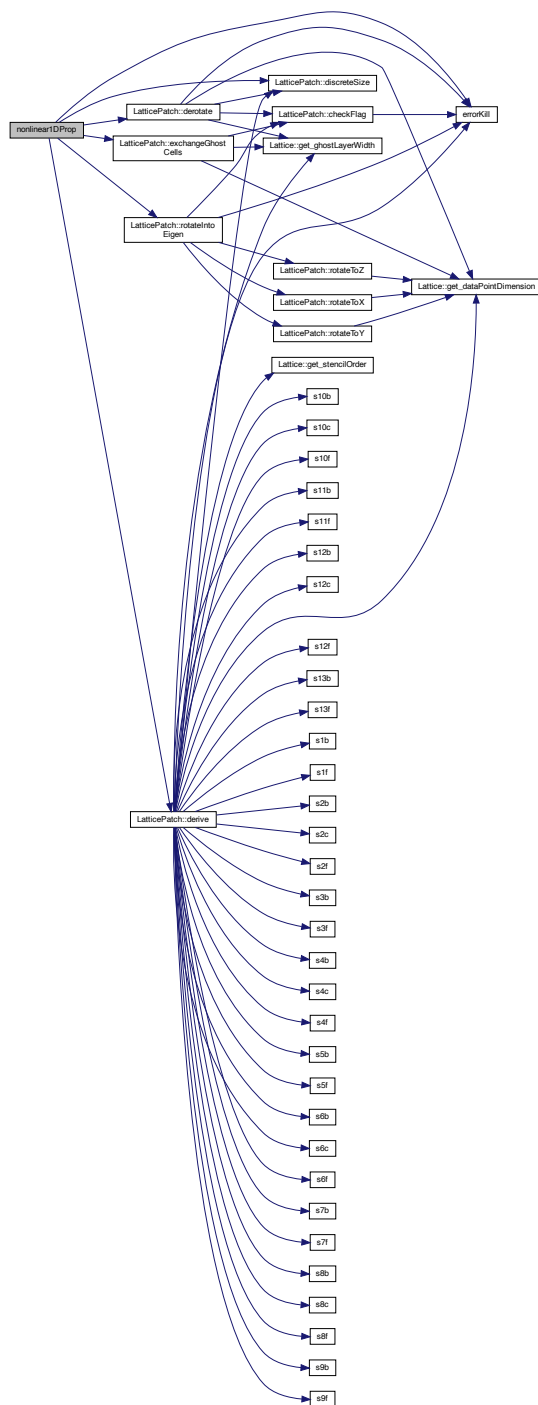
00240         h[2] * (JMM[2] * JMM[6] - (1 + JMM[0]) * JMM[8]) +
00241         h[1] * (1 - JMM[2] * JMM[12] + JMM[14] + JMM[0] * (1 + JMM[14])) +
00242         h[0] * (JMM[8] * JMM[12] - JMM[6] * (1 + JMM[14]));
00243     dudata[pp + 2] =
00244         h[2] * (1 - JMM[1] * JMM[6] + JMM[7] + JMM[0] * (1 + JMM[7])) +
00245         h[1] * (JMM[1] * JMM[12] - (1 + JMM[0]) * JMM[13]) +
00246         h[0] * (-(1 + JMM[7]) * JMM[12]) + JMM[6] * JMM[13]);
00247     pseudoDenom =
00248         -(1 + JMM[7]) * (-1 + JMM[2] * JMM[12]) +
00249         (JMM[2] * JMM[6] - JMM[8]) * JMM[13] + JMM[14] + JMM[7] * JMM[14] +
00250         JMM[0] * (1 + JMM[7] - JMM[8] * JMM[13] + (1 + JMM[7]) * JMM[14]) -
00251         JMM[1] * (-(JMM[8] * JMM[12]) + JMM[6] * (1 + JMM[14]));
00252     dudata[pp + 0] /= pseudoDenom;
00253     dudata[pp + 1] /= pseudoDenom;
00254     dudata[pp + 2] /= pseudoDenom;
00255     dudata[pp + 3] = h[3];
00256     dudata[pp + 4] = h[4];
00257     dudata[pp + 5] = h[5];
00258 }
00259 return;
00260 }

```

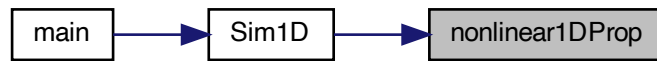
References [LatticePatch::buffData](#), [LatticePatch::derive\(\)](#), [LatticePatch::derotate\(\)](#), [LatticePatch::discreteSize\(\)](#), [errorKill\(\)](#), [LatticePatch::exchangeGhostCells\(\)](#), and [LatticePatch::rotateIntoEigen\(\)](#).

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

```

00294
00295
00296 sunrealtype *dxData = data->buffData[1 - 1];
00297 sunrealtype *dyData = data->buffData[2 - 1];
00298
00299 data->exchangeGhostCells(1);
00300 data->rotateIntoEigen(1);
00301 data->derive(1);
00302 data->derotate(1, dxData);
00303 data->exchangeGhostCells(2);
00304 data->rotateIntoEigen(2);
00305 data->derive(2);
00306 data->derotate(2, dyData);
00307
00308 sunrealtype f = NAN, g = NAN;
00309 sunrealtype lf = NAN, lff = NAN, lfg = NAN, lg = NAN, lgg = NAN;
00310 array<sunrealtype, 36> JMM;
00311 array<sunrealtype, 6> Quad;
00312 array<sunrealtype, 6> h;
00313 sunrealtype pseudoDenom = NAN;
00314 sunrealtype *udata = nullptr, *dudata = nullptr;
00315 udata = NV_DATA_P(u);
00316 dudata = NV_DATA_P(udot);
00317 int totalNP = data->discreteSize();
00318 for (int pp = 0; pp < totalNP * 6; pp += 6) {
00319     // 1
00320     f = 0.5 * ((Quad[0] = udata[pp] * udata[pp]) +
00321               (Quad[1] = udata[pp + 1] * udata[pp + 1]) +
00322               (Quad[2] = udata[pp + 2] * udata[pp + 2]) -
00323               (Quad[3] = udata[pp + 3] * udata[pp + 3]) -
00324               (Quad[4] = udata[pp + 4] * udata[pp + 4]) -
00325               (Quad[5] = udata[pp + 5] * udata[pp + 5]));
00326     g = udata[pp] * udata[pp + 3] + udata[pp + 1] * udata[pp + 4] +
00327         udata[pp + 2] * udata[pp + 5];
00328     // 2
00329     switch (*c) {
00330     case 0:
00331         lf = 0;
00332         lff = 0;
00333         lfg = 0;
00334         lg = 0;
00335         lgg = 0;
00336         break;

```

```

00337     case 2:
00338         lf = 0.000354046449700427580438254 * f * f +
00339             0.000191775160254398272737387 * g * g;
00340         lff = 0.0007080928994008551608765075 * f;
00341         lfg = 0.0003835503205087965454747749 * g;
00342         lg = 0.0003835503205087965454747749 * f * g;
00343         lgg = 0.0003835503205087965454747749 * f;
00344         break;
00345     case 1:
00346         lf = 0.000206527095658582755255648 * f;
00347         lff = 0.000206527095658582755255648;
00348         lfg = 0;
00349         lg = 0.0003614224174025198216973841 * g;
00350         lgg = 0.0003614224174025198216973841;
00351         break;
00352     case 3:
00353         lf = (0.000206527095658582755255648 + 0.000354046449700427580438254 * f) *
00354             f +
00355             0.000191775160254398272737387 * g * g;
00356         lff = 0.000206527095658582755255648 + 0.000708092899400855160876508 * f;
00357         lfg = 0.0003835503205087965454747749 * g;
00358         lg = (0.000361422417402519821697384 + 0.000383550320508796545474775 * f) *
00359             g;
00360         lgg = 0.000361422417402519821697384 + 0.000383550320508796545474775 * f;
00361         break;
00362     default:
00363         errorKill(
00364             "You need to specify a correct order in the weak-field expansion.");
00365     }
00366     // 3
00367     JMM[0] = lf + lff * Quad[0] +
00368         udata[3 + pp] * (2 * lfg * udata[pp] + lgg * udata[3 + pp]);
00369     JMM[6] =
00370         lff * udata[pp] * udata[1 + pp] + lfg * udata[1 + pp] * udata[3 + pp] +
00371         lfg * udata[pp] * udata[4 + pp] + lgg * udata[3 + pp] * udata[4 + pp];
00372     JMM[7] = lf + lff * Quad[1] +
00373         udata[4 + pp] * (2 * lfg * udata[1 + pp] + lgg * udata[4 + pp]);
00374     JMM[12] =
00375         lff * udata[pp] * udata[2 + pp] + lfg * udata[2 + pp] * udata[3 + pp] +
00376         lfg * udata[pp] * udata[5 + pp] + lgg * udata[3 + pp] * udata[5 + pp];
00377     JMM[13] = lff * udata[1 + pp] * udata[2 + pp] +
00378         lfg * udata[2 + pp] * udata[4 + pp] +
00379         lfg * udata[1 + pp] * udata[5 + pp] +
00380         lgg * udata[4 + pp] * udata[5 + pp];
00381     JMM[14] = lf + lff * Quad[2] +
00382         udata[5 + pp] * (2 * lfg * udata[2 + pp] + lgg * udata[5 + pp]);
00383     JMM[18] = lg + lfg * (Quad[0] - Quad[3 + 0]) +
00384         (-lff + lgg) * udata[pp] * udata[3 + pp];
00385     JMM[19] = -(udata[3 + pp] * (lff * udata[1 + pp] + lfg * udata[4 + pp])) +
00386         udata[pp] * (lfg * udata[1 + pp] + lgg * udata[4 + pp]);
00387     JMM[20] = -(udata[3 + pp] * (lff * udata[2 + pp] + lfg * udata[5 + pp])) +
00388         udata[pp] * (lfg * udata[2 + pp] + lgg * udata[5 + pp]);
00389     JMM[21] = -lf + lgg * Quad[0] +
00390         udata[3 + pp] * (-2 * lfg * udata[pp] + lff * udata[3 + pp]);
00391     JMM[24] = udata[1 + pp] * (lfg * udata[pp] + lgg * udata[3 + pp]) -
00392         (lff * udata[pp] + lfg * udata[3 + pp]) * udata[4 + pp];
00393     JMM[25] = lg + lfg * (Quad[1] - Quad[4 + 0]) +
00394         (-lff + lgg) * udata[1 + pp] * udata[4 + pp];
00395     JMM[26] = -(udata[4 + pp] * (lff * udata[2 + pp] + lfg * udata[5 + pp])) +
00396         udata[1 + pp] * (lfg * udata[2 + pp] + lgg * udata[5 + pp]);
00397     JMM[27] = lgg * udata[pp] * udata[1 + pp] +
00398         lff * udata[3 + pp] * udata[4 + pp] -
00399         lfg * (udata[1 + pp] * udata[3 + pp] + udata[pp] * udata[4 + pp]);
00400     JMM[28] = -lf + lgg * Quad[1] +
00401         udata[4 + pp] * (-2 * lfg * udata[1 + pp] + lff * udata[4 + pp]);
00402     JMM[30] = udata[2 + pp] * (lfg * udata[pp] + lgg * udata[3 + pp]) -
00403         (lff * udata[pp] + lfg * udata[3 + pp]) * udata[5 + pp];
00404     JMM[31] = udata[2 + pp] * (lfg * udata[1 + pp] + lgg * udata[4 + pp]) -
00405         (lff * udata[1 + pp] + lfg * udata[4 + pp]) * udata[5 + pp];
00406     JMM[32] = lg + lfg * (Quad[2] - Quad[5 + 0]) +
00407         (-lff + lgg) * udata[2 + pp] * udata[5 + pp];
00408     JMM[33] = lgg * udata[pp] * udata[2 + pp] +
00409         lff * udata[3 + pp] * udata[5 + pp] -
00410         lfg * (udata[2 + pp] * udata[3 + pp] + udata[pp] * udata[5 + pp]);
00411     JMM[34] =
00412         lgg * udata[1 + pp] * udata[2 + pp] +
00413         lff * udata[4 + pp] * udata[5 + pp] -
00414         lfg * (udata[2 + pp] * udata[4 + pp] + udata[1 + pp] * udata[5 + pp]);
00415     JMM[35] = -lf + lgg * Quad[2] +
00416         udata[5 + pp] * (-2 * lfg * udata[2 + pp] + lff * udata[5 + pp]);
00417     // 4
00418     for (int i = 0; i < 6; i++) {
00419         for (int j = i + 1; j < 6; j++) {
00420             JMM[i * 6 + j] = JMM[j * 6 + i];
00421         }
00422     }
00423     h[0] = 0;

```

```

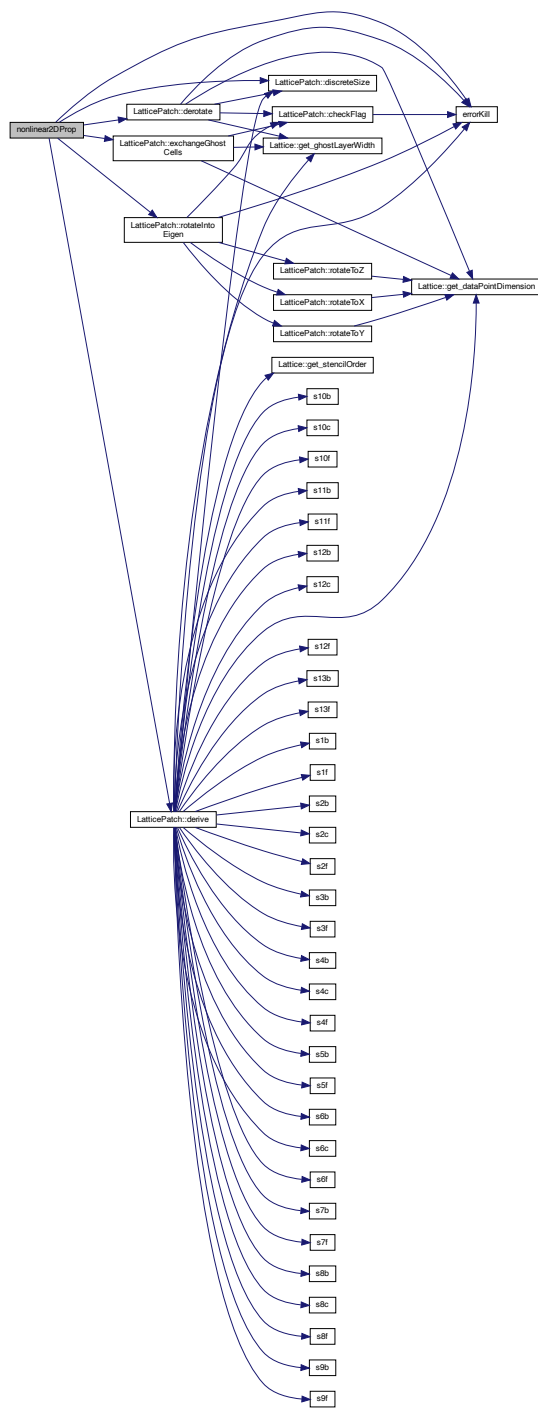
00424     h[1] = dxData[pp] * JMM[30] + dxData[1 + pp] * JMM[31] +
00425           dxData[2 + pp] * JMM[32] + dxData[3 + pp] * JMM[33] +
00426           dxData[4 + pp] * JMM[34] + dxData[5 + pp] * (-1 + JMM[35]);
00427     h[2] = -(dxData[pp] * JMM[24]) - dxData[1 + pp] * JMM[25] -
00428           dxData[2 + pp] * JMM[26] - dxData[3 + pp] * JMM[27] +
00429           dxData[4 + pp] * (1 - JMM[28]) - dxData[5 + pp] * JMM[29];
00430     h[3] = 0;
00431     h[4] = dxData[2 + pp];
00432     h[5] = -dxData[1 + pp];
00433     h[0] += -(dyData[pp] * JMM[30]) - dyData[1 + pp] * JMM[31] -
00434           dyData[2 + pp] * JMM[32] - dyData[3 + pp] * JMM[33] -
00435           dyData[4 + pp] * JMM[34] + dyData[5 + pp] * (1 - JMM[35]);
00436     h[1] += 0;
00437     h[2] += dyData[pp] * JMM[18] + dyData[1 + pp] * JMM[19] +
00438           dyData[2 + pp] * JMM[20] + dyData[3 + pp] * (-1 + JMM[21]) +
00439           dyData[4 + pp] * JMM[22] + dyData[5 + pp] * JMM[23];
00440     h[3] += -dyData[2 + pp];
00441     h[4] += 0;
00442     h[5] += dyData[pp];
00443     h[0] -= h[3] * JMM[3] + h[4] * JMM[4] + h[5] * JMM[5];
00444     h[1] -= h[3] * JMM[9] + h[4] * JMM[10] + h[5] * JMM[11];
00445     h[2] -= h[3] * JMM[15] + h[4] * JMM[16] + h[5] * JMM[17];
00446     dudata[pp + 0] =
00447         h[2] * (-(JMM[2] * (1 + JMM[7])) + JMM[1] * JMM[8]) +
00448         h[1] * (JMM[2] * JMM[13] - JMM[1] * (1 + JMM[14])) +
00449         h[0] * (1 - JMM[8] * JMM[13] + JMM[14] + JMM[7] * (1 + JMM[14]));
00450     dudata[pp + 1] =
00451         h[2] * (JMM[2] * JMM[6] - (1 + JMM[0]) * JMM[8]) +
00452         h[1] * (1 - JMM[2] * JMM[12] + JMM[14] + JMM[0] * (1 + JMM[14])) +
00453         h[0] * (JMM[8] * JMM[12] - JMM[6] * (1 + JMM[14]));
00454     dudata[pp + 2] =
00455         h[2] * (1 - JMM[1] * JMM[6] + JMM[7] + JMM[0] * (1 + JMM[7])) +
00456         h[1] * (JMM[1] * JMM[12] - (1 + JMM[0]) * JMM[13]) +
00457         h[0] * (-(1 + JMM[7]) * JMM[12]) + JMM[6] * JMM[13];
00458     pseudoDenom =
00459         -((1 + JMM[7]) * (-1 + JMM[2] * JMM[12])) +
00460         (JMM[2] * JMM[6] - JMM[8]) * JMM[13] + JMM[14] + JMM[7] * JMM[14] +
00461         JMM[0] * (1 + JMM[7] - JMM[8] * JMM[13] + (1 + JMM[7]) * JMM[14]) -
00462         JMM[1] * (-(JMM[8] * JMM[12]) + JMM[6] * (1 + JMM[14]));
00463     dudata[pp + 0] /= pseudoDenom;
00464     dudata[pp + 1] /= pseudoDenom;
00465     dudata[pp + 2] /= pseudoDenom;
00466     dudata[pp + 3] = h[3];
00467     dudata[pp + 4] = h[4];
00468     dudata[pp + 5] = h[5];
00469 }
00470 return;
00471 }

```

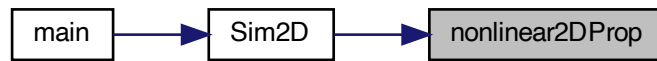
References [LatticePatch::buffData](#), [LatticePatch::derive\(\)](#), [LatticePatch::derotate\(\)](#), [LatticePatch::discreteSize\(\)](#), [errorKill\(\)](#), [LatticePatch::exchangeGhostCells\(\)](#), and [LatticePatch::rotateIntoEigen\(\)](#).

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

```

00510
00511
00512     sunrealtype *dxData = data->buffData[1 - 1];
00513     sunrealtype *dyData = data->buffData[2 - 1];
00514     sunrealtype *dzData = data->buffData[3 - 1];
00515
00516     data->exchangeGhostCells(1);
00517     data->rotateIntoEigen(1);
00518     data->derive(1);
00519     data->derotate(1,dxData);
00520     data->exchangeGhostCells(2);
00521     data->rotateIntoEigen(2);
00522     data->derive(2);
00523     data->derotate(2,dyData);
00524     data->exchangeGhostCells(3);
00525     data->rotateIntoEigen(3);
00526     data->derive(3);
00527     data->derotate(3,dzData);
00528
00529     sunrealtype f = NAN, g = NAN;
00530     sunrealtype lf = NAN, lff = NAN, lfg = NAN, lg = NAN, lgg = NAN;
00531     array<sunrealtype, 36> JMM;
00532     array<sunrealtype, 6> Quad;
00533     array<sunrealtype, 6> h;
00534     sunrealtype pseudoDenom = NAN;
00535     sunrealtype *udata = nullptr, *dudata = nullptr;
00536     udata = NV_DATA_P(u);
00537     dudata = NV_DATA_P(udot);
00538     int totalNP = data->discreteSize();
00539     for (int pp = 0; pp < totalNP * 6; pp += 6) {
00540         // 1
00541         f = 0.5 * ((Quad[0] = udata[pp] * udata[pp]) +
00542                 (Quad[1] = udata[pp + 1] * udata[pp + 1]) +
00543                 (Quad[2] = udata[pp + 2] * udata[pp + 2]) -
00544                 (Quad[3] = udata[pp + 3] * udata[pp + 3]) -
00545                 (Quad[4] = udata[pp + 4] * udata[pp + 4]) -
00546                 (Quad[5] = udata[pp + 5] * udata[pp + 5]));
00547         g = udata[pp] * udata[pp + 3] + udata[pp + 1] * udata[pp + 4] +
00548             udata[pp + 2] * udata[pp + 5];
00549         // 2
00550         switch (*c) {
00551             case 0:
00552                 lf = 0;

```



