

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

5.9.2.2 ~LatticePatch()	64
5.9.3 Member Function Documentation	64
5.9.3.1 bufferCreator()	65
5.9.3.2 checkFlag()	66
5.9.3.3 derive()	67
5.9.3.4 derotate()	72
5.9.3.5 discreteSize()	74
5.9.3.6 exchangeGhostCells()	75
5.9.3.7 exchangeGhostCells3D()	77
5.9.3.8 generateTranslocationLookup()	78
5.9.3.9 getDelta()	80
5.9.3.10 initializeBuffers()	81
5.9.3.11 initializeGhostLayer()	81
5.9.3.12 origin()	82
5.9.3.13 rotateIntoEigen()	83
5.9.3.14 rotateIntoEigen3D()	84
5.9.3.15 rotateToX()	85
5.9.3.16 rotateToY()	86
5.9.3.17 rotateToZ()	87
5.9.4 Friends And Related Function Documentation	88
5.9.4.1 generatePatchwork	88
5.9.5 Field Documentation	89
5.9.5.1 buffData	89
5.9.5.2 buffX	89
5.9.5.3 buffY	89
5.9.5.4 buffZ	90
5.9.5.5 du	90
5.9.5.6 duData	90
5.9.5.7 dx	90
5.9.5.8 dy	91
5.9.5.9 dz	91
5.9.5.10 envelopeLattice	91
5.9.5.11 gCAData	91
5.9.5.12 gCBData	92
5.9.5.13 gCFData	92
5.9.5.14 gCLData	92
5.9.5.15 gCRData	92
5.9.5.16 gCTData	93
5.9.5.17 ghostCellLeft	93
5.9.5.18 ghostCellLeftToSend	93
5.9.5.19 ghostCellRight	93
5.9.5.20 ghostCellRightToSend	94

5.9.5.21 ghostCells	94
5.9.5.22 ghostCellsToSend	94
5.9.5.23 ID	94
5.9.5.24 lgcTox	95
5.9.5.25 lgcToy	95
5.9.5.26 lgcToz	95
5.9.5.27 Llx	95
5.9.5.28 Lly	96
5.9.5.29 Llz	96
5.9.5.30 lx	96
5.9.5.31 ly	96
5.9.5.32 lz	97
5.9.5.33 nx	97
5.9.5.34 ny	97
5.9.5.35 nz	97
5.9.5.36 rgcTox	98
5.9.5.37 rgcToy	98
5.9.5.38 rgcToz	98
5.9.5.39 statusFlags	98
5.9.5.40 u	99
5.9.5.41 uAux	99
5.9.5.42 uAuxData	99
5.9.5.43 uData	99
5.9.5.44 uTox	100
5.9.5.45 uToy	100
5.9.5.46 uToz	100
5.9.5.47 x0	100
5.9.5.48 xTou	101
5.9.5.49 y0	101
5.9.5.50 yTou	101
5.9.5.51 z0	101
5.9.5.52 zTou	102
5.10 OutputManager Class Reference	102
5.10.1 Detailed Description	103
5.10.2 Constructor & Destructor Documentation	103
5.10.2.1 OutputManager()	103
5.10.3 Member Function Documentation	103
5.10.3.1 generateOutputFolder()	104
5.10.3.2 getSimCode()	105
5.10.3.3 outUState()	105
5.10.3.4 set_outputStyle()	107
5.10.3.5 SimCodeGenerator()	108

5.10.4 Field Documentation	108
5.10.4.1 outputStyle	108
5.10.4.2 Path	109
5.10.4.3 simCode	109
5.11 PlaneWave Class Reference	109
5.11.1 Detailed Description	110
5.11.2 Field Documentation	110
5.11.2.1 kx	110
5.11.2.2 ky	111
5.11.2.3 kz	111
5.11.2.4 phix	111
5.11.2.5 phiy	111
5.11.2.6 phiz	112
5.11.2.7 px	112
5.11.2.8 py	112
5.11.2.9 pz	112
5.12 planewave Struct Reference	113
5.12.1 Detailed Description	113
5.12.2 Field Documentation	113
5.12.2.1 k	113
5.12.2.2 p	113
5.12.2.3 phi	114
5.13 PlaneWave1D Class Reference	114
5.13.1 Detailed Description	115
5.13.2 Constructor & Destructor Documentation	115
5.13.2.1 PlaneWave1D()	115
5.13.3 Member Function Documentation	116
5.13.3.1 addToSpace()	116
5.14 PlaneWave2D Class Reference	116
5.14.1 Detailed Description	117
5.14.2 Constructor & Destructor Documentation	117
5.14.2.1 PlaneWave2D()	118
5.14.3 Member Function Documentation	118
5.14.3.1 addToSpace()	119
5.15 PlaneWave3D Class Reference	119
5.15.1 Detailed Description	120
5.15.2 Constructor & Destructor Documentation	120
5.15.2.1 PlaneWave3D()	120
5.15.3 Member Function Documentation	121
5.15.3.1 addToSpace()	121
5.16 Simulation Class Reference	122
5.16.1 Detailed Description	123

5.16.2 Constructor & Destructor Documentation	123
5.16.2.1 Simulation()	124
5.16.2.2 ~Simulation()	124
5.16.3 Member Function Documentation	125
5.16.3.1 addInitialConditions()	125
5.16.3.2 addPeriodicCLayerInX()	126
5.16.3.3 addPeriodicCLayerInXY()	127
5.16.3.4 advanceToTime()	128
5.16.3.5 checkFlag()	129
5.16.3.6 checkNoFlag()	130
5.16.3.7 get_cart_comm()	131
5.16.3.8 initializeCVODEobject()	131
5.16.3.9 initializePatchwork()	133
5.16.3.10 outAllFieldData()	134
5.16.3.11 setDiscreteDimensionsOfLattice()	135
5.16.3.12 setInitialConditions()	136
5.16.3.13 setPhysicalDimensionsOfLattice()	137
5.16.3.14 start()	138
5.16.4 Field Documentation	139
5.16.4.1 ccode_mem	139
5.16.4.2 icsettings	140
5.16.4.3 lattice	140
5.16.4.4 latticePatch	140
5.16.4.5 outputManager	140
5.16.4.6 statusFlags	141
5.16.4.7 t	141
5.17 TimeEvolution Class Reference	141
5.17.1 Detailed Description	142
5.17.2 Member Function Documentation	142
5.17.2.1 f()	142
5.17.3 Field Documentation	143
5.17.3.1 c	143
5.17.3.2 TimeEvolver	143
6 File Documentation	145
6.1 README.md File Reference	145
6.2 src/DerivationStencils.cpp File Reference	145
6.2.1 Detailed Description	145
6.3 DerivationStencils.cpp	146
6.4 src/DerivationStencils.h File Reference	146
6.4.1 Detailed Description	148
6.4.2 Function Documentation	148

6.4.2.1 s10b() [1/2]	148
6.4.2.2 s10b() [2/2]	149
6.4.2.3 s10c() [1/2]	149
6.4.2.4 s10c() [2/2]	150
6.4.2.5 s10f() [1/2]	150
6.4.2.6 s10f() [2/2]	151
6.4.2.7 s11b() [1/2]	151
6.4.2.8 s11b() [2/2]	152
6.4.2.9 s11f() [1/2]	152
6.4.2.10 s11f() [2/2]	153
6.4.2.11 s12b() [1/2]	153
6.4.2.12 s12b() [2/2]	154
6.4.2.13 s12c() [1/2]	155
6.4.2.14 s12c() [2/2]	155
6.4.2.15 s12f() [1/2]	156
6.4.2.16 s12f() [2/2]	156
6.4.2.17 s13b() [1/2]	157
6.4.2.18 s13b() [2/2]	157
6.4.2.19 s13f() [1/2]	158
6.4.2.20 s13f() [2/2]	158
6.4.2.21 s1b() [1/2]	159
6.4.2.22 s1b() [2/2]	159
6.4.2.23 s1f() [1/2]	160
6.4.2.24 s1f() [2/2]	160
6.4.2.25 s2b() [1/2]	161
6.4.2.26 s2b() [2/2]	161
6.4.2.27 s2c() [1/2]	162
6.4.2.28 s2c() [2/2]	162
6.4.2.29 s2f() [1/2]	163
6.4.2.30 s2f() [2/2]	163
6.4.2.31 s3b() [1/2]	164
6.4.2.32 s3b() [2/2]	164
6.4.2.33 s3f() [1/2]	165
6.4.2.34 s3f() [2/2]	165
6.4.2.35 s4b() [1/2]	166
6.4.2.36 s4b() [2/2]	166
6.4.2.37 s4c() [1/2]	167
6.4.2.38 s4c() [2/2]	167
6.4.2.39 s4f() [1/2]	168
6.4.2.40 s4f() [2/2]	168
6.4.2.41 s5b() [1/2]	169
6.4.2.42 s5b() [2/2]	169

6.4.2.43 s5f() [1/2]	170
6.4.2.44 s5f() [2/2]	170
6.4.2.45 s6b() [1/2]	171
6.4.2.46 s6b() [2/2]	171
6.4.2.47 s6c() [1/2]	172
6.4.2.48 s6c() [2/2]	172
6.4.2.49 s6f() [1/2]	173
6.4.2.50 s6f() [2/2]	173
6.4.2.51 s7b() [1/2]	174
6.4.2.52 s7b() [2/2]	174
6.4.2.53 s7f() [1/2]	175
6.4.2.54 s7f() [2/2]	175
6.4.2.55 s8b() [1/2]	176
6.4.2.56 s8b() [2/2]	176
6.4.2.57 s8c() [1/2]	177
6.4.2.58 s8c() [2/2]	177
6.4.2.59 s8f() [1/2]	178
6.4.2.60 s8f() [2/2]	178
6.4.2.61 s9b() [1/2]	179
6.4.2.62 s9b() [2/2]	179
6.4.2.63 s9f() [1/2]	180
6.4.2.64 s9f() [2/2]	180
6.5 DerivationStencils.h	181
6.6 src/ICSetters.cpp File Reference	184
6.6.1 Detailed Description	184
6.7 ICSetters.cpp	184
6.8 src/ICSetters.h File Reference	188
6.8.1 Detailed Description	190
6.9 ICSetters.h	190
6.10 src/LatticePatch.cpp File Reference	193
6.10.1 Detailed Description	194
6.10.2 Function Documentation	194
6.10.2.1 check_retval()	194
6.10.2.2 errorKill()	195
6.10.2.3 generatePatchwork()	196
6.11 LatticePatch.cpp	198
6.12 src/LatticePatch.h File Reference	210
6.12.1 Detailed Description	211
6.12.2 Enumeration Type Documentation	211
6.12.2.1 LatticeOptions	211
6.12.2.2 LatticePatchOptions	211
6.12.3 Function Documentation	212

6.12.3.1 check_retval()	212
6.12.3.2 errorKill()	213
6.13 LatticePatch.h	214
6.14 src/main.cpp File Reference	217
6.14.1 Detailed Description	218
6.14.2 Function Documentation	218
6.14.2.1 main()	218
6.15 main.cpp	226
6.16 src/Outputters.cpp File Reference	230
6.16.1 Detailed Description	230
6.17 Outputters.cpp	230
6.18 src/Outputters.h File Reference	232
6.18.1 Detailed Description	233
6.19 Outputters.h	233
6.20 src/SimulationClass.cpp File Reference	233
6.20.1 Detailed Description	234
6.21 SimulationClass.cpp	234
6.22 src/SimulationClass.h File Reference	238
6.22.1 Detailed Description	239
6.22.2 Enumeration Type Documentation	239
6.22.2.1 SimulationOptions	239
6.23 SimulationClass.h	239
6.24 src/SimulationFunctions.cpp File Reference	241
6.24.1 Detailed Description	241
6.24.2 Function Documentation	242
6.24.2.1 Sim1D()	242
6.24.2.2 Sim2D()	245
6.24.2.3 Sim3D()	248
6.24.2.4 timer()	251
6.25 SimulationFunctions.cpp	252
6.26 src/SimulationFunctions.h File Reference	255
6.26.1 Detailed Description	256
6.26.2 Function Documentation	256
6.26.2.1 Sim1D()	256
6.26.2.2 Sim2D()	259
6.26.2.3 Sim3D()	262
6.26.2.4 timer()	265
6.27 SimulationFunctions.h	266
6.28 src/TimeEvolutionFunctions.cpp File Reference	267
6.28.1 Detailed Description	267
6.28.2 Function Documentation	267
6.28.2.1 linear1DProp()	268

6.28.2.2 linear2DProp()	269
6.28.2.3 linear3DProp()	271
6.28.2.4 nonlinear1DProp()	273
6.28.2.5 nonlinear2DProp()	278
6.28.2.6 nonlinear3DProp()	282
6.29 TimeEvolutionFunctions.cpp	286
6.30 src/TimeEvolutionFunctions.h File Reference	295
6.30.1 Detailed Description	296
6.30.2 Function Documentation	296
6.30.2.1 linear1DProp()	296
6.30.2.2 linear2DProp()	298
6.30.2.3 linear3DProp()	299
6.30.2.4 nonlinear1DProp()	301
6.30.2.5 nonlinear2DProp()	306
6.30.2.6 nonlinear3DProp()	310
6.31 TimeEvolutionFunctions.h	314
Index	315

Chapter 1

HEWES: Heisenberg-Euler Weak-Field Expansion Simulator

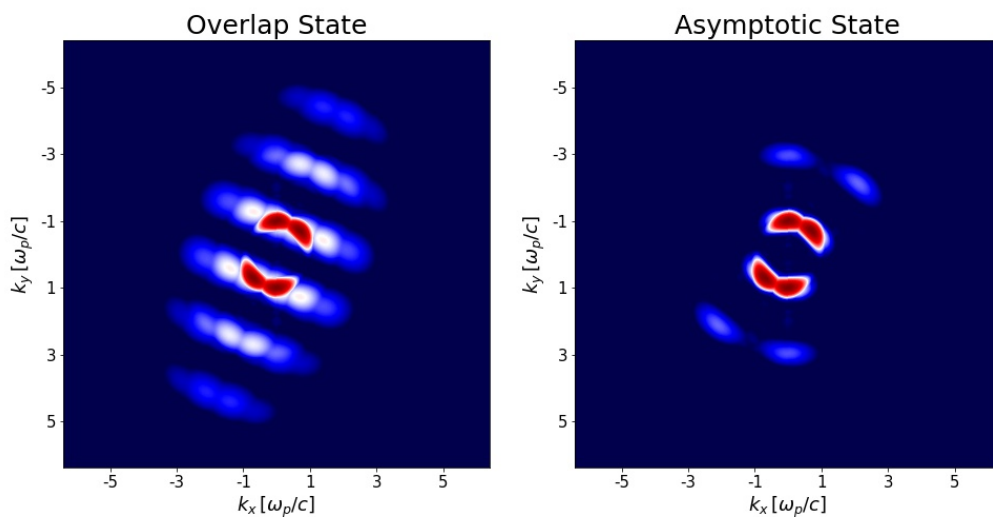


Figure 1.1 Harmonic Generation

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 output format. It can be 'c' for comma separated values (csv), or 'b' for binary. For csv format the name of the files written to the output directory is of the form `{step_number}_{process_number}.csv`. For binary output all data per step is written into one file and the step number is the name of the file.
- which electromagnetic waveform(s) you want to propagate.
You can choose between a plane wave (not much physical content, but useful for checks) and implementations of Gaussians in 1D, 2D, and 3D. Their parameters can be tuned.
A description of the wave implementations is given in [ref.pdf](#). Note that the 3D Gaussians, as they are implemented up to now, should be propagated in the xy-plane. More waveform implementations will follow in subsequent versions of the code.

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

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

```
mpirun -np 4 ./Simulation
```
- Monitor stdout and stderr. The unique simulation identifier number (starting timestep = name of data directory), the process steps, and the used wall times per step are printed on stdout. Errors are printed on stderr.
Note: Convergence of the employed 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 nonphysical dispersion effects. The wavelengths should neither be chosen too large (bulky wave) on a fine patchwork of narrow patches. Their communication might be problematic with too small halo layer depths. You would observe a blurring over time. The amplitudes need be below 1 – the critical field strength – for the weak-field expansion to be valid.

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

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

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

Example scenarios of colliding Gaussians are preconfigured for any dimension.

1.3.2 Note on Resource Occupation

The computational load depends mostly on the grid size and resolution. The order of accuracy of the numerical scheme and 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 either written in csv format to one file per MPI process, the ending of which (after an underscore) corresponds to the process number, as described above. This is not an elegant solution, but a portable way that also works fast and is straightforward to analyze.

Or, the option recommended for many larger write operations, in binary format with a single file per output step. Raw bytes are written to the files as they are in memory. This option is more performant and achieved with MPI IO. However, there is no guarantee of portability; postprocessing/conversion is required. The step number is the file name.

A `SimResults` folder is created in the chosen output directory if it does not exist and a folder named after the starting timestep of the simulation (in the form `yy-mm-dd_hh-MM-ss`) is created where the output files are written into. There are six columns in the csv files, corresponding to the six components of the electromagnetic field: `E_x, E_y, E_z, B_x, B_y, B_z`. Each row corresponds to one lattice point.

Postprocessing is required to read-in the files in order. A Python `module` taking care of this is provided.

Likewise, a Python `module` is provided to read the binary data of a selected field component into a numpy array – its portability, however, cannot be guaranteed.

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

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

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 [\[examples\]\(examples\)](#). 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	102
PlaneWave	109
PlaneWave1D	114
PlaneWave2D	116
PlaneWave3D	119
planewave	113
Simulation	122
TimeEvolution	141

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	102
PlaneWave	Super-class for plane waves	109
planewave	Plane wave structure	113
PlaneWave1D	Class for plane waves in 1D	114
PlaneWave2D	Class for plane waves in 2D	116
PlaneWave3D	Class for plane waves in 3D	119
Simulation	Simulation class to instantiate the whole walkthrough of a Simulation	122
TimeEvolution	Monostate TimeEvolution Class to propagate the field data in time in a given order of the HE weak-field expansion	141

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	145
src/ DerivationStencils.h	
Definition of derivation stencils from order 1 to 13	146
src/ ICSetters.cpp	
Implementation of the plane wave and Gaussian wave packets in 1D, 2D, 3D	184
src/ ICSetters.h	
Declaration of the plane wave and Gaussian wave packets in 1D, 2D, 3D	188
src/ LatticePatch.cpp	
Costruction of the overall envelope lattice and the lattice patches	193
src/ LatticePatch.h	
Declaration of the lattice and lattice patches	210
src/ main.cpp	
Main function to configure the user's simulation settings	217
src/ Outputters.cpp	
Generation of output writing to disk	230
src/ Outputters.h	
OutputManager class to outstream simulation data	232
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)	233
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	238
src/ SimulationFunctions.cpp	
Implementation of the complete simulation functions for 1D, 2D, and 3D, as called in the main function	241
src/ SimulationFunctions.h	
Full simulation functions for 1D, 2D, and 3D used in main.cpp	255
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	267
src/ TimeEvolutionFunctions.h	
Functions to propagate data vectors in time according to Maxwell's equations, and various orders in the HE weak-field expansion	295

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 [86](#) 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 125 of file `ICSetters.cpp`.

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

References `kx`, `ky`, `kz`, `phig`, `phix`, `phiy`, `phiz`, `px`, `py`, `pz`, `x0x`, `x0y`, and `x0z`.

5.1.3 Member Function Documentation

5.1.3.1 addToSpace()

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

function for the actual implementation in space

[Gauss1D](#) implementation in space

Definition at line 145 of file `ICSetters.cpp`.

```
00146 {
00147     const sunrealtype wavelength =
00148         sqrt(kx * kx + ky * ky + kz * kz); /* \f$ 1/\lambda \f$ */
00149     x = x - x0x; /* x-coordinate minus shift from origin */
00150     y = y - x0y; /* y-coordinate minus shift from origin */
00151     z = z - x0z; /* z-coordinate minus shift from origin */
00152     const sunrealtype kScalarX = (kx * x + ky * y + kz * z) * 2 *
00153         numbers::pi; /* \f$ 2\pi \cdot \vec{k} \cdot \vec{x} \f$ */
00154     const sunrealtype envelopeAmp =
00155         exp(-(x * x + y * y + z * z) / phig / phig); /* enveloping Gauss shape */
00156     // Gaussian wave definition
00157     const array<sunrealtype, 3> E{
00158         {
00159             px * cos(kScalarX - phix) * envelopeAmp, /* \f$ E_x \f$ */
00160             py * cos(kScalarX - phiy) * envelopeAmp, /* \f$ E_y \f$ */
00161             pz * cos(kScalarX - phiz) * envelopeAmp}}; /* \f$ E_z \f$ */
00162     // Put E-field into space
00163     pTo6Space[0] += E[0];
00164     pTo6Space[1] += E[1];
00165     pTo6Space[2] += E[2];
00166     // and B-field
00167     pTo6Space[3] += (ky * E[2] - kz * E[1]) / wavelength;
00168     pTo6Space[4] += (kz * E[0] - kx * E[2]) / wavelength;
00169     pTo6Space[5] += (kx * E[1] - ky * E[0]) / wavelength;
00170 }
```

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 89 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 91 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 93 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 113 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 101 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 103 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 105 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 95 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 97 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 99 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 107 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 109 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 111 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 [phiip](#)
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 140 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 173 of file `ICSetters.cpp`.

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

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 190 of file `ICSetters.cpp`.

```
00191 {
00192     //f$ \vec{x} = \vec{x}_0-\vec{dis} \f$ // coordinates minus distance to
00193     //origin
00194     x -= dis[0];
00195     y -= dis[1];
00196     // z-=dis[2];
00197     z = NAN;
00198     // \f$ z_g = \vec{x}\cdot\vec{e}_g \f$ projection on propagation axis
00199     const sunrealtype zg =
00200         x * axis[0] + y * axis[1]; //+z*axis[2]; // =z-z0 -> propagation
00201         //direction, minus origin
00202     // \f$ r = \sqrt{\vec{x}^2 - z_g^2} \f$ -> pythagoras of radius minus
00203     // projection on prop axis
00204     const sunrealtype r = sqrt((x * x + y * y /*+z*z*/)) -
00205         zg * zg; // radial distance to propagation axis
00206     // \f$ w(z) = w0\sqrt{1+(z_g/z_R)^2} \f$
00207     const sunrealtype wz = w0 * sqrt(1 + (zg * zg / zr / zr)); // waist at position z
00208     // \f$ g(z) = atan(z_g/z_r) \f$
00209     const sunrealtype gz = atan(zg / zr); // Gouy phase
00210     // \f$ R(z) = z_g*(1+(z_r/z_g)^2) \f$
00211     sunrealtype Rz = NAN; // beam curvature
00212     if (zg != 0)
00213         Rz = zg * (1 + (zr * zr / zg / zg));
00214     else
00215         Rz = 1e308;
00216     // wavenumber \f$ k = 2\pi/\lambda \f$
00217     const sunrealtype k = 2 * numbers::pi / lambda;
00218     // \f$ \Phi_F = kr^2/(2R(z))+g(z)-kz_g \f$
00219     const sunrealtype PhF =
00220         -k * r * r / (2 * Rz) + gz - k * zg; // to be inserted into cosine
00221     // \f$ G = \sqrt{w_0/w_z}\e^{-r/w(z)^2}\e^{(zg-Ph0)^2/PhA^2}\cos(PhF) \f$
00222     // CCode is a diva, no chance to remove the square in the second exponential
00223     // -> h too small
00224     const sunrealtype G2D = sqrt(w0 / wz) * exp(-r * r / wz / wz) *
00225         exp(-(zg - Ph0) * (zg - Ph0) / PhA / PhA) *
00226         cos(PhF); // gauss shape
00227     // \f$ c_\alpha = \vec{e}_x\cdot\vec{axis} \f$
00228     // projection components; do like this for CCode convergence -> otherwise
00229     // results in machine error values for non-existant field components if
00230     // axis[0] and axis[1] are given
00231     const sunrealtype ca =
```

```

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

```

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 160 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 162 of file [ICSetters.h](#).

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

5.2.4.3 Amp

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

amplitude A

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

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

5.2.4.4 axis

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

normalized propagation axis

Definition at line 145 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 143 of file [ICSetters.h](#).

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

5.2.4.6 lambda

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

wavelength λ

Definition at line 164 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 156 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 158 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 150 of file [ICSetters.h](#).

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

5.2.4.10 w0

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

taille ω_0

Definition at line 152 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 154 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 192 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 246 of file ICSetters.cpp.

```
00250                                     {
00251     dis = dis_; /** - center it approaches */
00252     axis = axis_; /** - direction from where it comes */
00253     Amp = Amp_; /** - amplitude */
00254     // pol=pol_;
00255     phip = phip_; /** - polarization rotation form TE-mode */
00256     w0 = w0_; /** - taille */
00257     zr = zr_; /** - Rayleigh length */
00258     Ph0 = Ph0_; /** - beam center */
00259     PhA = PhA_; /** - beam length */
00260     lambda = numbers::pi * w0 * w0 / zr;
00261     A1 = Amp * cos(phip);
00262     A2 = Amp * sin(phip);
00263 }
```

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 266 of file ICSetters.cpp.

```
00267                                     {
00268     x -= dis[0];
00269     y -= dis[1];
00270     z -= dis[2];
00271     const sunrealtype zg = x * axis[0] + y * axis[1] + z * axis[2];
00272     const sunrealtype r = sqrt((x * x + y * y + z * z) - zg * zg);
00273     const sunrealtype wz = w0 * sqrt(1 + (zg * zg / zr / zr));
00274     const sunrealtype gz = atan(zg / zr);
00275     sunrealtype Rz = NAN;
00276     if (zg != 0)
00277         Rz = zg * (1 + (zr * zr / zg / zg));
00278     else
00279         Rz = 1e308;
00280     const sunrealtype k = 2 * numbers::pi / lambda;
00281     const sunrealtype PhF = -k * r * r / (2 * Rz) + gz - k * zg;
00282     const sunrealtype G3D = (w0 / wz) * exp(-r * r / wz / wz) *
00283         exp(-(zg - Ph0) * (zg - Ph0) / PhA / PhA) * cos(PhF);
00284     const sunrealtype ca = axis[0];
00285     const sunrealtype sa = sqrt(1 - ca * ca);
00286     pTo6Space[0] += sa * (G3D * A2);
00287     pTo6Space[1] += -ca * (G3D * A2);
00288     pTo6Space[2] += G3D * A1;
00289     pTo6Space[3] += -sa * (G3D * A1);
00290     pTo6Space[4] += ca * (G3D * A1);
00291     pTo6Space[5] += G3D * A2;
00292 }
```

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 214 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 216 of file [ICSetters.h](#).

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

5.3.4.3 Amp

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

amplitude A

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

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

5.3.4.4 axis

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

normalized propagation axis

Definition at line 197 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 195 of file [ICSetters.h](#).

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

5.3.4.6 lambda

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

wavelength λ

Definition at line 218 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 210 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 212 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 202 of file [ICSetters.h](#).

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

5.3.4.10 w0

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

taille ω_0

Definition at line 206 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 208 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 27 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 28 of file [SimulationFunctions.h](#).

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

5.4.2.2 p

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

amplitude & polarization vector

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

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

5.4.2.3 phi

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

phase shift

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

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

5.4.2.4 phig

```
sunrealtype gaussian1D::phig
```

width

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

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

5.4.2.5 x0

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

shift from origin

Definition at line 30 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 36 of file [SimulationFunctions.h](#).

5.5.2 Field Documentation

5.5.2.1 amp

```
sunrealtype gaussian2D::amp
```

amplitude

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

5.5.2.2 axis

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

direction to center

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

5.5.2.3 ph0

```
sunrealtype gaussian2D::ph0
```

beam center

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

5.5.2.4 phA

```
sunrealtype gaussian2D::phA
```

beam length

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

5.5.2.5 phip

```
sunrealtype gaussian2D::phip
```

polarization rotation

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

5.5.2.6 w0

```
sunrealtype gaussian2D::w0
```

taille

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

5.5.2.7 x0

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

center

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

5.5.2.8 zr

```
sunrealtype gaussian2D::zr
```

Rayleigh length

Definition at line 42 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 48 of file [SimulationFunctions.h](#).

5.6.2 Field Documentation

5.6.2.1 amp

```
sunrealtype gaussian3D::amp
```

amplitude

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

5.6.2.2 axis

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

direction to center

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

5.6.2.3 ph0

```
sunrealtype gaussian3D::ph0
```

beam center

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

5.6.2.4 phA

```
sunrealtype gaussian3D::phA
```

beam length

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

5.6.2.5 phip

```
sunrealtype gaussian3D::phip
```

polarization rotation

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

5.6.2.6 w0

```
sunrealtype gaussian3D::w0
```

taille

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

5.6.2.7 x0

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

center

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

5.6.2.8 zr

```
sunrealtype gaussian3D::zr
```

Rayleigh length

Definition at line 54 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 236 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

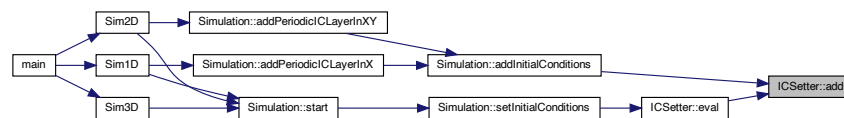
Definition at line 308 of file [ICSetters.cpp](#).

```
00309 {
00310     for (const auto &wave : planeWaves1D)
00311         wave.addToSpace(x, y, z, pTo6Space);
00312     for (const auto &wave : planeWaves2D)
00313         wave.addToSpace(x, y, z, pTo6Space);
00314     for (const auto &wave : planeWaves3D)
00315         wave.addToSpace(x, y, z, pTo6Space);
00316     for (const auto &wave : gauss1Ds)
00317         wave.addToSpace(x, y, z, pTo6Space);
00318     for (const auto &wave : gauss2Ds)
00319         wave.addToSpace(x, y, z, pTo6Space);
00320     for (const auto &wave : gauss3Ds)
00321         wave.addToSpace(x, y, z, pTo6Space);
00322 }
```

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

```
00345 {
00346     gauss1Ds.emplace_back(Gauss1D(k, p, xo, phig_, phi));
00347 }
```

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

```
00352 {
00353     gauss2Ds.emplace_back(
00354         Gauss2D(dis_, axis_, Amp_, phip_, w0_, zr_, Ph0_, PhA_));
00355 }
```

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

```
00360
00361     gauss3Ds.emplace_back(
00362         Gauss3D(dis_, axis_, Amp_, phip_, w0_, zr_, Ph0_, PhA_));
00363 }
```

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

```
00326
00327     planeWaves1D.emplace_back(PlaneWave1D(k, p, phi));
00328 }
```

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 331 of file `ICSetters.cpp`.

```
00332 {
00333     planeWaves2D.emplace_back(PlaneWave2D(k, p, phi));
00334 }
```

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

```
00338 {
00339     planeWaves3D.emplace_back(PlaneWave3D(k, p, phi));
00340 }
```

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

```
00297 {
00298     pTo6Space[0] = 0;
00299     pTo6Space[1] = 0;
00300     pTo6Space[2] = 0;
00301     pTo6Space[3] = 0;
00302     pTo6Space[4] = 0;
00303     pTo6Space[5] = 0;
00304     add(x, y, z, pTo6Space);
00305 }
```

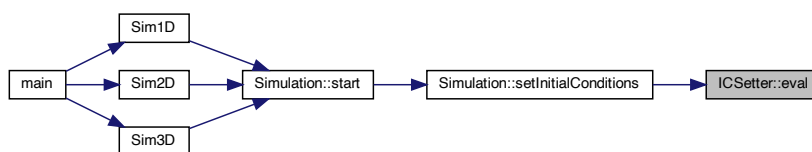
References [add\(\)](#).

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 245 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 247 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 249 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 239 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 241 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 243 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 46 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

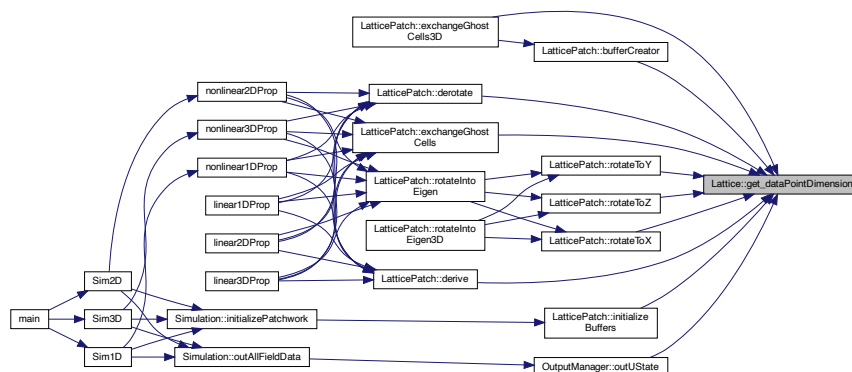
Definition at line 114 of file [LatticePatch.h](#).

```
00114         {
00115     return dataPointDimension;
00116 }
```

References [dataPointDimension](#).

Referenced by [LatticePatch::bufferCreator\(\)](#), [LatticePatch::derive\(\)](#), [LatticePatch::derotate\(\)](#), [LatticePatch::exchangeGhostCells\(\)](#), [LatticePatch::exchangeGhostCells3D\(\)](#), [LatticePatch::initializeBuffers\(\)](#), [OutputManager::outUState\(\)](#), [LatticePatch::rotateToX\(\)](#), [LatticePatch::rotateToY\(\)](#), and [LatticePatch::rotateToZ\(\)](#).

Here is the caller graph for this function:



5.8.3.2 `get_dx()`

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

getter function

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

```
00111 { return dx; }
```

References [dx](#).

5.8.3.3 `get_dy()`

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

getter function

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

```
00112 { return dy; }
```

References [dy](#).

5.8.3.4 `get_dz()`

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

getter function

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

```
00113 { return dz; }
```

References [dz](#).

5.8.3.5 get_ghostLayerWidth()

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

getter function

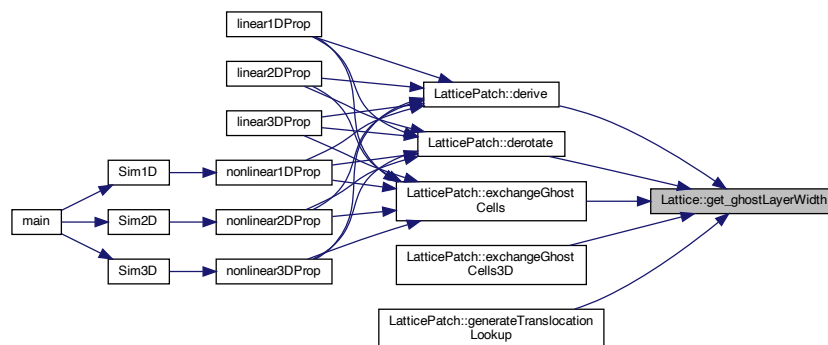
Definition at line 118 of file [LatticePatch.h](#).

```
00118 {
00119     return ghostLayerWidth;
00120 }
```

References [ghostLayerWidth](#).

Referenced by [LatticePatch::derive\(\)](#), [LatticePatch::derotate\(\)](#), [LatticePatch::exchangeGhostCells\(\)](#), [LatticePatch::exchangeGhostCells3D\(\)](#) 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

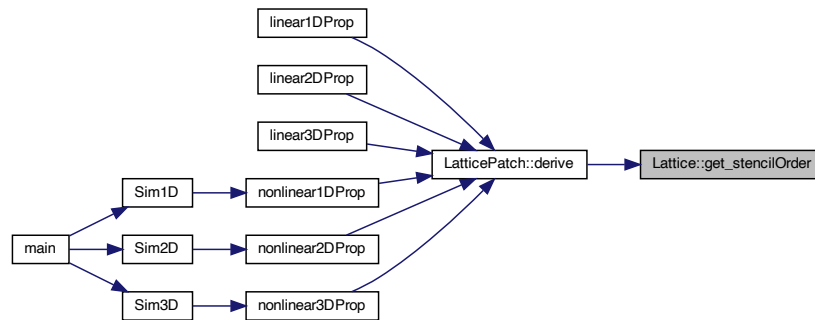
Definition at line 117 of file [LatticePatch.h](#).

```
00117 { 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

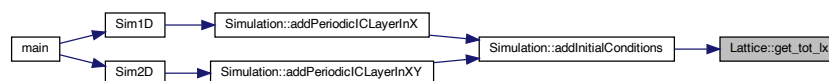
Definition at line 103 of file [LatticePatch.h](#).

```
00103 { 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 104 of file [LatticePatch.h](#).

```
00104 { 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 105 of file [LatticePatch.h](#).

```
00105 { 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 110 of file [LatticePatch.h](#).

```
00110 { 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 109 of file [LatticePatch.h](#).

```
00109 { 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 106 of file [LatticePatch.h](#).

```
00106 { 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 107 of file [LatticePatch.h](#).

```
00107 { 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 108 of file [LatticePatch.h](#).

```
00108 { 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.

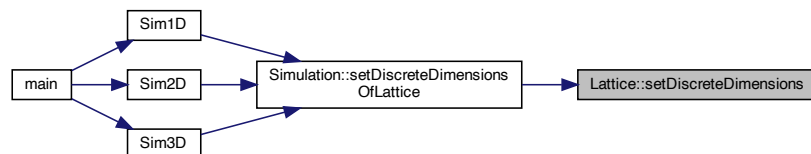
Definition at line 45 of file [LatticePatch.cpp](#).

```
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 85 of file [LatticePatch.h](#).

Referenced by [LatticePatch::exchangeGhostCells\(\)](#), [LatticePatch::exchangeGhostCells3D\(\)](#), [Simulation::get_cart_comm\(\)](#), [initializeCommunicator\(\)](#), [Simulation::initializeCVODEobject\(\)](#), [OutputManager::outUState\(\)](#), and [Simulation::Simulation\(\)](#).

5.8.4.2 dataPointDimension

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

dimension of each data point -> set once and for all

Definition at line 63 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 67 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 69 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 71 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 75 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 83 of file [LatticePatch.h](#).

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

5.8.4.8 n_prc

```
int Lattice::n_prc
```

number of MPI processes

Definition at line 81 of file [LatticePatch.h](#).

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

5.8.4.9 profobj

```
SUNProfiler Lattice::profobj
```

SUNProfiler object.

Definition at line 94 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 77 of file [LatticePatch.h](#).

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

5.8.4.11 stencilOrder

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

stencil order

Definition at line 73 of file [LatticePatch.h](#).

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

5.8.4.12 `sunctx`

```
SUNContext Lattice::sunctx
```

SUNContext object.

Definition at line 92 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 49 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 51 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 53 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 65 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 61 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 55 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 57 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 59 of file [LatticePatch.h](#).

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

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

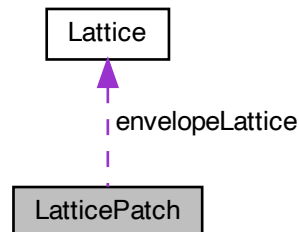
- [src/LatticePatch.h](#)
- [src/LatticePatch.cpp](#)

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 for translocation*
 - void [rotateIntoEigen](#) (const int dir)
- function to rotate u into Z-matrix eigenraum*
 - void [rotateIntoEigen3D](#) ()
- function to rotate as in [rotateIntoEigen](#) with special 3D halo buffers*
 - 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 [exchangeGhostCells3D](#) ()
- function to exchange ghost cells using a neighborhood collective operation*
 - void [bufferCreator](#) (int li, int mx, int my, int mz, int distToRight)
- outsourced convenience function to fill halo buffers with uData for 3D*
 - 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](#)
- sunrealtype * [gCBData](#)
- sunrealtype * [gCTData](#)
- sunrealtype * [gCFData](#)
- sunrealtype * [gCADATA](#)

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 137 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 bufferCreator()

```
void LatticePatch::bufferCreator (
    int li,
    int mx,
    int my,
    int mz,
    int distToRight )
```

outsourced convenience function to fill halo buffers with uData for 3D

Fill the halo buffers for neighborhood collectives.

Definition at line 669 of file [LatticePatch.cpp](#).

```
00670     {
00671         const int dPD = envelopeLattice->get_dataPointDimension();
00672         // Initialize running index ui for to be transferred uData
00673         int ui = 0;
00674
00675         // #pragma omp parallel for reduction(+:ui) reduction(+:li) -> don't, probably
00676         // bad idea to parallelize ghost cell exchange Loop over all planes and pick to
00677         // be transferred points (uData indices)
00678         #pragma distribute_point
00679         for (int iz = 0; iz < mz; iz++) {
00680             for (int iy = 0; iy < my; iy++) {
00681                 // start index of uData vector halo data to be transferred
00682                 // Here, in contrast to above, start at right boundary to send to left
00683                 // s.t. updated left values are the first indices (bec of neighborhood
00684                 // collective pattern) with each z-step add the whole xy-plane and with
00685                 // y-step the x-range -> iterate all x-ranges
00686                 ui = (iz * nx * ny + iy * nx + distToRight) * dPD;
00687                 // copy from uData from right boundary (at each dimension) into buffer,
00688                 // halo transfer size is given by x length at each step
00689                 // memcpy(&ghostCellsToSend[li], \
00690                     &uData[ui], \
00691                     sizeof(sunrealtype)*mx*dPD);
00692                 copy(&uData[ui], &uData[ui + mx * dPD], &ghostCellsToSend[li]);
00693                 // increase li by transferred indices in this loop-step
00694                 li += mx * dPD;
00695             }
00696         }
00697
00698         #pragma distribute_point
00699         for (int iz = 0; iz < mz; iz++) {
00700             for (int iy = 0; iy < my; iy++) {
00701                 // Now copy from left boundary into buffer
00702                 ui = (iz * nx * ny + iy * nx) * dPD;
00703                 //memcpy(&ghostCellsToSend[li], \
00704                     &uData[ui], \
00705                     sizeof(sunrealtype)*mx*dPD);
00706                 copy(&uData[ui], &uData[ui + mx * dPD], &ghostCellsToSend[li]);
00707                 li += mx * dPD;
00708             }
00709         }
00710     }
```

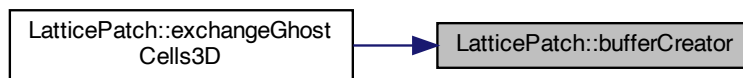
References [envelopeLattice](#), [Lattice::get_dataPointDimension\(\)](#), [ghostCellsToSend](#), [nx](#), [ny](#), and [uData](#).

Referenced by [exchangeGhostCells3D\(\)](#).

Here is the call graph for this function:



Here is the caller graph for this function:



5.9.3.2 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 713 of file [LatticePatch.cpp](#).

```

00713                                     {
00714     if (!(statusFlags & flag)) {
00715         string errorMessage;
00716         switch (flag) {
00717             case FLatticePatchSetUp:
00718                 errorMessage = "The Lattice patch was not set up please make sure to "
00719                               "initilize a Lattice topology";
00720                 break;
00721             case TranslocationLookupSetUp:
00722                 errorMessage = "The translocation lookup tables have not been generated, "
00723                               "please be sure to run generateTranslocationLookup()";
00724                 break;
00725             case GhostLayersInitialized:
00726                 errorMessage = "The space for the ghost layers has not been allocated, "
00727                               "please be sure to run initializeGhostLayer()";
00728                 break;
00729             case BuffersInitialized:
00730                 errorMessage = "The space for the buffers has not been allocated, please "
00731                               "be sure to run initializeBuffers()";
00732                 break;
00733             default:
00734                 errorMessage = "Uppss, you've made a non-standard error, sadly I can't "
00735                               "help you there";
00736                 break;
00737         }
00738         errorKill(errorMessage);
00739     }
00740     return;
00741 }
```

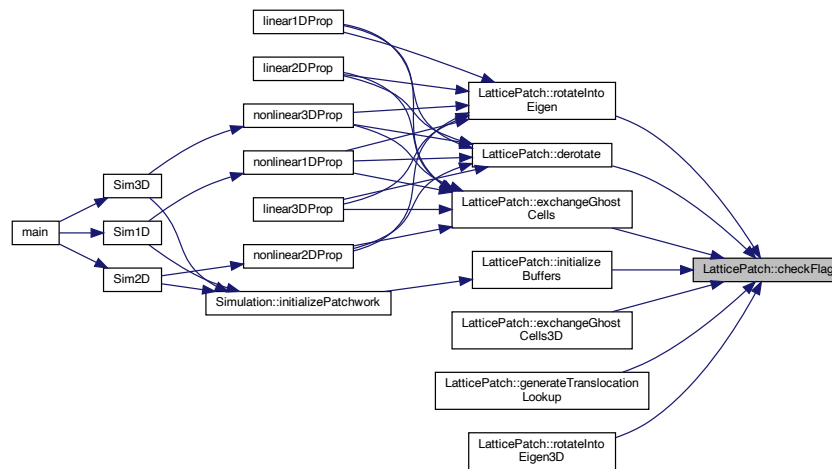
References [BuffersInitialized](#), [errorKill\(\)](#), [FLatticePatchSetUp](#), [GhostLayersInitialized](#), [statusFlags](#), and [TranslocationLookupSetUp](#).

Referenced by [derotate\(\)](#), [exchangeGhostCells\(\)](#), [exchangeGhostCells3D\(\)](#), [generateTranslocationLookup\(\)](#), [initializeBuffers\(\)](#), [rotateIntoEigen\(\)](#), and [rotateIntoEigen3D\(\)](#).

Here is the call graph for this function:



Here is the caller graph for this function:



5.9.3.3 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 744 of file [LatticePatch.cpp](#).

```

00744 {
00745     // ghost layer width
00746     const int gLW = envelopeLattice->get_ghostLayerWidth();
00747     // dimensionality of data points -> 6
00748     const int dPD = envelopeLattice->get_dataPointDimension();
00749     // total width of patch in given direction including ghost layers at ends
00750     const int dirWidth = discreteSize(dir) + 2 * gLW;
00751     // width of patch only in given direction
00752     const int dirWidth0 = discreteSize(dir);
00753     // size of plane perpendicular to given dimension
00754     const int perpPlainSize = discreteSize() / discreteSize(dir);
00755     // physical distance between points in that direction
00756     sunrealtype dxi = NAN;
00757     switch (dir) {
00758     case 1:
00759         dxi = dx;
00760         break;
00761     case 2:
00762         dxi = dy;
00763         break;
00764     case 3:
00765         dxi = dz;
00766         break;
00767     default:
00768         dxi = 1;
00769         errorKill("Tried to derive in the wrong direction");
00770         break;
00771     }
00772     // Derive according to chosen stencil accuracy order (which determines also
00773     // gLW)
00774     const int order = envelopeLattice->get_stencilOrder();
00775     switch (order) {
00776     case 1:

```

```

00777     for (int i = 0; i < perpPlainSize; i++) {
00778         for (int j = (i * dirWidth + gLW) * dPD;
00779             j < (i * dirWidth + gLW + dirWidthO) * dPD; j += dPD) {
00780             uAux[j + 0 - gLW * dPD] = s1b(&uAux[j + 0]) / dxi;
00781             uAux[j + 1 - gLW * dPD] = s1b(&uAux[j + 1]) / dxi;
00782             uAux[j + 2 - gLW * dPD] = s1f(&uAux[j + 2]) / dxi;
00783             uAux[j + 3 - gLW * dPD] = s1f(&uAux[j + 3]) / dxi;
00784             uAux[j + 4 - gLW * dPD] = s1f(&uAux[j + 4]) / dxi;
00785             uAux[j + 5 - gLW * dPD] = s1f(&uAux[j + 5]) / dxi;
00786         }
00787     }
00788     break;
00789 case 2:
00790     for (int i = 0; i < perpPlainSize; i++) {
00791         for (int j = (i * dirWidth + gLW) * dPD;
00792             j < (i * dirWidth + gLW + dirWidthO) * dPD; j += dPD) {
00793             uAux[j + 0 - gLW * dPD] = s2b(&uAux[j + 0]) / dxi;
00794             uAux[j + 1 - gLW * dPD] = s2b(&uAux[j + 1]) / dxi;
00795             uAux[j + 2 - gLW * dPD] = s2f(&uAux[j + 2]) / dxi;
00796             uAux[j + 3 - gLW * dPD] = s2f(&uAux[j + 3]) / dxi;
00797             uAux[j + 4 - gLW * dPD] = s2c(&uAux[j + 4]) / dxi;
00798             uAux[j + 5 - gLW * dPD] = s2c(&uAux[j + 5]) / dxi;
00799         }
00800     }
00801     break;
00802 case 3:
00803     for (int i = 0; i < perpPlainSize; i++) {
00804         for (int j = (i * dirWidth + gLW) * dPD;
00805             j < (i * dirWidth + gLW + dirWidthO) * dPD; j += dPD) {
00806             uAux[j + 0 - gLW * dPD] = s3b(&uAux[j + 0]) / dxi;
00807             uAux[j + 1 - gLW * dPD] = s3b(&uAux[j + 1]) / dxi;
00808             uAux[j + 2 - gLW * dPD] = s3f(&uAux[j + 2]) / dxi;
00809             uAux[j + 3 - gLW * dPD] = s3f(&uAux[j + 3]) / dxi;
00810             uAux[j + 4 - gLW * dPD] = s3f(&uAux[j + 4]) / dxi;
00811             uAux[j + 5 - gLW * dPD] = s3f(&uAux[j + 5]) / dxi;
00812         }
00813     }
00814     break;
00815 case 4:
00816     for (int i = 0; i < perpPlainSize; i++) {
00817         for (int j = (i * dirWidth + gLW) * dPD;
00818             j < (i * dirWidth + gLW + dirWidthO) * dPD; j += dPD) {
00819             uAux[j + 0 - gLW * dPD] = s4b(&uAux[j + 0]) / dxi;
00820             uAux[j + 1 - gLW * dPD] = s4b(&uAux[j + 1]) / dxi;
00821             uAux[j + 2 - gLW * dPD] = s4f(&uAux[j + 2]) / dxi;
00822             uAux[j + 3 - gLW * dPD] = s4f(&uAux[j + 3]) / dxi;
00823             uAux[j + 4 - gLW * dPD] = s4c(&uAux[j + 4]) / dxi;
00824             uAux[j + 5 - gLW * dPD] = s4c(&uAux[j + 5]) / dxi;
00825         }
00826     }
00827     break;
00828 case 5:
00829     for (int i = 0; i < perpPlainSize; i++) {
00830         for (int j = (i * dirWidth + gLW) * dPD;
00831             j < (i * dirWidth + gLW + dirWidthO) * dPD; j += dPD) {
00832             uAux[j + 0 - gLW * dPD] = s5b(&uAux[j + 0]) / dxi;
00833             uAux[j + 1 - gLW * dPD] = s5b(&uAux[j + 1]) / dxi;
00834             uAux[j + 2 - gLW * dPD] = s5f(&uAux[j + 2]) / dxi;
00835             uAux[j + 3 - gLW * dPD] = s5f(&uAux[j + 3]) / dxi;
00836             uAux[j + 4 - gLW * dPD] = s5f(&uAux[j + 4]) / dxi;
00837             uAux[j + 5 - gLW * dPD] = s5f(&uAux[j + 5]) / dxi;
00838         }
00839     }
00840     break;
00841 case 6:
00842     for (int i = 0; i < perpPlainSize; i++) {
00843         for (int j = (i * dirWidth + gLW) * dPD;
00844             j < (i * dirWidth + gLW + dirWidthO) * dPD; j += dPD) {
00845             uAux[j + 0 - gLW * dPD] = s6b(&uAux[j + 0]) / dxi;
00846             uAux[j + 1 - gLW * dPD] = s6b(&uAux[j + 1]) / dxi;
00847             uAux[j + 2 - gLW * dPD] = s6f(&uAux[j + 2]) / dxi;
00848             uAux[j + 3 - gLW * dPD] = s6f(&uAux[j + 3]) / dxi;
00849             uAux[j + 4 - gLW * dPD] = s6c(&uAux[j + 4]) / dxi;
00850             uAux[j + 5 - gLW * dPD] = s6c(&uAux[j + 5]) / dxi;
00851         }
00852     }
00853     break;
00854 case 7:
00855     for (int i = 0; i < perpPlainSize; i++) {
00856         for (int j = (i * dirWidth + gLW) * dPD;
00857             j < (i * dirWidth + gLW + dirWidthO) * dPD; j += dPD) {
00858             uAux[j + 0 - gLW * dPD] = s7b(&uAux[j + 0]) / dxi;
00859             uAux[j + 1 - gLW * dPD] = s7b(&uAux[j + 1]) / dxi;
00860             uAux[j + 2 - gLW * dPD] = s7f(&uAux[j + 2]) / dxi;
00861             uAux[j + 3 - gLW * dPD] = s7f(&uAux[j + 3]) / dxi;
00862             uAux[j + 4 - gLW * dPD] = s7f(&uAux[j + 4]) / dxi;
00863             uAux[j + 5 - gLW * dPD] = s7f(&uAux[j + 5]) / dxi;

```

```

00864     }
00865     }
00866     break;
00867 case 8:
00868     for (int i = 0; i < perpPlainSize; i++) {
00869         for (int j = (i * dirWidth + gLW) * dPD;
00870             j < (i * dirWidth + gLW + dirWidthO) * dPD; j += dPD) {
00871             uAux[j + 0 - gLW * dPD] = s8b(&uAux[j + 0]) / dxi;
00872             uAux[j + 1 - gLW * dPD] = s8b(&uAux[j + 1]) / dxi;
00873             uAux[j + 2 - gLW * dPD] = s8f(&uAux[j + 2]) / dxi;
00874             uAux[j + 3 - gLW * dPD] = s8f(&uAux[j + 3]) / dxi;
00875             uAux[j + 4 - gLW * dPD] = s8c(&uAux[j + 4]) / dxi;
00876             uAux[j + 5 - gLW * dPD] = s8c(&uAux[j + 5]) / dxi;
00877         }
00878     }
00879     break;
00880 case 9:
00881     for (int i = 0; i < perpPlainSize; i++) {
00882         for (int j = (i * dirWidth + gLW) * dPD;
00883             j < (i * dirWidth + gLW + dirWidthO) * dPD; j += dPD) {
00884             uAux[j + 0 - gLW * dPD] = s9b(&uAux[j + 0]) / dxi;
00885             uAux[j + 1 - gLW * dPD] = s9b(&uAux[j + 1]) / dxi;
00886             uAux[j + 2 - gLW * dPD] = s9f(&uAux[j + 2]) / dxi;
00887             uAux[j + 3 - gLW * dPD] = s9f(&uAux[j + 3]) / dxi;
00888             uAux[j + 4 - gLW * dPD] = s9f(&uAux[j + 4]) / dxi;
00889             uAux[j + 5 - gLW * dPD] = s9f(&uAux[j + 5]) / dxi;
00890         }
00891     }
00892     break;
00893 case 10:
00894     for (int i = 0; i < perpPlainSize; i++) {
00895         for (int j = (i * dirWidth + gLW) * dPD;
00896             j < (i * dirWidth + gLW + dirWidthO) * dPD; j += dPD) {
00897             uAux[j + 0 - gLW * dPD] = s10b(&uAux[j + 0]) / dxi;
00898             uAux[j + 1 - gLW * dPD] = s10b(&uAux[j + 1]) / dxi;
00899             uAux[j + 2 - gLW * dPD] = s10f(&uAux[j + 2]) / dxi;
00900             uAux[j + 3 - gLW * dPD] = s10f(&uAux[j + 3]) / dxi;
00901             uAux[j + 4 - gLW * dPD] = s10c(&uAux[j + 4]) / dxi;
00902             uAux[j + 5 - gLW * dPD] = s10c(&uAux[j + 5]) / dxi;
00903         }
00904     }
00905     break;
00906 case 11:
00907     for (int i = 0; i < perpPlainSize; i++) {
00908         for (int j = (i * dirWidth + gLW) * dPD;
00909             j < (i * dirWidth + gLW + dirWidthO) * dPD; j += dPD) {
00910             uAux[j + 0 - gLW * dPD] = s11b(&uAux[j + 0]) / dxi;
00911             uAux[j + 1 - gLW * dPD] = s11b(&uAux[j + 1]) / dxi;
00912             uAux[j + 2 - gLW * dPD] = s11f(&uAux[j + 2]) / dxi;
00913             uAux[j + 3 - gLW * dPD] = s11f(&uAux[j + 3]) / dxi;
00914             uAux[j + 4 - gLW * dPD] = s11f(&uAux[j + 4]) / dxi;
00915             uAux[j + 5 - gLW * dPD] = s11f(&uAux[j + 5]) / dxi;
00916         }
00917     }
00918     break;
00919 case 12:
00920     for (int i = 0; i < perpPlainSize; i++) {
00921         for (int j = (i * dirWidth + gLW) * dPD;
00922             j < (i * dirWidth + gLW + dirWidthO) * dPD; j += dPD) {
00923             uAux[j + 0 - gLW * dPD] = s12b(&uAux[j + 0]) / dxi;
00924             uAux[j + 1 - gLW * dPD] = s12b(&uAux[j + 1]) / dxi;
00925             uAux[j + 2 - gLW * dPD] = s12f(&uAux[j + 2]) / dxi;
00926             uAux[j + 3 - gLW * dPD] = s12f(&uAux[j + 3]) / dxi;
00927             uAux[j + 4 - gLW * dPD] = s12c(&uAux[j + 4]) / dxi;
00928             uAux[j + 5 - gLW * dPD] = s12c(&uAux[j + 5]) / dxi;
00929         }
00930     }
00931     break;
00932 case 13:
00933     // #pragma omp parallel for default(none) firstprivate(uAux)
00934     // shared(dxi, dirWidth, dirWidthO, gLW, dPD) collapse(2) schedule(static, 6)
00935     // #pragma ivdep
00936     // #pragma distribute_point -> No.
00937     // #pragma unroll_and_jam
00938     // Iterate through all points in the plane perpendicular to the given
00939     // direction
00940     for (int i = 0; i < perpPlainSize; i++) {
00941         // stencil functions range over 2*gLW+6 indices, attention to cache-line
00942         // false-sharing
00943         // #pragma omp simd safelen(2*gLW+dPD)
00944         // Iterate through the direction for each perpendicular plane point
00945         for (int j = (i * dirWidth + gLW /*to shift left by gLW below */) * dPD;
00946             j < (i * dirWidth + gLW + dirWidthO) * dPD; j += dPD) {
00947             /* Compute the stencil derivative for any of the six field components
00948              * with a ghostlayer width adjusted to the order of the finite
00949              * difference scheme */
00950             uAux[j + 0 - gLW * dPD] = s13b(&uAux[j + 0]) / dxi;

```

```

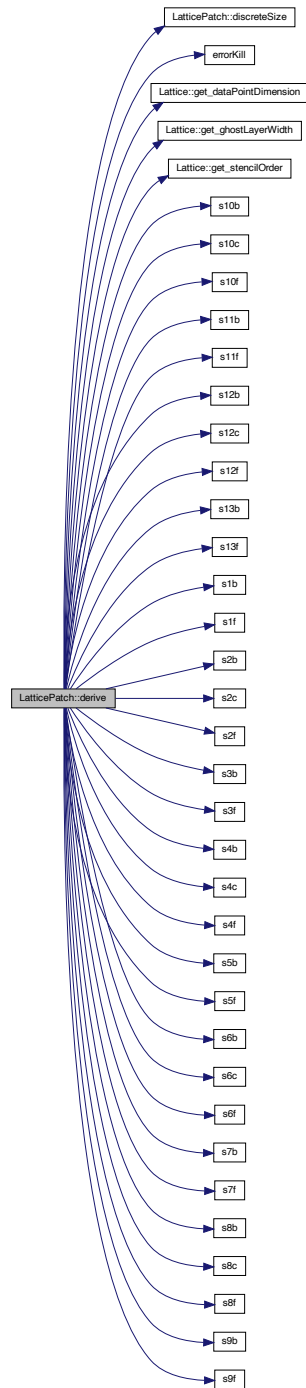
00951         uAux[j + 1 - gLW * dPD] = s13b(&uAux[j + 1]) / dxi;
00952         uAux[j + 2 - gLW * dPD] = s13f(&uAux[j + 2]) / dxi;
00953         uAux[j + 3 - gLW * dPD] = s13f(&uAux[j + 3]) / dxi;
00954         uAux[j + 4 - gLW * dPD] = s13f(&uAux[j + 4]) / dxi;
00955         uAux[j + 5 - gLW * dPD] = s13f(&uAux[j + 5]) / dxi;
00956     }
00957 }
00958 break;
00959
00960 default:
00961     errorKill("Please set an existing stencil order");
00962     break;
00963 }
00964 }

```

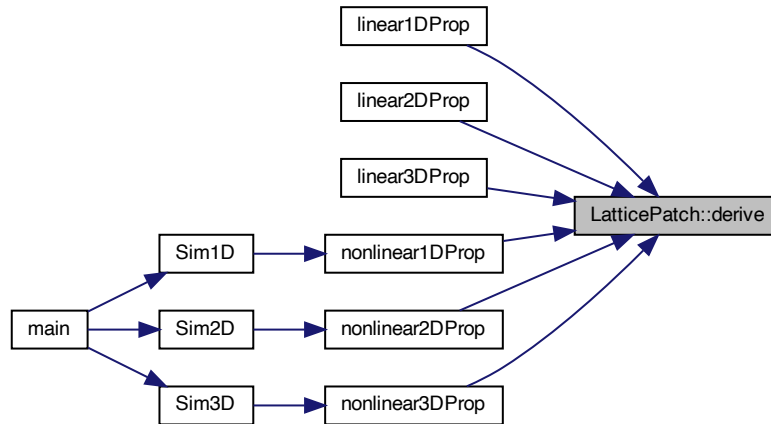
References [discreteSize\(\)](#), [dx](#), [dy](#), [dz](#), [envelopeLattice](#), [errorKill\(\)](#), [Lattice::get_dataPointDimension\(\)](#), [Lattice::get_ghostLayerWidth\(\)](#), [Lattice::get_stencilOrder\(\)](#), [s10b\(\)](#), [s10c\(\)](#), [s10f\(\)](#), [s11b\(\)](#), [s11f\(\)](#), [s12b\(\)](#),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



5.9.3.4 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 426 of file [LatticePatch.cpp](#).

```
00426                                     {
00427     // Check that the lattice as well as the translocation lookups have been set
00428     // up;
00429     checkFlag(FLatticePatchSetUp);
00430     checkFlag(TranslocationLookupSetUp);
00431     const int dPD = envelopeLattice->get_dataPointDimension();
00432     const int gLW = envelopeLattice->get_ghostLayerWidth();
00433     const int uSize = discreteSize();
00434     int ii = 0, target = 0;
00435     switch (dir) {
00436     case 1:
00437     #pragma ivdep
00438     #pragma omp simd // safelen(6) - also good
00439     #pragma distribute_point
00440         for (int i = 0; i < uSize; i++) {
00441             // get correct indices in u and rotation space
00442             target = dPD * i;
00443             ii = dPD * (uTox[i] - gLW);
00444             buffOut[target + 0] = uAux[5 + ii];
00445             buffOut[target + 1] = (-uAux[ii] + uAux[2 + ii]) / 2.;
00446             buffOut[target + 2] = (uAux[1 + ii] - uAux[3 + ii]) / 2.;
00447             buffOut[target + 3] = uAux[4 + ii];
00448             buffOut[target + 4] = (uAux[1 + ii] + uAux[3 + ii]) / 2.;
00449             buffOut[target + 5] = (uAux[ii] + uAux[2 + ii]) / 2.;
00450         }
00451         break;
00452     case 2:
00453     #pragma omp simd // safelen(6)
00454     #pragma distribute_point
```

```

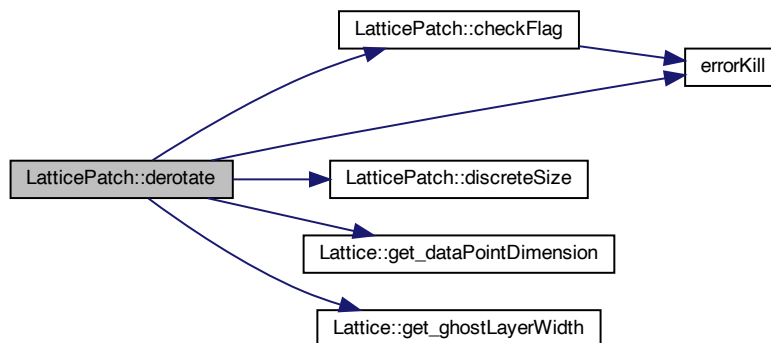
00455     for (int i = 0; i < uSize; i++) {
00456         target = dPD * i;
00457         ii = dPD * (uToy[i] - gLW);
00458         buffOut[target + 0] = (uAux[ii] - uAux[2 + ii]) / 2.;
00459         buffOut[target + 1] = uAux[5 + ii];
00460         buffOut[target + 2] = (-uAux[1 + ii] + uAux[3 + ii]) / 2.;
00461         buffOut[target + 3] = (uAux[1 + ii] + uAux[3 + ii]) / 2.;
00462         buffOut[target + 4] = uAux[4 + ii];
00463         buffOut[target + 5] = (uAux[ii] + uAux[2 + ii]) / 2.;
00464     }
00465     break;
00466 case 3:
00467 #pragma omp simd // safelen(6)
00468 #pragma distribute_point
00469     for (int i = 0; i < uSize; i++) {
00470         target = dPD * i;
00471         ii = dPD * (uToz[i] - gLW);
00472         buffOut[target + 0] = (-uAux[ii] + uAux[2 + ii]) / 2.;
00473         buffOut[target + 1] = (uAux[1 + ii] - uAux[3 + ii]) / 2.;
00474         buffOut[target + 2] = uAux[5 + ii];
00475         buffOut[target + 3] = (uAux[1 + ii] + uAux[3 + ii]) / 2.;
00476         buffOut[target + 4] = (uAux[ii] + uAux[2 + ii]) / 2.;
00477         buffOut[target + 5] = uAux[4 + ii];
00478     }
00479     break;
00480 default:
00481     errorKill("Tried to derotate from the wrong direction");
00482     break;
00483 }
00484 }

```

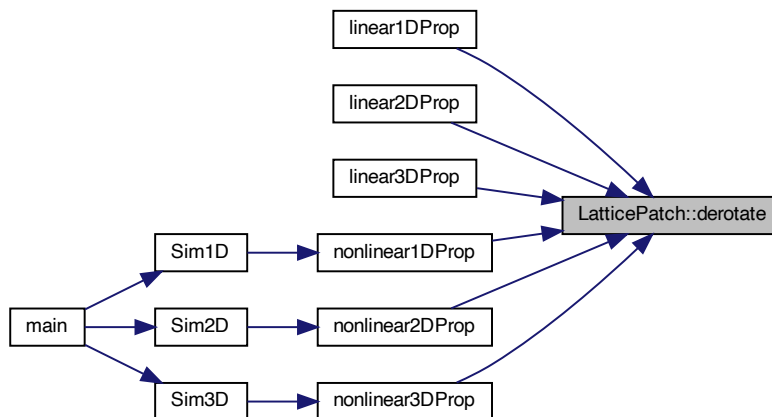
References [checkFlag\(\)](#), [discreteSize\(\)](#), [envelopeLattice](#), [errorKill\(\)](#), [FLatticePatchSetUp](#), [Lattice::get_dataPointDimension\(\)](#), [Lattice::get_ghostLayerWidth\(\)](#), [TranslocationLookupSetUp](#), [uAux](#), [uTox](#), [uToy](#), and [uToz](#).