```

00553     lff = 0;
00554     lfg = 0;
00555     lg = 0;
00556     lgg = 0;
00557     break;
00558 case 2:
00559     lf = 0.000354046449700427580438254 * f * f +
00560         0.000191775160254398272737387 * g * g;
00561     lff = 0.0007080928994008551608765075 * f;
00562     lfg = 0.0003835503205087965454747749 * g;
00563     lg = 0.0003835503205087965454747749 * f * g;
00564     lgg = 0.0003835503205087965454747749 * f;
00565     break;
00566 case 1:
00567     lf = 0.000206527095658582755255648 * f;
00568     lff = 0.000206527095658582755255648;
00569     lfg = 0;
00570     lg = 0.0003614224174025198216973841 * g;
00571     lgg = 0.0003614224174025198216973841;
00572     break;
00573 case 3:
00574     lf = (0.000206527095658582755255648 + 0.000354046449700427580438254 * f) *
00575         f +
00576         0.000191775160254398272737387 * g * g;
00577     lff = 0.000206527095658582755255648 + 0.000708092899400855160876508 * f;
00578     lfg = 0.0003835503205087965454747749 * g;
00579     lg = (0.000361422417402519821697384 + 0.000383550320508796545474775 * f) *
00580         g;
00581     lgg = 0.000361422417402519821697384 + 0.000383550320508796545474775 * f;
00582     break;
00583 default:
00584     errorKill(
00585         "You need to specify a correct order in the weak-field expansion.");
00586 }
00587 // 3
00588 JMM[0] = lf + lff * Quad[0] +
00589     udata[3 + pp] * (2 * lfg * udata[pp] + lgg * udata[3 + pp]);
00590 JMM[6] =
00591     lff * udata[pp] * udata[1 + pp] + lfg * udata[1 + pp] * udata[3 + pp] +
00592     lfg * udata[pp] * udata[4 + pp] + lgg * udata[3 + pp] * udata[4 + pp];
00593 JMM[7] = lf + lff * Quad[1] +
00594     udata[4 + pp] * (2 * lfg * udata[1 + pp] + lgg * udata[4 + pp]);
00595 JMM[12] =
00596     lff * udata[pp] * udata[2 + pp] + lfg * udata[2 + pp] * udata[3 + pp] +
00597     lfg * udata[pp] * udata[5 + pp] + lgg * udata[3 + pp] * udata[5 + pp];
00598 JMM[13] = lff * udata[1 + pp] * udata[2 + pp] +
00599     lfg * udata[2 + pp] * udata[4 + pp] +
00600     lfg * udata[1 + pp] * udata[5 + pp] +
00601     lgg * udata[4 + pp] * udata[5 + pp];
00602 JMM[14] = lf + lff * Quad[2] +
00603     udata[5 + pp] * (2 * lfg * udata[2 + pp] + lgg * udata[5 + pp]);
00604 JMM[18] = lg + lfg * (Quad[0] - Quad[3 + 0]) +
00605     (-lff + lgg) * udata[pp] * udata[3 + pp];
00606 JMM[19] = -(udata[3 + pp] * (lff * udata[1 + pp] + lfg * udata[4 + pp])) +
00607     udata[pp] * (lfg * udata[1 + pp] + lgg * udata[4 + pp]);
00608 JMM[20] = -(udata[3 + pp] * (lff * udata[2 + pp] + lfg * udata[5 + pp])) +
00609     udata[pp] * (lfg * udata[2 + pp] + lgg * udata[5 + pp]);
00610 JMM[21] = -lf + lgg * Quad[0] +
00611     udata[3 + pp] * (-2 * lfg * udata[pp] + lff * udata[3 + pp]);
00612 JMM[24] = udata[1 + pp] * (lfg * udata[pp] + lgg * udata[3 + pp]) -
00613     (lff * udata[pp] + lfg * udata[3 + pp]) * udata[4 + pp];
00614 JMM[25] = lg + lfg * (Quad[1] - Quad[4 + 0]) +
00615     (-lff + lgg) * udata[1 + pp] * udata[4 + pp];
00616 JMM[26] = -(udata[4 + pp] * (lff * udata[2 + pp] + lfg * udata[5 + pp])) +
00617     udata[1 + pp] * (lfg * udata[2 + pp] + lgg * udata[5 + pp]);
00618 JMM[27] = lgg * udata[pp] * udata[1 + pp] +
00619     lff * udata[3 + pp] * udata[4 + pp] -
00620     lfg * (udata[1 + pp] * udata[3 + pp] + udata[pp] * udata[4 + pp]);
00621 JMM[28] = -lf + lgg * Quad[1] +
00622     udata[4 + pp] * (-2 * lfg * udata[1 + pp] + lff * udata[4 + pp]);
00623 JMM[30] = udata[2 + pp] * (lfg * udata[pp] + lgg * udata[3 + pp]) -
00624     (lff * udata[pp] + lfg * udata[3 + pp]) * udata[5 + pp];
00625 JMM[31] = udata[2 + pp] * (lfg * udata[1 + pp] + lgg * udata[4 + pp]) -
00626     (lff * udata[1 + pp] + lfg * udata[4 + pp]) * udata[5 + pp];
00627 JMM[32] = lg + lfg * (Quad[2] - Quad[5 + 0]) +
00628     (-lff + lgg) * udata[2 + pp] * udata[5 + pp];
00629 JMM[33] = lgg * udata[pp] * udata[2 + pp] +
00630     lff * udata[3 + pp] * udata[5 + pp] -
00631     lfg * (udata[2 + pp] * udata[3 + pp] + udata[pp] * udata[5 + pp]);
00632 JMM[34] =
00633     lgg * udata[1 + pp] * udata[2 + pp] +
00634     lff * udata[4 + pp] * udata[5 + pp] -
00635     lfg * (udata[2 + pp] * udata[4 + pp] + udata[1 + pp] * udata[5 + pp]);
00636 JMM[35] = -lf + lgg * Quad[2] +
00637     udata[5 + pp] * (-2 * lfg * udata[2 + pp] + lff * udata[5 + pp]);
00638 // 4
00639 for (int i = 0; i < 6; i++) {

```

```

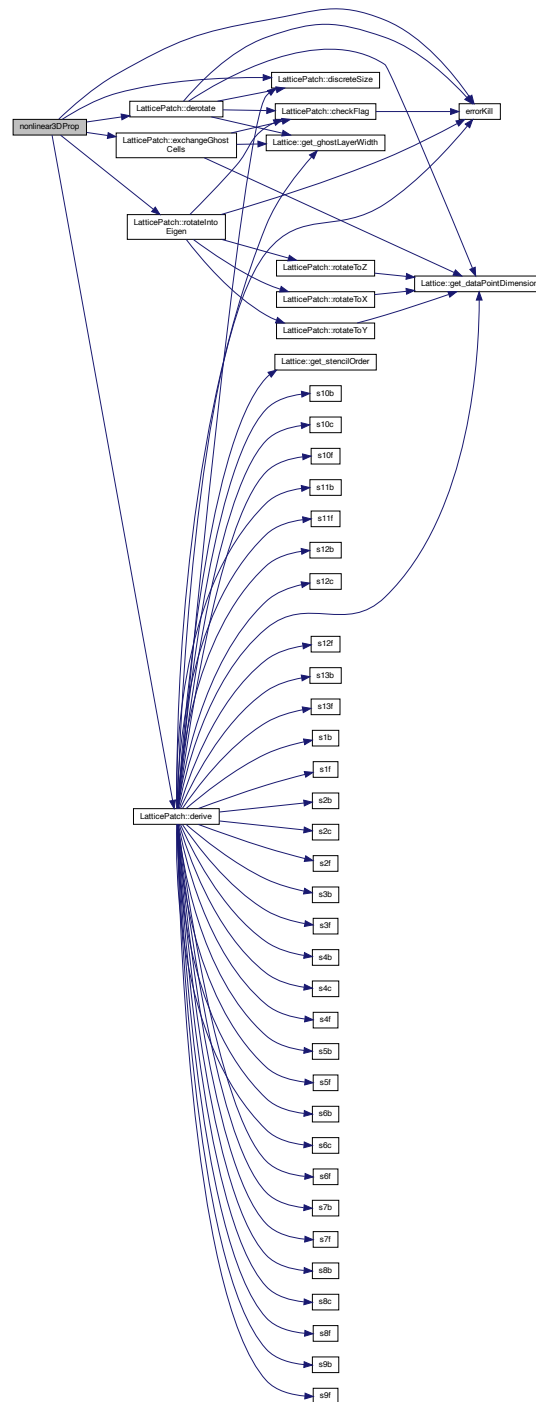
00640         for (int j = i + 1; j < 6; j++) {
00641             JMM[i * 6 + j] = JMM[j * 6 + i];
00642         }
00643     }
00644     h[0] = 0;
00645     h[1] = dxData[pp] * JMM[30] + dxData[1 + pp] * JMM[31] +
00646           dxData[2 + pp] * JMM[32] + dxData[3 + pp] * JMM[33] +
00647           dxData[4 + pp] * JMM[34] + dxData[5 + pp] * (-1 + JMM[35]);
00648     h[2] = -(dxData[pp] * JMM[24]) - dxData[1 + pp] * JMM[25] -
00649           dxData[2 + pp] * JMM[26] - dxData[3 + pp] * JMM[27] +
00650           dxData[4 + pp] * (1 - JMM[28]) - dxData[5 + pp] * JMM[29];
00651     h[3] = 0;
00652     h[4] = dxData[2 + pp];
00653     h[5] = -dxData[1 + pp];
00654     h[0] += -(dyData[pp] * JMM[30]) - dyData[1 + pp] * JMM[31] -
00655           dyData[2 + pp] * JMM[32] - dyData[3 + pp] * JMM[33] -
00656           dyData[4 + pp] * JMM[34] + dyData[5 + pp] * (1 - JMM[35]);
00657     h[1] += 0;
00658     h[2] += dyData[pp] * JMM[18] + dyData[1 + pp] * JMM[19] +
00659           dyData[2 + pp] * JMM[20] + dyData[3 + pp] * (-1 + JMM[21]) +
00660           dyData[4 + pp] * JMM[22] + dyData[5 + pp] * JMM[23];
00661     h[3] += -dyData[2 + pp];
00662     h[4] += 0;
00663     h[5] += dyData[pp];
00664     h[0] += dzData[pp] * JMM[24] + dzData[1 + pp] * JMM[25] +
00665           dzData[2 + pp] * JMM[26] + dzData[3 + pp] * JMM[27] +
00666           dzData[4 + pp] * (-1 + JMM[28]) + dzData[5 + pp] * JMM[29];
00667     h[1] += -(dzData[pp] * JMM[18]) - dzData[1 + pp] * JMM[19] -
00668           dzData[2 + pp] * JMM[20] + dzData[3 + pp] * (1 - JMM[21]) -
00669           dzData[4 + pp] * JMM[22] - dzData[5 + pp] * JMM[23];
00670     h[2] += 0;
00671     h[3] += dzData[1 + pp];
00672     h[4] += -dzData[pp];
00673     h[5] += 0;
00674     h[0] -= h[3] * JMM[3] + h[4] * JMM[4] + h[5] * JMM[5];
00675     h[1] -= h[3] * JMM[9] + h[4] * JMM[10] + h[5] * JMM[11];
00676     h[2] -= h[3] * JMM[15] + h[4] * JMM[16] + h[5] * JMM[17];
00677     dudata[pp + 0] =
00678         h[2] * (-(JMM[2] * (1 + JMM[7])) + JMM[1] * JMM[8]) +
00679         h[1] * (JMM[2] * JMM[13] - JMM[1] * (1 + JMM[14])) +
00680         h[0] * (1 - JMM[8] * JMM[13] + JMM[14] + JMM[7] * (1 + JMM[14]));
00681     dudata[pp + 1] =
00682         h[2] * (JMM[2] * JMM[6] - (1 + JMM[0]) * JMM[8]) +
00683         h[1] * (1 - JMM[2] * JMM[12] + JMM[14] + JMM[0] * (1 + JMM[14])) +
00684         h[0] * (JMM[8] * JMM[12] - JMM[6] * (1 + JMM[14]));
00685     dudata[pp + 2] =
00686         h[2] * (1 - JMM[1] * JMM[6] + JMM[7] + JMM[0] * (1 + JMM[7])) +
00687         h[1] * (JMM[1] * JMM[12] - (1 + JMM[0]) * JMM[13]) +
00688         h[0] * (-(1 + JMM[7]) * JMM[12]) + JMM[6] * JMM[13]);
00689     pseudoDenom =
00690         -((1 + JMM[7]) * (-1 + JMM[2] * JMM[12])) +
00691         (JMM[2] * JMM[6] - JMM[8]) * JMM[13] + JMM[14] + JMM[7] * JMM[14] +
00692         JMM[0] * (1 + JMM[7] - JMM[8] * JMM[13] + (1 + JMM[7]) * JMM[14]) -
00693         JMM[1] * (-(JMM[8] * JMM[12]) + JMM[6] * (1 + JMM[14]));
00694     dudata[pp + 0] /= pseudoDenom;
00695     dudata[pp + 1] /= pseudoDenom;
00696     dudata[pp + 2] /= pseudoDenom;
00697     dudata[pp + 3] = h[3];
00698     dudata[pp + 4] = h[4];
00699     dudata[pp + 5] = h[5];
00700 }
00701 return;
00702 }

```

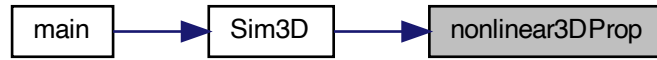
References [LatticePatch::buffData](#), [LatticePatch::derive\(\)](#), [LatticePatch::derotate\(\)](#), [LatticePatch::discreteSize\(\)](#), [errorKill\(\)](#), [LatticePatch::exchangeGhostCells\(\)](#), and [LatticePatch::rotateIntoEigen\(\)](#).

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 #include <math.h>
00011
00012 /// CCode right-hand-side function (CVRhsFn)
00013 int TimeEvolution::f(sunrealtype t, N_Vector u, N_Vector udot, void *data_loc) {
00014     // Set recover pointer to provided lattice patch where the data resides
00015     LatticePatch *data = nullptr;
00016     data = static_cast<LatticePatch *>(data_loc);
00017
00018     // pointers for update circle
00019     sunrealtype *udata = nullptr, *dudata = nullptr;
00020     sunrealtype *originaluData = nullptr, *originalduData = nullptr;
00021
00022     // Access NVECTOR_PARALLEL argument data with pointers
00023     udata = NV_DATA_P(u);
00024     dudata = NV_DATA_P(udot);
00025
00026     // Store original data location of the patch
00027     originaluData = data->uData;
00028     originalduData = data->duData;
00029     // Point patch data to arguments of f
00030     data->uData = udata;
00031     data->duData = dudata;
00032
00033     // Time-evolve these arguments (the field data) with specific propagator below
00034     TimeEvolver(data, u, udot, c);
00035
00036     // Refer patch data back to original location
00037     data->uData = originaluData;
00038     data->duData = originalduData;
00039
00040     return (0);
00041 }
00042
00043 /// only under-the-hood-callable Maxwell propagation in 1D
00044 /// unused parameters 2-4 for compliance with CVRhsFn
00045 /// same as the respective nonlinear function without nonlinear terms
00046 void linear1DProp(LatticePatch *data, N_Vector u, N_Vector udot, int *c) {
00047
00048     // pointers to temporal and spatial derivative data
00049     sunrealtype *duData = data->duData;
00050     sunrealtype *dxData = data->buffData[1 - 1];
00051
00052     // sequence along any dimension:
00053     data->exchangeGhostCells(1); // exchange halos
00054     data->rotateIntoEigen(
00055         1); // -> rotate all data to prepare derivative operation
00056     data->derive(1); // -> perform derivative on it
00057     data->derotate(
00058         1, dxData); // -> derotate derivative data to x-space for further use
00059
00060     int totalNP = data->discreteSize();
00061     int pp = 0;

```

```

00062     for (int i = 0; i < totalNP; i++) {
00063         pp = i * 6;
00064         /*
00065          simple vacuum Maxwell equations for spatial derivative only in x-direction
00066          temporal derivative is approximated by spatial derivative according to the
00067          numerical scheme with Jacobi=0 -> no polarization or magnetization terms
00068          */
00069         duData[pp + 0] = 0;
00070         duData[pp + 1] = -dxData[pp + 5];
00071         duData[pp + 2] = dxData[pp + 4];
00072         duData[pp + 3] = 0;
00073         duData[pp + 4] = dxData[pp + 2];
00074         duData[pp + 5] = -dxData[pp + 1];
00075     }
00076 }
00077
00078 /// nonlinear 1D HE propagation
00079 void nonlinear1DProp(LatticePatch *data, N_Vector u, N_Vector udot, int *c) {
00080
00081     // pointer to spatial derivative data sufficient, temporal derivative data
00082     // provided with udot
00083     sunrealtype *dxData = data->buffData[1 - 1];
00084
00085     // same sequence as in the linear case
00086     data->exchangeGhostCells(1);
00087     data->rotateIntoEigen(1);
00088     data->derive(1);
00089     data->derotate(1, dxData);
00090
00091     /*
00092     F and G are nonzero in the nonlinear case,
00093     polarization and magnetization contributions in Jacobi matrix style
00094     with derivatives of polarization and magnetization
00095     w.r.t. E- and B-field
00096     */
00097     sunrealtype f = NAN, g = NAN; // em field invariants F, G
00098     sunrealtype lff = NAN, lfg = NAN, lgg = NAN, lg = NAN,
00099     lgg = NAN; // derivatives of Lagrangian w.r.t. field invariants
00100     array<sunrealtype, 36> JMM; // Jacobi matrix
00101     array<sunrealtype, 6> Quad; // array to hold E^2 and B^2 components
00102     array<sunrealtype, 6> h; // holding temporal derivatives of E and B components
00103     // before operating (1+Z)^-1
00104     sunrealtype pseudoDenom = NAN; // needed for inversion of 1+Z
00105     sunrealtype *udata = nullptr,
00106     *dudata = nullptr; // pointers to data and temp. derivative data
00107     udata = NV_DATA_P(u);
00108     dudata = NV_DATA_P(udot);
00109     int totalNP = data->discreteSize(); // number of points in the patch
00110     for (int pp = 0; pp < totalNP * 6;
00111         pp += 6) { // loops through all 6dim points in the patch
00112         // for(int ppB=0;ppB<totalNP*6;ppB+=6*6){
00113         // for(int ppB=ppB;ppB<min(totalNP*6,ppB+6*6);ppB+=6){
00114         /// Calculation of the Jacobi matrix
00115         // 1. Calculate F and G
00116         f = 0.5 * ((Quad[0] = udata[pp] * udata[pp]) +
00117             (Quad[1] = udata[pp + 1] * udata[pp + 1]) +
00118             (Quad[2] = udata[pp + 2] * udata[pp + 2]) -
00119             (Quad[3] = udata[pp + 3] * udata[pp + 3]) -
00120             (Quad[4] = udata[pp + 4] * udata[pp + 4]) -
00121             (Quad[5] = udata[pp + 5] * udata[pp + 5]));
00122         g = udata[pp] * udata[pp + 3] + udata[pp + 1] * udata[pp + 4] +
00123             udata[pp + 2] * udata[pp + 5];
00124         // 2. Choose process/expansion order and assign derivative values of L
00125         // w.r.t. F, G
00126         switch (*c) {
00127         case 0:
00128             lff = 0;
00129             lfg = 0;
00130             lfg = 0;
00131             lg = 0;
00132             lgg = 0;
00133             break;
00134         case 2:
00135             lf = 0.000354046449700427580438254 * f * f +
00136                 0.000191775160254398272737387 * g * g;
00137             lff = 0.0007080928994008551608765075 * f;
00138             lfg = 0.0003835503205087965454747749 * g;
00139             lg = 0.0003835503205087965454747749 * f * g;
00140             lgg = 0.0003835503205087965454747749 * f;
00141             break;
00142         case 1:
00143             lf = 0.000206527095658582755255648 * f;
00144             lff = 0.000206527095658582755255648;
00145             lfg = 0;
00146             lg = 0.0003614224174025198216973841 * g;
00147             lgg = 0.0003614224174025198216973841;
00148             break;

```

```

00149     case 3:
00150         lf = (0.000206527095658582755255648 + 0.000354046449700427580438254 * f) *
00151             f +
00152             0.000191775160254398272737387 * g * g;
00153         lff = 0.000206527095658582755255648 + 0.0007080928994008551608765075 * f;
00154         lfg = 0.0003835503205087965454747749 * g;
00155         lg = (0.0003614224174025198216973841 +
00156             0.0003835503205087965454747749 * f) *
00157             g;
00158         lgg = 0.0003614224174025198216973841 + 0.0003835503205087965454747749 * f;
00159         break;
00160     default:
00161         errorKill(
00162             "You need to specify a correct order in the weak-field expansion.");
00163     }
00164     // 3. Assign Jacobi components
00165     JMM[0] = lf + lff * Quad[0] +
00166         udata[3 + pp] * (2 * lfg * udata[pp] + lgg * udata[3 + pp]);
00167     JMM[6] =
00168         lff * udata[pp] * udata[1 + pp] + lfg * udata[1 + pp] * udata[3 + pp] +
00169         lfg * udata[pp] * udata[4 + pp] + lgg * udata[3 + pp] * udata[4 + pp];
00170     JMM[7] = lf + lff * Quad[1] +
00171         udata[4 + pp] * (2 * lfg * udata[1 + pp] + lgg * udata[4 + pp]);
00172     JMM[12] =
00173         lff * udata[pp] * udata[2 + pp] + lfg * udata[2 + pp] * udata[3 + pp] +
00174         lfg * udata[pp] * udata[5 + pp] + lgg * udata[3 + pp] * udata[5 + pp];
00175     JMM[13] = lff * udata[1 + pp] * udata[2 + pp] +
00176         lfg * udata[2 + pp] * udata[4 + pp] +
00177         lfg * udata[1 + pp] * udata[5 + pp] +
00178         lgg * udata[4 + pp] * udata[5 + pp];
00179     JMM[14] = lf + lff * Quad[2] +
00180         udata[5 + pp] * (2 * lfg * udata[2 + pp] + lgg * udata[5 + pp]);
00181     JMM[18] = lg + lfg * (Quad[0] - Quad[3 + 0]) +
00182         (-lff + lgg) * udata[pp] * udata[3 + pp];
00183     JMM[19] = -(udata[3 + pp] * (lff * udata[1 + pp] + lfg * udata[4 + pp])) +
00184         udata[4 + pp] * (lfg * udata[1 + pp] + lgg * udata[4 + pp]);
00185     JMM[20] = -(udata[3 + pp] * (lff * udata[2 + pp] + lfg * udata[5 + pp])) +
00186         udata[pp] * (lfg * udata[2 + pp] + lgg * udata[5 + pp]);
00187     JMM[21] = -lf + lgg * Quad[0] +
00188         udata[3 + pp] * (-2 * lfg * udata[pp] + lff * udata[3 + pp]);
00189     JMM[24] = udata[1 + pp] * (lfg * udata[pp] + lgg * udata[3 + pp]) -
00190         (lff * udata[pp] + lfg * udata[3 + pp]) * udata[4 + pp];
00191     JMM[25] = lg + lfg * (Quad[1] - Quad[4 + 0]) +
00192         (-lff + lgg) * udata[1 + pp] * udata[4 + pp];
00193     JMM[26] = -(udata[4 + pp] * (lff * udata[2 + pp] + lfg * udata[5 + pp])) +
00194         udata[1 + pp] * (lfg * udata[2 + pp] + lgg * udata[5 + pp]);
00195     JMM[27] = lgg * udata[pp] * udata[1 + pp] +
00196         lff * udata[3 + pp] * udata[4 + pp] -
00197         lfg * (udata[1 + pp] * udata[3 + pp] + udata[pp] * udata[4 + pp]);
00198     JMM[28] = -lf + lgg * Quad[1] +
00199         udata[4 + pp] * (-2 * lfg * udata[1 + pp] + lff * udata[4 + pp]);
00200     JMM[30] = udata[2 + pp] * (lfg * udata[pp] + lgg * udata[3 + pp]) -
00201         (lff * udata[pp] + lfg * udata[3 + pp]) * udata[5 + pp];
00202     JMM[31] = udata[2 + pp] * (lfg * udata[1 + pp] + lgg * udata[4 + pp]) -
00203         (lff * udata[1 + pp] + lfg * udata[4 + pp]) * udata[5 + pp];
00204     JMM[32] = lg + lfg * (Quad[2] - Quad[5 + 0]) +
00205         (-lff + lgg) * udata[2 + pp] * udata[5 + pp];
00206     JMM[33] = lgg * udata[pp] * udata[2 + pp] +
00207         lff * udata[3 + pp] * udata[5 + pp] -
00208         lfg * (udata[2 + pp] * udata[3 + pp] + udata[pp] * udata[5 + pp]);
00209     JMM[34] =
00210         lgg * udata[1 + pp] * udata[2 + pp] +
00211         lff * udata[4 + pp] * udata[5 + pp] -
00212         lfg * (udata[2 + pp] * udata[4 + pp] + udata[1 + pp] * udata[5 + pp]);
00213     JMM[35] = -lf + lgg * Quad[2] +
00214         udata[5 + pp] * (-2 * lfg * udata[2 + pp] + lff * udata[5 + pp]);
00215     for (int i = 0; i < 6; i++) {
00216         for (int j = i + 1; j < 6; j++) {
00217             JMM[i * 6 + j] = JMM[j * 6 + i];
00218         }
00219     }
00220     // 4. Final values for temporal derivatives of field values
00221     h[0] = 0;
00222     h[1] = dxData[pp] * JMM[30] + dxData[1 + pp] * JMM[31] +
00223         dxData[2 + pp] * JMM[32] + dxData[3 + pp] * JMM[33] +
00224         dxData[4 + pp] * JMM[34] + dxData[5 + pp] * (-1 + JMM[35]);
00225     h[2] = -(dxData[pp] * JMM[24]) - dxData[1 + pp] * JMM[25] -
00226         dxData[2 + pp] * JMM[26] - dxData[3 + pp] * JMM[27] +
00227         dxData[4 + pp] * (1 - JMM[28]) - dxData[5 + pp] * JMM[29];
00228     h[3] = 0;
00229     h[4] = dxData[2 + pp];
00230     h[5] = -dxData[1 + pp];
00231     h[0] -= h[3] * JMM[3] + h[4] * JMM[4] + h[5] * JMM[5];
00232     h[1] -= h[3] * JMM[9] + h[4] * JMM[10] + h[5] * JMM[11];
00233     h[2] -= h[3] * JMM[15] + h[4] * JMM[16] + h[5] * JMM[17];
00234     // (1+Z)^-1 applies only to E components
00235     dudata[pp + 0] =

```

```

00236         h[2] * (-(JMM[2] * (1 + JMM[7])) + JMM[1] * JMM[8]) +
00237         h[1] * (JMM[2] * JMM[13] - JMM[1] * (1 + JMM[14])) +
00238         h[0] * (1 - JMM[8] * JMM[13] + JMM[14] + JMM[7] * (1 + JMM[14]));
00239     dudata[pp + 1] =
00240         h[2] * (JMM[2] * JMM[6] - (1 + JMM[0]) * JMM[8]) +
00241         h[1] * (1 - JMM[2] * JMM[12] + JMM[14] + JMM[0] * (1 + JMM[14])) +
00242         h[0] * (JMM[8] * JMM[12] - JMM[6] * (1 + JMM[14]));
00243     dudata[pp + 2] =
00244         h[2] * (1 - JMM[1] * JMM[6] + JMM[7] + JMM[0] * (1 + JMM[7])) +
00245         h[1] * (JMM[1] * JMM[12] - (1 + JMM[0]) * JMM[13]) +
00246         h[0] * (-(1 + JMM[7]) * JMM[12]) + JMM[6] * JMM[13]);
00247     pseudoDenom =
00248         -(1 + JMM[7]) * (-1 + JMM[2] * JMM[12])) +
00249         (JMM[2] * JMM[6] - JMM[8]) * JMM[13] + JMM[14] + JMM[7] * JMM[14] +
00250         JMM[0] * (1 + JMM[7] - JMM[8] * JMM[13] + (1 + JMM[7]) * JMM[14]) -
00251         JMM[1] * (-(JMM[8] * JMM[12]) + JMM[6] * (1 + JMM[14]));
00252     dudata[pp + 0] /= pseudoDenom;
00253     dudata[pp + 1] /= pseudoDenom;
00254     dudata[pp + 2] /= pseudoDenom;
00255     dudata[pp + 3] = h[3];
00256     dudata[pp + 4] = h[4];
00257     dudata[pp + 5] = h[5];
00258 }
00259 return;
00260 }
00261
00262 /// only under-the-hood-callable Maxwell propagation in 2D
00263 // unused parameters 2-4 for compliance with CVRhsFn
00264 // same as the respective nonlinear function without nonlinear terms
00265 void linear2DProp(LatticePatch *data, N_Vector u, N_Vector udot, int *c) {
00266
00267     sunrealtype *duData = data->duData;
00268     sunrealtype *dxData = data->buffData[1 - 1];
00269     sunrealtype *dyData = data->buffData[2 - 1];
00270
00271     data->exchangeGhostCells(1);
00272     data->rotateIntoEigen(1);
00273     data->derive(1);
00274     data->derotate(1, dxData);
00275     data->exchangeGhostCells(2);
00276     data->rotateIntoEigen(2);
00277     data->derive(2);
00278     data->derotate(2, dyData);
00279
00280     int totalNP = data->discreteSize();
00281     int pp = 0;
00282     for (int i = 0; i < totalNP; i++) {
00283         pp = i * 6;
00284         duData[pp + 0] = dyData[pp + 5];
00285         duData[pp + 1] = -dxData[pp + 5];
00286         duData[pp + 2] = -dyData[pp + 3] + dxData[pp + 4];
00287         duData[pp + 3] = -dyData[pp + 2];
00288         duData[pp + 4] = dxData[pp + 2];
00289         duData[pp + 5] = dyData[pp + 0] - dxData[pp + 1];
00290     }
00291 }
00292
00293 /// nonlinear 2D HE propagation
00294 void nonlinear2DProp(LatticePatch *data, N_Vector u, N_Vector udot, int *c) {
00295
00296     sunrealtype *dxData = data->buffData[1 - 1];
00297     sunrealtype *dyData = data->buffData[2 - 1];
00298
00299     data->exchangeGhostCells(1);
00300     data->rotateIntoEigen(1);
00301     data->derive(1);
00302     data->derotate(1, dxData);
00303     data->exchangeGhostCells(2);
00304     data->rotateIntoEigen(2);
00305     data->derive(2);
00306     data->derotate(2, dyData);
00307
00308     sunrealtype f = NAN, g = NAN;
00309     sunrealtype lf = NAN, lff = NAN, lfg = NAN, lg = NAN, lgg = NAN;
00310     array<sunrealtype, 36> JMM;
00311     array<sunrealtype, 6> Quad;
00312     array<sunrealtype, 6> h;
00313     sunrealtype pseudoDenom = NAN;
00314     sunrealtype *uData = nullptr, *duData = nullptr;
00315     uData = NV_DATA_P(u);
00316     duData = NV_DATA_P(udot);
00317     int totalNP = data->discreteSize();
00318     for (int pp = 0; pp < totalNP * 6; pp += 6) {
00319         // 1
00320         f = 0.5 * ((Quad[0] = uData[pp] * uData[pp]) +
00321                 (Quad[1] = uData[pp + 1] * uData[pp + 1]) +
00322                 (Quad[2] = uData[pp + 2] * uData[pp + 2]) -

```

```

00323         (Quad[3] = udata[pp + 3] * udata[pp + 3]) -
00324         (Quad[4] = udata[pp + 4] * udata[pp + 4]) -
00325         (Quad[5] = udata[pp + 5] * udata[pp + 5]));
00326     g = udata[pp] * udata[pp + 3] + udata[pp + 1] * udata[pp + 4] +
00327         udata[pp + 2] * udata[pp + 5];
00328     // 2
00329     switch (*c) {
00330     case 0:
00331         lf = 0;
00332         lff = 0;
00333         lfg = 0;
00334         lg = 0;
00335         lgg = 0;
00336         break;
00337     case 2:
00338         lf = 0.000354046449700427580438254 * f * f +
00339             0.000191775160254398272737387 * g * g;
00340         lff = 0.0007080928994008551608765075 * f;
00341         lfg = 0.0003835503205087965454747749 * g;
00342         lg = 0.0003835503205087965454747749 * f * g;
00343         lgg = 0.0003835503205087965454747749 * f;
00344         break;
00345     case 1:
00346         lf = 0.000206527095658582755255648 * f;
00347         lff = 0.000206527095658582755255648;
00348         lfg = 0;
00349         lg = 0.0003614224174025198216973841 * g;
00350         lgg = 0.0003614224174025198216973841;
00351         break;
00352     case 3:
00353         lf = (0.000206527095658582755255648 + 0.000354046449700427580438254 * f) *
00354             f +
00355             0.000191775160254398272737387 * g * g;
00356         lff = 0.000206527095658582755255648 + 0.000708092899400855160876508 * f;
00357         lfg = 0.0003835503205087965454747749 * g;
00358         lg = (0.000361422417402519821697384 + 0.000383550320508796545474775 * f) *
00359             g;
00360         lgg = 0.000361422417402519821697384 + 0.000383550320508796545474775 * f;
00361         break;
00362     default:
00363         errorKill(
00364             "You need to specify a correct order in the weak-field expansion.");
00365     }
00366     // 3
00367     JMM[0] = lf + lff * Quad[0] +
00368         udata[3 + pp] * (2 * lfg * udata[pp] + lgg * udata[3 + pp]);
00369     JMM[6] =
00370         lff * udata[pp] * udata[1 + pp] + lfg * udata[1 + pp] * udata[3 + pp] +
00371         lfg * udata[pp] * udata[4 + pp] + lgg * udata[3 + pp] * udata[4 + pp];
00372     JMM[7] = lf + lff * Quad[1] +
00373         udata[4 + pp] * (2 * lfg * udata[1 + pp] + lgg * udata[4 + pp]);
00374     JMM[12] =
00375         lff * udata[pp] * udata[2 + pp] + lfg * udata[2 + pp] * udata[3 + pp] +
00376         lfg * udata[pp] * udata[5 + pp] + lgg * udata[3 + pp] * udata[5 + pp];
00377     JMM[13] = lff * udata[1 + pp] * udata[2 + pp] +
00378         lfg * udata[2 + pp] * udata[4 + pp] +
00379         lfg * udata[1 + pp] * udata[5 + pp] +
00380         lgg * udata[4 + pp] * udata[5 + pp];
00381     JMM[14] = lf + lff * Quad[2] +
00382         udata[5 + pp] * (2 * lfg * udata[2 + pp] + lgg * udata[5 + pp]);
00383     JMM[18] = lg + lfg * (Quad[0] - Quad[3 + 0]) +
00384         (-lff + lgg) * udata[pp] * udata[3 + pp];
00385     JMM[19] = -(udata[3 + pp] * (lff * udata[1 + pp] + lfg * udata[4 + pp])) +
00386         udata[pp] * (lfg * udata[1 + pp] + lgg * udata[4 + pp]);
00387     JMM[20] = -(udata[3 + pp] * (lff * udata[2 + pp] + lfg * udata[5 + pp])) +
00388         udata[pp] * (lfg * udata[2 + pp] + lgg * udata[5 + pp]);
00389     JMM[21] = -lf + lgg * Quad[0] +
00390         udata[3 + pp] * (-2 * lfg * udata[pp] + lff * udata[3 + pp]);
00391     JMM[24] = udata[1 + pp] * (lfg * udata[pp] + lgg * udata[3 + pp]) -
00392         (lff * udata[pp] + lfg * udata[3 + pp]) * udata[4 + pp];
00393     JMM[25] = lg + lfg * (Quad[1] - Quad[4 + 0]) +
00394         (-lff + lgg) * udata[1 + pp] * udata[4 + pp];
00395     JMM[26] = -(udata[4 + pp] * (lff * udata[2 + pp] + lfg * udata[5 + pp])) +
00396         udata[1 + pp] * (lfg * udata[2 + pp] + lgg * udata[5 + pp]);
00397     JMM[27] = lgg * udata[pp] * udata[1 + pp] +
00398         lff * udata[3 + pp] * udata[4 + pp] -
00399         lfg * (udata[1 + pp] * udata[3 + pp] + udata[pp] * udata[4 + pp]);
00400     JMM[28] = -lf + lgg * Quad[1] +
00401         udata[4 + pp] * (-2 * lfg * udata[1 + pp] + lff * udata[4 + pp]);
00402     JMM[30] = udata[2 + pp] * (lfg * udata[pp] + lgg * udata[3 + pp]) -
00403         (lff * udata[pp] + lfg * udata[3 + pp]) * udata[5 + pp];
00404     JMM[31] = udata[2 + pp] * (lfg * udata[1 + pp] + lgg * udata[4 + pp]) -
00405         (lff * udata[1 + pp] + lfg * udata[4 + pp]) * udata[5 + pp];
00406     JMM[32] = lg + lfg * (Quad[2] - Quad[5 + 0]) +
00407         (-lff + lgg) * udata[2 + pp] * udata[5 + pp];
00408     JMM[33] = lgg * udata[pp] * udata[2 + pp] +
00409         lff * udata[3 + pp] * udata[5 + pp] -

```



```

00410         lfg * (udata[2 + pp] * udata[3 + pp] + udata[pp] * udata[5 + pp]));
00411     JMM[34] =
00412         lgg * udata[1 + pp] * udata[2 + pp] +
00413         lff * udata[4 + pp] * udata[5 + pp] -
00414         lfg * (udata[2 + pp] * udata[4 + pp] + udata[1 + pp] * udata[5 + pp]);
00415     JMM[35] = -lf + lgg * Quad[2] +
00416         udata[5 + pp] * (-2 * lfg * udata[2 + pp] + lff * udata[5 + pp]);
00417     // 4
00418     for (int i = 0; i < 6; i++) {
00419         for (int j = i + 1; j < 6; j++) {
00420             JMM[i * 6 + j] = JMM[j * 6 + i];
00421         }
00422     }
00423     h[0] = 0;
00424     h[1] = dxData[pp] * JMM[30] + dxData[1 + pp] * JMM[31] +
00425         dxData[2 + pp] * JMM[32] + dxData[3 + pp] * JMM[33] +
00426         dxData[4 + pp] * JMM[34] + dxData[5 + pp] * (-1 + JMM[35]);
00427     h[2] = -(dxData[pp] * JMM[24]) - dxData[1 + pp] * JMM[25] -
00428         dxData[2 + pp] * JMM[26] - dxData[3 + pp] * JMM[27] +
00429         dxData[4 + pp] * (1 - JMM[28]) - dxData[5 + pp] * JMM[29];
00430     h[3] = 0;
00431     h[4] = dxData[2 + pp];
00432     h[5] = -dxData[1 + pp];
00433     h[0] += -(dyData[pp] * JMM[30]) - dyData[1 + pp] * JMM[31] -
00434         dyData[2 + pp] * JMM[32] - dyData[3 + pp] * JMM[33] -
00435         dyData[4 + pp] * JMM[34] + dyData[5 + pp] * (1 - JMM[35]);
00436     h[1] += 0;
00437     h[2] += dyData[pp] * JMM[18] + dyData[1 + pp] * JMM[19] +
00438         dyData[2 + pp] * JMM[20] + dyData[3 + pp] * (-1 + JMM[21]) +
00439         dyData[4 + pp] * JMM[22] + dyData[5 + pp] * JMM[23];
00440     h[3] += -dyData[2 + pp];
00441     h[4] += 0;
00442     h[5] += dyData[pp];
00443     h[0] -= h[3] * JMM[3] + h[4] * JMM[4] + h[5] * JMM[5];
00444     h[1] -= h[3] * JMM[9] + h[4] * JMM[10] + h[5] * JMM[11];
00445     h[2] -= h[3] * JMM[15] + h[4] * JMM[16] + h[5] * JMM[17];
00446     dudata[pp + 0] =
00447         h[2] * (-JMM[2] * (1 + JMM[7])) + JMM[1] * JMM[8] +
00448         h[1] * (JMM[2] * JMM[13] - JMM[1] * (1 + JMM[14])) +
00449         h[0] * (1 - JMM[8] * JMM[13] + JMM[14] + JMM[7] * (1 + JMM[14]));
00450     dudata[pp + 1] =
00451         h[2] * (JMM[2] * JMM[6] - (1 + JMM[0]) * JMM[8]) +
00452         h[1] * (1 - JMM[2] * JMM[12] + JMM[14] + JMM[0] * (1 + JMM[14])) +
00453         h[0] * (JMM[8] * JMM[12] - JMM[6] * (1 + JMM[14]));
00454     dudata[pp + 2] =
00455         h[2] * (1 - JMM[1] * JMM[6] + JMM[7] + JMM[0] * (1 + JMM[7])) +
00456         h[1] * (JMM[1] * JMM[12] - (1 + JMM[0]) * JMM[13]) +
00457         h[0] * (-((1 + JMM[7]) * JMM[12]) + JMM[6] * JMM[13]);
00458     pseudoDenom =
00459         -((1 + JMM[7]) * (-1 + JMM[2] * JMM[12])) +
00460         (JMM[2] * JMM[6] - JMM[8]) * JMM[13] + JMM[14] + JMM[7] * JMM[14] +
00461         JMM[0] * (1 + JMM[7] - JMM[8] * JMM[13] + (1 + JMM[7]) * JMM[14]) -
00462         JMM[1] * (-JMM[8] * JMM[12]) + JMM[6] * (1 + JMM[14]);
00463     dudata[pp + 0] /= pseudoDenom;
00464     dudata[pp + 1] /= pseudoDenom;
00465     dudata[pp + 2] /= pseudoDenom;
00466     dudata[pp + 3] = h[3];
00467     dudata[pp + 4] = h[4];
00468     dudata[pp + 5] = h[5];
00469 }
00470 return;
00471 }
00472
00473 /// only under-the-hood-callable Maxwell propagation in 3D
00474 // unused parameters 2-4 for compliance with CVRhsFn
00475 // same as the respective nonlinear function without nonlinear terms
00476 void linear3DProp(LatticePatch *data, N_Vector u, N_Vector udot, int *c) {
00477
00478     sunrealtype *duData = data->duData;
00479     sunrealtype *dxData = data->buffData[1 - 1];
00480     sunrealtype *dyData = data->buffData[2 - 1];
00481     sunrealtype *dzData = data->buffData[3 - 1];
00482
00483     data->exchangeGhostCells(1);
00484     data->rotateIntoEigen(1);
00485     data->derive(1);
00486     data->derotate(1, dxData);
00487     data->exchangeGhostCells(2);
00488     data->rotateIntoEigen(2);
00489     data->derive(2);
00490     data->derotate(2, dyData);
00491     data->exchangeGhostCells(3);
00492     data->rotateIntoEigen(3);
00493     data->derive(3);
00494     data->derotate(3, dzData);
00495
00496     int totalNP = data->discreteSize();