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:



5.9.3.5 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

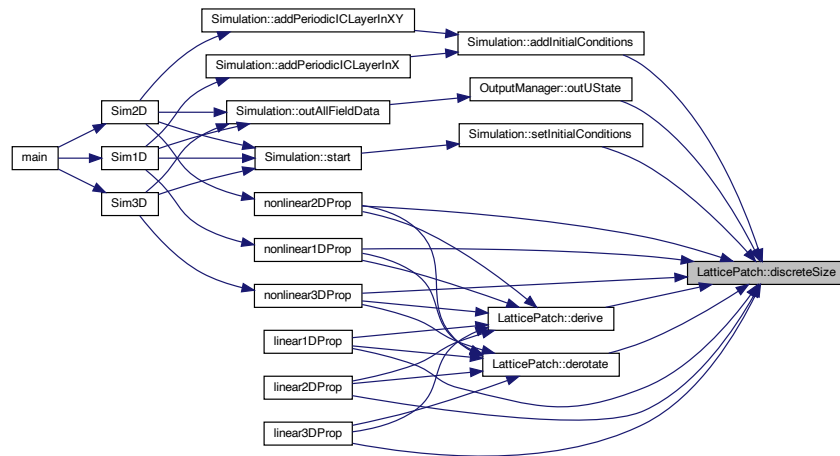
Definition at line 196 of file [LatticePatch.cpp](#).

```
00196 {
00197     switch (dir) {
00198     case 0:
00199         return nx * ny * nz;
00200     case 1:
00201         return nx;
00202     case 2:
00203         return ny;
00204     case 3:
00205         return nz;
00206     // case 4: return uAux.size(); // for debugging
00207     default:
00208         return -1;
00209     }
00210 }
```

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:



5.9.3.6 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 502 of file [LatticePatch.cpp](#).

```
00502 {
00503     // Check that the lattice has been set up
00504     checkFlag(FLatticeDimensionSet);
00505     checkFlag(FLatticePatchSetUp);
00506     // Variables to per dimension calculate the halo indices, and distance to
00507     // other side halo boundary
00508     int mx = 1, my = 1, mz = 1, distToRight = 1;
00509     const int gLW = envelopeLattice->get_ghostLayerWidth();
00510     // In the chosen direction m is set to ghost layer width while the others
00511     // remain to form the plane
00512     switch (dir) {
00513     case 1:
00514         mx = gLW;
00515         my = ny;
00516         mz = nz;
00517         distToRight = (nx - gLW);
00518         break;
00519     case 2:
00520         mx = nx;
00521         my = gLW;
00522         mz = nz;
00523         distToRight = nx * (ny - gLW);
00524         break;
00525     case 3:
00526         mx = nx;
00527         my = ny;
00528         mz = gLW;
00529         distToRight = nx * ny * (nz - gLW);
00530         break;
00531     }
00532     // total number of exchanged points
00533     const int dPD = envelopeLattice->get_dataPointDimension();
00534     const int exchangeSize = mx * my * mz * dPD;
00535     // provide size of the halos for ghost cells
```

```

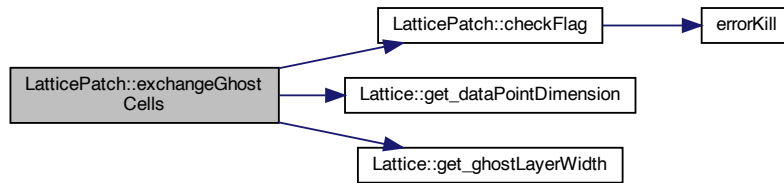
00536     ghostCellLeft.resize(exchangeSize);
00537     ghostCellRight.resize(ghostCellLeft.size());
00538     ghostCellLeftToSend.resize(ghostCellLeft.size());
00539     ghostCellRightToSend.resize(ghostCellLeft.size());
00540     gCLData = &ghostCellLeft[0];
00541     gCRData = &ghostCellRight[0];
00542     statusFlags |= GhostLayersInitialized;
00543
00544     // Initialize running index li for the halo buffers, and index ui of uData for
00545     // data transfer
00546     int li = 0, ui = 0;
00547
00548     // #pragma omp parallel for reduction(+:ui) reduction(+:li) -> don't probably
00549     // bad idea to parallelize ghost cell exchange Loop over to be copied points in
00550     // z and y direction
00551     #pragma distribute_point
00552     for (int iz = 0; iz < mz; iz++) {
00553         for (int iy = 0; iy < my; iy++) {
00554             // uData vector start index of halo data to be transferred
00555             // with each z-step add the whole xy-plane and with y-step the x-range ->
00556             // iterate all x-ranges
00557             ui = (iz * nx * ny + iy * nx) * dPD;
00558             // copy left halo data from uData to buffer, transfer size is given by
00559             // x-length (not x-range) perhaps faster but more fragile C lib copy
00560             // operation (contained in cstring header)
00561             /*
00562             memcpy(&ghostCellLeftToSend[li],
00563                 &uData[ui],
00564                 sizeof(sunrealtype)*mx*dPD);
00565             // increase ui by the distance to vis-a-vis boundary and copy right halo
00566             data to buffer ui+=distToRight*dPD; memcpy(&ghostCellRightToSend[li],
00567                 &uData[ui],
00568                 sizeof(sunrealtype)*mx*dPD);
00569             */
00570             // perhaps more safe but slower copy operation (contained in algorithm
00571             // header) performance highly system dependent
00572             copy(&uData[ui], &uData[ui + mx * dPD], &ghostCellLeftToSend[li]);
00573             ui += distToRight * dPD;
00574             copy(&uData[ui], &uData[ui + mx * dPD], &ghostCellRightToSend[li]);
00575
00576             // increase halo index by transferred items per y-iteration step
00577             // (x-length)
00578             li += mx * dPD;
00579         }
00580     }
00581
00582     /* Send and receive the data to and from neighboring latticePatches */
00583     // Adjust direction to cartesian communicator
00584     int dim = 2; // default for dir==1
00585     if (dir == 2) {
00586         dim = 1;
00587     } else if (dir == 3) {
00588         dim = 0;
00589     }
00590     int rank_source = 0, rank_dest = 0;
00591     MPI_Cart_shift(envelopeLattice->comm, dim, -1, &rank_source,
00592         &rank_dest); // s.t. rank_dest is left & v.v.
00593
00594     // nonblocking Isend/Irecv
00595
00596     MPI_Request requests[4];
00597     MPI_Irecv(&ghostCellRight[0], exchangeSize, MPI_SUNREALTYPE, rank_source, 1,
00598         envelopeLattice->comm, &requests[0]);
00599     MPI_Isend(&ghostCellLeftToSend[0], exchangeSize, MPI_SUNREALTYPE, rank_dest,
00600         1, envelopeLattice->comm, &requests[1]);
00601     MPI_Irecv(&ghostCellLeft[0], exchangeSize, MPI_SUNREALTYPE, rank_dest, 2,
00602         envelopeLattice->comm, &requests[2]);
00603     MPI_Isend(&ghostCellRightToSend[0], exchangeSize, MPI_SUNREALTYPE,
00604         rank_source, 2, envelopeLattice->comm, &requests[3]);
00605     MPI_Waitall(4, requests, MPI_STATUS_IGNORE);
00606
00607
00608     // blocking Sendrecv:
00609     /*
00610     MPI_Sendrecv(&ghostCellLeftToSend[0], exchangeSize, MPI_SUNREALTYPE,
00611         rank_dest, 1, &ghostCellRight[0], exchangeSize, MPI_SUNREALTYPE,
00612         rank_source, 1, envelopeLattice->comm, MPI_STATUS_IGNORE);
00613     MPI_Sendrecv(&ghostCellRightToSend[0], exchangeSize, MPI_SUNREALTYPE,
00614         rank_source, 2, &ghostCellLeft[0], exchangeSize, MPI_SUNREALTYPE,
00615         rank_dest, 2, envelopeLattice->comm, MPI_STATUS_IGNORE);
00616     */
00617 }

```

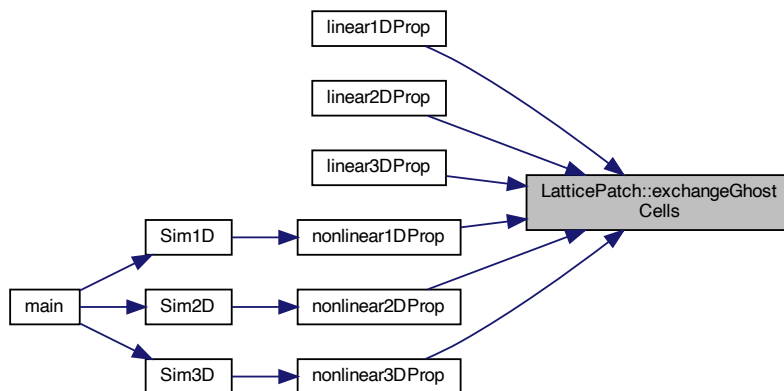
References [checkFlag\(\)](#), [Lattice::comm](#), [envelopeLattice](#), [FLatticeDimensionSet](#), [FLatticePatchSetUp](#), [gCLData](#), [gCRData](#), [Lattice::get_dataPointDimension\(\)](#), [Lattice::get_ghostLayerWidth\(\)](#), [ghostCellLeft](#), [ghostCellLeftToSend](#), [ghostCellRight](#),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



5.9.3.7 exchangeGhostCells3D()

```
void LatticePatch::exchangeGhostCells3D ( )
```

function to exchange ghost cells using a neighborhood collective operation

Exchange ghost cells with a neighborhood collective operation.

Definition at line 620 of file [LatticePatch.cpp](#).

```

00620     {
00621     // Check that the lattice has been set up
00622     checkFlag(FLatticeDimensionSet);
00623     // ghostlayerwidth
00624     const int gLW = envelopeLattice->get_ghostLayerWidth();
00625     // datapoint dimension
00626     const int dPD = envelopeLattice->get_dataPointDimension();
00627     // total number of exchanged points per halo
00628     const int n = nx; // only cubic patches allowed -> use general length n
  
```

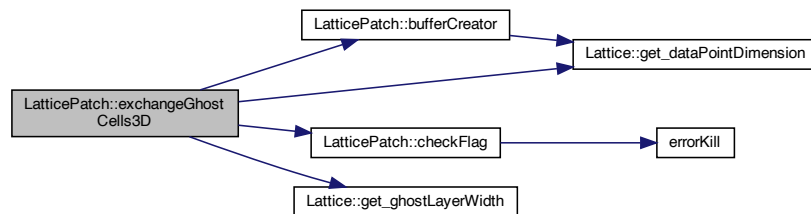
```

00629  const int exchangeSize = n * n * gLW * dPD;
00630  // Give ghostCells the total size of the ghost layers (the six halos)
00631  const int tot_exchangeSize = 6 * exchangeSize;
00632  ghostCells.resize(tot_exchangeSize);
00633  ghostCellsToSend.resize(ghostCells.size());
00634  // ghost cell data in all directions: left,right,bottom,top,front,abft; but
00635  // with MPI dim order "lefts" are the first receivers and "rights" are the
00636  // first senders -> see buffer creator below
00637  gCFData = &ghostCells[0];
00638  gCADATA = &ghostCells[exchangeSize];
00639  gCBData = &ghostCells[2 * exchangeSize];
00640  gCTData = &ghostCells[3 * exchangeSize];
00641  gCLData = &ghostCells[4 * exchangeSize];
00642  gCRData = &ghostCells[5 * exchangeSize];
00643  statusFlags |= GhostLayersInitialized;
00644
00645  checkFlag(FLatticePatchSetUp);
00646  // variables to set to ghost layer width and point distance to next
00647  // communication point -> depends on direction
00648
00649  int li = 0; // running index for buffers
00650  int distToRight = 0; // distance to vis-a-vis halo data, varies per dim
00651  // filling buffers along the MPI dim order
00652  distToRight = n * n * (n - gLW);
00653  bufferCreator(li, n, n, gLW, distToRight);
00654  li += 2 * exchangeSize; // li increases by two exchange sizes per dim
00655
00656  distToRight = n * (n - gLW);
00657  bufferCreator(li, n, gLW, n, distToRight);
00658  li += 2 * exchangeSize;
00659
00660  distToRight = (n - gLW);
00661  bufferCreator(li, gLW, n, n, distToRight);
00662
00663  MPI_Neighbor_alltoall(&ghostCellsToSend[0], exchangeSize, MPI_SUNREALTYPE,
00664                      &ghostCells[0], exchangeSize, MPI_SUNREALTYPE,
00665                      envelopeLattice->comm);
00666 }

```

References [bufferCreator\(\)](#), [checkFlag\(\)](#), [Lattice::comm](#), [envelopeLattice](#), [FLatticeDimensionSet](#), [FLatticePatchSetUp](#), [gCADATA](#), [gCBData](#), [gCFData](#), [gCLData](#), [gCRData](#), [gCTData](#), [Lattice::get_dataPointDimension\(\)](#), [Lattice::get_ghostLayerWidth\(\)](#), [ghostCells](#), [ghostCellsToSend](#), [GhostLayersInitialized](#), [nx](#), and [statusFlags](#).

Here is the call graph for this function:



5.9.3.8 generateTranslocationLookup()

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

function to fill out the lookup tables for translocation

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

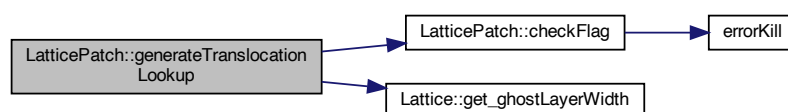
```

00246 {
00247     // Check that the lattice has been set up
00248     checkFlag(FLatticeDimensionSet);
00249     // lenghts for auxilliary layers, including ghost layers
00250     const int gLW = envelopeLattice->get_ghostLayerWidth();
00251     const int mx = nx + 2 * gLW;
00252     const int my = ny + 2 * gLW;
00253     const int mz = nz + 2 * gLW;
00254     // sizes for lookup vectors
00255     // generate u->uAux
00256     uTox.resize(nx * ny * nz);
00257     uToy.resize(nx * ny * nz);
00258     uToz.resize(nx * ny * nz);
00259     // generate uAux->u with length including halo
00260     xTou.resize(mx * ny * nz);
00261     yTou.resize(nx * my * nz);
00262     zTou.resize(nx * ny * mz);
00263     // variables for cartesian position in the 3D discrete lattice
00264     int px = 0, py = 0, pz = 0;
00265     for (unsigned int i = 0; i < uToy.size(); i++) { // loop over all points in the patch
00266         // calculate cartesian coordinates
00267         px = i % nx;
00268         py = (i / nx) % ny;
00269         pz = (i / nx) / ny;
00270         // fill lookups extended by halos (useful for y and z direction)
00271         uTox[i] = (px + gLW) + py * mx +
00272             pz * mx * ny; // unroll (de-flatten) cartesian dimension
00273         xTou[px + py * mx + pz * mx * ny] =
00274             i; // match cartesian point to u location
00275         uToy[i] = (py + gLW) + pz * my + px * my * nz;
00276         yTou[py + pz * my + px * my * nz] = i;
00277         uToz[i] = (pz + gLW) + px * mz + py * mz * nx;
00278         zTou[pz + px * mz + py * mz * nx] = i;
00279     }
00280     // same for ghost layer lookup tables
00281     lgcTox.resize(gLW * ny * nz);
00282     rgcTox.resize(gLW * ny * nz);
00283     for (unsigned int i = 0; i < lgcTox.size(); i++) {
00284         px = i % gLW;
00285         py = (i / gLW) % ny;
00286         pz = (i / gLW) / ny;
00287         lgcTox[i] = px + py * mx + pz * mx * ny;
00288         rgcTox[i] = px + nx + gLW + py * mx + pz * mx * ny;
00289     }
00290     lgcToy.resize(gLW * nx * nz);
00291     rgcToy.resize(gLW * nx * nz);
00292     for (unsigned int i = 0; i < lgcToy.size(); i++) {
00293         px = i % nx;
00294         py = (i / nx) % gLW;
00295         pz = (i / nx) / gLW;
00296         lgcToy[i] = py + pz * my + px * my * nz;
00297         rgcToy[i] = py + ny + gLW + pz * my + px * my * nz;
00298     }
00299     lgcToz.resize(gLW * nx * ny);
00300     rgcToz.resize(gLW * nx * ny);
00301     for (unsigned int i = 0; i < lgcToz.size(); i++) {
00302         px = i % nx;
00303         py = (i / nx) % ny;
00304         pz = (i / nx) / ny;
00305         lgcToz[i] = pz + px * mz + py * mz * nx;
00306         rgcToz[i] = pz + nz + gLW + px * mz + py * mz * nx;
00307     }
00308     statusFlags |= TranslocationLookupSetUp;
00309 }

```

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:



5.9.3.9 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 228 of file [LatticePatch.cpp](#).

```
00228                                     {
00229     switch (dir) {
00230     case 1:
00231         return dx;
00232     case 2:
00233         return dy;
00234     case 3:
00235         return dz;
00236     default:
00237         errorKill(
00238             "LatticePatch::getDelta function called with wrong dir parameter");
00239         return -1;
00240     }
00241 }
```

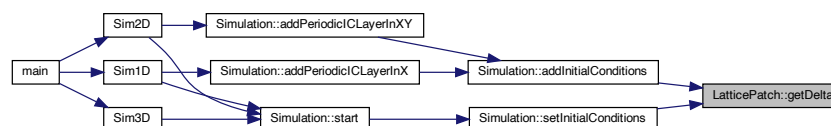
References [dx](#), [dy](#), [dz](#), and [errorKill\(\)](#).

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

Here is the call graph for this function:



Here is the caller graph for this function:



5.9.3.10 initializeBuffers()

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

initialize buffers to save derivatives

Create buffers to save derivative values, optimizing computational load.

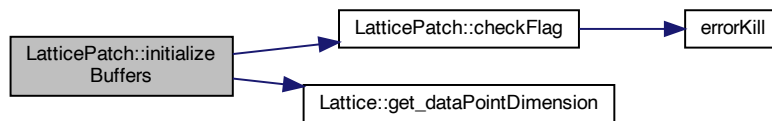
Definition at line 487 of file [LatticePatch.cpp](#).

```
00487 {
00488     // Check that the lattice has been set up
00489     checkFlag(FLatticeDimensionSet);
00490     const int dPD = envelopeLattice->get_dataPointDimension();
00491     buffX.resize(nx * ny * nz * dPD);
00492     buffY.resize(nx * ny * nz * dPD);
00493     buffZ.resize(nx * ny * nz * dPD);
00494     // Set pointers used for propagation functions
00495     buffData[0] = &buffX[0];
00496     buffData[1] = &buffY[0];
00497     buffData[2] = &buffZ[0];
00498     statusFlags |= BuffersInitialized;
00499 }
```

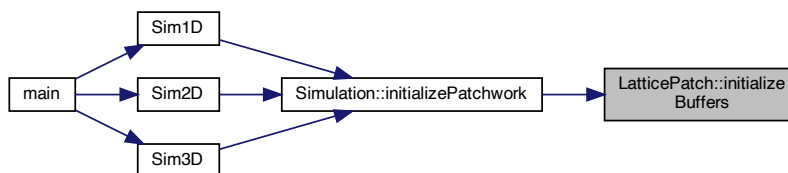
References [buffData](#), [BuffersInitialized](#), [buffX](#), [buffY](#), [buffZ](#), [checkFlag\(\)](#), [envelopeLattice](#), [FLatticeDimensionSet](#), [Lattice::get_dataPointDimension\(\)](#), [nx](#), [ny](#), [nz](#), and [statusFlags](#).

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

Here is the call graph for this function:



Here is the caller graph for this function:



5.9.3.11 initializeGhostLayer()

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

initialize ghost cells for halo exchange

5.9.3.12 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 213 of file [LatticePatch.cpp](#).

```
00213 {
00214     switch (dir) {
00215     case 1:
00216         return x0;
00217     case 2:
00218         return y0;
00219     case 3:
00220         return z0;
00221     default:
00222         errorKill("LatticePatch::origin function called with wrong dir parameter");
00223         return -1;
00224     }
00225 }
```

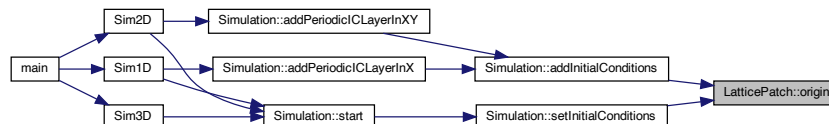
References [errorKill\(\)](#), [x0](#), [y0](#), and [z0](#).

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

Here is the call graph for this function:



Here is the caller graph for this function:



5.9.3.13 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

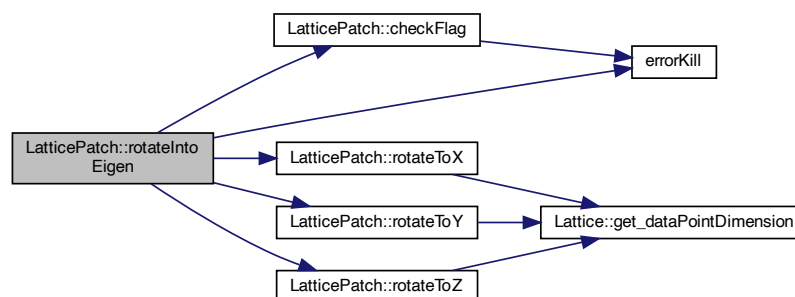
Definition at line 314 of file [LatticePatch.cpp](#).

```
00314 {
00315     // Check that the lattice, ghost layers as well as the translocation lookups
00316     // have been set up;
00317     checkFlag(FLatticePatchSetUp);
00318     checkFlag(TranslocationLookupSetUp);
00319     checkFlag(GhostLayersInitialized); // this check is only after call to
00320                                         // exchange ghost cells
00321     switch (dir) {
00322     case 1:
00323         rotateToX(uAuxData, gCLData, lgcTox);
00324         rotateToX(uAuxData, uData, uTox);
00325         rotateToX(uAuxData, gCRData, rgcTox);
00326         break;
00327     case 2:
00328         rotateToY(uAuxData, gCLData, lgcToy);
00329         rotateToY(uAuxData, uData, uToy);
00330         rotateToY(uAuxData, gCRData, rgcToy);
00331         break;
00332     case 3:
00333         rotateToZ(uAuxData, gCLData, lgcToz);
00334         rotateToZ(uAuxData, uData, uToz);
00335         rotateToZ(uAuxData, gCRData, rgcToz);
00336         break;
00337     default:
00338         errorKill("Tried to rotate into the wrong direction");
00339         break;
00340     }
00341 }
```

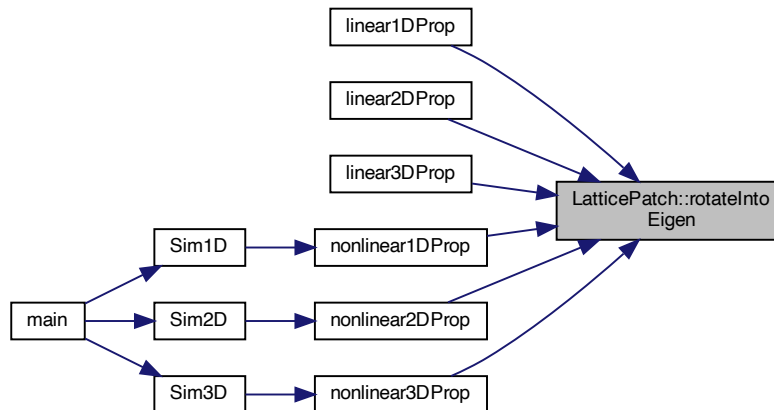
References [checkFlag\(\)](#), [errorKill\(\)](#), [FLatticePatchSetUp](#), [gCLData](#), [gCRData](#), [GhostLayersInitialized](#), [lgcTox](#), [lgcToy](#), [lgcToz](#), [rgcTox](#), [rgcToy](#), [rgcToz](#), [rotateToX\(\)](#), [rotateToY\(\)](#), [rotateToZ\(\)](#), [TranslocationLookupSetUp](#),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



5.9.3.14 rotateIntoEigen3D()

```
void LatticePatch::rotateIntoEigen3D ( )
```

function to rotate as in `rotateIntoEigen` with special 3D halo buffers

Same as `rotateIntoEigen` but for neighborhood 3D halo buffers.

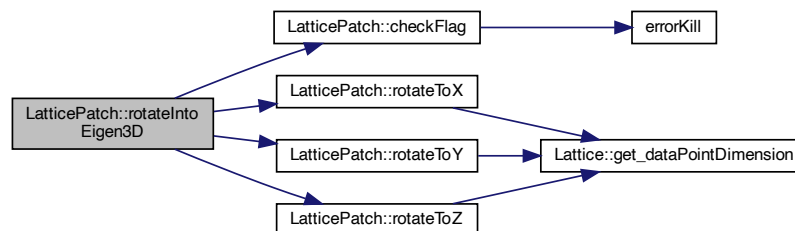
Definition at line 344 of file [LatticePatch.cpp](#).

```

00344 {
00345     checkFlag(FLatticePatchSetUp);
00346     checkFlag(TranslocationLookupSetUp);
00347     checkFlag(GhostLayersInitialized);
00348     rotateToX(uAuxData, gCLData, lgcTox);
00349     rotateToX(uAuxData, uData, uTox);
00350     rotateToX(uAuxData, gCRData, rgcTox);
00351     rotateToY(uAuxData, gCBData, lgcToy);
00352     rotateToY(uAuxData, uData, uToy);
00353     rotateToY(uAuxData, gCTData, rgcToy);
00354     rotateToZ(uAuxData, gCFData, lgcToz);
00355     rotateToZ(uAuxData, uData, uToz);
00356     rotateToZ(uAuxData, gCAData, rgcToz);
00357 }
```

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

Here is the call graph for this function:



5.9.3.15 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 361 of file [LatticePatch.cpp](#).

```
00363 {
00364     int ii = 0, target = 0;
00365     #pragma ivdep
00366     #pragma omp simd // safelen(6) - also good
00367     #pragma distribute_point
00368     for (unsigned int i = 0; i < lookup.size(); i++) {
00369         // get correct u-vector and spatial indices along previously defined lookup
00370         // tables
00371         target = envelopeLattice->get_dataPointDimension() * lookup[i];
00372         ii = envelopeLattice->get_dataPointDimension() * i;
00373         outArray[target + 0] = -inArray[1 + ii] + inArray[5 + ii];
00374         outArray[target + 1] = inArray[2 + ii] + inArray[4 + ii];
00375         outArray[target + 2] = inArray[1 + ii] + inArray[5 + ii];
00376         outArray[target + 3] = -inArray[2 + ii] + inArray[4 + ii];
00377         outArray[target + 4] = inArray[3 + ii];
00378         outArray[target + 5] = inArray[ii];
00379     }
00380 }
```

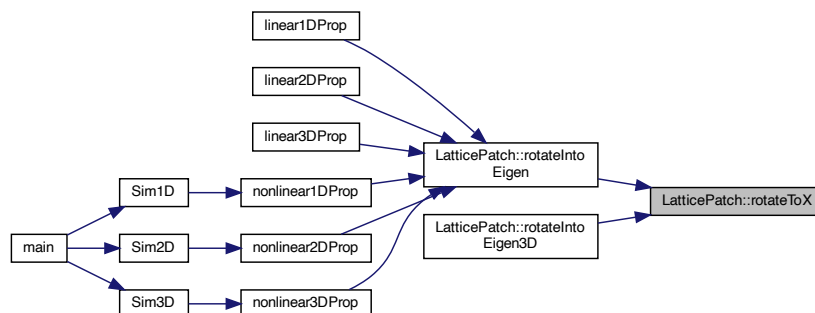
References [envelopeLattice](#), and [Lattice::get_dataPointDimension\(\)](#).

Referenced by [rotateIntoEigen\(\)](#), and [rotateIntoEigen3D\(\)](#).

Here is the call graph for this function:



Here is the caller graph for this function:



5.9.3.16 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 R_y into eigenspace of Z matrix and write to auxiliary vector

Definition at line 384 of file [LatticePatch.cpp](#).

```
00386                                     {
00387     int ii = 0, target = 0;
00388     #pragma ivdep
00389     #pragma omp simd // safelen(6)
00390     #pragma distribute_point
00391     for (unsigned int i = 0; i < lookup.size(); i++) {
00392         target = envelopeLattice->get_dataPointDimension() * lookup[i];
00393         ii = envelopeLattice->get_dataPointDimension() * i;
00394         outArray[target + 0] = inArray[ii] + inArray[5 + ii];
00395         outArray[target + 1] = -inArray[2 + ii] + inArray[3 + ii];
00396         outArray[target + 2] = -inArray[ii] + inArray[5 + ii];
00397         outArray[target + 3] = inArray[2 + ii] + inArray[3 + ii];
00398         outArray[target + 4] = inArray[4 + ii];
00399         outArray[target + 5] = inArray[1 + ii];
00400     }
00401 }
```

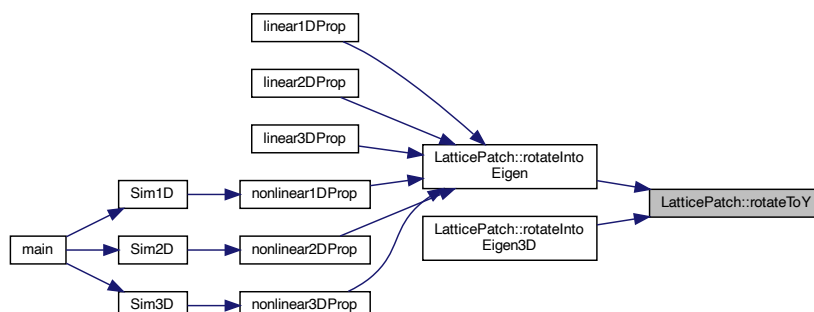
References [envelopeLattice](#), and [Lattice::get_dataPointDimension\(\)](#).

Referenced by [rotateIntoEigen\(\)](#), and [rotateIntoEigen3D\(\)](#).

Here is the call graph for this function:



Here is the caller graph for this function:



5.9.3.17 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 405 of file [LatticePatch.cpp](#).

```
00407
00408     int ii = 0, target = 0;
00409     #pragma ivdep
00410     #pragma omp simd // safelen(6)
00411     #pragma distribute_point
00412     for (unsigned int i = 0; i < lookup.size(); i++) {
00413         target = envelopeLattice->get_dataPointDimension() * lookup[i];
00414         ii = envelopeLattice->get_dataPointDimension() * i;
00415         outArray[target + 0] = -inArray[ii] + inArray[4 + ii];
00416         outArray[target + 1] = inArray[1 + ii] + inArray[3 + ii];
00417         outArray[target + 2] = inArray[ii] + inArray[4 + ii];
00418         outArray[target + 3] = -inArray[1 + ii] + inArray[3 + ii];
00419         outArray[target + 4] = inArray[5 + ii];
00420         outArray[target + 5] = inArray[2 + ii];
00421     }
00422 }
```

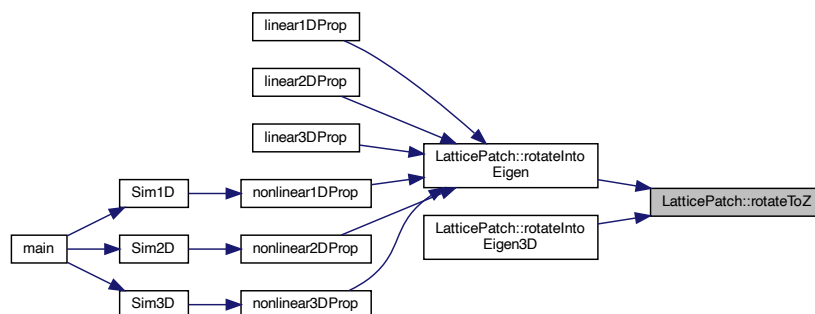
References [envelopeLattice](#), and [Lattice::get_dataPointDimension\(\)](#).

Referenced by [rotateIntoEigen\(\)](#), and [rotateIntoEigen3D\(\)](#).

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     /*
00140     // Experiment: Resolution can be adapted for each process/patch
00141     const int Scaler=4; // Better hand over as parameter via 'initializePatchwork'
00142     if(my_prc==0){patchToMold.dx=envelopeLattice.get_dx();}
00143     else if(my_prc==1){patchToMold.dx=envelopeLattice.get_dx()*Scaler;}
00144     else{errorKill("Only do this resolution barrier test with 2 processes.");}
00145     */
00146     patchToMold.dx = envelopeLattice.get_dx();
00147     patchToMold.dy = envelopeLattice.get_dy();
00148     patchToMold.dz = envelopeLattice.get_dz();
00149     patchToMold.x0 = firstXPoint * patchToMold.dx;
00150     patchToMold.y0 = firstYPoint * patchToMold.dy;
00151     patchToMold.z0 = firstZPoint * patchToMold.dz;
00152     patchToMold.LIx = LIx;
00153     patchToMold.LIy = LIy;
00154     patchToMold.LIz = LIz;
00155     patchToMold.nx = local_NOXP;
00156     patchToMold.ny = local_NOYP;
00157     patchToMold.nz = local_NOZP;
00158     patchToMold.lx = patchToMold.nx * patchToMold.dx;
00159     patchToMold.ly = patchToMold.ny * patchToMold.dy;
00160     patchToMold.lz = patchToMold.nz * patchToMold.dz;
00161     /* // Check name of lattice communicator
00162     char lattice_comm_name[MPI_MAX_OBJECT_NAME];
00163     int lattice_namelen;
00164     MPI_Comm_get_name(envelopeLattice.comm, lattice_comm_name, &lattice_namelen);
00165     cout<<"envelopeLattice.comm gives " << lattice_comm_name << endl;
00166     */
00167     // Create and allocate memory for parallel vectors with defined local and
00168     // global lenghts *
00169     * (-> CNode problem sizes Nlocal and N)
```

```

00170      * for field data and temporal derivatives and set extra pointers to them */
00171      patchToMold.u =
00172          N_VNew_Parallel(envelopeLattice.comm, local_NODP,
00173                          envelopeLattice.get_tot_noDP(), envelopeLattice.sunctx);
00174      patchToMold.uData = NV_DATA_P(patchToMold.u);
00175      patchToMold.du =
00176          N_VNew_Parallel(envelopeLattice.comm, local_NODP,
00177                          envelopeLattice.get_tot_noDP(), envelopeLattice.sunctx);
00178      patchToMold.duData = NV_DATA_P(patchToMold.du);
00179      // Allocate space for auxiliary uAux so that the lattice and all possible
00180      // directions of ghost layers fit
00181      const int s1 = patchToMold.nx, s2 = patchToMold.ny, s3 = patchToMold.nz;
00182      const int s_min = min(s1, min(s2, s3));
00183      patchToMold.uAux.resize(s1 * s2 * s3 / s_min * (s_min + 2 * gLW) * dPD);
00184      patchToMold.uAuxData = &patchToMold.uAux[0];
00185      patchToMold.envelopeLattice = &envelopeLattice;
00186      // Set patch "name" to process number -> only for debugging
00187      // patchToMold.ID=my_prc;
00188      // set flag
00189      patchToMold.statusFlags = FLatticePatchSetUp;
00190      patchToMold.generateTranslocationLookup();
00191      return 0;
00192 }

```

5.9.5 Field Documentation

5.9.5.1 buffData

array<sunrealtype *, 3> LatticePatch::buffData

pointer to spatial derivative data buffers

Definition at line 221 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 179 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 179 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 179 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 209 of file [LatticePatch.h](#).

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

5.9.5.6 duData

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

pointer to time-derivative data

Definition at line 215 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 164 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 166 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 168 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 170 of file [LatticePatch.h](#).

Referenced by [bufferCreator\(\)](#), [derive\(\)](#), [derotate\(\)](#), [exchangeGhostCells\(\)](#), [exchangeGhostCells3D\(\)](#), [generateTranslocationLookup\(\)](#), [initializeBuffers\(\)](#), [rotateToX\(\)](#), [rotateToY\(\)](#), and [rotateToZ\(\)](#).

5.9.5.11 gCAData

```
sunrealtype * LatticePatch::gCAData
```

pointer to halo data

Definition at line 218 of file [LatticePatch.h](#).

Referenced by [exchangeGhostCells3D\(\)](#), and [rotateIntoEigen3D\(\)](#).

5.9.5.12 gCBData

```
sunrealtype * LatticePatch::gCBData
```

pointer to halo data

Definition at line 218 of file [LatticePatch.h](#).

Referenced by [exchangeGhostCells3D\(\)](#), and [rotateIntoEigen3D\(\)](#).

5.9.5.13 gCFData

```
sunrealtype * LatticePatch::gCFData
```

pointer to halo data

Definition at line 218 of file [LatticePatch.h](#).

Referenced by [exchangeGhostCells3D\(\)](#), and [rotateIntoEigen3D\(\)](#).

5.9.5.14 gCLData

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

pointer to halo data

Definition at line 218 of file [LatticePatch.h](#).

Referenced by [exchangeGhostCells\(\)](#), [exchangeGhostCells3D\(\)](#), [rotateIntoEigen\(\)](#), and [rotateIntoEigen3D\(\)](#).

5.9.5.15 gCRData

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

pointer to halo data

Definition at line 218 of file [LatticePatch.h](#).

Referenced by [exchangeGhostCells\(\)](#), [exchangeGhostCells3D\(\)](#), [rotateIntoEigen\(\)](#), and [rotateIntoEigen3D\(\)](#).

5.9.5.16 gCTData

```
sunrealtype * LatticePatch::gCTData
```

pointer to halo data

Definition at line 218 of file [LatticePatch.h](#).

Referenced by [exchangeGhostCells3D\(\)](#), and [rotateIntoEigen3D\(\)](#).

5.9.5.17 ghostCellLeft

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

buffer for passing ghost cell data

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

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

5.9.5.18 ghostCellLeftToSend

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

buffer for passing ghost cell data

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

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

5.9.5.19 ghostCellRight

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

buffer for passing ghost cell data

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

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

5.9.5.20 ghostCellRightToSend

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

buffer for passing ghost cell data

Definition at line 184 of file [LatticePatch.h](#).

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

5.9.5.21 ghostCells

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

buffer for passing ghost cell data

Definition at line 184 of file [LatticePatch.h](#).

Referenced by [exchangeGhostCells3D\(\)](#).

5.9.5.22 ghostCellsToSend

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

buffer for passing ghost cell data

Definition at line 184 of file [LatticePatch.h](#).

Referenced by [bufferCreator\(\)](#), and [exchangeGhostCells3D\(\)](#).

5.9.5.23 ID

```
int LatticePatch::ID
```

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

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

5.9.5.24 lgcTox

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

ghost cell translocation lookup table

Definition at line 188 of file [LatticePatch.h](#).

Referenced by [generateTranslocationLookup\(\)](#), [rotateIntoEigen\(\)](#), and [rotateIntoEigen3D\(\)](#).

5.9.5.25 lgcToy

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

ghost cell translocation lookup table

Definition at line 188 of file [LatticePatch.h](#).

Referenced by [generateTranslocationLookup\(\)](#), [rotateIntoEigen\(\)](#), and [rotateIntoEigen3D\(\)](#).

5.9.5.26 lgcToz

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

ghost cell translocation lookup table

Definition at line 188 of file [LatticePatch.h](#).

Referenced by [generateTranslocationLookup\(\)](#), [rotateIntoEigen\(\)](#), and [rotateIntoEigen3D\(\)](#).

5.9.5.27 Llx

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

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

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

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

5.9.5.28 Lly

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

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

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

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

5.9.5.29 LIz

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

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

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

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

5.9.5.30 lx

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

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

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

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

5.9.5.31 ly

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

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

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

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

5.9.5.32 lz

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

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

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

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

5.9.5.33 nx

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

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

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

Referenced by [bufferCreator\(\)](#), [discreteSize\(\)](#), [exchangeGhostCells\(\)](#), [exchangeGhostCells3D\(\)](#), [generateTranslocationLookup\(\)](#), [initializeBuffers\(\)](#), and [LatticePatch\(\)](#).

5.9.5.34 ny

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

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

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

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

5.9.5.35 nz

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

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

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

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

5.9.5.36 `rgcTox`

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

ghost cell translocation lookup table

Definition at line 188 of file [LatticePatch.h](#).

Referenced by [generateTranslocationLookup\(\)](#), [rotateIntoEigen\(\)](#), and [rotateIntoEigen3D\(\)](#).

5.9.5.37 `rgcToy`

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

ghost cell translocation lookup table

Definition at line 188 of file [LatticePatch.h](#).

Referenced by [generateTranslocationLookup\(\)](#), [rotateIntoEigen\(\)](#), and [rotateIntoEigen3D\(\)](#).

5.9.5.38 `rgcToz`

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

ghost cell translocation lookup table

Definition at line 188 of file [LatticePatch.h](#).

Referenced by [generateTranslocationLookup\(\)](#), [rotateIntoEigen\(\)](#), and [rotateIntoEigen3D\(\)](#).

5.9.5.39 `statusFlags`

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

char for checking flags

Definition at line 191 of file [LatticePatch.h](#).

Referenced by [checkFlag\(\)](#), [exchangeGhostCells\(\)](#), [exchangeGhostCells3D\(\)](#), [generateTranslocationLookup\(\)](#), [initializeBuffers\(\)](#), [LatticePatch\(\)](#), and [~LatticePatch\(\)](#).

5.9.5.40 u

```
N_Vector LatticePatch::u
```

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

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

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

5.9.5.41 uAux

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

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

Definition at line 176 of file [LatticePatch.h](#).

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

5.9.5.42 uAuxData

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

pointer to auxiliary data vector

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

Referenced by [rotateIntoEigen\(\)](#), and [rotateIntoEigen3D\(\)](#).

5.9.5.43 uData

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

pointer to field data

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

Referenced by [Simulation::addInitialConditions\(\)](#), [bufferCreator\(\)](#), [exchangeGhostCells\(\)](#), [TimeEvolution::f\(\)](#), [OutputManager::outUState\(\)](#), [rotateIntoEigen\(\)](#), [rotateIntoEigen3D\(\)](#), and [Simulation::setInitialConditions\(\)](#).

5.9.5.44 uTox

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

translocation lookup table

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

Referenced by [derotate\(\)](#), [generateTranslocationLookup\(\)](#), [rotateIntoEigen\(\)](#), and [rotateIntoEigen3D\(\)](#).

5.9.5.45 uToy

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

translocation lookup table

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

Referenced by [derotate\(\)](#), [generateTranslocationLookup\(\)](#), [rotateIntoEigen\(\)](#), and [rotateIntoEigen3D\(\)](#).

5.9.5.46 uToz

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

translocation lookup table

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

Referenced by [derotate\(\)](#), [generateTranslocationLookup\(\)](#), [rotateIntoEigen\(\)](#), and [rotateIntoEigen3D\(\)](#).

5.9.5.47 x0

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

origin of the patch in physical space; x-coordinate

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

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

5.9.5.48 xTou

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

translocation lookup table

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

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

5.9.5.49 y0

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

origin of the patch in physical space; y-coordinate

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

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

5.9.5.50 yTou

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

translocation lookup table

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

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

5.9.5.51 z0

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

origin of the patch in physical space; z-coordinate

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

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

5.9.5.52 zTou

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

translocation lookup table

Definition at line 173 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 data
- void [set_outputStyle](#) (const char _outputStyle)
set the output style
- void [outUState](#) (const int &state, const [Lattice](#) &lattice, const [LatticePatch](#) &latticePatch)
function to write data to disk in specified way
- const string & [getSimCode](#) () const
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
- char [outputStyle](#)
output style; csv or binary

5.10.1 Detailed Description

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

Definition at line 25 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      outputStyle = 'c';  
00012  }
```

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

Here is the call graph for this function:



5.10.3 Member Function Documentation

5.10.3.1 generateOutputFolder()

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

function that creates folder to save simulation data

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

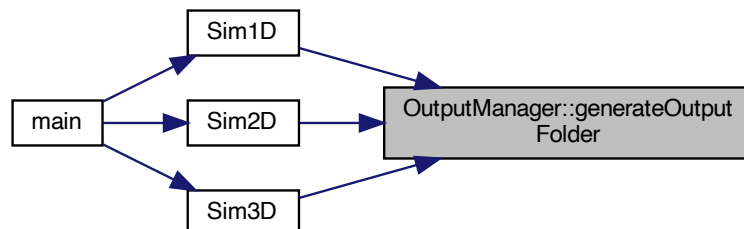
Definition at line 40 of file [Outputters.cpp](#).

```
00040 {
00041     // Do this only once for the first process
00042     int myPrc;
00043     MPI_Comm_rank(MPI_COMM_WORLD, &myPrc);
00044     if (myPrc == 0) {
00045         if (!fs::is_directory(dir))
00046             fs::create_directory(dir);
00047         if (!fs::is_directory(dir + "/SimResults"))
00048             fs::create_directory(dir + "/SimResults");
00049         if (!fs::is_directory(dir + "/SimResults/" + simCode))
00050             fs::create_directory(dir + "/SimResults/" + simCode);
00051     }
00052     // path variable for the output generation
00053     Path = dir + "/SimResults/" + simCode + "/";
00054
00055     // Logging configurations from main.cpp
00056     ifstream fin("main.cpp");
00057     ofstream fout(Path + "config.txt");
00058     string line;
00059     int begin=1000;
00060     for (int i = 1; !fin.eof(); i++) {
00061         getline(fin, line);
00062         if (line.starts_with("    //----- B")) {
00063             begin=i;
00064         }
00065         if (i < begin) {
00066             continue;
00067         }
00068         fout << line << endl;
00069         if (line.starts_with("    //----- E")) {
00070             break;
00071         }
00072     }
00073     return;
00074 }
```

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

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

Here is the caller graph for this function:



5.10.3.2 getSimCode()

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

simCode getter function

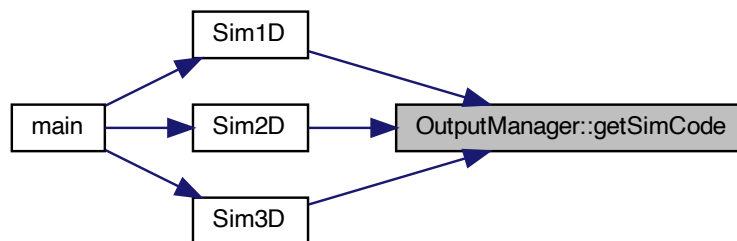
Definition at line 45 of file [Outputters.h](#).

```
00045 { return simCode; }
```

References [simCode](#).

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

Here is the caller graph for this function:



5.10.3.3 outUState()

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

function to write data to disk in specified way

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

Definition at line 85 of file [Outputters.cpp](#).

```
00086                                     {
00087     switch(outputStyle){
00088         case 'c': { // one csv file per process
00089             ofstream ofs;
00090             ofs.open(Path + to_string(state) + "_" + to_string(lattice.my_prc) + ".csv");
00091             // Precision of sunrealtype in significant decimal digits; 15 for IEEE double
00092             ofs << setprecision(numeric_limits<sunrealtype>::digits10);
00093
00094             // Walk through each lattice point
00095             for (int i = 0; i < latticePatch.discreteSize() * 6; i += 6) {
00096                 // Six columns to contain the field data: Ex,Ey,Ez,Bx,By,Bz
00097                 ofs << latticePatch.uData[i + 0] << "," << latticePatch.uData[i + 1] << ","
00098                     << latticePatch.uData[i + 2] << "," << latticePatch.uData[i + 3] << ","
00099                     << latticePatch.uData[i + 4] << "," << latticePatch.uData[i + 5]
```

```

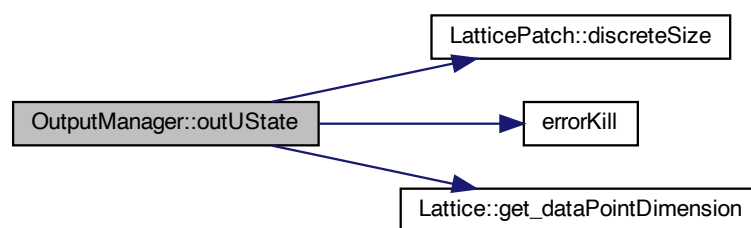
00100         « endl;
00101     }
00102     ofs.close();
00103     break;
00104     }
00105
00106     case 'b': { // a single binary file
00107         // Open the output file
00108         MPI_File fh;
00109         const string filename = Path+to_string(state);
00110         MPI_File_open(lattice.comm, &filename[0], MPI_MODE_WRONLY|MPI_MODE_CREATE,
00111             MPI_INFO_NULL, &fh);
00112         // number of datapoints in the patch with process offset
00113         const int count = latticePatch.discreteSize()*lattice.get_dataPointDimension();
00114         MPI_Offset offset = lattice.my_prc*count*sizeof(MPI_SUNREALTYPE);
00115         // Go to offset in file and write data to it; maximal precision in
00116         // "native" representation
00117         MPI_File_set_view(fh, offset, MPI_SUNREALTYPE, MPI_SUNREALTYPE, "native",
00118             MPI_INFO_NULL);
00119         MPI_File_write_all(fh, latticePatch.uData, count, MPI_SUNREALTYPE, MPI_STATUS_IGNORE);
00120         /*
00121         MPI_Request write_request;
00122         MPI_File_iwrite_all(fh, latticePatch.uData, count, MPI_SUNREALTYPE, &write_request);
00123         MPI_Wait(&write_request, MPI_STATUS_IGNORE);
00124         */
00125         /*
00126         MPI_File_write_at_all(fh, offset, latticePatch.uData, count, MPI_SUNREALTYPE,
00127             MPI_STATUS_IGNORE);
00128         */
00129         break;
00130     }
00131     default: {
00132         errorKill("No valid output style defined.\
00133             Choose between (c): one csv file per process, (b) one binary file");
00134         break;
00135     }
00136     return;
00137 }
00138 }

```

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

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

Here is the call graph for this function:



Here is the caller graph for this function:



5.10.3.4 set_outputStyle()

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

set the output style

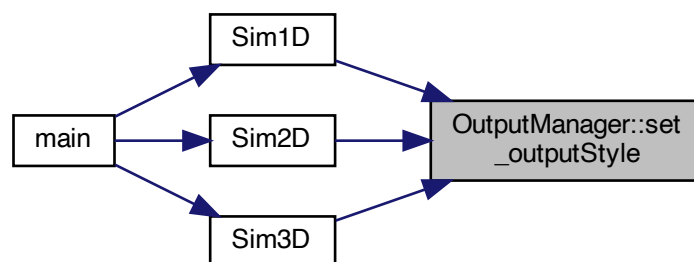
Definition at line 76 of file [Outputters.cpp](#).

```
00076                                     {
00077     outputStyle = _outputStyle;
00078 }
```

References [outputStyle](#).

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

Here is the caller graph for this function:



5.10.3.5 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
00032     // too 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 outputStyle

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

output style; csv or binary

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

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

5.10.4.2 Path

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

variable for the path to the output folder

Definition at line 32 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 30 of file [Outputters.h](#).

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

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

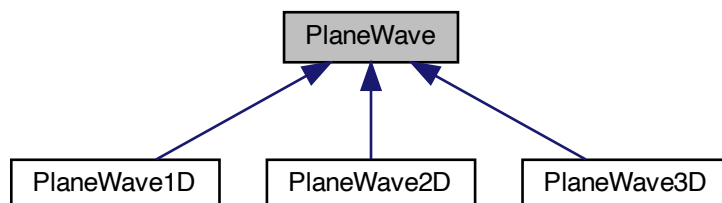
- [src/Outputters.h](#)
- [src/Outputters.cpp](#)

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 23 of file [ICSetters.h](#).

5.11.2 Field Documentation

5.11.2.1 kx

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

wavenumber k_x

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

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

5.11.2.2 ky

sunrealtype PlaneWave::ky [protected]

wavenumber k_y

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

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

5.11.2.3 kz

sunrealtype PlaneWave::kz [protected]

wavenumber k_z

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

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

5.11.2.4 phix

sunrealtype PlaneWave::phix [protected]

phase shift in x-direction, ϕ_x

Definition at line 38 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 40 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 42 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 32 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 34 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 36 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 [20](#) 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 [21](#) of file [SimulationFunctions.h](#).

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

5.12.2.2 [p](#)

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

amplitde & polarization vector

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

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

5.12.2.3 phi

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

phase shift

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

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

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

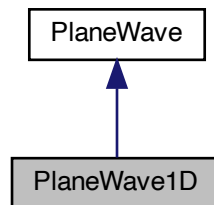
- [src/SimulationFunctions.h](#)

5.13 PlaneWave1D Class Reference

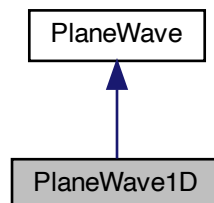
class for plane waves in 1D

```
#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 46 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 28 of file [ICSetters.cpp](#).

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

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

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

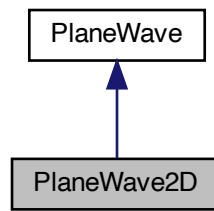
- [src/ICSetters.h](#)
- [src/ICSetters.cpp](#)

5.14 PlaneWave2D Class Reference

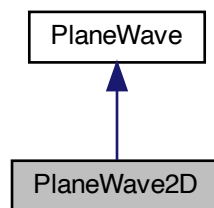
class for plane waves in 2D

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

Inheritance diagram for PlaneWave2D:



Collaboration diagram for PlaneWave2D:



Public Member Functions

- [PlaneWave2D](#) (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 58 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 50 of file [ICSetters.cpp](#).

```
00051 {
00052     kx = k[0]; /** - wavevectors \f$ k_x \f$ */
00053     ky = k[1]; /** - \f$ k_y \f$ */
00054     kz = k[2]; /** - \f$ k_z \f$ normalized to \f$ 1/\lambda \f$ */
00055     // Amplitude bug: lower by factor 9
00056     px = p[0] / 9; /** - amplitude (polarization) in x-direction \f$ p_x \f$ */
00057     py = p[1] / 9; /** - amplitude (polarization) in y-direction \f$ p_y \f$ */
00058     pz = p[2] / 9; /** - amplitude (polarization) in z-direction \f$ p_z \f$ */
00059     phix = phi[0]; /** - phase shift in x-direction \f$ \phi_x \f$ */
00060     phiy = phi[1]; /** - phase shift in y-direction \f$ \phi_y \f$ */
00061     phiz = phi[2]; /** - phase shift in z-direction \f$ \phi_z \f$ */
00062 }
```

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

5.14.3 Member Function Documentation

5.14.3.1 addToSpace()

```
void PlaneWave2D::addToSpace (
    sunrealtype x,
    sunrealtype y,
    sunrealtype z,
    sunrealtype * pTo6Space ) const
```

function for the actual implementation in the lattice

[PlaneWave2D](#) implementation in space

Definition at line 66 of file [ICSetters.cpp](#).

```
00067 {
00068     const sunrealtype wavelength =
00069         sqrt(kx * kx + ky * ky + kz * kz); /* \f$ 1/\lambda \f$ */
00070     const sunrealtype kScalarX = (kx * x + ky * y + kz * z) * 2 *
00071         numbers::pi; /* \f$ 2\pi \ \vec{k} \cdot \vec{x} \f$ */
00072     // Plane wave definition
00073     const array<sunrealtype, 3> E{{ /* E-field vector */
00074         px * cos(kScalarX - phix), /* \f$ E_x \f$ */
00075         py * cos(kScalarX - phiy), /* \f$ E_y \f$ */
00076         pz * cos(kScalarX - phiz)}}; /* \f$ E_z \f$ */
00077     // Put E-field into space
00078     pTo6Space[0] += E[0];
00079     pTo6Space[1] += E[1];
00080     pTo6Space[2] += E[2];
00081     // and B-field
00082     pTo6Space[3] += (ky * E[2] - kz * E[1]) / wavelength;
00083     pTo6Space[4] += (kz * E[0] - kx * E[2]) / wavelength;
00084     pTo6Space[5] += (kx * E[1] - ky * E[0]) / wavelength;
00085 }
```

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

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

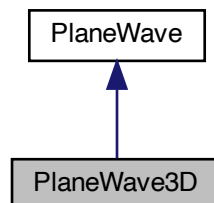
- [src/ICSetters.h](#)
- [src/ICSetters.cpp](#)

5.15 PlaneWave3D Class Reference

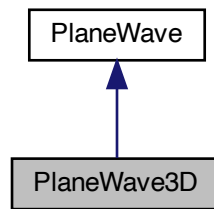
class for plane waves in 3D

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

Inheritance diagram for PlaneWave3D:



Collaboration diagram for PlaneWave3D:



Public Member Functions

- [PlaneWave3D](#) (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 70 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 88 of file [ICSetters.cpp](#).

```
00089
00090     kx = k[0];      /** - wavevectors \f$ k_x \f$ */
00091     ky = k[1];      /** - \f$ k_y \f$ */
00092     kz = k[2];      /** - \f$ k_z \f$ normalized to \f$ 1/\lambda \f$ */
00093     px = p[0];      /** - amplitude (polarization) in x-direction \f$ p_x \f$ */
00094     py = p[1];      /** - amplitude (polarization) in y-direction \f$ p_y \f$ */
00095     pz = p[2];      /** - amplitude (polarization) in z-direction \f$ p_z \f$ */
00096     phix = phi[0];  /** - phase shift in x-direction \f$ \phi_x \f$ */
00097     phiy = phi[1];  /** - phase shift in y-direction \f$ \phi_y \f$ */
00098     phiz = phi[2];  /** - phase shift in z-direction \f$ \phi_z \f$ */
00099 }
```

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

5.15.3 Member Function Documentation

5.15.3.1 addToSpace()

```
void PlaneWave3D::addToSpace (
    sunrealtype x,
    sunrealtype y,
    sunrealtype z,
    sunrealtype * pTo6Space ) const
```

function for the actual implementation in space

[PlaneWave3D](#) implementation in space

Definition at line 103 of file [ICSetters.cpp](#).

```
00104
00105     const sunrealtype wavelength =
00106         sqrt(kx * kx + ky * ky + kz * kz); /* \f$ 1/\lambda \f$ */
00107     const sunrealtype kScalarX = (kx * x + ky * y + kz * z) * 2 *
00108         numbers::pi; /* \f$ 2\pi \cdot \vec{k} \cdot \vec{x} \f$ */
00109     // Plane wave definition
00110     const array<sunrealtype, 3> E{ /* E-field vector \f$ \vec{E} \f$ */
00111         px * cos(kScalarX - phix), /* \f$ E_x \f$ */
00112         py * cos(kScalarX - phiy), /* \f$ E_y \f$ */
00113         pz * cos(kScalarX - phiz) }; /* \f$ E_z \f$ */
00114     // Put E-field into space
00115     pTo6Space[0] += E[0];
00116     pTo6Space[1] += E[1];
00117     pTo6Space[2] += E[2];
00118     // and B-field
00119     pTo6Space[3] += (ky * E[2] - kz * E[1]) / wavelength;
00120     pTo6Space[4] += (kz * E[0] - kx * E[2]) / wavelength;
00121     pTo6Space[5] += (kx * E[1] - ky * E[0]) / wavelength;
00122 }
```

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

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

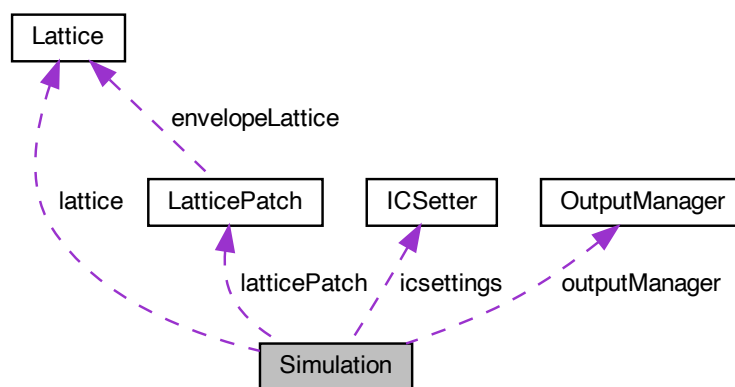
- [src/ICSetters.h](#)
- [src/ICSetters.cpp](#)

5.16 Simulation Class Reference

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

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

Collaboration diagram for Simulation:



Public Member Functions

- [Simulation](#) (const int nx, const int ny, const int nz, const int StencilOrder, const bool periodicity)
constructor function for the creation of the cartesian communicator
- [~Simulation](#) ()
destructor function freeing C-Code 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 sunrealtypes &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 * [cvmem](#)
Pointer to CVode memory object – public to avoid cross library errors.

Private Attributes

- [Lattice lattice](#)
Lattice object.
- [LatticePatch latticePatch](#)
LatticePatch object.
- sunrealtypes 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 37 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

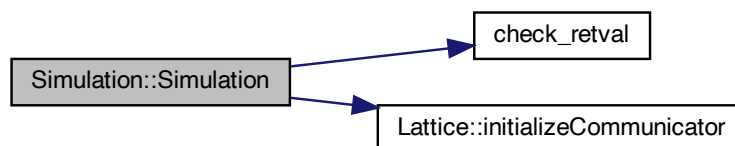
Along with the simulation object, create the cartesian communicator and SUNContext object

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(cvode_mem); // TODO write this function as in cvodes
00038         // cvAdvDiff_bnd.c SUNDIALS_MARK_FUNCTION_END(lattice.profobj);
00039         CcodeFree(&cvode_mem);
00040         SUNContext_Free(&lattice.sunctx);
00041     }
00042 }
```

References [cvode_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.

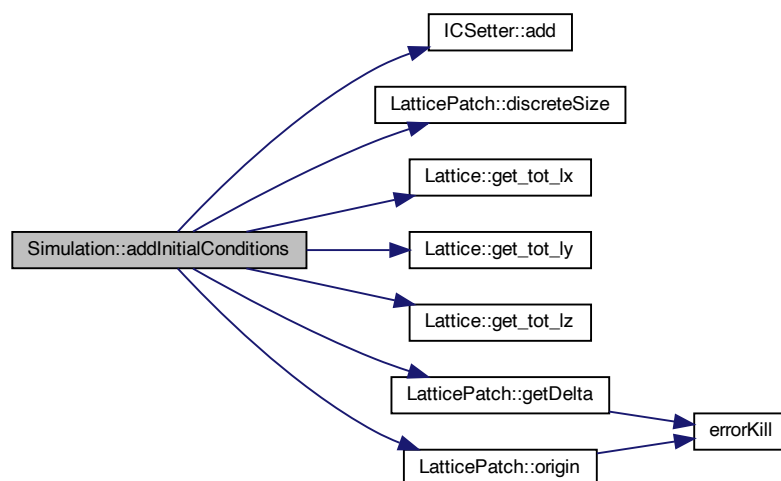
Definition at line 185 of file [SimulationClass.cpp](#).

```
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 }
```

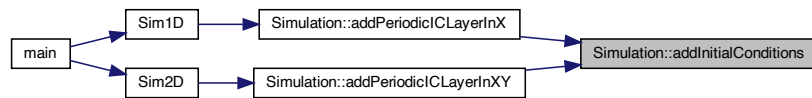
References [ICSetter::add\(\)](#), [LatticePatch::discreteSize\(\)](#), [Lattice::get_tot_lx\(\)](#), [Lattice::get_tot_ly\(\)](#), [Lattice::get_tot_lz\(\)](#), [LatticePatch::getDelta\(\)](#), [icsettings](#), [lattice](#), [latticePatch](#), [LatticePatch::origin\(\)](#), and [LatticePatch::uData](#).

Referenced by [addPeriodicCLayerInX\(\)](#), and [addPeriodicCLayerInXY\(\)](#).

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.

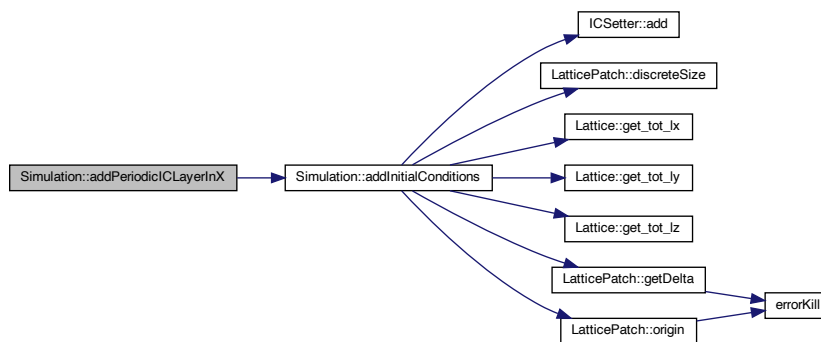
Definition at line 209 of file [SimulationClass.cpp](#).

```
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.

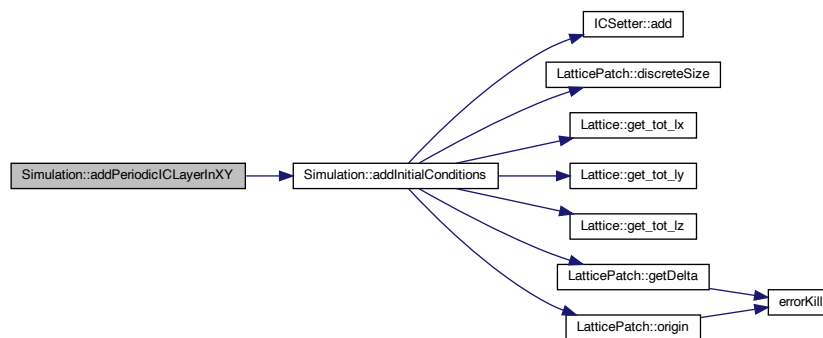
Definition at line 216 of file [SimulationClass.cpp](#).

```
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 }
```

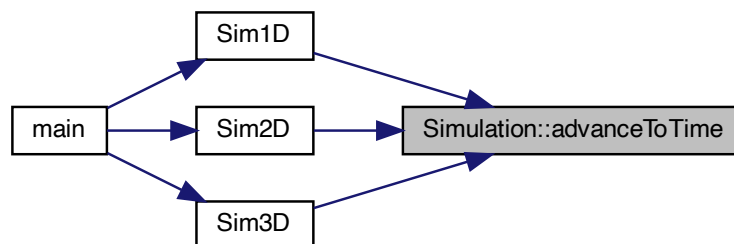
References [checkFlag\(\)](#), [cvode_mem](#), [latticePatch](#), [SimulationStarted](#), [t](#), and [LatticePatch::u](#).

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 }
```

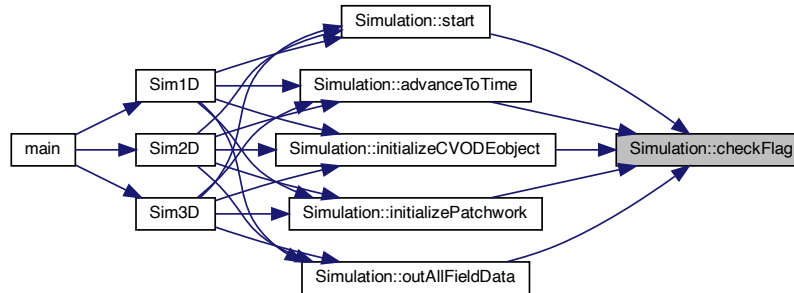
References [CvodeObjectSetUp](#), [errorKill\(\)](#), [LatticeDiscreteSetUp](#), [LatticePatchworkSetUp](#), [LatticePhysicalSetUp](#), [SimulationStarted](#), and [statusFlags](#).

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 }
```

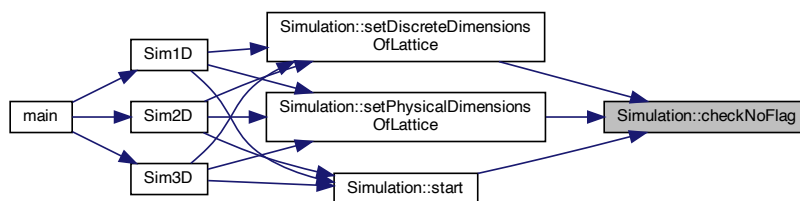
References [CvodeObjectSetUp](#), [errorKill\(\)](#), [LatticeDiscreteSetUp](#), [LatticePatchworkSetUp](#), [LatticePhysicalSetUp](#), [SimulationStarted](#), and [statusFlags](#).

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 61 of file [SimulationClass.h](#).

```
00061 { 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);
```

```

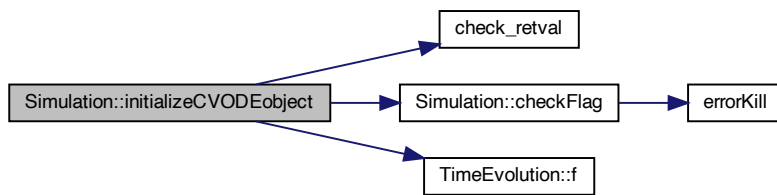
00077
00078 // CCode settings return value
00079 int retval = 0;
00080
00081 // Set the profiler
00082 retval = SUNContext_GetProfiler(lattice.sunctx, &lattice.profoobj);
00083 if (check_retval(&retval, "SUNContext_GetProfiler", 1, lattice.my_prc))
00084     MPI_Abort(lattice.comm, 1);
00085 // if (flag != CV_SUCCESS) { printf("SUNContext_GetProfiler failed,
00086 // flag=%d.\n", flag);
00087 //     MPI_Abort(lattice.comm, 1); }
00088
00089 // SUNDIALS_MARK_FUNCTION_BEGIN(profoobj);
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 }

```

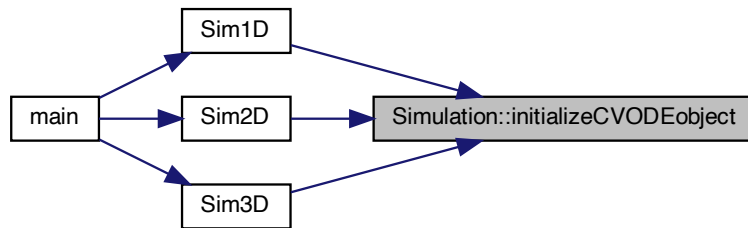
References [check_retval\(\)](#), [checkFlag\(\)](#), [Lattice::comm](#), [ccode_mem](#), [CcodeObjectSetUp](#), [TimeEvolution::f\(\)](#), [lattice](#), [latticePatch](#), [Lattice::my_prc](#), [Lattice::profoobj](#), [SimulationStarted](#), [statusFlags](#), [Lattice::sunctx](#), and [LatticePatch::u](#).

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](#).

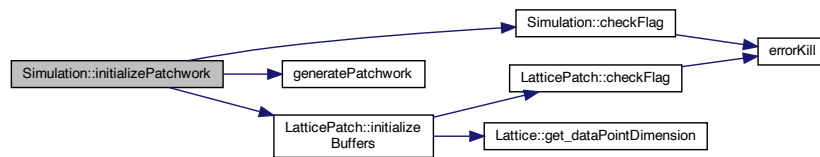
```

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     }
  
```

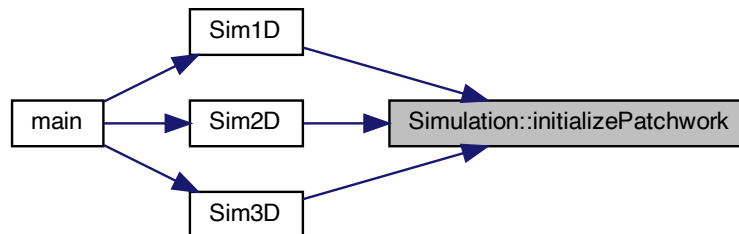
References [checkFlag\(\)](#), [generatePatchwork\(\)](#), [LatticePatch::initializeBuffers\(\)](#), [lattice](#), [LatticeDiscreteSetUp](#), [latticePatch](#), [LatticePatchworkSetUp](#), [LatticePhysicalSetUp](#), and [statusFlags](#).

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.

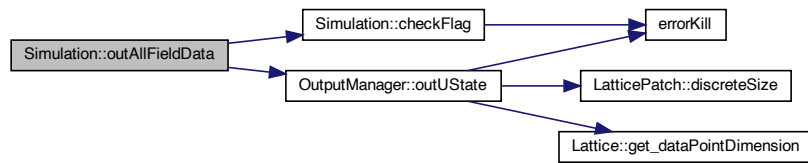
Definition at line 241 of file [SimulationClass.cpp](#).

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

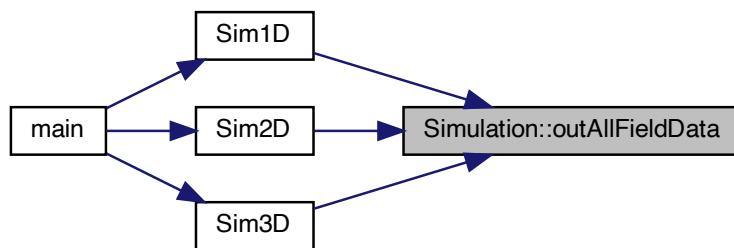
References [checkFlag\(\)](#), [lattice](#), [latticePatch](#), [outputManager](#), [OutputManager::outUState\(\)](#), and [SimulationStarted](#).

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

Here is the call graph for this function:



Here is the caller graph for this function:



5.16.3.11 setDiscreteDimensionsOfLattice()

```

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

function to set discrete dimensions of the lattice

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

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

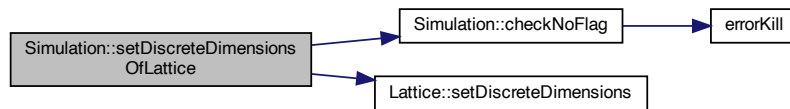
```

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

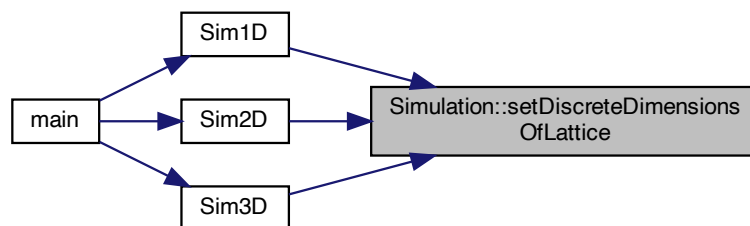
References [checkNoFlag\(\)](#), [lattice](#), [LatticeDiscreteSetUp](#), [LatticePatchworkSetUp](#), [Lattice::setDiscreteDimensions\(\)](#), and [statusFlags](#).

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](#).

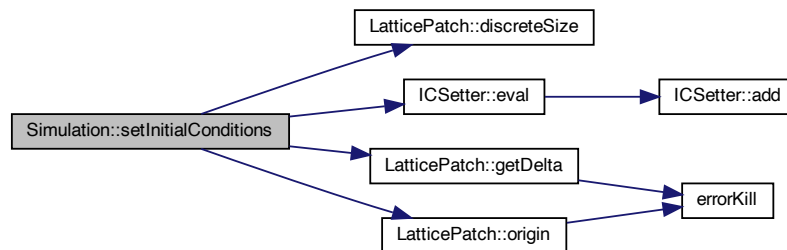
```

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 }
  
```

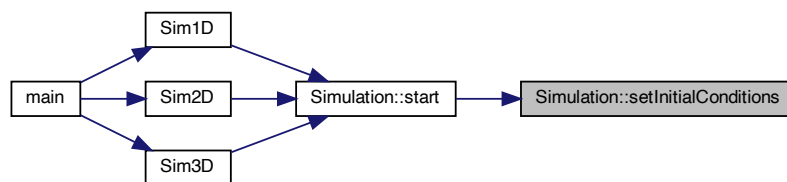
References [LatticePatch::discreteSize\(\)](#), [ICSetter::eval\(\)](#), [LatticePatch::getDelta\(\)](#), [icsettings](#), [latticePatch](#), [LatticePatch::origin\(\)](#), and [LatticePatch::uData](#).

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](#).

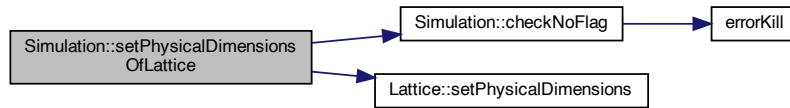
```

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

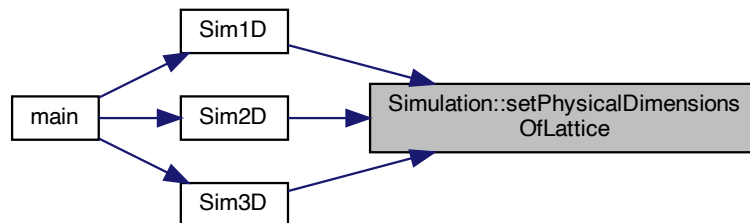
References [checkNoFlag\(\)](#), [lattice](#), [LatticePatchworkSetUp](#), [LatticePhysicalSetUp](#), [Lattice::setPhysicalDimensions\(\)](#), and [statusFlags](#).