```

```

00497     int pp = 0;
00498     for (int i = 0; i < totalNP; i++) {
00499         pp = i * 6;
00500         duData[pp + 0] = dyData[pp + 5] - dzData[pp + 4];
00501         duData[pp + 1] = dzData[pp + 3] - dxData[pp + 5];
00502         duData[pp + 2] = dxData[pp + 4] - dyData[pp + 3];
00503         duData[pp + 3] = -dyData[pp + 2] + dzData[pp + 1];
00504         duData[pp + 4] = -dzData[pp + 0] + dxData[pp + 2];
00505         duData[pp + 5] = -dxData[pp + 1] + dyData[pp + 0];
00506     }
00507 }
00508
00509 /// nonlinear 3D HE propagation
00510 void nonlinear3DProp(LatticePatch *data, N_Vector u, N_Vector udot, int *c) {
00511
00512     sunrealtype *dxData = data->buffData[1 - 1];
00513     sunrealtype *dyData = data->buffData[2 - 1];
00514     sunrealtype *dzData = data->buffData[3 - 1];
00515
00516     data->exchangeGhostCells(1);
00517     data->rotateIntoEigen(1);
00518     data->derive(1);
00519     data->derotate(1,dxData);
00520     data->exchangeGhostCells(2);
00521     data->rotateIntoEigen(2);
00522     data->derive(2);
00523     data->derotate(2,dyData);
00524     data->exchangeGhostCells(3);
00525     data->rotateIntoEigen(3);
00526     data->derive(3);
00527     data->derotate(3,dzData);
00528
00529     sunrealtype f = NAN, g = NAN;
00530     sunrealtype lff = NAN, lff = NAN, lfg = NAN, lg = NAN, lgg = NAN;
00531     array<sunrealtype, 36> JMM;
00532     array<sunrealtype, 6> Quad;
00533     array<sunrealtype, 6> h;
00534     sunrealtype pseudoDenom = NAN;
00535     sunrealtype *udata = nullptr, *dudata = nullptr;
00536     udata = NV_DATA_P(u);
00537     dudata = NV_DATA_P(udot);
00538     int totalNP = data->discreteSize();
00539     for (int pp = 0; pp < totalNP * 6; pp += 6) {
00540         // 1
00541         f = 0.5 * ((Quad[0] = udata[pp] * udata[pp]) +
00542                 (Quad[1] = udata[pp + 1] * udata[pp + 1]) +
00543                 (Quad[2] = udata[pp + 2] * udata[pp + 2]) -
00544                 (Quad[3] = udata[pp + 3] * udata[pp + 3]) -
00545                 (Quad[4] = udata[pp + 4] * udata[pp + 4]) -
00546                 (Quad[5] = udata[pp + 5] * udata[pp + 5]));
00547         g = udata[pp] * udata[pp + 3] + udata[pp + 1] * udata[pp + 4] +
00548             udata[pp + 2] * udata[pp + 5];
00549         // 2
00550         switch (*c) {
00551             case 0:
00552                 lff = 0;
00553                 lff = 0;
00554                 lfg = 0;
00555                 lg = 0;
00556                 lgg = 0;
00557                 break;
00558             case 2:
00559                 lf = 0.000354046449700427580438254 * f * f +
00560                     0.000191775160254398272737387 * g * g;
00561                 lff = 0.0007080928994008551608765075 * f;
00562                 lfg = 0.0003835503205087965454747749 * g;
00563                 lg = 0.0003835503205087965454747749 * f * g;
00564                 lgg = 0.0003835503205087965454747749 * f;
00565                 break;
00566             case 1:
00567                 lf = 0.000206527095658582755255648 * f;
00568                 lff = 0.000206527095658582755255648;
00569                 lfg = 0;
00570                 lg = 0.0003614224174025198216973841 * g;
00571                 lgg = 0.0003614224174025198216973841;
00572                 break;
00573             case 3:
00574                 lf = (0.000206527095658582755255648 + 0.000354046449700427580438254 * f) *
00575                     f +
00576                     0.000191775160254398272737387 * g * g;
00577                 lff = 0.000206527095658582755255648 + 0.000708092899400855160876508 * f;
00578                 lfg = 0.0003835503205087965454747749 * g;
00579                 lg = (0.000361422417402519821697384 + 0.000383550320508796545474775 * f) *
00580                     g;
00581                 lgg = 0.000361422417402519821697384 + 0.000383550320508796545474775 * f;
00582                 break;
00583             default:

```

```

00584     errorKill(
00585         "You need to specify a correct order in the weak-field expansion.");
00586     }
00587     // 3
00588     JMM[0] = lf + lff * Quad[0] +
00589         udata[3 + pp] * (2 * lfg * udata[pp] + lgg * udata[3 + pp]);
00590     JMM[6] =
00591         lff * udata[pp] * udata[1 + pp] + lfg * udata[1 + pp] * udata[3 + pp] +
00592         lfg * udata[pp] * udata[4 + pp] + lgg * udata[3 + pp] * udata[4 + pp];
00593     JMM[7] = lf + lff * Quad[1] +
00594         udata[4 + pp] * (2 * lfg * udata[1 + pp] + lgg * udata[4 + pp]);
00595     JMM[12] =
00596         lff * udata[pp] * udata[2 + pp] + lfg * udata[2 + pp] * udata[3 + pp] +
00597         lfg * udata[pp] * udata[5 + pp] + lgg * udata[3 + pp] * udata[5 + pp];
00598     JMM[13] = lff * udata[1 + pp] * udata[2 + pp] +
00599         lfg * udata[2 + pp] * udata[4 + pp] +
00600         lfg * udata[1 + pp] * udata[5 + pp] +
00601         lgg * udata[4 + pp] * udata[5 + pp];
00602     JMM[14] = lf + lff * Quad[2] +
00603         udata[5 + pp] * (2 * lfg * udata[2 + pp] + lgg * udata[5 + pp]);
00604     JMM[18] = lg + lfg * (Quad[0] - Quad[3 + 0]) +
00605         (-lff + lgg) * udata[pp] * udata[3 + pp];
00606     JMM[19] = -(udata[3 + pp] * (lff * udata[1 + pp] + lfg * udata[4 + pp])) +
00607         udata[pp] * (lfg * udata[1 + pp] + lgg * udata[4 + pp]);
00608     JMM[20] = -(udata[3 + pp] * (lff * udata[2 + pp] + lfg * udata[5 + pp])) +
00609         udata[pp] * (lfg * udata[2 + pp] + lgg * udata[5 + pp]);
00610     JMM[21] = -lf + lgg * Quad[0] +
00611         udata[3 + pp] * (-2 * lfg * udata[pp] + lff * udata[3 + pp]);
00612     JMM[24] = udata[1 + pp] * (lfg * udata[pp] + lgg * udata[3 + pp]) -
00613         (lff * udata[pp] + lfg * udata[3 + pp]) * udata[4 + pp];
00614     JMM[25] = lg + lfg * (Quad[1] - Quad[4 + 0]) +
00615         (-lff + lgg) * udata[1 + pp] * udata[4 + pp];
00616     JMM[26] = -(udata[4 + pp] * (lff * udata[2 + pp] + lfg * udata[5 + pp])) +
00617         udata[1 + pp] * (lfg * udata[2 + pp] + lgg * udata[5 + pp]);
00618     JMM[27] = lgg * udata[pp] * udata[1 + pp] +
00619         lff * udata[3 + pp] * udata[4 + pp] -
00620         lfg * (udata[1 + pp] * udata[3 + pp] + udata[pp] * udata[4 + pp]);
00621     JMM[28] = -lf + lgg * Quad[1] +
00622         udata[4 + pp] * (-2 * lfg * udata[1 + pp] + lff * udata[4 + pp]);
00623     JMM[30] = udata[2 + pp] * (lfg * udata[pp] + lgg * udata[3 + pp]) -
00624         (lff * udata[pp] + lfg * udata[3 + pp]) * udata[5 + pp];
00625     JMM[31] = udata[2 + pp] * (lfg * udata[1 + pp] + lgg * udata[4 + pp]) -
00626         (lff * udata[1 + pp] + lfg * udata[4 + pp]) * udata[5 + pp];
00627     JMM[32] = lg + lfg * (Quad[2] - Quad[5 + 0]) +
00628         (-lff + lgg) * udata[2 + pp] * udata[5 + pp];
00629     JMM[33] = lgg * udata[pp] * udata[2 + pp] +
00630         lff * udata[3 + pp] * udata[5 + pp] -
00631         lfg * (udata[2 + pp] * udata[3 + pp] + udata[pp] * udata[5 + pp]);
00632     JMM[34] =
00633         lgg * udata[1 + pp] * udata[2 + pp] +
00634         lff * udata[4 + pp] * udata[5 + pp] -
00635         lfg * (udata[2 + pp] * udata[4 + pp] + udata[1 + pp] * udata[5 + pp]);
00636     JMM[35] = -lf + lgg * Quad[2] +
00637         udata[5 + pp] * (-2 * lfg * udata[2 + pp] + lff * udata[5 + pp]);
00638     // 4
00639     for (int i = 0; i < 6; i++) {
00640         for (int j = i + 1; j < 6; j++) {
00641             JMM[i * 6 + j] = JMM[j * 6 + i];
00642         }
00643     }
00644     h[0] = 0;
00645     h[1] = dxData[pp] * JMM[30] + dxData[1 + pp] * JMM[31] +
00646         dxData[2 + pp] * JMM[32] + dxData[3 + pp] * JMM[33] +
00647         dxData[4 + pp] * JMM[34] + dxData[5 + pp] * (-1 + JMM[35]);
00648     h[2] = -(dxData[pp] * JMM[24]) - dxData[1 + pp] * JMM[25] -
00649         dxData[2 + pp] * JMM[26] - dxData[3 + pp] * JMM[27] +
00650         dxData[4 + pp] * (1 - JMM[28]) - dxData[5 + pp] * JMM[29];
00651     h[3] = 0;
00652     h[4] = dxData[2 + pp];
00653     h[5] = -dxData[1 + pp];
00654     h[0] += -(dyData[pp] * JMM[30]) - dyData[1 + pp] * JMM[31] -
00655         dyData[2 + pp] * JMM[32] - dyData[3 + pp] * JMM[33] -
00656         dyData[4 + pp] * JMM[34] + dyData[5 + pp] * (1 - JMM[35]);
00657     h[1] += 0;
00658     h[2] += dyData[pp] * JMM[18] + dyData[1 + pp] * JMM[19] +
00659         dyData[2 + pp] * JMM[20] + dyData[3 + pp] * (-1 + JMM[21]) +
00660         dyData[4 + pp] * JMM[22] + dyData[5 + pp] * JMM[23];
00661     h[3] += -dyData[2 + pp];
00662     h[4] += 0;
00663     h[5] += dyData[pp];
00664     h[0] += dzData[pp] * JMM[24] + dzData[1 + pp] * JMM[25] +
00665         dzData[2 + pp] * JMM[26] + dzData[3 + pp] * JMM[27] +
00666         dzData[4 + pp] * (-1 + JMM[28]) + dzData[5 + pp] * JMM[29];
00667     h[1] += -(dzData[pp] * JMM[18]) - dzData[1 + pp] * JMM[19] -
00668         dzData[2 + pp] * JMM[20] + dzData[3 + pp] * (1 - JMM[21]) -
00669         dzData[4 + pp] * JMM[22] - dzData[5 + pp] * JMM[23];
00670     h[2] += 0;

```

```

00671     h[3] += dzData[1 + pp];
00672     h[4] += -dzData[pp];
00673     h[5] += 0;
00674     h[0] -= h[3] * JMM[3] + h[4] * JMM[4] + h[5] * JMM[5];
00675     h[1] -= h[3] * JMM[9] + h[4] * JMM[10] + h[5] * JMM[11];
00676     h[2] -= h[3] * JMM[15] + h[4] * JMM[16] + h[5] * JMM[17];
00677     dudata[pp + 0] =
00678         h[2] * (-(JMM[2] * (1 + JMM[7])) + JMM[1] * JMM[8]) +
00679         h[1] * (JMM[2] * JMM[13] - JMM[1] * (1 + JMM[14])) +
00680         h[0] * (1 - JMM[8] * JMM[13] + JMM[14] + JMM[7] * (1 + JMM[14]));
00681     dudata[pp + 1] =
00682         h[2] * (JMM[2] * JMM[6] - (1 + JMM[0]) * JMM[8]) +
00683         h[1] * (1 - JMM[2] * JMM[12] + JMM[14] + JMM[0] * (1 + JMM[14])) +
00684         h[0] * (JMM[8] * JMM[12] - JMM[6] * (1 + JMM[14]));
00685     dudata[pp + 2] =
00686         h[2] * (1 - JMM[1] * JMM[6] + JMM[7] + JMM[0] * (1 + JMM[7])) +
00687         h[1] * (JMM[1] * JMM[12] - (1 + JMM[0]) * JMM[13]) +
00688         h[0] * (-(1 + JMM[7]) * JMM[12]) + JMM[6] * JMM[13]);
00689     pseudoDenom =
00690         -((1 + JMM[7]) * (-1 + JMM[2] * JMM[12])) +
00691         (JMM[2] * JMM[6] - JMM[8]) * JMM[13] + JMM[14] + JMM[7] * JMM[14] +
00692         JMM[0] * (1 + JMM[7] - JMM[8] * JMM[13] + (1 + JMM[7]) * JMM[14]) -
00693         JMM[1] * (-(JMM[8] * JMM[12]) + JMM[6] * (1 + JMM[14]));
00694     dudata[pp + 0] /= pseudoDenom;
00695     dudata[pp + 1] /= pseudoDenom;
00696     dudata[pp + 2] /= pseudoDenom;
00697     dudata[pp + 3] = h[3];
00698     dudata[pp + 4] = h[4];
00699     dudata[pp + 5] = h[5];
00700 }
00701 return;
00702 }

```

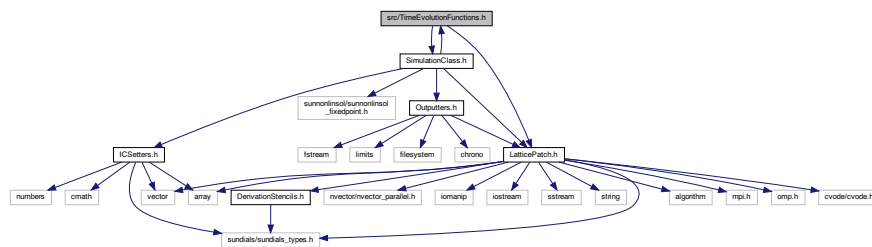
6.30 src/TimeEvolutionFunctions.h File Reference

Functions to propagate data vectors in time according to Maxwell's equations, and various orders in the HE weak-field expansion.

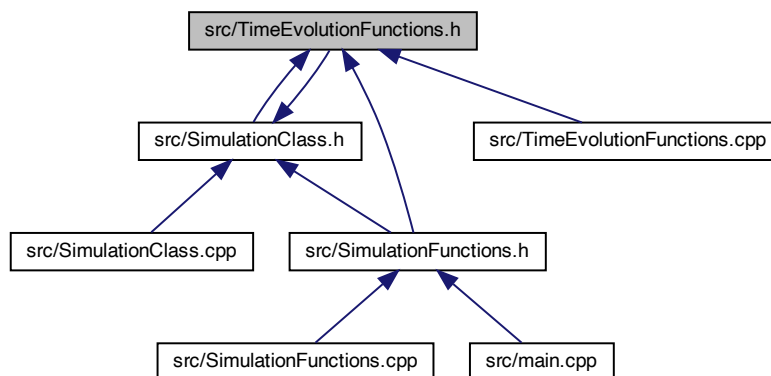
```
#include "LatticePatch.h"
```

```
#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.

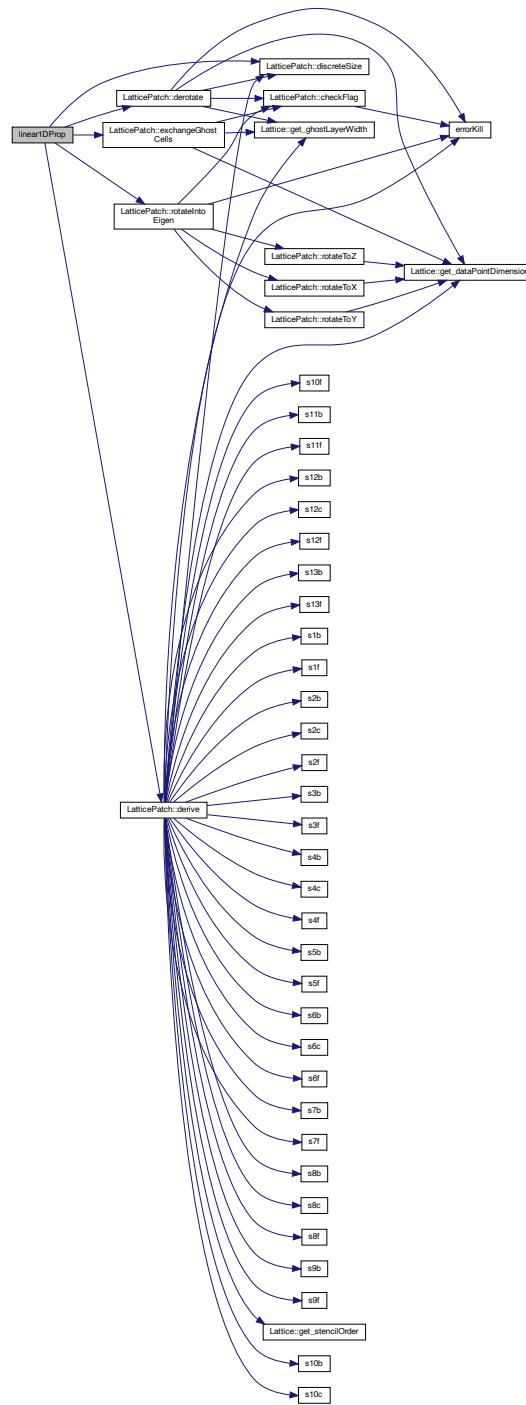
Maxwell propagation function for 1D – only for reference.

Definition at line 46 of file [TimeEvolutionFunctions.cpp](#).

```
00046 {
00047
00048 // pointers to temporal and spatial derivative data
00049 sunrealtype *duData = data->duData;
00050 sunrealtype *dxData = data->buffData[1 - 1];
00051
00052 // sequence along any dimension:
00053 data->exchangeGhostCells(1); // exchange halos
00054 data->rotateIntoEigen(
00055     1); // -> rotate all data to prepare derivative operation
00056 data->derive(1); // -> perform derivative on it
00057 data->derotate(
00058     1, dxData); // -> derotate derivative data to x-space for further use
00059
00060 int totalNP = data->discreteSize();
00061 int pp = 0;
00062 for (int i = 0; i < totalNP; i++) {
00063     pp = i * 6;
00064     /*
00065     simple vacuum Maxwell equations for spatial derivative only in x-direction
00066     temporal derivative is approximated by spatial derivative according to the
00067     numerical scheme with Jacobi=0 -> no polarization or magnetization terms
00068     */
00069     duData[pp + 0] = 0;
00070     duData[pp + 1] = -dxData[pp + 5];
00071     duData[pp + 2] = dxData[pp + 4];
00072     duData[pp + 3] = 0;
00073     duData[pp + 4] = dxData[pp + 2];
00074     duData[pp + 5] = -dxData[pp + 1];
00075 }
00076 }
```

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

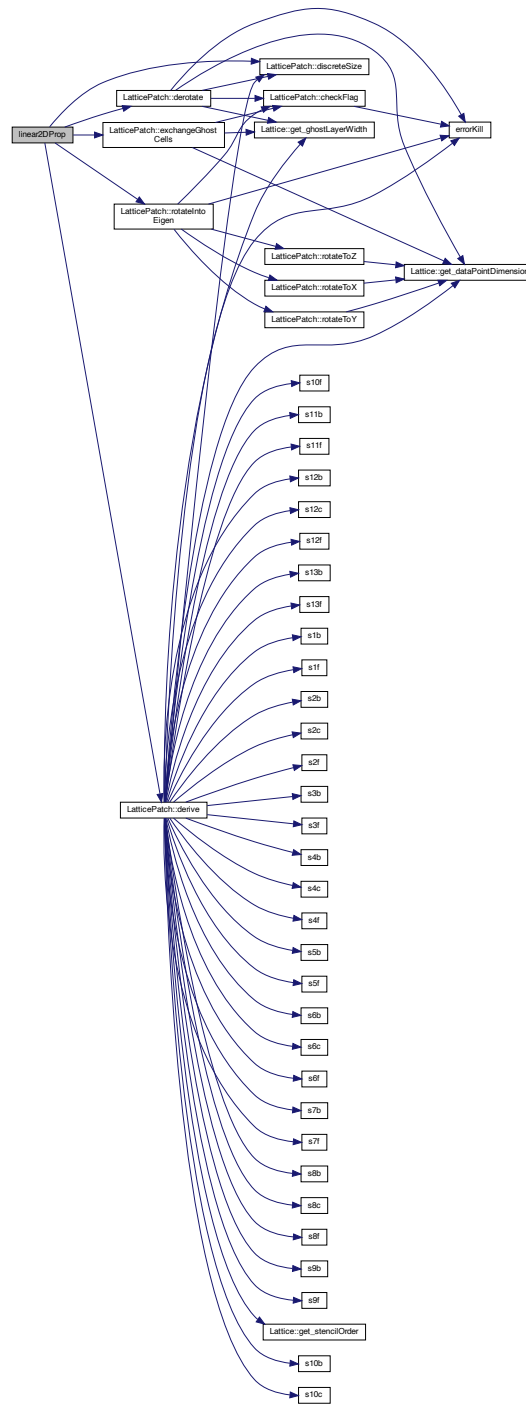
```

00265
00266
00267     sunrealtype *duData = data->duData;
00268     sunrealtype *dxData = data->buffData[1 - 1];
00269     sunrealtype *dyData = data->buffData[2 - 1];
00270
00271     data->exchangeGhostCells(1);
00272     data->rotateIntoEigen(1);
00273     data->derive(1);
00274     data->derotate(1, dxData);
00275     data->exchangeGhostCells(2);
00276     data->rotateIntoEigen(2);
00277     data->derive(2);
00278     data->derotate(2, dyData);
00279
00280     int totalNP = data->discreteSize();
00281     int pp = 0;
00282     for (int i = 0; i < totalNP; i++) {
00283         pp = i * 6;
00284         duData[pp + 0] = dyData[pp + 5];
00285         duData[pp + 1] = -dxData[pp + 5];
00286         duData[pp + 2] = -dyData[pp + 3] + dxData[pp + 4];
00287         duData[pp + 3] = -dyData[pp + 2];
00288         duData[pp + 4] = dxData[pp + 2];
00289         duData[pp + 5] = dyData[pp + 0] - dxData[pp + 1];
00290     }
00291 }

```

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

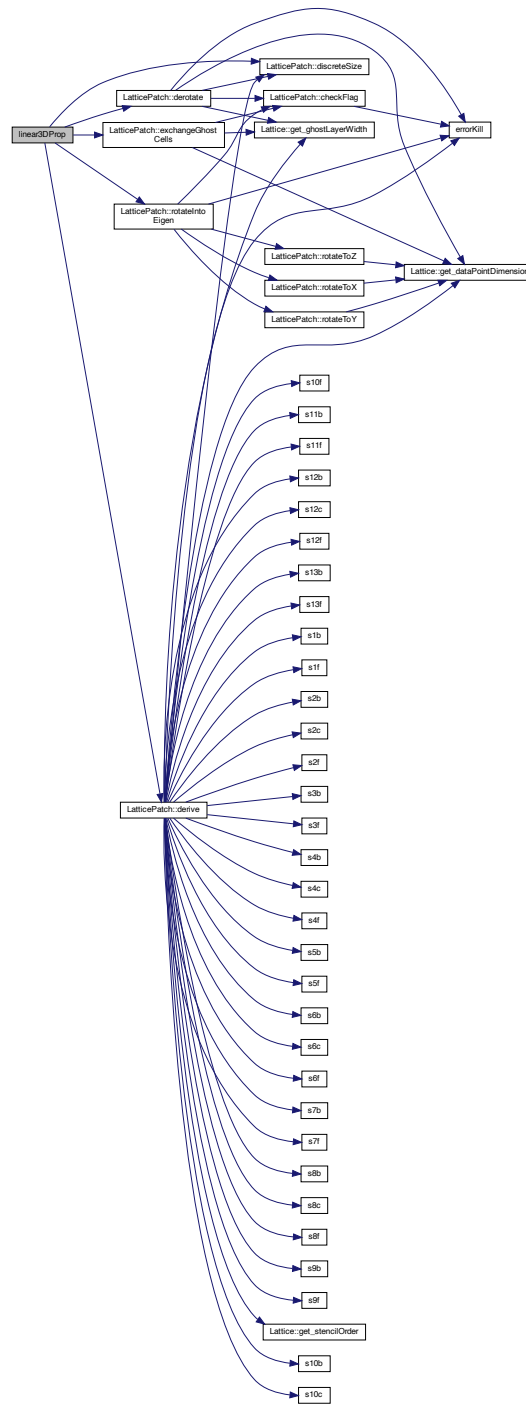
```

00476                                     {
00477
00478     sunrealtype *duData = data->duData;
00479     sunrealtype *dxData = data->buffData[1 - 1];
00480     sunrealtype *dyData = data->buffData[2 - 1];
00481     sunrealtype *dzData = data->buffData[3 - 1];
00482
00483     data->exchangeGhostCells(1);
00484     data->rotateIntoEigen(1);
00485     data->derive(1);
00486     data->derotate(1, dxData);
00487     data->exchangeGhostCells(2);
00488     data->rotateIntoEigen(2);
00489     data->derive(2);
00490     data->derotate(2, dyData);
00491     data->exchangeGhostCells(3);
00492     data->rotateIntoEigen(3);
00493     data->derive(3);
00494     data->derotate(3, dzData);
00495
00496     int totalNP = data->discreteSize();
00497     int pp = 0;
00498     for (int i = 0; i < totalNP; i++) {
00499         pp = i * 6;
00500         duData[pp + 0] = dyData[pp + 5] - dzData[pp + 4];
00501         duData[pp + 1] = dzData[pp + 3] - dxData[pp + 5];
00502         duData[pp + 2] = dxData[pp + 4] - dyData[pp + 3];
00503         duData[pp + 3] = -dyData[pp + 2] + dzData[pp + 1];
00504         duData[pp + 4] = -dzData[pp + 0] + dxData[pp + 2];
00505         duData[pp + 5] = -dxData[pp + 1] + dyData[pp + 0];
00506     }
00507 }

```

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. Calculation of the Jacobi matrix

Definition at line 79 of file [TimeEvolutionFunctions.cpp](#).

```

00079
00080
00081 // pointer to spatial derivative data sufficient, temporal derivative data
00082 // provided with udot
00083 sunrealtype *dxData = data->buffData[1 - 1];
00084
00085 // same sequence as in the linear case
00086 data->exchangeGhostCells(1);
00087 data->rotateIntoEigen(1);
00088 data->derive(1);
00089 data->derotate(1, dxData);
00090
00091 /*
00092 F and G are nonzero in the nonlinear case,
00093 polarization and magnetization contributions in Jacobi matrix style
00094 with derivatives of polarization and magnetization
00095 w.r.t. E- and B-field
00096 */
00097 sunrealtype f = NAN, g = NAN; // em field invariants F, G
00098 sunrealtype lff = NAN, lfg = NAN, lgg = NAN, lg = NAN,
00099 lgg = NAN; // derivatives of Lagrangian w.r.t. field invariants
00100 array<sunrealtype, 36> JMM; // Jacobi matrix
00101 array<sunrealtype, 6> Quad; // array to hold E^2 and B^2 components
00102 array<sunrealtype, 6> h; // holding temporal derivatives of E and B components
00103 // before operating (1+Z)^-1
00104 sunrealtype pseudoDenom = NAN; // needed for inversion of 1+Z
00105 sunrealtype *udata = nullptr,
00106 *dudata = nullptr; // pointers to data and temp. derivative data
00107 udata = NV_DATA_P(u);
00108 dudata = NV_DATA_P(udot);
00109 int totalNP = data->discreteSize(); // number of points in the patch
00110 for (int pp = 0; pp < totalNP * 6;
00111      pp += 6) { // loops through all 6dim points in the patch
00112     // for(int ppB=0;ppB<totalNP*6;ppB+=6*6){
00113     // for(int ppB;ppB<min(totalNP*6,ppB+6*6);ppB+=6){
00114     /// Calculation of the Jacobi matrix
00115     // 1. Calculate F and G
00116     f = 0.5 * ((Quad[0] = udata[pp] * udata[pp]) +
00117               (Quad[1] = udata[pp + 1] * udata[pp + 1]) +
00118               (Quad[2] = udata[pp + 2] * udata[pp + 2]) -
00119               (Quad[3] = udata[pp + 3] * udata[pp + 3]) -
00120               (Quad[4] = udata[pp + 4] * udata[pp + 4]) -
00121               (Quad[5] = udata[pp + 5] * udata[pp + 5]));
00122     g = udata[pp] * udata[pp + 3] + udata[pp + 1] * udata[pp + 4] +
00123         udata[pp + 2] * udata[pp + 5];
00124     // 2. Choose process/expansion order and assign derivative values of L
00125     // w.r.t. F, G
00126     switch (*c) {
00127     case 0:
00128         lff = 0;
00129         lfg = 0;
00130         lfg = 0;
00131         lg = 0;
00132         lgg = 0;
00133         break;
00134     case 2:
00135         lff = 0.000354046449700427580438254 * f * f +
00136             0.000191775160254398272737387 * g * g;
00137         lff = 0.0007080928994008551608765075 * f;
00138         lfg = 0.0003835503205087965454747749 * g;
00139         lg = 0.0003835503205087965454747749 * f * g;
00140         lgg = 0.0003835503205087965454747749 * f;
00141         break;
00142     case 1:
00143         lff = 0.000206527095658582755255648 * f;
00144         lff = 0.000206527095658582755255648;
00145         lfg = 0;
00146         lg = 0.0003614224174025198216973841 * g;
00147         lgg = 0.0003614224174025198216973841;
00148         break;
00149     case 3:
00150         lff = (0.000206527095658582755255648 + 0.000354046449700427580438254 * f) *
00151             f +
00152             0.000191775160254398272737387 * g * g;

```

```

00153     lff = 0.000206527095658582755255648 + 0.0007080928994008551608765075 * f;
00154     lfg = 0.0003835503205087965454747749 * g;
00155     lg = (0.0003614224174025198216973841 +
00156           0.0003835503205087965454747749 * f) *
00157         g;
00158     lgg = 0.0003614224174025198216973841 + 0.0003835503205087965454747749 * f;
00159     break;
00160     default:
00161         errorKill(
00162             "You need to specify a correct order in the weak-field expansion.");
00163     }
00164     // 3. Assign Jacobi components
00165     JMM[0] = lf + lff * Quad[0] +
00166         udata[3 + pp] * (2 * lfg * udata[pp] + lgg * udata[3 + pp]);
00167     JMM[6] =
00168         lff * udata[pp] * udata[1 + pp] + lfg * udata[1 + pp] * udata[3 + pp] +
00169         lfg * udata[pp] * udata[4 + pp] + lgg * udata[3 + pp] * udata[4 + pp];
00170     JMM[7] = lf + lff * Quad[1] +
00171         udata[4 + pp] * (2 * lfg * udata[1 + pp] + lgg * udata[4 + pp]);
00172     JMM[12] =
00173         lff * udata[pp] * udata[2 + pp] + lfg * udata[2 + pp] * udata[3 + pp] +
00174         lfg * udata[pp] * udata[5 + pp] + lgg * udata[3 + pp] * udata[5 + pp];
00175     JMM[13] = lff * udata[1 + pp] * udata[2 + pp] +
00176         lfg * udata[2 + pp] * udata[4 + pp] +
00177         lfg * udata[1 + pp] * udata[5 + pp] +
00178         lgg * udata[4 + pp] * udata[5 + pp];
00179     JMM[14] = lf + lff * Quad[2] +
00180         udata[5 + pp] * (2 * lfg * udata[2 + pp] + lgg * udata[5 + pp]);
00181     JMM[18] = lg + lfg * (Quad[0] - Quad[3 + 0]) +
00182         (-lff + lgg) * udata[pp] * udata[3 + pp];
00183     JMM[19] = -(udata[3 + pp] * (lff * udata[1 + pp] + lfg * udata[4 + pp])) +
00184         udata[pp] * (lfg * udata[1 + pp] + lgg * udata[4 + pp]);
00185     JMM[20] = -(udata[3 + pp] * (lff * udata[2 + pp] + lfg * udata[5 + pp])) +
00186         udata[pp] * (lfg * udata[2 + pp] + lgg * udata[5 + pp]);
00187     JMM[21] = -lf + lgg * Quad[0] +
00188         udata[3 + pp] * (-2 * lfg * udata[pp] + lff * udata[3 + pp]);
00189     JMM[24] = udata[1 + pp] * (lfg * udata[pp] + lgg * udata[3 + pp]) -
00190         (lff * udata[pp] + lfg * udata[3 + pp]) * udata[4 + pp];
00191     JMM[25] = lg + lfg * (Quad[1] - Quad[4 + 0]) +
00192         (-lff + lgg) * udata[1 + pp] * udata[4 + pp];
00193     JMM[26] = -(udata[4 + pp] * (lff * udata[2 + pp] + lfg * udata[5 + pp])) +
00194         udata[1 + pp] * (lfg * udata[2 + pp] + lgg * udata[5 + pp]);
00195     JMM[27] = lgg * udata[pp] * udata[1 + pp] +
00196         lff * udata[3 + pp] * udata[4 + pp] -
00197         lfg * (udata[1 + pp] * udata[3 + pp] + udata[pp] * udata[4 + pp]);
00198     JMM[28] = -lf + lgg * Quad[1] +
00199         udata[4 + pp] * (-2 * lfg * udata[1 + pp] + lff * udata[4 + pp]);
00200     JMM[30] = udata[2 + pp] * (lfg * udata[pp] + lgg * udata[3 + pp]) -
00201         (lff * udata[pp] + lfg * udata[3 + pp]) * udata[5 + pp];
00202     JMM[31] = udata[2 + pp] * (lfg * udata[1 + pp] + lgg * udata[4 + pp]) -
00203         (lff * udata[1 + pp] + lfg * udata[4 + pp]) * udata[5 + pp];
00204     JMM[32] = lg + lfg * (Quad[2] - Quad[5 + 0]) +
00205         (-lff + lgg) * udata[2 + pp] * udata[5 + pp];
00206     JMM[33] = lgg * udata[pp] * udata[2 + pp] +
00207         lff * udata[3 + pp] * udata[5 + pp] -
00208         lfg * (udata[2 + pp] * udata[3 + pp] + udata[pp] * udata[5 + pp]);
00209     JMM[34] =
00210         lgg * udata[1 + pp] * udata[2 + pp] +
00211         lff * udata[4 + pp] * udata[5 + pp] -
00212         lfg * (udata[2 + pp] * udata[4 + pp] + udata[1 + pp] * udata[5 + pp]);
00213     JMM[35] = -lf + lgg * Quad[2] +
00214         udata[5 + pp] * (-2 * lfg * udata[2 + pp] + lff * udata[5 + pp]);
00215     for (int i = 0; i < 6; i++) {
00216         for (int j = i + 1; j < 6; j++) {
00217             JMM[i * 6 + j] = JMM[j * 6 + i];
00218         }
00219     }
00220     // 4. Final values for temporal derivatives of field values
00221     h[0] = 0;
00222     h[1] = dxData[pp] * JMM[30] + dxData[1 + pp] * JMM[31] +
00223         dxData[2 + pp] * JMM[32] + dxData[3 + pp] * JMM[33] +
00224         dxData[4 + pp] * JMM[34] + dxData[5 + pp] * (-1 + JMM[35]);
00225     h[2] = -(dxData[pp] * JMM[24]) - dxData[1 + pp] * JMM[25] -
00226         dxData[2 + pp] * JMM[26] - dxData[3 + pp] * JMM[27] +
00227         dxData[4 + pp] * (1 - JMM[28]) - dxData[5 + pp] * JMM[29];
00228     h[3] = 0;
00229     h[4] = dxData[2 + pp];
00230     h[5] = -dxData[1 + pp];
00231     h[0] -= h[3] * JMM[3] + h[4] * JMM[4] + h[5] * JMM[5];
00232     h[1] -= h[3] * JMM[9] + h[4] * JMM[10] + h[5] * JMM[11];
00233     h[2] -= h[3] * JMM[15] + h[4] * JMM[16] + h[5] * JMM[17];
00234     // (1+2)^-1 applies only to E components
00235     dudata[pp + 0] =
00236         h[2] * (-JMM[2] * (1 + JMM[7])) + JMM[1] * JMM[8] +
00237         h[1] * (JMM[2] * JMM[13] - JMM[1] * (1 + JMM[14])) +
00238         h[0] * (1 - JMM[8] * JMM[13] + JMM[14] + JMM[7] * (1 + JMM[14]));
00239     dudata[pp + 1] =

```