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](#).

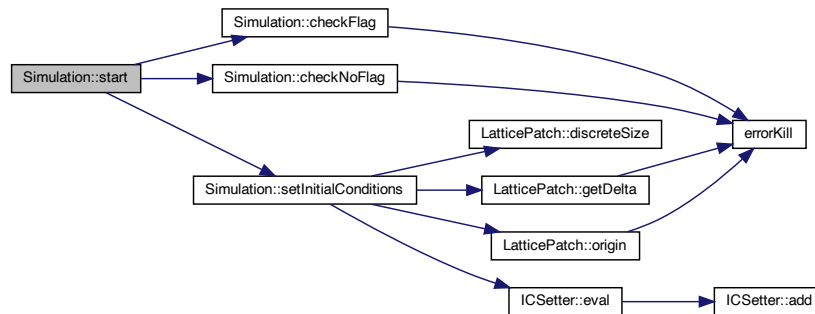
```

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

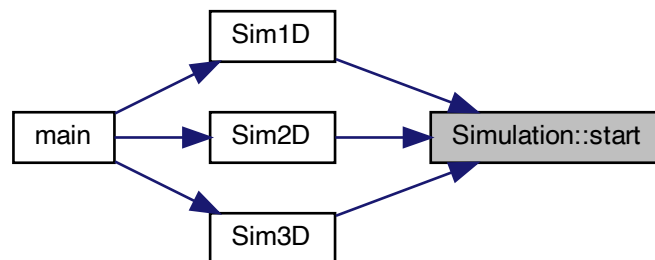
References [checkFlag\(\)](#), [checkNoFlag\(\)](#), [CvodeObjectSetUp](#), [LatticeDiscreteSetUp](#), [LatticePatchworkSetUp](#), [LatticePhysicalSetUp](#), [setInitialConditions\(\)](#), [SimulationStarted](#), and [statusFlags](#).

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 cvode_mem

```
void* Simulation::cvode_mem
```

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

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

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

5.16.4.2 icsettings

`ICSetter` `Simulation::icsettings`

IC Setter object.

Definition at line 50 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 40 of file [SimulationClass.h](#).

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

5.16.4.4 latticePatch

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

`LatticePatch` object.

Definition at line 42 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 52 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 46 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 44 of file [SimulationClass.h](#).

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

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

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

5.17 TimeEvolution Class Reference

monostate [TimeEvolution](#) Class to propagate the field data in time in a given order of the HE weak-field expansion

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

Static Public Member Functions

- static int [f](#)(sunrealtype t, N_Vector u, N_Vector udot, void *data_loc)
CVODE right hand side function (CVRhsFn) to provide IVP of the ODE.

Static Public Attributes

- static int * [c](#) = nullptr
choice which processes of the weak field expansion are included
- static void(* [TimeEvolver](#))([LatticePatch](#) *, N_Vector, N_Vector, int *) = [nonlinear1DProp](#)
Pointer to functions for differentiation and time evolution.

5.17.1 Detailed Description

monostate [TimeEvolution](#) Class to propagate the field data in time in a given order of the HE weak-field expansion

Definition at line 15 of file [TimeEvolutionFunctions.h](#).

5.17.2 Member Function Documentation

5.17.2.1 f()

```
int TimeEvolution::f (
    sunrealtype t,
    N_Vector u,
    N_Vector udot,
    void * data_loc ) [static]
```

CVODE right hand side function (CVRhsFn) to provide IVP of the ODE.

Cvode right-hand-side function (CVRhsFn)

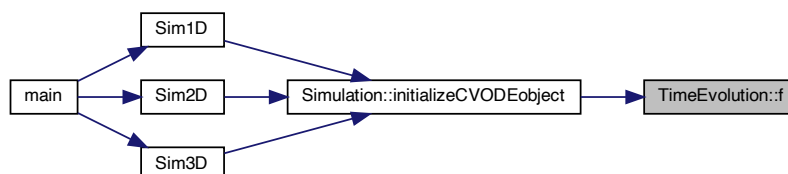
Definition at line 13 of file [TimeEvolutionFunctions.cpp](#).

```
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 18 of file [TimeEvolutionFunctions.h](#).

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

5.17.3.2 TimeEvolver

```
void(* TimeEvolution::TimeEvolver)(LatticePatch *, N_Vector, N_Vector, int *) = nonlinear1DProp  
[static]
```

Pointer to functions for differentiation and time evolution.

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

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

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

- [src/TimeEvolutionFunctions.h](#)
- [src/SimulationFunctions.cpp](#)
- [src/TimeEvolutionFunctions.cpp](#)

Chapter 6

File Documentation

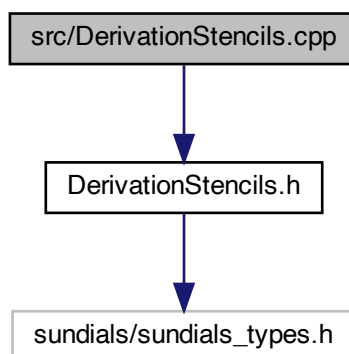
6.1 README.md File Reference

6.2 src/DerivationStencils.cpp File Reference

Empty. All definitions in the header.

```
#include "DerivationStencils.h"
```

Include dependency graph for DerivationStencils.cpp:



6.2.1 Detailed Description

Empty. All definitions in the header.

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

6.3 DerivationStencils.cpp

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

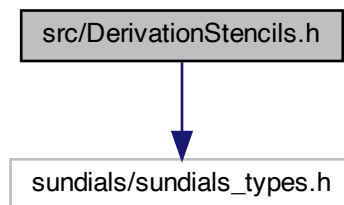
```
00001 ///////////////////////////////////////////////////////////////////
00002 /// @file DerivationStencils.cpp
00003 /// @brief Empty. All definitions in the header.
00004 ///////////////////////////////////////////////////////////////////
00005 #include "DerivationStencils.h"
```

6.4 src/DerivationStencils.h File Reference

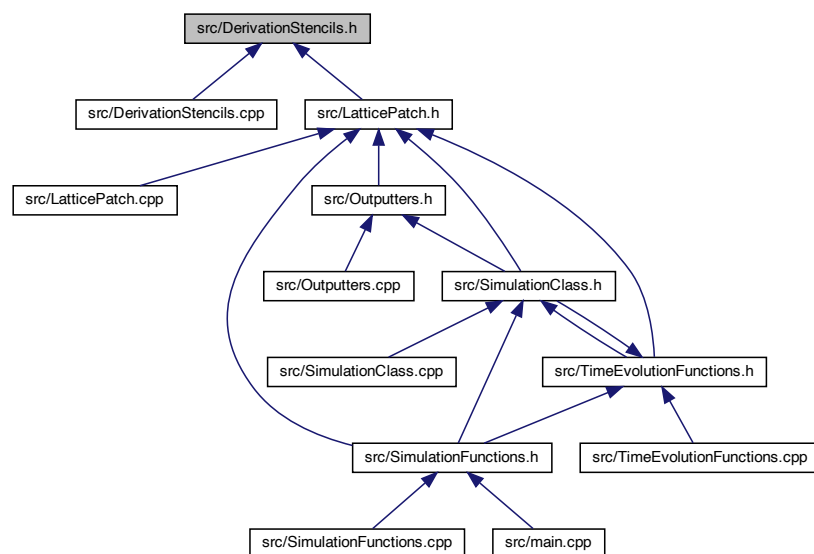
Definition of derivation stencils from order 1 to 13.

```
#include <sundials/sundials_types.h>
```

Include dependency graph for DerivationStencils.h:



This graph shows which files directly or indirectly include this file:



Functions

- sunrealtype [s1f](#) (sunrealtype *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]

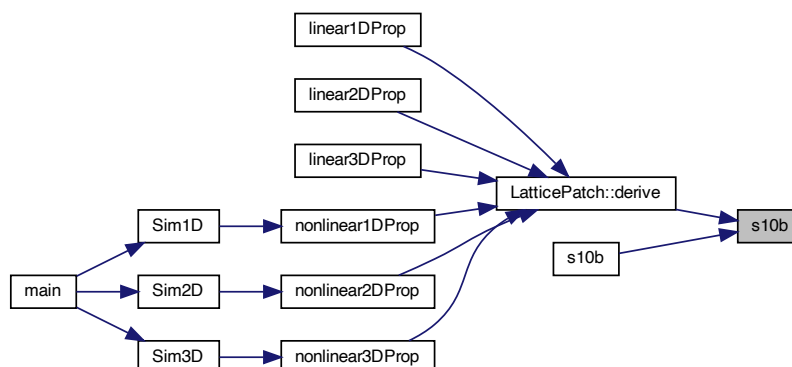
```
sunrealtype s10b (
    const sunrealtype * udata,
    int nx ) [inline]
```

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

```
00144 {
00145     return 1.0 / 840.0 * udata[-4 * nx] - 1.0 / 63.0 * udata[-3 * nx] +
00146           3.0 / 28.0 * udata[-2 * nx] - 4.0 / 7.0 * udata[-1 * nx] -
00147           11.0 / 30.0 * udata[0] + 6.0 / 5.0 * udata[1 * nx] -
00148           1.0 / 2.0 * udata[2 * nx] + 4.0 / 21.0 * udata[3 * nx] -
00149           3.0 / 56.0 * udata[4 * nx] + 1.0 / 105.0 * udata[5 * nx] -
00150           1.0 / 1260.0 * udata[6 * nx];
00151 }
```

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 244 of file [DerivationStencils.h](#).

```
00244 { return s10b(udata, 6); }
```

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

Here is the call graph for this function:



6.4.2.3 s10c() [1/2]

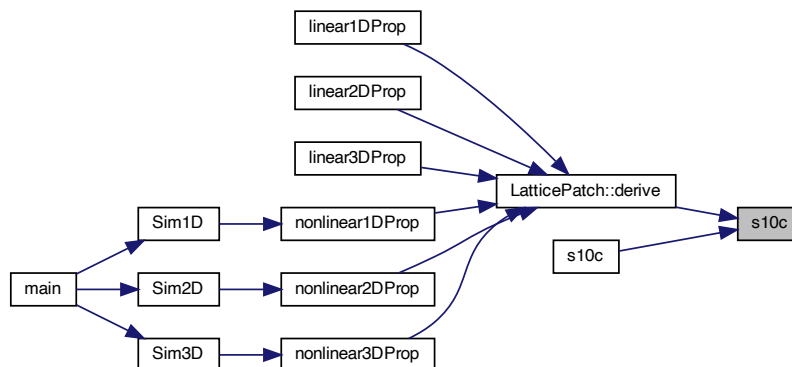
```
sunrealtype s10c (
    const sunrealtype * udata,
    int nx ) [inline]
```

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

```
00137 {
00138     return -1.0 / 1260.0 * udata[-5 * nx] + 5.0 / 504.0 * udata[-4 * nx] -
00139            5.0 / 84.0 * udata[-3 * nx] + 5.0 / 21.0 * udata[-2 * nx] -
00140            5.0 / 6.0 * udata[-1 * nx] + 0 + 5.0 / 6.0 * udata[1 * nx] -
00141            5.0 / 21.0 * udata[2 * nx] + 5.0 / 84.0 * udata[3 * nx] -
00142            5.0 / 504.0 * udata[4 * nx] + 1.0 / 1260.0 * udata[5 * nx];
00143 }
```

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 243 of file [DerivationStencils.h](#).

```
00243 { return s10c(udata, 6); }
```

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

Here is the call graph for this function:



6.4.2.5 s10f() [1/2]

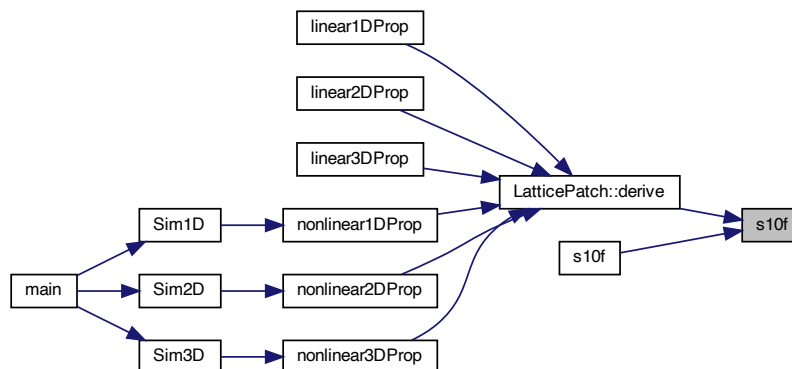
```
sunrealtype s10f (
    const sunrealtype * udata,
    int nx ) [inline]
```

Definition at line 129 of file [DerivationStencils.h](#).

```
00129 {
00130     return 1.0 / 1260.0 * udata[-6 * nx] - 1.0 / 105.0 * udata[-5 * nx] +
00131            3.0 / 56.0 * udata[-4 * nx] - 4.0 / 21.0 * udata[-3 * nx] +
00132            1.0 / 2.0 * udata[-2 * nx] - 6.0 / 5.0 * udata[-1 * nx] +
00133            11.0 / 30.0 * udata[0] + 4.0 / 7.0 * udata[1 * nx] -
00134            3.0 / 28.0 * udata[2 * nx] + 1.0 / 63.0 * udata[3 * nx] -
00135            1.0 / 840.0 * udata[4 * nx];
00136 }
```

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 242 of file [DerivationStencils.h](#).

```
00242 { 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 246 of file [DerivationStencils.h](#).

```
00246 { return s11b(udata, 6); }
```

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

Here is the call graph for this function:



6.4.2.8 s11b() [2/2]

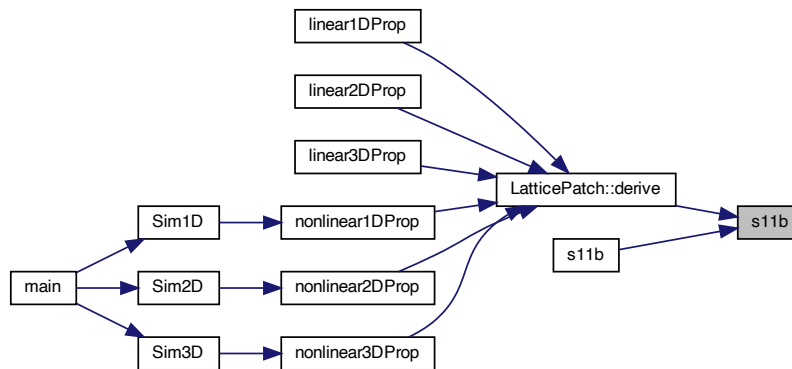
```
sunrealtype s11b (
    sunrealtype * udata,
    int nx ) [inline]
```

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

```
00160 {
00161     return -1.0 / 2310.0 * udata[-5 * nx] + 1.0 / 168.0 * udata[-4 * nx] -
00162            5.0 / 126.0 * udata[-3 * nx] + 5.0 / 28.0 * udata[-2 * nx] -
00163            5.0 / 7.0 * udata[-1 * nx] - 1.0 / 6.0 * udata[0] + udata[1 * nx] -
00164            5.0 / 14.0 * udata[2 * nx] + 5.0 / 42.0 * udata[3 * nx] -
00165            5.0 / 168.0 * udata[4 * nx] + 1.0 / 210.0 * udata[5 * nx] -
00166            1.0 / 2772.0 * udata[6 * nx];
00167 }
```

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

Here is the caller graph for this function:



6.4.2.9 s11f() [1/2]

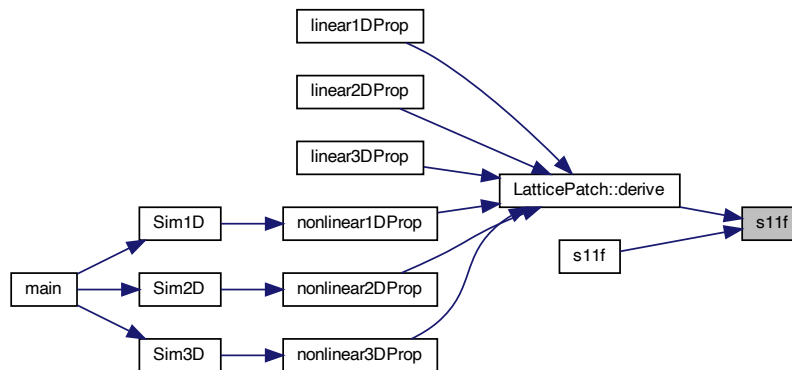
```
sunrealtype s11f (
    const sunrealtype * udata,
    int nx ) [inline]
```

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

```
00152 {
00153     return 1.0 / 2772.0 * udata[-6 * nx] - 1.0 / 210.0 * udata[-5 * nx] +
00154            5.0 / 168.0 * udata[-4 * nx] - 5.0 / 42.0 * udata[-3 * nx] +
00155            5.0 / 14.0 * udata[-2 * nx] - 1.0 / 1.0 * udata[-1 * nx] +
00156            1.0 / 6.0 * udata[0] + 5.0 / 7.0 * udata[1 * nx] -
00157            5.0 / 28.0 * udata[2 * nx] + 5.0 / 126.0 * udata[3 * nx] -
00158            1.0 / 168.0 * udata[4 * nx] + 1.0 / 2310.0 * udata[5 * nx];
00159 }
```

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 245 of file [DerivationStencils.h](#).

```

00245 { 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 185 of file [DerivationStencils.h](#).

```

00185

```

```

{

```

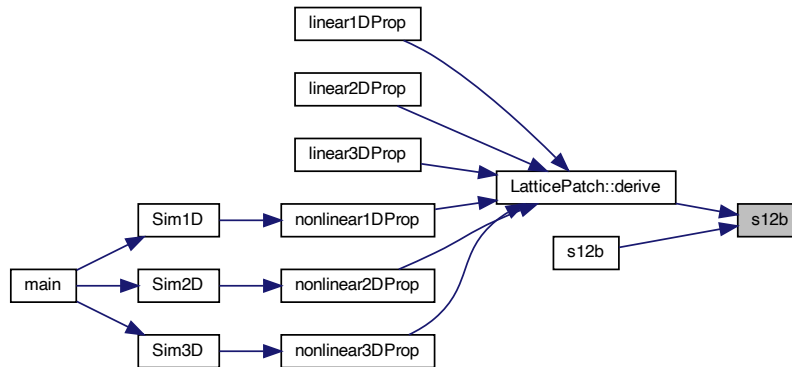
```

00186     return -1.0 / 3960.0 * udata[-5 * nx] + 1.0 / 264.0 * udata[-4 * nx] -
00187           1.0 / 36.0 * udata[-3 * nx] + 5.0 / 36.0 * udata[-2 * nx] -
00188           5.0 / 8.0 * udata[-1 * nx] - 13.0 / 42.0 * udata[0] +
00189           7.0 / 6.0 * udata[1 * nx] - 1.0 / 2.0 * udata[2 * nx] +
00190           5.0 / 24.0 * udata[3 * nx] - 5.0 / 72.0 * udata[4 * nx] +
00191           1.0 / 60.0 * udata[5 * nx] - 1.0 / 396.0 * udata[6 * nx] +
00192           1.0 / 5544.0 * udata[7 * nx];
00193 }

```

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 249 of file [DerivationStencils.h](#).

```

00249 { return s12b(udata, 6); }

```

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

Here is the call graph for this function:



6.4.2.13 s12c() [1/2]

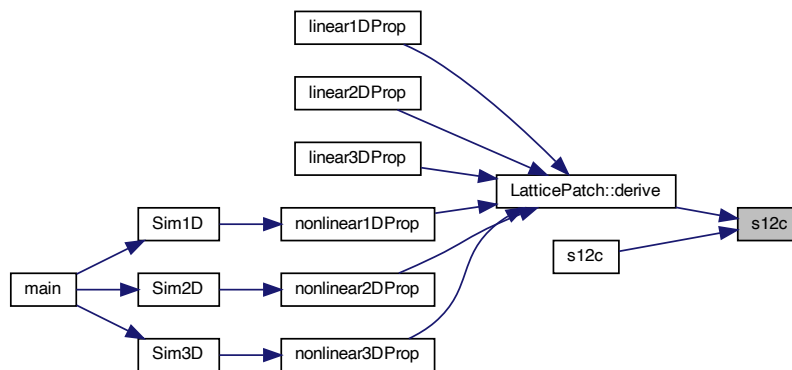
```
sunrealtype s12c (
    const sunrealtype * udata,
    int nx ) [inline]
```

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

```
00177 {
00178     return 1.0 / 5544.0 * udata[-6 * nx] - 1.0 / 385.0 * udata[-5 * nx] +
00179            1.0 / 56.0 * udata[-4 * nx] - 5.0 / 63.0 * udata[-3 * nx] +
00180            15.0 / 56.0 * udata[-2 * nx] - 6.0 / 7.0 * udata[-1 * nx] + 0 +
00181            6.0 / 7.0 * udata[1 * nx] - 15.0 / 56.0 * udata[2 * nx] +
00182            5.0 / 63.0 * udata[3 * nx] - 1.0 / 56.0 * udata[4 * nx] +
00183            1.0 / 385.0 * udata[5 * nx] - 1.0 / 5544.0 * udata[6 * nx];
00184 }
```

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 248 of file [DerivationStencils.h](#).

```
00248 { return s12c(udata, 6); }
```

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

Here is the call graph for this function:



6.4.2.15 s12f() [1/2]

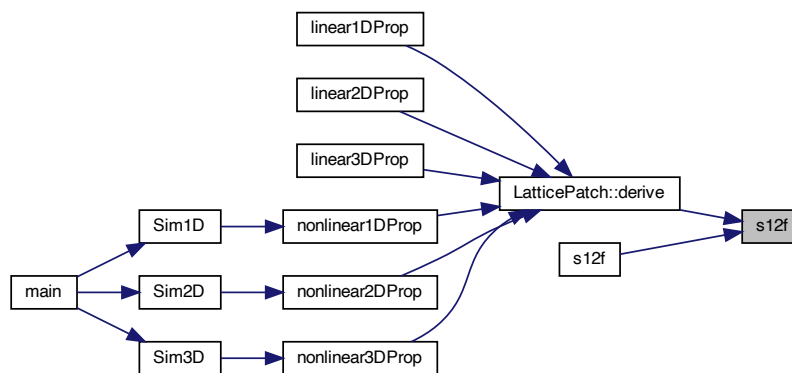
```
sunrealtype s12f (
    const sunrealtype * udata,
    int nx ) [inline]
```

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

```
00168 {
00169     return -1.0 / 5544.0 * udata[-7 * nx] + 1.0 / 396.0 * udata[-6 * nx] -
00170            1.0 / 60.0 * udata[-5 * nx] + 5.0 / 72.0 * udata[-4 * nx] -
00171            5.0 / 24.0 * udata[-3 * nx] + 1.0 / 2.0 * udata[-2 * nx] -
00172            7.0 / 6.0 * udata[-1 * nx] + 13.0 / 42.0 * udata[0] +
00173            5.0 / 8.0 * udata[1 * nx] - 5.0 / 36.0 * udata[2 * nx] +
00174            1.0 / 36.0 * udata[3 * nx] - 1.0 / 264.0 * udata[4 * nx] +
00175            1.0 / 3960.0 * udata[5 * nx];
00176 }
```

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 247 of file [DerivationStencils.h](#).

```
00247 { 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 251 of file [DerivationStencils.h](#).

```
00251 { return s13b(udata, 6); }
```

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

Here is the call graph for this function:

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

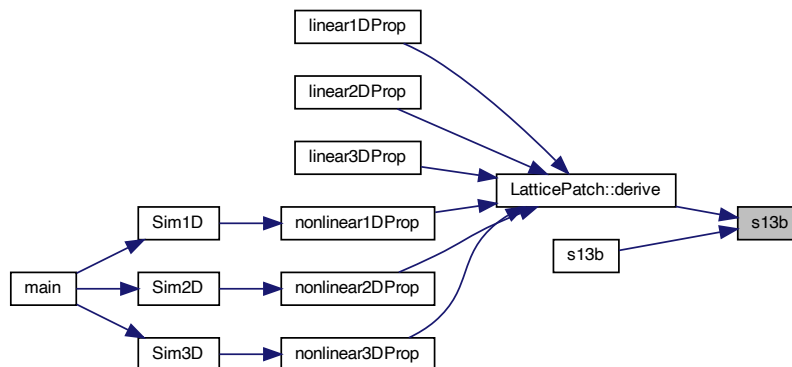
```
sunrealtype s13b (
    sunrealtype * udata,
    int nx ) [inline]
```

Definition at line 206 of file [DerivationStencils.h](#).

```
00206 {
00207     return 1.0 / 10296.0 * udata[-6 * nx] - 1.0 / 660.0 * udata[-5 * nx] +
00208           1.0 / 88.0 * udata[-4 * nx] - 1.0 / 18.0 * udata[-3 * nx] +
00209           5.0 / 24.0 * udata[-2 * nx] - 3.0 / 4.0 * udata[-1 * nx] -
00210           1.0 / 7.0 * udata[0] + udata[1 * nx] - 3.0 / 8.0 * udata[2 * nx] +
00211           5.0 / 36.0 * udata[3 * nx] - 1.0 / 24.0 * udata[4 * nx] +
00212           1.0 / 110.0 * udata[5 * nx] - 1.0 / 792.0 * udata[6 * nx] +
00213           1.0 / 12012.0 * udata[7 * nx];
00214 }
```

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

Here is the caller graph for this function:



6.4.2.19 s13f() [1/2]

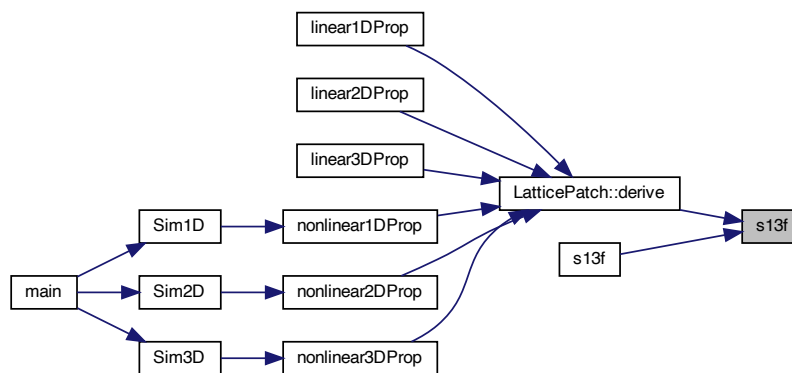
```
sunrealtype s13f (
    const sunrealtype * udata,
    int nx ) [inline]
```

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 250 of file [DerivationStencils.h](#).

```
00250 { 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 221 of file [DerivationStencils.h](#).

```
00221 { return s1b(udata, 6); }
```

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

Here is the call graph for this function:

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

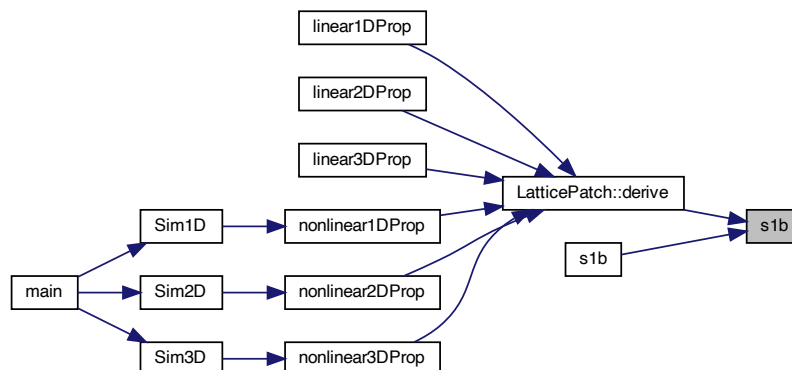
```
sunrealtype s1b (
    sunrealtype * udata,
    int nx ) [inline]
```

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

```
00019 {
00020     return -1.0 / 1.0 * udata[0] + udata[1 * nx];
00021 }
```

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 220 of file [DerivationStencils.h](#).

```
00220 { return s1f(udata, 6); }
```

References [s1f\(\)](#).

Here is the call graph for this function:

**6.4.2.24 s1f()** [2/2]

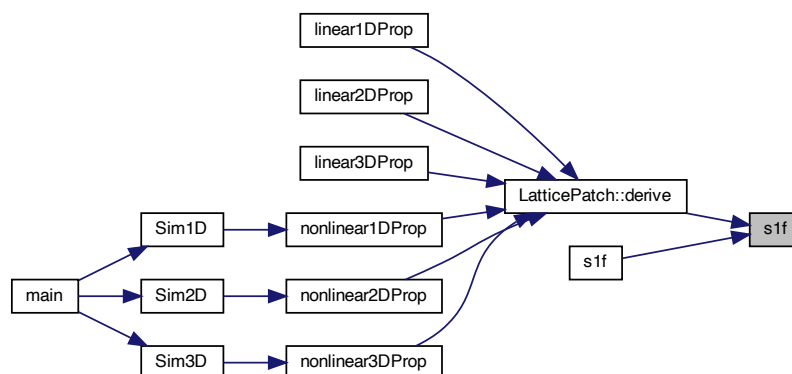
```
sunrealtype s1f (
    sunrealtype * udata,
    int nx ) [inline]
```

Definition at line 15 of file [DerivationStencils.h](#).

```
00015 {
00016     return -1.0 / 1.0 * udata[-1 * nx] + udata[0];
00017 }
```

Referenced by [LatticePatch::derive\(\)](#), and [s1f\(\)](#).

Here is the caller graph for this function:



6.4.2.25 s2b() [1/2]

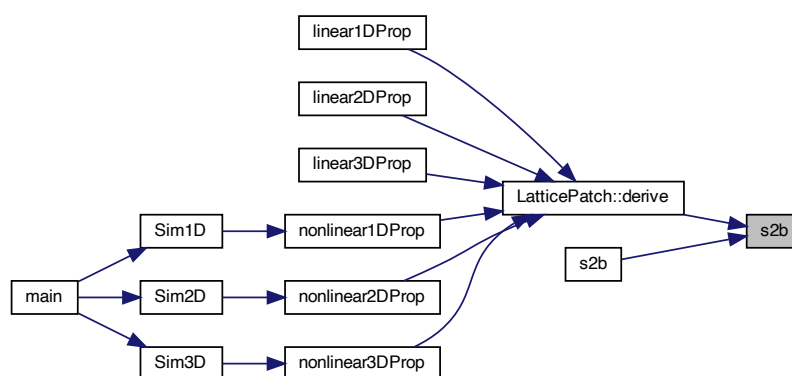
```
sunrealtype s2b (
    const sunrealtype * udata,
    int nx ) [inline]
```

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

```
00030 {
00031     return -3.0 / 2.0 * udata[0] + 2.0 / 1.0 * udata[1 * nx] -
00032           1.0 / 2.0 * udata[2 * nx];
00033 }
```

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 224 of file [DerivationStencils.h](#).

```
00224 { 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 27 of file [DerivationStencils.h](#).

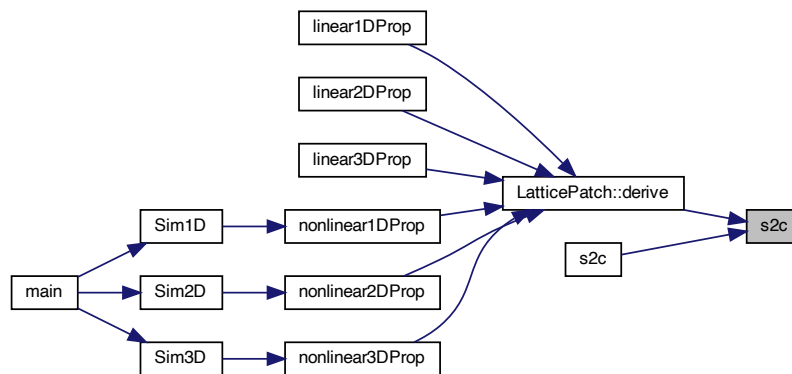
```

00027 {
00028     return -1.0 / 2.0 * udata[-1 * nx] + 0 + 1.0 / 2.0 * udata[1 * nx];
00029 }

```

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 223 of file [DerivationStencils.h](#).

```

00223 { return s2c(udata, 6); }

```

References [s2c\(\)](#).

Here is the call graph for this function:



6.4.2.29 s2f() [1/2]

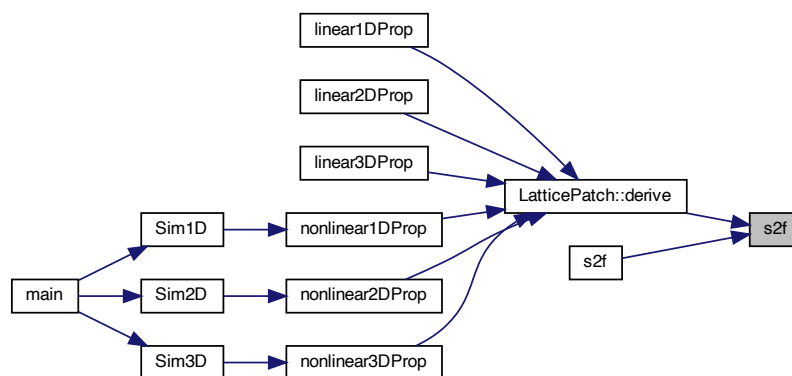
```
sunrealtype s2f (
    const sunrealtype * udata,
    int nx ) [inline]
```

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

```
00023 {
00024     return 1.0 / 2.0 * udata[-2 * nx] - 2.0 / 1.0 * udata[-1 * nx] +
00025           3.0 / 2.0 * udata[0];
00026 }
```

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 222 of file [DerivationStencils.h](#).

```
00222 { 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 226 of file [DerivationStencils.h](#).

```
00226 { return s3b(udata, 6); }
```

References [s3b\(\)](#).

Here is the call graph for this function:



6.4.2.32 s3b() [2/2]

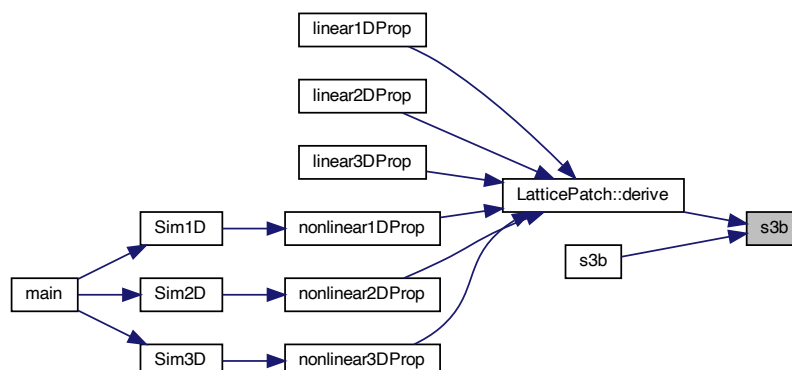
```
sunrealtype s3b (
    sunrealtype * udata,
    int nx ) [inline]
```

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

```
00038 {
00039     return -1.0 / 3.0 * udata[-1 * nx] - 1.0 / 2.0 * udata[0] + udata[1 * nx] -
00040           1.0 / 6.0 * udata[2 * nx];
00041 }
```

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 34 of file [DerivationStencils.h](#).

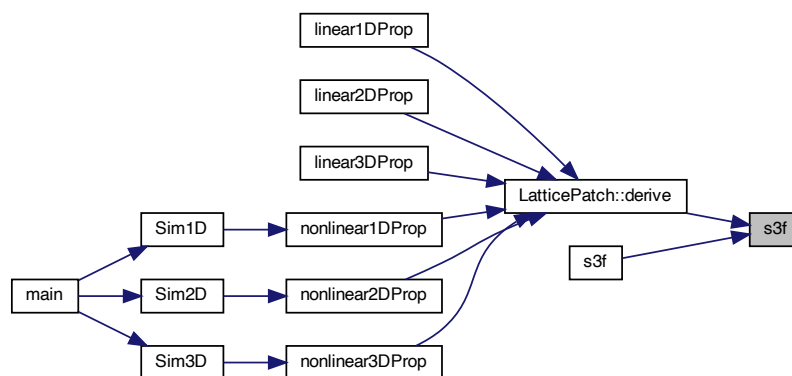
```

00034 {
00035     return 1.0 / 6.0 * udata[-2 * nx] - 1.0 / 1.0 * udata[-1 * nx] +
00036           1.0 / 2.0 * udata[0] + 1.0 / 3.0 * udata[1 * nx];
00037 }

```

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 225 of file [DerivationStencils.h](#).

```

00225 { return s3f(udata, 6); }

```

References [s3f\(\)](#).

Here is the call graph for this function:



6.4.2.35 `s4b()` [1/2]

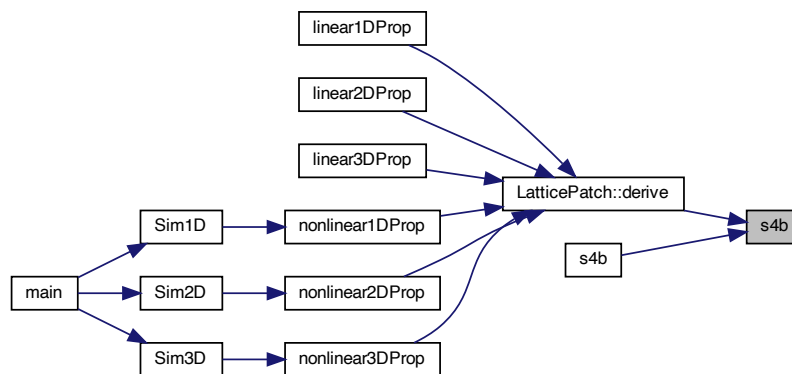
```
sunrealtype s4b (
    const sunrealtype * udata,
    int nx ) [inline]
```

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

```
00051 {
00052     return -1.0 / 4.0 * udata[-1 * nx] - 5.0 / 6.0 * udata[0] +
00053           3.0 / 2.0 * udata[1 * nx] - 1.0 / 2.0 * udata[2 * nx] +
00054           1.0 / 12.0 * udata[3 * nx];
00055 }
```

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 229 of file [DerivationStencils.h](#).

```
00229 { return s4b(udata, 6); }
```

References [s4b\(\)](#).

Here is the call graph for this function:



6.4.2.37 s4c() [1/2]

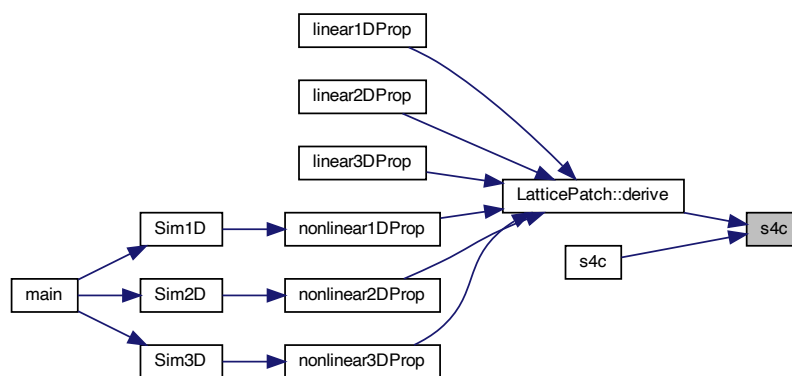
```
sunrealtype s4c (
    const sunrealtype * udata,
    int nx ) [inline]
```

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

```
00047 {
00048     return 1.0 / 12.0 * udata[-2 * nx] - 2.0 / 3.0 * udata[-1 * nx] + 0 +
00049           2.0 / 3.0 * udata[1 * nx] - 1.0 / 12.0 * udata[2 * nx];
00050 }
```

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 228 of file [DerivationStencils.h](#).

```
00228 { return s4c(udata, 6); }
```

References [s4c\(\)](#).

Here is the call graph for this function:



6.4.2.39 s4f() [1/2]

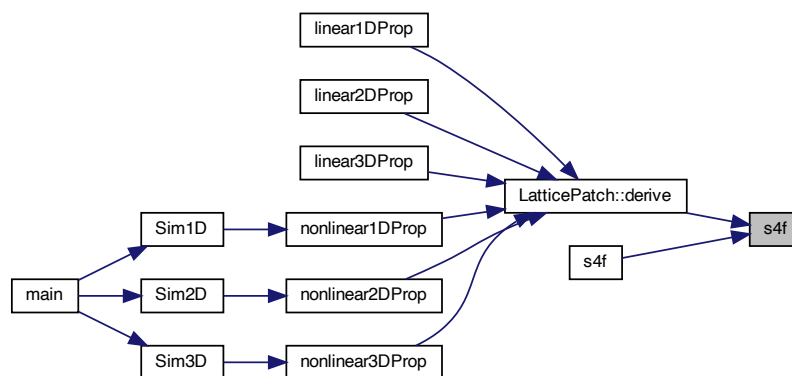
```
sunrealtype s4f (
    const sunrealtype * udata,
    int nx ) [inline]
```

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

```
00042 {
00043     return -1.0 / 12.0 * udata[-3 * nx] + 1.0 / 2.0 * udata[-2 * nx] -
00044           3.0 / 2.0 * udata[-1 * nx] + 5.0 / 6.0 * udata[0] +
00045           1.0 / 4.0 * udata[1 * nx];
00046 }
```

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 227 of file [DerivationStencils.h](#).

```
00227 { 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 231 of file [DerivationStencils.h](#).

```
00231 { return s5b(udata, 6); }
```

References [s5b\(\)](#).

Here is the call graph for this function:

**6.4.2.42 s5b()** [2/2]

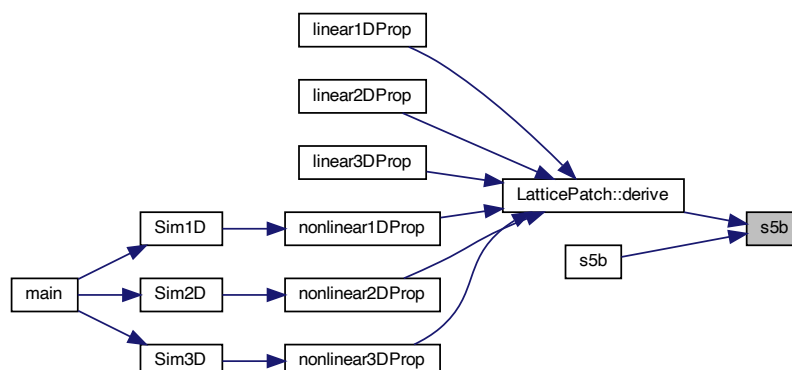
```
sunrealtype s5b (
    sunrealtype * udata,
    int nx ) [inline]
```

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

```
00061 {
00062     return 1.0 / 20.0 * udata[-2 * nx] - 1.0 / 2.0 * udata[-1 * nx] -
00063           1.0 / 3.0 * udata[0] + udata[1 * nx] - 1.0 / 4.0 * udata[2 * nx] +
00064           1.0 / 30.0 * udata[3 * nx];
00065 }
```

Referenced by [LatticePatch::derive\(\)](#), and [s5b\(\)](#).

Here is the caller graph for this function:



6.4.2.43 s5f() [1/2]

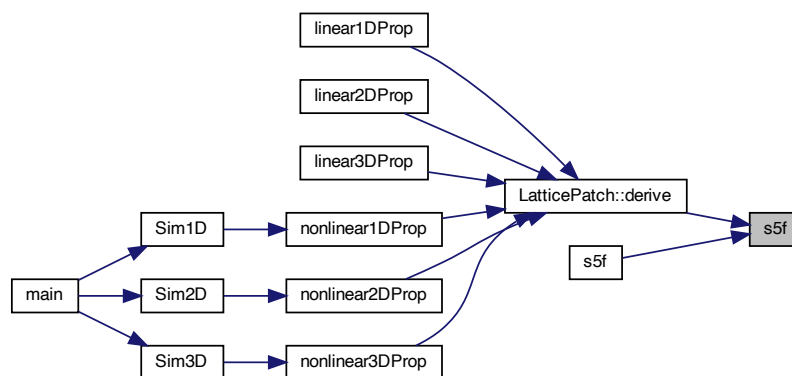
```
sunrealtype s5f (
    const sunrealtype * udata,
    int nx ) [inline]
```

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

```
00056 {
00057     return -1.0 / 30.0 * udata[-3 * nx] + 1.0 / 4.0 * udata[-2 * nx] -
00058           1.0 / 1.0 * udata[-1 * nx] + 1.0 / 3.0 * udata[0] +
00059           1.0 / 2.0 * udata[1 * nx] - 1.0 / 20.0 * udata[2 * nx];
00060 }
```

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 230 of file [DerivationStencils.h](#).

```
00230 { 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 77 of file [DerivationStencils.h](#).

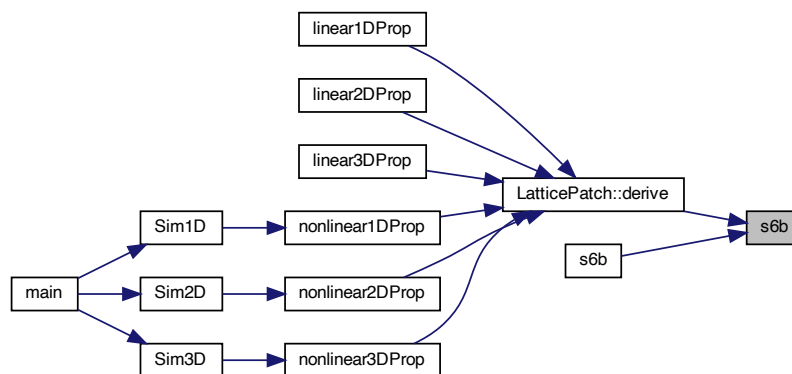
```

00077 {
00078     return 1.0 / 30.0 * udata[-2 * nx] - 2.0 / 5.0 * udata[-1 * nx] -
00079           7.0 / 12.0 * udata[0] + 4.0 / 3.0 * udata[1 * nx] -
00080           1.0 / 2.0 * udata[2 * nx] + 2.0 / 15.0 * udata[3 * nx] -
00081           1.0 / 60.0 * udata[4 * nx];
00082 }

```

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 234 of file [DerivationStencils.h](#).

```

00234 { return s6b(udata, 6); }

```

References [s6b\(\)](#).

Here is the call graph for this function:



6.4.2.47 s6c() [1/2]

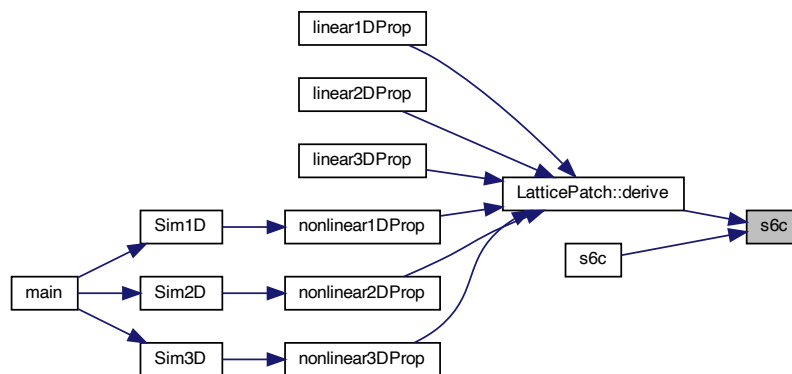
```
sunrealtype s6c (
    const sunrealtype * udata,
    int nx ) [inline]
```

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

```
00072 {
00073     return -1.0 / 60.0 * udata[-3 * nx] + 3.0 / 20.0 * udata[-2 * nx] -
00074           3.0 / 4.0 * udata[-1 * nx] + 0 + 3.0 / 4.0 * udata[1 * nx] -
00075           3.0 / 20.0 * udata[2 * nx] + 1.0 / 60.0 * udata[3 * nx];
00076 }
```

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 233 of file [DerivationStencils.h](#).

```
00233 { 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 66 of file [DerivationStencils.h](#).

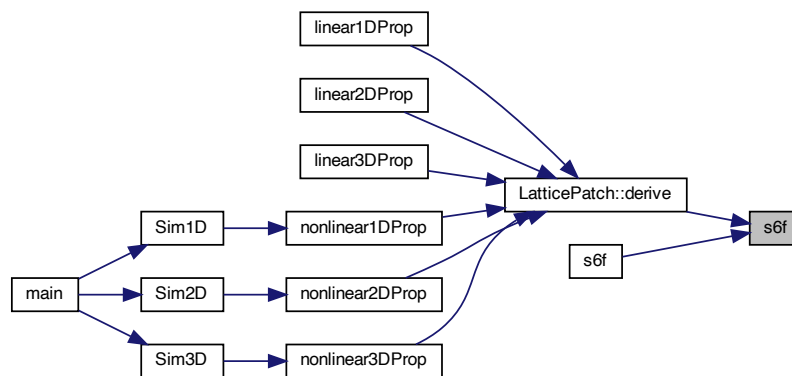
```

00066 {
00067     return 1.0 / 60.0 * udata[-4 * nx] - 2.0 / 15.0 * udata[-3 * nx] +
00068           1.0 / 2.0 * udata[-2 * nx] - 4.0 / 3.0 * udata[-1 * nx] +
00069           7.0 / 12.0 * udata[0] + 2.0 / 5.0 * udata[1 * nx] -
00070           1.0 / 30.0 * udata[2 * nx];
00071 }

```

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 232 of file [DerivationStencils.h](#).

```

00232 { 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 236 of file [DerivationStencils.h](#).

```
00236 { return s7b(udata, 6); }
```

References [s7b\(\)](#).

Here is the call graph for this function:

**6.4.2.52 s7b()** [2/2]

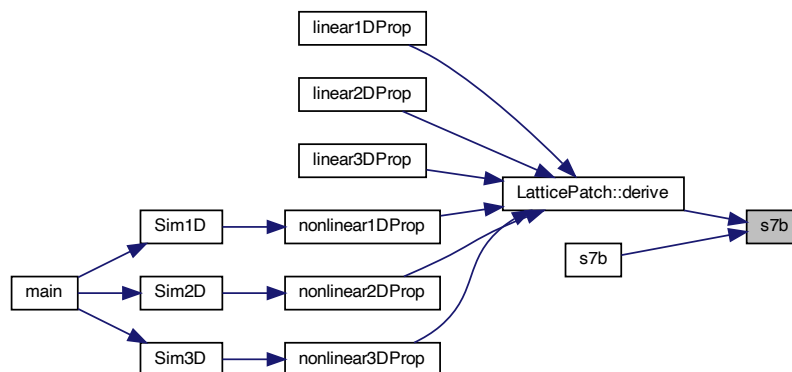
```
sunrealtype s7b (
    sunrealtype * udata,
    int nx ) [inline]
```

Definition at line 89 of file [DerivationStencils.h](#).

```
00089 {
00090     return -1.0 / 105.0 * udata[-3 * nx] + 1.0 / 10.0 * udata[-2 * nx] -
00091            3.0 / 5.0 * udata[-1 * nx] - 1.0 / 4.0 * udata[0] + udata[1 * nx] -
00092            3.0 / 10.0 * udata[2 * nx] + 1.0 / 15.0 * udata[3 * nx] -
00093            1.0 / 140.0 * udata[4 * nx];
00094 }
```

Referenced by [LatticePatch::derive\(\)](#), and [s7b\(\)](#).

Here is the caller graph for this function:



6.4.2.53 s7f() [1/2]

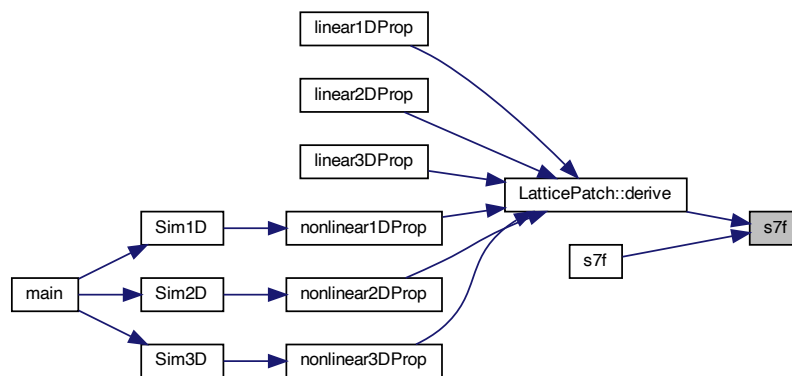
```
sunrealtype s7f (
    const sunrealtype * udata,
    int nx ) [inline]
```

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

```
00083 {
00084     return 1.0 / 140.0 * udata[-4 * nx] - 1.0 / 15.0 * udata[-3 * nx] +
00085           3.0 / 10.0 * udata[-2 * nx] - 1.0 / 1.0 * udata[-1 * nx] +
00086           1.0 / 4.0 * udata[0] + 3.0 / 5.0 * udata[1 * nx] -
00087           1.0 / 10.0 * udata[2 * nx] + 1.0 / 105.0 * udata[3 * nx];
00088 }
```

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 235 of file [DerivationStencils.h](#).

```
00235 { return s7f(udata, 6); }
```

References [s7f\(\)](#).

Here is the call graph for this function:



6.4.2.55 s8b() [1/2]

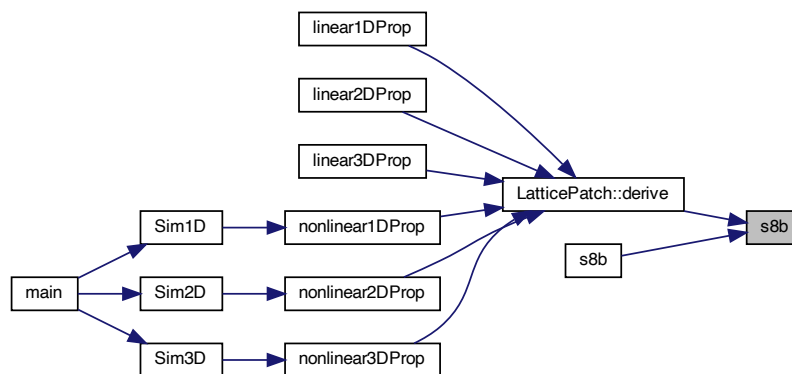
```
sunrealtype s8b (
    const sunrealtype * udata,
    int nx ) [inline]
```

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

```
00108 {
00109     return -1.0 / 168.0 * udata[-3 * nx] + 1.0 / 14.0 * udata[-2 * nx] -
00110            1.0 / 2.0 * udata[-1 * nx] - 9.0 / 20.0 * udata[0] +
00111            5.0 / 4.0 * udata[1 * nx] - 1.0 / 2.0 * udata[2 * nx] +
00112            1.0 / 6.0 * udata[3 * nx] - 1.0 / 28.0 * udata[4 * nx] +
00113            1.0 / 280.0 * udata[5 * nx];
00114 }
```

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 239 of file [DerivationStencils.h](#).

```
00239 { 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 102 of file [DerivationStencils.h](#).

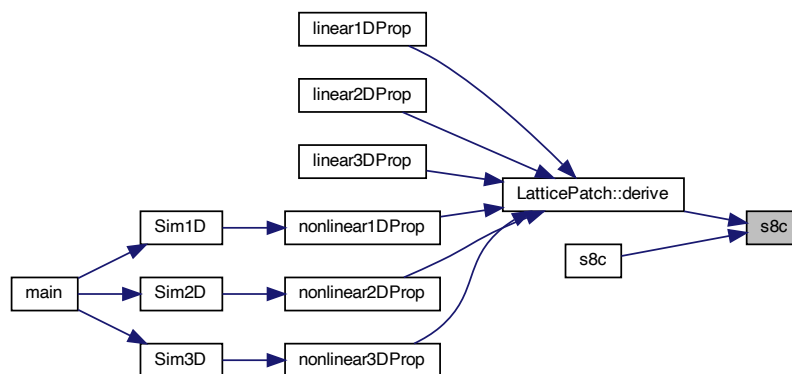
```

00102 {
00103     return 1.0 / 280.0 * udata[-4 * nx] - 4.0 / 105.0 * udata[-3 * nx] +
00104           1.0 / 5.0 * udata[-2 * nx] - 4.0 / 5.0 * udata[-1 * nx] + 0 +
00105           4.0 / 5.0 * udata[1 * nx] - 1.0 / 5.0 * udata[2 * nx] +
00106           4.0 / 105.0 * udata[3 * nx] - 1.0 / 280.0 * udata[4 * nx];
00107 }

```

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 238 of file [DerivationStencils.h](#).

```

00238 { return s8c(udata, 6); }

```

References [s8c\(\)](#).

Here is the call graph for this function:



6.4.2.59 s8f() [1/2]

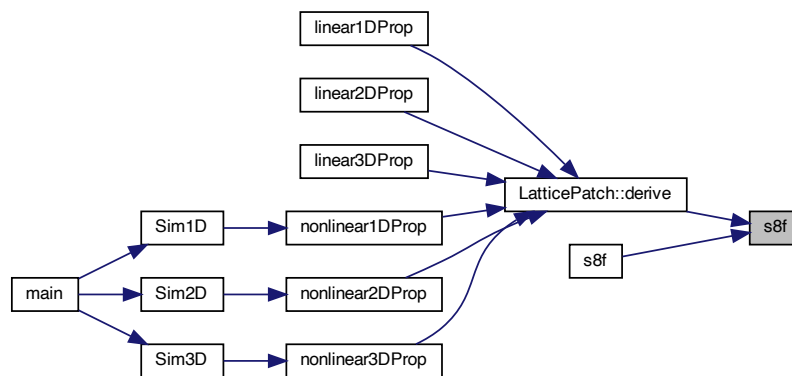
```
sunrealtype s8f (
    const sunrealtype * udata,
    int nx ) [inline]
```

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

```
00095 {
00096     return -1.0 / 280.0 * udata[-5 * nx] + 1.0 / 28.0 * udata[-4 * nx] -
00097            1.0 / 6.0 * udata[-3 * nx] + 1.0 / 2.0 * udata[-2 * nx] -
00098            5.0 / 4.0 * udata[-1 * nx] + 9.0 / 20.0 * udata[0] +
00099            1.0 / 2.0 * udata[1 * nx] - 1.0 / 14.0 * udata[2 * nx] +
00100            1.0 / 168.0 * udata[3 * nx];
00101 }
```

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 237 of file [DerivationStencils.h](#).

```
00237 { 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 241 of file [DerivationStencils.h](#).

```
00241 { return s9b(udata, 6); }
```

References [s9b\(\)](#).

Here is the call graph for this function:

**6.4.2.62 s9b()** [2/2]

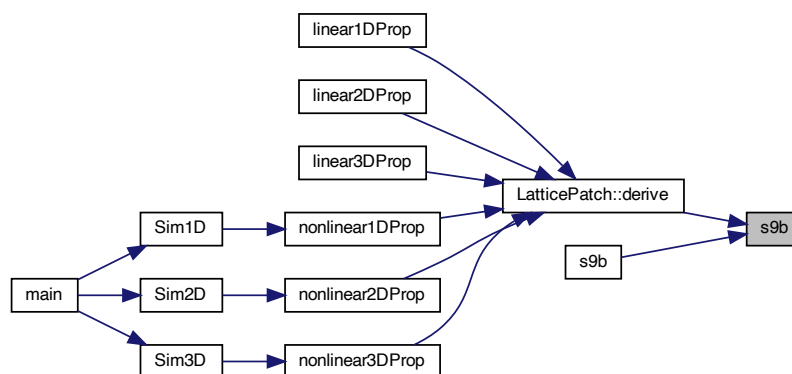
```
sunrealtype s9b (
    sunrealtype * udata,
    int nx ) [inline]
```

Definition at line 122 of file [DerivationStencils.h](#).

```
00122 {
00123     return 1.0 / 504.0 * udata[-4 * nx] - 1.0 / 42.0 * udata[-3 * nx] +
00124            1.0 / 7.0 * udata[-2 * nx] - 2.0 / 3.0 * udata[-1 * nx] -
00125            1.0 / 5.0 * udata[0] + udata[1 * nx] - 1.0 / 3.0 * udata[2 * nx] +
00126            2.0 / 21.0 * udata[3 * nx] - 1.0 / 56.0 * udata[4 * nx] +
00127            1.0 / 630.0 * udata[5 * nx];
00128 }
```

Referenced by [LatticePatch::derive\(\)](#), and [s9b\(\)](#).