```

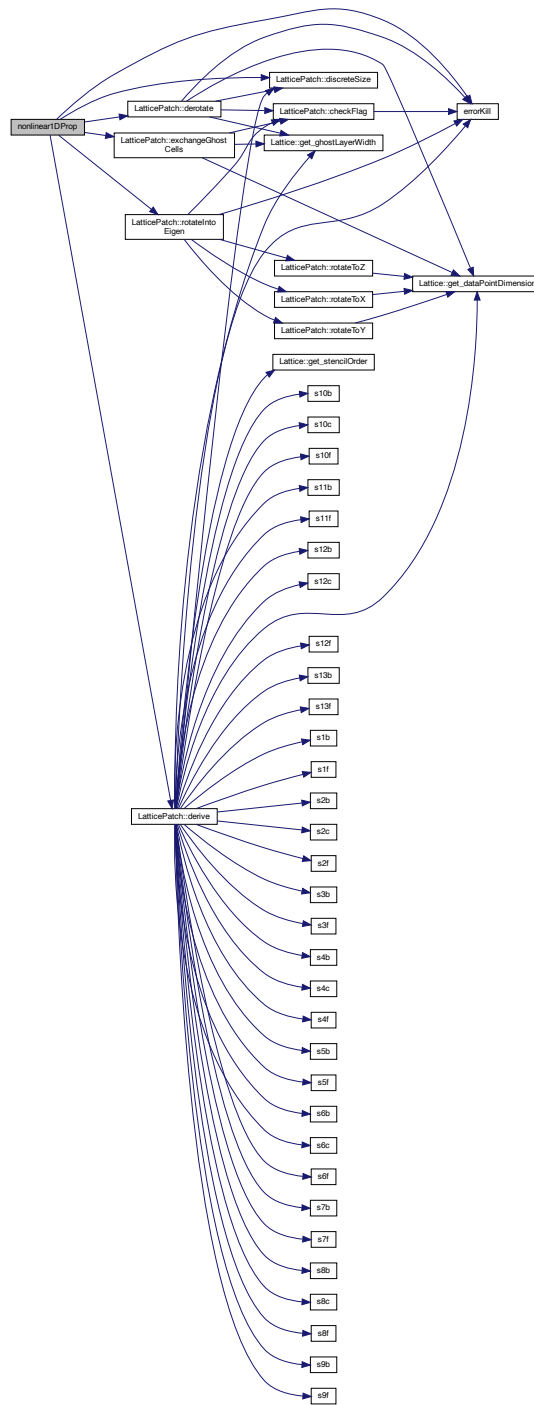
00240         h[2] * (JMM[2] * JMM[6] - (1 + JMM[0]) * JMM[8]) +
00241         h[1] * (1 - JMM[2] * JMM[12] + JMM[14] + JMM[0] * (1 + JMM[14])) +
00242         h[0] * (JMM[8] * JMM[12] - JMM[6] * (1 + JMM[14]));
00243     dudata[pp + 2] =
00244         h[2] * (1 - JMM[1] * JMM[6] + JMM[7] + JMM[0] * (1 + JMM[7])) +
00245         h[1] * (JMM[1] * JMM[12] - (1 + JMM[0]) * JMM[13]) +
00246         h[0] * (-(1 + JMM[7]) * JMM[12]) + JMM[6] * JMM[13]);
00247     pseudoDenom =
00248         -(1 + JMM[7]) * (-1 + JMM[2] * JMM[12]) +
00249         (JMM[2] * JMM[6] - JMM[8]) * JMM[13] + JMM[14] + JMM[7] * JMM[14] +
00250         JMM[0] * (1 + JMM[7] - JMM[8] * JMM[13] + (1 + JMM[7]) * JMM[14]) -
00251         JMM[1] * (-(JMM[8] * JMM[12]) + JMM[6] * (1 + JMM[14]));
00252     dudata[pp + 0] /= pseudoDenom;
00253     dudata[pp + 1] /= pseudoDenom;
00254     dudata[pp + 2] /= pseudoDenom;
00255     dudata[pp + 3] = h[3];
00256     dudata[pp + 4] = h[4];
00257     dudata[pp + 5] = h[5];
00258 }
00259 return;
00260 }

```

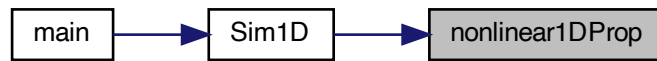
References [LatticePatch::buffData](#), [LatticePatch::derive\(\)](#), [LatticePatch::derotate\(\)](#), [LatticePatch::discreteSize\(\)](#), [errorKill\(\)](#), [LatticePatch::exchangeGhostCells\(\)](#), and [LatticePatch::rotateIntoEigen\(\)](#).

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

```

00294
00295
00296     sunrealtype *dxData = data->buffData[1 - 1];
00297     sunrealtype *dyData = data->buffData[2 - 1];
00298
00299     data->exchangeGhostCells(1);
00300     data->rotateIntoEigen(1);
00301     data->derive(1);
00302     data->derotate(1, dxData);
00303     data->exchangeGhostCells(2);
00304     data->rotateIntoEigen(2);
00305     data->derive(2);
00306     data->derotate(2, dyData);
00307
00308     sunrealtype f = NAN, g = NAN;
00309     sunrealtype lf = NAN, lff = NAN, lfg = NAN, lg = NAN, lgg = NAN;
00310     array<sunrealtype, 36> JMM;
00311     array<sunrealtype, 6> Quad;
00312     array<sunrealtype, 6> h;
00313     sunrealtype pseudoDenom = NAN;
00314     sunrealtype *udata = nullptr, *dudata = nullptr;
00315     udata = NV_DATA_P(u);
00316     dudata = NV_DATA_P(udot);
00317     int totalNP = data->discreteSize();
00318     for (int pp = 0; pp < totalNP * 6; pp += 6) {
00319         // 1
00320         f = 0.5 * ((Quad[0] = udata[pp] * udata[pp]) +
00321                 (Quad[1] = udata[pp + 1] * udata[pp + 1]) +
00322                 (Quad[2] = udata[pp + 2] * udata[pp + 2]) -
00323                 (Quad[3] = udata[pp + 3] * udata[pp + 3]) -
00324                 (Quad[4] = udata[pp + 4] * udata[pp + 4]) -
00325                 (Quad[5] = udata[pp + 5] * udata[pp + 5]));
00326         g = udata[pp] * udata[pp + 3] + udata[pp + 1] * udata[pp + 4] +
00327             udata[pp + 2] * udata[pp + 5];
00328         // 2
00329         switch (*c) {
00330             case 0:
00331                 lf = 0;
00332                 lff = 0;
00333                 lfg = 0;
00334                 lg = 0;
00335                 lgg = 0;
00336                 break;

```



```

00337     case 2:
00338         lf = 0.000354046449700427580438254 * f * f +
00339             0.000191775160254398272737387 * g * g;
00340         lff = 0.0007080928994008551608765075 * f;
00341         lfg = 0.0003835503205087965454747749 * g;
00342         lg = 0.0003835503205087965454747749 * f * g;
00343         lgg = 0.0003835503205087965454747749 * f;
00344         break;
00345     case 1:
00346         lf = 0.000206527095658582755255648 * f;
00347         lff = 0.000206527095658582755255648;
00348         lfg = 0;
00349         lg = 0.0003614224174025198216973841 * g;
00350         lgg = 0.0003614224174025198216973841;
00351         break;
00352     case 3:
00353         lf = (0.000206527095658582755255648 + 0.000354046449700427580438254 * f) *
00354             f +
00355             0.000191775160254398272737387 * g * g;
00356         lff = 0.000206527095658582755255648 + 0.000708092899400855160876508 * f;
00357         lfg = 0.0003835503205087965454747749 * g;
00358         lg = (0.000361422417402519821697384 + 0.000383550320508796545474775 * f) *
00359             g;
00360         lgg = 0.000361422417402519821697384 + 0.000383550320508796545474775 * f;
00361         break;
00362     default:
00363         errorKill(
00364             "You need to specify a correct order in the weak-field expansion.");
00365     }
00366     // 3
00367     JMM[0] = lf + lff * Quad[0] +
00368         udata[3 + pp] * (2 * lfg * udata[pp] + lgg * udata[3 + pp]);
00369     JMM[6] =
00370         lff * udata[pp] * udata[1 + pp] + lfg * udata[1 + pp] * udata[3 + pp] +
00371         lfg * udata[pp] * udata[4 + pp] + lgg * udata[3 + pp] * udata[4 + pp];
00372     JMM[7] = lf + lff * Quad[1] +
00373         udata[4 + pp] * (2 * lfg * udata[1 + pp] + lgg * udata[4 + pp]);
00374     JMM[12] =
00375         lff * udata[pp] * udata[2 + pp] + lfg * udata[2 + pp] * udata[3 + pp] +
00376         lfg * udata[pp] * udata[5 + pp] + lgg * udata[3 + pp] * udata[5 + pp];
00377     JMM[13] = lff * udata[1 + pp] * udata[2 + pp] +
00378         lfg * udata[2 + pp] * udata[4 + pp] +
00379         lfg * udata[1 + pp] * udata[5 + pp] +
00380         lgg * udata[4 + pp] * udata[5 + pp];
00381     JMM[14] = lf + lff * Quad[2] +
00382         udata[5 + pp] * (2 * lfg * udata[2 + pp] + lgg * udata[5 + pp]);
00383     JMM[18] = lg + lfg * (Quad[0] - Quad[3 + 0]) +
00384         (-lff + lgg) * udata[pp] * udata[3 + pp];
00385     JMM[19] = -(udata[3 + pp] * (lff * udata[1 + pp] + lfg * udata[4 + pp])) +
00386         udata[pp] * (lfg * udata[1 + pp] + lgg * udata[4 + pp]);
00387     JMM[20] = -(udata[3 + pp] * (lff * udata[2 + pp] + lfg * udata[5 + pp])) +
00388         udata[pp] * (lfg * udata[2 + pp] + lgg * udata[5 + pp]);
00389     JMM[21] = -lf + lgg * Quad[0] +
00390         udata[3 + pp] * (-2 * lfg * udata[pp] + lff * udata[3 + pp]);
00391     JMM[24] = udata[1 + pp] * (lfg * udata[pp] + lgg * udata[3 + pp]) -
00392         (lff * udata[pp] + lfg * udata[3 + pp]) * udata[4 + pp];
00393     JMM[25] = lg + lfg * (Quad[1] - Quad[4 + 0]) +
00394         (-lff + lgg) * udata[1 + pp] * udata[4 + pp];
00395     JMM[26] = -(udata[4 + pp] * (lff * udata[2 + pp] + lfg * udata[5 + pp])) +
00396         udata[1 + pp] * (lfg * udata[2 + pp] + lgg * udata[5 + pp]);
00397     JMM[27] = lgg * udata[pp] * udata[1 + pp] +
00398         lff * udata[3 + pp] * udata[4 + pp] -
00399         lfg * (udata[1 + pp] * udata[3 + pp] + udata[pp] * udata[4 + pp]);
00400     JMM[28] = -lf + lgg * Quad[1] +
00401         udata[4 + pp] * (-2 * lfg * udata[1 + pp] + lff * udata[4 + pp]);
00402     JMM[30] = udata[2 + pp] * (lfg * udata[pp] + lgg * udata[3 + pp]) -
00403         (lff * udata[pp] + lfg * udata[3 + pp]) * udata[5 + pp];
00404     JMM[31] = udata[2 + pp] * (lfg * udata[1 + pp] + lgg * udata[4 + pp]) -
00405         (lff * udata[1 + pp] + lfg * udata[4 + pp]) * udata[5 + pp];
00406     JMM[32] = lg + lfg * (Quad[2] - Quad[5 + 0]) +
00407         (-lff + lgg) * udata[2 + pp] * udata[5 + pp];
00408     JMM[33] = lgg * udata[pp] * udata[2 + pp] +
00409         lff * udata[3 + pp] * udata[5 + pp] -
00410         lfg * (udata[2 + pp] * udata[3 + pp] + udata[pp] * udata[5 + pp]);
00411     JMM[34] =
00412         lgg * udata[1 + pp] * udata[2 + pp] +
00413         lff * udata[4 + pp] * udata[5 + pp] -
00414         lfg * (udata[2 + pp] * udata[4 + pp] + udata[1 + pp] * udata[5 + pp]);
00415     JMM[35] = -lf + lgg * Quad[2] +
00416         udata[5 + pp] * (-2 * lfg * udata[2 + pp] + lff * udata[5 + pp]);
00417     // 4
00418     for (int i = 0; i < 6; i++) {
00419         for (int j = i + 1; j < 6; j++) {
00420             JMM[i * 6 + j] = JMM[j * 6 + i];
00421         }
00422     }
00423     h[0] = 0;

```

```

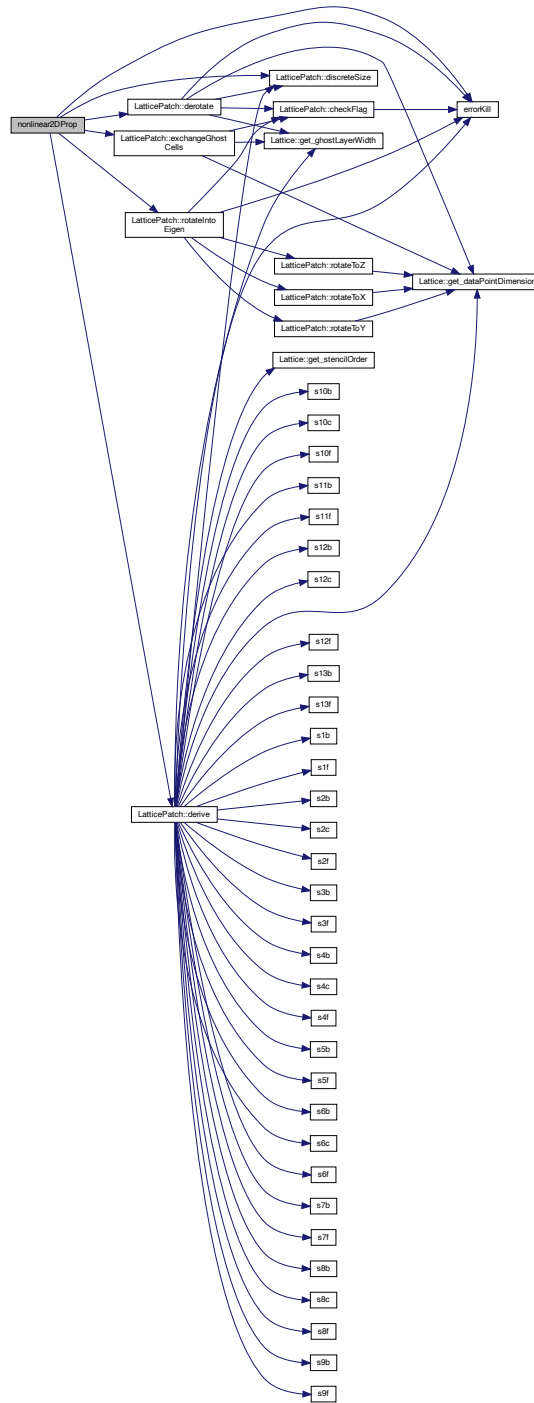
00424     h[1] = dxData[pp] * JMM[30] + dxData[1 + pp] * JMM[31] +
00425           dxData[2 + pp] * JMM[32] + dxData[3 + pp] * JMM[33] +
00426           dxData[4 + pp] * JMM[34] + dxData[5 + pp] * (-1 + JMM[35]);
00427     h[2] = -(dxData[pp] * JMM[24]) - dxData[1 + pp] * JMM[25] -
00428           dxData[2 + pp] * JMM[26] - dxData[3 + pp] * JMM[27] +
00429           dxData[4 + pp] * (1 - JMM[28]) - dxData[5 + pp] * JMM[29];
00430     h[3] = 0;
00431     h[4] = dxData[2 + pp];
00432     h[5] = -dxData[1 + pp];
00433     h[0] += -(dyData[pp] * JMM[30]) - dyData[1 + pp] * JMM[31] -
00434           dyData[2 + pp] * JMM[32] - dyData[3 + pp] * JMM[33] -
00435           dyData[4 + pp] * JMM[34] + dyData[5 + pp] * (1 - JMM[35]);
00436     h[1] += 0;
00437     h[2] += dyData[pp] * JMM[18] + dyData[1 + pp] * JMM[19] +
00438           dyData[2 + pp] * JMM[20] + dyData[3 + pp] * (-1 + JMM[21]) +
00439           dyData[4 + pp] * JMM[22] + dyData[5 + pp] * JMM[23];
00440     h[3] += -dyData[2 + pp];
00441     h[4] += 0;
00442     h[5] += dyData[pp];
00443     h[0] -= h[3] * JMM[3] + h[4] * JMM[4] + h[5] * JMM[5];
00444     h[1] -= h[3] * JMM[9] + h[4] * JMM[10] + h[5] * JMM[11];
00445     h[2] -= h[3] * JMM[15] + h[4] * JMM[16] + h[5] * JMM[17];
00446     dudata[pp + 0] =
00447         h[2] * (-(JMM[2] * (1 + JMM[7])) + JMM[1] * JMM[8]) +
00448         h[1] * (JMM[2] * JMM[13] - JMM[1] * (1 + JMM[14])) +
00449         h[0] * (1 - JMM[8] * JMM[13] + JMM[14] + JMM[7] * (1 + JMM[14]));
00450     dudata[pp + 1] =
00451         h[2] * (JMM[2] * JMM[6] - (1 + JMM[0]) * JMM[8]) +
00452         h[1] * (1 - JMM[2] * JMM[12] + JMM[14] + JMM[0] * (1 + JMM[14])) +
00453         h[0] * (JMM[8] * JMM[12] - JMM[6] * (1 + JMM[14]));
00454     dudata[pp + 2] =
00455         h[2] * (1 - JMM[1] * JMM[6] + JMM[7] + JMM[0] * (1 + JMM[7])) +
00456         h[1] * (JMM[1] * JMM[12] - (1 + JMM[0]) * JMM[13]) +
00457         h[0] * (-(1 + JMM[7]) * JMM[12]) + JMM[6] * JMM[13];
00458     pseudoDenom =
00459         -((1 + JMM[7]) * (-1 + JMM[2] * JMM[12])) +
00460         (JMM[2] * JMM[6] - JMM[8]) * JMM[13] + JMM[14] + JMM[7] * JMM[14] +
00461         JMM[0] * (1 + JMM[7] - JMM[8] * JMM[13] + (1 + JMM[7]) * JMM[14]) -
00462         JMM[1] * (-(JMM[8] * JMM[12]) + JMM[6] * (1 + JMM[14]));
00463     dudata[pp + 0] /= pseudoDenom;
00464     dudata[pp + 1] /= pseudoDenom;
00465     dudata[pp + 2] /= pseudoDenom;
00466     dudata[pp + 3] = h[3];
00467     dudata[pp + 4] = h[4];
00468     dudata[pp + 5] = h[5];
00469 }
00470 return;
00471 }

```

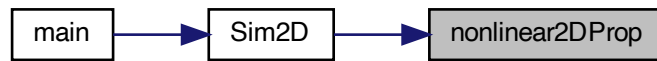
References [LatticePatch::buffData](#), [LatticePatch::derive\(\)](#), [LatticePatch::derotate\(\)](#), [LatticePatch::discreteSize\(\)](#), [errorKill\(\)](#), [LatticePatch::exchangeGhostCells\(\)](#), and [LatticePatch::rotateIntoEigen\(\)](#).

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

```

00510
00511
00512     sunrealtype *dxData = data->buffData[1 - 1];
00513     sunrealtype *dyData = data->buffData[2 - 1];
00514     sunrealtype *dzData = data->buffData[3 - 1];
00515
00516     data->exchangeGhostCells(1);
00517     data->rotateIntoEigen(1);
00518     data->derive(1);
00519     data->derotate(1,dxData);
00520     data->exchangeGhostCells(2);
00521     data->rotateIntoEigen(2);
00522     data->derive(2);
00523     data->derotate(2,dyData);
00524     data->exchangeGhostCells(3);
00525     data->rotateIntoEigen(3);
00526     data->derive(3);
00527     data->derotate(3,dzData);
00528
00529     sunrealtype f = NAN, g = NAN;
00530     sunrealtype lf = NAN, lff = NAN, lfg = NAN, lg = NAN, lgg = NAN;
00531     array<sunrealtype, 36> JMM;
00532     array<sunrealtype, 6> Quad;
00533     array<sunrealtype, 6> h;
00534     sunrealtype pseudoDenom = NAN;
00535     sunrealtype *udata = nullptr, *dudata = nullptr;
00536     udata = NV_DATA_P(u);
00537     dudata = NV_DATA_P(udot);
00538     int totalNP = data->discreteSize();
00539     for (int pp = 0; pp < totalNP * 6; pp += 6) {
00540         // 1
00541         f = 0.5 * ((Quad[0] = udata[pp] * udata[pp]) +
00542                 (Quad[1] = udata[pp + 1] * udata[pp + 1]) +
00543                 (Quad[2] = udata[pp + 2] * udata[pp + 2]) -
00544                 (Quad[3] = udata[pp + 3] * udata[pp + 3]) -
00545                 (Quad[4] = udata[pp + 4] * udata[pp + 4]) -
00546                 (Quad[5] = udata[pp + 5] * udata[pp + 5]));
00547         g = udata[pp] * udata[pp + 3] + udata[pp + 1] * udata[pp + 4] +
00548             udata[pp + 2] * udata[pp + 5];
00549         // 2
00550         switch (*c) {
00551             case 0:
00552                 lf = 0;

```