Here is the caller graph for this function:



6.4.2.63 s9f() [1/2]

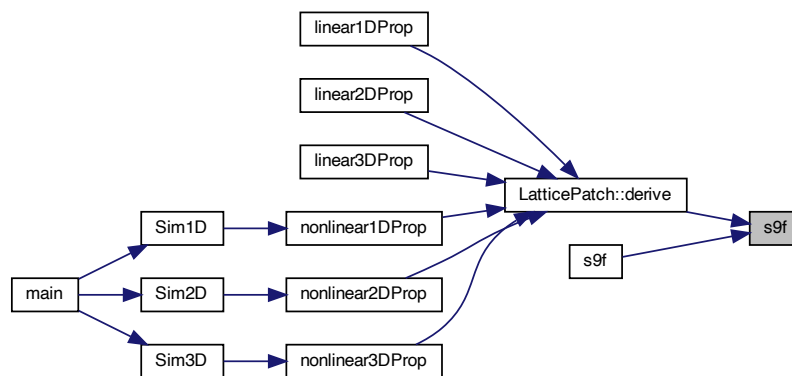
```
sunrealtype s9f (
    const sunrealtype * udata,
    int nx ) [inline]
```

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

```
00115 {
00116     return -1.0 / 630.0 * udata[-5 * nx] + 1.0 / 56.0 * udata[-4 * nx] -
00117            2.0 / 21.0 * udata[-3 * nx] + 1.0 / 3.0 * udata[-2 * nx] -
00118            1.0 / 1.0 * udata[-1 * nx] + 1.0 / 5.0 * udata[0] +
00119            2.0 / 3.0 * udata[1 * nx] - 1.0 / 7.0 * udata[2 * nx] +
00120            1.0 / 42.0 * udata[3 * nx] - 1.0 / 504.0 * udata[4 * nx];
00121 }
```

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 240 of file [DerivationStencils.h](#).

```
00240 { 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 #pragma once
00007
00008 #include <sundials/sundials_types.h> /* definition of type sunrealtype */
00009
00010 //////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
00011 // Stencils with variable nx -- data point dimension //
00012 //////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
00013
00014 // Downwind (forward) differentiating
00015 inline sunrealtype s1f(sunrealtype *udata, int nx) {
00016     return -1.0 / 1.0 * udata[-1 * nx] + udata[0];
00017 }
00018 // Upwind (backward) differentiating
00019 inline sunrealtype s1b(sunrealtype *udata, int nx) {
00020     return -1.0 / 1.0 * udata[0] + udata[1 * nx];
00021 }
00022
00023 inline sunrealtype s2f(const sunrealtype *udata, int nx) {
00024     return 1.0 / 2.0 * udata[-2 * nx] - 2.0 / 1.0 * udata[-1 * nx] +
00025         3.0 / 2.0 * udata[0];
00026 }
00027 inline sunrealtype s2c(const sunrealtype *udata, int nx) {
00028     return -1.0 / 2.0 * udata[-1 * nx] + 0 + 1.0 / 2.0 * udata[1 * nx];
00029 }
00030 inline sunrealtype s2b(const sunrealtype *udata, int nx) {
00031     return -3.0 / 2.0 * udata[0] + 2.0 / 1.0 * udata[1 * nx] -
00032         1.0 / 2.0 * udata[2 * nx];
00033 }
00034 inline sunrealtype s3f(const sunrealtype *udata, int nx) {
00035     return 1.0 / 6.0 * udata[-2 * nx] - 1.0 / 1.0 * udata[-1 * nx] +
00036         1.0 / 2.0 * udata[0] + 1.0 / 3.0 * udata[1 * nx];
00037 }
00038 inline sunrealtype s3b(sunrealtype *udata, int nx) {
00039     return -1.0 / 3.0 * udata[-1 * nx] - 1.0 / 2.0 * udata[0] + udata[1 * nx] -
00040         1.0 / 6.0 * udata[2 * nx];
00041 }
00042 inline sunrealtype s4f(const sunrealtype *udata, int nx) {
00043     return -1.0 / 12.0 * udata[-3 * nx] + 1.0 / 2.0 * udata[-2 * nx] -
00044         3.0 / 2.0 * udata[-1 * nx] + 5.0 / 6.0 * udata[0] +
00045         1.0 / 4.0 * udata[1 * nx];
00046 }
00047 inline sunrealtype s4c(const sunrealtype *udata, int nx) {
00048     return 1.0 / 12.0 * udata[-2 * nx] - 2.0 / 3.0 * udata[-1 * nx] + 0 +
00049         2.0 / 3.0 * udata[1 * nx] - 1.0 / 12.0 * udata[2 * nx];
00050 }
00051 inline sunrealtype s4b(const sunrealtype *udata, int nx) {
00052     return -1.0 / 4.0 * udata[-1 * nx] - 5.0 / 6.0 * udata[0] +
00053         3.0 / 2.0 * udata[1 * nx] - 1.0 / 2.0 * udata[2 * nx] +
00054         1.0 / 12.0 * udata[3 * nx];
00055 }
00056 inline sunrealtype s5f(const sunrealtype *udata, int nx) {
00057     return -1.0 / 30.0 * udata[-3 * nx] + 1.0 / 4.0 * udata[-2 * nx] -
00058         1.0 / 1.0 * udata[-1 * nx] + 1.0 / 3.0 * udata[0] +
00059         1.0 / 2.0 * udata[1 * nx] - 1.0 / 20.0 * udata[2 * nx];
00060 }
00061 inline sunrealtype s5b(sunrealtype *udata, int nx) {
00062     return 1.0 / 20.0 * udata[-2 * nx] - 1.0 / 2.0 * udata[-1 * nx] -
00063         1.0 / 3.0 * udata[0] + udata[1 * nx] - 1.0 / 4.0 * udata[2 * nx] +
00064         1.0 / 30.0 * udata[3 * nx];
00065 }
00066 inline sunrealtype s6f(const sunrealtype *udata, int nx) {
00067     return 1.0 / 60.0 * udata[-4 * nx] - 2.0 / 15.0 * udata[-3 * nx] +
00068         1.0 / 2.0 * udata[-2 * nx] - 4.0 / 3.0 * udata[-1 * nx] +
00069         7.0 / 12.0 * udata[0] + 2.0 / 5.0 * udata[1 * nx] -
00070         1.0 / 30.0 * udata[2 * nx];
00071 }
00072 inline sunrealtype s6c(const sunrealtype *udata, int nx) {
00073     return -1.0 / 60.0 * udata[-3 * nx] + 3.0 / 20.0 * udata[-2 * nx] -
00074         3.0 / 4.0 * udata[-1 * nx] + 0 + 3.0 / 4.0 * udata[1 * nx] -
00075         3.0 / 20.0 * udata[2 * nx] + 1.0 / 60.0 * udata[3 * nx];
00076 }
00077 inline sunrealtype s6b(const sunrealtype *udata, int nx) {
00078     return 1.0 / 30.0 * udata[-2 * nx] - 2.0 / 5.0 * udata[-1 * nx] -
00079         7.0 / 12.0 * udata[0] + 4.0 / 3.0 * udata[1 * nx] -
00080         1.0 / 2.0 * udata[2 * nx] + 2.0 / 15.0 * udata[3 * nx] -
00081         1.0 / 60.0 * udata[4 * nx];
00082 }
```

```

00083 inline sunrealtype s7f(const sunrealtype *udata, int nx) {
00084     return 1.0 / 140.0 * udata[-4 * nx] - 1.0 / 15.0 * udata[-3 * nx] +
00085         3.0 / 10.0 * udata[-2 * nx] - 1.0 / 1.0 * udata[-1 * nx] +
00086         1.0 / 4.0 * udata[0] + 3.0 / 5.0 * udata[1 * nx] -
00087         1.0 / 10.0 * udata[2 * nx] + 1.0 / 105.0 * udata[3 * nx];
00088 }
00089 inline sunrealtype s7b(sunrealtype *udata, int nx) {
00090     return -1.0 / 105.0 * udata[-3 * nx] + 1.0 / 10.0 * udata[-2 * nx] -
00091         3.0 / 5.0 * udata[-1 * nx] - 1.0 / 4.0 * udata[0] + udata[1 * nx] -
00092         3.0 / 10.0 * udata[2 * nx] + 1.0 / 15.0 * udata[3 * nx] -
00093         1.0 / 140.0 * udata[4 * nx];
00094 }
00095 inline sunrealtype s8f(const sunrealtype *udata, int nx) {
00096     return -1.0 / 280.0 * udata[-5 * nx] + 1.0 / 28.0 * udata[-4 * nx] -
00097         1.0 / 6.0 * udata[-3 * nx] + 1.0 / 2.0 * udata[-2 * nx] -
00098         5.0 / 4.0 * udata[-1 * nx] + 9.0 / 20.0 * udata[0] +
00099         1.0 / 2.0 * udata[1 * nx] - 1.0 / 14.0 * udata[2 * nx] +
00100         1.0 / 168.0 * udata[3 * nx];
00101 }
00102 inline sunrealtype s8c(const sunrealtype *udata, int nx) {
00103     return 1.0 / 280.0 * udata[-4 * nx] - 4.0 / 105.0 * udata[-3 * nx] +
00104         1.0 / 5.0 * udata[-2 * nx] - 4.0 / 5.0 * udata[-1 * nx] + 0 +
00105         4.0 / 5.0 * udata[1 * nx] - 1.0 / 5.0 * udata[2 * nx] +
00106         4.0 / 105.0 * udata[3 * nx] - 1.0 / 280.0 * udata[4 * nx];
00107 }
00108 inline sunrealtype s8b(const sunrealtype *udata, int nx) {
00109     return -1.0 / 168.0 * udata[-3 * nx] + 1.0 / 14.0 * udata[-2 * nx] -
00110         1.0 / 2.0 * udata[-1 * nx] - 9.0 / 20.0 * udata[0] +
00111         5.0 / 4.0 * udata[1 * nx] - 1.0 / 2.0 * udata[2 * nx] +
00112         1.0 / 6.0 * udata[3 * nx] - 1.0 / 28.0 * udata[4 * nx] +
00113         1.0 / 280.0 * udata[5 * nx];
00114 }
00115 inline sunrealtype s9f(const sunrealtype *udata, int nx) {
00116     return -1.0 / 630.0 * udata[-5 * nx] + 1.0 / 56.0 * udata[-4 * nx] -
00117         2.0 / 21.0 * udata[-3 * nx] + 1.0 / 3.0 * udata[-2 * nx] -
00118         1.0 / 1.0 * udata[-1 * nx] + 1.0 / 5.0 * udata[0] +
00119         2.0 / 3.0 * udata[1 * nx] - 1.0 / 7.0 * udata[2 * nx] +
00120         1.0 / 42.0 * udata[3 * nx] - 1.0 / 504.0 * udata[4 * nx];
00121 }
00122 inline sunrealtype s9b(sunrealtype *udata, int nx) {
00123     return 1.0 / 504.0 * udata[-4 * nx] - 1.0 / 42.0 * udata[-3 * nx] +
00124         1.0 / 7.0 * udata[-2 * nx] - 2.0 / 3.0 * udata[-1 * nx] -
00125         1.0 / 5.0 * udata[0] + udata[1 * nx] - 1.0 / 3.0 * udata[2 * nx] +
00126         2.0 / 21.0 * udata[3 * nx] - 1.0 / 56.0 * udata[4 * nx] +
00127         1.0 / 630.0 * udata[5 * nx];
00128 }
00129 inline sunrealtype s10f(const sunrealtype *udata, int nx) {
00130     return 1.0 / 1260.0 * udata[-6 * nx] - 1.0 / 105.0 * udata[-5 * nx] +
00131         3.0 / 56.0 * udata[-4 * nx] - 4.0 / 21.0 * udata[-3 * nx] +
00132         1.0 / 2.0 * udata[-2 * nx] - 6.0 / 5.0 * udata[-1 * nx] +
00133         11.0 / 30.0 * udata[0] + 4.0 / 7.0 * udata[1 * nx] -
00134         3.0 / 28.0 * udata[2 * nx] + 1.0 / 63.0 * udata[3 * nx] -
00135         1.0 / 840.0 * udata[4 * nx];
00136 }
00137 inline sunrealtype s10c(const sunrealtype *udata, int nx) {
00138     return -1.0 / 1260.0 * udata[-5 * nx] + 5.0 / 504.0 * udata[-4 * nx] -
00139         5.0 / 84.0 * udata[-3 * nx] + 5.0 / 21.0 * udata[-2 * nx] -
00140         5.0 / 6.0 * udata[-1 * nx] + 0 + 5.0 / 6.0 * udata[1 * nx] -
00141         5.0 / 21.0 * udata[2 * nx] + 5.0 / 84.0 * udata[3 * nx] -
00142         5.0 / 504.0 * udata[4 * nx] + 1.0 / 1260.0 * udata[5 * nx];
00143 }
00144 inline sunrealtype s10b(const sunrealtype *udata, int nx) {
00145     return 1.0 / 840.0 * udata[-4 * nx] - 1.0 / 63.0 * udata[-3 * nx] +
00146         3.0 / 28.0 * udata[-2 * nx] - 4.0 / 7.0 * udata[-1 * nx] -
00147         11.0 / 30.0 * udata[0] + 6.0 / 5.0 * udata[1 * nx] -
00148         1.0 / 2.0 * udata[2 * nx] + 4.0 / 21.0 * udata[3 * nx] -
00149         3.0 / 56.0 * udata[4 * nx] + 1.0 / 105.0 * udata[5 * nx] -
00150         1.0 / 1260.0 * udata[6 * nx];
00151 }
00152 inline sunrealtype s11f(const sunrealtype *udata, int nx) {
00153     return 1.0 / 2772.0 * udata[-6 * nx] - 1.0 / 210.0 * udata[-5 * nx] +
00154         5.0 / 168.0 * udata[-4 * nx] - 5.0 / 42.0 * udata[-3 * nx] +
00155         5.0 / 14.0 * udata[-2 * nx] - 1.0 / 1.0 * udata[-1 * nx] +
00156         1.0 / 6.0 * udata[0] + 5.0 / 7.0 * udata[1 * nx] -
00157         5.0 / 28.0 * udata[2 * nx] + 5.0 / 126.0 * udata[3 * nx] -
00158         1.0 / 168.0 * udata[4 * nx] + 1.0 / 2310.0 * udata[5 * nx];
00159 }
00160 inline sunrealtype s11b(sunrealtype *udata, int nx) {
00161     return -1.0 / 2310.0 * udata[-5 * nx] + 1.0 / 168.0 * udata[-4 * nx] -
00162         5.0 / 126.0 * udata[-3 * nx] + 5.0 / 28.0 * udata[-2 * nx] -
00163         5.0 / 7.0 * udata[-1 * nx] - 1.0 / 6.0 * udata[0] + udata[1 * nx] -
00164         5.0 / 14.0 * udata[2 * nx] + 5.0 / 42.0 * udata[3 * nx] -
00165         5.0 / 168.0 * udata[4 * nx] + 1.0 / 210.0 * udata[5 * nx] -
00166         1.0 / 2772.0 * udata[6 * nx];
00167 }
00168 inline sunrealtype s12f(const sunrealtype *udata, int nx) {
00169     return -1.0 / 5544.0 * udata[-7 * nx] + 1.0 / 396.0 * udata[-6 * nx] -

```

```

00170         1.0 / 60.0 * udata[-5 * nx] + 5.0 / 72.0 * udata[-4 * nx] -
00171         5.0 / 24.0 * udata[-3 * nx] + 1.0 / 2.0 * udata[-2 * nx] -
00172         7.0 / 6.0 * udata[-1 * nx] + 13.0 / 42.0 * udata[0] +
00173         5.0 / 8.0 * udata[1 * nx] - 5.0 / 36.0 * udata[2 * nx] +
00174         1.0 / 36.0 * udata[3 * nx] - 1.0 / 264.0 * udata[4 * nx] +
00175         1.0 / 3960.0 * udata[5 * nx];
00176     }
00177     inline sunrealtype s12c(const sunrealtype *udata, int nx) {
00178         return 1.0 / 5544.0 * udata[-6 * nx] - 1.0 / 385.0 * udata[-5 * nx] +
00179         1.0 / 56.0 * udata[-4 * nx] - 5.0 / 63.0 * udata[-3 * nx] +
00180         15.0 / 56.0 * udata[-2 * nx] - 6.0 / 7.0 * udata[-1 * nx] + 0 +
00181         6.0 / 7.0 * udata[1 * nx] - 15.0 / 56.0 * udata[2 * nx] +
00182         5.0 / 63.0 * udata[3 * nx] - 1.0 / 56.0 * udata[4 * nx] +
00183         1.0 / 385.0 * udata[5 * nx] - 1.0 / 5544.0 * udata[6 * nx];
00184     }
00185     inline sunrealtype s12b(const sunrealtype *udata, int nx) {
00186         return -1.0 / 3960.0 * udata[-5 * nx] + 1.0 / 264.0 * udata[-4 * nx] -
00187         1.0 / 36.0 * udata[-3 * nx] + 5.0 / 36.0 * udata[-2 * nx] -
00188         5.0 / 8.0 * udata[-1 * nx] - 13.0 / 42.0 * udata[0] +
00189         7.0 / 6.0 * udata[1 * nx] - 1.0 / 2.0 * udata[2 * nx] +
00190         5.0 / 24.0 * udata[3 * nx] - 5.0 / 72.0 * udata[4 * nx] +
00191         1.0 / 60.0 * udata[5 * nx] - 1.0 / 396.0 * udata[6 * nx] +
00192         1.0 / 5544.0 * udata[7 * nx];
00193     }
00194     // #pragma omp declare simd notinbranch uniform(nx) simdlen(13) // Is this safe
00195     // here? Do I need critical or atomic?
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     // #pragma omp declare simd notinbranch uniform(nx) simdlen(13)
00206     inline sunrealtype s13b(sunrealtype *udata, int nx) {
00207         return 1.0 / 10296.0 * udata[-6 * nx] - 1.0 / 660.0 * udata[-5 * nx] +
00208         1.0 / 88.0 * udata[-4 * nx] - 1.0 / 18.0 * udata[-3 * nx] +
00209         5.0 / 24.0 * udata[-2 * nx] - 3.0 / 4.0 * udata[-1 * nx] -
00210         1.0 / 7.0 * udata[0] + udata[1 * nx] - 3.0 / 8.0 * udata[2 * nx] +
00211         5.0 / 36.0 * udata[3 * nx] - 1.0 / 24.0 * udata[4 * nx] +
00212         1.0 / 110.0 * udata[5 * nx] - 1.0 / 792.0 * udata[6 * nx] +
00213         1.0 / 12012.0 * udata[7 * nx];
00214     }
00215
00216     // Stencils with nx fixed to 6//
00217
00218
00219
00220     inline sunrealtype s1f(sunrealtype *udata) { return s1f(udata, 6); }
00221     inline sunrealtype s1b(sunrealtype *udata) { return s1b(udata, 6); }
00222     inline sunrealtype s2f(sunrealtype *udata) { return s2f(udata, 6); }
00223     inline sunrealtype s2c(sunrealtype *udata) { return s2c(udata, 6); }
00224     inline sunrealtype s2b(sunrealtype *udata) { return s2b(udata, 6); }
00225     inline sunrealtype s3f(sunrealtype *udata) { return s3f(udata, 6); }
00226     inline sunrealtype s3b(sunrealtype *udata) { return s3b(udata, 6); }
00227     inline sunrealtype s4f(sunrealtype *udata) { return s4f(udata, 6); }
00228     inline sunrealtype s4c(sunrealtype *udata) { return s4c(udata, 6); }
00229     inline sunrealtype s4b(sunrealtype *udata) { return s4b(udata, 6); }
00230     inline sunrealtype s5f(sunrealtype *udata) { return s5f(udata, 6); }
00231     inline sunrealtype s5b(sunrealtype *udata) { return s5b(udata, 6); }
00232     inline sunrealtype s6f(sunrealtype *udata) { return s6f(udata, 6); }
00233     inline sunrealtype s6c(sunrealtype *udata) { return s6c(udata, 6); }
00234     inline sunrealtype s6b(sunrealtype *udata) { return s6b(udata, 6); }
00235     inline sunrealtype s7f(sunrealtype *udata) { return s7f(udata, 6); }
00236     inline sunrealtype s7b(sunrealtype *udata) { return s7b(udata, 6); }
00237     inline sunrealtype s8f(sunrealtype *udata) { return s8f(udata, 6); }
00238     inline sunrealtype s8c(sunrealtype *udata) { return s8c(udata, 6); }
00239     inline sunrealtype s8b(sunrealtype *udata) { return s8b(udata, 6); }
00240     inline sunrealtype s9f(sunrealtype *udata) { return s9f(udata, 6); }
00241     inline sunrealtype s9b(sunrealtype *udata) { return s9b(udata, 6); }
00242     inline sunrealtype s10f(sunrealtype *udata) { return s10f(udata, 6); }
00243     inline sunrealtype s10c(sunrealtype *udata) { return s10c(udata, 6); }
00244     inline sunrealtype s10b(sunrealtype *udata) { return s10b(udata, 6); }
00245     inline sunrealtype s11f(sunrealtype *udata) { return s11f(udata, 6); }
00246     inline sunrealtype s11b(sunrealtype *udata) { return s11b(udata, 6); }
00247     inline sunrealtype s12f(sunrealtype *udata) { return s12f(udata, 6); }
00248     inline sunrealtype s12c(sunrealtype *udata) { return s12c(udata, 6); }
00249     inline sunrealtype s12b(sunrealtype *udata) { return s12b(udata, 6); }
00250     inline sunrealtype s13f(sunrealtype *udata) { return s13f(udata, 6); }
00251     inline sunrealtype s13b(sunrealtype *udata) { return s13b(udata, 6); }
00252

```

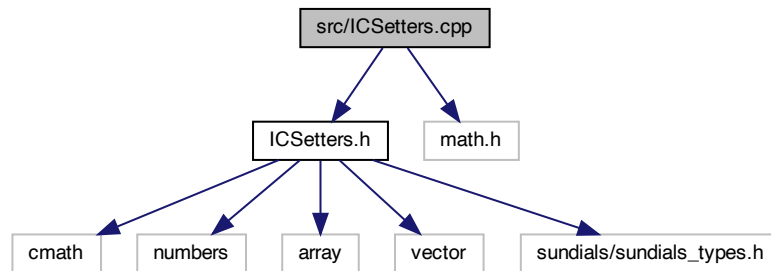
6.6 src/ICSetters.cpp File Reference

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 /**#pragma omp declare simd uniform(x,y,z) linear(pTo6Space:6)
00028 void PlaneWave1D::addToSpace(const sunrealtype x, const sunrealtype y, const sunrealtype z,
00029                             sunrealtype *pTo6Space) const {
00030     const sunrealtype wavelength =
00031         sqrt(kx * kx + ky * ky + kz * kz); /** \f$ 1/\lambda \f$ */

```

```

00032     const sunrealtype kScalarX = (kx * x + ky * y + kz * z) * 2 *
00033                                     numbers::pi; /* \f$ 2\pi \ \vec{k} \cdot \vec{x} \f$ */
00034     // Plane wave definition
00035     const array<sunrealtype, 3> E{{ /* E-field vector */
00036                                     px * cos(kScalarX - phix), /* \f$ E_x \f$ */
00037                                     py * cos(kScalarX - phiy), /* \f$ E_y \f$ */
00038                                     pz * cos(kScalarX - phiz)}}; /* \f$ E_z \f$ */
00039     // Put E-field into space
00040     pTo6Space[0] += E[0];
00041     pTo6Space[1] += E[1];
00042     pTo6Space[2] += E[2];
00043     // and B-field
00044     pTo6Space[3] += (ky * E[2] - kz * E[1]) / wavelength;
00045     pTo6Space[4] += (kz * E[0] - kx * E[2]) / wavelength;
00046     pTo6Space[5] += (kx * E[1] - ky * E[0]) / wavelength;
00047 }
00048
00049 /** PlaneWave2D construction with */
00050 PlaneWave2D::PlaneWave2D(vector<sunrealtype> k, vector<sunrealtype> p,
00051                          vector<sunrealtype> phi) {
00052     kx = k[0]; /** - wavevectors \f$ k_x \f$ */
00053     ky = k[1]; /** - \f$ k_y \f$ */
00054     kz = k[2]; /** - \f$ k_z \f$ normalized to \f$ 1/\lambda \f$ */
00055     // Amplitude bug: lower by factor 9
00056     px = p[0] / 9; /** - amplitude (polarization) in x-direction \f$ p_x \f$ */
00057     py = p[1] / 9; /** - amplitude (polarization) in y-direction \f$ p_y \f$ */
00058     pz = p[2] / 9; /** - amplitude (polarization) in z-direction \f$ p_z \f$ */
00059     phix = phi[0]; /** - phase shift in x-direction \f$ \phi_x \f$ */
00060     phiy = phi[1]; /** - phase shift in y-direction \f$ \phi_y \f$ */
00061     phiz = phi[2]; /** - phase shift in z-direction \f$ \phi_z \f$ */
00062 }
00063
00064 /** PlaneWave2D implementation in space */
00065 // #pragma omp declare simd uniform(x,y,z) linear(pTo6Space:6)
00066 void PlaneWave2D::addToSpace(const sunrealtype x, const sunrealtype y, const sunrealtype z,
00067                             sunrealtype *pTo6Space) const {
00068     const sunrealtype wavelength =
00069         sqrt(kx * kx + ky * ky + kz * kz); /* \f$ 1/\lambda \f$ */
00070     const sunrealtype kScalarX = (kx * x + ky * y + kz * z) * 2 *
00071                                     numbers::pi; /* \f$ 2\pi \ \vec{k} \cdot \vec{x} \f$ */
00072     // Plane wave definition
00073     const array<sunrealtype, 3> E{{ /* E-field vector */
00074                                     px * cos(kScalarX - phix), /* \f$ E_x \f$ */
00075                                     py * cos(kScalarX - phiy), /* \f$ E_y \f$ */
00076                                     pz * cos(kScalarX - phiz)}}; /* \f$ E_z \f$ */
00077     // Put E-field into space
00078     pTo6Space[0] += E[0];
00079     pTo6Space[1] += E[1];
00080     pTo6Space[2] += E[2];
00081     // and B-field
00082     pTo6Space[3] += (ky * E[2] - kz * E[1]) / wavelength;
00083     pTo6Space[4] += (kz * E[0] - kx * E[2]) / wavelength;
00084     pTo6Space[5] += (kx * E[1] - ky * E[0]) / wavelength;
00085 }
00086
00087 /** PlaneWave3D construction with */
00088 PlaneWave3D::PlaneWave3D(vector<sunrealtype> k, vector<sunrealtype> p,
00089                          vector<sunrealtype> phi) {
00090     kx = k[0]; /** - wavevectors \f$ k_x \f$ */
00091     ky = k[1]; /** - \f$ k_y \f$ */
00092     kz = k[2]; /** - \f$ k_z \f$ normalized to \f$ 1/\lambda \f$ */
00093     px = p[0]; /** - amplitude (polarization) in x-direction \f$ p_x \f$ */
00094     py = p[1]; /** - amplitude (polarization) in y-direction \f$ p_y \f$ */
00095     pz = p[2]; /** - amplitude (polarization) in z-direction \f$ p_z \f$ */
00096     phix = phi[0]; /** - phase shift in x-direction \f$ \phi_x \f$ */
00097     phiy = phi[1]; /** - phase shift in y-direction \f$ \phi_y \f$ */
00098     phiz = phi[2]; /** - phase shift in z-direction \f$ \phi_z \f$ */
00099 }
00100
00101 /** PlaneWave3D implementation in space */
00102 // #pragma omp declare simd uniform(x,y,z) linear(pTo6Space:6)
00103 void PlaneWave3D::addToSpace(sunrealtype x, sunrealtype y, sunrealtype z,
00104                             sunrealtype *pTo6Space) const {
00105     const sunrealtype wavelength =
00106         sqrt(kx * kx + ky * ky + kz * kz); /* \f$ 1/\lambda \f$ */
00107     const sunrealtype kScalarX = (kx * x + ky * y + kz * z) * 2 *
00108                                     numbers::pi; /* \f$ 2\pi \ \vec{k} \cdot \vec{x} \f$ */
00109     // Plane wave definition
00110     const array<sunrealtype, 3> E{{ /* E-field vector \f$ \vec{E} \f$ */
00111                                     px * cos(kScalarX - phix), /* \f$ E_x \f$ */
00112                                     py * cos(kScalarX - phiy), /* \f$ E_y \f$ */
00113                                     pz * cos(kScalarX - phiz)}}; /* \f$ E_z \f$ */
00114     // Put E-field into space
00115     pTo6Space[0] += E[0];
00116     pTo6Space[1] += E[1];
00117     pTo6Space[2] += E[2];
00118     // and B-field

```

```

00119 pTo6Space[3] += (ky * E[2] - kz * E[1]) / wavelength;
00120 pTo6Space[4] += (kz * E[0] - kx * E[2]) / wavelength;
00121 pTo6Space[5] += (kx * E[1] - ky * E[0]) / wavelength;
00122 }
00123
00124 /** Gauss1D construction with */
00125 Gauss1D::Gauss1D(vector<sunrealtype> k, vector<sunrealtype> p,
00126                  vector<sunrealtype> xo, sunrealtype phig_,
00127                  vector<sunrealtype> phi) {
00128     kx = k[0]; /** - wavevectors \f$ k_x \f$ */
00129     ky = k[1]; /** - \f$ k_y \f$ */
00130     kz = k[2]; /** - \f$ k_z \f$ normalized to \f$ 1/\lambda \f$ */
00131     px = p[0]; /** - amplitude (polarization) in x-direction */
00132     py = p[1]; /** - amplitude (polarization) in y-direction */
00133     pz = p[2]; /** - amplitude (polarization) in z-direction */
00134     phix = phi[0]; /** - phase shift in x-direction */
00135     phiy = phi[1]; /** - phase shift in y-direction */
00136     phiz = phi[2]; /** - phase shift in z-direction */
00137     phig = phig_; /** - width */
00138     x0x = xo[0]; /** - shift from origin in x-direction */
00139     x0y = xo[1]; /** - shift from origin in y-direction */
00140     x0z = xo[2]; /** - shift from origin in z-direction */
00141 }
00142
00143 /** Gauss1D implementation in space */
00144 // #pragma omp declare simd uniform(x,y,z) linear(pTo6Space:6)
00145 void Gauss1D::addToSpace(sunrealtype x, sunrealtype y, sunrealtype z,
00146                          sunrealtype *pTo6Space) const {
00147     const sunrealtype wavelength =
00148         sqrt(kx * kx + ky * ky + kz * kz); /** \f$ 1/\lambda \f$ */
00149     x = x - x0x; /** x-coordinate minus shift from origin */
00150     y = y - x0y; /** y-coordinate minus shift from origin */
00151     z = z - x0z; /** z-coordinate minus shift from origin */
00152     const sunrealtype kScalarX = (kx * x + ky * y + kz * z) * 2 *
00153         numbers::pi; /** \f$ 2\pi \cdot \vec{k} \cdot \vec{x} \f$ */
00154     const sunrealtype envelopeAmp =
00155         exp(-(x * x + y * y + z * z) / phig / phig); /** enveloping Gauss shape */
00156     /** Gaussian wave definition
00157     const array<sunrealtype, 3> E{
00158         {
00159             /** E-field vector */
00160             px * cos(kScalarX - phix) * envelopeAmp, /** \f$ E_x \f$ */
00161             py * cos(kScalarX - phiy) * envelopeAmp, /** \f$ E_y \f$ */
00162             pz * cos(kScalarX - phiz) * envelopeAmp}}; /** \f$ E_z \f$ */
00163     /** Put E-field into space
00164     pTo6Space[0] += E[0];
00165     pTo6Space[1] += E[1];
00166     pTo6Space[2] += E[2];
00167     /** and B-field
00168     pTo6Space[3] += (ky * E[2] - kz * E[1]) / wavelength;
00169     pTo6Space[4] += (kz * E[0] - kx * E[2]) / wavelength;
00170     pTo6Space[5] += (kx * E[1] - ky * E[0]) / wavelength;
00171 }
00172
00173 /** Gauss2D construction with */
00174 Gauss2D::Gauss2D(vector<sunrealtype> dis_, vector<sunrealtype> axis_,
00175                  sunrealtype Amp_, sunrealtype phip_, sunrealtype w0_,
00176                  sunrealtype zr_, sunrealtype Ph0_, sunrealtype PhA_) {
00177     dis = dis_; /** - center it approaches */
00178     axis = axis_; /** - direction form where it comes */
00179     Amp = Amp_; /** - amplitude */
00180     phip = phip_; /** - polarization rotation from TE-mode */
00181     w0 = w0_; /** - taille */
00182     zr = zr_; /** - Rayleigh length */
00183     Ph0 = Ph0_; /** - beam center */
00184     PhA = PhA_; /** - beam length */
00185     A1 = Amp * cos(phip); /** amplitude in z-direction
00186     A2 = Amp * sin(phip); /** amplitude on xy-plane
00187     lambda = numbers::pi * w0 * w0 / zr; /** formula for wavelength
00188 }
00189 // #pragma omp declare simd uniform(x,y,z) linear(pTo6Space:6)
00190 void Gauss2D::addToSpace(sunrealtype x, sunrealtype y, sunrealtype z,
00191                          sunrealtype *pTo6Space) const {
00192     /** \f$ \vec{x} = \vec{x}_0 - \vec{dis} \f$ // coordinates minus distance to
00193     // origin
00194     x -= dis[0];
00195     y -= dis[1];
00196     // z -= dis[2];
00197     z = NAN;
00198     /** \f$ z_g = \vec{x} \cdot \vec{e}_g \f$ projection on propagation axis
00199     const sunrealtype zg =
00200         x * axis[0] + y * axis[1]; // +z*axis[2]; // =z-z0 -> propagation
00201         // direction, minus origin
00202     /** \f$ r = \sqrt{\vec{x}^2 - z_g^2} \f$ -> pythagoras of radius minus
00203     // projection on prop axis
00204     const sunrealtype r = sqrt((x * x + y * y + **z**z) -
00205                                zg * zg); /** radial distance to propagation axis

```



```

00206 // \f$ w(z) = w0\sqrt{1+(z_g/z_R)^2} \f$
00207 const sunrealtype wz = w0 * sqrt(1 + (zg * zg / zr / zr)); // waist at position z
00208 // \f$ g(z) = atan(z_g/z_r) \f$
00209 const sunrealtype gz = atan(zg / zr); // Gouy phase
00210 // \f$ R(z) = z_g*(1+(z_r/z_g)^2) \f$
00211 sunrealtype Rz = NAN; // beam curvature
00212 if (zg != 0)
00213     Rz = zg * (1 + (zr * zr / zg / zg));
00214 else
00215     Rz = 1e308;
00216 // wavenumber \f$ k = 2\pi/\lambda \f$
00217 const sunrealtype k = 2 * numbers::pi / lambda;
00218 // \f$ \Phi_F = kr^2/(2R(z))+g(z)-kz_g \f$
00219 const sunrealtype PhF =
00220     -k * r * r / (2 * Rz) + gz - k * zg; // to be inserted into cosine
00221 // \f$ G = \sqrt{w_0/w_z} e^{-(r/w(z))^2} e^{(zg-Ph0)^2/PhA^2} \cos(PhF) \f$
00222 // CNode is a diva, no chance to remove the square in the second exponential
00223 // -> h too small
00224 const sunrealtype G2D = sqrt(w0 / wz) * exp(-r * r / wz / wz) *
00225     exp(-(zg - Ph0) * (zg - Ph0) / PhA / PhA) *
00226     cos(PhF); // gauss shape
00227 // \f$ c_\alpha = \vec{e}_x \cdot \vec{axis} \f$
00228 // projection components; do like this for CNode convergence -> otherwise
00229 // results in machine error values for non-existent field components if
00230 // axis[0] and axis[1] are given
00231 const sunrealtype ca =
00232     axis[0]; // x-component of propagation axis which is given as parameter
00233 const sunrealtype sa = sqrt(1 - ca * ca); // no z-component for 2D propagation
00234 // E-field to space: polarization in xy-plane (A2) is projection of
00235 // z-polarization (A1) on x- and y-directions
00236 pTo6Space[0] += sa * (G2D * A2);
00237 pTo6Space[1] += -ca * (G2D * A2);
00238 pTo6Space[2] += G2D * A1;
00239 // B-field -> negative derivative wrt polarization shift of E-field
00240 pTo6Space[3] += -sa * (G2D * A1);
00241 pTo6Space[4] += ca * (G2D * A1);
00242 pTo6Space[5] += G2D * A2;
00243 }
00244
00245 /** Gauss3D construction with */
00246 Gauss3D::Gauss3D(vector<sunrealtype> dis_, vector<sunrealtype> axis_,
00247     sunrealtype Amp_,
00248     // vector<sunrealtype> pol_,
00249     sunrealtype phip_, sunrealtype w0_, sunrealtype zr_,
00250     sunrealtype Ph0_, sunrealtype PhA_) {
00251     dis = dis_; /** - center it approaches */
00252     axis = axis_; /** - direction from where it comes */
00253     Amp = Amp_; /** - amplitude */
00254     // pol=pol_;
00255     phip = phip_; /** - polarization rotation form TE-mode */
00256     w0 = w0_; /** - taille */
00257     zr = zr_; /** - Rayleigh length */
00258     Ph0 = Ph0_; /** - beam center */
00259     PhA = PhA_; /** - beam length */
00260     lambda = numbers::pi * w0 * w0 / zr;
00261     A1 = Amp * cos(phip);
00262     A2 = Amp * sin(phip);
00263 }
00264
00265 /** Gauss3D implementation in space */
00266 void Gauss3D::addToSpace(sunrealtype x, sunrealtype y, sunrealtype z,
00267     sunrealtype *pTo6Space) const {
00268     x -= dis[0];
00269     y -= dis[1];
00270     z -= dis[2];
00271     const sunrealtype zg = x * axis[0] + y * axis[1] + z * axis[2];
00272     const sunrealtype r = sqrt((x * x + y * y + z * z) - zg * zg);
00273     const sunrealtype wz = w0 * sqrt(1 + (zg * zg / zr / zr));
00274     const sunrealtype gz = atan(zg / zr);
00275     sunrealtype Rz = NAN;
00276     if (zg != 0)
00277         Rz = zg * (1 + (zr * zr / zg / zg));
00278     else
00279         Rz = 1e308;
00280     const sunrealtype k = 2 * numbers::pi / lambda;
00281     const sunrealtype PhF = -k * r * r / (2 * Rz) + gz - k * zg;
00282     const sunrealtype G3D = (w0 / wz) * exp(-r * r / wz / wz) *
00283         exp(-(zg - Ph0) * (zg - Ph0) / PhA / PhA) * cos(PhF);
00284     const sunrealtype ca = axis[0];
00285     const sunrealtype sa = sqrt(1 - ca * ca);
00286     pTo6Space[0] += sa * (G3D * A2);
00287     pTo6Space[1] += -ca * (G3D * A2);
00288     pTo6Space[2] += G3D * A1;
00289     pTo6Space[3] += -sa * (G3D * A1);
00290     pTo6Space[4] += ca * (G3D * A1);
00291     pTo6Space[5] += G3D * A2;
00292 }

```

```

00293
00294 /** Evaluate lattice point values to zero and add field values */
00295 // # pragma omp declare simd uniform(x,y,z) linear(pTo6Space:6)
00296 void ICSetter::eval(sunrealtype x, unrealtype y, unrealtype z,
00297                    unrealtype *pTo6Space) {
00298     pTo6Space[0] = 0;
00299     pTo6Space[1] = 0;
00300     pTo6Space[2] = 0;
00301     pTo6Space[3] = 0;
00302     pTo6Space[4] = 0;
00303     pTo6Space[5] = 0;
00304     add(x, y, z, pTo6Space);
00305 }
00306
00307 /** Add all initial field values to the lattice space */
00308 void ICSetter::add(sunrealtype x, unrealtype y, unrealtype z,
00309                  unrealtype *pTo6Space) {
00310     for (const auto &wave : planeWaves1D)
00311         wave.addToSpace(x, y, z, pTo6Space);
00312     for (const auto &wave : planeWaves2D)
00313         wave.addToSpace(x, y, z, pTo6Space);
00314     for (const auto &wave : planeWaves3D)
00315         wave.addToSpace(x, y, z, pTo6Space);
00316     for (const auto &wave : gauss1Ds)
00317         wave.addToSpace(x, y, z, pTo6Space);
00318     for (const auto &wave : gauss2Ds)
00319         wave.addToSpace(x, y, z, pTo6Space);
00320     for (const auto &wave : gauss3Ds)
00321         wave.addToSpace(x, y, z, pTo6Space);
00322 }
00323
00324 /** Add plane waves in 1D to their container vector */
00325 void ICSetter::addPlaneWave1D(vector<unrealtype> k, vector<unrealtype> p,
00326                              vector<unrealtype> phi) {
00327     planeWaves1D.emplace_back(PlaneWave1D(k, p, phi));
00328 }
00329
00330 /** Add plane waves in 2D to their container vector */
00331 void ICSetter::addPlaneWave2D(vector<unrealtype> k, vector<unrealtype> p,
00332                              vector<unrealtype> phi) {
00333     planeWaves2D.emplace_back(PlaneWave2D(k, p, phi));
00334 }
00335
00336 /** Add plane waves in 3D to their container vector */
00337 void ICSetter::addPlaneWave3D(vector<unrealtype> k, vector<unrealtype> p,
00338                              vector<unrealtype> phi) {
00339     planeWaves3D.emplace_back(PlaneWave3D(k, p, phi));
00340 }
00341
00342 /** Add Gaussian waves in 1D to their container vector */
00343 void ICSetter::addGauss1D(vector<unrealtype> k, vector<unrealtype> p,
00344                          vector<unrealtype> xo, unrealtype phig_,
00345                          vector<unrealtype> phi) {
00346     gauss1Ds.emplace_back(Gauss1D(k, p, xo, phig_, phi));
00347 }
00348
00349 /** Add Gaussian waves in 2D to their container vector */
00350 void ICSetter::addGauss2D(vector<unrealtype> dis_, vector<unrealtype> axis_,
00351                          unrealtype Amp_, unrealtype phip_, unrealtype w0_,
00352                          unrealtype zr_, unrealtype Ph0_, unrealtype PhA_) {
00353     gauss2Ds.emplace_back(
00354         Gauss2D(dis_, axis_, Amp_, phip_, w0_, zr_, Ph0_, PhA_));
00355 }
00356
00357 /** Add Gaussian waves in 3D to their container vector */
00358 void ICSetter::addGauss3D(vector<unrealtype> dis_, vector<unrealtype> axis_,
00359                          unrealtype Amp_, unrealtype phip_, unrealtype w0_,
00360                          unrealtype zr_, unrealtype Ph0_, unrealtype PhA_) {
00361     gauss3Ds.emplace_back(
00362         Gauss3D(dis_, axis_, Amp_, phip_, w0_, zr_, Ph0_, PhA_));
00363 }

```

6.8 src/ICSetters.h File Reference

Declaration of the plane wave and Gaussian wave packets in 1D, 2D, 3D.

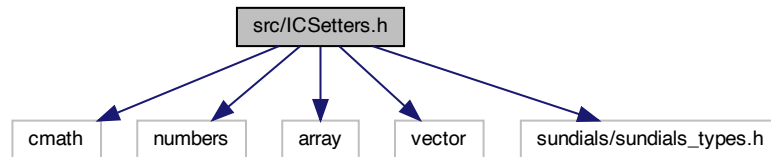
```

#include <cmath>
#include <numbers>
#include <array>

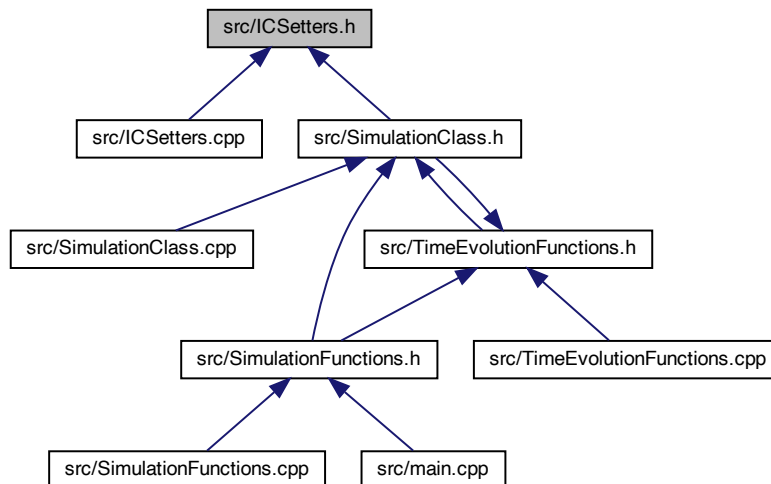
```

```
#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 #pragma once
00007
00008 // math, constants, vector, and array
00009 #include <cmath>
00010 // #include <mathimf.h>
00011 #include <numbers>
00012 #include <array>
00013 #include <vector>
00014
00015 #include <sundials/sundials_types.h> /* definition of type sunrealtype */
00016
00017 using namespace std;
00018
00019 /** @brief super-class for plane waves
00020  *
00021  * They are given in the form  $\vec{E} = \vec{E}_0 \cos(\vec{k} \cdot \vec{x} - \vec{\phi})$ 
00022  *  $\vec{\phi}$ 
00023  */
00024 class PlaneWave {
00025 protected:
00026     /// wavenumber  $k_x$ 
00027     sunrealtype kx;
00028     /// wavenumber  $k_y$ 
00029     sunrealtype ky;
00030     /// wavenumber  $k_z$ 
00031     sunrealtype kz;
00032     /// polarization & amplitude in x-direction,  $p_x$ 
00033     sunrealtype px;
00034     /// polarization & amplitude in y-direction,  $p_y$ 
00035     sunrealtype py;
00036     /// polarization & amplitude in z-direction,  $p_z$ 
00037     sunrealtype pz;
00038     /// phase shift in x-direction,  $\phi_x$ 
00039     sunrealtype phix;
00040     /// phase shift in y-direction,  $\phi_y$ 
00041     sunrealtype phiy;
00042     /// phase shift in z-direction,  $\phi_z$ 
00043     sunrealtype phiz;
00044 };
00045
00046 /** @brief class for plane waves in 1D */
00047 class PlaneWave1D : public PlaneWave {
00048 public:
00049     /// construction with default parameters
00050     PlaneWave1D(vector<sunrealtype> k = {1, 0, 0},
00051                vector<sunrealtype> p = {0, 0, 1},
00052                vector<sunrealtype> phi = {0, 0, 0});
00053     /// function for the actual implementation in the lattice
00054     void addToSpace(sunrealtype x, sunrealtype y, sunrealtype z,
00055                   sunrealtype *pTo6Space) const;
00056 };
00057
00058 /** @brief class for plane waves in 2D */
00059 class PlaneWave2D : public PlaneWave {
00060 public:

```

```

00060    /// construction with default parameters
00061    PlaneWave2D(vector<sunrealtype> k = {1, 0, 0},
00062               vector<sunrealtype> p = {0, 0, 1},
00063               vector<sunrealtype> phi = {0, 0, 0});
00064    /// function for the actual implementation in the lattice
00065    void addToSpace(sunrealtype x, sunrealtype y, sunrealtype z,
00066                   sunrealtype *pTo6Space) const;
00067 };
00068
00069 /** @brief class for plane waves in 3D */
00070 class PlaneWave3D : public PlaneWave {
00071 public:
00072    /// construction with default parameters
00073    PlaneWave3D(vector<sunrealtype> k = {1, 0, 0},
00074               vector<sunrealtype> p = {0, 0, 1},
00075               vector<sunrealtype> phi = {0, 0, 0});
00076    /// function for the actual implementation in space
00077    void addToSpace(sunrealtype x, sunrealtype y, sunrealtype z,
00078                   sunrealtype *pTo6Space) const;
00079 };
00080
00081 /** @brief class for Gaussian waves in 1D
00082  *
00083  * They are given in the form  $\vec{E} = \vec{p} \cdot \exp\left(-\frac{(\vec{x} - \vec{x}_0)^2}{\Phi_g^2}\right) \cos(\vec{k} \cdot \vec{x})$ 
00084  */
00085 */
00086 class Gauss1D {
00087 private:
00088    /// wavenumber  $k_x$ 
00089    sunrealtype kx;
00090    /// wavenumber  $k_y$ 
00091    sunrealtype ky;
00092    /// wavenumber  $k_z$ 
00093    sunrealtype kz;
00094    /// polarization & amplitude in x-direction,  $p_x$ 
00095    sunrealtype px;
00096    /// polarization & amplitude in y-direction,  $p_y$ 
00097    sunrealtype py;
00098    /// polarization & amplitude in z-direction,  $p_z$ 
00099    sunrealtype pz;
00100    /// phase shift in x-direction,  $\phi_x$ 
00101    sunrealtype phix;
00102    /// phase shift in y-direction,  $\phi_y$ 
00103    sunrealtype phiy;
00104    /// phase shift in z-direction,  $\phi_z$ 
00105    sunrealtype phiz;
00106    /// center of pulse in x-direction,  $x_0$ 
00107    sunrealtype x0x;
00108    /// center of pulse in y-direction,  $y_0$ 
00109    sunrealtype x0y;
00110    /// center of pulse in z-direction,  $z_0$ 
00111    sunrealtype x0z;
00112    /// pulse width  $\Phi_g$ 
00113    sunrealtype phig;
00114
00115 public:
00116    /// construction with default parameters
00117    Gauss1D(vector<sunrealtype> k = {1, 0, 0}, vector<sunrealtype> p = {0, 0, 1},
00118            sunrealtype x0 = {0, 0, 0}, sunrealtype phig_ = 1.01,
00119            vector<sunrealtype> phi = {0, 0, 0});
00120    /// function for the actual implementation in space
00121    void addToSpace(sunrealtype x, sunrealtype y, sunrealtype z,
00122                   sunrealtype *pTo6Space) const;
00123
00124 public:
00125 };
00126
00127 /** @brief class for Gaussian waves in 2D
00128  *
00129  * They are given in the form
00130  *  $E = A \cdot \sqrt{\frac{\omega_0}{\omega(z)}} \cdot \exp\left(-\frac{(z_g - \Phi_0)^2}{\Phi_A^2}\right) \cdot \cos\left(\frac{k}{r^2} R(z) + g(z) - k \cdot z_g\right)$  with
00131  * -  $\vec{k}$  - propagation direction (subtracted distance to origin)  $z_g$ 
00132  * -  $r$  - radial distance to propagation axis  $r = \sqrt{x^2 + z_g^2}$ 
00133  * -  $k = 2\pi / \lambda$ 
00134  * -  $w_0$  - waist at position  $z$ ,  $\omega(z) = w_0 \cdot \sqrt{1 + (z_g/z_R)^2}$ 
00135  * - Gouy phase  $g(z) = \tan^{-1}(z_g/z_R)$ 
00136  * - beam curvature  $R(z) = z_g \cdot (1 + (z_r/z_g)^2)$ 
00137  * obtained via the chosen parameters */
00138 */
00139
00140 class Gauss2D {
00141 private:
00142    /// distance maximum to origin
00143    vector<sunrealtype> dis;
00144    /// normalized propagation axis
00145    vector<sunrealtype> axis;
00146    /// amplitude  $A$ 

```

```

00147     sunrealtype Amp;
00148     /// polarization rotation from TE-mode around propagation direction
00149     // that determines \f$ \vec{\epsilon}\f$ above
00150     sunrealtype phip;
00151     /// taille \f$ \omega_0 \f$
00152     sunrealtype w0;
00153     /// Rayleigh length \f$ z_R = \pi \omega_0^2 / \lambda \f$
00154     sunrealtype zr;
00155     /// center of beam \f$ \Phi_0 \f$
00156     sunrealtype Ph0;
00157     /// length of beam \f$ \Phi_A \f$
00158     sunrealtype PhA;
00159     /// amplitude projection on TE-mode
00160     sunrealtype A1;
00161     /// amplitude projection on xy-plane
00162     sunrealtype A2;
00163     /// wavelength \f$ \lambda \f$
00164     sunrealtype lambda;
00165
00166 public:
00167     /// construction with default parameters
00168     Gauss2D(vector<sunrealtype> dis_ = {0, 0, 0},
00169             vector<sunrealtype> axis_ = {1, 0, 0}, sunrealtype Amp_ = 1.01,
00170             sunrealtype phip_ = 0, sunrealtype w0_ = 1e-5, sunrealtype zr_ = 4e-5,
00171             sunrealtype Ph0_ = 2e-5, sunrealtype PhA_ = 0.45e-5);
00172     /// function for the actual implementation in space
00173     void addToSpace(sunrealtype x, sunrealtype y, sunrealtype z,
00174                    sunrealtype *pTo6Space) const;
00175
00176 public:
00177 };
00178
00179 /** @brief class for Gaussian waves in 3D
00180  *
00181  * They are given in the form
00182  * \f$ \vec{E} = A \, \vec{\epsilon} \, \frac{\omega_0}{\omega(z)} \, \exp
00183  * \left( -r/\omega(z) \right)^2 \, \exp \left( -((z_g - \Phi_0)/\Phi_A)^2 \right)
00184  * \, \cos \left( \frac{k}{r^2} R(z) + g(z) - k \, z_g \right) \f$ with
00185  * - propagation direction (subtracted distance to origin) \f$ z_g \f$
00186  * - radial distance to propagation axis \f$ r = \sqrt{\vec{x}^2 - z_g^2} \f$
00187  * - \f$ k = 2\pi / \lambda \f$
00188  * - waist at position z, \f$ \omega(z) = w_0 \, \sqrt{1 + (z_g/z_R)^2} \f$
00189  * - Gouy phase \f$ g(z) = \tan^{-1}(z_g/z_R) \f$
00190  * - beam curvature \f$ R(z) = z_g \, (1 + (z_r/z_g)^2) \f$
00191  * obtained via the chosen parameters */
00192 class Gauss3D {
00193 private:
00194     /// distance maximum to origin
00195     vector<sunrealtype> dis;
00196     /// normalized propagation axis
00197     vector<sunrealtype> axis;
00198     /// amplitude \f$ A \f$
00199     sunrealtype Amp;
00200     /// polarization rotation from TE-mode around propagation direction
00201     // that determines \f$ \vec{\epsilon}\f$ above
00202     sunrealtype phip;
00203     // polarization
00204     // vector<sunrealtype> pol;
00205     /// taille \f$ \omega_0 \f$
00206     sunrealtype w0;
00207     /// Rayleigh length \f$ z_R = \pi \omega_0^2 / \lambda \f$
00208     sunrealtype zr;
00209     /// center of beam \f$ \Phi_0 \f$
00210     sunrealtype Ph0;
00211     /// length of beam \f$ \Phi_A \f$
00212     sunrealtype PhA;
00213     /// amplitude projection on TE-mode (z-axis)
00214     sunrealtype A1;
00215     /// amplitude projection on xy-plane
00216     sunrealtype A2;
00217     /// wavelength \f$ \lambda \f$
00218     sunrealtype lambda;
00219
00220 public:
00221     /// construction with default parameters
00222     Gauss3D(vector<sunrealtype> dis_ = {0, 0, 0},
00223             vector<sunrealtype> axis_ = {1, 0, 0}, sunrealtype Amp_ = 1.01,
00224             sunrealtype phip_ = 0,
00225             // sunrealtype pol_={0,0,1},
00226             sunrealtype w0_ = 1e-5, sunrealtype zr_ = 4e-5,
00227             sunrealtype Ph0_ = 2e-5, sunrealtype PhA_ = 0.45e-5);
00228     /// function for the actual implementation in space
00229     void addToSpace(sunrealtype x, sunrealtype y, sunrealtype z,
00230                    sunrealtype *pTo6Space) const;
00231
00232 public:
00233 };

```

```

00234
00235 /** @brief ICSetter class to initialize wave types with default parameters */
00236 class ICSetter {
00237 private:
00238     /// container vector for plane waves in 1D
00239     vector<PlaneWave1D> planeWaves1D;
00240     /// container vector for plane waves in 2D
00241     vector<PlaneWave2D> planeWaves2D;
00242     /// container vector for plane waves in 3D
00243     vector<PlaneWave3D> planeWaves3D;
00244     /// container vector for Gaussian waves in 1D
00245     vector<Gauss1D> gauss1Ds;
00246     /// container vector for Gaussian waves in 2D
00247     vector<Gauss2D> gauss2Ds;
00248     /// container vector for Gaussian waves in 3D
00249     vector<Gauss3D> gauss3Ds;
00250
00251 public:
00252     /// function to set all coordinates to zero and then 'add' the field values
00253     void eval(sunrealtype x, unrealtype y, unrealtype z,
00254             unrealtype *pTo6Space);
00255     /// function to fill the lattice space with initial field values
00256     /// of all field vector containers
00257     void add(sunrealtype x, unrealtype y, unrealtype z, unrealtype *pTo6Space);
00258     /// function to add plane waves in 1D to their container vector
00259     void addPlaneWave1D(vector<unrealtype> k = {1, 0, 0},
00260                        vector<unrealtype> p = {0, 0, 1},
00261                        vector<unrealtype> phi = {0, 0, 0});
00262     /// function to add plane waves in 2D to their container vector
00263     void addPlaneWave2D(vector<unrealtype> k = {1, 0, 0},
00264                        vector<unrealtype> p = {0, 0, 1},
00265                        vector<unrealtype> phi = {0, 0, 0});
00266     /// function to add plane waves in 3D to their container vector
00267     void addPlaneWave3D(vector<unrealtype> k = {1, 0, 0},
00268                        vector<unrealtype> p = {0, 0, 1},
00269                        vector<unrealtype> phi = {0, 0, 0});
00270     /// function to add Gaussian waves in 1D to their container vector
00271     void addGauss1D(vector<unrealtype> k = {1, 0, 0},
00272                    vector<unrealtype> p = {0, 0, 1},
00273                    vector<unrealtype> xo = {0, 0, 0}, unrealtype phig_ = 1.01,
00274                    vector<unrealtype> phi = {0, 0, 0});
00275     /// function to add Gaussian waves in 2D to their container vector
00276     void addGauss2D(vector<unrealtype> dis_ = {0, 0, 0},
00277                    vector<unrealtype> axis_ = {1, 0, 0},
00278                    unrealtype Amp_ = 1.01, unrealtype phip_ = 0,
00279                    unrealtype w0_ = 1e-5, unrealtype zr_ = 4e-5,
00280                    unrealtype Ph0_ = 2e-5, unrealtype PhA_ = 0.45e-5);
00281     /// function to add Gaussian waves in 3D to their container vector
00282     void addGauss3D(vector<unrealtype> dis_ = {0, 0, 0},
00283                    vector<unrealtype> axis_ = {1, 0, 0},
00284                    unrealtype Amp_ = 1.01, unrealtype phip_ = 0,
00285                    unrealtype w0_ = 1e-5, unrealtype zr_ = 4e-5,
00286                    unrealtype Ph0_ = 2e-5, unrealtype PhA_ = 0.45e-5);
00287 };
00288

```

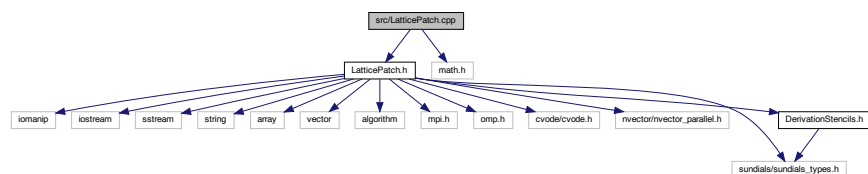
6.10 src/LatticePatch.cpp File Reference

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 )
```

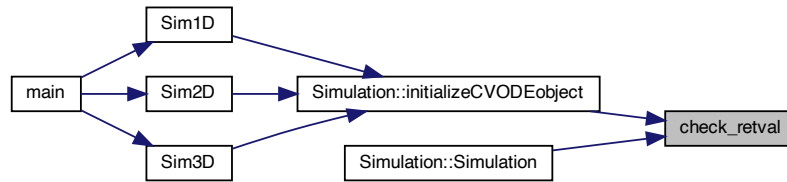
Check function return value. From CCode examples. opt == 0 means SUNDIALS function allocates memory so check if returned NULL pointer opt == 1 means SUNDIALS function returns an integer value so check if retval < 0 opt == 2 means function allocates memory so check if returned NULL pointer

Definition at line 982 of file [LatticePatch.cpp](#).

```
00982                                     {
00983     int *retval = nullptr;
00984
00985     /* Check if SUNDIALS function returned NULL pointer - no memory allocated */
00986     if (opt == 0 && returnvalue == nullptr) {
00987         fprintf(stderr,
00988             "\nSUNDIALS_ERROR(%d): %s() failed - returned NULL pointer\n\n", id,
00989             funcname);
00990         return (1);
00991     }
00992
00993     /* Check if retval < 0 */
00994     else if (opt == 1) {
00995         retval = (int *)returnvalue;
00996         if (*retval < 0) {
00997             fprintf(stderr, "\nSUNDIALS_ERROR(%d): %s() failed with retval = %d\n\n",
00998                 id, funcname, *retval);
00999             return (1);
01000         }
01001     }
01002
01003     /* Check if function returned NULL pointer - no memory allocated */
01004     else if (opt == 2 && returnvalue == nullptr) {
01005         fprintf(stderr,
01006             "\nMEMORY_ERROR(%d): %s() failed - returned NULL pointer\n\n", id,
01007             funcname);
01008         return (1);
01009     }
01010
01011     return (0);
01012 }
```


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 )
```

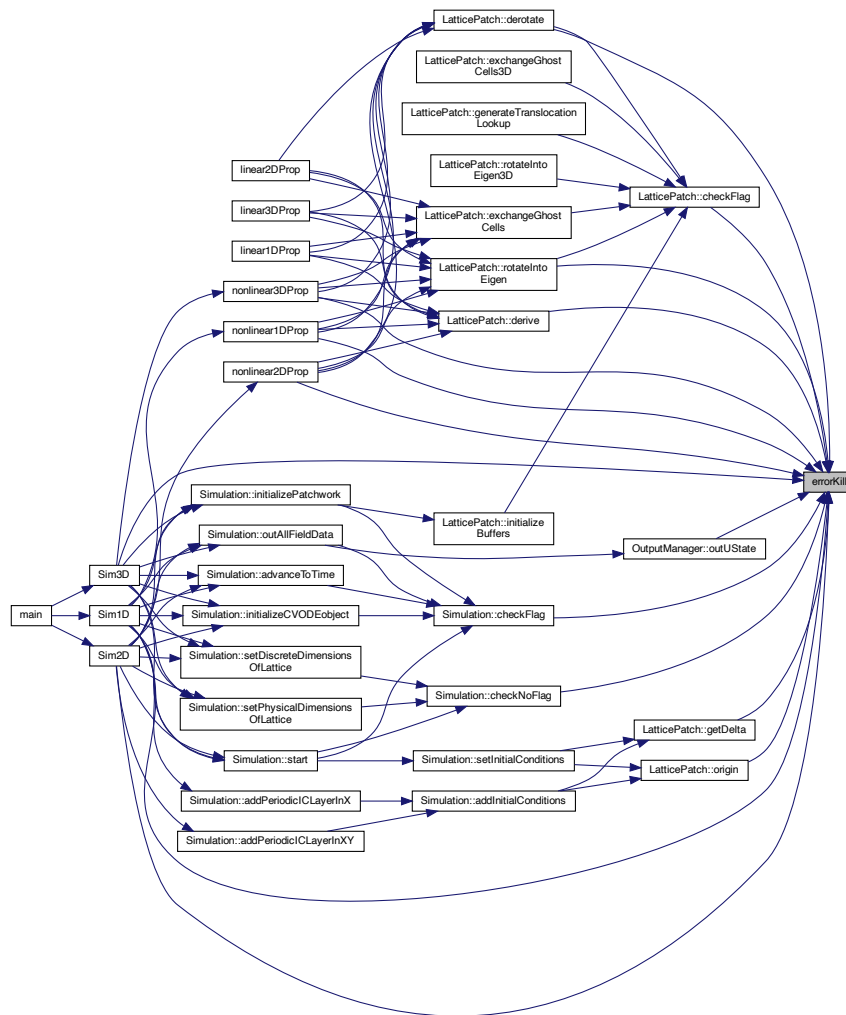
Print a specific error message to stdout.

Definition at line 969 of file [LatticePatch.cpp](#).

```
00969 {
00970     cerr << endl << "Error: " << errorMessage << " Aborting..." << endl;
00971     MPI_Abort(MPI_COMM_WORLD, 1);
00972     return;
00973 }
```

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

Here is the caller graph for this function:



6.10.2.3 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
```

```
{
```

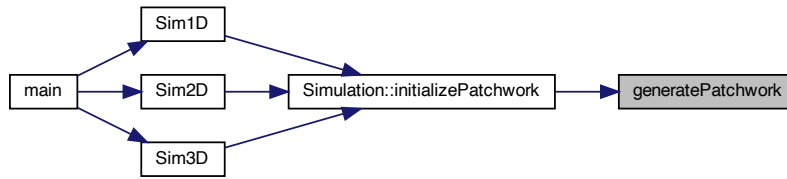
```

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 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     /*
00140     // Experiment: Resolution can be adapted for each process/patch
00141     const int Scaler=4; // Better hand over as parameter via 'initializePatchwork'
00142     if(my_prc==0){patchToMold.dx=envelopeLattice.get_dx();}
00143     else if(my_prc==1){patchToMold.dx=envelopeLattice.get_dx()*Scaler;}
00144     else{errorKill("Only do this resolution barrier test with 2 processes.");}
00145     */
00146     patchToMold.dx = envelopeLattice.get_dx();
00147     patchToMold.dy = envelopeLattice.get_dy();
00148     patchToMold.dz = envelopeLattice.get_dz();
00149     patchToMold.x0 = firstXPoint * patchToMold.dx;
00150     patchToMold.y0 = firstYPoint * patchToMold.dy;
00151     patchToMold.z0 = firstZPoint * patchToMold.dz;
00152     patchToMold.LIx = LIx;
00153     patchToMold.LIy = LIy;
00154     patchToMold.LIz = LIz;
00155     patchToMold.nx = local_NOXP;
00156     patchToMold.ny = local_NOYP;
00157     patchToMold.nz = local_NOZP;
00158     patchToMold.lx = patchToMold.nx * patchToMold.dx;
00159     patchToMold.ly = patchToMold.ny * patchToMold.dy;
00160     patchToMold.lz = patchToMold.nz * patchToMold.dz;
00161     /* // Check name of lattice communicator
00162     char lattice_comm_name[MPI_MAX_OBJECT_NAME];
00163     int lattice_namelen;
00164     MPI_Comm_get_name(envelopeLattice.comm, lattice_comm_name, &lattice_namelen);
00165     cout<<"envelopeLattice.comm gives " << lattice_comm_name << endl;
00166     */
00167     /* Create and allocate memory for parallel vectors with defined local and
00168     * global lenghts *
00169     * (-> CNode problem sizes Nlocal and N)
00170     * for field data and temporal derivatives and set extra pointers to them */
00171     patchToMold.u =
00172         N_VNew_Parallel(envelopeLattice.comm, local_NODP,
00173             envelopeLattice.get_tot_nodp(), envelopeLattice.sunctx);
00174     patchToMold.uData = NV_DATA_P(patchToMold.u);
00175     patchToMold.du =
00176         N_VNew_Parallel(envelopeLattice.comm, local_NODP,
00177             envelopeLattice.get_tot_nodp(), envelopeLattice.sunctx);
00178     patchToMold.duData = NV_DATA_P(patchToMold.du);
00179     // Allocate space for auxiliary uAux so that the lattice and all possible
00180     // directions of ghost layers fit
00181     const int s1 = patchToMold.nx, s2 = patchToMold.ny, s3 = patchToMold.nz;
00182     const int s_min = min(s1, min(s2, s3));
00183     patchToMold.uAux.resize(s1 * s2 * s3 / s_min * (s_min + 2 * gLW) * dPD);
00184     patchToMold.uAuxData = &patchToMold.uAux[0];
00185     patchToMold.envelopeLattice = &envelopeLattice;
00186     // Set patch "name" to process number -> only for debugging
00187     // patchToMold.ID=my_prc;
00188     // set flag
00189     patchToMold.statusFlags = FLatticePatchSetUp;
00190     patchToMold.generateTranslocationLookup();
00191     return 0;
00192 }

```

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

Here is the caller graph for this function:



6.11 LatticePatch.cpp

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

```

00001 //////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
00002 /// @file LatticePatch.cpp
00003 /// @brief Construction of the overall envelope lattice and the lattice patches
00004 //////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
00005
00006 #include "LatticePatch.h"
00007
00008 #include <math.h>
00009
00010 //////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
00011 /// Implementation of Lattice component functions ///
00012 //////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
00013
00014 /// Initialize the cartesian communicator
00015 void Lattice::initializeCommunicator(const int nx, const int ny,
00016                                     const int nz, const bool per) {
00017     const int dims[3] = {nz, ny, nx};
00018     const int periods[3] = {static_cast<int>(per), static_cast<int>(per),
00019                             static_cast<int>(per)};
00020     // Create the cartesian communicator for MPI_COMM_WORLD
00021     MPI_Cart_create(MPI_COMM_WORLD, 3, dims, periods, 1, &comm);
00022     // Set MPI variables of the lattice
00023     MPI_Comm_size(comm, &(n_prc));
00024     MPI_Comm_rank(comm, &(my_prc));
00025     // Associate name to the communicator to identify it -> for debugging and
00026     // nicer error messages
00027     constexpr char lattice_comm_name[] = "Lattice";
00028     MPI_Comm_set_name(comm, lattice_comm_name);
00029
00030     // Test if process naming is the same for both communicators
00031     /*
00032     int MYPRC;
00033     MPI_Comm_rank(MPI_COMM_WORLD, &MYPRC);
00034     cout<<"\r"<<my_prc<<"\t"<<MYPRC<<endl;
00035     */
00036 }
00037
00038 /// Construct the lattice and set the stencil order
00039 Lattice::Lattice(const int St0) : stencilOrder(St0),
00040     ghostLayerWidth(St0/2+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     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 }
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     /*
00140     // Experiment: Resolution can be adapted for each process/patch
00141     const int Scaler=4; // Better hand over as parameter via 'initializePatchwork'
00142     if(my_prc==0){patchToMold.dx=envelopeLattice.get_dx();}
00143     else if(my_prc==1){patchToMold.dx=envelopeLattice.get_dx()*Scaler;}
00144     else{errorKill("Only do this resolution barrier test with 2 processes.");}
00145     */

```

```

00146 patchToMold.dx = envelopeLattice.get_dx();
00147 patchToMold.dy = envelopeLattice.get_dy();
00148 patchToMold.dz = envelopeLattice.get_dz();
00149 patchToMold.x0 = firstXPoint * patchToMold.dx;
00150 patchToMold.y0 = firstYPoint * patchToMold.dy;
00151 patchToMold.z0 = firstZPoint * patchToMold.dz;
00152 patchToMold.LIx = LIx;
00153 patchToMold.LIy = LIy;
00154 patchToMold.LIz = LIz;
00155 patchToMold.nx = local_NOXP;
00156 patchToMold.ny = local_NOYP;
00157 patchToMold.nz = local_NOZP;
00158 patchToMold.lx = patchToMold.nx * patchToMold.dx;
00159 patchToMold.ly = patchToMold.ny * patchToMold.dy;
00160 patchToMold.lz = patchToMold.nz * patchToMold.dz;
00161 /* // Check name of lattice communicator
00162 char lattice_comm_name[MPI_MAX_OBJECT_NAME];
00163 int lattice_namelen;
00164 MPI_Comm_get_name(envelopeLattice.comm, lattice_comm_name, &lattice_namelen);
00165 cout<<"envelopeLattice.comm gives " << lattice_comm_name << endl;
00166 */
00167 /* Create and allocate memory for parallel vectors with defined local and
00168 * global lengths *
00169 * (-> CNode problem sizes Nlocal and N)
00170 * for field data and temporal derivatives and set extra pointers to them */
00171 patchToMold.u =
00172     N_VNew_Parallel(envelopeLattice.comm, local_NODP,
00173                     envelopeLattice.get_tot_nodp(), envelopeLattice.sunctx);
00174 patchToMold.uData = NV_DATA_P(patchToMold.u);
00175 patchToMold.du =
00176     N_VNew_Parallel(envelopeLattice.comm, local_NODP,
00177                     envelopeLattice.get_tot_nodp(), envelopeLattice.sunctx);
00178 patchToMold.duData = NV_DATA_P(patchToMold.du);
00179 // Allocate space for auxiliary uAux so that the lattice and all possible
00180 // directions of ghost layers fit
00181 const int s1 = patchToMold.nx, s2 = patchToMold.ny, s3 = patchToMold.nz;
00182 const int s_min = min(s1, min(s2, s3));
00183 patchToMold.uAux.resize(s1 * s2 * s3 / s_min * (s_min + 2 * gLW) * dPD);
00184 patchToMold.uAuxData = &patchToMold.uAux[0];
00185 patchToMold.envelopeLattice = &envelopeLattice;
00186 // Set patch "name" to process number -> only for debugging
00187 // patchToMold.ID=my_prc;
00188 // set flag
00189 patchToMold.statusFlags = FLatticePatchSetUp;
00190 patchToMold.generateTranslocationLookup();
00191 return 0;
00192 }
00193
00194 /// Return the discrete size of the patch: number of lattice patch points in
00195 /// specified dimension
00196 int LatticePatch::discreteSize(int dir) const {
00197     switch (dir) {
00198     case 0:
00199         return nx * ny * nz;
00200     case 1:
00201         return nx;
00202     case 2:
00203         return ny;
00204     case 3:
00205         return nz;
00206     // case 4: return uAux.size(); // for debugging
00207     default:
00208         return -1;
00209     }
00210 }
00211
00212 /// Return the physical origin of the patch in a dimension
00213 sunrealtype LatticePatch::origin(const int dir) const {
00214     switch (dir) {
00215     case 1:
00216         return x0;
00217     case 2:
00218         return y0;
00219     case 3:
00220         return z0;
00221     default:
00222         errorKill("LatticePatch::origin function called with wrong dir parameter");
00223         return -1;
00224     }
00225 }
00226
00227 /// Return the distance between points in the patch in a dimension
00228 sunrealtype LatticePatch::getDelta(const int dir) const {
00229     switch (dir) {
00230     case 1:
00231         return dx;
00232     case 2:

```

```

00233     return dy;
00234 case 3:
00235     return dz;
00236 default:
00237     errorKill(
00238         "LatticePatch::getDelta function called with wrong dir parameter");
00239     return -1;
00240 }
00241 }
00242
00243 /** To avoid cache misses:
00244  * create vectors to translate u vector into space coordinates and vice versa
00245  * and same for left and right ghost layers to space */
00246 void LatticePatch::generateTranslocationLookup() {
00247     // Check that the lattice has been set up
00248     checkFlag(FLatticeDimensionSet);
00249     // lengths for auxilliary layers, including ghost layers
00250     const int gLW = envelopeLattice->get_ghostLayerWidth();
00251     const int mx = nx + 2 * gLW;
00252     const int my = ny + 2 * gLW;
00253     const int mz = nz + 2 * gLW;
00254     // sizes for lookup vectors
00255     // generate u->uAux
00256     uTox.resize(nx * ny * nz);
00257     uToy.resize(nx * ny * nz);
00258     uToz.resize(nx * ny * nz);
00259     // generate uAux->u with length including halo
00260     xTou.resize(mx * ny * nz);
00261     yTou.resize(nx * my * nz);
00262     zTou.resize(nx * ny * mz);
00263     // variables for cartesian position in the 3D discrete lattice
00264     int px = 0, py = 0, pz = 0;
00265     for (unsigned int i = 0; i < uToy.size(); i++) { // loop over all points in the patch
00266         // calculate cartesian coordinates
00267         px = i % nx;
00268         py = (i / nx) % ny;
00269         pz = (i / nx) / ny;
00270         // fill lookups extended by halos (useful for y and z direction)
00271         uTox[i] = (px + gLW) + py * mx +
00272             pz * mx * ny; // unroll (de-flatten) cartesian dimension
00273         xTou[px + py * mx + pz * mx * ny] =
00274             i; // match cartesian point to u location
00275         uToy[i] = (py + gLW) + pz * my + px * my * nz;
00276         yTou[py + pz * my + px * my * nz] = i;
00277         uToz[i] = (pz + gLW) + px * mz + py * mz * nx;
00278         zTou[pz + px * mz + py * mz * nx] = i;
00279     }
00280     // same for ghost layer lookup tables
00281     lgcTox.resize(gLW * ny * nz);
00282     rgcTox.resize(gLW * ny * nz);
00283     for (unsigned int i = 0; i < lgcTox.size(); i++) {
00284         px = i % gLW;
00285         py = (i / gLW) % ny;
00286         pz = (i / gLW) / ny;
00287         lgcTox[i] = px + py * mx + pz * mx * ny;
00288         rgcTox[i] = px + nx + gLW + py * mx + pz * mx * ny;
00289     }
00290     lgcToy.resize(gLW * nx * nz);
00291     rgcToy.resize(gLW * nx * nz);
00292     for (unsigned int i = 0; i < lgcToy.size(); i++) {
00293         px = i % nx;
00294         py = (i / nx) % gLW;
00295         pz = (i / nx) / gLW;
00296         lgcToy[i] = py + pz * my + px * my * nz;
00297         rgcToy[i] = py + ny + gLW + pz * my + px * my * nz;
00298     }
00299     lgcToz.resize(gLW * nx * ny);
00300     rgcToz.resize(gLW * nx * ny);
00301     for (unsigned int i = 0; i < lgcToz.size(); i++) {
00302         px = i % nx;
00303         py = (i / nx) % ny;
00304         pz = (i / nx) / ny;
00305         lgcToz[i] = pz + px * mz + py * mz * nx;
00306         rgcToz[i] = pz + nz + gLW + px * mz + py * mz * nx;
00307     }
00308     statusFlags |= TranslocationLookupSetUp;
00309 }
00310
00311 /** Rotate into eigenraum along R matrices of paper using below rotation
00312  * functions
00313  * -> uAuxData gets the rotated left-halo-, inner-patch-, right-halo-data */
00314 void LatticePatch::rotateIntoEigen(const int dir) {
00315     // Check that the lattice, ghost layers as well as the translocation lookups
00316     // have been set up;
00317     checkFlag(FLatticePatchSetUp);
00318     checkFlag(TranslocationLookupSetUp);
00319     checkFlag(GhostLayersInitialized); // this check is only after call to

```

```

00320                                     // exchange ghost cells
00321 switch (dir) {
00322 case 1:
00323     rotateToX(uAuxData, gCLData, lgcTox);
00324     rotateToX(uAuxData, uData, uTox);
00325     rotateToX(uAuxData, gCRData, rgcTox);
00326     break;
00327 case 2:
00328     rotateToY(uAuxData, gCLData, lgcToy);
00329     rotateToY(uAuxData, uData, uToy);
00330     rotateToY(uAuxData, gCRData, rgcToy);
00331     break;
00332 case 3:
00333     rotateToZ(uAuxData, gCLData, lgcToz);
00334     rotateToZ(uAuxData, uData, uToz);
00335     rotateToZ(uAuxData, gCRData, rgcToz);
00336     break;
00337 default:
00338     errorKill("Tried to rotate into the wrong direction");
00339     break;
00340 }
00341 }
00342
00343 /// Same as 'rotateIntoEigen' but for neighborhood 3D halo buffers
00344 void LatticePatch::rotateIntoEigen3D() {
00345     checkFlag(FLatticePatchSetUp);
00346     checkFlag(TranslocationLookupSetUp);
00347     checkFlag(GhostLayersInitialized);
00348     rotateToX(uAuxData, gCLData, lgcTox);
00349     rotateToX(uAuxData, uData, uTox);
00350     rotateToX(uAuxData, gCRData, rgcTox);
00351     rotateToY(uAuxData, gCBData, lgcToy);
00352     rotateToY(uAuxData, uData, uToy);
00353     rotateToY(uAuxData, gCTData, rgcToy);
00354     rotateToZ(uAuxData, gCFData, lgcToz);
00355     rotateToZ(uAuxData, uData, uToz);
00356     rotateToZ(uAuxData, gCADATA, rgcToz);
00357 }
00358
00359 /// Rotate halo and inner-patch data vectors with rotation matrix Rx into
00360 /// eigenspace of Z matrix and write to auxiliary vector
00361 inline void LatticePatch::rotateToX(sunrealtype *outArray,
00362                                     const unrealtype *inArray,
00363                                     const vector<int> &lookup) {
00364     int ii = 0, target = 0;
00365 #pragma ivdep
00366 #pragma omp simd // safelen(6) - also good
00367 #pragma distribute_point
00368 for (unsigned int i = 0; i < lookup.size(); i++) {
00369     // get correct u-vector and spatial indices along previously defined lookup
00370     // tables
00371     target = envelopeLattice->get_dataPointDimension() * lookup[i];
00372     ii = envelopeLattice->get_dataPointDimension() * i;
00373     outArray[target + 0] = -inArray[1 + ii] + inArray[5 + ii];
00374     outArray[target + 1] = inArray[2 + ii] + inArray[4 + ii];
00375     outArray[target + 2] = inArray[1 + ii] + inArray[5 + ii];
00376     outArray[target + 3] = -inArray[2 + ii] + inArray[4 + ii];
00377     outArray[target + 4] = inArray[3 + ii];
00378     outArray[target + 5] = inArray[ii];
00379 }
00380 }
00381
00382 /// Rotate halo and inner-patch data vectors with rotation matrix Ry into
00383 /// eigenspace of Z matrix and write to auxiliary vector
00384 inline void LatticePatch::rotateToY(sunrealtype *outArray,
00385                                     const unrealtype *inArray,
00386                                     const vector<int> &lookup) {
00387     int ii = 0, target = 0;
00388 #pragma ivdep
00389 #pragma omp simd // safelen(6)
00390 #pragma distribute_point
00391 for (unsigned int i = 0; i < lookup.size(); i++) {
00392     target = envelopeLattice->get_dataPointDimension() * lookup[i];
00393     ii = envelopeLattice->get_dataPointDimension() * i;
00394     outArray[target + 0] = inArray[ii] + inArray[5 + ii];
00395     outArray[target + 1] = -inArray[2 + ii] + inArray[3 + ii];
00396     outArray[target + 2] = -inArray[ii] + inArray[5 + ii];
00397     outArray[target + 3] = inArray[2 + ii] + inArray[3 + ii];
00398     outArray[target + 4] = inArray[4 + ii];
00399     outArray[target + 5] = inArray[1 + ii];
00400 }
00401 }
00402
00403 /// Rotate halo and inner-patch data vectors with rotation matrix Rz into
00404 /// eigenspace of Z matrix and write to auxiliary vector
00405 inline void LatticePatch::rotateToZ(sunrealtype *outArray,
00406                                     const unrealtype *inArray,

```



```

00407                                     const vector<int> &lookup) {
00408     int ii = 0, target = 0;
00409     #pragma ivdep
00410     #pragma omp simd // safelen(6)
00411     #pragma distribute_point
00412     for (unsigned int i = 0; i < lookup.size(); i++) {
00413         target = envelopeLattice->get_dataPointDimension() * lookup[i];
00414         ii = envelopeLattice->get_dataPointDimension() * i;
00415         outArray[target + 0] = -inArray[ii] + inArray[4 + ii];
00416         outArray[target + 1] = inArray[1 + ii] + inArray[3 + ii];
00417         outArray[target + 2] = inArray[ii] + inArray[4 + ii];
00418         outArray[target + 3] = -inArray[1 + ii] + inArray[3 + ii];
00419         outArray[target + 4] = inArray[5 + ii];
00420         outArray[target + 5] = inArray[2 + ii];
00421     }
00422 }
00423
00424 /// Derotate uAux with transposed rotation matrices and write to derivative
00425 /// buffer -- normalization is done here by the factor 1/2
00426 void LatticePatch::derotate(int dir, sunrealttype *buffOut) {
00427     // Check that the lattice as well as the translocation lookups have been set
00428     // up;
00429     checkFlag(FLatticePatchSetUp);
00430     checkFlag(TranslocationLookupSetUp);
00431     const int dPD = envelopeLattice->get_dataPointDimension();
00432     const int gLW = envelopeLattice->get_ghostLayerWidth();
00433     const int uSize = discreteSize();
00434     int ii = 0, target = 0;
00435     switch (dir) {
00436     case 1:
00437         #pragma ivdep
00438         #pragma omp simd // safelen(6) - also good
00439         #pragma distribute_point
00440         for (int i = 0; i < uSize; i++) {
00441             // get correct indices in u and rotation space
00442             target = dPD * i;
00443             ii = dPD * (uTox[i] - gLW);
00444             buffOut[target + 0] = uAux[5 + ii];
00445             buffOut[target + 1] = (-uAux[ii] + uAux[2 + ii]) / 2.;
00446             buffOut[target + 2] = (uAux[1 + ii] - uAux[3 + ii]) / 2.;
00447             buffOut[target + 3] = uAux[4 + ii];
00448             buffOut[target + 4] = (uAux[1 + ii] + uAux[3 + ii]) / 2.;
00449             buffOut[target + 5] = (uAux[ii] + uAux[2 + ii]) / 2.;
00450         }
00451         break;
00452     case 2:
00453         #pragma omp simd // safelen(6)
00454         #pragma distribute_point
00455         for (int i = 0; i < uSize; i++) {
00456             target = dPD * i;
00457             ii = dPD * (uToy[i] - gLW);
00458             buffOut[target + 0] = (uAux[ii] - uAux[2 + ii]) / 2.;
00459             buffOut[target + 1] = uAux[5 + ii];
00460             buffOut[target + 2] = (-uAux[1 + ii] + uAux[3 + ii]) / 2.;
00461             buffOut[target + 3] = (uAux[1 + ii] + uAux[3 + ii]) / 2.;
00462             buffOut[target + 4] = uAux[4 + ii];
00463             buffOut[target + 5] = (uAux[ii] + uAux[2 + ii]) / 2.;
00464         }
00465         break;
00466     case 3:
00467         #pragma omp simd // safelen(6)
00468         #pragma distribute_point
00469         for (int i = 0; i < uSize; i++) {
00470             target = dPD * i;
00471             ii = dPD * (uToz[i] - gLW);
00472             buffOut[target + 0] = (-uAux[ii] + uAux[2 + ii]) / 2.;
00473             buffOut[target + 1] = (uAux[1 + ii] - uAux[3 + ii]) / 2.;
00474             buffOut[target + 2] = uAux[5 + ii];
00475             buffOut[target + 3] = (uAux[1 + ii] + uAux[3 + ii]) / 2.;
00476             buffOut[target + 4] = (uAux[ii] + uAux[2 + ii]) / 2.;
00477             buffOut[target + 5] = uAux[4 + ii];
00478         }
00479         break;
00480     default:
00481         errorKill("Tried to derotate from the wrong direction");
00482         break;
00483     }
00484 }
00485
00486 /// Create buffers to save derivative values, optimizing computational load
00487 void LatticePatch::initializeBuffers() {
00488     // Check that the lattice has been set up
00489     checkFlag(FLatticeDimensionSet);
00490     const int dPD = envelopeLattice->get_dataPointDimension();
00491     buffFX.resize(nx * ny * nz * dPD);
00492     buffFY.resize(nx * ny * nz * dPD);
00493     buffFZ.resize(nx * ny * nz * dPD);

```

```

00494 // Set pointers used for propagation functions
00495 buffData[0] = &buffX[0];
00496 buffData[1] = &buffY[0];
00497 buffData[2] = &buffZ[0];
00498 statusFlags |= BuffersInitialized;
00499 }
00500
00501 /// Perform the ghost cell exchange in a specified direction
00502 void LatticePatch::exchangeGhostCells(const int dir) {
00503 // Check that the lattice has been set up
00504 checkFlag(FLatticeDimensionSet);
00505 checkFlag(FLatticePatchSetUp);
00506 // Variables to per dimension calculate the halo indices, and distance to
00507 // other side halo boundary
00508 int mx = 1, my = 1, mz = 1, distToRight = 1;
00509 const int gLW = envelopeLattice->get_ghostLayerWidth();
00510 // In the chosen direction m is set to ghost layer width while the others
00511 // remain to form the plane
00512 switch (dir) {
00513 case 1:
00514     mx = gLW;
00515     my = ny;
00516     mz = nz;
00517     distToRight = (nx - gLW);
00518     break;
00519 case 2:
00520     mx = nx;
00521     my = gLW;
00522     mz = nz;
00523     distToRight = nx * (ny - gLW);
00524     break;
00525 case 3:
00526     mx = nx;
00527     my = ny;
00528     mz = gLW;
00529     distToRight = nx * ny * (nz - gLW);
00530     break;
00531 }
00532 // total number of exchanged points
00533 const int dPD = envelopeLattice->get_dataPointDimension();
00534 const int exchangeSize = mx * my * mz * dPD;
00535 // provide size of the halos for ghost cells
00536 ghostCellLeft.resize(exchangeSize);
00537 ghostCellRight.resize(ghostCellLeft.size());
00538 ghostCellLeftToSend.resize(ghostCellLeft.size());
00539 ghostCellRightToSend.resize(ghostCellLeft.size());
00540 gCLData = &ghostCellLeft[0];
00541 gCRData = &ghostCellRight[0];
00542 statusFlags |= GhostLayersInitialized;
00543
00544 // Initialize running index li for the halo buffers, and index ui of uData for
00545 // data transfer
00546 int li = 0, ui = 0;
00547
00548 // #pragma omp parallel for reduction(+:ui) reduction(+:li) -> don't probably
00549 // bad idea to parallelize ghost cell exchange Loop over to be copied points in
00550 // z and y direction
00551 #pragma distribute_point
00552 for (int iz = 0; iz < mz; iz++) {
00553     for (int iy = 0; iy < my; iy++) {
00554         // uData vector start index of halo data to be transferred
00555         // with each z-step add the whole xy-plane and with y-step the x-range ->
00556         // iterate all x-ranges
00557         ui = (iz * nx * ny + iy * nx) * dPD;
00558         // copy left halo data from uData to buffer, transfer size is given by
00559         // x-length (not x-range) perhaps faster but more fragile C lib copy
00560         // operation (contained in cstring header)
00561         /*
00562         memcpy(&ghostCellLeftToSend[li],
00563             &uData[ui],
00564             sizeof(sunrealtype)*mx*dPD);
00565         // increase ui by the distance to vis-a-vis boundary and copy right halo
00566         data to buffer ui+=distToRight*dPD; memcpy(&ghostCellRightToSend[li],
00567             &uData[ui],
00568             sizeof(sunrealtype)*mx*dPD);
00569         */
00570         // perhaps more safe but slower copy operation (contained in algorithm
00571         // header) performance highly system dependent
00572         copy(&uData[ui], &uData[ui + mx * dPD], &ghostCellLeftToSend[li]);
00573         ui += distToRight * dPD;
00574         copy(&uData[ui], &uData[ui + mx * dPD], &ghostCellRightToSend[li]);
00575
00576         // increase halo index by transferred items per y-iteration step
00577         // (x-length)
00578         li += mx * dPD;
00579     }
00580 }

```

```

00581
00582  /* Send and receive the data to and from neighboring latticePatches */
00583  // Adjust direction to cartesian communicator
00584  int dim = 2; // default for dir==1
00585  if (dir == 2) {
00586      dim = 1;
00587  } else if (dir == 3) {
00588      dim = 0;
00589  }
00590  int rank_source = 0, rank_dest = 0;
00591  MPI_Cart_shift(envelopeLattice->comm, dim, -1, &rank_source,
00592               &rank_dest); // s.t. rank_dest is left & v.v.
00593
00594  // nonblocking Isend/Irecv
00595
00596  MPI_Request requests[4];
00597  MPI_Irecv(&ghostCellRight[0], exchangeSize, MPI_SUNREALTYPE, rank_source, 1,
00598           envelopeLattice->comm, &requests[0]);
00599  MPI_Isend(&ghostCellLeftToSend[0], exchangeSize, MPI_SUNREALTYPE, rank_dest,
00600           1, envelopeLattice->comm, &requests[1]);
00601  MPI_Irecv(&ghostCellLeft[0], exchangeSize, MPI_SUNREALTYPE, rank_dest, 2,
00602           envelopeLattice->comm, &requests[2]);
00603  MPI_Isend(&ghostCellRightToSend[0], exchangeSize, MPI_SUNREALTYPE,
00604           rank_source, 2, envelopeLattice->comm, &requests[3]);
00605  MPI_Waitall(4, requests, MPI_STATUS_IGNORE);
00606
00607
00608  // blocking Sendrecv:
00609  /*
00610  MPI_Sendrecv(&ghostCellLeftToSend[0], exchangeSize, MPI_SUNREALTYPE,
00611              rank_dest, 1, &ghostCellRight[0], exchangeSize, MPI_SUNREALTYPE,
00612              rank_source, 1, envelopeLattice->comm, MPI_STATUS_IGNORE);
00613  MPI_Sendrecv(&ghostCellRightToSend[0], exchangeSize, MPI_SUNREALTYPE,
00614              rank_source, 2, &ghostCellLeft[0], exchangeSize, MPI_SUNREALTYPE,
00615              rank_dest, 2, envelopeLattice->comm, MPI_STATUS_IGNORE);
00616  */
00617  }
00618
00619  /// Exchange ghost cells with a neighborhood collective operation
00620  void LatticePatch::exchangeGhostCells3D() {
00621      // Check that the lattice has been set up
00622      checkFlag(FLatticeDimensionSet);
00623      // ghostlayerwidth
00624      const int gLW = envelopeLattice->get_ghostLayerWidth();
00625      // datapoint dimension
00626      const int dPD = envelopeLattice->get_dataPointDimension();
00627      // total number of exchanged points per halo
00628      const int n = nx; // only cubic patches allowed -> use general length n
00629      const int exchangeSize = n * n * gLW * dPD;
00630      // Give ghostCells the total size of the ghost layers (the six halos)
00631      const int tot_exchangeSize = 6 * exchangeSize;
00632      ghostCells.resize(tot_exchangeSize);
00633      ghostCellsToSend.resize(ghostCells.size());
00634      // ghost cell data in all directions: left,right,bottom,top,front,abaf; but
00635      // with MPI dim order "lefts" are the first receivers and "rights" are the
00636      // first senders -> see buffer creator below
00637      gCFData = &ghostCells[0];
00638      gCAData = &ghostCells[exchangeSize];
00639      gCBData = &ghostCells[2 * exchangeSize];
00640      gCTData = &ghostCells[3 * exchangeSize];
00641      gCLData = &ghostCells[4 * exchangeSize];
00642      gCRData = &ghostCells[5 * exchangeSize];
00643      statusFlags |= GhostLayersInitialized;
00644
00645      checkFlag(FLatticePatchSetUp);
00646      // variables to set to ghost layer width and point distance to next
00647      // communication point -> depends on direction
00648
00649      int li = 0; // running index for buffers
00650      int distToRight = 0; // distance to vis-a-vis halo data, varies per dim
00651      // filling buffers along the MPI dim order
00652      distToRight = n * n * (n - gLW);
00653      bufferCreator(li, n, n, gLW, distToRight);
00654      li += 2 * exchangeSize; // li increases by two exchange sizes per dim
00655
00656      distToRight = n * (n - gLW);
00657      bufferCreator(li, n, gLW, n, distToRight);
00658      li += 2 * exchangeSize;
00659
00660      distToRight = (n - gLW);
00661      bufferCreator(li, gLW, n, n, distToRight);
00662
00663      MPI_Neighbor_alltoall(&ghostCellsToSend[0], exchangeSize, MPI_SUNREALTYPE,
00664                          &ghostCells[0], exchangeSize, MPI_SUNREALTYPE,
00665                          envelopeLattice->comm);
00666  }
00667

```

```

00668 /// Fill the halo buffers for neighborhood collectives
00669 void LatticePatch::bufferCreator(int li, int mx, int my, int mz,
00670     int distToRight) {
00671     const int dPD = envelopeLattice->get_dataPointDimension();
00672     // Initialize running index ui for to be transferred uData
00673     int ui = 0;
00674
00675     // #pragma omp parallel for reduction(+:ui) reduction(+:li) -> don't, probably
00676     // bad idea to parallelize ghost cell exchange Loop over all planes and pick to
00677     // be transferred points (uData indices)
00678     #pragma distribute_point
00679     for (int iz = 0; iz < mz; iz++) {
00680         for (int iy = 0; iy < my; iy++) {
00681             // start index of uData vector halo data to be transferred
00682             // Here, in contrast to above, start at right boundary to send to left
00683             // s.t. updated left values are the first indices (bec of neighborhood
00684             // collective pattern) with each z-step add the whole xy-plane and with
00685             // y-step the x-range -> iterate all x-ranges
00686             ui = (iz * nx * ny + iy * nx + distToRight) * dPD;
00687             // copy from uData from right boundary (at each dimension) into buffer,
00688             // halo transfer size is given by x length at each step
00689             // memcpy(&ghostCellsToSend[li], \
00690                 &uData[ui], \
00691                 sizeof(sunrealtype)*mx*dPD);
00692             copy(&uData[ui], &uData[ui + mx * dPD], &ghostCellsToSend[li]);
00693             // increase li by transferred indices in this loop-step
00694             li += mx * dPD;
00695         }
00696     }
00697
00698     #pragma distribute_point
00699     for (int iz = 0; iz < mz; iz++) {
00700         for (int iy = 0; iy < my; iy++) {
00701             // Now copy from left boundary into buffer
00702             ui = (iz * nx * ny + iy * nx) * dPD;
00703             //memcpy(&ghostCellsToSend[li], \
00704                 &uData[ui], \
00705                 sizeof(sunrealtype)*mx*dPD);
00706             copy(&uData[ui], &uData[ui + mx * dPD], &ghostCellsToSend[li]);
00707             li += mx * dPD;
00708         }
00709     }
00710 }
00711
00712 /// Check if all flags are set
00713 void LatticePatch::checkFlag(unsigned int flag) const {
00714     if (!(statusFlags & flag)) {
00715         string errorMessage;
00716         switch (flag) {
00717             case FLatticePatchSetUp:
00718                 errorMessage = "The Lattice patch was not set up please make sure to "
00719                     "initilize a Lattice topology";
00720                 break;
00721             case TranslocationLookupSetUp:
00722                 errorMessage = "The translocation lookup tables have not been generated, "
00723                     "please be sure to run generateTranslocationLookup()";
00724                 break;
00725             case GhostLayersInitialized:
00726                 errorMessage = "The space for the ghost layers has not been allocated, "
00727                     "please be sure to run initializeGhostLayer()";
00728                 break;
00729             case BuffersInitialized:
00730                 errorMessage = "The space for the buffers has not been allocated, please "
00731                     "be sure to run initializeBuffers()";
00732                 break;
00733             default:
00734                 errorMessage = "Uppss, you've made a non-standard error, sadly I can't "
00735                     "help you there";
00736                 break;
00737         }
00738         errorKill(errorMessage);
00739     }
00740     return;
00741 }
00742
00743 /// Calculate derivatives in the patch (uAux) in the specified direction
00744 void LatticePatch::derive(const int dir) {
00745     // ghost layer width
00746     const int gLW = envelopeLattice->get_ghostLayerWidth();
00747     // dimensionality of data points -> 6
00748     const int dPD = envelopeLattice->get_dataPointDimension();
00749     // total width of patch in given direction including ghost layers at ends
00750     const int dirWidth = discreteSize(dir) + 2 * gLW;
00751     // width of patch only in given direction
00752     const int dirWidth0 = discreteSize(dir);
00753     // size of plane perpendicular to given dimension
00754     const int perpPlainSize = discreteSize() / discreteSize(dir);

```

```

00755 // physical distance between points in that direction
00756 sunrealtype dxi = NAN;
00757 switch (dir) {
00758 case 1:
00759     dxi = dx;
00760     break;
00761 case 2:
00762     dxi = dy;
00763     break;
00764 case 3:
00765     dxi = dz;
00766     break;
00767 default:
00768     dxi = 1;
00769     errorKill("Tried to derive in the wrong direction");
00770     break;
00771 }
00772 // Derive according to chosen stencil accuracy order (which determines also
00773 // gLW)
00774 const int order = envelopeLattice->get_stencilOrder();
00775 switch (order) {
00776 case 1:
00777     for (int i = 0; i < perpPlainSize; i++) {
00778         for (int j = (i * dirWidth + gLW) * dPD;
00779             j < (i * dirWidth + gLW + dirWidthO) * dPD; j += dPD) {
00780             uAux[j + 0 - gLW * dPD] = s1b(&uAux[j + 0]) / dxi;
00781             uAux[j + 1 - gLW * dPD] = s1b(&uAux[j + 1]) / dxi;
00782             uAux[j + 2 - gLW * dPD] = s1f(&uAux[j + 2]) / dxi;
00783             uAux[j + 3 - gLW * dPD] = s1f(&uAux[j + 3]) / dxi;
00784             uAux[j + 4 - gLW * dPD] = s1f(&uAux[j + 4]) / dxi;
00785             uAux[j + 5 - gLW * dPD] = s1f(&uAux[j + 5]) / dxi;
00786         }
00787     }
00788     break;
00789 case 2:
00790     for (int i = 0; i < perpPlainSize; i++) {
00791         for (int j = (i * dirWidth + gLW) * dPD;
00792             j < (i * dirWidth + gLW + dirWidthO) * dPD; j += dPD) {
00793             uAux[j + 0 - gLW * dPD] = s2b(&uAux[j + 0]) / dxi;
00794             uAux[j + 1 - gLW * dPD] = s2b(&uAux[j + 1]) / dxi;
00795             uAux[j + 2 - gLW * dPD] = s2f(&uAux[j + 2]) / dxi;
00796             uAux[j + 3 - gLW * dPD] = s2f(&uAux[j + 3]) / dxi;
00797             uAux[j + 4 - gLW * dPD] = s2c(&uAux[j + 4]) / dxi;
00798             uAux[j + 5 - gLW * dPD] = s2c(&uAux[j + 5]) / dxi;
00799         }
00800     }
00801     break;
00802 case 3:
00803     for (int i = 0; i < perpPlainSize; i++) {
00804         for (int j = (i * dirWidth + gLW) * dPD;
00805             j < (i * dirWidth + gLW + dirWidthO) * dPD; j += dPD) {
00806             uAux[j + 0 - gLW * dPD] = s3b(&uAux[j + 0]) / dxi;
00807             uAux[j + 1 - gLW * dPD] = s3b(&uAux[j + 1]) / dxi;
00808             uAux[j + 2 - gLW * dPD] = s3f(&uAux[j + 2]) / dxi;
00809             uAux[j + 3 - gLW * dPD] = s3f(&uAux[j + 3]) / dxi;
00810             uAux[j + 4 - gLW * dPD] = s3f(&uAux[j + 4]) / dxi;
00811             uAux[j + 5 - gLW * dPD] = s3f(&uAux[j + 5]) / dxi;
00812         }
00813     }
00814     break;
00815 case 4:
00816     for (int i = 0; i < perpPlainSize; i++) {
00817         for (int j = (i * dirWidth + gLW) * dPD;
00818             j < (i * dirWidth + gLW + dirWidthO) * dPD; j += dPD) {
00819             uAux[j + 0 - gLW * dPD] = s4b(&uAux[j + 0]) / dxi;
00820             uAux[j + 1 - gLW * dPD] = s4b(&uAux[j + 1]) / dxi;
00821             uAux[j + 2 - gLW * dPD] = s4f(&uAux[j + 2]) / dxi;
00822             uAux[j + 3 - gLW * dPD] = s4f(&uAux[j + 3]) / dxi;
00823             uAux[j + 4 - gLW * dPD] = s4c(&uAux[j + 4]) / dxi;
00824             uAux[j + 5 - gLW * dPD] = s4c(&uAux[j + 5]) / dxi;
00825         }
00826     }
00827     break;
00828 case 5:
00829     for (int i = 0; i < perpPlainSize; i++) {
00830         for (int j = (i * dirWidth + gLW) * dPD;
00831             j < (i * dirWidth + gLW + dirWidthO) * dPD; j += dPD) {
00832             uAux[j + 0 - gLW * dPD] = s5b(&uAux[j + 0]) / dxi;
00833             uAux[j + 1 - gLW * dPD] = s5b(&uAux[j + 1]) / dxi;
00834             uAux[j + 2 - gLW * dPD] = s5f(&uAux[j + 2]) / dxi;
00835             uAux[j + 3 - gLW * dPD] = s5f(&uAux[j + 3]) / dxi;
00836             uAux[j + 4 - gLW * dPD] = s5f(&uAux[j + 4]) / dxi;
00837             uAux[j + 5 - gLW * dPD] = s5f(&uAux[j + 5]) / dxi;
00838         }
00839     }
00840     break;
00841 case 6:

```

```

00842     for (int i = 0; i < perpPlainSize; i++) {
00843         for (int j = (i * dirWidth + gLW) * dPD;
00844             j < (i * dirWidth + gLW + dirWidthO) * dPD; j += dPD) {
00845             uAux[j + 0 - gLW * dPD] = s6b(&uAux[j + 0]) / dxi;
00846             uAux[j + 1 - gLW * dPD] = s6b(&uAux[j + 1]) / dxi;
00847             uAux[j + 2 - gLW * dPD] = s6f(&uAux[j + 2]) / dxi;
00848             uAux[j + 3 - gLW * dPD] = s6f(&uAux[j + 3]) / dxi;
00849             uAux[j + 4 - gLW * dPD] = s6c(&uAux[j + 4]) / dxi;
00850             uAux[j + 5 - gLW * dPD] = s6c(&uAux[j + 5]) / dxi;
00851         }
00852     }
00853     break;
00854 case 7:
00855     for (int i = 0; i < perpPlainSize; i++) {
00856         for (int j = (i * dirWidth + gLW) * dPD;
00857             j < (i * dirWidth + gLW + dirWidthO) * dPD; j += dPD) {
00858             uAux[j + 0 - gLW * dPD] = s7b(&uAux[j + 0]) / dxi;
00859             uAux[j + 1 - gLW * dPD] = s7b(&uAux[j + 1]) / dxi;
00860             uAux[j + 2 - gLW * dPD] = s7f(&uAux[j + 2]) / dxi;
00861             uAux[j + 3 - gLW * dPD] = s7f(&uAux[j + 3]) / dxi;
00862             uAux[j + 4 - gLW * dPD] = s7f(&uAux[j + 4]) / dxi;
00863             uAux[j + 5 - gLW * dPD] = s7f(&uAux[j + 5]) / dxi;
00864         }
00865     }
00866     break;
00867 case 8:
00868     for (int i = 0; i < perpPlainSize; i++) {
00869         for (int j = (i * dirWidth + gLW) * dPD;
00870             j < (i * dirWidth + gLW + dirWidthO) * dPD; j += dPD) {
00871             uAux[j + 0 - gLW * dPD] = s8b(&uAux[j + 0]) / dxi;
00872             uAux[j + 1 - gLW * dPD] = s8b(&uAux[j + 1]) / dxi;
00873             uAux[j + 2 - gLW * dPD] = s8f(&uAux[j + 2]) / dxi;
00874             uAux[j + 3 - gLW * dPD] = s8f(&uAux[j + 3]) / dxi;
00875             uAux[j + 4 - gLW * dPD] = s8c(&uAux[j + 4]) / dxi;
00876             uAux[j + 5 - gLW * dPD] = s8c(&uAux[j + 5]) / dxi;
00877         }
00878     }
00879     break;
00880 case 9:
00881     for (int i = 0; i < perpPlainSize; i++) {
00882         for (int j = (i * dirWidth + gLW) * dPD;
00883             j < (i * dirWidth + gLW + dirWidthO) * dPD; j += dPD) {
00884             uAux[j + 0 - gLW * dPD] = s9b(&uAux[j + 0]) / dxi;
00885             uAux[j + 1 - gLW * dPD] = s9b(&uAux[j + 1]) / dxi;
00886             uAux[j + 2 - gLW * dPD] = s9f(&uAux[j + 2]) / dxi;
00887             uAux[j + 3 - gLW * dPD] = s9f(&uAux[j + 3]) / dxi;
00888             uAux[j + 4 - gLW * dPD] = s9f(&uAux[j + 4]) / dxi;
00889             uAux[j + 5 - gLW * dPD] = s9f(&uAux[j + 5]) / dxi;
00890         }
00891     }
00892     break;
00893 case 10:
00894     for (int i = 0; i < perpPlainSize; i++) {
00895         for (int j = (i * dirWidth + gLW) * dPD;
00896             j < (i * dirWidth + gLW + dirWidthO) * dPD; j += dPD) {
00897             uAux[j + 0 - gLW * dPD] = s10b(&uAux[j + 0]) / dxi;
00898             uAux[j + 1 - gLW * dPD] = s10b(&uAux[j + 1]) / dxi;
00899             uAux[j + 2 - gLW * dPD] = s10f(&uAux[j + 2]) / dxi;
00900             uAux[j + 3 - gLW * dPD] = s10f(&uAux[j + 3]) / dxi;
00901             uAux[j + 4 - gLW * dPD] = s10c(&uAux[j + 4]) / dxi;
00902             uAux[j + 5 - gLW * dPD] = s10c(&uAux[j + 5]) / dxi;
00903         }
00904     }
00905     break;
00906 case 11:
00907     for (int i = 0; i < perpPlainSize; i++) {
00908         for (int j = (i * dirWidth + gLW) * dPD;
00909             j < (i * dirWidth + gLW + dirWidthO) * dPD; j += dPD) {
00910             uAux[j + 0 - gLW * dPD] = s11b(&uAux[j + 0]) / dxi;
00911             uAux[j + 1 - gLW * dPD] = s11b(&uAux[j + 1]) / dxi;
00912             uAux[j + 2 - gLW * dPD] = s11f(&uAux[j + 2]) / dxi;
00913             uAux[j + 3 - gLW * dPD] = s11f(&uAux[j + 3]) / dxi;
00914             uAux[j + 4 - gLW * dPD] = s11f(&uAux[j + 4]) / dxi;
00915             uAux[j + 5 - gLW * dPD] = s11f(&uAux[j + 5]) / dxi;
00916         }
00917     }
00918     break;
00919 case 12:
00920     for (int i = 0; i < perpPlainSize; i++) {
00921         for (int j = (i * dirWidth + gLW) * dPD;
00922             j < (i * dirWidth + gLW + dirWidthO) * dPD; j += dPD) {
00923             uAux[j + 0 - gLW * dPD] = s12b(&uAux[j + 0]) / dxi;
00924             uAux[j + 1 - gLW * dPD] = s12b(&uAux[j + 1]) / dxi;
00925             uAux[j + 2 - gLW * dPD] = s12f(&uAux[j + 2]) / dxi;
00926             uAux[j + 3 - gLW * dPD] = s12f(&uAux[j + 3]) / dxi;
00927             uAux[j + 4 - gLW * dPD] = s12c(&uAux[j + 4]) / dxi;
00928             uAux[j + 5 - gLW * dPD] = s12c(&uAux[j + 5]) / dxi;

```

```

00929     }
00930     }
00931     break;
00932 case 13:
00933     // #pragma omp parallel for default(none) firstprivate(uAux)
00934     // shared(dxi, dirWidth, dirWidth0, gLW, dPD) collapse(2) schedule(static, 6)
00935     // #pragma ivdep
00936     // #pragma distribute_point -> No.
00937     // #pragma unroll_and_jam
00938     // Iterate through all points in the plane perpendicular to the given
00939     // direction
00940     for (int i = 0; i < perpPlainSize; i++) {
00941         // stencil functions range over 2*gLW+6 indices, attention to cache-line
00942         // false-sharing
00943         // #pragma omp simd safelen(2*gLW+dPD)
00944         // Iterate through the direction for each perpendicular plane point
00945         for (int j = (i * dirWidth + gLW /*to shift left by gLW below */) * dPD;
00946             j < (i * dirWidth + gLW + dirWidth0) * dPD; j += dPD) {
00947             /* Compute the stencil derivative for any of the six field components
00948              * with a ghostlayer width adjusted to the order of the finite
00949              * difference scheme */
00950             uAux[j + 0 - gLW * dPD] = s13b(&uAux[j + 0]) / dxi;
00951             uAux[j + 1 - gLW * dPD] = s13b(&uAux[j + 1]) / dxi;
00952             uAux[j + 2 - gLW * dPD] = s13f(&uAux[j + 2]) / dxi;
00953             uAux[j + 3 - gLW * dPD] = s13f(&uAux[j + 3]) / dxi;
00954             uAux[j + 4 - gLW * dPD] = s13f(&uAux[j + 4]) / dxi;
00955             uAux[j + 5 - gLW * dPD] = s13f(&uAux[j + 5]) / dxi;
00956         }
00957     }
00958     break;
00959 default:
00960     errorKill("Please set an existing stencil order");
00961     break;
00962 }
00963 }
00964 }
00965
00966 // Helper functions
00967
00968 // Print a specific error message to stdout
00969 void errorKill(const string & errorMessage) {
00970     cerr << endl << "Error: " << errorMessage << " Aborting..." << endl;
00971     MPI_Abort(MPI_COMM_WORLD, 1);
00972     return;
00973 }
00974
00975 /** Check function return value. From CNode examples.
00976     opt == 0 means SUNDIALS function allocates memory so check if
00977     returned NULL pointer
00978     opt == 1 means SUNDIALS function returns an integer value so check if
00979     retval < 0
00980     opt == 2 means function allocates memory so check if returned
00981     NULL pointer */
00982 int check_retval(void *returnvalue, const char *funcname, int opt, int id) {
00983     int *retval = nullptr;
00984
00985     /* Check if SUNDIALS function returned NULL pointer - no memory allocated */
00986     if (opt == 0 && returnvalue == nullptr) {
00987         fprintf(stderr,
00988             "\nSUNDIALS_ERROR(%d): %s() failed - returned NULL pointer\n\n", id,
00989             funcname);
00990         return (1);
00991     }
00992
00993     /* Check if retval < 0 */
00994     else if (opt == 1) {
00995         retval = (int *)returnvalue;
00996         if (*retval < 0) {
00997             fprintf(stderr, "\nSUNDIALS_ERROR(%d): %s() failed with retval = %d\n\n",
00998                 id, funcname, *retval);
00999             return (1);
01000         }
01001     }
01002
01003     /* Check if function returned NULL pointer - no memory allocated */
01004     else if (opt == 2 && returnvalue == nullptr) {
01005         fprintf(stderr,
01006             "\nMEMORY_ERROR(%d): %s() failed - returned NULL pointer\n\n", id,
01007             funcname);
01008         return (1);
01009     }
01010
01011     return (0);
01012 }

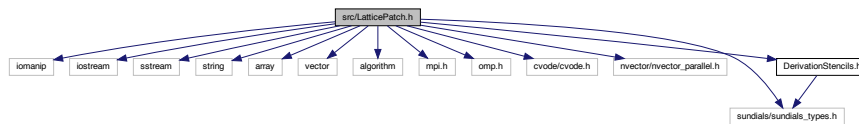
```

6.12 src/LatticePatch.h File Reference

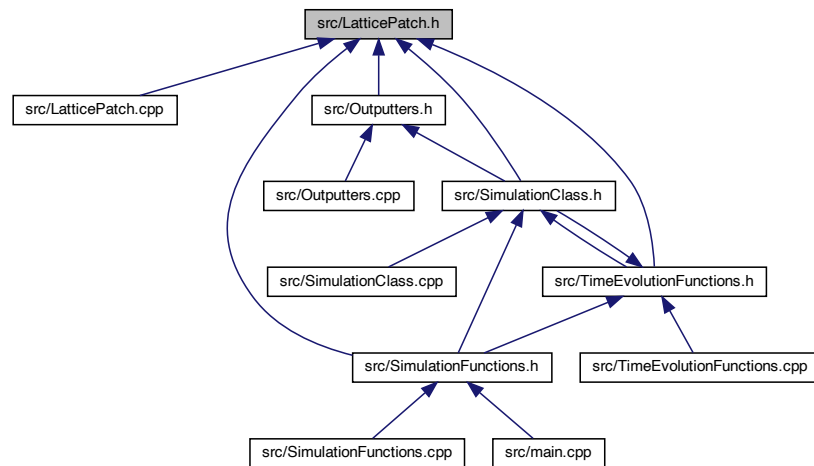
Declaration of the lattice and lattice patches.

```
#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 35 of file [LatticePatch.h](#).

```
00035 {
00036     FLatticeDimensionSet = 0x01, // 1
00037     /*OPT_B = 0x02, // 2
00038     OPT_C = 0x04, // 4
00039     OPT_D = 0x08, // 8
00040     OPT_E = 0x10, // 16
00041     OPT_F = 0x20,*/ // 32
00042 };
```

6.12.2.2 LatticePatchOptions

enum [LatticePatchOptions](#)

lattice patch construction checking flags

Enumerator

FLatticePatchSetUp	
TranslocationLookupSetUp	
GhostLayersInitialized	
BuffersInitialized	

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

```

00125     {
00126         FLatticePatchSetUp = 0x01,
00127         TranslocationLookupSetUp = 0x02,
00128         GhostLayersInitialized = 0x04,
00129         BuffersInitialized = 0x08
00130         /*OPT_D = 0x08,
00131         OPT_E = 0x10,
00132         OPT_F = 0x20,*/
00133     };

```

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

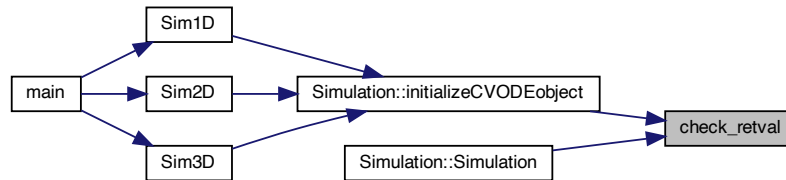
```

00982     {
00983         int *retval = nullptr;
00984
00985         /* Check if SUNDIALS function returned NULL pointer - no memory allocated */
00986         if (opt == 0 && returnvalue == nullptr) {
00987             fprintf(stderr,
00988                 "\nSUNDIALS_ERROR(%d): %s() failed - returned NULL pointer\n\n", id,
00989                 funcname);
00990             return (1);
00991         }
00992
00993         /* Check if retval < 0 */
00994         else if (opt == 1) {
00995             retval = (int *)returnvalue;
00996             if (*retval < 0) {
00997                 fprintf(stderr, "\nSUNDIALS_ERROR(%d): %s() failed with retval = %d\n\n",
00998                     id, funcname, *retval);
00999                 return (1);
01000             }
01001         }
01002
01003         /* Check if function returned NULL pointer - no memory allocated */
01004         else if (opt == 2 && returnvalue == nullptr) {
01005             fprintf(stderr,
01006                 "\nMEMORY_ERROR(%d): %s() failed - returned NULL pointer\n\n", id,
01007                 funcname);
01008             return (1);
01009         }
01010
01011         return (0);
01012     }

```

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 )
```

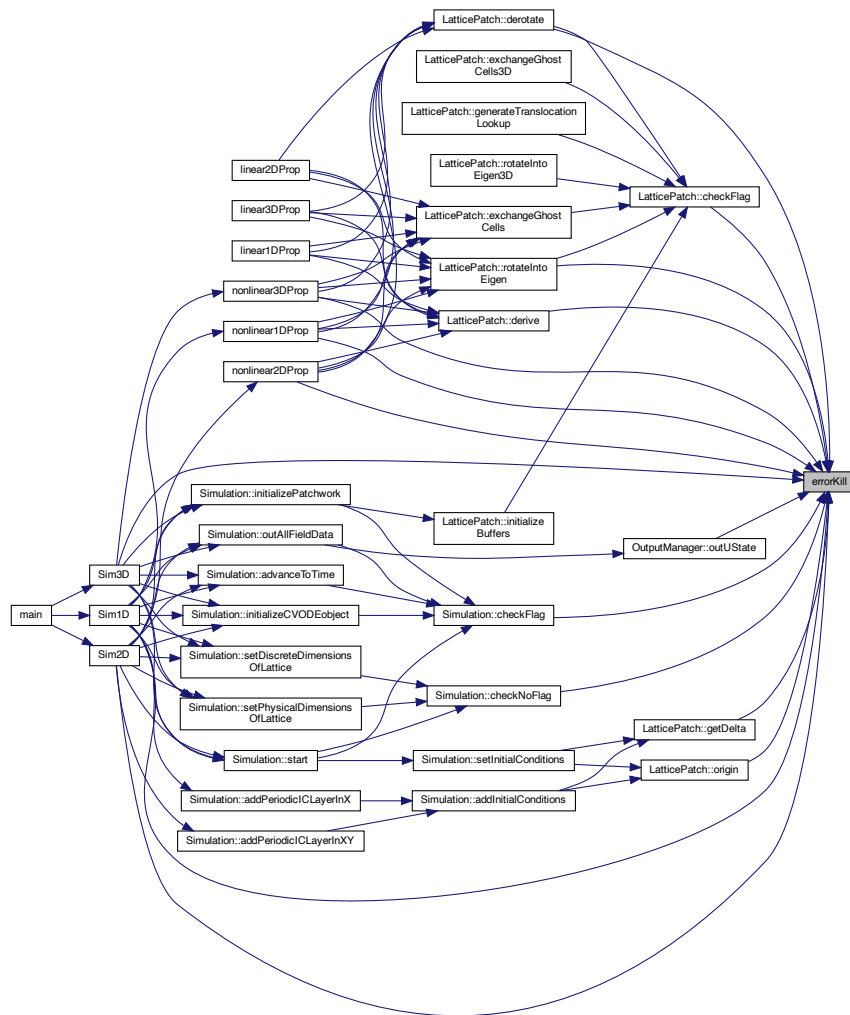
Print a specific error message to stdout.

Definition at line 969 of file [LatticePatch.cpp](#).

```
00969 {
00970     cerr << endl << "Error: " << errorMessage << " Aborting..." << endl;
00971     MPI_Abort(MPI_COMM_WORLD, 1);
00972     return;
00973 }
```

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

Here is the caller graph for this function:



6.13 LatticePatch.h

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

```

00001 //////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
00002 /// @file LatticePatch.h
00003 /// @brief Declaration of the lattice and lattice patches
00004 //////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
00005
00006 #pragma once
00007
00008 // IO
00009 #include <iomanip>
00010 #include <iostream>
00011 #include <sstream>
00012
00013 // string, container, algorithm
00014 #include <string>
00015 // #include <string_view>
00016 #include <array>
00017 #include <vector>
00018 #include <algorithm>
00019
00020 // MPI & OpenMP
00021 #include <mpi.h>
00022 #include <omp.h>

```

```

00023
00024 // Sundials
00025 #include <cvode/cvode.h> /* prototypes for CVODE fcts. */
00026 #include <nvector/nvector_parallel.h> /* definition of N_Vector and macros */
00027 #include <sundials/sundials_types.h> /* definition of type sunrealtype */
00028
00029 // stencils
00030 #include "DerivationStencils.h"
00031
00032 using namespace std;
00033
00034 // lattice construction checking flags
00035 enum LatticeOptions {
00036     FLatticeDimensionSet = 0x01, // 1
00037     /*OPT_B = 0x02, // 2
00038     OPT_C = 0x04, // 4
00039     OPT_D = 0x08, // 8
00040     OPT_E = 0x10, // 16
00041     OPT_F = 0x20,*/ // 32
00042 };
00043
00044 /** @brief Lattice class for the construction of the enveloping discrete
00045  * simulation space */
00046 class Lattice {
00047 private:
00048     /// physical size of the lattice in x-direction
00049     sunrealtype tot_lx;
00050     /// physical size of the lattice in y-direction
00051     sunrealtype tot_ly;
00052     /// physical size of the lattice in z-direction
00053     sunrealtype tot_lz;
00054     /// number of points in x-direction
00055     sunindextype tot_nx;
00056     /// number of points in y-direction
00057     sunindextype tot_ny;
00058     /// number of points in z-direction
00059     sunindextype tot_nz;
00060     /// total number of lattice points
00061     sunindextype tot_noP;
00062     /// dimension of each data point -> set once and for all
00063     static constexpr int dataPointDimension = 6;
00064     /// number of lattice points times data dimension of each point
00065     sunindextype tot_noDP;
00066     /// physical distance between lattice points in x-direction
00067     sunrealtype dx;
00068     /// physical distance between lattice points in y-direction
00069     sunrealtype dy;
00070     /// physical distance between lattice points in z-direction
00071     sunrealtype dz;
00072     /// stencil order
00073     const int stencilOrder;
00074     /// required width of ghost layers (depends on the stencil order)
00075     const int ghostLayerWidth;
00076     /// char for checking if lattice flags are set
00077     unsigned char statusFlags;
00078
00079 public:
00080     /// number of MPI processes
00081     int n_prc;
00082     /// number of MPI process
00083     int my_prc;
00084     /// personal communicator of the lattice
00085     MPI_Comm comm;
00086     /// function to create and deploy the cartesian communicator
00087     void initializeCommunicator(const int nx, const int ny,
00088                               const int nz, const bool per);
00089     /// default construction
00090     Lattice(const int StO);
00091     /// SUNContext object
00092     SUNContext sunctx;
00093     /// SUNProfiler object
00094     SUNProfiler profobj;
00095     /// component function for resizing the discrete dimensions of the lattice
00096     void setDiscreteDimensions(const sunindextype _nx,
00097                               const sunindextype _ny, const sunindextype _nz);
00098     /// component function for resizing the physical size of the lattice
00099     void setPhysicalDimensions(const sunrealtype _lx,
00100                               const sunrealtype _ly, const sunrealtype _lz);
00101     ///@{
00102     /** getter function */
00103     [[nodiscard]] const sunrealtype &get_tot_lx() const { return tot_lx; }
00104     [[nodiscard]] const sunrealtype &get_tot_ly() const { return tot_ly; }
00105     [[nodiscard]] const sunrealtype &get_tot_lz() const { return tot_lz; }
00106     [[nodiscard]] const sunindextype &get_tot_nx() const { return tot_nx; }
00107     [[nodiscard]] const sunindextype &get_tot_ny() const { return tot_ny; }
00108     [[nodiscard]] const sunindextype &get_tot_nz() const { return tot_nz; }
00109     [[nodiscard]] const sunindextype &get_tot_noP() const { return tot_noP; }

```

```

00110 [[nodiscard]] const sunindextype &get_tot_noDP() const { return tot_noDP; }
00111 [[nodiscard]] const sunrealtype &get_dx() const { return dx; }
00112 [[nodiscard]] const sunrealtype &get_dy() const { return dy; }
00113 [[nodiscard]] const sunrealtype &get_dz() const { return dz; }
00114 [[nodiscard]] constexpr int get_dataPointDimension() const {
00115     return dataPointDimension;
00116 }
00117 [[nodiscard]] const int &get_stencilOrder() const { return stencilOrder; }
00118 [[nodiscard]] const int &get_ghostLayerWidth() const {
00119     return ghostLayerWidth;
00120 }
00121 ///  

00122 };
00123  

00124 ///  

00125 // lattice patch construction checking flags
00126 enum LatticePatchOptions {
00127     FLatticePatchSetUp = 0x01,
00128     TranslocationLookupSetUp = 0x02,
00129     GhostLayersInitialized = 0x04,
00130     BuffersInitialized = 0x08
00131     /*OPT_D = 0x08,
00132     OPT_E = 0x10,
00133     OPT_F = 0x20,*/
00134 };
00135 ///  

00136 // @brief LatticePatch class for the construction of the patches in the
00137 // * enveloping lattice */
00138 class LatticePatch {
00139 private:
00140     ///  

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 LIy;
00151     // inner position of lattice-patch in the lattice patchwork; z-points
00152     sunindextype LIz;
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     // buffer for passing ghost cell data */
00184     vector<sunrealtype> ghostCellLeft, ghostCellRight, ghostCellLeftToSend,
00185     ghostCellRightToSend, ghostCellsToSend, ghostCells;
00186     ///  

00187     // ghost cell translocation lookup table */
00188     vector<int> lgcTox, rgcTox, lgcToy, rgcToy, lgcToz, rgcToz;
00189     ///  

00190     // char for checking flags */
00191     unsigned char statusFlags;
00192     ///  

00193     // rotate and translocate an input array according to a lookup into an output
00194     // * array */
00195     inline void rotateToX(sunrealtype *outArray, const sunrealtype *inArray,
00196         const vector<int> &lookup);

```

```

00197 inline void rotateToY(sunrealtype *outArray, const unrealtype *inArray,
00198                       const vector<int> &lookup);
00199 inline void rotateToZ(sunrealtype *outArray, const unrealtype *inArray,
00200                       const vector<int> &lookup);
00201 ///@}
00202 public:
00203 /// ID of the LatticePatch, corresponds to process number
00204 /// (required solely for debugging)
00205 int ID;
00206 /// N_Vector for saving field components u=(E,B) in lattice points
00207 N_Vector u;
00208 /// N_Vector for saving temporal derivatives of the field data
00209 N_Vector du;
00210 /// pointer to field data
00211 unrealtype *uData;
00212 /// pointer to auxiliary data vector
00213 unrealtype *uAuxData;
00214 /// pointer to time-derivative data
00215 unrealtype *duData;
00216 ///@{
00217 /** pointer to halo data */
00218 unrealtype *gCLData, *gCRData, *gCBData, *gCTData, *gCFData, *gCADData;
00219 ///@}
00220 /// pointer to spatial derivative data buffers
00221 array<unrealtype *, 3> buffData;
00222 /// constructor setting up a default first lattice patch
00223 LatticePatch();
00224 /// destructor freeing parallel vectors
00225 ~LatticePatch();
00226 /// friend function for creating the patchwork slicing of the overall lattice
00227 friend int generatePatchwork(const Lattice &envelopeLattice,
00228                             LatticePatch &patchToMold, const int DLx,
00229                             const int DLy, const int DLz);
00230 /// function to get the discrete size of the LatticePatch
00231 // (0 direction corresponds to total)
00232 int discreteSize(int dir=0) const;
00233 /// function to get the origin of the patch
00234 unrealtype origin(const int dir) const;
00235 /// function to get distance between points
00236 unrealtype getDelta(const int dir) const;
00237 /// function to fill out the lookup tables for translocation
00238 // and de-translocation of data point
00239 void generateTranslocationLookup();
00240 /// function to rotate u into Z-matrix eigenraum
00241 // and make it the primary lattice direction of dir
00242 void rotateIntoEigen(const int dir);
00243 /// function to rotate as in 'rotateIntoEigen' with special 3D halo buffers
00244 void rotateIntoEigen3D();
00245 /// function to derotate uAux into dudata lattice direction of x
00246 void derotate(int dir, unrealtype *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 exchange ghost cells using a neighborhood collective operation
00254 // for 3D simulations; requires cubic patches
00255 void exchangeGhostCells3D();
00256 /// outsourced convenience function to fill halo buffers with uData for 3D
00257 void bufferCreator(int li, int mx, int my, int mz, int distToRight);
00258 /// function to derive the centered values in uAux and save them noncentered
00259 void derive(const int dir);
00260 /// function to check if a flag has been set and if not abort
00261 void checkFlag(unsigned int flag) const;
00262 };
00263
00264 // helper function for error messages
00265 void errorKill(const string & errorMessage);
00266
00267 // helper function to check for CNode success
00268 int check_retval(void *returnvalue, const char *funcname, int opt, int id);
00269

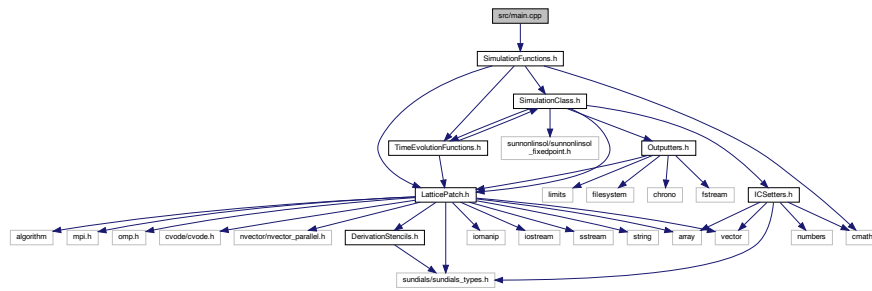
```

6.14 src/main.cpp File Reference

Main function to configure the user's simulation settings.

```
#include "SimulationFunctions.h"
```

Include dependency graph for main.cpp:



Functions

- int [main](#) (int argc, char *argv[])

6.14.1 Detailed Description

Main function to configure the user's simulation settings.

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

6.14.2 Function Documentation

6.14.2.1 main()

```
int main (
    int argc,
    char * argv[ ] )
```

Determine the output directory.

A "SimResults" folder will be created if non-existent with a subdirectory named in the identifier format "yy-mm-dd_hh-MM-ss" that contains the csv files

A 1D simulation with specified

- relative and absolute tolerances of the CVode solver
- accuracy order of the stencils in the range 1-13
- physical length of the lattice in meters
- number of lattice points
- periodic or vanishing boundary values

- included processes of the weak-field expansion, see [README.md](#)
- physical total simulation time
- discrete time steps
- output step multiples
- output in csv (c) or binary (b)

Add electromagnetic waves.

A plane wave with

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

Another plane wave with

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

A Gaussian wave with

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

Another Gaussian with

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

A 2D simulation with specified

- relative and absolute tolerances of the CVode solver
- accuracy order of the stencils in the range 1-13

- physical length of the lattice in the given dimensions in meters
- number of lattice points per dimension
- slicing of discrete dimensions into patches
- periodic or vanishing boundary values
- included processes of the weak-field expansion, see [README.md](#)
- physical total simulation time
- discrete time steps
- output step multiples
- output in csv (c) or binary (b)

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
- output in csv (c) or binary (b)

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 = "/gpfs/scratch/uh3o1/ru68dab/ru68dab/HE/outputs";
00026     //constexpr auto outputDirectory = "/home/andi/Documents/";
00027     constexpr auto outputDirectory = "/Users/andi/Documents/";
00028
00029     if(rank==0 && !filesystem::exists(outputDirectory)) {
00030         cerr<<"\nOutput directory nonexistent.\n";
00031         MPI_Abort(comm,1);
00032     }
00033
00034
00035     //----- BEGIN OF CONFIGURATION -----//
00036
00037     //////////////// -- 1D -- ///////////////////
00038     /** A 1D simulation with specified */
00039
00040     /// Specify your settings here ///
00041     constexpr array<sunrealtype,2> CNodeTolerances={1.0e-16,1.0e-16}; /// - relative and absolute
    tolerances of the CNode solver
```

```

00042     constexpr int StencilOrder=13;                                     /// - accuracy order of the
stencils in the range 1-13
00043     constexpr sunrealtype physical_sidelength=300e-6;                 /// - physical length of the
lattice in meters
00044     constexpr sunindextype latticepoints=6e3;                         /// - number of lattice points
00045     constexpr bool periodic=true;                                     /// - periodic or vanishing
boundary values
00046     int processOrder=3;                                              /// - included processes of the
weak-field expansion, see README.md
00047     constexpr sunrealtype simulationTime=100.0e-6l;                  /// - physical total
simulation time
00048     constexpr int numberOfSteps=100;                                  /// - discrete time steps
00049     constexpr int outputStep=1;                                       /// - output step multiples
00050     constexpr char outputStyle='c';                                   /// - output in csv (c) or binary
(b)
00051
00052     /// Add electromagnetic waves.
00053     planewave plane1;          /// A plane wave with
00054     plane1.k = {1e5,0,0};      /// - wavevector (normalized to \f$ 1/\lambda \f$)
00055     plane1.p = {0,0,0.1};      /// - amplitude/polarization
00056     plane1.phi = {0,0,0};      /// - phase shift
00057     planewave plane2;          /// Another plane wave with
00058     plane2.k = {-1e6,0,0};      /// - wavevector (normalized to \f$ 1/\lambda \f$)
00059     plane2.p = {0,0,0.5};      /// - amplitude/polarization
00060     plane2.phi = {0,0,0};      /// - phase shift
00061     // Do not comment out this vector, even if no plane wave is used. But if, emplace used plane
waves.
00062     vector<planewave> planewaves;
00063     //planewaves.emplace_back(plane1);
00064     //planewaves.emplace_back(plane2);
00065
00066     gaussian1D gauss1;          /// A Gaussian wave with
00067     gauss1.k = {1.0e6,0,0};      /// - wavevector (normalized to \f$ 1/\lambda \f$)
00068     gauss1.p = {0,0,0.1};      /// - polarization/amplitude
00069     gauss1.x0 = {100e-6,0,0};    /// - shift from origin
00070     gauss1.phig = 5e-6;          /// - width
00071     gauss1.phi = {0,0,0};        /// - phase shift
00072     gaussian1D gauss2;          /// Another Gaussian with
00073     gauss2.k = {-0.2e6,0,0};      /// - wavevector (normalized to \f$ 1/\lambda \f$)
00074     gauss2.p = {0,0,0.5};      /// - polarization/amplitude
00075     gauss2.x0 = {200e-6,0,0};    /// - shift from origin
00076     gauss2.phig = 15e-6;         /// - width
00077     gauss2.phi = {0,0,0};        /// - phase shift
00078     // Do not comment out this vector, even if no Gaussian wave is used. But if, emplace used Gaussian
waves.
00079     vector<gaussian1D> Gaussians1D;
00080     Gaussians1D.emplace_back(gauss1);
00081     Gaussians1D.emplace_back(gauss2);
00082
00083     /// Do not change this below ///
00084     int *interactions = &processOrder;
00085     Sim1D(CVodeTolerances,StencilOrder,physical_sidelength,latticepoints,
periodic,interactions,simulationTime,numberOfSteps,
00087     outputDirectory,outputStep,outputStyle,
00088     planewaves,Gaussians1D);
00089
00090     //////////////////////////////////////
00091
00092     ////////////////////////////////////// -- 2D -- //////////////////////////////////////
00093     /** A 2D simulation with specified */
00094
00095     /// Specify your settings here ///
00096     constexpr array<sunrealtype,2> CVodeTolerances={1.0e-12,1.0e-12}; /// - relative and absolute
tolerances of the CVode solver
00098     constexpr int StencilOrder=13;                                     /// - accuracy order of the
stencils in the range 1-13
00099     constexpr array<sunrealtype,2> physical_sidelengths={80e-6,80e-6}; /// - physical length of the
lattice in the given dimensions in meters
00100     constexpr array<sunindextype,2> latticepoints_per_dim={800,800};   /// - number of lattice points
per dimension
00101     constexpr array<int,2> patches_per_dim={2,2};                     /// - slicing of discrete
dimensions into patches
00102     constexpr bool periodic=true;                                     /// - periodic or vanishing
boundary values
00103     int processOrder=3;                                              /// - included processes of the
weak-field expansion, see README.md
00104     constexpr sunrealtype simulationTime=4e-6l;                      /// - physical total simulation
time
00105     constexpr int numberOfSteps=10;                                  /// - discrete time steps
00106     constexpr int outputStep=1;                                       /// - output step multiples
00107     constexpr char outputStyle='c';                                   /// - output in csv (c) or binary
(b)
00108
00109     /// Add electromagnetic waves.
00110     planewave plane1;          /// A plane wave with
00111     plane1.k = {1e5,0,0};      /// - wavevector (normalized to \f$ 1/\lambda \f$)

```

```

00112     plane1.p = {0,0,0.1};          /// - amplitude/polarization
00113     plane1.phi = {0,0,0};          /// - phase shift
00114     planewave plane2;              /// Another plane wave with
00115     plane2.k = {-1e6,0,0};          /// - wavevector
00116     plane2.p = {0,0,0.5};          /// - amplitude/polarization
00117     plane2.phi = {0,0,0};          /// - phase shift
00118     // Do not comment out this vector, even if no plane wave is used. But if, emplace used plane
waves.
00119     vector<planewave> planewaves;
00120     //planewaves.emplace_back(plane1);
00121     //planewaves.emplace_back(plane2);
00122
00123     gaussian2D gauss1;              /// A Gaussian wave with
00124     gauss1.x0 = {40e-6,40e-6};      /// - center it approaches
00125     gauss1.axis = {1,0};             /// - normalized direction _from_ which the wave approaches the
center
00126     gauss1.amp = 0.5;                /// - amplitude
00127     gauss1.phip = 2*atan(0);          /// - polarization rotation from TE-mode (z-axis)
00128     gauss1.w0 = 2.3e-6;              /// - taille
00129     gauss1.zr = 16.619e-6;           /// - Rayleigh length
00130     /// the wavelength is determined by the relation \f$ \lambda = \pi w_0^2 / z_R \f$
00131     gauss1.ph0 = 2e-5;                /// - beam center
00132     gauss1.phA = 0.45e-5;            /// - beam length
00133     gaussian2D gauss2;              /// Another Gaussian wave with
00134     gauss2.x0 = {40e-6,40e-6};      /// - center it approaches
00135     gauss2.axis = {-0.7071,0.7071};  /// - normalized direction from which the wave approaches the
center
00136     gauss2.amp = 0.5;                /// - amplitude
00137     gauss2.phip = 2*atan(0);          /// - polarization rotation fom TE-mode (z-axis)
00138     gauss2.w0 = 2.3e-6;              /// - taille
00139     gauss2.zr = 16.619e-6;           /// - Rayleigh length
00140     gauss2.ph0 = 2e-5;                /// - beam center
00141     gauss2.phA = 0.45e-5;            /// - beam length
00142     // Do not comment out this vector, even if no Gaussian wave is used. But if, emplace used Gaussian
waves.
00143     vector<gaussian2D> Gaussians2D;
00144     Gaussians2D.emplace_back(gauss1);
00145     Gaussians2D.emplace_back(gauss2);
00146
00147     /// Do not change this below ///
00148     static_assert(latticepoints_per_dim[0]*patches_per_dim[0]==0 &&
00149         latticepoints_per_dim[1]*patches_per_dim[1]==0,
00150         "The number of lattice points in each dimension must be "
00151         "divisible by the number of patches in that direction.");
00152     int * interactions = &processOrder;
00153     Sim2D(CVodeTolerances,StencilOrder,physical_sidelengths,
00154         latticepoints_per_dim,patches_per_dim,periodic,interactions,
00155         simulationTime,numberOfSteps,outputDirectory,outputStep,
00156         outputStyle,planewaves,Gaussians2D);
00157
00158     //////////////////////////////////////
00159
00160
00161     ////////////////////////////////// -- 3D -- //////////////////////////////////
00162     /** A 3D simulation with specified */
00163
00164     /// Specify your settings here ///
00165     constexpr array<sunrealtype,2> CVodeTolerances={1.0e-12,1.0e-12};    /// - relative and
absolute tolerances of the CVode solver
00166     constexpr int StencilOrder=4;    /// - accuracy order of
the stencils in the range 1-13
00167     constexpr array<sunrealtype,3> physical_sidelengths={80e-6,80e-6,20e-6};    /// - physical dimensions
in meters
00168     constexpr array<sunindextype,3> latticepoints_per_dim={160,160,40};    /// - number of lattice
points in any dimension
00169     constexpr array<int,3> patches_per_dim= {2,2,1};    /// - slicing of discrete
dimensions into patches
00170     constexpr bool periodic=false;    /// - perodic or
non-periodic boundaries
00171     int processOrder=3;    /// - processes of the
weak-field expansion, see README.md
00172     constexpr sunrealtype simulationTime=2e-6;    /// - physical total
simulation time
00173     constexpr int numberOfSteps=5;    /// - discrete time steps
00174     constexpr int outputStep=1;    /// - output step
multiples
00175     constexpr char outputStyle='b';    /// - output in csv (c) or binary
(b)
00176     //
00177     /// Add electromagnetic waves.
00178     planewave plane1;    /// A plane wave with
00179     plane1.k = {1e5,0,0};    /// - wavevector (normalized to \f$ 1/\lambda \f$)
00180     plane1.p = {0,0,0.1};    /// - amplitude/polarization
00181     plane1.phi = {0,0,0};    /// - phase shift
00182     planewave plane2;    /// Another plane wave with
00183     plane2.k = {-1e6,0,0};    /// - wavevector (normalized to \f$ 1/\lambda \f$)

```

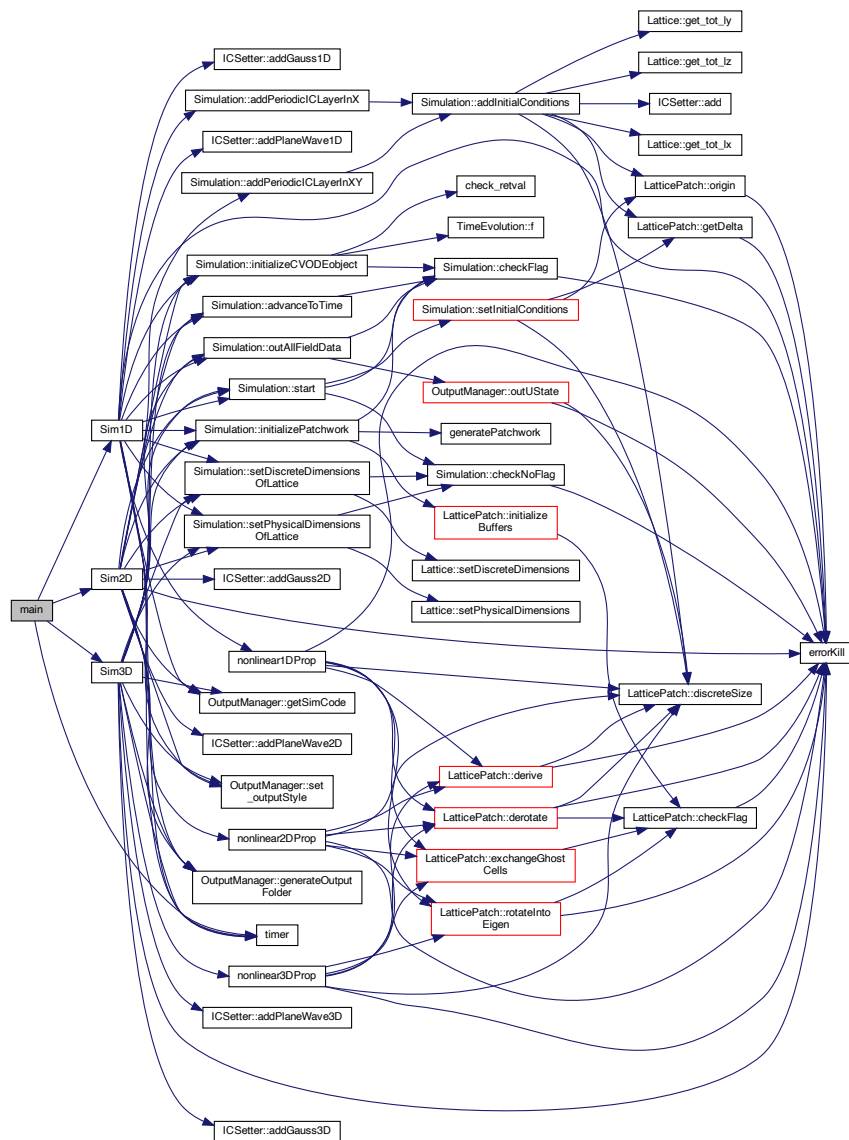
```