```

00553     lff = 0;
00554     lfg = 0;
00555     lg = 0;
00556     lgg = 0;
00557     break;
00558 case 2:
00559     lf = 0.000354046449700427580438254 * f * f +
00560         0.000191775160254398272737387 * g * g;
00561     lff = 0.0007080928994008551608765075 * f;
00562     lfg = 0.0003835503205087965454747749 * g;
00563     lg = 0.0003835503205087965454747749 * f * g;
00564     lgg = 0.0003835503205087965454747749 * f;
00565     break;
00566 case 1:
00567     lf = 0.000206527095658582755255648 * f;
00568     lff = 0.000206527095658582755255648;
00569     lfg = 0;
00570     lg = 0.0003614224174025198216973841 * g;
00571     lgg = 0.0003614224174025198216973841;
00572     break;
00573 case 3:
00574     lf = (0.000206527095658582755255648 + 0.000354046449700427580438254 * f) *
00575         f +
00576         0.000191775160254398272737387 * g * g;
00577     lff = 0.000206527095658582755255648 + 0.000708092899400855160876508 * f;
00578     lfg = 0.0003835503205087965454747749 * g;
00579     lg = (0.000361422417402519821697384 + 0.000383550320508796545474775 * f) *
00580         g;
00581     lgg = 0.000361422417402519821697384 + 0.000383550320508796545474775 * f;
00582     break;
00583 default:
00584     errorKill(
00585         "You need to specify a correct order in the weak-field expansion.");
00586 }
00587 // 3
00588 JMM[0] = lf + lff * Quad[0] +
00589     udata[3 + pp] * (2 * lfg * udata[pp] + lgg * udata[3 + pp]);
00590 JMM[6] =
00591     lff * udata[pp] * udata[1 + pp] + lfg * udata[1 + pp] * udata[3 + pp] +
00592     lfg * udata[pp] * udata[4 + pp] + lgg * udata[3 + pp] * udata[4 + pp];
00593 JMM[7] = lf + lff * Quad[1] +
00594     udata[4 + pp] * (2 * lfg * udata[1 + pp] + lgg * udata[4 + pp]);
00595 JMM[12] =
00596     lff * udata[pp] * udata[2 + pp] + lfg * udata[2 + pp] * udata[3 + pp] +
00597     lfg * udata[pp] * udata[5 + pp] + lgg * udata[3 + pp] * udata[5 + pp];
00598 JMM[13] = lff * udata[1 + pp] * udata[2 + pp] +
00599     lfg * udata[2 + pp] * udata[4 + pp] +
00600     lfg * udata[1 + pp] * udata[5 + pp] +
00601     lgg * udata[4 + pp] * udata[5 + pp];
00602 JMM[14] = lf + lff * Quad[2] +
00603     udata[5 + pp] * (2 * lfg * udata[2 + pp] + lgg * udata[5 + pp]);
00604 JMM[18] = lg + lfg * (Quad[0] - Quad[3 + 0]) +
00605     (-lff + lgg) * udata[pp] * udata[3 + pp];
00606 JMM[19] = -(udata[3 + pp] * (lff * udata[1 + pp] + lfg * udata[4 + pp])) +
00607     udata[pp] * (lfg * udata[1 + pp] + lgg * udata[4 + pp]);
00608 JMM[20] = -(udata[3 + pp] * (lff * udata[2 + pp] + lfg * udata[5 + pp])) +
00609     udata[pp] * (lfg * udata[2 + pp] + lgg * udata[5 + pp]);
00610 JMM[21] = -lf + lgg * Quad[0] +
00611     udata[3 + pp] * (-2 * lfg * udata[pp] + lff * udata[3 + pp]);
00612 JMM[24] = udata[1 + pp] * (lfg * udata[pp] + lgg * udata[3 + pp]) -
00613     (lff * udata[pp] + lfg * udata[3 + pp]) * udata[4 + pp];
00614 JMM[25] = lg + lfg * (Quad[1] - Quad[4 + 0]) +
00615     (-lff + lgg) * udata[1 + pp] * udata[4 + pp];
00616 JMM[26] = -(udata[4 + pp] * (lff * udata[2 + pp] + lfg * udata[5 + pp])) +
00617     udata[1 + pp] * (lfg * udata[2 + pp] + lgg * udata[5 + pp]);
00618 JMM[27] = lgg * udata[pp] * udata[1 + pp] +
00619     lff * udata[3 + pp] * udata[4 + pp] -
00620     lfg * (udata[1 + pp] * udata[3 + pp] + udata[pp] * udata[4 + pp]);
00621 JMM[28] = -lf + lgg * Quad[1] +
00622     udata[4 + pp] * (-2 * lfg * udata[1 + pp] + lff * udata[4 + pp]);
00623 JMM[30] = udata[2 + pp] * (lfg * udata[pp] + lgg * udata[3 + pp]) -
00624     (lff * udata[pp] + lfg * udata[3 + pp]) * udata[5 + pp];
00625 JMM[31] = udata[2 + pp] * (lfg * udata[1 + pp] + lgg * udata[4 + pp]) -
00626     (lff * udata[1 + pp] + lfg * udata[4 + pp]) * udata[5 + pp];
00627 JMM[32] = lg + lfg * (Quad[2] - Quad[5 + 0]) +
00628     (-lff + lgg) * udata[2 + pp] * udata[5 + pp];
00629 JMM[33] = lgg * udata[pp] * udata[2 + pp] +
00630     lff * udata[3 + pp] * udata[5 + pp] -
00631     lfg * (udata[2 + pp] * udata[3 + pp] + udata[pp] * udata[5 + pp]);
00632 JMM[34] =
00633     lgg * udata[1 + pp] * udata[2 + pp] +
00634     lff * udata[4 + pp] * udata[5 + pp] -
00635     lfg * (udata[2 + pp] * udata[4 + pp] + udata[1 + pp] * udata[5 + pp]);
00636 JMM[35] = -lf + lgg * Quad[2] +
00637     udata[5 + pp] * (-2 * lfg * udata[2 + pp] + lff * udata[5 + pp]);
00638 // 4
00639 for (int i = 0; i < 6; i++) {

```

```

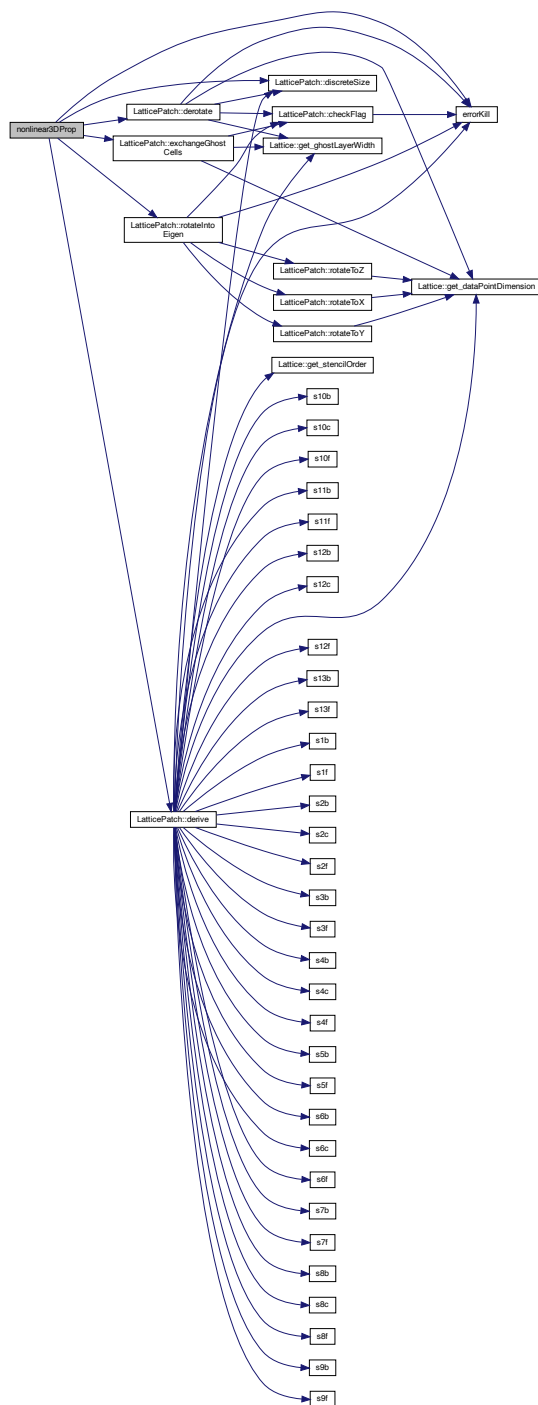
00640         for (int j = i + 1; j < 6; j++) {
00641             JMM[i * 6 + j] = JMM[j * 6 + i];
00642         }
00643     }
00644     h[0] = 0;
00645     h[1] = dxData[pp] * JMM[30] + dxData[1 + pp] * JMM[31] +
00646           dxData[2 + pp] * JMM[32] + dxData[3 + pp] * JMM[33] +
00647           dxData[4 + pp] * JMM[34] + dxData[5 + pp] * (-1 + JMM[35]);
00648     h[2] = -(dxData[pp] * JMM[24]) - dxData[1 + pp] * JMM[25] -
00649           dxData[2 + pp] * JMM[26] - dxData[3 + pp] * JMM[27] +
00650           dxData[4 + pp] * (1 - JMM[28]) - dxData[5 + pp] * JMM[29];
00651     h[3] = 0;
00652     h[4] = dxData[2 + pp];
00653     h[5] = -dxData[1 + pp];
00654     h[0] += -(dyData[pp] * JMM[30]) - dyData[1 + pp] * JMM[31] -
00655           dyData[2 + pp] * JMM[32] - dyData[3 + pp] * JMM[33] -
00656           dyData[4 + pp] * JMM[34] + dyData[5 + pp] * (1 - JMM[35]);
00657     h[1] += 0;
00658     h[2] += dyData[pp] * JMM[18] + dyData[1 + pp] * JMM[19] +
00659           dyData[2 + pp] * JMM[20] + dyData[3 + pp] * (-1 + JMM[21]) +
00660           dyData[4 + pp] * JMM[22] + dyData[5 + pp] * JMM[23];
00661     h[3] += -dyData[2 + pp];
00662     h[4] += 0;
00663     h[5] += dyData[pp];
00664     h[0] += dzData[pp] * JMM[24] + dzData[1 + pp] * JMM[25] +
00665           dzData[2 + pp] * JMM[26] + dzData[3 + pp] * JMM[27] +
00666           dzData[4 + pp] * (-1 + JMM[28]) + dzData[5 + pp] * JMM[29];
00667     h[1] += -(dzData[pp] * JMM[18]) - dzData[1 + pp] * JMM[19] -
00668           dzData[2 + pp] * JMM[20] + dzData[3 + pp] * (1 - JMM[21]) -
00669           dzData[4 + pp] * JMM[22] - dzData[5 + pp] * JMM[23];
00670     h[2] += 0;
00671     h[3] += dzData[1 + pp];
00672     h[4] += -dzData[pp];
00673     h[5] += 0;
00674     h[0] -= h[3] * JMM[3] + h[4] * JMM[4] + h[5] * JMM[5];
00675     h[1] -= h[3] * JMM[9] + h[4] * JMM[10] + h[5] * JMM[11];
00676     h[2] -= h[3] * JMM[15] + h[4] * JMM[16] + h[5] * JMM[17];
00677     dudata[pp + 0] =
00678         h[2] * (-(JMM[2] * (1 + JMM[7])) + JMM[1] * JMM[8]) +
00679         h[1] * (JMM[2] * JMM[13] - JMM[1] * (1 + JMM[14])) +
00680         h[0] * (1 - JMM[8] * JMM[13] + JMM[14] + JMM[7] * (1 + JMM[14]));
00681     dudata[pp + 1] =
00682         h[2] * (JMM[2] * JMM[6] - (1 + JMM[0]) * JMM[8]) +
00683         h[1] * (1 - JMM[2] * JMM[12] + JMM[14] + JMM[0] * (1 + JMM[14])) +
00684         h[0] * (JMM[8] * JMM[12] - JMM[6] * (1 + JMM[14]));
00685     dudata[pp + 2] =
00686         h[2] * (1 - JMM[1] * JMM[6] + JMM[7] + JMM[0] * (1 + JMM[7])) +
00687         h[1] * (JMM[1] * JMM[12] - (1 + JMM[0]) * JMM[13]) +
00688         h[0] * (-(1 + JMM[7]) * JMM[12]) + JMM[6] * JMM[13];
00689     pseudoDenom =
00690         -((1 + JMM[7]) * (-1 + JMM[2] * JMM[12])) +
00691         (JMM[2] * JMM[6] - JMM[8]) * JMM[13] + JMM[14] + JMM[7] * JMM[14] +
00692         JMM[0] * (1 + JMM[7] - JMM[8] * JMM[13] + (1 + JMM[7]) * JMM[14]) -
00693         JMM[1] * (-(JMM[8] * JMM[12]) + JMM[6] * (1 + JMM[14]));
00694     dudata[pp + 0] /= pseudoDenom;
00695     dudata[pp + 1] /= pseudoDenom;
00696     dudata[pp + 2] /= pseudoDenom;
00697     dudata[pp + 3] = h[3];
00698     dudata[pp + 4] = h[4];
00699     dudata[pp + 5] = h[5];
00700 }
00701 return;
00702 }

```

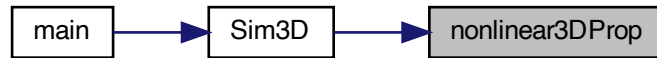
References [LatticePatch::buffData](#), [LatticePatch::derive\(\)](#), [LatticePatch::derotate\(\)](#), [LatticePatch::discreteSize\(\)](#), [errorKill\(\)](#), [LatticePatch::exchangeGhostCells\(\)](#), and [LatticePatch::rotateIntoEigen\(\)](#).

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 // Include Guard
00009 #ifndef TIMEEVOLVER
00010 #define TIMEEVOLVER
00011
00012 #include "LatticePatch.h"
00013 #include "SimulationClass.h"
00014
00015 /** @brief monostate TimeEvolution Class to propagate the field data in time in
00016  * a given order of the HE weak-field expansion */
00017 class TimeEvolution {
00018 public:
00019     /// choice which processes of the weak field expansion are included
00020     static int *c;
00021
00022     /// Pointer to functions for differentiation and time evolution
00023     static void (*TimeEvolver)(LatticePatch *, N_Vector, N_Vector, int *);
00024
00025     /// CVODE right hand side function (CVRhsFn) to provide IVP of the ODE
00026     static int f(sunrealtype t, N_Vector u, N_Vector udot, void *data_loc);
00027 };
00028
00029 /// Maxwell propagation function for 1D - only for reference
00030 void linear1DProp(LatticePatch *data, N_Vector u, N_Vector udot, int *c);
00031 /// HE propagation function for 1D
00032 void nonlinear1DProp(LatticePatch *data, N_Vector u, N_Vector udot, int *c);
00033 /// Maxwell propagation function for 2D - only for reference
00034 void linear2DProp(LatticePatch *data, N_Vector u, N_Vector udot, int *c);
00035 /// HE propagation function for 2D
00036 void nonlinear2DProp(LatticePatch *data, N_Vector u, N_Vector udot, int *c);
00037 /// Maxwell propagation function for 3D - only for reference
00038 void linear3DProp(LatticePatch *data, N_Vector u, N_Vector udot, int *c);
00039 /// HE propagation function for 3D
00040 void nonlinear3DProp(LatticePatch *data, N_Vector u, N_Vector udot, int *c);
00041
00042 // End of Includeguard
00043 #endif
  
```


Index

- ~LatticePatch
 - LatticePatch, [64](#)
- ~Simulation
 - Simulation, [118](#)
- A1
 - Gauss2D, [22](#)
 - Gauss3D, [27](#)
- A2
 - Gauss2D, [22](#)
 - Gauss3D, [28](#)
- add
 - ICSetter, [37](#)
- addGauss1D
 - ICSetter, [38](#)
- addGauss2D
 - ICSetter, [39](#)
- addGauss3D
 - ICSetter, [39](#)
- addInitialConditions
 - Simulation, [118](#)
- addPeriodicICLayerInX
 - Simulation, [119](#)
- addPeriodicICLayerInXY
 - Simulation, [120](#)
- addPlaneWave1D
 - ICSetter, [40](#)
- addPlaneWave2D
 - ICSetter, [41](#)
- addPlaneWave3D
 - ICSetter, [41](#)
- addToSpace
 - Gauss1D, [15](#)
 - Gauss2D, [21](#)
 - Gauss3D, [27](#)
 - PlaneWave1D, [109](#)
 - PlaneWave2D, [112](#)
 - PlaneWave3D, [114](#)
- advanceToTime
 - Simulation, [121](#)
- Amp
 - Gauss2D, [22](#)
 - Gauss3D, [28](#)
- amp
 - gaussian2D, [33](#)
 - gaussian3D, [35](#)
- axis
 - Gauss2D, [22](#)
 - Gauss3D, [28](#)
 - gaussian2D, [33](#)
 - gaussian3D, [35](#)
- buffData
 - LatticePatch, [85](#)
- BuffersInitialized
 - LatticePatch.h, [204](#)
- buffX
 - LatticePatch, [85](#)
- buffY
 - LatticePatch, [86](#)
- buffZ
 - LatticePatch, [86](#)
- c
 - TimeEvolution, [137](#)
- check_retval
 - LatticePatch.cpp, [188](#)
 - LatticePatch.h, [204](#)
- checkFlag
 - LatticePatch, [64](#)
 - Simulation, [122](#)
- checkNoFlag
 - Simulation, [124](#)
- comm
 - Lattice, [55](#)
- cnode_mem
 - Simulation, [133](#)
- CvodeObjectSetUp
 - SimulationClass.h, [231](#)
- dataPointDimension
 - Lattice, [55](#)
- DerivationStencils.h
 - s10b, [142](#)
 - s10c, [143](#)
 - s10f, [144](#)
 - s11b, [145](#)
 - s11f, [146](#), [147](#)
 - s12b, [147](#), [148](#)
 - s12c, [148](#), [149](#)
 - s12f, [149](#), [150](#)
 - s13b, [150](#), [151](#)
 - s13f, [151](#), [152](#)
 - s1b, [152](#), [153](#)
 - s1f, [153](#), [154](#)
 - s2b, [154](#), [155](#)
 - s2c, [155](#), [156](#)
 - s2f, [156](#), [157](#)
 - s3b, [157](#), [158](#)
 - s3f, [158](#), [159](#)

- s4b, [159](#), [160](#)
- s4c, [160](#), [161](#)
- s4f, [161](#), [162](#)
- s5b, [162](#), [163](#)
- s5f, [163](#), [164](#)
- s6b, [164](#), [165](#)
- s6c, [165](#), [166](#)
- s6f, [166](#), [167](#)
- s7b, [167](#), [168](#)
- s7f, [168](#), [169](#)
- s8b, [169](#), [170](#)
- s8c, [170](#), [171](#)
- s8f, [171](#), [172](#)
- s9b, [172](#), [173](#)
- s9f, [173](#), [174](#)
- derive
 - LatticePatch, [66](#)
- derotate
 - LatticePatch, [70](#)
- dis
 - Gauss2D, [23](#)
 - Gauss3D, [28](#)
- discreteSize
 - LatticePatch, [72](#)
- du
 - LatticePatch, [86](#)
- duData
 - LatticePatch, [86](#)
- dx
 - Lattice, [55](#)
 - LatticePatch, [87](#)
- dy
 - Lattice, [55](#)
 - LatticePatch, [87](#)
- dz
 - Lattice, [56](#)
 - LatticePatch, [87](#)
- envelopeLattice
 - LatticePatch, [87](#)
- errorKill
 - LatticePatch.cpp, [189](#)
 - LatticePatch.h, [205](#)
- eval
 - ICSetter, [42](#)
- exchangeGhostCells
 - LatticePatch, [73](#)
- f
 - TimeEvolution, [136](#)
- FLatticeDimensionSet
 - LatticePatch.h, [203](#)
- FLatticePatchSetUp
 - LatticePatch.h, [204](#)
- Gauss1D, [13](#)
 - addToSpace, [15](#)
 - Gauss1D, [14](#)
 - kx, [16](#)
 - ky, [16](#)
 - kz, [16](#)
 - phig, [16](#)
 - phix, [16](#)
 - phiy, [17](#)
 - phiz, [17](#)
 - px, [17](#)
 - py, [17](#)
 - pz, [18](#)
 - x0x, [18](#)
 - x0y, [18](#)
 - x0z, [18](#)
- gauss1Ds
 - ICSetter, [43](#)
- Gauss2D, [19](#)
 - A1, [22](#)
 - A2, [22](#)
 - addToSpace, [21](#)
 - Amp, [22](#)
 - axis, [22](#)
 - dis, [23](#)
 - Gauss2D, [20](#)
 - lambda, [23](#)
 - Ph0, [23](#)
 - PhA, [23](#)
 - phip, [24](#)
 - w0, [24](#)
 - zr, [24](#)
- gauss2Ds
 - ICSetter, [43](#)
- Gauss3D, [25](#)
 - A1, [27](#)
 - A2, [28](#)
 - addToSpace, [27](#)
 - Amp, [28](#)
 - axis, [28](#)
 - dis, [28](#)
 - Gauss3D, [26](#)
 - lambda, [29](#)
 - Ph0, [29](#)
 - PhA, [29](#)
 - phip, [29](#)
 - w0, [30](#)
 - zr, [30](#)
- gauss3Ds
 - ICSetter, [43](#)
- gaussian1D, [30](#)
 - k, [31](#)
 - p, [31](#)
 - phi, [31](#)
 - phig, [31](#)
 - x0, [32](#)
- gaussian2D, [32](#)
 - amp, [33](#)
 - axis, [33](#)
 - ph0, [33](#)
 - PhA, [33](#)
 - phip, [33](#)