00184     plane2.p = {0,0,0.5};           /// - amplitude/polarization
00185     plane2.phi = {0,0,0};           /// - phase shift
00186     // Do not comment out this vector, even if no plane wave is used. But if, emplace used plane
waves.
00187     vector<planewave> planewaves;
00188     //planewaves.emplace_back(plane1);
00189     //planewaves.emplace_back(plane2);
00190
00191     gaussian3D gauss1;               /// A Gaussian wave with
00192     gauss1.x0 = {40e-6,40e-6,10e-6}; /// - center it approaches
00193     gauss1.axis = {1,0,0};           /// - normalized direction _from_ which the wave approaches
the center
00194     gauss1.amp = 0.05;               /// - amplitude
00195     gauss1.phip = 2*atan(0);          /// - polarization rotation from TE-mode (z-axis)
00196     gauss1.w0 = 3.5e-6;              /// - taille
00197     gauss1.zr = 19.242e-6;           /// - Rayleigh length
00198     /// the wavelength is determined by the relation \f$ \lambda = \pi*w_0^2/z_R \f$
00199     gauss1.ph0 = 2e-5;               /// - beam center
00200     gauss1.phA = 0.45e-5;            /// - beam length
00201     gaussian3D gauss2;               /// Another Gaussian wave with
00202     gauss2.x0 = {40e-6,40e-6,10e-6}; /// - center it approaches
00203     gauss2.axis = {0,1,0};           /// - normalized direction from which the wave approaches the
center
00204     gauss2.amp = 0.05;               /// - amplitude
00205     gauss2.phip = 2*atan(0);          /// - polarization rotation from TE-mode (z-axis)
00206     gauss2.w0 = 3.5e-6;              /// - taille
00207     gauss2.zr = 19.242e-6;           /// - Rayleigh length
00208     gauss2.ph0 = 2e-5;               /// - beam center
00209     gauss2.phA = 0.45e-5;            /// - beam length
00210     // Do not comment out this vector, even if no Gaussian wave is used. But if, emplace used Gaussian
waves.
00211     vector<gaussian3D> Gaussians3D;
00212     Gaussians3D.emplace_back(gauss1);
00213     Gaussians3D.emplace_back(gauss2);
00214
00215     /// Do not change this below ///
00216     static_assert(latticepoints_per_dim[0]*patches_per_dim[0]==0 &&
00217         latticepoints_per_dim[1]*patches_per_dim[1]==0 &&
00218         latticepoints_per_dim[2]*patches_per_dim[2]==0,
00219         "The number of lattice points in each dimension must be "
00220         "divisible by the number of patches in that direction.");
00221     int *interactions = &processOrder;
00222     Sim3D(CVodeTolerances,StencilOrder,physical_sidelengths,
00223         latticepoints_per_dim,patches_per_dim,periodic,interactions,
00224         simulationTime,numberOfSteps,outputDirectory,outputStep,
00225         outputStyle,planewaves,Gaussians3D);
00226
00227     //////////////////////////////////////
00228
00229     //----- END OF CONFIGURATION -----//
00230
00231     double tf=MPI_Wtime(); // Overall finish time
00232     if(rank==0) {cout<<endl; timer(ti,tf);} // Print the elapsed time
00233
00234     // Finalize MPI environment
00235     MPI_Finalize();
00236
00237     return 0;
00238 }

```

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 = "/gpfs/scratch/uh3o1/ru68dab/ru68dab/HE/outputs";
00026     //constexpr auto outputDirectory = "/home/andi/Documents/";
00027     constexpr auto outputDirectory = "/Users/andi/Documents/";
00028
00029     if(rank==0 && !filesystem::exists(outputDirectory)) {
00030         cerr<<"\nOutput directory nonexistent.\n";
00031         MPI_Abort(comm,1);
00032     }
00033
00034
00035     //----- BEGIN OF CONFIGURATION -----//
00036
00037     //////////////// -- 1D -- ////////////////
00038     /** A 1D simulation with specified */
00039
00040     /** Specify your settings here */
00041     constexpr array<sunrealtype,2> CNodeTolerances={1.0e-16,1.0e-16}; // - relative and absolute
00042     //constexpr int StencilOrder=13; // - accuracy order of the
00043     //stencils in the range 1-13
00044     constexpr sunrealtype physical_sidlength=300e-6; // - physical length of the
00045     //lattice in meters
00046     constexpr sunindextype latticepoints=6e3; // - number of lattice points
00047     constexpr bool periodic=true; // - periodic or vanishing
00048     //boundary values
00049     int processOrder=3; // - included processes of the
00050     //weak-field expansion, see README.md
00051     //constexpr sunrealtype simulationTime=100.0e-6l; // - physical total
00052     //simulation time
00053     constexpr int numberOfSteps=100; // - discrete time steps
00054     constexpr int outputStep=1; // - output step multiples
00055     constexpr char outputStyle='c'; // - output in csv (c) or binary
00056     (b)
00057
00058     /** Add electromagnetic waves.
00059     planewave planel; // A plane wave with
00060     planel.k = {1e5,0,0}; // - wavevector (normalized to \f$ 1/\lambda \f$)
00061     planel.p = {0,0,0.1}; // - amplitude/polarization
00062     planel.phi = {0,0,0}; // - phase shift
00063     planewave plane2; // Another plane wave with
00064     plane2.k = {-1e6,0,0}; // - wavevector (normalized to \f$ 1/\lambda \f$)
00065     plane2.p = {0,0,0.5}; // - amplitude/polarization
00066     plane2.phi = {0,0,0}; // - phase shift
00067     // Do not comment out this vector, even if no plane wave is used. But if, emplace used plane
00068     waves.
00069     vector<planewave> planewaves;
00070     //planewaves.emplace_back(planel);
00071     //planewaves.emplace_back(plane2);
00072
00073     gaussian1D gauss1; // A Gaussian wave with
00074     gauss1.k = {1.0e6,0,0}; // - wavevector (normalized to \f$ 1/\lambda \f$)
00075     gauss1.p = {0,0,0.1}; // - polarization/amplitude
00076     gauss1.x0 = {100e-6,0,0}; // - shift from origin
00077     gauss1.phig = 5e-6; // - width
00078     gauss1.phi = {0,0,0}; // - phase shift
00079     gaussian1D gauss2; // Another Gaussian with
00080     gauss2.k = {-0.2e6,0,0}; // - wavevector (normalized to \f$ 1/\lambda \f$)
00081     gauss2.p = {0,0,0.5}; // - polarization/amplitude
00082     gauss2.x0 = {200e-6,0,0}; // - shift from origin
00083     gauss2.phig = 15e-6; // - width
00084     gauss2.phi = {0,0,0}; // - phase shift
00085     // Do not comment out this vector, even if no Gaussian wave is used. But if, emplace used Gaussian
00086     waves.
00087     vector<gaussian1D> Gaussians1D;
00088     Gaussians1D.emplace_back(gauss1);
00089     Gaussians1D.emplace_back(gauss2);
00090
00091     /** Do not change this below */
00092     int *interactions = &processOrder;
00093     Sim1D(CNodeTolerances,StencilOrder,physical_sidlength,latticepoints,
00094           periodic,interactions,simulationTime,numberOfSteps,
00095           outputDirectory,outputStep,outputStyle,
00096           planewaves,Gaussians1D);
00097
00098     ////////////////
00099
00100     //////////////// -- 2D -- ////////////////
00101     /** A 2D simulation with specified */

```

```

00095
00096     /// Specify your settings here ///
00097     constexpr array<sunrealtype,2> CNodeTolerances={1.0e-12,1.0e-12}; /// - relative and absolute
tolerances of the CNode solver
00098     constexpr int StencilOrder=13; /// - accuracy order of the
stencils in the range 1-13
00099     constexpr array<sunrealtype,2> physical_sidelengths={80e-6,80e-6}; /// - physical length of the
lattice in the given dimensions in meters
00100     constexpr array<sunindextype,2> latticepoints_per_dim={800,800}; /// - number of lattice points
per dimension
00101     constexpr array<int,2> patches_per_dim={2,2}; /// - slicing of discrete
dimensions into patches
00102     constexpr bool periodic=true; /// - periodic or vanishing
boundary values
00103     int processOrder=3; /// - included processes of the
weak-field expansion, see README.md
00104     constexpr sunrealtype simulationTime=4e-6l; /// - physical total simulation
time
00105     constexpr int numberOfSteps=10; /// - discrete time steps
00106     constexpr int outputStep=1; /// - output step multiples
00107     constexpr char outputStyle='c'; /// - output in csv (c) or binary
(b)
00108
00109     /// Add electromagnetic waves.
00110     planewave plane1; /// A plane wave with
00111     plane1.k = {1e5,0,0}; /// - wavevector (normalized to  $\frac{1}{\lambda}$ )
00112     plane1.p = {0,0,0.1}; /// - amplitude/polarization
00113     plane1.phi = {0,0,0}; /// - phase shift
00114     planewave plane2; /// Another plane wave with
00115     plane2.k = {-1e6,0,0}; /// - wavevector
00116     plane2.p = {0,0,0.5}; /// - amplitude/polarization
00117     plane2.phi = {0,0,0}; /// - phase shift
00118     // Do not comment out this vector, even if no plane wave is used. But if, emplace used plane
waves.
00119     vector<planewave> planewaves;
00120     //planewaves.emplace_back(plane1);
00121     //planewaves.emplace_back(plane2);
00122
00123     gaussian2D gauss1; /// A Gaussian wave with
00124     gauss1.x0 = {40e-6,40e-6}; /// - center it approaches
00125     gauss1.axis = {1,0}; /// - normalized direction _from_ which the wave approaches the
center
00126     gauss1.amp = 0.5; /// - amplitude
00127     gauss1.phip = 2*atan(0); /// - polarization rotation from TE-mode (z-axis)
00128     gauss1.w0 = 2.3e-6; /// - taille
00129     gauss1.zr = 16.619e-6; /// - Rayleigh length
00130     /// the wavelength is determined by the relation  $\frac{1}{\lambda} = \frac{\pi w_0^2}{z_R}$ 
00131     gauss1.ph0 = 2e-5; /// - beam center
00132     gauss1.phA = 0.45e-5; /// - beam length
00133     gaussian2D gauss2; /// Another Gaussian wave with
00134     gauss2.x0 = {40e-6,40e-6}; /// - center it approaches
00135     gauss2.axis = {-0.7071,0.7071}; /// - normalized direction from which the wave approaches the
center
00136     gauss2.amp = 0.5; /// - amplitude
00137     gauss2.phip = 2*atan(0); /// - polarization rotation fom TE-mode (z-axis)
00138     gauss2.w0 = 2.3e-6; /// - taille
00139     gauss2.zr = 16.619e-6; /// - Rayleigh length
00140     gauss2.ph0 = 2e-5; /// - beam center
00141     gauss2.phA = 0.45e-5; /// - beam length
00142     // Do not comment out this vector, even if no Gaussian wave is used. But if, emplace used Gaussian
waves.
00143     vector<gaussian2D> Gaussians2D;
00144     Gaussians2D.emplace_back(gauss1);
00145     Gaussians2D.emplace_back(gauss2);
00146
00147     /// Do not change this below ///
00148     static_assert(latticepoints_per_dim[0]*patches_per_dim[0]==0 &&
latticepoints_per_dim[1]*patches_per_dim[1]==0,
00149         "The number of lattice points in each dimension must be "
00150         "divisible by the number of patches in that direction.");
00151     int * interactions = &processOrder;
00152     Sim2D(CNodeTolerances,StencilOrder,physical_sidelengths,
latticepoints_per_dim,patches_per_dim,periodic,interactions,
00153         simulationTime,numberOfSteps,outputDirectory,outputStep,
00154         outputStyle,planewaves,Gaussians2D);
00155
00156     //////////////////////////////////////
00157
00158     ////////////////////////////////////// -- 3D -- //////////////////////////////////////
00159
00160     /** A 3D simulation with specified */
00161
00162     /// Specify your settings here ///
00163
00164     constexpr array<sunrealtype,2> CNodeTolerances={1.0e-12,1.0e-12}; /// - relative and
absolute tolerances of the CNode solver
00165     constexpr int StencilOrder=4; /// - accuracy order of
the stencils in the range 1-13

```

```

00167     constexpr array<sunrealtype,3> physical_sidelengths={80e-6,80e-6,20e-6}; /// - physical dimensions
in meters
00168     constexpr array<sunindextype,3> latticepoints_per_dim={160,160,40};      /// - number of lattice
points in any dimension
00169     constexpr array<int,3> patches_per_dim= {2,2,1};                        /// - slicing of discrete
dimensions into patches
00170     constexpr bool periodic=false;                                          /// - perodic or
non-periodic boundaries
00171     int processOrder=3;                                                    /// - processes of the
weak-field expansion, see README.md
00172     constexpr sunrealtype simulationTime=2e-6;                             /// - physical total
simulation time
00173     constexpr int numberOfSteps=5;                                          /// - discrete time steps
00174     constexpr int outputStep=1;                                            /// - output step
multiples
00175     constexpr char outputStyle='b';                                        /// - output in csv (c) or binary
(b)
00176                                     //
00177     /// Add electromagnetic waves.
00178     planewave plane1;                                                       /// A plane wave with
00179     plane1.k = {1e5,0,0};                                                  /// - wavevector (normalized to \f$ 1/\lambda \f$)
00180     plane1.p = {0,0,0.1};                                                  /// - amplitude/polarization
00181     plane1.phi = {0,0,0};                                                  /// - phase shift
00182     planewave plane2;                                                     /// Another plane wave with
00183     plane2.k = {-1e6,0,0};                                                  /// - wavevector (normalized to \f$ 1/\lambda \f$)
00184     plane2.p = {0,0,0.5};                                                  /// - amplitude/polarization
00185     plane2.phi = {0,0,0};                                                  /// - phase shift
00186     // Do not comment out this vector, even if no plane wave is used. But if, emplace used plane
waves.
00187     vector<planewave> planewaves;
00188     //planewaves.emplace_back(plane1);
00189     //planewaves.emplace_back(plane2);
00190
00191     gaussian3D gauss1;                                                     /// A Gaussian wave with
00192     gauss1.x0 = {40e-6,40e-6,10e-6};                                       /// - center it approaches
00193     gauss1.axis = {1,0,0};                                                  /// - normalized direction _from_ which the wave approaches
the center
00194     gauss1.amp = 0.05;                                                      /// - amplitude
00195     gauss1.phip = 2*atan(0);                                                 /// - polarization rotation from TE-mode (z-axis)
00196     gauss1.w0 = 3.5e-6;                                                     /// - taille
00197     gauss1.zr = 19.242e-6;                                                  /// - Rayleigh length
00198     /// the wavelength is determined by the relation \f$ \lambda = \pi*w_0^2/z_R \f$
00199     gauss1.ph0 = 2e-5;                                                      /// - beam center
00200     gauss1.phA = 0.45e-5;                                                  /// - beam length
00201     gaussian3D gauss2;                                                     /// Another Gaussian wave with
00202     gauss2.x0 = {40e-6,40e-6,10e-6};                                       /// - center it approaches
00203     gauss2.axis = {0,1,0};                                                  /// - normalized direction from which the wave approaches the
center
00204     gauss2.amp = 0.05;                                                      /// - amplitude
00205     gauss2.phip = 2*atan(0);                                                 /// - polarization rotation from TE-mode (z-axis)
00206     gauss2.w0 = 3.5e-6;                                                     /// - taille
00207     gauss2.zr = 19.242e-6;                                                  /// - Rayleigh length
00208     gauss2.ph0 = 2e-5;                                                      /// - beam center
00209     gauss2.phA = 0.45e-5;                                                  /// - beam length
00210     // Do not comment out this vector, even if no Gaussian wave is used. But if, emplace used Gaussian
waves.
00211     vector<gaussian3D> Gaussians3D;
00212     Gaussians3D.emplace_back(gauss1);
00213     Gaussians3D.emplace_back(gauss2);
00214
00215     /// Do not change this below ///
00216     static_assert(latticepoints_per_dim[0]*patches_per_dim[0]==0 &&
00217         latticepoints_per_dim[1]*patches_per_dim[1]==0 &&
00218         latticepoints_per_dim[2]*patches_per_dim[2]==0,
00219         "The number of lattice points in each dimension must be "
00220         "divisible by the number of patches in that direction.");
00221     int *interactions = &processOrder;
00222     Sim3D(CVodeTolerances,StencilOrder,physical_sidelengths,
00223         latticepoints_per_dim,patches_per_dim,periodic,interactions,
00224         simulationTime,numberOfSteps,outputDirectory,outputStep,
00225         outputStyle,planewaves,Gaussians3D);
00226
00227     //////////////////////////////////////
00228
00229     //----- END OF CONFIGURATION -----//
00230
00231     double tf=MPI_Wtime(); // Overall finish time
00232     if(rank==0) {cout<<endl; timer(ti,tf);} // Print the elapsed time
00233
00234     // Finalize MPI environment
00235     MPI_Finalize();
00236
00237     return 0;
00238 }

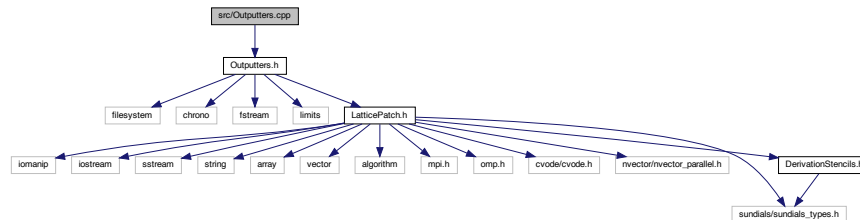
```

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     outputStyle = 'c';
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
00032     // too 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 direcorry "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     int myPrc;
00043     MPI_Comm_rank(MPI_COMM_WORLD, &myPrc);
00044     if (myPrc == 0) {
00045         if (!fs::is_directory(dir))
00046             fs::create_directory(dir);
00047         if (!fs::is_directory(dir + "/SimResults"))
00048             fs::create_directory(dir + "/SimResults");
00049         if (!fs::is_directory(dir + "/SimResults/" + simCode))
00050             fs::create_directory(dir + "/SimResults/" + simCode);
00051     }
00052     // path variable for the output generation
00053     Path = dir + "/SimResults/" + simCode + "/";
00054
00055     // Logging configurations from main.cpp
00056     ifstream fin("main.cpp");
00057     ofstream fout(Path + "config.txt");
00058     string line;
00059     int begin=1000;
00060     for (int i = 1; !fin.eof(); i++) {
00061         getline(fin, line);
00062         if (line.starts_with("    //----- B")) {
00063             begin=i;
00064         }
00065         if (i < begin) {
00066             continue;
00067         }
00068         fout << line << endl;
00069         if (line.starts_with("    //----- E")) {
00070             break;
00071         }
00072     }
00073     return;
00074 }
00075
00076 void OutputManager::set_outputStyle(const char _outputStyle){
00077     outputStyle = _outputStyle;
00078 }
00079
00080 /** Write the field data either in csv format to one file per each process
00081 * (patch) or in binary form to a single file. Files are stores inthe simCode
00082 * directory. For csv files the state (simulation step) denotes the
00083 * prefix and the suffix after an underscore is given by the process/patch
00084 * number. Binary files are simply named after the step number. */
00085 void OutputManager::outUState(const int &state, const Lattice &lattice,
00086     const LatticePatch &latticePatch) {
00087     switch(outputStyle){
00088         case 'c': { // one csv file per process
00089             ofstream ofs;
00090             ofs.open(Path + to_string(state) + "_" + to_string(lattice.my_prc) + ".csv");
00091             // Precision of sunrealtype in significant decimal digits; 15 for IEEE double
00092             ofs << setprecision(numeric_limits<sunrealtype>::digits10);
00093
00094             // Walk through each lattice point
00095             for (int i = 0; i < latticePatch.discreteSize() * 6; i += 6) {
00096                 // Six columns to contain the field data: Ex,Ey,Ez,Bx,By,Bz
00097                 ofs << latticePatch.uData[i + 0] << "," << latticePatch.uData[i + 1] << ","
00098                     << latticePatch.uData[i + 2] << "," << latticePatch.uData[i + 3] << ","
00099                     << latticePatch.uData[i + 4] << "," << latticePatch.uData[i + 5]
00100                     << endl;
00101             }
00102             ofs.close();
00103             break;
00104         }
00105         case 'b': { // a single binary file
00106             // Open the output file
00107             MPI_File fh;
00108             const string filename = Path+to_string(state);
00109             MPI_File_open(lattice.comm,&filename[0],MPI_MODE_WRONLY|MPI_MODE_CREATE,
00110                 MPI_INFO_NULL,&fh);
00111             // number of datapoints in the patch with process offset
00112             const int count = latticePatch.discreteSize()*lattice.get_dataPointDimension();
00113             MPI_Offset offset = lattice.my_prc*count*sizeof(MPI_SUNREALTYPE);
00114             // Go to offset in file and write data to it; maximal precision in
00115             // "native" representation
00116             MPI_File_set_view(fh,offset,MPI_SUNREALTYPE,MPI_SUNREALTYPE,"native",
00117                 MPI_INFO_NULL);
00118             MPI_File_write_all(fh,latticePatch.uData,count,MPI_SUNREALTYPE,MPI_STATUS_IGNORE);
00119             /*
00120             MPI_Request write_request;
00121             MPI_File_iwrite_all(fh,latticePatch.uData,count,MPI_SUNREALTYPE,&write_request);
00122             MPI_Wait(&write_request,MPI_STATUS_IGNORE);
00123             */

```

```

00124  */
00125  /*
00126  MPI_File_write_at_all(fh,offset,latticePatch.uData,count,MPI_SUNREALTYPE,
00127                      MPI_STATUS_IGNORE);
00128  */
00129  break;
00130  }
00131  default: {
00132  errorKill("No valid output style defined.\
00133          Choose between (c): one csv file per process, (b) one binary file");
00134  break;
00135  }
00136  return;
00137  }
00138  }
00139

```

6.18 src/Outputters.h File Reference

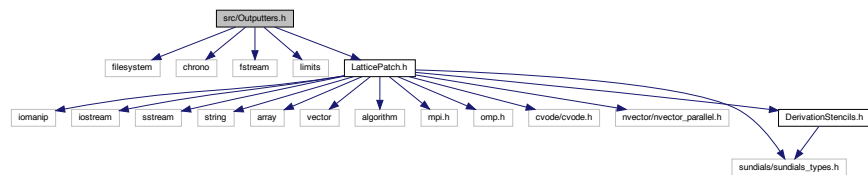
[OutputManager](#) class to outstream simulation data.

```

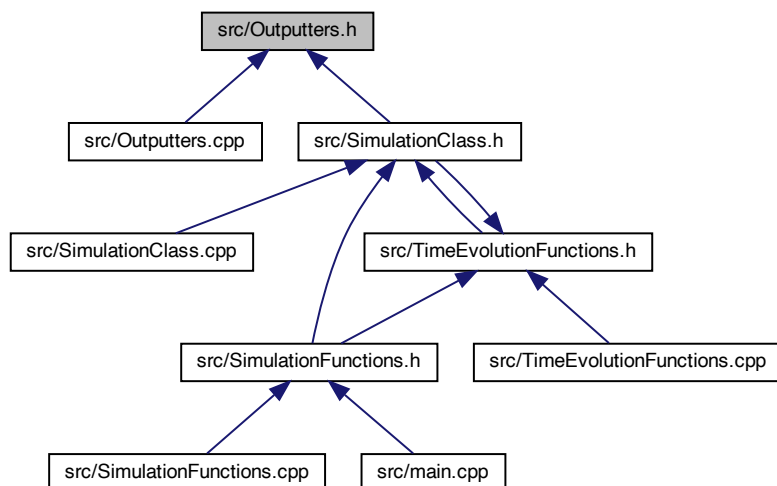
#include <filesystem>
#include <chrono>
#include <fstream>
#include <limits>
#include "LatticePatch.h"

```

Include dependency graph for Outputters.h:



This graph shows which files directly or indirectly include this file:



Data Structures

- class [OutputManager](#)

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

6.18.1 Detailed Description

[OutputManager](#) class to outstream simulation data.

Definition in file [Outputters.h](#).

6.19 Outputters.h

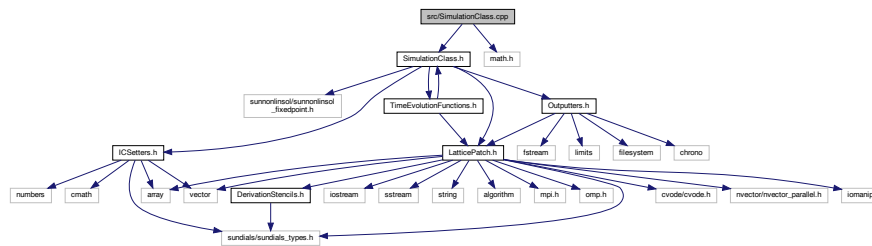
[Go to the documentation of this file.](#)

```
00001 ///////////////////////////////////////////////////////////////////
00002 /// @file Outputters.h
00003 /// @brief OutputManager class to outstream simulation data
00004 ///////////////////////////////////////////////////////////////////
00005
00006 #pragma once
00007
00008 // perform operations on the filesystem
00009 #include <filesystem>
00010
00011 // output controlling with limits and timestep
00012 #include <chrono>
00013 #include <fstream>
00014 #include <limits>
00015
00016 // project subfile header
00017 #include "LatticePatch.h"
00018
00019 using namespace std;
00020 namespace fs = std::filesystem;
00021 namespace chrono = std::chrono;
00022
00023 /** @brief Output Manager class to generate and coordinate output writing to
00024  * disk */
00025 class OutputManager {
00026 private:
00027     /// function to create the Code of the Simulations
00028     static string SimCodeGenerator();
00029     /// variable to save the SimCode generated at execution
00030     string simCode;
00031     /// variable for the path to the output folder
00032     string Path;
00033     /// output style; csv or binary
00034     char outputStyle;
00035 public:
00036     /// default constructor
00037     OutputManager();
00038     /// function that creates folder to save simulation data
00039     void generateOutputFolder(const string &dir);
00040     /// set the output style
00041     void set_outputStyle(const char _outputStyle);
00042     /// function to write data to disk in specified way
00043     void outUState(const int &state, const Lattice &lattice, const LatticePatch &latticePatch);
00044     /// simCode getter function
00045     [[nodiscard]] const string &getSimCode() const { return simCode; }
00046 };
00047
```

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 CVode solver memory and Sundials context object with the finish of
00033 /// the simulation
00034 Simulation::~Simulation() {
00035     // Free solver memory
00036     if (statusFlags & CvsolveObjectSetup) {
00037         // PrintFinalStats(cvsolve_mem); // TODO write this function as in cvsolve
00038         // cvAdvDiff_bnd.c SUNDIALS_MARK_FUNCTION_END(lattice.profbj);
00039         CvsolveFree(&cvsolve_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 lenghts 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 CVMODE
00074 void Simulation::initializeCVMODEObject(const sunrealtype reltol,
00075         const sunrealtype abstol) {
00076     checkFlag(SimulationStarted);
00077
00078     // CVMODE 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 CVMODE object -- returns a pointer to the cvmode memory structure
00092     // with Adams method (Adams-Moulton formula) solver chosen for non-stiff ODE
00093     cvmode_mem = CVMODECreate(CV_ADAMS, lattice.sunctx);
00094
00095     // Specify user data and attach it to the main cvmode memory block
00096     retval = CVMODESetUserData(
00097         cvmode_mem,
00098         &latticePatch); // patch contains the user data as used in CVRhsFn
00099     if (check_retval(&retval, "CVMODESetUserData", 1, lattice.my_prc))
00100         MPI_Abort(lattice.comm, 1);
00101     // if (flag != CV_SUCCESS) { printf("CVMODESetUserData failed, flag=%d.\n",
00102     // flag);
00103     //     MPI_Abort(lattice.comm, 1); }
00104
00105     // Initialize CVMODE 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 cvmode
00108     retval = CVMODEInit(cvmode_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, "CVMODEInit", 1, lattice.my_prc))
00112         MPI_Abort(lattice.comm, 1);
00113     // if (flag != CV_SUCCESS) { printf("CVMODEInit 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 CVMODE
00118     SUNNonlinearSolver NLS =
00119         SUNNonlinSol_FixedPoint(latticePatch.u, 0, lattice.sunctx);
00120     retval = CVMODESetNonlinearSolver(cvmode_mem, NLS);
00121     if (check_retval(&retval, "CVMODESetNonlinearSolver", 1, lattice.my_prc))
00122         MPI_Abort(lattice.comm, 1);
00123     // if (flag != CV_SUCCESS) { printf("CVMODESetNonlinearSolver 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 = CVMODESetMaxNumSteps(cvmode_mem, 10000);
00130     if (check_retval(&retval, "CVMODESetMaxNumSteps", 1, lattice.my_prc))

```

```

00131     MPI_Abort(lattice.comm, 1);
00132     // if (flag != CV_SUCCESS) { printf("CvodeSetMaxNumSteps 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 = CvodeSStolerances(cvode_mem, reltol, abstol);
00139     if (check_retval(&retval, "CvodeSStolerances", 1, lattice.my_prc))
00140         MPI_Abort(lattice.comm, 1);
00141     // if (flag != CV_SUCCESS) { printf("CvodeSStolerances failed, flag=%d.\n",
00142     // flag);
00143     //     MPI_Abort(lattice.comm, 1); }
00144
00145     statusFlags |= CvodeObjectSetUp;
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(CvodeObjectSetUp);
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 = CVode(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("CVode 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, lattice, 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

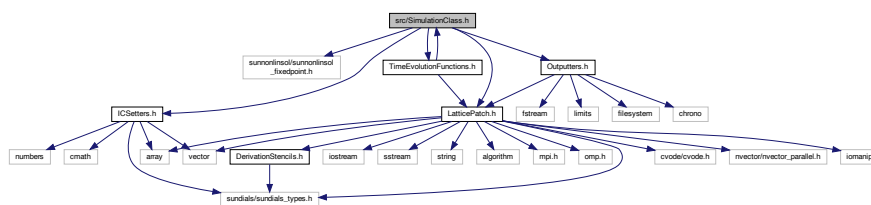
Class for the [Simulation](#) object calling all functionality: from wave settings over lattice construction, time evolution and outputs initialization of the CNode object.

```

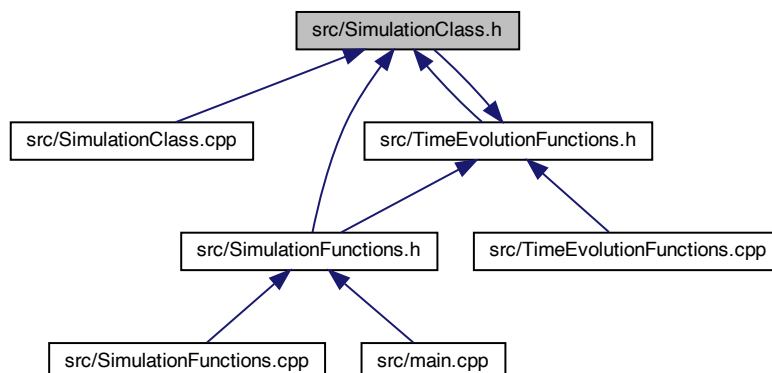
#include "sunnonlinsol/sunnonlinsol_fixedpoint.h"
#include "ICSetters.h"
#include "LatticePatch.h"
#include "Outputters.h"
#include "TimeEvolutionFunctions.h"

```

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 22 of file [SimulationClass.h](#).

```
00022 {
00023     LatticeDiscreteSetUp = 0x01,
00024     LatticePhysicalSetUp = 0x02,
00025     LatticePatchworkSetUp = 0x04, // not used anymore
00026     CvodeObjectSetUp = 0x08,
00027     SimulationStarted = 0x10
00028     /*OPT_B = 0x02,
00029     OPT_C = 0x04,
00030     OPT_D = 0x08,
00031     OPT_E = 0x10,
00032     OPT_F = 0x20,*/
00033 };
```

6.23 SimulationClass.h

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

```
00001 //////////////////////////////////////
```

```

00002 /// @file SimulationClass.h
00003 /// @brief Class for the Simulation object calling all functionality:
00004 /// from wave settings over lattice construction, time evolution and outputs
00005 /// initialization of the CNode object
00006 ///////////////////////////////////////////////////////////////////
00007
00008 #pragma once
00009
00010 /* access to the fixed point SUNNonlinearSolver */
00011 #include "sunnonlinsol/sunnonlinsol_fixedpoint.h"
00012
00013 // project subfile headers
00014 #include "ICSetters.h"
00015 #include "LatticePatch.h"
00016 #include "Outputters.h"
00017 #include "TimeEvolutionFunctions.h"
00018
00019 using namespace std;
00020
00021 /// simulation checking flags
00022 enum SimulationOptions {
00023     LatticeDiscreteSetUp = 0x01,
00024     LatticePhysicalSetUp = 0x02,
00025     LatticePatchworkSetUp = 0x04, // not used anymore
00026     CnodeObjectSetUp = 0x08,
00027     SimulationStarted = 0x10
00028     /*OPT_B = 0x02,
00029     OPT_C = 0x04,
00030     OPT_D = 0x08,
00031     OPT_E = 0x10,
00032     OPT_F = 0x20,*/
00033 };
00034
00035 /** @brief Simulation class to instantiate the whole walkthrough of a Simulation
00036 */
00037 class Simulation {
00038 private:
00039     /// Lattice object
00040     Lattice lattice;
00041     /// LatticePatch object
00042     LatticePatch latticePatch;
00043     /// current time of the simulation
00044     sunrealtype t;
00045     /// char for checking simulation flags
00046     unsigned char statusFlags;
00047
00048 public:
00049     /// IC Setter object
00050     ICSetter icSettings;
00051     /// Output Manager object
00052     OutputManager outputManager;
00053     /// Pointer to CNode memory object -- public to avoid cross library errors
00054     void *cnode_mem;
00055     /// constructor function for the creation of the cartesian communicator
00056     Simulation(const int nx, const int ny, const int nz, const int StencilOrder,
00057               const bool periodicity);
00058     /// destructor function freeing CNode memory and Sundials context
00059     ~Simulation();
00060     /// Reference to the cartesian communicator of the lattice -> for debugging
00061     MPI_Comm *get_cart_comm() { return &lattice.comm; };
00062     /// function to set discrete dimensions of the lattice
00063     void setDiscreteDimensionsOfLattice(const sunindextype _tot_nx,
00064                                         const sunindextype _tot_ny, const sunindextype _tot_nz);
00065     /// function to set physical dimensions of the lattice
00066     void setPhysicalDimensionsOfLattice(const sunrealtype lx, const sunrealtype ly,
00067                                         const sunrealtype lz);
00068     /// function to initialize the Patchwork
00069     void initializePatchwork(const int nx, const int ny, const int nz);
00070     /// function to initialize the CVODE object with all requirements
00071     void initializeCVODEobject(const sunrealtype reltol,
00072                               const sunrealtype abstol);
00073     /// function to start the simulation for time iteration
00074     void start();
00075     /// functions to set the initial field configuration onto the lattice
00076     void setInitialConditions();
00077     /// functions to add initial periodic field configurations
00078     void addInitialConditions(const int xm, const int ym, const int zm = 0);
00079     /// function to add a periodic IC Layer in one dimension
00080     void addPeriodicICLayerInX();
00081     /// function to add periodic IC Layers in two dimensions
00082     void addPeriodicICLayerInXY();
00083     /// function to advance solution in time with CVODE
00084     void advanceToTime(const sunrealtype &tEnd);
00085     /// function to generate Output of the whole field at a given time
00086     void outAllFieldData(const int &state);
00087     /// function to check that a flag has been set and if not print an error
00088     // message and cause an abort on all ranks

```

```

00089 void checkFlag(unsigned int flag) const;
00090 /// function to check that if flag has not been set and if print an error
00091 // message and cause an abort on all ranks
00092 void checkNoFlag(unsigned int flag) const;
00093 };
00094

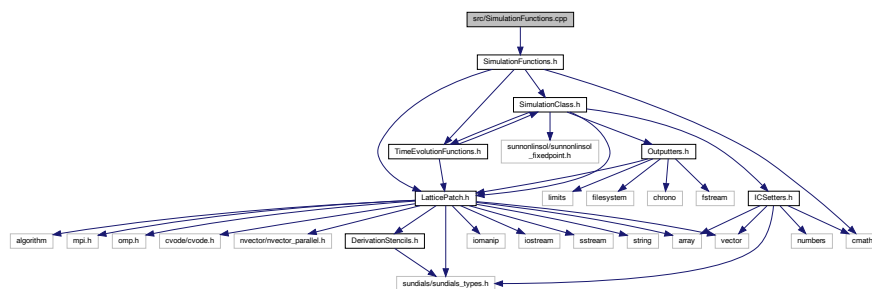
```

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 char outputStyle, 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 char outputStyle, 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 char outputStyle, 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 > 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 char outputStyle,
    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](#).

```
00030 {
00031
00032 // MPI data
00033 int myPrc = 0, nprc = 0;
00034 MPI_Comm_size(MPI_COMM_WORLD, &nprc);
00035 MPI_Comm_rank(MPI_COMM_WORLD, &myPrc);
00036
00037 // Check feasibility of the patchwork decomposition
00038 if (myPrc == 0) {
00039     if (disc_dim % nprc != 0) {
00040         errorKill("The number of lattice points must be "
00041                 "divisible by the number of processes.");
00042     }
00043 }
00044
00045 // Initialize the simulation, set up the cartesian communicator
00046 array<int, 3> patches = {nprc, 1, 1};
00047 Simulation sim(patches[0], patches[1], patches[2], StencilOrder, periodic);
00048
00049 // Configure the patchwork
00050 sim.setPhysicalDimensionsOfLattice(phys_dim, 1, 1);
00051 sim.setDiscreteDimensionsOfLattice(disc_dim, 1, 1);
00052 sim.initializePatchwork(patches[0], patches[1], patches[2]);
00053
00054 // Add em-waves
00055 for (const auto &gauss : gaussians)
00056     sim.icsettings.addGauss1D(gauss.k, gauss.p, gauss.x0, gauss.phig,
00057                             gauss.phi);
00058 for (const auto &plane : planes)
00059     sim.icsettings.addPlaneWave1D(plane.k, plane.p, plane.phi);
00060
00061 // Check that the patchwork is ready and set the initial conditions
00062 sim.start();
00063 sim.addPeriodicICLayerInX();
00064
00065 // Initialize CNode with abs and rel tolerances
00066 sim.initializeCNodeObject(CNodeTol[0], CNodeTol[1]);
00067
00068 // Configure the time evolution function
00069 TimeEvolution::c = interactions;
00070 TimeEvolution::TimeEvolver = nonlinear1DProp;
00071
00072 // Configure the output
00073 sim.outputManager.generateOutputFolder(outputDirectory);
00074 if (!myPrc) {
```



```

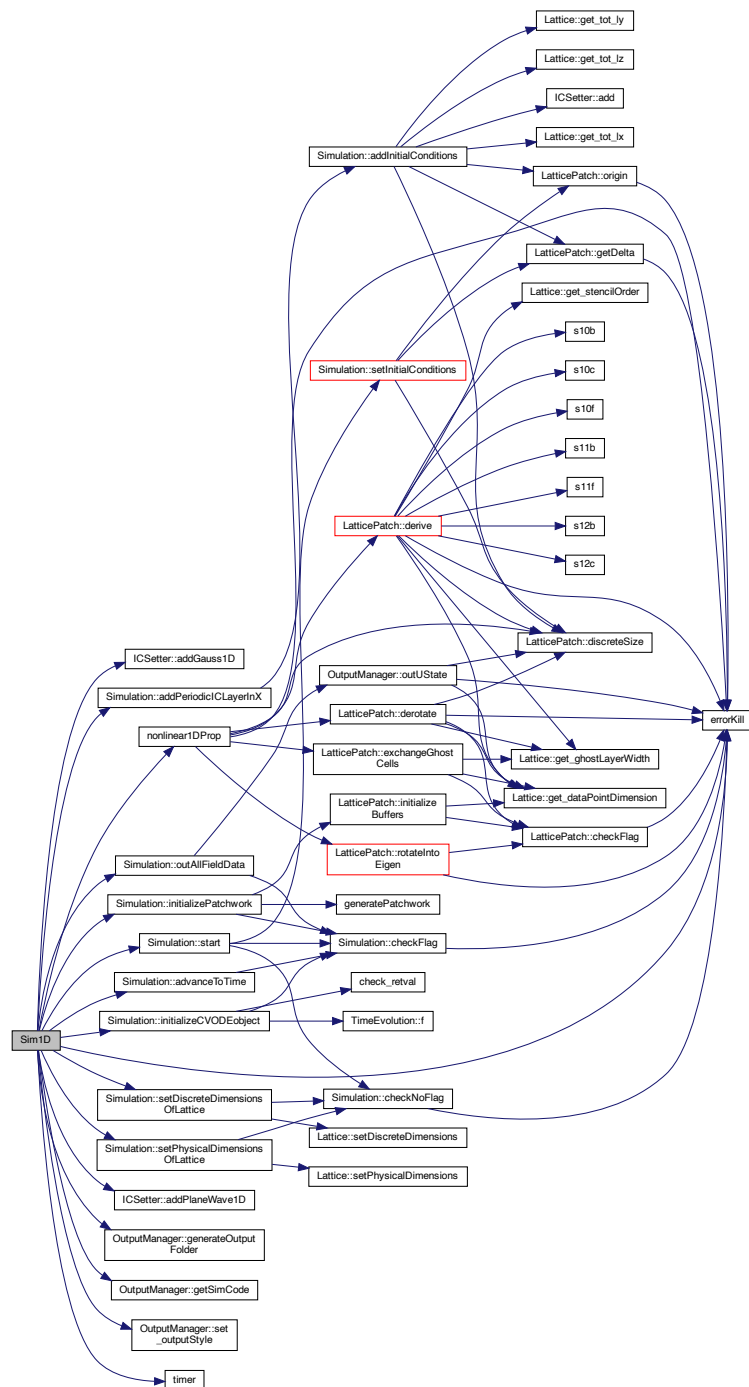
00075     cout << "Simulation code: " << sim.outputManager.getSimCode() << endl;
00076 }
00077 sim.outputManager.set_outputStyle(outputStyle);
00078
00079 double ts = MPI_Wtime();
00080
00081 //sim.outAllFieldData(0); // output of initial state
00082 // Conduct the propagation in space and time
00083 for (int step = 1; step <= numberOfSteps; step++) {
00084     sim.advanceToTime(endTime / numberOfSteps * step);
00085     if (step % outputStep == 0) {
00086         sim.outAllFieldData(step);
00087     }
00088     double tn = MPI_Wtime();
00089     if (!myPrc) {
00090         cout << "\rStep " << step << "\t\t" << flush;
00091         timer(ts, tn);
00092     }
00093 }
00094
00095 return;
00096 }

```

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

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

Here is the call graph for this function:



Here is the caller graph for this function:



6.24.2.2 Sim2D()

```

void Sim2D (
    const 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 char outputStyle,
    const vector< planewave > & planes,
    const vector< gaussian2D > & gaussians )
  
```

complete 2D [Simulation](#) function

Conduct the complete 2D simulation process

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

```

00104                                     {
00105
00106     // MPI data
00107     int myPrc = 0, nprc = 0; // Get process rank and number of processes
00108     MPI_Comm_rank(MPI_COMM_WORLD,
00109                   &myPrc); // Return process rank, number \in [1,nprc]
00110     MPI_Comm_size(MPI_COMM_WORLD,
00111                   &nprc); // Return number of processes (communicator size)
00112
00113     // Check feasibility of the patchwork decomposition
00114     if (myPrc == 0) {
00115         if (nprc != patches[0] * patches[1]) {
00116             errorKill(
00117                 "The number of MPI processes must match the number of patches.");
00118         }
00119     }
00120
00121     // Initialize the simulation, set up the cartesian communicator
00122     Simulation sim(patches[0], patches[1], 1, StencilOrder, periodic);
00123
00124     /* // Check that lattice communicator is unique; same as used in patchwork
00125     generation char cart_comm_name[MPI_MAX_OBJECT_NAME]; int cart_namelen;
00126     MPI_Comm_get_name(*sim.get_cart_comm(), cart_comm_name, &cart_namelen);
00127     printf("sim.get_cart_comm gives %s \n", cart_comm_name);
00128     */
00129
00130     // Configure the patchwork
00131     sim.setPhysicalDimensionsOfLattice(phys_dims[0],
  
```

```

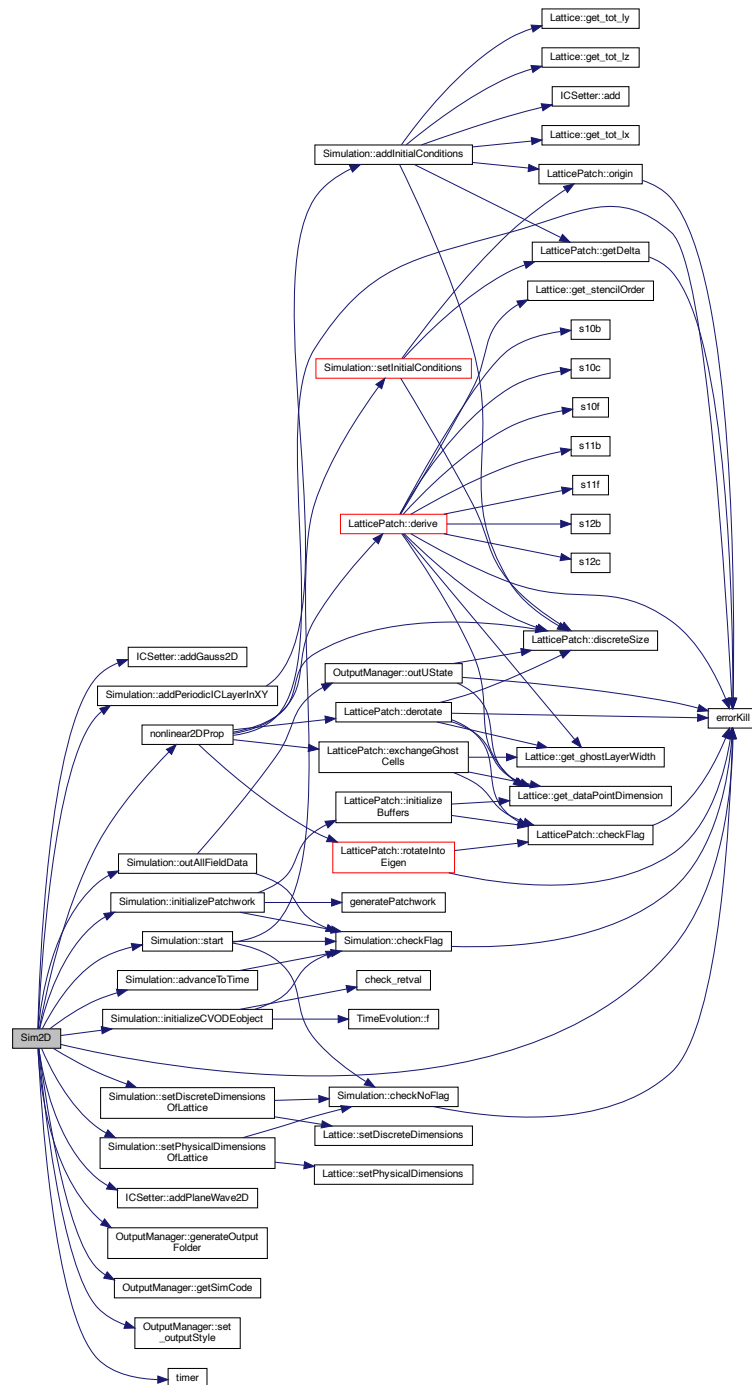
00132                                     phys_dims[1],
00133                                     1); // spacing of the lattice
00134     sim.setDiscreteDimensionsOfLattice(
00135         disc_dims[0], disc_dims[1], 1); // Spacing equivalence to points
00136     sim.initializePatchwork(patches[0], patches[1], 1);
00137
00138     // Add em-waves
00139     for (const auto &gauss : gaussians)
00140         sim.icsettings.addGauss2D(gauss.x0, gauss.axis, gauss.amp, gauss.phip,
00141             gauss.w0, gauss.zr, gauss.ph0, gauss.phA);
00142     for (const auto &plane : planes)
00143         sim.icsettings.addPlaneWave2D(plane.k, plane.p, plane.phi);
00144
00145     // Check that the patchwork is ready and set the initial conditions
00146     sim.start(); // Check if the lattice is set up, set initial field
00147                 // configuration
00148     sim.addPeriodicICLayerInXY(); // insure periodicity in propagation directions
00149
00150     // Initialize CNode with rel and abs tolerances
00151     sim.initializeCVODEobject(CNodeTol[0], CNodeTol[1]);
00152
00153     // Configure the time evolution function
00154     TimeEvolution::c = interactions;
00155     TimeEvolution::TimeEvolver = nonlinear2DProp;
00156
00157     // Configure the output
00158     sim.outputManager.generateOutputFolder(outputDirectory);
00159     if (!myPrc) {
00160         cout << "Simulation code: " << sim.outputManager.getSimCode() << endl;
00161     }
00162     sim.outputManager.set_outputStyle(outputStyle);
00163
00164     double ts = MPI_Wtime();
00165
00166     //sim.outAllFieldData(0); // output of initial state
00167     // Conduct the propagation in space and time
00168     for (int step = 1; step <= numberOfSteps; step++) {
00169         sim.advanceToTime(endTime / numberOfSteps * step);
00170         if (step % outputStep == 0) {
00171             sim.outAllFieldData(step);
00172         }
00173         double tn = MPI_Wtime();
00174         if (!myPrc) {
00175             cout << "\rStep " << step << "\t\t" << flush;
00176             timer(ts, tn);
00177         }
00178     }
00179
00180     return;
00181 }

```

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

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

Here is the call graph for this function:



Here is the caller graph for this function:



6.24.2.3 Sim3D()

```

void Sim3D (
    const 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 char outputStyle,
    const vector< planewave > & planes,
    const vector< gaussian3D > & gaussians )
  
```

complete 3D [Simulation](#) function

Conduct the complete 3D simulation process

Definition at line 184 of file [SimulationFunctions.cpp](#).

```

00190
00191
00192 // MPI data
00193 int myPrc = 0, nprc = 0; // Get process rank and number of process
00194 MPI_Comm_rank(MPI_COMM_WORLD,
00195               &myPrc); // rank of the process inside the world communicator
00196 MPI_Comm_size(MPI_COMM_WORLD,
00197               &nprc); // Size of the communicator is the number of processes
00198
00199 // Check feasibility of the patchwork decomposition
00200 if (myPrc == 0) {
00201     if (nprc != patches[0] * patches[1] * patches[2]) {
00202         errorKill(
00203             "The number of MPI processes must match the number of patches.");
00204     }
00205     if ( ( disc_dims[0] / patches[0] != disc_dims[1] / patches[1] ) |
00206         ( disc_dims[0] / patches[0] != disc_dims[2] / patches[2] ) ) {
00207         clog
00208             « "\nWarning: Patches should be cubic in terms of the lattice "
00209             "points for the computational efficiency of larger simulations.\n";
00210     }
00211 }
00212
00213 // Initialize the simulation, set up the cartesian communicator
00214 Simulation sim(patches[0], patches[1], patches[2],
00215               StencilOrder, periodic); // Simulation object with slicing
00216
00217 // Create the SUNContext object associated with the thread of execution
  
```

```

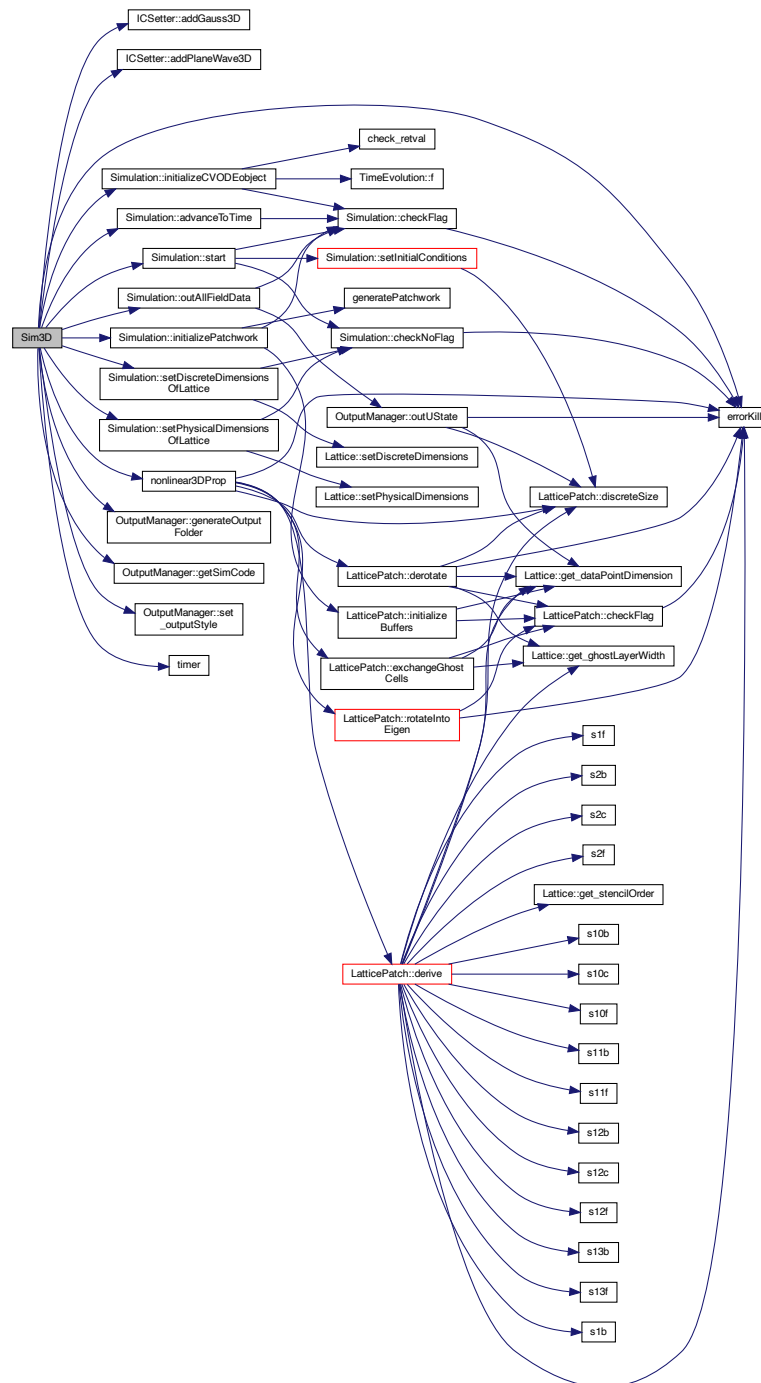
00218     sim.setPhysicalDimensionsOfLattice(phys_dims[0], phys_dims[1],
00219                                       phys_dims[2]); // spacing of the box
00220     sim.setDiscreteDimensionsOfLattice(
00221         disc_dims[0], disc_dims[1],
00222         disc_dims[2]); // Spacing equivalence to points
00223     sim.initializePatchwork(patches[0], patches[1], patches[2]);
00224
00225     // Add em-waves
00226     for (const auto &plane : planes)
00227         sim.icsettings.addPlaneWave3D(plane.k, plane.p, plane.phi);
00228     for (const auto &gauss : gaussians)
00229         sim.icsettings.addGauss3D(gauss.x0, gauss.axis, gauss.amp, gauss.phip,
00230                                   gauss.w0, gauss.zr, gauss.ph0, gauss.phA);
00231
00232     // Check that the patchwork is ready and set the initial conditions
00233     sim.start();
00234
00235     // Initialize CNode with abs and rel tolerances
00236     sim.initializeCNodeObject(CNodeTol[0], CNodeTol[1]);
00237
00238     // Configure the time evolution function
00239     TimeEvolution::c = interactions;
00240     TimeEvolution::TimeEvolver = nonlinear3DProp;
00241
00242     // Configure the output
00243     sim.outputManager.generateOutputFolder(outputDirectory);
00244     if (!myProc) {
00245         cout << "Simulation code: " << sim.outputManager.getSimCode() << endl;
00246     }
00247     sim.outputManager.set_outputStyle(outputStyle);
00248
00249     double ts = MPI_Wtime();
00250
00251     //sim.outAllFieldData(0); // output of initial state
00252     // Conduct the propagation in space and time
00253     for (int step = 1; step <= numberOfSteps; step++) {
00254         sim.advanceToTime(endTime / numberOfSteps * step);
00255         if (step % outputStep == 0) {
00256             sim.outAllFieldData(step);
00257         }
00258         double tn = MPI_Wtime();
00259         if (!myProc) {
00260             cout << "\rStep " << step << "\t\t" << flush;
00261             timer(ts, tn);
00262         }
00263     }
00264     return;
00265 }

```

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

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

Here is the call graph for this function:



Here is the caller graph for this function:



6.24.2.4 timer()

```
void timer (  
    double & t1,  
    double & t2 )
```

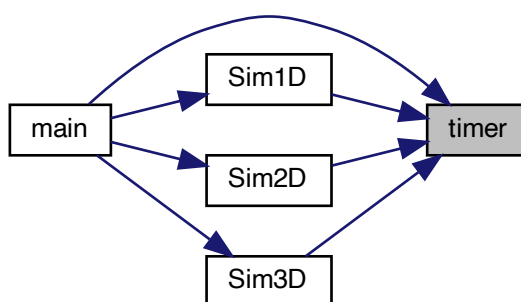
Calculate and print the total simulation time

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<sunrealttype,2> CNodeTol, const int StencilOrder,
00024           const sunrealttype phys_dim, const sunindextype disc_dim,
00025           const bool periodic, int *interactions,
00026           const sunrealttype endTime, const int numberOfSteps,
00027           const string outputDirectory, const int outputStep,
00028           const char outputStyle,
00029           const vector<planewave> &planes,
00030           const vector<gaussian1D> &gaussians) {
00031
00032     // MPI data
00033     int myPrc = 0, nprc = 0;
00034     MPI_Comm_size(MPI_COMM_WORLD, &nprc);
00035     MPI_Comm_rank(MPI_COMM_WORLD, &myPrc);
00036
00037     // Check feasibility of the patchwork decomposition
00038     if (myPrc == 0) {
00039         if (disc_dim % nprc != 0) {
00040             errorKill("The number of lattice points must be "
00041                     "divisible by the number of processes.");
00042         }
00043     }
00044
00045     // Initialize the simulation, set up the cartesian communicator
00046     array<int, 3> patches = {nprc, 1, 1};
00047     Simulation sim(patches[0], patches[1], patches[2], StencilOrder, periodic);
00048
00049     // Configure the patchwork
00050     sim.setPhysicalDimensionsOfLattice(phys_dim, 1, 1);
00051     sim.setDiscreteDimensionsOfLattice(disc_dim, 1, 1);
00052     sim.initializePatchwork(patches[0], patches[1], patches[2]);
00053
00054     // Add em-waves
00055     for (const auto &gauss : gaussians)
00056         sim.lcsettings.addGauss1D(gauss.k, gauss.p, gauss.x0, gauss.phig,
00057                                   gauss.phi);
00058     for (const auto &plane : planes)
00059         sim.lcsettings.addPlaneWave1D(plane.k, plane.p, plane.phi);
00060
00061     // Check that the patchwork is ready and set the initial conditions
00062     sim.start();
00063     sim.addPeriodicICLayerInX();
00064
00065     // Initialize CNode with abs and rel tolerances
00066     sim.initializeCNodeObject(CNodeTol[0], CNodeTol[1]);
00067
00068     // Configure the time evolution function
00069     TimeEvolution::c = interactions;
00070     TimeEvolution::TimeEvolver = nonlinear1DProp;
00071
00072     // Configure the output
00073     sim.outputManager.generateOutputFolder(outputDirectory);
00074     if (!myPrc) {
00075         cout << "Simulation code: " << sim.outputManager.getSimCode() << endl;
00076     }
00077     sim.outputManager.set_outputStyle(outputStyle);
00078
00079     double ts = MPI_Wtime();
00080
00081     //sim.outAllFieldData(0); // output of initial state
00082     // Conduct the propagation in space and time

```

```

00083     for (int step = 1; step <= numberOfSteps; step++) {
00084         sim.advanceToTime(endTime / numberOfSteps * step);
00085         if (step % outputStep == 0) {
00086             sim.outAllFieldData(step);
00087         }
00088         double tn = MPI_Wtime();
00089         if (!myPrc) {
00090             cout << "\rStep " << step << "\t\t" << flush;
00091             timer(ts, tn);
00092         }
00093     }
00094
00095     return;
00096 }
00097
00098 /** Conduct the complete 2D simulation process */
00099 void Sim2D(const array<sunrealtype,2> CNodeTol, int const StencilOrder,
00100           const array<sunrealtype,2> phys_dims, const array<sunindextype,2> disc_dims,
00101           const array<int,2> patches, const bool periodic, int *interactions,
00102           const sunrealtype endTime, const int numberOfSteps,
00103           const string outputDirectory, const int outputStep, const char outputStyle,
00104           const vector<planewave> &planes, const vector<gaussian2D> &gaussians) {
00105
00106     // MPI data
00107     int myPrc = 0, nprc = 0; // Get process rank and number of processes
00108     MPI_Comm_rank(MPI_COMM_WORLD,
00109                   &myPrc); // Return process rank, number \in [1,nprc]
00110     MPI_Comm_size(MPI_COMM_WORLD,
00111                   &nprc); // Return number of processes (communicator size)
00112
00113     // Check feasibility of the patchwork decomposition
00114     if (myPrc == 0) {
00115         if (nprc != patches[0] * patches[1]) {
00116             errorKill(
00117                 "The number of MPI processes must match the number of patches.");
00118         }
00119     }
00120
00121     // Initialize the simulation, set up the cartesian communicator
00122     Simulation sim(patches[0], patches[1], 1, StencilOrder, periodic);
00123
00124     /* // Check that lattice communicator is unique; same as used in patchwork
00125     generation char cart_comm_name[MPI_MAX_OBJECT_NAME]; int cart_namelen;
00126     MPI_Comm_get_name(*sim.get_cart_comm(), cart_comm_name, &cart_namelen);
00127     printf("sim.get_cart_comm gives %s \n", cart_comm_name);
00128     */
00129
00130     // Configure the patchwork
00131     sim.setPhysicalDimensionsOfLattice(phys_dims[0],
00132                                       phys_dims[1],
00133                                       1); // spacing of the lattice
00134     sim.setDiscreteDimensionsOfLattice(
00135         disc_dims[0], disc_dims[1], 1); // Spacing equivalence to points
00136     sim.initializePatchwork(patches[0], patches[1], 1);
00137
00138     // Add em-waves
00139     for (const auto &gauss : gaussians)
00140         sim.icsettings.addGauss2D(gauss.x0, gauss.axis, gauss.amp, gauss.phip,
00141                                   gauss.w0, gauss.zr, gauss.ph0, gauss.phA);
00142     for (const auto &plane : planes)
00143         sim.icsettings.addPlaneWave2D(plane.k, plane.p, plane.phi);
00144
00145     // Check that the patchwork is ready and set the initial conditions
00146     sim.start(); // Check if the lattice is set up, set initial field
00147                // configuration
00148     sim.addPeriodicICLayerInXY(); // insure periodicity in propagation directions
00149
00150     // Initialize CNode with rel and abs tolerances
00151     sim.initializeCNodeObject(CNodeTol[0], CNodeTol[1]);
00152
00153     // Configure the time evolution function
00154     TimeEvolution::c = interactions;
00155     TimeEvolution::TimeEvolver = nonlinear2DProp;
00156
00157     // Configure the output
00158     sim.outputManager.generateOutputFolder(outputDirectory);
00159     if (!myPrc) {
00160         cout << "Simulation code: " << sim.outputManager.getSimCode() << endl;
00161     }
00162     sim.outputManager.set_outputStyle(outputStyle);
00163
00164     double ts = MPI_Wtime();
00165
00166     //sim.outAllFieldData(0); // output of initial state
00167     // Conduct the propagation in space and time
00168     for (int step = 1; step <= numberOfSteps; step++) {
00169         sim.advanceToTime(endTime / numberOfSteps * step);

```

```

00170     if (step % outputStep == 0) {
00171         sim.outAllFieldData(step);
00172     }
00173     double tn = MPI_Wtime();
00174     if (!myPrc) {
00175         cout << "\rStep " << step << "\t\t" << flush;
00176         timer(ts, tn);
00177     }
00178 }
00179
00180 return;
00181 }
00182
00183 /** Conduct the complete 3D simulation process */
00184 void Sim3D(const array<sunrealtype,2> CNodeTol, const int StencilOrder,
00185           const array<sunrealtype,3> phys_dims,
00186           const array<sunindextype,3> disc_dims, const array<int,3> patches,
00187           const bool periodic, int *interactions, const sunrealtype endTime,
00188           const int numberOfSteps, const string outputDirectory,
00189           const int outputStep, const char outputStyle,
00190           const vector<planewave> &planes, const vector<gaussian3D> &gaussians) {
00191
00192     // MPI data
00193     int myPrc = 0, nprc = 0; // Get process rank and number of process
00194     MPI_Comm_rank(MPI_COMM_WORLD,
00195                   &myPrc); // rank of the process inside the world communicator
00196     MPI_Comm_size(MPI_COMM_WORLD,
00197                   &nprc); // Size of the communicator is the number of processes
00198
00199     // Check feasibility of the patchwork decomposition
00200     if (myPrc == 0) {
00201         if (nprc != patches[0] * patches[1] * patches[2]) {
00202             errorKill(
00203                 "The number of MPI processes must match the number of patches.");
00204         }
00205         if ( ( disc_dims[0] / patches[0] != disc_dims[1] / patches[1] ) |
00206             ( disc_dims[0] / patches[0] != disc_dims[2] / patches[2] ) ) {
00207             clog
00208                 << "\nWarning: Patches should be cubic in terms of the lattice "
00209                 << "points for the computational efficiency of larger simulations.\n";
00210         }
00211     }
00212
00213     // Initialize the simulation, set up the cartesian communicator
00214     Simulation sim(patches[0], patches[1], patches[2],
00215                   StencilOrder, periodic); // Simulation object with slicing
00216
00217     // Create the SUNContext object associated with the thread of execution
00218     sim.setPhysicalDimensionsOfLattice(phys_dims[0], phys_dims[1],
00219                                       phys_dims[2]); // spacing of the box
00220     sim.setDiscreteDimensionsOfLattice(
00221         disc_dims[0], disc_dims[1],
00222         disc_dims[2]); // Spacing equivalence to points
00223     sim.initializePatchwork(patches[0], patches[1], patches[2]);
00224
00225     // Add em-waves
00226     for (const auto &plane : planes)
00227         sim.icsettings.addPlaneWave3D(plane.k, plane.p, plane.phi);
00228     for (const auto &gauss : gaussians)
00229         sim.icsettings.addGauss3D(gauss.x0, gauss.axis, gauss.amp, gauss.phip,
00230                                   gauss.w0, gauss.zr, gauss.ph0, gauss.phA);
00231
00232     // Check that the patchwork is ready and set the initial conditions
00233     sim.start();
00234
00235     // Initialize CNode with abs and rel tolerances
00236     sim.initializeCNodeObject(CNodeTol[0], CNodeTol[1]);
00237
00238     // Configure the time evolution function
00239     TimeEvolution::c = interactions;
00240     TimeEvolution::TimeEvolver = nonlinear3DProp;
00241
00242     // Configure the output
00243     sim.outputManager.generateOutputFolder(outputDirectory);
00244     if (!myPrc) {
00245         cout << "Simulation code: " << sim.outputManager.getSimCode() << endl;
00246     }
00247     sim.outputManager.set_outputStyle(outputStyle);
00248
00249     double ts = MPI_Wtime();
00250
00251     //sim.outAllFieldData(0); // output of initial state
00252     // Conduct the propagation in space and time
00253     for (int step = 1; step <= numberOfSteps; step++) {
00254         sim.advanceToTime(endTime / numberOfSteps * step);
00255         if (step % outputStep == 0) {
00256             sim.outAllFieldData(step);

```


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 char, 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 char, 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 char, 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 char outputStyle,
    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](#).

```
00030 {
00031
00032     // MPI data
00033     int myPrc = 0, nprc = 0;
00034     MPI_Comm_size(MPI_COMM_WORLD, &nprc);
00035     MPI_Comm_rank(MPI_COMM_WORLD, &myPrc);
```

```

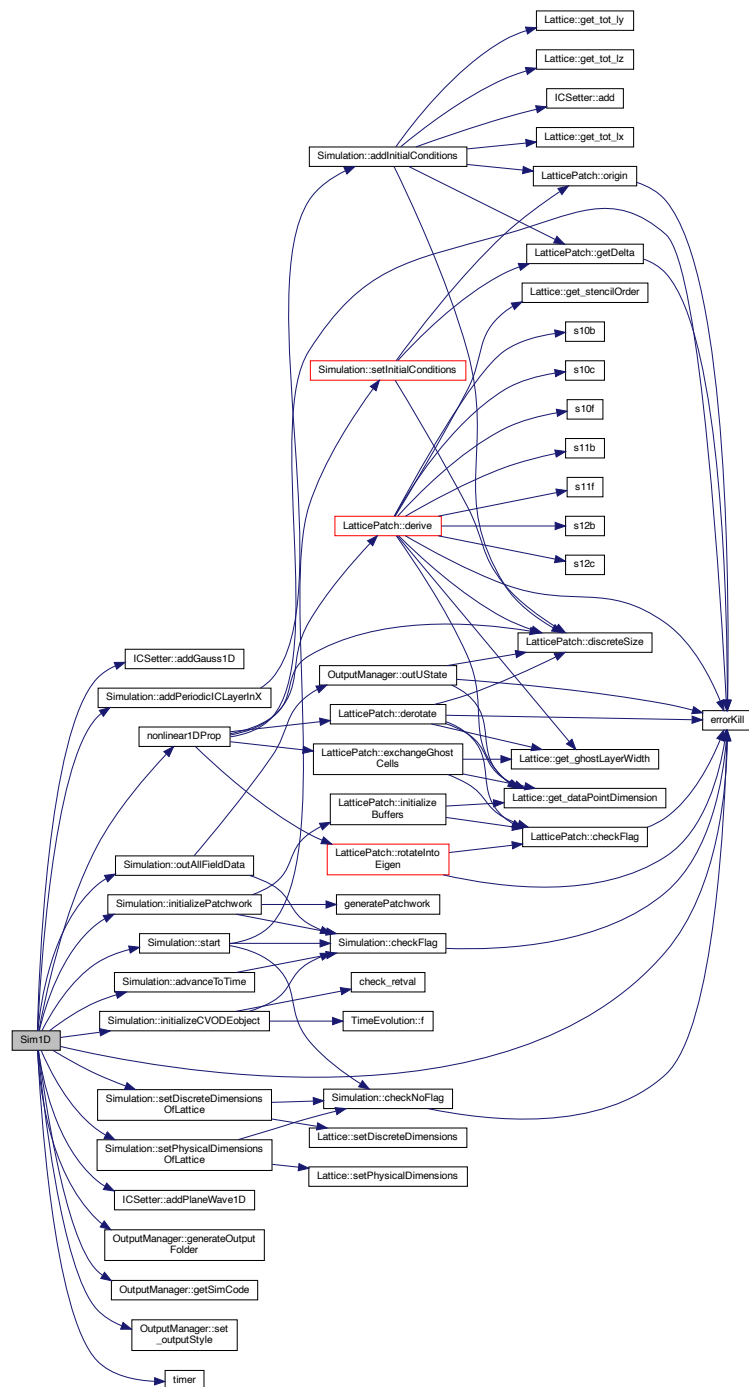
00036
00037 // Check feasibility of the patchwork decomposition
00038 if (myPrc == 0) {
00039     if (disc_dim % nprc != 0) {
00040         errorKill("The number of lattice points must be "
00041             "divisible by the number of processes.");
00042     }
00043 }
00044
00045 // Initialize the simulation, set up the cartesian communicator
00046 array<int, 3> patches = {nprc, 1, 1};
00047 Simulation sim(patches[0], patches[1], patches[2], StencilOrder, periodic);
00048
00049 // Configure the patchwork
00050 sim.setPhysicalDimensionsOfLattice(phys_dim,1,1);
00051 sim.setDiscreteDimensionsOfLattice(disc_dim,1,1);
00052 sim.initializePatchwork(patches[0], patches[1], patches[2]);
00053
00054 // Add em-waves
00055 for (const auto &gauss : gaussians)
00056     sim.icsettings.addGauss1D(gauss.k, gauss.p, gauss.x0, gauss.phig,
00057         gauss.phi);
00058 for (const auto &plane : planes)
00059     sim.icsettings.addPlaneWave1D(plane.k, plane.p, plane.phi);
00060
00061 // Check that the patchwork is ready and set the initial conditions
00062 sim.start();
00063 sim.addPeriodicICLayerInX();
00064
00065 // Initialize CNode with abs and rel tolerances
00066 sim.initializeCNodeObject(CNodeTol[0], CNodeTol[1]);
00067
00068 // Configure the time evolution function
00069 TimeEvolution::c = interactions;
00070 TimeEvolution::TimeEvolver = nonlinear1DProp;
00071
00072 // Configure the output
00073 sim.outputManager.generateOutputFolder(outputDirectory);
00074 if (!myPrc) {
00075     cout << "Simulation code: " << sim.outputManager.getSimCode() << endl;
00076 }
00077 sim.outputManager.set_outputStyle(outputStyle);
00078
00079 double ts = MPI_Wtime();
00080
00081 //sim.outAllFieldData(0); // output of initial state
00082 // Conduct the propagation in space and time
00083 for (int step = 1; step <= numberOfSteps; step++) {
00084     sim.advanceToTime(endTime / numberOfSteps * step);
00085     if (step % outputStep == 0) {
00086         sim.outAllFieldData(step);
00087     }
00088     double tn = MPI_Wtime();
00089     if (!myPrc) {
00090         cout << "\rStep " << step << "\t\t" << flush;
00091         timer(ts, tn);
00092     }
00093 }
00094
00095 return;
00096 }

```

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

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 char outputStyle,
    const vector< planewave > & planes,
    const vector< gaussian2D > & gaussians )
  
```

complete 2D [Simulation](#) function

Conduct the complete 2D simulation process

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

```

00104                                     {
00105
00106     // MPI data
00107     int myPrc = 0, nprc = 0; // Get process rank and number of processes
00108     MPI_Comm_rank(MPI_COMM_WORLD,
00109                   &myPrc); // Return process rank, number \in [1,nprc]
00110     MPI_Comm_size(MPI_COMM_WORLD,
00111                   &nprc); // Return number of processes (communicator size)
00112
00113     // Check feasibility of the patchwork decomposition
00114     if (myPrc == 0) {
00115         if (nprc != patches[0] * patches[1]) {
00116             errorKill(
00117                 "The number of MPI processes must match the number of patches.");
00118         }
00119     }
00120
00121     // Initialize the simulation, set up the cartesian communicator
00122     Simulation sim(patches[0], patches[1], 1, StencilOrder, periodic);
00123
00124     /* // Check that lattice communicator is unique; same as used in patchwork
00125     generation char cart_comm_name[MPI_MAX_OBJECT_NAME]; int cart_namelen;
00126     MPI_Comm_get_name(*sim.get_cart_comm(), cart_comm_name, &cart_namelen);
00127     printf("sim.get_cart_comm gives %s \n", cart_comm_name);
00128     */
00129
00130     // Configure the patchwork
00131     sim.setPhysicalDimensionsOfLattice(phys_dims[0],
  
```

```

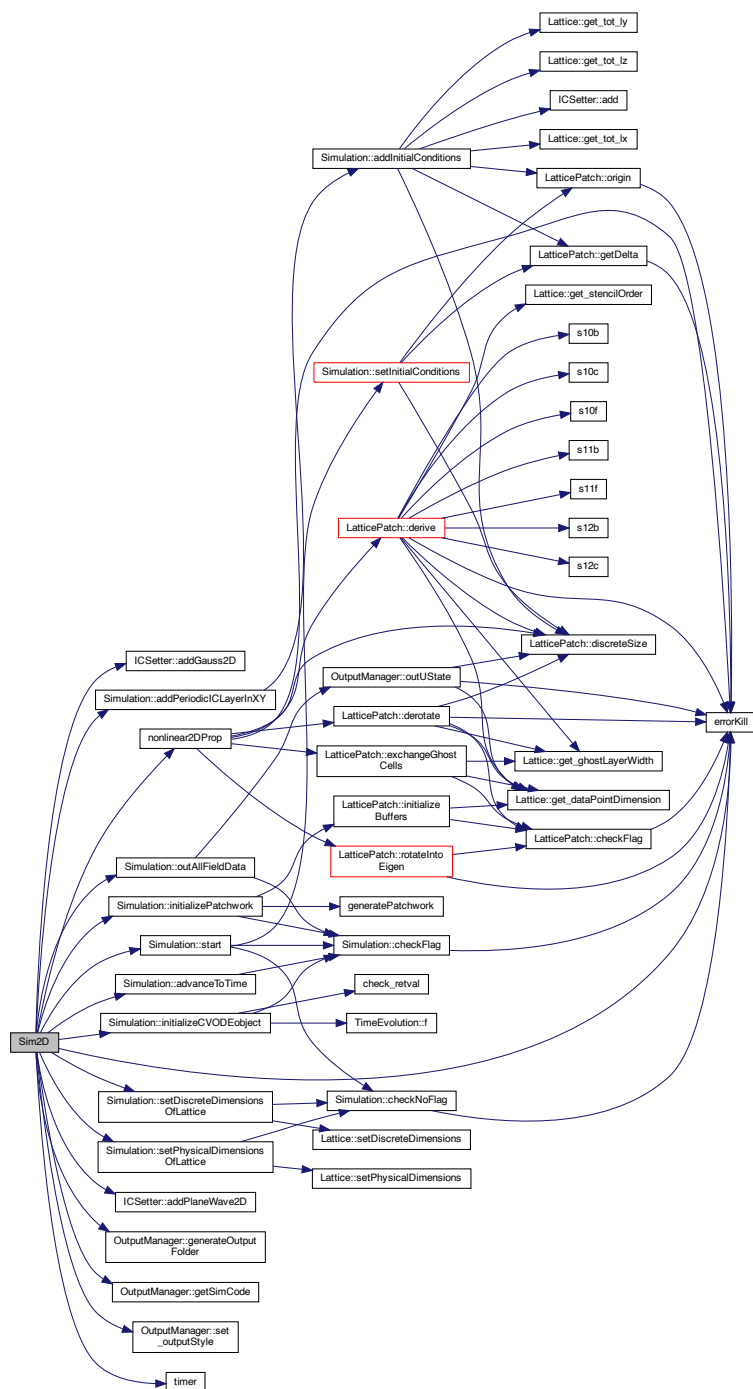
00132                                     phys_dims[1],
00133                                     1); // spacing of the lattice
00134     sim.setDiscreteDimensionsOfLattice(
00135         disc_dims[0], disc_dims[1], 1); // Spacing equivalence to points
00136     sim.initializePatchwork(patches[0], patches[1], 1);
00137
00138     // Add em-waves
00139     for (const auto &gauss : gaussians)
00140         sim.icsettings.addGauss2D(gauss.x0, gauss.axis, gauss.amp, gauss.phip,
00141             gauss.w0, gauss.zr, gauss.ph0, gauss.phA);
00142     for (const auto &plane : planes)
00143         sim.icsettings.addPlaneWave2D(plane.k, plane.p, plane.phi);
00144
00145     // Check that the patchwork is ready and set the initial conditions
00146     sim.start(); // Check if the lattice is set up, set initial field
00147                 // configuration
00148     sim.addPeriodicICLayerInXY(); // insure periodicity in propagation directions
00149
00150     // Initialize CNode with rel and abs tolerances
00151     sim.initializeCVODEobject(CNodeTol[0], CNodeTol[1]);
00152
00153     // Configure the time evolution function
00154     TimeEvolution::c = interactions;
00155     TimeEvolution::TimeEvolver = nonlinear2DProp;
00156
00157     // Configure the output
00158     sim.outputManager.generateOutputFolder(outputDirectory);
00159     if (!myPrc) {
00160         cout << "Simulation code: " << sim.outputManager.getSimCode() << endl;
00161     }
00162     sim.outputManager.set_outputStyle(outputStyle);
00163
00164     double ts = MPI_Wtime();
00165
00166     //sim.outAllFieldData(0); // output of initial state
00167     // Conduct the propagation in space and time
00168     for (int step = 1; step <= numberOfSteps; step++) {
00169         sim.advanceToTime(endTime / numberOfSteps * step);
00170         if (step % outputStep == 0) {
00171             sim.outAllFieldData(step);
00172         }
00173         double tn = MPI_Wtime();
00174         if (!myPrc) {
00175             cout << "\rStep " << step << "\t\t" << flush;
00176             timer(ts, tn);
00177         }
00178     }
00179
00180     return;
00181 }

```

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

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

Here is the call graph for this function:



Here is the caller graph for this function:



6.26.2.3 Sim3D()

```

void Sim3D (
    const 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 char outputStyle,
    const vector< planewave > & planes,
    const vector< gaussian3D > & gaussians )
  
```

complete 3D [Simulation](#) function

Conduct the complete 3D simulation process

Definition at line 184 of file [SimulationFunctions.cpp](#).

```

00190
00191
00192 // MPI data
00193 int myPrc = 0, nprc = 0; // Get process rank and number of process
00194 MPI_Comm_rank(MPI_COMM_WORLD,
00195               &myPrc); // rank of the process inside the world communicator
00196 MPI_Comm_size(MPI_COMM_WORLD,
00197               &nprc); // Size of the communicator is the number of processes
00198
00199 // Check feasibility of the patchwork decomposition
00200 if (myPrc == 0) {
00201     if (nprc != patches[0] * patches[1] * patches[2]) {
00202         errorKill(
00203             "The number of MPI processes must match the number of patches.");
00204     }
00205     if ( ( disc_dims[0] / patches[0] != disc_dims[1] / patches[1] ) |
00206         ( disc_dims[0] / patches[0] != disc_dims[2] / patches[2] ) ) {
00207         clog
00208             « "\nWarning: Patches should be cubic in terms of the lattice "
00209             "points for the computational efficiency of larger simulations.\n";
00210     }
00211 }
00212
00213 // Initialize the simulation, set up the cartesian communicator
00214 Simulation sim(patches[0], patches[1], patches[2],
00215               StencilOrder, periodic); // Simulation object with slicing
00216
00217 // Create the SUNContext object associated with the thread of execution
  
```

```

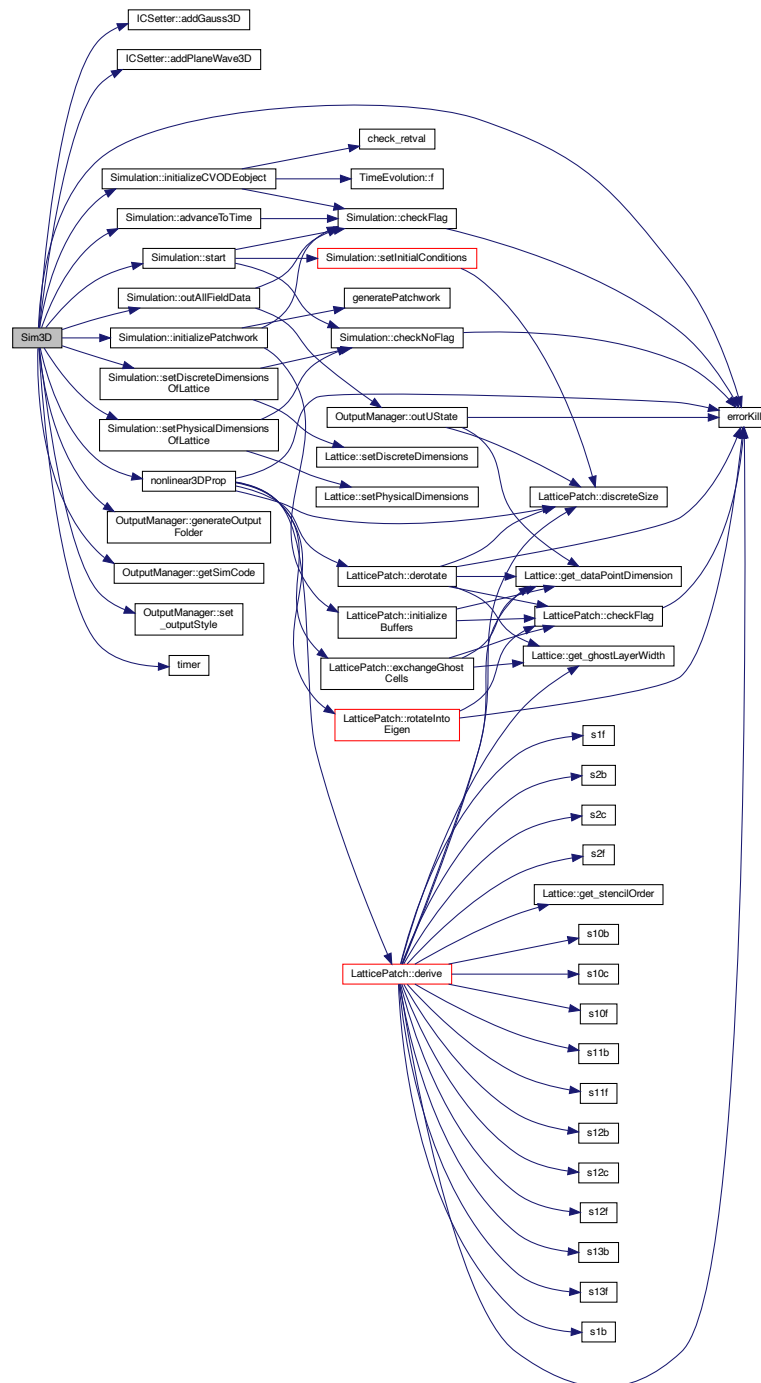
00218     sim.setPhysicalDimensionsOfLattice(phys_dims[0], phys_dims[1],
00219                                       phys_dims[2]); // spacing of the box
00220     sim.setDiscreteDimensionsOfLattice(
00221         disc_dims[0], disc_dims[1],
00222         disc_dims[2]); // Spacing equivalence to points
00223     sim.initializePatchwork(patchwork[0], patchwork[1], patchwork[2]);
00224
00225     // Add em-waves
00226     for (const auto &plane : planes)
00227         sim.icsettings.addPlaneWave3D(plane.k, plane.p, plane.phi);
00228     for (const auto &gauss : gaussians)
00229         sim.icsettings.addGauss3D(gauss.x0, gauss.axis, gauss.amp, gauss.phip,
00230                                   gauss.w0, gauss.zr, gauss.ph0, gauss.phA);
00231
00232     // Check that the patchwork is ready and set the initial conditions
00233     sim.start();
00234
00235     // Initialize CNode with abs and rel tolerances
00236     sim.initializeCNodeObject(CNodeTol[0], CNodeTol[1]);
00237
00238     // Configure the time evolution function
00239     TimeEvolution::c = interactions;
00240     TimeEvolution::TimeEvolver = nonlinear3DProp;
00241
00242     // Configure the output
00243     sim.outputManager.generateOutputFolder(outputDirectory);
00244     if (!myProc) {
00245         cout << "Simulation code: " << sim.outputManager.getSimCode() << endl;
00246     }
00247     sim.outputManager.set_outputStyle(outputStyle);
00248
00249     double ts = MPI_Wtime();
00250
00251     //sim.outAllFieldData(0); // output of initial state
00252     // Conduct the propagation in space and time
00253     for (int step = 1; step <= numberOfSteps; step++) {
00254         sim.advanceToTime(endTime / numberOfSteps * step);
00255         if (step % outputStep == 0) {
00256             sim.outAllFieldData(step);
00257         }
00258         double tn = MPI_Wtime();
00259         if (!myProc) {
00260             cout << "\rStep " << step << "\t\t" << flush;
00261             timer(ts, tn);
00262         }
00263     }
00264     return;
00265 }

```

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

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

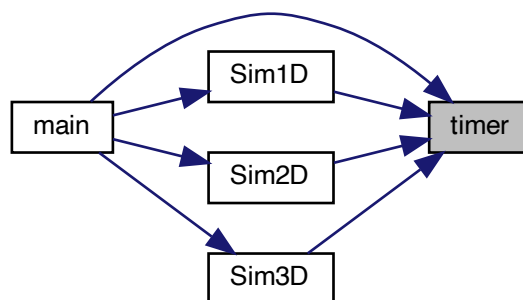
Calculate and print the total simulation time

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 #pragma once
00007
00008 // math
00009 #include <cmath>
00010 // #include <mathimf.h>
00011
00012 // project subfile headers
00013 #include "LatticePatch.h"
00014 #include "SimulationClass.h"
00015 #include "TimeEvolutionFunctions.h"
00016
00017 /***** EM-wave structures *****/
00018
00019 /// plane wave structure
00020 struct planewave {
00021     vector<sunrealtype> k; /**< wavevector (normalized to \f$ 1/\lambda \f$) */
00022     vector<sunrealtype> p; /**< amplitude & polarization vector */
00023     vector<sunrealtype> phi; /**< phase shift */
00024 };
00025
00026 /// 1D Gaussian wave structure
00027 struct gaussian1D {
00028     vector<sunrealtype> k; /**< wavevector (normalized to \f$ 1/\lambda \f$) */
00029     vector<sunrealtype> p; /**< amplitude & polarization vector */
00030     vector<sunrealtype> x0; /**< shift from origin */
00031     sunrealtype phig; /**< width */
00032     vector<sunrealtype> phi; /**< phase shift */
00033 };
00034
00035 /// 2D Gaussian wave structure
00036 struct gaussian2D {
00037     vector<sunrealtype> x0; /**< center */
00038     vector<sunrealtype> axis; /**< direction to center */
00039     sunrealtype amp; /**< amplitude */
00040     sunrealtype phip; /**< polarization rotation */
00041     sunrealtype w0; /**< taille */
00042     sunrealtype zr; /**< Rayleigh length */
00043     sunrealtype ph0; /**< beam center */
00044     sunrealtype phA; /**< beam length */
00045 };
00046
00047 /// 3D Gaussian wave structure
00048 struct gaussian3D {
00049     vector<sunrealtype> x0; /**< center */
00050     vector<sunrealtype> axis; /**< direction to center */
00051     sunrealtype amp; /**< amplitude */
00052     sunrealtype phip; /**< polarization rotation */
00053     sunrealtype w0; /**< taille */
00054     sunrealtype zr; /**< Rayleigh length */
00055     sunrealtype ph0; /**< beam center */
00056     sunrealtype phA; /**< beam length */
00057 };
00058
00059 /***** simulation function declarations *****/
00060
00061 /// complete 1D Simulation function
00062 void Sim1D(const array<sunrealtype,2>, const int, const sunrealtype,
00063           const sunindextype, const bool, int *, const sunrealtype, const int,
00064           const string, const int, const char, const vector<planewave> &,
00065           const vector<gaussian1D> &);
00066 /// complete 2D Simulation function
00067 void Sim2D(const array<sunrealtype,2>, const int, const array<sunrealtype,2>,
00068           const array<sunindextype,2>, const array<int,2>, const bool, int *,
00069           const sunrealtype, const int, const string, const int, const char,
00070           const vector<planewave> &, const vector<gaussian2D> &);
00071 /// complete 3D Simulation function
00072 void Sim3D(const array<sunrealtype,2>, const int, const array<sunrealtype,3>,
00073           const array<sunindextype,3>, const array<int,3>, const bool, int *,
00074           const sunrealtype, const int, const string, const int, const char,
00075           const vector<planewave> &, const vector<gaussian3D> &);
00076
00077 /** MPI timer function */
00078 void timer(double &, double &);

```

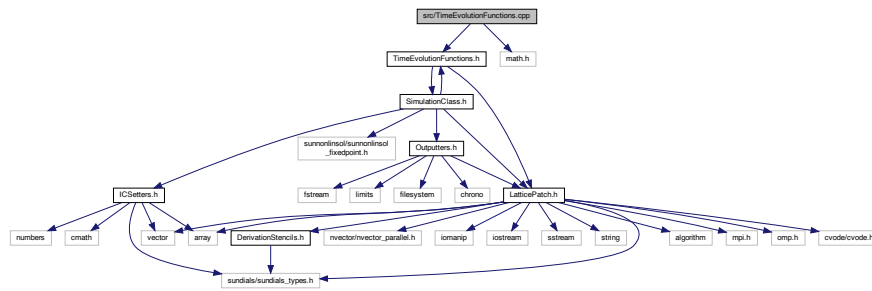

6.28 src/TimeEvolutionFunctions.cpp File Reference

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

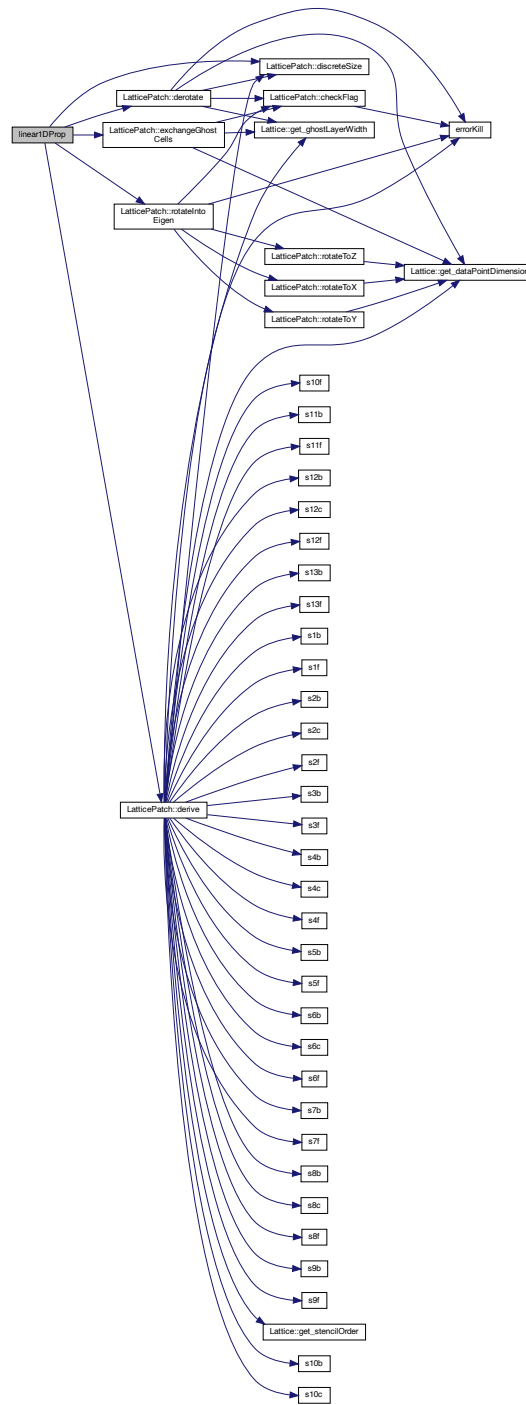
Maxwell propagation function for 1D – only for reference.

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 #pragma distribute_point
00063 for (int i = 0; i < totalNP; i++) {
00064     pp = i * 6;
00065     /*
00066     simple vacuum Maxwell equations for spatial derivative only in x-direction
00067     temporal derivative is approximated by spatial derivative according to the
00068     numerical scheme with Jacobi=0 -> no polarization or magnetization terms
00069     */
00070     duData[pp + 0] = 0;
00071     duData[pp + 1] = -dxData[pp + 5];
00072     duData[pp + 2] = dxData[pp + 4];
00073     duData[pp + 3] = 0;
00074     duData[pp + 4] = dxData[pp + 2];
00075     duData[pp + 5] = -dxData[pp + 1];
00076 }
00077 }
```

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

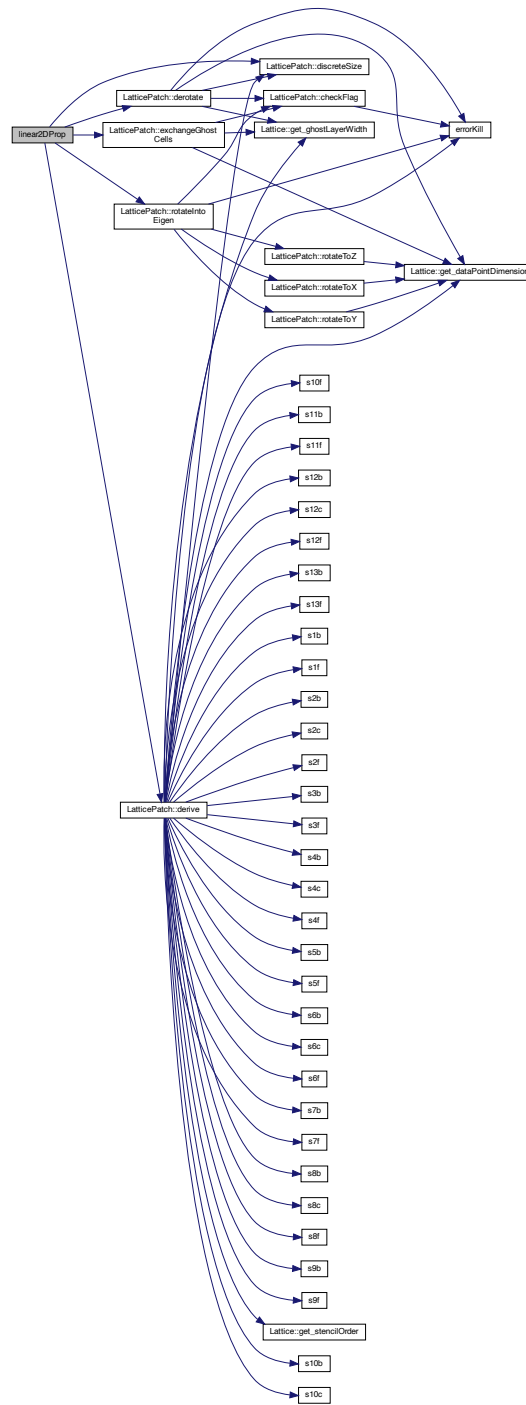
```

00273                                     {
00274
00275     sunrealtype *duData = data->duData;
00276     sunrealtype *dxData = data->buffData[1 - 1];
00277     sunrealtype *dyData = data->buffData[2 - 1];
00278
00279     data->exchangeGhostCells(1);
00280     data->rotateIntoEigen(1);
00281     data->derive(1);
00282     data->derotate(1, dxData);
00283     data->exchangeGhostCells(2);
00284     data->rotateIntoEigen(2);
00285     data->derive(2);
00286     data->derotate(2, dyData);
00287
00288     int totalNP = data->discreteSize();
00289     int pp = 0;
00290     #pragma distribute_point
00291     for (int i = 0; i < totalNP; i++) {
00292         pp = i * 6;
00293         duData[pp + 0] = dyData[pp + 5];
00294         duData[pp + 1] = -dxData[pp + 5];
00295         duData[pp + 2] = -dyData[pp + 3] + dxData[pp + 4];
00296         duData[pp + 3] = -dyData[pp + 2];
00297         duData[pp + 4] = dxData[pp + 2];
00298         duData[pp + 5] = dyData[pp + 0] - dxData[pp + 1];
00299     }
00300 }

```

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

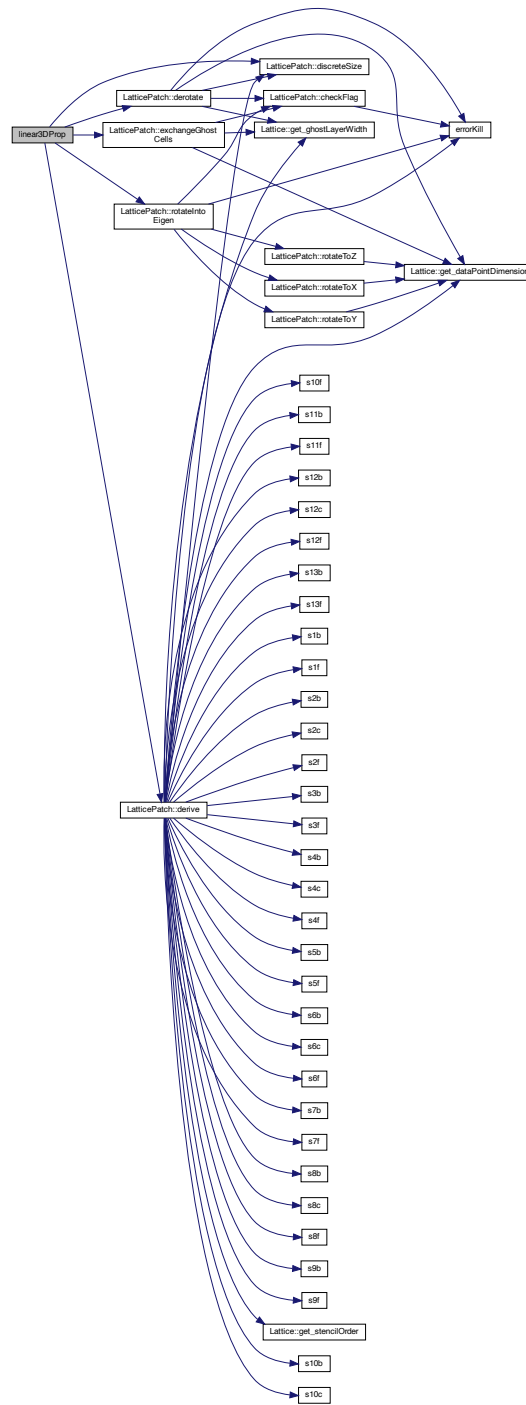
```

00489
00490
00491     sunrealtype *duData = data->duData;
00492     sunrealtype *dxData = data->buffData[1 - 1];
00493     sunrealtype *dyData = data->buffData[2 - 1];
00494     sunrealtype *dzData = data->buffData[3 - 1];
00495
00496     /* Under the hood call of point-to-point or collective communication */
00497     // Point-to-Point:
00498     data->exchangeGhostCells(1);
00499     data->rotateIntoEigen(1);
00500     data->derive(1);
00501     data->derotate(1, dxData);
00502     data->exchangeGhostCells(2);
00503     data->rotateIntoEigen(2);
00504     data->derive(2);
00505     data->derotate(2, dyData);
00506     data->exchangeGhostCells(3);
00507     data->rotateIntoEigen(3);
00508     data->derive(3);
00509     data->derotate(3, dzData);
00510
00511     // Collective:
00512     /*
00513         data->exchangeGhostCells3D();
00514         data->rotateIntoEigen3D();
00515         data->derive(1);
00516         data->derotate(1, dxData);
00517         data->derive(2);
00518         data->derotate(2, dyData);
00519         data->derive(3);
00520         data->derotate(3, dzData);
00521     */
00522
00523     int totalNP = data->discreteSize();
00524     int pp = 0;
00525     #pragma distribute_point
00526     for (int i = 0; i < totalNP; i++) {
00527         pp = i * 6;
00528         duData[pp + 0] = dyData[pp + 5] - dzData[pp + 4];
00529         duData[pp + 1] = dzData[pp + 3] - dxData[pp + 5];
00530         duData[pp + 2] = dxData[pp + 4] - dyData[pp + 3];
00531         duData[pp + 3] = -dyData[pp + 2] + dzData[pp + 1];
00532         duData[pp + 4] = -dzData[pp + 0] + dxData[pp + 2];
00533         duData[pp + 5] = -dxData[pp + 1] + dyData[pp + 0];
00534     }
00535 }

```

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

```

00080                                                                 {
00081
00082     // pointer to spatial derivative data sufficient, temporal derivative data
00083     // provided with udot
00084     sunrealtype *dxData = data->buffData[1 - 1];
00085
00086     // same sequence as in the linear case
00087     data->exchangeGhostCells(1);
00088     data->rotateIntoEigen(1);
00089     data->derive(1);
00090     data->derotate(1, dxData);
00091
00092     /*
00093     F and G are nonzero in the nonlinear case,
00094     polarization and magnetization contributions in Jacobi matrix style
00095     with derivatives of polarization and magnetization
00096     w.r.t. E- and B-field
00097     */
00098     sunrealtype f = NAN, g = NAN; // em field invariants F, G
00099     sunrealtype lf = NAN, lff = NAN, lfg = NAN, lg = NAN,
00100         lgg = NAN; // derivatives of Lagrangian w.r.t. field invariants
00101     array<sunrealtype, 36> JMM; // Jacobi matrix
00102     array<sunrealtype, 6> Quad; // array to hold E^2 and B^2 components
00103     array<sunrealtype, 6> h; // holding temporal derivatives of E and B components
00104         // before operating (1+Z)^-1
00105     sunrealtype pseudoDenom = NAN; // needed for inversion of 1+Z
00106     sunrealtype *udata = nullptr,
00107         *dudata = nullptr; // pointers to data and temp. derivative data
00108     udata = NV_DATA_P(u);
00109     dudata = NV_DATA_P(udot);
00110     int totalNP = data->discreteSize(); // number of points in the patch
00111     // #pragma omp parallel for private(...) reduction(...) -> unsafe due to
00112     // reductions and many variables, how to deal with / reduction?
00113     // #pragma block_loop
00114     // #pragma unroll_and_jam
00115     // #pragma distribute_point
00116     for (int pp = 0; pp < totalNP * 6;
00117         pp += 6) { // loops through all 6dim points in the patch
00118         // for(int ppB=0;ppB<totalNP*6;ppB+=6*6){
00119         // for(int pp=ppB;pp<min(totalNP*6,ppB+6*6);pp+=6){
00120         // Calculation of the Jacobi matrix
00121         // 1. Calculate F and G
00122         f = 0.5 * ((Quad[0] = udata[pp] * udata[pp]) +
00123             (Quad[1] = udata[pp + 1] * udata[pp + 1]) +
00124             (Quad[2] = udata[pp + 2] * udata[pp + 2]) -
00125             (Quad[3] = udata[pp + 3] * udata[pp + 3]) -
00126             (Quad[4] = udata[pp + 4] * udata[pp + 4]) -
00127             (Quad[5] = udata[pp + 5] * udata[pp + 5]));
00128         g = udata[pp] * udata[pp + 3] + udata[pp + 1] * udata[pp + 4] +
00129             udata[pp + 2] * udata[pp + 5];
00130         // 2. Choose process/expansion order and assign derivative values of L
00131         // w.r.t. F, G
00132         switch (*c) {
00133         case 0:
00134             lf = 0;
00135             lff = 0;
00136             lfg = 0;
00137             lg = 0;
00138             lgg = 0;
00139             break;
00140         case 2:
00141             lf = 0.000354046449700427580438254 * f * f +
00142                 0.000191775160254398272737387 * g * g;
00143             lff = 0.0007080928994008551608765075 * f;
00144             lfg = 0.0003835503205087965454747749 * g;
00145             lg = 0.0003835503205087965454747749 * f * g;
00146             lgg = 0.0003835503205087965454747749 * f;
00147             break;
00148         case 1:
00149             lf = 0.000206527095658582755255648 * f;
00150             lff = 0.000206527095658582755255648;
00151             lfg = 0;
00152             lg = 0.0003614224174025198216973841 * g;
00153             lgg = 0.0003614224174025198216973841;

```



```

00154         break;
00155     case 3:
00156         lf = (0.000206527095658582755255648 + 0.000354046449700427580438254 * f) *
00157             f +
00158             0.000191775160254398272737387 * g * g;
00159         lff = 0.000206527095658582755255648 + 0.0007080928994008551608765075 * f;
00160         lfg = 0.0003835503205087965454747749 * g;
00161         lg = (0.0003614224174025198216973841 +
00162             0.0003835503205087965454747749 * f) *
00163             g;
00164         lgg = 0.0003614224174025198216973841 + 0.0003835503205087965454747749 * f;
00165         break;
00166     default:
00167         errorKill(
00168             "You need to specify a correct order in the weak-field expansion.");
00169     }
00170     // 3. Assign Jacobi components
00171     JMM[0] = lf + lff * Quad[0] +
00172         udata[3 + pp] * (2 * lfg * udata[pp] + lgg * udata[3 + pp]);
00173     JMM[6] =
00174         lff * udata[pp] * udata[1 + pp] + lfg * udata[1 + pp] * udata[3 + pp] +
00175         lfg * udata[pp] * udata[4 + pp] + lgg * udata[3 + pp] * udata[4 + pp];
00176     JMM[7] = lf + lff * Quad[1] +
00177         udata[4 + pp] * (2 * lfg * udata[1 + pp] + lgg * udata[4 + pp]);
00178     JMM[12] =
00179         lff * udata[pp] * udata[2 + pp] + lfg * udata[2 + pp] * udata[3 + pp] +
00180         lfg * udata[pp] * udata[5 + pp] + lgg * udata[3 + pp] * udata[5 + pp];
00181     JMM[13] = lff * udata[1 + pp] * udata[2 + pp] +
00182         lfg * udata[2 + pp] * udata[4 + pp] +
00183         lfg * udata[1 + pp] * udata[5 + pp] +
00184         lgg * udata[4 + pp] * udata[5 + pp];
00185     JMM[14] = lf + lff * Quad[2] +
00186         udata[5 + pp] * (2 * lfg * udata[2 + pp] + lgg * udata[5 + pp]);
00187     JMM[18] = lg + lfg * (Quad[0] - Quad[3 + 0]) +
00188         (-lff + lgg) * udata[pp] * udata[3 + pp];
00189     JMM[19] = -(udata[3 + pp] * (lff * udata[1 + pp] + lfg * udata[4 + pp])) +
00190         udata[pp] * (lfg * udata[1 + pp] + lgg * udata[4 + pp]);
00191     JMM[20] = -(udata[3 + pp] * (lff * udata[2 + pp] + lfg * udata[5 + pp])) +
00192         udata[pp] * (lfg * udata[2 + pp] + lgg * udata[5 + pp]);
00193     JMM[21] = -lf + lgg * Quad[0] +
00194         udata[3 + pp] * (-2 * lfg * udata[pp] + lff * udata[3 + pp]);
00195     JMM[24] = udata[1 + pp] * (lfg * udata[pp] + lgg * udata[3 + pp]) -
00196         (lff * udata[pp] + lfg * udata[3 + pp]) * udata[4 + pp];
00197     JMM[25] = lg + lfg * (Quad[1] - Quad[4 + 0]) +
00198         (-lff + lgg) * udata[1 + pp] * udata[4 + pp];
00199     JMM[26] = -(udata[4 + pp] * (lff * udata[2 + pp] + lfg * udata[5 + pp])) +
00200         udata[1 + pp] * (lfg * udata[2 + pp] + lgg * udata[5 + pp]);
00201     JMM[27] = lgg * udata[pp] * udata[1 + pp] +
00202         lff * udata[3 + pp] * udata[4 + pp] -
00203         lfg * (udata[1 + pp] * udata[3 + pp] + udata[pp] * udata[4 + pp]);
00204     JMM[28] = -lf + lgg * Quad[1] +
00205         udata[4 + pp] * (-2 * lfg * udata[1 + pp] + lff * udata[4 + pp]);
00206     JMM[30] = udata[2 + pp] * (lfg * udata[pp] + lgg * udata[3 + pp]) -
00207         (lff * udata[pp] + lfg * udata[3 + pp]) * udata[5 + pp];
00208     JMM[31] = udata[2 + pp] * (lfg * udata[1 + pp] + lgg * udata[4 + pp]) -
00209         (lff * udata[1 + pp] + lfg * udata[4 + pp]) * udata[5 + pp];
00210     JMM[32] = lg + lfg * (Quad[2] - Quad[5 + 0]) +
00211         (-lff + lgg) * udata[2 + pp] * udata[5 + pp];
00212     JMM[33] = lgg * udata[pp] * udata[2 + pp] +
00213         lff * udata[3 + pp] * udata[5 + pp] -
00214         lfg * (udata[2 + pp] * udata[3 + pp] + udata[pp] * udata[5 + pp]);
00215     JMM[34] =
00216         lgg * udata[1 + pp] * udata[2 + pp] +
00217         lff * udata[4 + pp] * udata[5 + pp] -
00218         lfg * (udata[2 + pp] * udata[4 + pp] + udata[1 + pp] * udata[5 + pp]);
00219     JMM[35] = -lf + lgg * Quad[2] +
00220         udata[5 + pp] * (-2 * lfg * udata[2 + pp] + lff * udata[5 + pp]);
00221     // #pragma unroll_and_jam
00222     // #pragma distribute_point
00223     for (int i = 0; i < 6; i++) {
00224         for (int j = i + 1; j < 6; j++) {
00225             JMM[i * 6 + j] = JMM[j * 6 + i];
00226         }
00227     }
00228     // 4. Final values for temporal derivatives of field values
00229     h[0] = 0;
00230     h[1] = dxData[pp] * JMM[30] + dxData[1 + pp] * JMM[31] +
00231         dxData[2 + pp] * JMM[32] + dxData[3 + pp] * JMM[33] +
00232         dxData[4 + pp] * JMM[34] + dxData[5 + pp] * (-1 + JMM[35]);
00233     h[2] = -(dxData[pp] * JMM[24]) - dxData[1 + pp] * JMM[25] -
00234         dxData[2 + pp] * JMM[26] - dxData[3 + pp] * JMM[27] +
00235         dxData[4 + pp] * (1 - JMM[28]) - dxData[5 + pp] * JMM[29];
00236     h[3] = 0;
00237     h[4] = dxData[2 + pp];
00238     h[5] = -dxData[1 + pp];
00239     h[0] -= h[3] * JMM[3] + h[4] * JMM[4] + h[5] * JMM[5];
00240     h[1] -= h[3] * JMM[9] + h[4] * JMM[10] + h[5] * JMM[11];

```

```

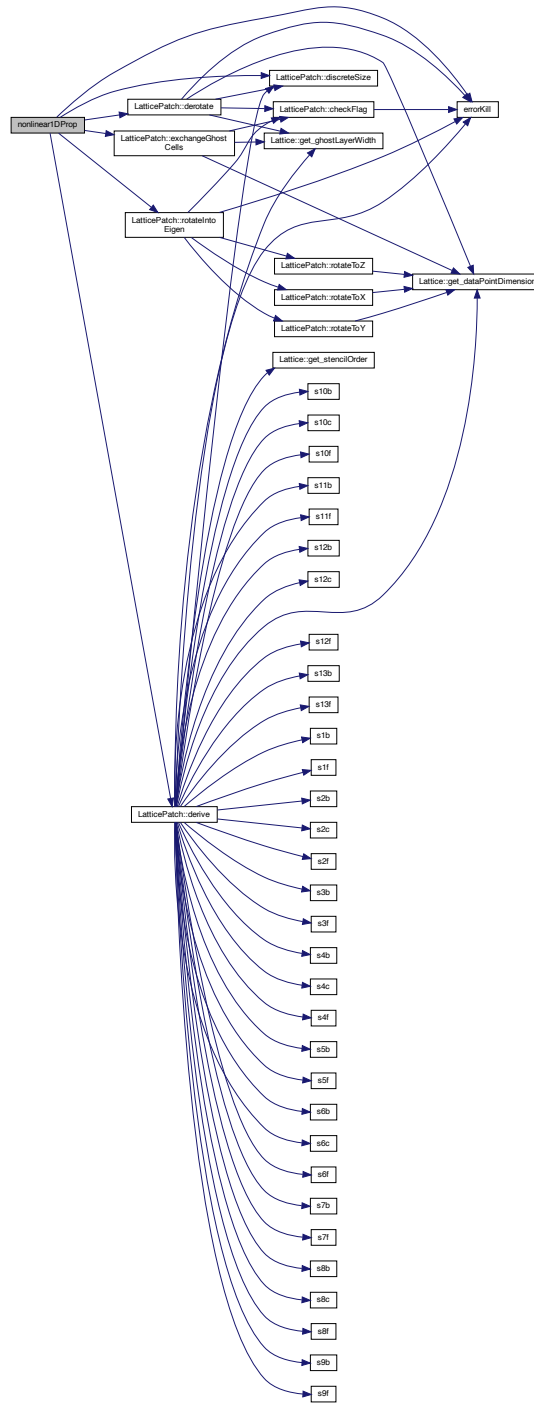
00241     h[2] -= h[3] * JMM[15] + h[4] * JMM[16] + h[5] * JMM[17];
00242     // (1+2)^-1 applies only to E components
00243     dudata[pp + 0] =
00244         h[2] * (-(JMM[2] * (1 + JMM[7])) + JMM[1] * JMM[8]) +
00245         h[1] * (JMM[2] * JMM[13] - JMM[1] * (1 + JMM[14])) +
00246         h[0] * (1 - JMM[8] * JMM[13] + JMM[14] + JMM[7] * (1 + JMM[14]));
00247     dudata[pp + 1] =
00248         h[2] * (JMM[2] * JMM[6] - (1 + JMM[0]) * JMM[8]) +
00249         h[1] * (1 - JMM[2] * JMM[12] + JMM[14] + JMM[0] * (1 + JMM[14])) +
00250         h[0] * (JMM[8] * JMM[12] - JMM[6] * (1 + JMM[14]));
00251     dudata[pp + 2] =
00252         h[2] * (1 - JMM[1] * JMM[6] + JMM[7] + JMM[0] * (1 + JMM[7])) +
00253         h[1] * (JMM[1] * JMM[12] - (1 + JMM[0]) * JMM[13]) +
00254         h[0] * (-(1 + JMM[7]) * JMM[12]) + JMM[6] * JMM[13]);
00255     pseudoDenom =
00256         -(1 + JMM[7]) * (-1 + JMM[2] * JMM[12]) +
00257         (JMM[2] * JMM[6] - JMM[8]) * JMM[13] + JMM[14] + JMM[7] * JMM[14] +
00258         JMM[0] * (1 + JMM[7] - JMM[8] * JMM[13] + (1 + JMM[7]) * JMM[14]) -
00259         JMM[1] * (-(JMM[8] * JMM[12]) + JMM[6] * (1 + JMM[14]));
00260     dudata[pp + 0] /= pseudoDenom;
00261     dudata[pp + 1] /= pseudoDenom;
00262     dudata[pp + 2] /= pseudoDenom;
00263     dudata[pp + 3] = h[3];
00264     dudata[pp + 4] = h[4];
00265     dudata[pp + 5] = h[5];
00266 }
00267 return;
00268 }

```

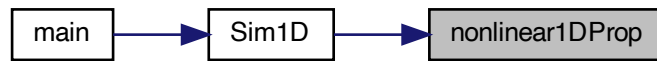
References [LatticePatch::buffData](#), [LatticePatch::derive\(\)](#), [LatticePatch::derotate\(\)](#), [LatticePatch::discreteSize\(\)](#), [errorKill\(\)](#), [LatticePatch::exchangeGhostCells\(\)](#), and [LatticePatch::rotateIntoEigen\(\)](#).

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

```

00303
00304
00305     sunrealtype *dxData = data->buffData[1 - 1];
00306     sunrealtype *dyData = data->buffData[2 - 1];
00307
00308     data->exchangeGhostCells(1);
00309     data->rotateIntoEigen(1);
00310     data->derive(1);
00311     data->derotate(1, dxData);
00312     data->exchangeGhostCells(2);
00313     data->rotateIntoEigen(2);
00314     data->derive(2);
00315     data->derotate(2, dyData);
00316
00317     sunrealtype f = NAN, g = NAN;
00318     sunrealtype lff = NAN, lff = NAN, lfg = NAN, lg = NAN, lgg = NAN;
00319     array<sunrealtype, 36> JMM;
00320     array<sunrealtype, 6> Quad;
00321     array<sunrealtype, 6> h;
00322     sunrealtype pseudoDenom = NAN;
00323     sunrealtype *udata = nullptr, *dudata = nullptr;
00324     udata = NV_DATA_P(u);
00325     dudata = NV_DATA_P(udot);
00326     int totalNP = data->discreteSize();
00327     ##pragma distribute_point
00328     ##pragma unroll_and_jam
00329     for (int pp = 0; pp < totalNP * 6; pp += 6) {
00330         // 1
00331         f = 0.5 * ((Quad[0] = udata[pp] * udata[pp]) +
00332                   (Quad[1] = udata[pp + 1] * udata[pp + 1]) +
00333                   (Quad[2] = udata[pp + 2] * udata[pp + 2]) -
00334                   (Quad[3] = udata[pp + 3] * udata[pp + 3]) -
00335                   (Quad[4] = udata[pp + 4] * udata[pp + 4]) -
00336                   (Quad[5] = udata[pp + 5] * udata[pp + 5]));
00337         g = udata[pp] * udata[pp + 3] + udata[pp + 1] * udata[pp + 4] +
00338             udata[pp + 2] * udata[pp + 5];
00339         // 2
00340         switch (*c) {
00341         case 0:
00342             lff = 0;
00343             lff = 0;
00344             lfg = 0;
00345             lg = 0;
  
```

```

00346     lgg = 0;
00347     break;
00348 case 2:
00349     lf = 0.000354046449700427580438254 * f * f +
00350         0.000191775160254398272737387 * g * g;
00351     lff = 0.0007080928994008551608765075 * f;
00352     lfg = 0.0003835503205087965454747749 * g;
00353     lg = 0.0003835503205087965454747749 * f * g;
00354     lgg = 0.0003835503205087965454747749 * f;
00355     break;
00356 case 1:
00357     lf = 0.000206527095658582755255648 * f;
00358     lff = 0.000206527095658582755255648;
00359     lfg = 0;
00360     lg = 0.0003614224174025198216973841 * g;
00361     lgg = 0.0003614224174025198216973841;
00362     break;
00363 case 3:
00364     lf = (0.000206527095658582755255648 + 0.000354046449700427580438254 * f) *
00365         f +
00366         0.000191775160254398272737387 * g * g;
00367     lff = 0.000206527095658582755255648 + 0.000708092899400855160876508 * f;
00368     lfg = 0.0003835503205087965454747749 * g;
00369     lg = (0.000361422417402519821697384 + 0.000383550320508796545474775 * f) *
00370         g;
00371     lgg = 0.000361422417402519821697384 + 0.000383550320508796545474775 * f;
00372     break;
00373 default:
00374     errorKill(
00375         "You need to specify a correct order in the weak-field expansion.");
00376 }
00377 // 3
00378 JMM[0] = lf + lff * Quad[0] +
00379     udata[3 + pp] * (2 * lfg * udata[pp] + lgg * udata[3 + pp]);
00380 JMM[6] =
00381     lff * udata[pp] * udata[1 + pp] + lfg * udata[1 + pp] * udata[3 + pp] +
00382     lfg * udata[pp] * udata[4 + pp] + lgg * udata[3 + pp] * udata[4 + pp];
00383 JMM[7] = lf + lff * Quad[1] +
00384     udata[4 + pp] * (2 * lfg * udata[1 + pp] + lgg * udata[4 + pp]);
00385 JMM[12] =
00386     lff * udata[pp] * udata[2 + pp] + lfg * udata[2 + pp] * udata[3 + pp] +
00387     lfg * udata[pp] * udata[5 + pp] + lgg * udata[3 + pp] * udata[5 + pp];
00388 JMM[13] = lff * udata[1 + pp] * udata[2 + pp] +
00389     lfg * udata[2 + pp] * udata[4 + pp] +
00390     lfg * udata[1 + pp] * udata[5 + pp] +
00391     lgg * udata[4 + pp] * udata[5 + pp];
00392 JMM[14] = lf + lff * Quad[2] +
00393     udata[5 + pp] * (2 * lfg * udata[2 + pp] + lgg * udata[5 + pp]);
00394 JMM[18] = lg + lfg * (Quad[0] - Quad[3 + 0]) +
00395     (-lff + lgg) * udata[pp] * udata[3 + pp];
00396 JMM[19] = -(udata[3 + pp] * (lff * udata[1 + pp] + lfg * udata[4 + pp])) +
00397     udata[pp] * (lfg * udata[1 + pp] + lgg * udata[4 + pp]);
00398 JMM[20] = -(udata[3 + pp] * (lff * udata[2 + pp] + lfg * udata[5 + pp])) +
00399     udata[pp] * (lfg * udata[2 + pp] + lgg * udata[5 + pp]);
00400 JMM[21] = -lf + lgg * Quad[0] +
00401     udata[3 + pp] * (-2 * lfg * udata[pp] + lff * udata[3 + pp]);
00402 JMM[24] = udata[1 + pp] * (lfg * udata[pp] + lgg * udata[3 + pp]) -
00403     (lff * udata[pp] + lfg * udata[3 + pp]) * udata[4 + pp];
00404 JMM[25] = lg + lfg * (Quad[1] - Quad[4 + 0]) +
00405     (-lff + lgg) * udata[1 + pp] * udata[4 + pp];
00406 JMM[26] = -(udata[4 + pp] * (lff * udata[2 + pp] + lfg * udata[5 + pp])) +
00407     udata[1 + pp] * (lfg * udata[2 + pp] + lgg * udata[5 + pp]);
00408 JMM[27] = lgg * udata[pp] * udata[1 + pp] +
00409     lff * udata[3 + pp] * udata[4 + pp] -
00410     lfg * (udata[1 + pp] * udata[3 + pp] + udata[pp] * udata[4 + pp]);
00411 JMM[28] = -lf + lgg * Quad[1] +
00412     udata[4 + pp] * (-2 * lfg * udata[1 + pp] + lff * udata[4 + pp]);
00413 JMM[30] = udata[2 + pp] * (lfg * udata[pp] + lgg * udata[3 + pp]) -
00414     (lff * udata[pp] + lfg * udata[3 + pp]) * udata[5 + pp];
00415 JMM[31] = udata[2 + pp] * (lfg * udata[1 + pp] + lgg * udata[4 + pp]) -
00416     (lff * udata[1 + pp] + lfg * udata[4 + pp]) * udata[5 + pp];
00417 JMM[32] = lg + lfg * (Quad[2] - Quad[5 + 0]) +
00418     (-lff + lgg) * udata[2 + pp] * udata[5 + pp];
00419 JMM[33] = lgg * udata[pp] * udata[2 + pp] +
00420     lff * udata[3 + pp] * udata[5 + pp] -
00421     lfg * (udata[2 + pp] * udata[3 + pp] + udata[pp] * udata[5 + pp]);
00422 JMM[34] =
00423     lgg * udata[1 + pp] * udata[2 + pp] +
00424     lff * udata[4 + pp] * udata[5 + pp] -
00425     lfg * (udata[2 + pp] * udata[4 + pp] + udata[1 + pp] * udata[5 + pp]);
00426 JMM[35] = -lf + lgg * Quad[2] +
00427     udata[5 + pp] * (-2 * lfg * udata[2 + pp] + lff * udata[5 + pp]);
00428 // 4
00429 // #pragma distribute_point
00430 // #pragma unroll_and_jam
00431 for (int i = 0; i < 6; i++) {
00432     for (int j = i + 1; j < 6; j++) {

```

```

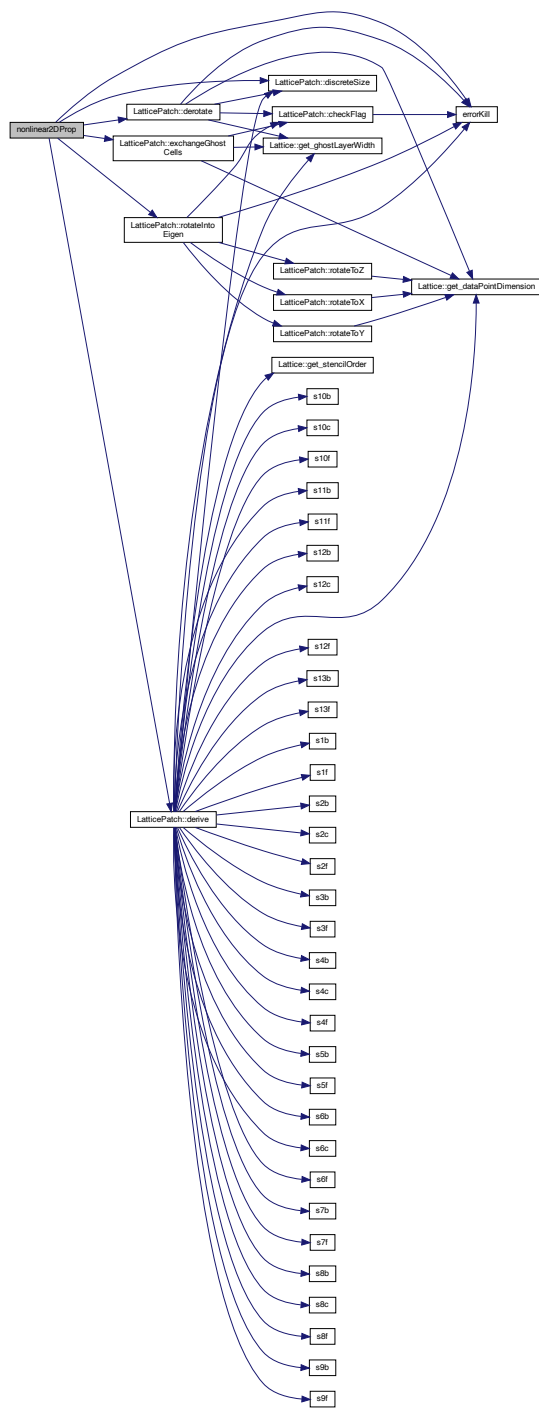
00433     JMM[i * 6 + j] = JMM[j * 6 + i];
00434 }
00435 }
00436 h[0] = 0;
00437 h[1] = dxData[pp] * JMM[30] + dxData[1 + pp] * JMM[31] +
00438     dxData[2 + pp] * JMM[32] + dxData[3 + pp] * JMM[33] +
00439     dxData[4 + pp] * JMM[34] + dxData[5 + pp] * (-1 + JMM[35]);
00440 h[2] = -(dxData[pp] * JMM[24]) - dxData[1 + pp] * JMM[25] -
00441     dxData[2 + pp] * JMM[26] - dxData[3 + pp] * JMM[27] +
00442     dxData[4 + pp] * (1 - JMM[28]) - dxData[5 + pp] * JMM[29];
00443 h[3] = 0;
00444 h[4] = dxData[2 + pp];
00445 h[5] = -dxData[1 + pp];
00446 h[0] += -(dyData[pp] * JMM[30]) - dyData[1 + pp] * JMM[31] -
00447     dyData[2 + pp] * JMM[32] - dyData[3 + pp] * JMM[33] -
00448     dyData[4 + pp] * JMM[34] + dyData[5 + pp] * (1 - JMM[35]);
00449 h[1] += 0;
00450 h[2] += dyData[pp] * JMM[18] + dyData[1 + pp] * JMM[19] +
00451     dyData[2 + pp] * JMM[20] + dyData[3 + pp] * (-1 + JMM[21]) +
00452     dyData[4 + pp] * JMM[22] + dyData[5 + pp] * JMM[23];
00453 h[3] += -dyData[2 + pp];
00454 h[4] += 0;
00455 h[5] += dyData[pp];
00456 h[0] -= h[3] * JMM[3] + h[4] * JMM[4] + h[5] * JMM[5];
00457 h[1] -= h[3] * JMM[9] + h[4] * JMM[10] + h[5] * JMM[11];
00458 h[2] -= h[3] * JMM[15] + h[4] * JMM[16] + h[5] * JMM[17];
00459 dudata[pp + 0] =
00460     h[2] * (-(JMM[2] * (1 + JMM[7])) + JMM[1] * JMM[8]) +
00461     h[1] * (JMM[2] * JMM[13] - JMM[1] * (1 + JMM[14])) +
00462     h[0] * (1 - JMM[8] * JMM[13] + JMM[14] + JMM[7] * (1 + JMM[14]));
00463 dudata[pp + 1] =
00464     h[2] * (JMM[2] * JMM[6] - (1 + JMM[0]) * JMM[8]) +
00465     h[1] * (1 - JMM[2] * JMM[12] + JMM[14] + JMM[0] * (1 + JMM[14])) +
00466     h[0] * (JMM[8] * JMM[12] - JMM[6] * (1 + JMM[14]));
00467 dudata[pp + 2] =
00468     h[2] * (1 - JMM[1] * JMM[6] + JMM[7] + JMM[0] * (1 + JMM[7])) +
00469     h[1] * (JMM[1] * JMM[12] - (1 + JMM[0]) * JMM[13]) +
00470     h[0] * (-(1 + JMM[7]) * JMM[12]) + JMM[6] * JMM[13];
00471 pseudoDenom =
00472     -(1 + JMM[7]) * (-1 + JMM[2] * JMM[12]) +
00473     (JMM[2] * JMM[6] - JMM[8]) * JMM[13] + JMM[14] + JMM[7] * JMM[14] +
00474     JMM[0] * (1 + JMM[7] - JMM[8] * JMM[13] + (1 + JMM[7]) * JMM[14]) -
00475     JMM[1] * (-(JMM[8] * JMM[12]) + JMM[6] * (1 + JMM[14]));
00476 dudata[pp + 0] /= pseudoDenom;
00477 dudata[pp + 1] /= pseudoDenom;
00478 dudata[pp + 2] /= pseudoDenom;
00479 dudata[pp + 3] = h[3];
00480 dudata[pp + 4] = h[4];
00481 dudata[pp + 5] = h[5];
00482 }
00483 return;
00484 }

```

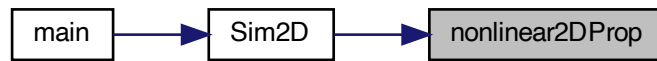
References [LatticePatch::buffData](#), [LatticePatch::derive\(\)](#), [LatticePatch::derotate\(\)](#), [LatticePatch::discreteSize\(\)](#), [errorKill\(\)](#), [LatticePatch::exchangeGhostCells\(\)](#), and [LatticePatch::rotateIntoEigen\(\)](#).

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

```

00538
00539
00540 sunrealtype *dxData = data->buffData[1 - 1];
00541 sunrealtype *dyData = data->buffData[2 - 1];
00542 sunrealtype *dzData = data->buffData[3 - 1];
00543
00544 /* Under the hood call of point-to-point or collective communication */
00545 // Point-to-Point:
00546
00547 data->exchangeGhostCells(1);
00548 data->rotateIntoEigen(1);
00549 data->derive(1);
00550 data->derotate(1, dxData);
00551 data->exchangeGhostCells(2);
00552 data->rotateIntoEigen(2);
00553 data->derive(2);
00554 data->derotate(2, dyData);
00555 data->exchangeGhostCells(3);
00556 data->rotateIntoEigen(3);
00557 data->derive(3);
00558 data->derotate(3, dzData);
00559
00560 // Collective:
00561 /*
00562 data->exchangeGhostCells3D();
00563 data->rotateIntoEigen3D();
00564 data->derive(1);
00565 data->derotate(1, dxData);
00566 data->derive(2);
00567 data->derotate(2, dyData);
00568 data->derive(3);
00569 data->derotate(3, dzData);
00570 */
00571
00572 sunrealtype f = NAN, g = NAN;
00573 sunrealtype lff = NAN, lffg = NAN, lfg = NAN, lg = NAN, lgg = NAN;
00574 array<sunrealtype, 36> JMM;
00575 array<sunrealtype, 6> Quad;
00576 array<sunrealtype, 6> h;
00577 sunrealtype pseudoDenom = NAN;
00578 sunrealtype *udata = nullptr, *dudata = nullptr;
00579 udata = NV_DATA_P(u);
00580 dudata = NV_DATA_P(udot);
  
```



```

00581 int totalNP = data->discreteSize();
00582 // #pragma distribute_point
00583 // #pragma unroll_and_jam
00584 for (int pp = 0; pp < totalNP * 6; pp += 6) {
00585     // 1
00586     f = 0.5 * ((Quad[0] = udata[pp] * udata[pp]) +
00587               (Quad[1] = udata[pp + 1] * udata[pp + 1]) +
00588               (Quad[2] = udata[pp + 2] * udata[pp + 2]) -
00589               (Quad[3] = udata[pp + 3] * udata[pp + 3]) -
00590               (Quad[4] = udata[pp + 4] * udata[pp + 4]) -
00591               (Quad[5] = udata[pp + 5] * udata[pp + 5]));
00592     g = udata[pp] * udata[pp + 3] + udata[pp + 1] * udata[pp + 4] +
00593         udata[pp + 2] * udata[pp + 5];
00594     // 2
00595     switch (*c) {
00596     case 0:
00597         lff = 0;
00598         lff = 0;
00599         lfg = 0;
00600         lg = 0;
00601         lgg = 0;
00602         break;
00603     case 2:
00604         lf = 0.000354046449700427580438254 * f * f +
00605             0.000191775160254398272737387 * g * g;
00606         lff = 0.0007080928994008551608765075 * f;
00607         lfg = 0.0003835503205087965454747749 * g;
00608         lg = 0.0003835503205087965454747749 * f * g;
00609         lgg = 0.0003835503205087965454747749 * f;
00610         break;
00611     case 1:
00612         lf = 0.000206527095658582755255648 * f;
00613         lff = 0.000206527095658582755255648;
00614         lfg = 0;
00615         lg = 0.0003614224174025198216973841 * g;
00616         lgg = 0.0003614224174025198216973841;
00617         break;
00618     case 3:
00619         lf = (0.000206527095658582755255648 + 0.000354046449700427580438254 * f) *
00620             f +
00621             0.000191775160254398272737387 * g * g;
00622         lff = 0.000206527095658582755255648 + 0.000708092899400855160876508 * f;
00623         lfg = 0.0003835503205087965454747749 * g;
00624         lg = (0.000361422417402519821697384 + 0.000383550320508796545474775 * f) *
00625             g;
00626         lgg = 0.000361422417402519821697384 + 0.000383550320508796545474775 * f;
00627         break;
00628     default:
00629         errorKill(
00630             "You need to specify a correct order in the weak-field expansion.");
00631     }
00632     // 3
00633     JMM[0] = lf + lff * Quad[0] +
00634         udata[3 + pp] * (2 * lfg * udata[pp] + lgg * udata[3 + pp]);
00635     JMM[6] =
00636         lff * udata[pp] * udata[1 + pp] + lfg * udata[1 + pp] * udata[3 + pp] +
00637         lfg * udata[pp] * udata[4 + pp] + lgg * udata[3 + pp] * udata[4 + pp];
00638     JMM[7] = lf + lff * Quad[1] +
00639         udata[4 + pp] * (2 * lfg * udata[1 + pp] + lgg * udata[4 + pp]);
00640     JMM[12] =
00641         lff * udata[pp] * udata[2 + pp] + lfg * udata[2 + pp] * udata[3 + pp] +
00642         lfg * udata[pp] * udata[5 + pp] + lgg * udata[3 + pp] * udata[5 + pp];
00643     JMM[13] = lff * udata[1 + pp] * udata[2 + pp] +
00644         lfg * udata[2 + pp] * udata[4 + pp] +
00645         lfg * udata[1 + pp] * udata[5 + pp] +
00646         lgg * udata[4 + pp] * udata[5 + pp];
00647     JMM[14] = lf + lff * Quad[2] +
00648         udata[5 + pp] * (2 * lfg * udata[2 + pp] + lgg * udata[5 + pp]);
00649     JMM[18] = lg + lfg * (Quad[0] - Quad[3 + 0]) +
00650         (-lff + lgg) * udata[pp] * udata[3 + pp];
00651     JMM[19] = -(udata[3 + pp] * (lff * udata[1 + pp] + lfg * udata[4 + pp])) +
00652         udata[pp] * (lfg * udata[1 + pp] + lgg * udata[4 + pp]);
00653     JMM[20] = -(udata[3 + pp] * (lff * udata[2 + pp] + lfg * udata[5 + pp])) +
00654         udata[pp] * (lfg * udata[2 + pp] + lgg * udata[5 + pp]);
00655     JMM[21] = -lf + lgg * Quad[0] +
00656         udata[3 + pp] * (-2 * lfg * udata[pp] + lff * udata[3 + pp]);
00657     JMM[24] = udata[1 + pp] * (lfg * udata[pp] + lgg * udata[3 + pp]) -
00658         (lff * udata[pp] + lfg * udata[3 + pp]) * udata[4 + pp];
00659     JMM[25] = lg + lfg * (Quad[1] - Quad[4 + 0]) +
00660         (-lff + lgg) * udata[1 + pp] * udata[4 + pp];
00661     JMM[26] = -(udata[4 + pp] * (lff * udata[2 + pp] + lfg * udata[5 + pp])) +
00662         udata[1 + pp] * (lfg * udata[2 + pp] + lgg * udata[5 + pp]);
00663     JMM[27] = lgg * udata[pp] * udata[1 + pp] +
00664         lff * udata[3 + pp] * udata[4 + pp] -
00665         lfg * (udata[1 + pp] * udata[3 + pp] + udata[pp] * udata[4 + pp]);
00666     JMM[28] = -lf + lgg * Quad[1] +
00667         udata[4 + pp] * (-2 * lfg * udata[1 + pp] + lff * udata[4 + pp]);

```