- w0, [33](#)
- x0, [34](#)
- zr, [34](#)
- gaussian3D, [34](#)
 - amp, [35](#)
 - axis, [35](#)
 - ph0, [35](#)
 - phA, [35](#)
 - phip, [35](#)
 - w0, [36](#)
 - x0, [36](#)
 - zr, [36](#)
- gCLData
 - LatticePatch, [88](#)
- gCRData
 - LatticePatch, [88](#)
- generateOutputFolder
 - OutputManager, [98](#)
- generatePatchwork
 - LatticePatch, [84](#)
 - LatticePatch.cpp, [190](#)
- generateTranslocationLookup
 - LatticePatch, [75](#)
- get_cart_comm
 - Simulation, [125](#)
- get_dataPointDimension
 - Lattice, [47](#)
- get_dx
 - Lattice, [47](#)
- get_dy
 - Lattice, [48](#)
- get_dz
 - Lattice, [48](#)
- get_ghostLayerWidth
 - Lattice, [48](#)
- get_stencilOrder
 - Lattice, [49](#)
- get_tot_lx
 - Lattice, [49](#)
- get_tot_ly
 - Lattice, [50](#)
- get_tot_lz
 - Lattice, [50](#)
- get_tot_noDP
 - Lattice, [51](#)
- get_tot_noP
 - Lattice, [51](#)
- get_tot_nx
 - Lattice, [51](#)
- get_tot_ny
 - Lattice, [52](#)
- get_tot_nz
 - Lattice, [52](#)
- getDelta
 - LatticePatch, [76](#)
- getSimCode
 - OutputManager, [99](#)
- ghostCellLeft
 - LatticePatch, [88](#)
- ghostCellLeftToSend
 - LatticePatch, [88](#)
- ghostCellRight
 - LatticePatch, [89](#)
- ghostCellRightToSend
 - LatticePatch, [89](#)
- ghostCells
 - LatticePatch, [89](#)
- ghostCellsToSend
 - LatticePatch, [89](#)
- GhostLayersInitialized
 - LatticePatch.h, [204](#)
- ghostLayerWidth
 - Lattice, [56](#)
- ICSetter, [36](#)
 - add, [37](#)
 - addGauss1D, [38](#)
 - addGauss2D, [39](#)
 - addGauss3D, [39](#)
 - addPlaneWave1D, [40](#)
 - addPlaneWave2D, [41](#)
 - addPlaneWave3D, [41](#)
 - eval, [42](#)
 - gauss1Ds, [43](#)
 - gauss2Ds, [43](#)
 - gauss3Ds, [43](#)
 - planeWaves1D, [44](#)
 - planeWaves2D, [44](#)
 - planeWaves3D, [44](#)
- icsettings
 - Simulation, [133](#)
- ID
 - LatticePatch, [89](#)
- initializeBuffers
 - LatticePatch, [77](#)
- initializeCommunicator
 - Lattice, [52](#)
- initializeCVODEobject
 - Simulation, [125](#)
- initializeGhostLayer
 - LatticePatch, [78](#)
- initializePatchwork
 - Simulation, [127](#)
- k
 - gaussian1D, [31](#)
 - planewave, [106](#)
- kx
 - Gauss1D, [16](#)
 - PlaneWave, [104](#)
- ky
 - Gauss1D, [16](#)
 - PlaneWave, [104](#)
- kz
 - Gauss1D, [16](#)
 - PlaneWave, [104](#)

- lambda
 - Gauss2D, [23](#)
 - Gauss3D, [29](#)
- Lattice, [45](#)
 - comm, [55](#)
 - dataPointDimension, [55](#)
 - dx, [55](#)
 - dy, [55](#)
 - dz, [56](#)
 - get_dataPointDimension, [47](#)
 - get_dx, [47](#)
 - get_dy, [48](#)
 - get_dz, [48](#)
 - get_ghostLayerWidth, [48](#)
 - get_stencilOrder, [49](#)
 - get_tot_lx, [49](#)
 - get_tot_ly, [50](#)
 - get_tot_lz, [50](#)
 - get_tot_noDP, [51](#)
 - get_tot_noP, [51](#)
 - get_tot_nx, [51](#)
 - get_tot_ny, [52](#)
 - get_tot_nz, [52](#)
 - ghostLayerWidth, [56](#)
 - initializeCommunicator, [52](#)
 - Lattice, [46](#)
 - my_prc, [56](#)
 - n_prc, [56](#)
 - profobj, [57](#)
 - setDiscreteDimensions, [53](#)
 - setPhysicalDimensions, [54](#)
 - statusFlags, [57](#)
 - stencilOrder, [57](#)
 - sunctx, [57](#)
 - tot_lx, [58](#)
 - tot_ly, [58](#)
 - tot_lz, [58](#)
 - tot_noDP, [58](#)
 - tot_noP, [59](#)
 - tot_nx, [59](#)
 - tot_ny, [59](#)
 - tot_nz, [59](#)
- lattice
 - Simulation, [134](#)
- LatticeDiscreteSetUp
 - SimulationClass.h, [231](#)
- LatticeOptions
 - LatticePatch.h, [203](#)
- LatticePatch, [60](#)
 - ~LatticePatch, [64](#)
 - buffData, [85](#)
 - buffX, [85](#)
 - buffY, [86](#)
 - buffZ, [86](#)
 - checkFlag, [64](#)
 - derive, [66](#)
 - derotate, [70](#)
 - discreteSize, [72](#)
 - du, [86](#)
 - duData, [86](#)
 - dx, [87](#)
 - dy, [87](#)
 - dz, [87](#)
 - envelopeLattice, [87](#)
 - exchangeGhostCells, [73](#)
 - gCLData, [88](#)
 - gCRData, [88](#)
 - generatePatchwork, [84](#)
 - generateTranslocationLookup, [75](#)
 - getDelta, [76](#)
 - ghostCellLeft, [88](#)
 - ghostCellLeftToSend, [88](#)
 - ghostCellRight, [89](#)
 - ghostCellRightToSend, [89](#)
 - ghostCells, [89](#)
 - ghostCellsToSend, [89](#)
 - ID, [89](#)
 - initializeBuffers, [77](#)
 - initializeGhostLayer, [78](#)
 - LatticePatch, [63](#)
 - lgcTox, [90](#)
 - lgcToy, [90](#)
 - lgcToz, [90](#)
 - Llx, [90](#)
 - Lly, [91](#)
 - Llz, [91](#)
 - lx, [91](#)
 - ly, [91](#)
 - lz, [92](#)
 - nx, [92](#)
 - ny, [92](#)
 - nz, [92](#)
 - origin, [78](#)
 - rgcTox, [93](#)
 - rgcToy, [93](#)
 - rgcToz, [93](#)
 - rotateIntoEigen, [79](#)
 - rotateToX, [81](#)
 - rotateToY, [82](#)
 - rotateToZ, [83](#)
 - statusFlags, [93](#)
 - u, [94](#)
 - uAux, [94](#)
 - uAuxData, [94](#)
 - uData, [94](#)
 - uTox, [95](#)
 - uToy, [95](#)
 - uToz, [95](#)
 - x0, [95](#)
 - xTou, [96](#)
 - y0, [96](#)
 - yTou, [96](#)
 - z0, [96](#)
 - zTou, [97](#)
- latticePatch
 - Simulation, [134](#)

- LatticePatch.cpp
 - check_retval, 188
 - errorKill, 189
 - generatePatchwork, 190
- LatticePatch.h
 - BuffersInitialized, 204
 - check_retval, 204
 - errorKill, 205
 - FLatticeDimensionSet, 203
 - FLatticePatchSetUp, 204
 - GhostLayersInitialized, 204
 - LatticeOptions, 203
 - LatticePatchOptions, 203
 - TranslocationLookupSetUp, 204
- LatticePatchOptions
 - LatticePatch.h, 203
- LatticePatchworkSetUp
 - SimulationClass.h, 231
- LatticePhysicalSetUp
 - SimulationClass.h, 231
- lgcTox
 - LatticePatch, 90
- lgcToy
 - LatticePatch, 90
- lgcToz
 - LatticePatch, 90
- linear1DProp
 - TimeEvolutionFunctions.cpp, 259
 - TimeEvolutionFunctions.h, 287
- linear2DProp
 - TimeEvolutionFunctions.cpp, 261
 - TimeEvolutionFunctions.h, 289
- linear3DProp
 - TimeEvolutionFunctions.cpp, 263
 - TimeEvolutionFunctions.h, 291
- Llx
 - LatticePatch, 90
- Lly
 - LatticePatch, 91
- Llz
 - LatticePatch, 91
- lx
 - LatticePatch, 91
- ly
 - LatticePatch, 91
- lz
 - LatticePatch, 92
- main
 - main.cpp, 210
- main.cpp
 - main, 210
- my_prc
 - Lattice, 56
- myPrc
 - OutputManager, 102
- n_prc
 - Lattice, 56
- nonlinear1DProp
 - TimeEvolutionFunctions.cpp, 265
 - TimeEvolutionFunctions.h, 293
- nonlinear2DProp
 - TimeEvolutionFunctions.cpp, 270
 - TimeEvolutionFunctions.h, 298
- nonlinear3DProp
 - TimeEvolutionFunctions.cpp, 274
 - TimeEvolutionFunctions.h, 302
- nx
 - LatticePatch, 92
- ny
 - LatticePatch, 92
- nz
 - LatticePatch, 92
- origin
 - LatticePatch, 78
- outAllFieldData
 - Simulation, 128
- OutputManager, 97
 - generateOutputFolder, 98
 - getSimCode, 99
 - myPrc, 102
 - OutputManager, 98
 - outUState, 100
 - Path, 102
 - simCode, 102
 - SimCodeGenerator, 101
- outputManager
 - Simulation, 134
- outUState
 - OutputManager, 100
- p
 - gaussian1D, 31
 - planewave, 106
- Path
 - OutputManager, 102
- Ph0
 - Gauss2D, 23
 - Gauss3D, 29
- ph0
 - gaussian2D, 33
 - gaussian3D, 35
- PhA
 - Gauss2D, 23
 - Gauss3D, 29
- phA
 - gaussian2D, 33
 - gaussian3D, 35
- phi
 - gaussian1D, 31
 - planewave, 107
- phig
 - Gauss1D, 16
 - gaussian1D, 31
- phip
 - Gauss2D, 24

- Gauss3D, [29](#)
- gaussian2D, [33](#)
- gaussian3D, [35](#)
- phix
 - Gauss1D, [16](#)
 - PlaneWave, [104](#)
- phiy
 - Gauss1D, [17](#)
 - PlaneWave, [104](#)
- phiz
 - Gauss1D, [17](#)
 - PlaneWave, [105](#)
- PlaneWave, [103](#)
 - kx, [104](#)
 - ky, [104](#)
 - kz, [104](#)
 - phix, [104](#)
 - phiy, [104](#)
 - phiz, [105](#)
 - px, [105](#)
 - py, [105](#)
 - pz, [105](#)
- planewave, [106](#)
 - k, [106](#)
 - p, [106](#)
 - phi, [107](#)
- PlaneWave1D, [107](#)
 - addToSpace, [109](#)
 - PlaneWave1D, [108](#)
- PlaneWave2D, [110](#)
 - addToSpace, [112](#)
 - PlaneWave2D, [111](#)
- PlaneWave3D, [112](#)
 - addToSpace, [114](#)
 - PlaneWave3D, [113](#)
- planeWaves1D
 - ICSetter, [44](#)
- planeWaves2D
 - ICSetter, [44](#)
- planeWaves3D
 - ICSetter, [44](#)
- profobj
 - Lattice, [57](#)
- px
 - Gauss1D, [17](#)
 - PlaneWave, [105](#)
- py
 - Gauss1D, [17](#)
 - PlaneWave, [105](#)
- pz
 - Gauss1D, [18](#)
 - PlaneWave, [105](#)
- README.md, [139](#)
- rgcTox
 - LatticePatch, [93](#)
- rgcToy
 - LatticePatch, [93](#)
- rgcToz
 - LatticePatch, [93](#)
- rotateIntoEigen
 - LatticePatch, [79](#)
- rotateToX
 - LatticePatch, [81](#)
- rotateToY
 - LatticePatch, [82](#)
- rotateToZ
 - LatticePatch, [83](#)
- s10b
 - DerivationStencils.h, [142](#)
- s10c
 - DerivationStencils.h, [143](#)
- s10f
 - DerivationStencils.h, [144](#)
- s11b
 - DerivationStencils.h, [145](#)
- s11f
 - DerivationStencils.h, [146](#), [147](#)
- s12b
 - DerivationStencils.h, [147](#), [148](#)
- s12c
 - DerivationStencils.h, [148](#), [149](#)
- s12f
 - DerivationStencils.h, [149](#), [150](#)
- s13b
 - DerivationStencils.h, [150](#), [151](#)
- s13f
 - DerivationStencils.h, [151](#), [152](#)
- s1b
 - DerivationStencils.h, [152](#), [153](#)
- s1f
 - DerivationStencils.h, [153](#), [154](#)
- s2b
 - DerivationStencils.h, [154](#), [155](#)
- s2c
 - DerivationStencils.h, [155](#), [156](#)
- s2f
 - DerivationStencils.h, [156](#), [157](#)
- s3b
 - DerivationStencils.h, [157](#), [158](#)
- s3f
 - DerivationStencils.h, [158](#), [159](#)
- s4b
 - DerivationStencils.h, [159](#), [160](#)
- s4c
 - DerivationStencils.h, [160](#), [161](#)
- s4f
 - DerivationStencils.h, [161](#), [162](#)
- s5b
 - DerivationStencils.h, [162](#), [163](#)
- s5f
 - DerivationStencils.h, [163](#), [164](#)
- s6b
 - DerivationStencils.h, [164](#), [165](#)
- s6c
 - DerivationStencils.h, [165](#), [166](#)
- s6f

- DerivationStencils.h, [166](#), [167](#)
- s7b
 - DerivationStencils.h, [167](#), [168](#)
- s7f
 - DerivationStencils.h, [168](#), [169](#)
- s8b
 - DerivationStencils.h, [169](#), [170](#)
- s8c
 - DerivationStencils.h, [170](#), [171](#)
- s8f
 - DerivationStencils.h, [171](#), [172](#)
- s9b
 - DerivationStencils.h, [172](#), [173](#)
- s9f
 - DerivationStencils.h, [173](#), [174](#)
- setDiscreteDimensions
 - Lattice, [53](#)
- setDiscreteDimensionsOfLattice
 - Simulation, [129](#)
- setInitialConditions
 - Simulation, [130](#)
- setPhysicalDimensions
 - Lattice, [54](#)
- setPhysicalDimensionsOfLattice
 - Simulation, [131](#)
- Sim1D
 - SimulationFunctions.cpp, [233](#)
 - SimulationFunctions.h, [248](#)
- Sim2D
 - SimulationFunctions.cpp, [237](#)
 - SimulationFunctions.h, [251](#)
- Sim3D
 - SimulationFunctions.cpp, [240](#)
 - SimulationFunctions.h, [254](#)
- simCode
 - OutputManager, [102](#)
- SimCodeGenerator
 - OutputManager, [101](#)
- Simulation, [115](#)
 - ~Simulation, [118](#)
 - addInitialConditions, [118](#)
 - addPeriodicICLayerInX, [119](#)
 - addPeriodicICLayerInXY, [120](#)
 - advanceToTime, [121](#)
 - checkFlag, [122](#)
 - checkNoFlag, [124](#)
 - cvmem, [133](#)
 - get_cart_comm, [125](#)
 - icsettings, [133](#)
 - initializeCVODEobject, [125](#)
 - initializePatchwork, [127](#)
 - lattice, [134](#)
 - latticePatch, [134](#)
 - outAllFieldData, [128](#)
 - outputManager, [134](#)
 - setDiscreteDimensionsOfLattice, [129](#)
 - setInitialConditions, [130](#)
 - setPhysicalDimensionsOfLattice, [131](#)
 - Simulation, [117](#)
 - start, [132](#)
 - statusFlags, [134](#)
 - t, [135](#)
- SimulationClass.h
 - CvmemObjectSetUp, [231](#)
 - LatticeDiscreteSetUp, [231](#)
 - LatticePatchworkSetUp, [231](#)
 - LatticePhysicalSetUp, [231](#)
 - SimulationOptions, [231](#)
 - SimulationStarted, [231](#)
- SimulationFunctions.cpp
 - Sim1D, [233](#)
 - Sim2D, [237](#)
 - Sim3D, [240](#)
 - timer, [243](#)
- SimulationFunctions.h
 - Sim1D, [248](#)
 - Sim2D, [251](#)
 - Sim3D, [254](#)
 - timer, [257](#)
- SimulationOptions
 - SimulationClass.h, [231](#)
- SimulationStarted
 - SimulationClass.h, [231](#)
- src/DerivationStencils.cpp, [139](#), [140](#)
- src/DerivationStencils.h, [140](#), [175](#)
- src/ICSetters.cpp, [178](#)
- src/ICSetters.h, [182](#), [184](#)
- src/LatticePatch.cpp, [187](#), [192](#)
- src/LatticePatch.h, [202](#), [206](#)
- src/main.cpp, [209](#), [218](#)
- src/Outputters.cpp, [221](#), [222](#)
- src/Outputters.h, [223](#), [225](#)
- src/SimulationClass.cpp, [225](#), [226](#)
- src/SimulationClass.h, [229](#), [231](#)
- src/SimulationFunctions.cpp, [232](#), [244](#)
- src/SimulationFunctions.h, [247](#), [258](#)
- src/TimeEvolutionFunctions.cpp, [259](#), [278](#)
- src/TimeEvolutionFunctions.h, [286](#), [306](#)
- start
 - Simulation, [132](#)
- statusFlags
 - Lattice, [57](#)
 - LatticePatch, [93](#)
 - Simulation, [134](#)
- stencilOrder
 - Lattice, [57](#)
- sunctx
 - Lattice, [57](#)
- t
 - Simulation, [135](#)
- TimeEvolution, [135](#)
 - c, [137](#)
 - f, [136](#)
 - TimeEvolver, [137](#)
- TimeEvolutionFunctions.cpp
 - linear1DProp, [259](#)

- linear2DProp, [261](#)
 - linear3DProp, [263](#)
 - nonlinear1DProp, [265](#)
 - nonlinear2DProp, [270](#)
 - nonlinear3DProp, [274](#)
- TimeEvolutionFunctions.h
 - linear1DProp, [287](#)
 - linear2DProp, [289](#)
 - linear3DProp, [291](#)
 - nonlinear1DProp, [293](#)
 - nonlinear2DProp, [298](#)
 - nonlinear3DProp, [302](#)
- TimeEvolver
 - TimeEvolution, [137](#)
- timer
 - SimulationFunctions.cpp, [243](#)
 - SimulationFunctions.h, [257](#)
- tot_lx
 - Lattice, [58](#)
- tot_ly
 - Lattice, [58](#)
- tot_lz
 - Lattice, [58](#)
- tot_noDP
 - Lattice, [58](#)
- tot_noP
 - Lattice, [59](#)
- tot_nx
 - Lattice, [59](#)
- tot_ny
 - Lattice, [59](#)
- tot_nz
 - Lattice, [59](#)
- TranslocationLookupSetUp
 - LatticePatch.h, [204](#)
- u
 - LatticePatch, [94](#)
- uAux
 - LatticePatch, [94](#)
- uAuxData
 - LatticePatch, [94](#)
- uData
 - LatticePatch, [94](#)
- uTox
 - LatticePatch, [95](#)
- uToy
 - LatticePatch, [95](#)
- uToz
 - LatticePatch, [95](#)
- w0
 - Gauss2D, [24](#)
 - Gauss3D, [30](#)
 - gaussian2D, [33](#)
 - gaussian3D, [36](#)
- x0
 - gaussian1D, [32](#)
- gaussian2D, [34](#)
 - gaussian3D, [36](#)
 - LatticePatch, [95](#)
- x0x
 - Gauss1D, [18](#)
- x0y
 - Gauss1D, [18](#)
- x0z
 - Gauss1D, [18](#)
- xTou
 - LatticePatch, [96](#)
- y0
 - LatticePatch, [96](#)
- yTou
 - LatticePatch, [96](#)
- z0
 - LatticePatch, [96](#)
- zr
 - Gauss2D, [24](#)
 - Gauss3D, [30](#)
 - gaussian2D, [34](#)
 - gaussian3D, [36](#)
- zTou
 - LatticePatch, [97](#)