```

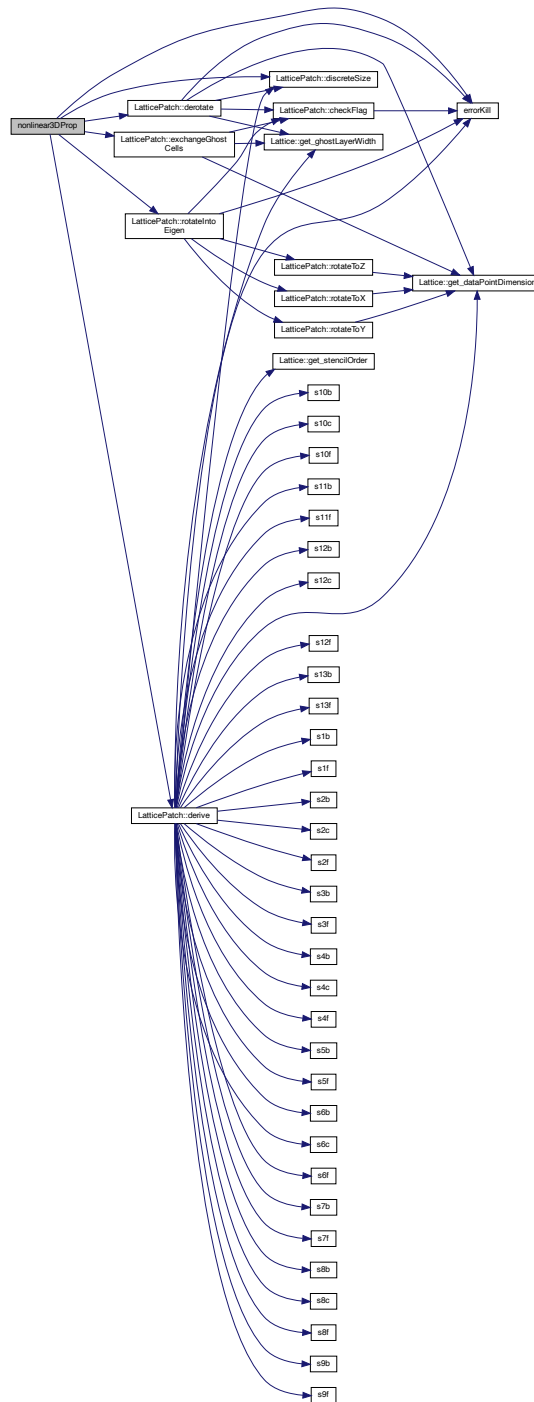
00668 JMM[30] = udata[2 + pp] * (lfg * udata[pp] + lgg * udata[3 + pp]) -
00669         (lff * udata[pp] + lfg * udata[3 + pp]) * udata[5 + pp];
00670 JMM[31] = udata[2 + pp] * (lfg * udata[1 + pp] + lgg * udata[4 + pp]) -
00671         (lff * udata[1 + pp] + lfg * udata[4 + pp]) * udata[5 + pp];
00672 JMM[32] = lg + lfg * (Quad[2] - Quad[5 + 0]) +
00673         (-lff + lgg) * udata[2 + pp] * udata[5 + pp];
00674 JMM[33] = lgg * udata[pp] * udata[2 + pp] +
00675         lff * udata[3 + pp] * udata[5 + pp] -
00676         lfg * (udata[2 + pp] * udata[3 + pp] + udata[pp] * udata[5 + pp]);
00677 JMM[34] =
00678         lgg * udata[1 + pp] * udata[2 + pp] +
00679         lff * udata[4 + pp] * udata[5 + pp] -
00680         lfg * (udata[2 + pp] * udata[4 + pp] + udata[1 + pp] * udata[5 + pp]);
00681 JMM[35] = -lf + lgg * Quad[2] +
00682         udata[5 + pp] * (-2 * lfg * udata[2 + pp] + lff * udata[5 + pp]);
00683 // 4
00684 // #pragma distribute_point
00685 // #pragma unroll_and_jam
00686 for (int i = 0; i < 6; i++) {
00687     for (int j = i + 1; j < 6; j++) {
00688         JMM[i * 6 + j] = JMM[j * 6 + i];
00689     }
00690 }
00691 h[0] = 0;
00692 h[1] = dxData[pp] * JMM[30] + dxData[1 + pp] * JMM[31] +
00693         dxData[2 + pp] * JMM[32] + dxData[3 + pp] * JMM[33] +
00694         dxData[4 + pp] * JMM[34] + dxData[5 + pp] * (-1 + JMM[35]);
00695 h[2] = -(dxData[pp] * JMM[24]) - dxData[1 + pp] * JMM[25] -
00696         dxData[2 + pp] * JMM[26] - dxData[3 + pp] * JMM[27] +
00697         dxData[4 + pp] * (1 - JMM[28]) - dxData[5 + pp] * JMM[29];
00698 h[3] = 0;
00699 h[4] = dxData[2 + pp];
00700 h[5] = -dxData[1 + pp];
00701 h[0] += -(dyData[pp] * JMM[30]) - dyData[1 + pp] * JMM[31] -
00702         dyData[2 + pp] * JMM[32] - dyData[3 + pp] * JMM[33] -
00703         dyData[4 + pp] * JMM[34] + dyData[5 + pp] * (1 - JMM[35]);
00704 h[1] += 0;
00705 h[2] += dyData[pp] * JMM[18] + dyData[1 + pp] * JMM[19] +
00706         dzData[2 + pp] * JMM[20] + dyData[3 + pp] * (-1 + JMM[21]) +
00707         dyData[4 + pp] * JMM[22] + dyData[5 + pp] * JMM[23];
00708 h[3] += -dyData[2 + pp];
00709 h[4] += 0;
00710 h[5] += dyData[pp];
00711 h[0] += dzData[pp] * JMM[24] + dzData[1 + pp] * JMM[25] +
00712         dzData[2 + pp] * JMM[26] + dzData[3 + pp] * JMM[27] +
00713         dzData[4 + pp] * (-1 + JMM[28]) + dzData[5 + pp] * JMM[29];
00714 h[1] += -(dzData[pp] * JMM[18]) - dzData[1 + pp] * JMM[19] -
00715         dzData[2 + pp] * JMM[20] + dzData[3 + pp] * (1 - JMM[21]) -
00716         dzData[4 + pp] * JMM[22] - dzData[5 + pp] * JMM[23];
00717 h[2] += 0;
00718 h[3] += dzData[1 + pp];
00719 h[4] += -dzData[pp];
00720 h[5] += 0;
00721 h[0] -= h[3] * JMM[3] + h[4] * JMM[4] + h[5] * JMM[5];
00722 h[1] -= h[3] * JMM[9] + h[4] * JMM[10] + h[5] * JMM[11];
00723 h[2] -= h[3] * JMM[15] + h[4] * JMM[16] + h[5] * JMM[17];
00724 dudata[pp + 0] =
00725         h[2] * (-JMM[2] * (1 + JMM[7])) + JMM[1] * JMM[8] +
00726         h[1] * (JMM[2] * JMM[13] - JMM[1] * (1 + JMM[14])) +
00727         h[0] * (1 - JMM[8] * JMM[13] + JMM[14] + JMM[7] * (1 + JMM[14]));
00728 dudata[pp + 1] =
00729         h[2] * (JMM[2] * JMM[6] - (1 + JMM[0]) * JMM[8]) +
00730         h[1] * (1 - JMM[2] * JMM[12] + JMM[14] + JMM[0] * (1 + JMM[14])) +
00731         h[0] * (JMM[8] * JMM[12] - JMM[6] * (1 + JMM[14]));
00732 dudata[pp + 2] =
00733         h[2] * (1 - JMM[1] * JMM[6] + JMM[7] + JMM[0] * (1 + JMM[7])) +
00734         h[1] * (JMM[1] * JMM[12] - (1 + JMM[0]) * JMM[13]) +
00735         h[0] * (-((1 + JMM[7]) * JMM[12]) + JMM[6] * JMM[13]);
00736 pseudoDenom =
00737         -((1 + JMM[7]) * (-1 + JMM[2] * JMM[12])) +
00738         (JMM[2] * JMM[6] - JMM[8]) * JMM[13] + JMM[14] + JMM[7] * JMM[14] +
00739         JMM[0] * (1 + JMM[7] - JMM[8] * JMM[13] + (1 + JMM[7]) * JMM[14]) -
00740         JMM[1] * (-JMM[8] * JMM[12]) + JMM[6] * (1 + JMM[14]);
00741 dudata[pp + 0] /= pseudoDenom;
00742 dudata[pp + 1] /= pseudoDenom;
00743 dudata[pp + 2] /= pseudoDenom;
00744 dudata[pp + 3] = h[3];
00745 dudata[pp + 4] = h[4];
00746 dudata[pp + 5] = h[5];
00747 }
00748 return;
00749 }

```

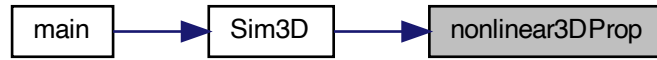
References [LatticePatch::buffData](#), [LatticePatch::derive\(\)](#), [LatticePatch::derotate\(\)](#), [LatticePatch::discreteSize\(\)](#), [errorKill\(\)](#), [LatticePatch::exchangeGhostCells\(\)](#), and [LatticePatch::rotateIntoEigen\(\)](#).

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 #pragma distribute_point
00063 for (int i = 0; i < totalNP; i++) {
00064     pp = i * 6;
00065     /*
00066     simple vacuum Maxwell equations for spatial derivative only in x-direction
00067     temporal derivative is approximated by spatial derivative according to the
00068     numerical scheme with Jacobi=0 -> no polarization or magnetization terms
00069     */
00070     duData[pp + 0] = 0;
00071     duData[pp + 1] = -dxData[pp + 5];
00072     duData[pp + 2] = dxData[pp + 4];
00073     duData[pp + 3] = 0;
00074     duData[pp + 4] = dxData[pp + 2];
00075     duData[pp + 5] = -dxData[pp + 1];
00076 }
00077 }
00078
00079 /// nonlinear 1D HE propagation
00080 void nonlinear1DProp(LatticePatch *data, N_Vector u, N_Vector udot, int *c) {
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 lf = NAN, lff = NAN, lfg = 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     // #pragma omp parallel for private(...) reduction(...) -> unsafe due to
00111     // reductions and many variables, how to deal with / reduction?
00112     // #pragma block_loop
00113     // #pragma unroll_and_jam
00114     // #pragma distribute_point
00115     for (int pp = 0; pp < totalNP * 6;
00116         pp += 6) { // loops through all 6dim points in the patch
00117         // for(int ppB=0;ppB<totalNP*6;ppB+=6*6){
00118         // for(int pp=ppB;pp<min(totalNP*6,ppB+6*6);pp+=6){
00119
00120         /// Calculation of the Jacobi matrix
00121         // 1. Calculate F and G
00122         f = 0.5 * ((Quad[0] = udata[pp] * udata[pp]) +
00123             (Quad[1] = udata[pp + 1] * udata[pp + 1]) +
00124             (Quad[2] = udata[pp + 2] * udata[pp + 2]) -
00125             (Quad[3] = udata[pp + 3] * udata[pp + 3]) -
00126             (Quad[4] = udata[pp + 4] * udata[pp + 4]) -
00127             (Quad[5] = udata[pp + 5] * udata[pp + 5]));
00128         g = udata[pp] * udata[pp + 3] + udata[pp + 1] * udata[pp + 4] +
00129             udata[pp + 2] * udata[pp + 5];
00130         // 2. Choose process/expansion order and assign derivative values of L
00131         // w.r.t. F, G
00132         switch (*c) {
00133         case 0:
00134             lf = 0;
00135             lff = 0;
00136             lfg = 0;
00137             lg = 0;
00138             lgg = 0;
00139             break;
00140         case 2:
00141             lf = 0.000354046449700427580438254 * f * f +
00142                 0.000191775160254398272737387 * g * g;
00143             lff = 0.0007080928994008551608765075 * f;
00144             lfg = 0.0003835503205087965454747749 * g;
00145             lg = 0.0003835503205087965454747749 * f * g;
00146             lgg = 0.0003835503205087965454747749 * f;
00147             break;
00148         case 1:

```

```

00149     lf = 0.000206527095658582755255648 * f;
00150     lff = 0.000206527095658582755255648;
00151     lfg = 0;
00152     lg = 0.0003614224174025198216973841 * g;
00153     lgg = 0.0003614224174025198216973841;
00154     break;
00155 case 3:
00156     lf = (0.000206527095658582755255648 + 0.000354046449700427580438254 * f) *
00157         f +
00158         0.000191775160254398272737387 * g * g;
00159     lff = 0.000206527095658582755255648 + 0.0007080928994008551608765075 * f;
00160     lfg = 0.0003835503205087965454747749 * g;
00161     lg = (0.0003614224174025198216973841 +
00162         0.0003835503205087965454747749 * f) *
00163         g;
00164     lgg = 0.0003614224174025198216973841 + 0.0003835503205087965454747749 * f;
00165     break;
00166 default:
00167     errorKill(
00168         "You need to specify a correct order in the weak-field expansion.");
00169 }
00170 // 3. Assign Jacobi components
00171 JMM[0] = lf + lff * Quad[0] +
00172     udata[3 + pp] * (2 * lfg * udata[pp] + lgg * udata[3 + pp]);
00173 JMM[6] =
00174     lff * udata[pp] * udata[1 + pp] + lfg * udata[1 + pp] * udata[3 + pp] +
00175     lfg * udata[pp] * udata[4 + pp] + lgg * udata[3 + pp] * udata[4 + pp];
00176 JMM[7] = lf + lff * Quad[1] +
00177     udata[4 + pp] * (2 * lfg * udata[1 + pp] + lgg * udata[4 + pp]);
00178 JMM[12] =
00179     lff * udata[pp] * udata[2 + pp] + lfg * udata[2 + pp] * udata[3 + pp] +
00180     lfg * udata[pp] * udata[5 + pp] + lgg * udata[3 + pp] * udata[5 + pp];
00181 JMM[13] = lff * udata[1 + pp] * udata[2 + pp] +
00182     lfg * udata[2 + pp] * udata[4 + pp] +
00183     lfg * udata[1 + pp] * udata[5 + pp] +
00184     lgg * udata[4 + pp] * udata[5 + pp];
00185 JMM[14] = lf + lff * Quad[2] +
00186     udata[5 + pp] * (2 * lfg * udata[2 + pp] + lgg * udata[5 + pp]);
00187 JMM[18] = lg + lfg * (Quad[0] - Quad[3 + 0]) +
00188     (-lff + lgg) * udata[pp] * udata[3 + pp];
00189 JMM[19] = -(udata[3 + pp] * (lff * udata[1 + pp] + lfg * udata[4 + pp])) +
00190     udata[pp] * (lfg * udata[1 + pp] + lgg * udata[4 + pp]);
00191 JMM[20] = -(udata[3 + pp] * (lff * udata[2 + pp] + lfg * udata[5 + pp])) +
00192     udata[pp] * (lfg * udata[2 + pp] + lgg * udata[5 + pp]);
00193 JMM[21] = -lf + lgg * Quad[0] +
00194     udata[3 + pp] * (-2 * lfg * udata[pp] + lff * udata[3 + pp]);
00195 JMM[24] = udata[1 + pp] * (lfg * udata[pp] + lgg * udata[3 + pp]) -
00196     (lff * udata[pp] + lfg * udata[3 + pp]) * udata[4 + pp];
00197 JMM[25] = lg + lfg * (Quad[1] - Quad[4 + 0]) +
00198     (-lff + lgg) * udata[1 + pp] * udata[4 + pp];
00199 JMM[26] = -(udata[4 + pp] * (lff * udata[2 + pp] + lfg * udata[5 + pp])) +
00200     udata[1 + pp] * (lfg * udata[2 + pp] + lgg * udata[5 + pp]);
00201 JMM[27] = lgg * udata[pp] * udata[1 + pp] +
00202     lff * udata[3 + pp] * udata[4 + pp] -
00203     lfg * (udata[1 + pp] * udata[3 + pp] + udata[pp] * udata[4 + pp]);
00204 JMM[28] = -lf + lgg * Quad[1] +
00205     udata[4 + pp] * (-2 * lfg * udata[1 + pp] + lff * udata[4 + pp]);
00206 JMM[30] = udata[2 + pp] * (lfg * udata[pp] + lgg * udata[3 + pp]) -
00207     (lff * udata[pp] + lfg * udata[3 + pp]) * udata[5 + pp];
00208 JMM[31] = udata[2 + pp] * (lfg * udata[1 + pp] + lgg * udata[4 + pp]) -
00209     (lff * udata[1 + pp] + lfg * udata[4 + pp]) * udata[5 + pp];
00210 JMM[32] = lg + lfg * (Quad[2] - Quad[5 + 0]) +
00211     (-lff + lgg) * udata[2 + pp] * udata[5 + pp];
00212 JMM[33] = lgg * udata[pp] * udata[2 + pp] +
00213     lff * udata[3 + pp] * udata[5 + pp] -
00214     lfg * (udata[2 + pp] * udata[3 + pp] + udata[pp] * udata[5 + pp]);
00215 JMM[34] =
00216     lgg * udata[1 + pp] * udata[2 + pp] +
00217     lff * udata[4 + pp] * udata[5 + pp] -
00218     lfg * (udata[2 + pp] * udata[4 + pp] + udata[1 + pp] * udata[5 + pp]);
00219 JMM[35] = -lf + lgg * Quad[2] +
00220     udata[5 + pp] * (-2 * lfg * udata[2 + pp] + lff * udata[5 + pp]);
00221 // #pragma unroll_and_jam
00222 // #pragma distribute_point
00223 for (int i = 0; i < 6; i++) {
00224     for (int j = i + 1; j < 6; j++) {
00225         JMM[i * 6 + j] = JMM[j * 6 + i];
00226     }
00227 }
00228 // 4. Final values for temporal derivatives of field values
00229 h[0] = 0;
00230 h[1] = dxData[pp] * JMM[30] + dxData[1 + pp] * JMM[31] +
00231     dxData[2 + pp] * JMM[32] + dxData[3 + pp] * JMM[33] +
00232     dxData[4 + pp] * JMM[34] + dxData[5 + pp] * (-1 + JMM[35]);
00233 h[2] = -(dxData[pp] * JMM[24]) - dxData[1 + pp] * JMM[25] -
00234     dxData[2 + pp] * JMM[26] - dxData[3 + pp] * JMM[27] +
00235     dxData[4 + pp] * (1 - JMM[28]) - dxData[5 + pp] * JMM[29];

```

```

00236     h[3] = 0;
00237     h[4] = dxData[2 + pp];
00238     h[5] = -dxData[1 + pp];
00239     h[0] -= h[3] * JMM[3] + h[4] * JMM[4] + h[5] * JMM[5];
00240     h[1] -= h[3] * JMM[9] + h[4] * JMM[10] + h[5] * JMM[11];
00241     h[2] -= h[3] * JMM[15] + h[4] * JMM[16] + h[5] * JMM[17];
00242     // (1+Z)^-1 applies only to E components
00243     dudata[pp + 0] =
00244         h[2] * (-(JMM[2] * (1 + JMM[7])) + JMM[1] * JMM[8]) +
00245         h[1] * (JMM[2] * JMM[13] - JMM[1] * (1 + JMM[14])) +
00246         h[0] * (1 - JMM[8] * JMM[13] + JMM[14] + JMM[7] * (1 + JMM[14]));
00247     dudata[pp + 1] =
00248         h[2] * (JMM[2] * JMM[6] - (1 + JMM[0]) * JMM[8]) +
00249         h[1] * (1 - JMM[2] * JMM[12] + JMM[14] + JMM[0] * (1 + JMM[14])) +
00250         h[0] * (JMM[8] * JMM[12] - JMM[6] * (1 + JMM[14]));
00251     dudata[pp + 2] =
00252         h[2] * (1 - JMM[1] * JMM[6] + JMM[7] + JMM[0] * (1 + JMM[7])) +
00253         h[1] * (JMM[1] * JMM[12] - (1 + JMM[0]) * JMM[13]) +
00254         h[0] * (-(1 + JMM[7]) * JMM[12]) + JMM[6] * JMM[13]);
00255     pseudoDenom =
00256         -(1 + JMM[7]) * (-1 + JMM[2] * JMM[12])) +
00257         (JMM[2] * JMM[6] - JMM[8]) * JMM[13] + JMM[14] + JMM[7] * JMM[14] +
00258         JMM[0] * (1 + JMM[7] - JMM[8] * JMM[13] + (1 + JMM[7]) * JMM[14]) -
00259         JMM[1] * (-(JMM[8] * JMM[12]) + JMM[6] * (1 + JMM[14]));
00260     dudata[pp + 0] /= pseudoDenom;
00261     dudata[pp + 1] /= pseudoDenom;
00262     dudata[pp + 2] /= pseudoDenom;
00263     dudata[pp + 3] = h[3];
00264     dudata[pp + 4] = h[4];
00265     dudata[pp + 5] = h[5];
00266 }
00267 return;
00268 }
00269
00270 /// only under-the-hood-callable Maxwell propagation in 2D
00271 // unused parameters 2-4 for compliance with CVRhsFn
00272 // same as the respective nonlinear function without nonlinear terms
00273 void linear2DProp(LatticePatch *data, N_Vector u, N_Vector udot, int *c) {
00274
00275     sunrealtype *duData = data->duData;
00276     sunrealtype *dxData = data->buffData[1 - 1];
00277     sunrealtype *dyData = data->buffData[2 - 1];
00278
00279     data->exchangeGhostCells(1);
00280     data->rotateIntoEigen(1);
00281     data->derive(1);
00282     data->derotate(1, dxData);
00283     data->exchangeGhostCells(2);
00284     data->rotateIntoEigen(2);
00285     data->derive(2);
00286     data->derotate(2, dyData);
00287
00288     int totalNP = data->discreteSize();
00289     int pp = 0;
00290 #pragma distribute_point
00291     for (int i = 0; i < totalNP; i++) {
00292         pp = i * 6;
00293         duData[pp + 0] = dyData[pp + 5];
00294         duData[pp + 1] = -dxData[pp + 5];
00295         duData[pp + 2] = -dyData[pp + 3] + dxData[pp + 4];
00296         duData[pp + 3] = -dyData[pp + 2];
00297         duData[pp + 4] = dxData[pp + 2];
00298         duData[pp + 5] = dyData[pp + 0] - dxData[pp + 1];
00299     }
00300 }
00301
00302 /// nonlinear 2D HE propagation
00303 void nonlinear2DProp(LatticePatch *data, N_Vector u, N_Vector udot, int *c) {
00304
00305     sunrealtype *dxData = data->buffData[1 - 1];
00306     sunrealtype *dyData = data->buffData[2 - 1];
00307
00308     data->exchangeGhostCells(1);
00309     data->rotateIntoEigen(1);
00310     data->derive(1);
00311     data->derotate(1, dxData);
00312     data->exchangeGhostCells(2);
00313     data->rotateIntoEigen(2);
00314     data->derive(2);
00315     data->derotate(2, dyData);
00316
00317     sunrealtype f = NAN, g = NAN;
00318     sunrealtype lf = NAN, lff = NAN, lfg = NAN, lg = NAN, lgg = NAN;
00319     array<sunrealtype, 36> JMM;
00320     array<sunrealtype, 6> Quad;
00321     array<sunrealtype, 6> h;
00322     sunrealtype pseudoDenom = NAN;

```

```

00323 sunrealtype *udata = nullptr, *dudata = nullptr;
00324 udata = NV_DATA_P(u);
00325 dudata = NV_DATA_P(udot);
00326 int totalNP = data->discreteSize();
00327 ///pragma distribute_point
00328 ///pragma unroll_and_jam
00329 for (int pp = 0; pp < totalNP * 6; pp += 6) {
00330     /// 1
00331     f = 0.5 * ((Quad[0] = udata[pp] * udata[pp]) +
00332               (Quad[1] = udata[pp + 1] * udata[pp + 1]) +
00333               (Quad[2] = udata[pp + 2] * udata[pp + 2]) -
00334               (Quad[3] = udata[pp + 3] * udata[pp + 3]) -
00335               (Quad[4] = udata[pp + 4] * udata[pp + 4]) -
00336               (Quad[5] = udata[pp + 5] * udata[pp + 5]));
00337     g = udata[pp] * udata[pp + 3] + udata[pp + 1] * udata[pp + 4] +
00338         udata[pp + 2] * udata[pp + 5];
00339     /// 2
00340     switch (*c) {
00341     case 0:
00342         lf = 0;
00343         lff = 0;
00344         lfg = 0;
00345         lg = 0;
00346         lgg = 0;
00347         break;
00348     case 2:
00349         lf = 0.000354046449700427580438254 * f * f +
00350             0.000191775160254398272737387 * g * g;
00351         lff = 0.0007080928994008551608765075 * f;
00352         lfg = 0.0003835503205087965454747749 * g;
00353         lg = 0.0003835503205087965454747749 * f * g;
00354         lgg = 0.0003835503205087965454747749 * f;
00355         break;
00356     case 1:
00357         lf = 0.000206527095658582755255648 * f;
00358         lff = 0.000206527095658582755255648;
00359         lfg = 0;
00360         lg = 0.0003614224174025198216973841 * g;
00361         lgg = 0.0003614224174025198216973841;
00362         break;
00363     case 3:
00364         lf = (0.000206527095658582755255648 + 0.000354046449700427580438254 * f) *
00365             f +
00366             0.000191775160254398272737387 * g * g;
00367         lff = 0.000206527095658582755255648 + 0.000708092899400855160876508 * f;
00368         lfg = 0.0003835503205087965454747749 * g;
00369         lg = (0.000361422417402519821697384 + 0.000383550320508796545474775 * f) *
00370             g;
00371         lgg = 0.000361422417402519821697384 + 0.000383550320508796545474775 * f;
00372         break;
00373     default:
00374         errorKill(
00375             "You need to specify a correct order in the weak-field expansion.");
00376     }
00377     /// 3
00378     JMM[0] = lf + lff * Quad[0] +
00379         udata[3 + pp] * (2 * lfg * udata[pp] + lgg * udata[3 + pp]);
00380     JMM[6] =
00381         lff * udata[pp] * udata[1 + pp] + lfg * udata[1 + pp] * udata[3 + pp] +
00382         lfg * udata[pp] * udata[4 + pp] + lgg * udata[3 + pp] * udata[4 + pp];
00383     JMM[7] = lf + lff * Quad[1] +
00384         udata[4 + pp] * (2 * lfg * udata[1 + pp] + lgg * udata[4 + pp]);
00385     JMM[12] =
00386         lff * udata[pp] * udata[2 + pp] + lfg * udata[2 + pp] * udata[3 + pp] +
00387         lfg * udata[pp] * udata[5 + pp] + lgg * udata[3 + pp] * udata[5 + pp];
00388     JMM[13] = lff * udata[1 + pp] * udata[2 + pp] +
00389         lfg * udata[2 + pp] * udata[4 + pp] +
00390         lfg * udata[1 + pp] * udata[5 + pp] +
00391         lgg * udata[4 + pp] * udata[5 + pp];
00392     JMM[14] = lf + lff * Quad[2] +
00393         udata[5 + pp] * (2 * lfg * udata[2 + pp] + lgg * udata[5 + pp]);
00394     JMM[18] = lg + lfg * (Quad[0] - Quad[3 + 0]) +
00395         (-lff + lgg) * udata[pp] * udata[3 + pp];
00396     JMM[19] = -(udata[3 + pp] * (lff * udata[1 + pp] + lfg * udata[4 + pp])) +
00397         udata[pp] * (lfg * udata[1 + pp] + lgg * udata[4 + pp]);
00398     JMM[20] = -(udata[3 + pp] * (lff * udata[2 + pp] + lfg * udata[5 + pp])) +
00399         udata[pp] * (lfg * udata[2 + pp] + lgg * udata[5 + pp]);
00400     JMM[21] = -lf + lgg * Quad[0] +
00401         udata[3 + pp] * (-2 * lfg * udata[pp] + lff * udata[3 + pp]);
00402     JMM[24] = udata[1 + pp] * (lfg * udata[pp] + lgg * udata[3 + pp]) -
00403         (lff * udata[pp] + lfg * udata[3 + pp]) * udata[4 + pp];
00404     JMM[25] = lg + lfg * (Quad[1] - Quad[4 + 0]) +
00405         (-lff + lgg) * udata[1 + pp] * udata[4 + pp];
00406     JMM[26] = -(udata[4 + pp] * (lff * udata[2 + pp] + lfg * udata[5 + pp])) +
00407         udata[1 + pp] * (lfg * udata[2 + pp] + lgg * udata[5 + pp]);
00408     JMM[27] = lgg * udata[pp] * udata[1 + pp] +
00409         lff * udata[3 + pp] * udata[4 + pp] -

```



```

00410         lfg * (udata[1 + pp] * udata[3 + pp] + udata[pp] * udata[4 + pp]);
00411 JMM[28] = -lff + lgg * Quad[1] +
00412         udata[4 + pp] * (-2 * lfg * udata[1 + pp] + lff * udata[4 + pp]);
00413 JMM[30] = udata[2 + pp] * (lfg * udata[pp] + lgg * udata[3 + pp]) -
00414         (lff * udata[pp] + lfg * udata[3 + pp]) * udata[5 + pp];
00415 JMM[31] = udata[2 + pp] * (lfg * udata[1 + pp] + lgg * udata[4 + pp]) -
00416         (lff * udata[1 + pp] + lfg * udata[4 + pp]) * udata[5 + pp];
00417 JMM[32] = lg + lfg * (Quad[2] - Quad[5 + 0]) +
00418         (-lff + lgg) * udata[2 + pp] * udata[5 + pp];
00419 JMM[33] = lgg * udata[pp] * udata[2 + pp] +
00420         lff * udata[3 + pp] * udata[5 + pp] -
00421         lfg * (udata[2 + pp] * udata[3 + pp] + udata[pp] * udata[5 + pp]);
00422 JMM[34] =
00423         lgg * udata[1 + pp] * udata[2 + pp] +
00424         lff * udata[4 + pp] * udata[5 + pp] -
00425         lfg * (udata[2 + pp] * udata[4 + pp] + udata[1 + pp] * udata[5 + pp]);
00426 JMM[35] = -lff + lgg * Quad[2] +
00427         udata[5 + pp] * (-2 * lfg * udata[2 + pp] + lff * udata[5 + pp]);
00428 // 4
00429 // #pragma distribute_point
00430 // #pragma unroll_and_jam
00431 for (int i = 0; i < 6; i++) {
00432     for (int j = i + 1; j < 6; j++) {
00433         JMM[i * 6 + j] = JMM[j * 6 + i];
00434     }
00435 }
00436 h[0] = 0;
00437 h[1] = dxData[pp] * JMM[30] + dxData[1 + pp] * JMM[31] +
00438         dxData[2 + pp] * JMM[32] + dxData[3 + pp] * JMM[33] +
00439         dxData[4 + pp] * JMM[34] + dxData[5 + pp] * (-1 + JMM[35]);
00440 h[2] = -(dxData[pp] * JMM[24]) - dxData[1 + pp] * JMM[25] -
00441         dxData[2 + pp] * JMM[26] - dxData[3 + pp] * JMM[27] +
00442         dxData[4 + pp] * (1 - JMM[28]) - dxData[5 + pp] * JMM[29];
00443 h[3] = 0;
00444 h[4] = dxData[2 + pp];
00445 h[5] = -dxData[1 + pp];
00446 h[0] += -(dyData[pp] * JMM[30]) - dyData[1 + pp] * JMM[31] -
00447         dyData[2 + pp] * JMM[32] - dyData[3 + pp] * JMM[33] -
00448         dyData[4 + pp] * JMM[34] + dyData[5 + pp] * (1 - JMM[35]);
00449 h[1] += 0;
00450 h[2] += dyData[pp] * JMM[18] + dyData[1 + pp] * JMM[19] +
00451         dyData[2 + pp] * JMM[20] + dyData[3 + pp] * (-1 + JMM[21]) +
00452         dyData[4 + pp] * JMM[22] + dyData[5 + pp] * JMM[23];
00453 h[3] += -dyData[2 + pp];
00454 h[4] += 0;
00455 h[5] += dyData[pp];
00456 h[0] -= h[3] * JMM[3] + h[4] * JMM[4] + h[5] * JMM[5];
00457 h[1] -= h[3] * JMM[9] + h[4] * JMM[10] + h[5] * JMM[11];
00458 h[2] -= h[3] * JMM[15] + h[4] * JMM[16] + h[5] * JMM[17];
00459 dudata[pp + 0] =
00460         h[2] * (-(JMM[2] * (1 + JMM[7])) + JMM[1] * JMM[8]) +
00461         h[1] * (JMM[2] * JMM[13] - JMM[1] * (1 + JMM[14])) +
00462         h[0] * (1 - JMM[8] * JMM[13] + JMM[14] + JMM[7] * (1 + JMM[14]));
00463 dudata[pp + 1] =
00464         h[2] * (JMM[2] * JMM[6] - (1 + JMM[0]) * JMM[8]) +
00465         h[1] * (1 - JMM[2] * JMM[12] + JMM[14] + JMM[0] * (1 + JMM[14])) +
00466         h[0] * (JMM[8] * JMM[12] - JMM[6] * (1 + JMM[14]));
00467 dudata[pp + 2] =
00468         h[2] * (1 - JMM[1] * JMM[6] + JMM[7] + JMM[0] * (1 + JMM[7])) +
00469         h[1] * (JMM[1] * JMM[12] - (1 + JMM[0]) * JMM[13]) +
00470         h[0] * (-(1 + JMM[7]) * JMM[12]) + JMM[6] * JMM[13]);
00471 pseudoDenom =
00472         -(1 + JMM[7]) * (-1 + JMM[2] * JMM[12])) +
00473         (JMM[2] * JMM[6] - JMM[8]) * JMM[13] + JMM[14] + JMM[7] * JMM[14] +
00474         JMM[0] * (1 + JMM[7] - JMM[8] * JMM[13] + (1 + JMM[7]) * JMM[14]) -
00475         JMM[1] * (-JMM[8] * JMM[12]) + JMM[6] * (1 + JMM[14]);
00476 dudata[pp + 0] /= pseudoDenom;
00477 dudata[pp + 1] /= pseudoDenom;
00478 dudata[pp + 2] /= pseudoDenom;
00479 dudata[pp + 3] = h[3];
00480 dudata[pp + 4] = h[4];
00481 dudata[pp + 5] = h[5];
00482 }
00483 return;
00484 }
00485
00486 /// only under-the-hood-callable Maxwell propagation in 3D
00487 /// unused parameters 2-4 for compliance with CVRhsFn
00488 /// same as the respective nonlinear function without nonlinear terms
00489 void linear3DProp(LatticePatch *data, N_Vector u, N_Vector udot, int *c) {
00490     sunrealtype *duData = data->duData;
00491     sunrealtype *dxData = data->buffData[1 - 1];
00492     sunrealtype *dyData = data->buffData[2 - 1];
00493     sunrealtype *dzData = data->buffData[3 - 1];
00494
00495     /* Under the hood call of point-to-point or collective communication */

```

```

00497 // Point-to-Point:
00498 data->exchangeGhostCells(1);
00499 data->rotateIntoEigen(1);
00500 data->derive(1);
00501 data->derotate(1, dxData);
00502 data->exchangeGhostCells(2);
00503 data->rotateIntoEigen(2);
00504 data->derive(2);
00505 data->derotate(2, dyData);
00506 data->exchangeGhostCells(3);
00507 data->rotateIntoEigen(3);
00508 data->derive(3);
00509 data->derotate(3, dzData);
00510
00511 // Collective:
00512 /*
00513     data->exchangeGhostCells3D();
00514     data->rotateIntoEigen3D();
00515     data->derive(1);
00516     data->derotate(1, dxData);
00517     data->derive(2);
00518     data->derotate(2, dyData);
00519     data->derive(3);
00520     data->derotate(3, dzData);
00521 */
00522
00523 int totalNP = data->discreteSize();
00524 int pp = 0;
00525 #pragma distribute_point
00526 for (int i = 0; i < totalNP; i++) {
00527     pp = i * 6;
00528     duData[pp + 0] = dyData[pp + 5] - dzData[pp + 4];
00529     duData[pp + 1] = dzData[pp + 3] - dxData[pp + 5];
00530     duData[pp + 2] = dxData[pp + 4] - dyData[pp + 3];
00531     duData[pp + 3] = -dyData[pp + 2] + dzData[pp + 1];
00532     duData[pp + 4] = -dzData[pp + 0] + dxData[pp + 2];
00533     duData[pp + 5] = -dxData[pp + 1] + dyData[pp + 0];
00534 }
00535 }
00536
00537 /// nonlinear 3D HE propagation
00538 void nonlinear3DProp(LatticePatch *data, N_Vector u, N_Vector udot, int *c) {
00539
00540     sunrealtype *dxData = data->buffData[1 - 1];
00541     sunrealtype *dyData = data->buffData[2 - 1];
00542     sunrealtype *dzData = data->buffData[3 - 1];
00543
00544     /* Under the hood call of point-to-point or collective communication */
00545     // Point-to-Point:
00546
00547     data->exchangeGhostCells(1);
00548     data->rotateIntoEigen(1);
00549     data->derive(1);
00550     data->derotate(1, dxData);
00551     data->exchangeGhostCells(2);
00552     data->rotateIntoEigen(2);
00553     data->derive(2);
00554     data->derotate(2, dyData);
00555     data->exchangeGhostCells(3);
00556     data->rotateIntoEigen(3);
00557     data->derive(3);
00558     data->derotate(3, dzData);
00559
00560     // Collective:
00561     /*
00562         data->exchangeGhostCells3D();
00563         data->rotateIntoEigen3D();
00564         data->derive(1);
00565         data->derotate(1, dxData);
00566         data->derive(2);
00567         data->derotate(2, dyData);
00568         data->derive(3);
00569         data->derotate(3, dzData);
00570     */
00571
00572     sunrealtype f = NAN, g = NAN;
00573     sunrealtype lf = NAN, lff = NAN, lfg = NAN, lg = NAN, lgg = NAN;
00574     array<sunrealtype, 36> JMM;
00575     array<sunrealtype, 6> Quad;
00576     array<sunrealtype, 6> h;
00577     sunrealtype pseudoDenom = NAN;
00578     sunrealtype *udata = nullptr, *dudata = nullptr;
00579     udata = NV_DATA_P(u);
00580     dudata = NV_DATA_P(udot);
00581     int totalNP = data->discreteSize();
00582     ///#pragma distribute_point
00583     ///#pragma unroll_and_jam

```

```

00584   for (int pp = 0; pp < totalNP * 6; pp += 6) {
00585       // 1
00586       f = 0.5 * ((Quad[0] = udata[pp] * udata[pp]) +
00587                 (Quad[1] = udata[pp + 1] * udata[pp + 1]) +
00588                 (Quad[2] = udata[pp + 2] * udata[pp + 2]) -
00589                 (Quad[3] = udata[pp + 3] * udata[pp + 3]) -
00590                 (Quad[4] = udata[pp + 4] * udata[pp + 4]) -
00591                 (Quad[5] = udata[pp + 5] * udata[pp + 5]));
00592       g = udata[pp] * udata[pp + 3] + udata[pp + 1] * udata[pp + 4] +
00593         udata[pp + 2] * udata[pp + 5];
00594       // 2
00595       switch (*c) {
00596       case 0:
00597         lf = 0;
00598         lff = 0;
00599         lfg = 0;
00600         lg = 0;
00601         lgg = 0;
00602         break;
00603       case 2:
00604         lf = 0.000354046449700427580438254 * f * f +
00605           0.000191775160254398272737387 * g * g;
00606         lff = 0.0007080928994008551608765075 * f;
00607         lfg = 0.0003835503205087965454747749 * g;
00608         lg = 0.0003835503205087965454747749 * f * g;
00609         lgg = 0.0003835503205087965454747749 * f;
00610         break;
00611       case 1:
00612         lf = 0.000206527095658582755255648 * f;
00613         lff = 0.000206527095658582755255648;
00614         lfg = 0;
00615         lg = 0.0003614224174025198216973841 * g;
00616         lgg = 0.0003614224174025198216973841;
00617         break;
00618       case 3:
00619         lf = (0.000206527095658582755255648 + 0.000354046449700427580438254 * f) *
00620           f +
00621           0.000191775160254398272737387 * g * g;
00622         lff = 0.000206527095658582755255648 + 0.000708092899400855160876508 * f;
00623         lfg = 0.0003835503205087965454747749 * g;
00624         lg = (0.000361422417402519821697384 + 0.000383550320508796545474775 * f) *
00625           g;
00626         lgg = 0.000361422417402519821697384 + 0.000383550320508796545474775 * f;
00627         break;
00628       default:
00629         errorKill(
00630           "You need to specify a correct order in the weak-field expansion.");
00631       }
00632       // 3
00633       JMM[0] = lf + lff * Quad[0] +
00634         udata[3 + pp] * (2 * lfg * udata[pp] + lgg * udata[3 + pp]);
00635       JMM[6] =
00636         lff * udata[pp] * udata[1 + pp] + lfg * udata[1 + pp] * udata[3 + pp] +
00637         lfg * udata[pp] * udata[4 + pp] + lgg * udata[3 + pp] * udata[4 + pp];
00638       JMM[7] = lf + lff * Quad[1] +
00639         udata[4 + pp] * (2 * lfg * udata[1 + pp] + lgg * udata[4 + pp]);
00640       JMM[12] =
00641         lff * udata[pp] * udata[2 + pp] + lfg * udata[2 + pp] * udata[3 + pp] +
00642         lfg * udata[pp] * udata[5 + pp] + lgg * udata[3 + pp] * udata[5 + pp];
00643       JMM[13] = lff * udata[1 + pp] * udata[2 + pp] +
00644         lfg * udata[2 + pp] * udata[4 + pp] +
00645         lfg * udata[1 + pp] * udata[5 + pp] +
00646         lgg * udata[4 + pp] * udata[5 + pp];
00647       JMM[14] = lf + lff * Quad[2] +
00648         udata[5 + pp] * (2 * lfg * udata[2 + pp] + lgg * udata[5 + pp]);
00649       JMM[18] = lg + lfg * (Quad[0] - Quad[3 + 0]) +
00650         (-lff + lgg) * udata[pp] * udata[3 + pp];
00651       JMM[19] = -(udata[3 + pp] * (lff * udata[1 + pp] + lfg * udata[4 + pp])) +
00652         udata[pp] * (lfg * udata[1 + pp] + lgg * udata[4 + pp]);
00653       JMM[20] = -(udata[3 + pp] * (lff * udata[2 + pp] + lfg * udata[5 + pp])) +
00654         udata[pp] * (lfg * udata[2 + pp] + lgg * udata[5 + pp]);
00655       JMM[21] = -lf + lgg * Quad[0] +
00656         udata[3 + pp] * (-2 * lfg * udata[pp] + lff * udata[3 + pp]);
00657       JMM[24] = udata[1 + pp] * (lfg * udata[pp] + lgg * udata[3 + pp]) -
00658         (lff * udata[pp] + lfg * udata[3 + pp]) * udata[4 + pp];
00659       JMM[25] = lg + lfg * (Quad[1] - Quad[4 + 0]) +
00660         (-lff + lgg) * udata[1 + pp] * udata[4 + pp];
00661       JMM[26] = -(udata[4 + pp] * (lff * udata[2 + pp] + lfg * udata[5 + pp])) +
00662         udata[1 + pp] * (lfg * udata[2 + pp] + lgg * udata[5 + pp]);
00663       JMM[27] = lgg * udata[pp] * udata[1 + pp] +
00664         lff * udata[3 + pp] * udata[4 + pp] -
00665         lfg * (udata[1 + pp] * udata[3 + pp] + udata[pp] * udata[4 + pp]);
00666       JMM[28] = -lf + lgg * Quad[1] +
00667         udata[4 + pp] * (-2 * lfg * udata[1 + pp] + lff * udata[4 + pp]);
00668       JMM[30] = udata[2 + pp] * (lfg * udata[pp] + lgg * udata[3 + pp]) -
00669         (lff * udata[pp] + lfg * udata[3 + pp]) * udata[5 + pp];
00670       JMM[31] = udata[2 + pp] * (lfg * udata[1 + pp] + lgg * udata[4 + pp]) -

```

```

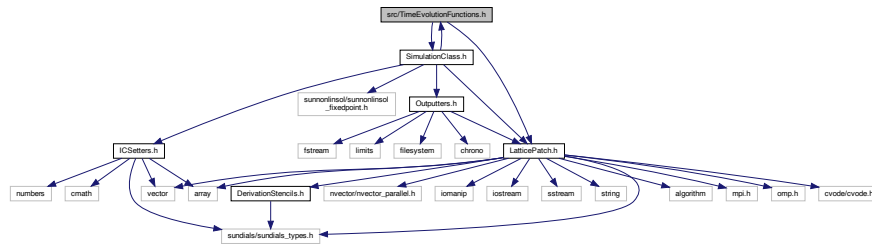
00671         (lff * udata[1 + pp] + lfg * udata[4 + pp]) * udata[5 + pp];
00672 JMM[32] = lg + lfg * (Quad[2] - Quad[5 + 0]) +
00673         (-lff + lgg) * udata[2 + pp] * udata[5 + pp];
00674 JMM[33] = lgg * udata[pp] * udata[2 + pp] +
00675         lff * udata[3 + pp] * udata[5 + pp] -
00676         lfg * (udata[2 + pp] * udata[3 + pp] + udata[pp] * udata[5 + pp]);
00677 JMM[34] =
00678         lgg * udata[1 + pp] * udata[2 + pp] +
00679         lff * udata[4 + pp] * udata[5 + pp] -
00680         lfg * (udata[2 + pp] * udata[4 + pp] + udata[1 + pp] * udata[5 + pp]);
00681 JMM[35] = -lf + lgg * Quad[2] +
00682         udata[5 + pp] * (-2 * lfg * udata[2 + pp] + lff * udata[5 + pp]);
00683 // 4
00684 //#pragma distribute_point
00685 //#pragma unroll_and_jam
00686 for (int i = 0; i < 6; i++) {
00687     for (int j = i + 1; j < 6; j++) {
00688         JMM[i * 6 + j] = JMM[j * 6 + i];
00689     }
00690 }
00691 h[0] = 0;
00692 h[1] = dxData[pp] * JMM[30] + dxData[1 + pp] * JMM[31] +
00693         dxData[2 + pp] * JMM[32] + dxData[3 + pp] * JMM[33] +
00694         dxData[4 + pp] * JMM[34] + dxData[5 + pp] * (-1 + JMM[35]);
00695 h[2] = -(dxData[pp] * JMM[24]) - dxData[1 + pp] * JMM[25] -
00696         dxData[2 + pp] * JMM[26] - dxData[3 + pp] * JMM[27] +
00697         dxData[4 + pp] * (1 - JMM[28]) - dxData[5 + pp] * JMM[29];
00698 h[3] = 0;
00699 h[4] = dxData[2 + pp];
00700 h[5] = -dxData[1 + pp];
00701 h[0] += -(dyData[pp] * JMM[30]) - dyData[1 + pp] * JMM[31] -
00702         dyData[2 + pp] * JMM[32] - dyData[3 + pp] * JMM[33] -
00703         dyData[4 + pp] * JMM[34] + dyData[5 + pp] * (1 - JMM[35]);
00704 h[1] += 0;
00705 h[2] += dyData[pp] * JMM[18] + dyData[1 + pp] * JMM[19] +
00706         dyData[2 + pp] * JMM[20] + dyData[3 + pp] * (-1 + JMM[21]) +
00707         dyData[4 + pp] * JMM[22] + dyData[5 + pp] * JMM[23];
00708 h[3] += -dyData[2 + pp];
00709 h[4] += 0;
00710 h[5] += dyData[pp];
00711 h[0] += dzData[pp] * JMM[24] + dzData[1 + pp] * JMM[25] +
00712         dzData[2 + pp] * JMM[26] + dzData[3 + pp] * JMM[27] +
00713         dzData[4 + pp] * (-1 + JMM[28]) + dzData[5 + pp] * JMM[29];
00714 h[1] += -(dzData[pp] * JMM[18]) - dzData[1 + pp] * JMM[19] -
00715         dzData[2 + pp] * JMM[20] + dzData[3 + pp] * (1 - JMM[21]) -
00716         dzData[4 + pp] * JMM[22] - dzData[5 + pp] * JMM[23];
00717 h[2] += 0;
00718 h[3] += dzData[1 + pp];
00719 h[4] += -dzData[pp];
00720 h[5] += 0;
00721 h[0] -= h[3] * JMM[3] + h[4] * JMM[4] + h[5] * JMM[5];
00722 h[1] -= h[3] * JMM[9] + h[4] * JMM[10] + h[5] * JMM[11];
00723 h[2] -= h[3] * JMM[15] + h[4] * JMM[16] + h[5] * JMM[17];
00724 dudata[pp + 0] =
00725         h[2] * (-JMM[2] * (1 + JMM[7])) + JMM[1] * JMM[8]) +
00726         h[1] * (JMM[2] * JMM[13] - JMM[1] * (1 + JMM[14])) +
00727         h[0] * (1 - JMM[8] * JMM[13] + JMM[14] + JMM[7] * (1 + JMM[14]));
00728 dudata[pp + 1] =
00729         h[2] * (JMM[2] * JMM[6] - (1 + JMM[0]) * JMM[8]) +
00730         h[1] * (1 - JMM[2] * JMM[12] + JMM[14] + JMM[0] * (1 + JMM[14])) +
00731         h[0] * (JMM[8] * JMM[12] - JMM[6] * (1 + JMM[14]));
00732 dudata[pp + 2] =
00733         h[2] * (1 - JMM[1] * JMM[6] + JMM[7] + JMM[0] * (1 + JMM[7])) +
00734         h[1] * (JMM[1] * JMM[12] - (1 + JMM[0]) * JMM[13]) +
00735         h[0] * (-((1 + JMM[7]) * JMM[12]) + JMM[6] * JMM[13]);
00736 pseudoDenom =
00737         -((1 + JMM[7]) * (-1 + JMM[2] * JMM[12])) +
00738         (JMM[2] * JMM[6] - JMM[8]) * JMM[13] + JMM[14] + JMM[7] * JMM[14] +
00739         JMM[0] * (1 + JMM[7] - JMM[8] * JMM[13] + (1 + JMM[7]) * JMM[14]) -
00740         JMM[1] * (-JMM[8] * JMM[12]) + JMM[6] * (1 + JMM[14]));
00741 dudata[pp + 0] /= pseudoDenom;
00742 dudata[pp + 1] /= pseudoDenom;
00743 dudata[pp + 2] /= pseudoDenom;
00744 dudata[pp + 3] = h[3];
00745 dudata[pp + 4] = h[4];
00746 dudata[pp + 5] = h[5];
00747 }
00748 return;
00749 }

```

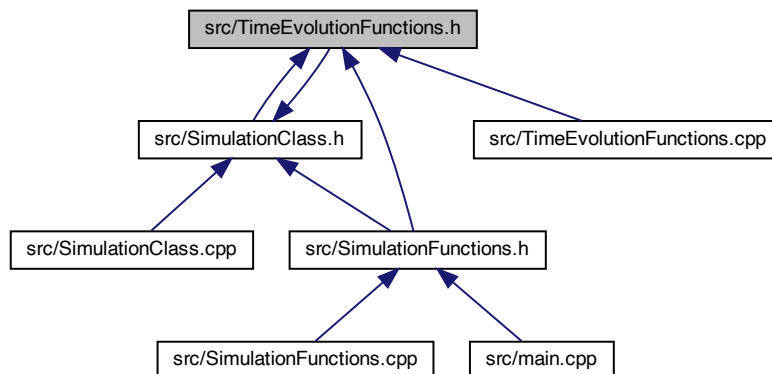
Functions to propagate data vectors in time according to Maxwell's equations, and various orders in the HE weak-field expansion.

```
#include "SimulationClass.h"
```

Include dependency graph for TimeEvolutionFunctions.h:



This graph shows which files directly or indirectly include this file:



- class TimeEvolution

*monostate **TimeEvolution** Class to propagate the field data in time in a given order of the HE weak-field expansion*

- 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 #pragma distribute_point
00063 for (int i = 0; i < totalNP; i++) {
00064     pp = i * 6;
00065     /*
00066     simple vacuum Maxwell equations for spatial derivative only in x-direction
00067     temporal derivative is approximated by spatial derivative according to the
00068     numerical scheme with Jacobi=0 -> no polarization or magnetization terms
00069     */
00070     duData[pp + 0] = 0;
00071     duData[pp + 1] = -dxData[pp + 5];
00072     duData[pp + 2] = dxData[pp + 4];
00073     duData[pp + 3] = 0;
00074     duData[pp + 4] = dxData[pp + 2];
00075     duData[pp + 5] = -dxData[pp + 1];
```

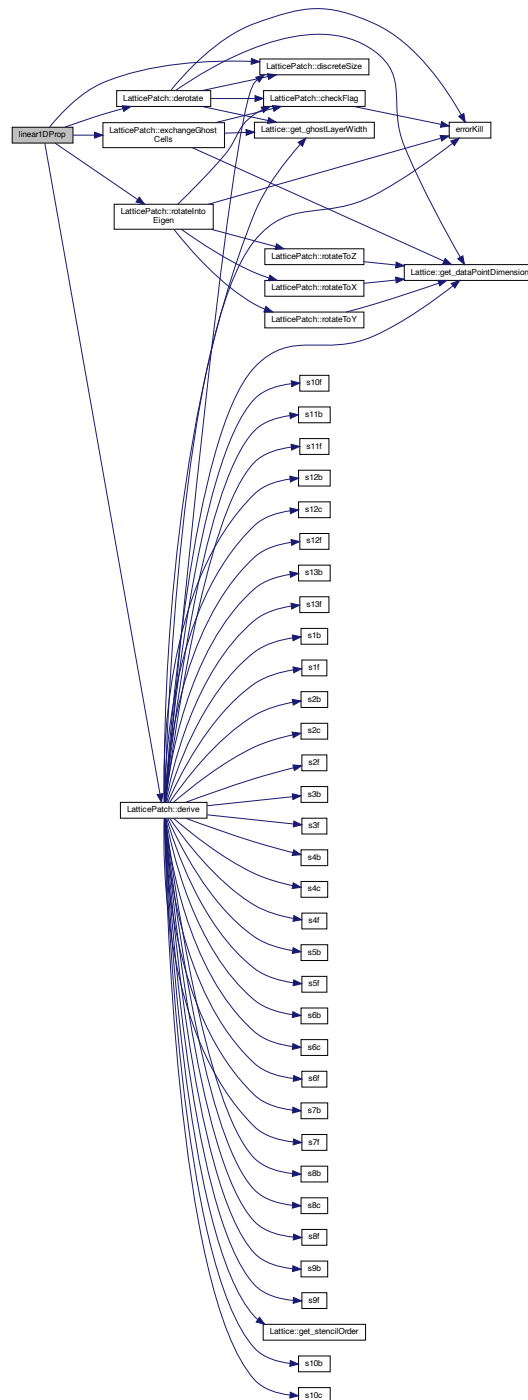
```

00076     }
00077 }

```

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.

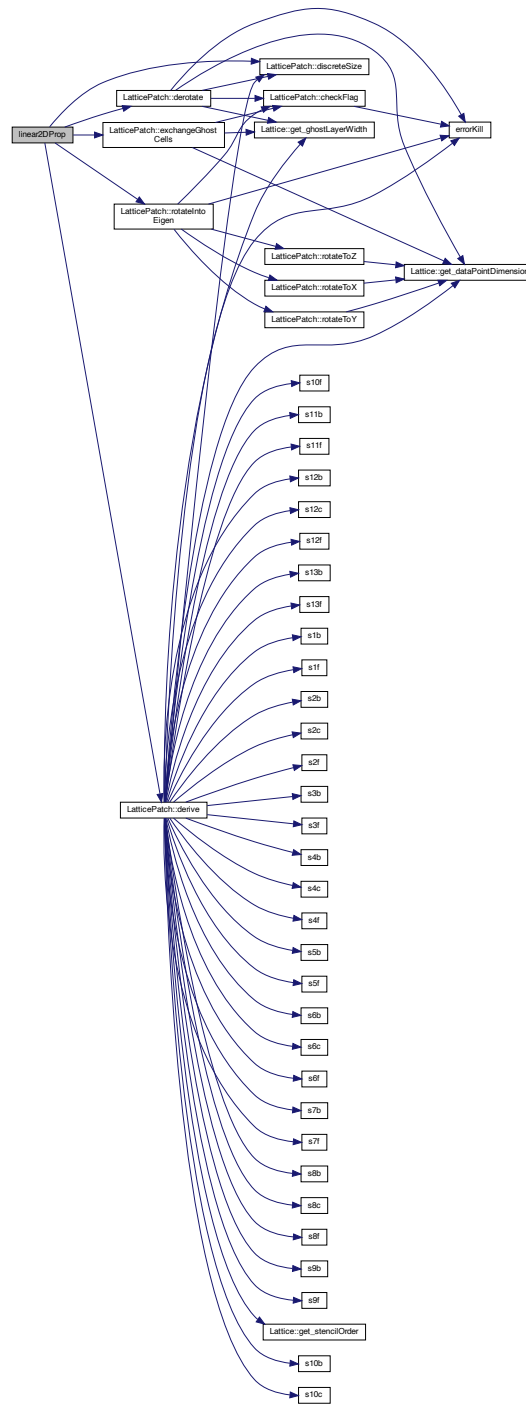
Maxwell propagation function for 2D – only for reference.

Definition at line 273 of file [TimeEvolutionFunctions.cpp](#).

```
00273 {
00274
00275     sunrealtype *duData = data->duData;
00276     sunrealtype *dxData = data->buffData[1 - 1];
00277     sunrealtype *dyData = data->buffData[2 - 1];
00278
00279     data->exchangeGhostCells(1);
00280     data->rotateIntoEigen(1);
00281     data->derive(1);
00282     data->derotate(1, dxData);
00283     data->exchangeGhostCells(2);
00284     data->rotateIntoEigen(2);
00285     data->derive(2);
00286     data->derotate(2, dyData);
00287
00288     int totalNP = data->discreteSize();
00289     int pp = 0;
00290     #pragma distribute_point
00291     for (int i = 0; i < totalNP; i++) {
00292         pp = i * 6;
00293         duData[pp + 0] = dyData[pp + 5];
00294         duData[pp + 1] = -dxData[pp + 5];
00295         duData[pp + 2] = -dyData[pp + 3] + dxData[pp + 4];
00296         duData[pp + 3] = -dyData[pp + 2];
00297         duData[pp + 4] = dxData[pp + 2];
00298         duData[pp + 5] = dyData[pp + 0] - dxData[pp + 1];
00299     }
00300 }
```

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

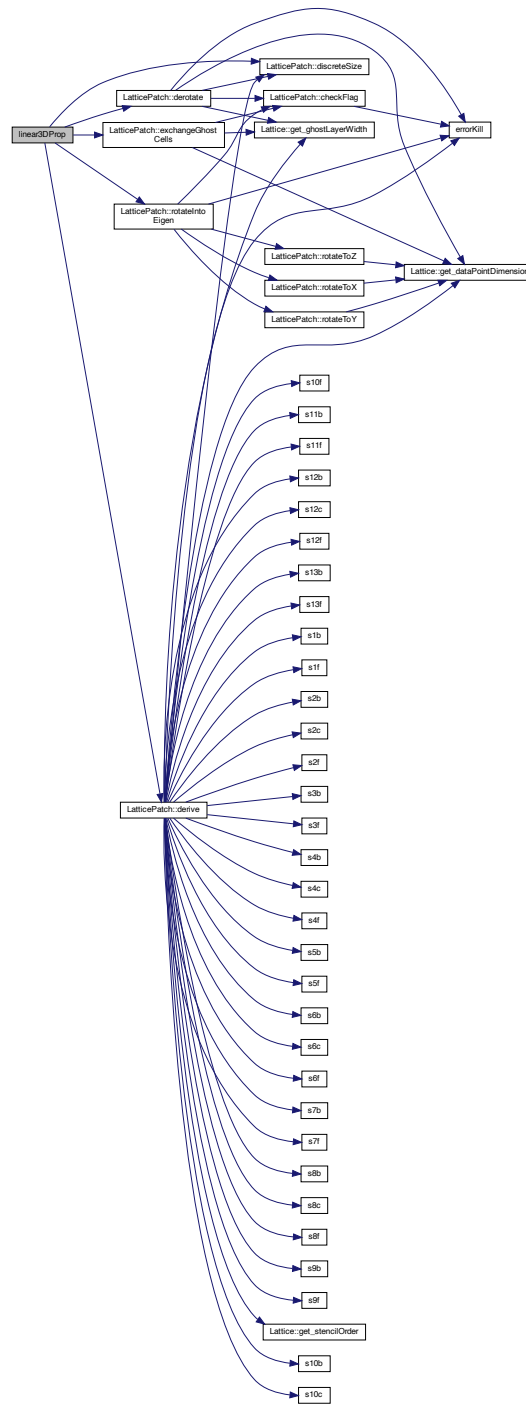
```

00489
00490
00491     sunrealtype *duData = data->duData;
00492     sunrealtype *dxData = data->buffData[1 - 1];
00493     sunrealtype *dyData = data->buffData[2 - 1];
00494     sunrealtype *dzData = data->buffData[3 - 1];
00495
00496     /* Under the hood call of point-to-point or collective communication */
00497     // Point-to-Point:
00498     data->exchangeGhostCells(1);
00499     data->rotateIntoEigen(1);
00500     data->derive(1);
00501     data->derotate(1, dxData);
00502     data->exchangeGhostCells(2);
00503     data->rotateIntoEigen(2);
00504     data->derive(2);
00505     data->derotate(2, dyData);
00506     data->exchangeGhostCells(3);
00507     data->rotateIntoEigen(3);
00508     data->derive(3);
00509     data->derotate(3, dzData);
00510
00511     // Collective:
00512     /*
00513         data->exchangeGhostCells3D();
00514         data->rotateIntoEigen3D();
00515         data->derive(1);
00516         data->derotate(1, dxData);
00517         data->derive(2);
00518         data->derotate(2, dyData);
00519         data->derive(3);
00520         data->derotate(3, dzData);
00521     */
00522
00523     int totalNP = data->discreteSize();
00524     int pp = 0;
00525     #pragma distribute_point
00526     for (int i = 0; i < totalNP; i++) {
00527         pp = i * 6;
00528         duData[pp + 0] = dyData[pp + 5] - dzData[pp + 4];
00529         duData[pp + 1] = dzData[pp + 3] - dxData[pp + 5];
00530         duData[pp + 2] = dxData[pp + 4] - dyData[pp + 3];
00531         duData[pp + 3] = -dyData[pp + 2] + dzData[pp + 1];
00532         duData[pp + 4] = -dzData[pp + 0] + dxData[pp + 2];
00533         duData[pp + 5] = -dxData[pp + 1] + dyData[pp + 0];
00534     }
00535 }

```

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

```

00080
00081
00082 // pointer to spatial derivative data sufficient, temporal derivative data
00083 // provided with udot
00084 sunrealtype *dxData = data->buffData[1 - 1];
00085
00086 // same sequence as in the linear case
00087 data->exchangeGhostCells(1);
00088 data->rotateIntoEigen(1);
00089 data->derive(1);
00090 data->derotate(1, dxData);
00091
00092 /*
00093 F and G are nonzero in the nonlinear case,
00094 polarization and magnetization contributions in Jacobi matrix style
00095 with derivatives of polarization and magnetization
00096 w.r.t. E- and B-field
00097 */
00098 sunrealtype f = NAN, g = NAN; // em field invariants F, G
00099 sunrealtype lf = NAN, lff = NAN, lfg = NAN, lg = NAN,
00100 lgg = NAN; // derivatives of Lagrangian w.r.t. field invariants
00101 array<sunrealtype, 36> JMM; // Jacobi matrix
00102 array<sunrealtype, 6> Quad; // array to hold E^2 and B^2 components
00103 array<sunrealtype, 6> h; // holding temporal derivatives of E and B components
00104 // before operating (1+Z)^-1
00105 sunrealtype pseudoDenom = NAN; // needed for inversion of 1+Z
00106 sunrealtype *udata = nullptr,
00107 *dudata = nullptr; // pointers to data and temp. derivative data
00108 udata = NV_DATA_P(u);
00109 dudata = NV_DATA_P(udot);
00110 int totalNP = data->discreteSize(); // number of points in the patch
00111 // #pragma omp parallel for private(...) reduction(...) -> unsafe due to
00112 // reductions and many variables, how to deal with / reduction?
00113 // #pragma block_loop
00114 // #pragma unroll_and_jam
00115 // #pragma distribute_point
00116 for (int pp = 0; pp < totalNP * 6;
00117      pp += 6) { // loops through all 6dim points in the patch
00118     // for(int ppB=0;ppB<totalNP*6;ppB+=6*6){
00119     // for(int pp=ppB;pp<min(totalNP*6,ppB+6*6);pp+=6){
00120     // Calculation of the Jacobi matrix
00121     // 1. Calculate F and G
00122     f = 0.5 * ((Quad[0] = udata[pp] * udata[pp]) +
00123               (Quad[1] = udata[pp + 1] * udata[pp + 1]) +
00124               (Quad[2] = udata[pp + 2] * udata[pp + 2]) -
00125               (Quad[3] = udata[pp + 3] * udata[pp + 3]) -
00126               (Quad[4] = udata[pp + 4] * udata[pp + 4]) -
00127               (Quad[5] = udata[pp + 5] * udata[pp + 5]));
00128     g = udata[pp] * udata[pp + 3] + udata[pp + 1] * udata[pp + 4] +
00129         udata[pp + 2] * udata[pp + 5];
00130     // 2. Choose process/expansion order and assign derivative values of L
00131     // w.r.t. F, G
00132     switch (*c) {
00133     case 0:
00134         lf = 0;
00135         lff = 0;
00136         lfg = 0;
00137         lg = 0;
00138         lgg = 0;
00139         break;
00140     case 2:
00141         lf = 0.000354046449700427580438254 * f * f +
00142             0.000191775160254398272737387 * g * g;
00143         lff = 0.0007080928994008551608765075 * f;
00144         lfg = 0.0003835503205087965454747749 * g;
00145         lg = 0.0003835503205087965454747749 * f * g;
00146         lgg = 0.0003835503205087965454747749 * f;
00147         break;
00148     case 1:
00149         lf = 0.000206527095658582755255648 * f;
00150         lff = 0.000206527095658582755255648;
00151         lfg = 0;
00152         lg = 0.0003614224174025198216973841 * g;
00153         lgg = 0.0003614224174025198216973841;

```

```

00154         break;
00155     case 3:
00156         lf = (0.000206527095658582755255648 + 0.000354046449700427580438254 * f) *
00157             f +
00158             0.000191775160254398272737387 * g * g;
00159         lff = 0.000206527095658582755255648 + 0.0007080928994008551608765075 * f;
00160         lfg = 0.0003835503205087965454747749 * g;
00161         lg = (0.0003614224174025198216973841 +
00162             0.0003835503205087965454747749 * f) *
00163             g;
00164         lgg = 0.0003614224174025198216973841 + 0.0003835503205087965454747749 * f;
00165         break;
00166     default:
00167         errorKill(
00168             "You need to specify a correct order in the weak-field expansion.");
00169     }
00170     // 3. Assign Jacobi components
00171     JMM[0] = lf + lff * Quad[0] +
00172         udata[3 + pp] * (2 * lfg * udata[pp] + lgg * udata[3 + pp]);
00173     JMM[6] =
00174         lff * udata[pp] * udata[1 + pp] + lfg * udata[1 + pp] * udata[3 + pp] +
00175         lfg * udata[pp] * udata[4 + pp] + lgg * udata[3 + pp] * udata[4 + pp];
00176     JMM[7] = lf + lff * Quad[1] +
00177         udata[4 + pp] * (2 * lfg * udata[1 + pp] + lgg * udata[4 + pp]);
00178     JMM[12] =
00179         lff * udata[pp] * udata[2 + pp] + lfg * udata[2 + pp] * udata[3 + pp] +
00180         lfg * udata[pp] * udata[5 + pp] + lgg * udata[3 + pp] * udata[5 + pp];
00181     JMM[13] = lff * udata[1 + pp] * udata[2 + pp] +
00182         lfg * udata[2 + pp] * udata[4 + pp] +
00183         lfg * udata[1 + pp] * udata[5 + pp] +
00184         lgg * udata[4 + pp] * udata[5 + pp];
00185     JMM[14] = lf + lff * Quad[2] +
00186         udata[5 + pp] * (2 * lfg * udata[2 + pp] + lgg * udata[5 + pp]);
00187     JMM[18] = lg + lfg * (Quad[0] - Quad[3 + 0]) +
00188         (-lff + lgg) * udata[pp] * udata[3 + pp];
00189     JMM[19] = -(udata[3 + pp] * (lff * udata[1 + pp] + lfg * udata[4 + pp])) +
00190         udata[pp] * (lfg * udata[1 + pp] + lgg * udata[4 + pp]);
00191     JMM[20] = -(udata[3 + pp] * (lff * udata[2 + pp] + lfg * udata[5 + pp])) +
00192         udata[pp] * (lfg * udata[2 + pp] + lgg * udata[5 + pp]);
00193     JMM[21] = -lf + lgg * Quad[0] +
00194         udata[3 + pp] * (-2 * lfg * udata[pp] + lff * udata[3 + pp]);
00195     JMM[24] = udata[1 + pp] * (lfg * udata[pp] + lgg * udata[3 + pp]) -
00196         (lff * udata[pp] + lfg * udata[3 + pp]) * udata[4 + pp];
00197     JMM[25] = lg + lfg * (Quad[1] - Quad[4 + 0]) +
00198         (-lff + lgg) * udata[1 + pp] * udata[4 + pp];
00199     JMM[26] = -(udata[4 + pp] * (lff * udata[2 + pp] + lfg * udata[5 + pp])) +
00200         udata[1 + pp] * (lfg * udata[2 + pp] + lgg * udata[5 + pp]);
00201     JMM[27] = lgg * udata[pp] * udata[1 + pp] +
00202         lff * udata[3 + pp] * udata[4 + pp] -
00203         lfg * (udata[1 + pp] * udata[3 + pp] + udata[pp] * udata[4 + pp]);
00204     JMM[28] = -lf + lgg * Quad[1] +
00205         udata[4 + pp] * (-2 * lfg * udata[1 + pp] + lff * udata[4 + pp]);
00206     JMM[30] = udata[2 + pp] * (lfg * udata[pp] + lgg * udata[3 + pp]) -
00207         (lff * udata[pp] + lfg * udata[3 + pp]) * udata[5 + pp];
00208     JMM[31] = udata[2 + pp] * (lfg * udata[1 + pp] + lgg * udata[4 + pp]) -
00209         (lff * udata[1 + pp] + lfg * udata[4 + pp]) * udata[5 + pp];
00210     JMM[32] = lg + lfg * (Quad[2] - Quad[5 + 0]) +
00211         (-lff + lgg) * udata[2 + pp] * udata[5 + pp];
00212     JMM[33] = lgg * udata[pp] * udata[2 + pp] +
00213         lff * udata[3 + pp] * udata[5 + pp] -
00214         lfg * (udata[2 + pp] * udata[3 + pp] + udata[pp] * udata[5 + pp]);
00215     JMM[34] =
00216         lgg * udata[1 + pp] * udata[2 + pp] +
00217         lff * udata[4 + pp] * udata[5 + pp] -
00218         lfg * (udata[2 + pp] * udata[4 + pp] + udata[1 + pp] * udata[5 + pp]);
00219     JMM[35] = -lf + lgg * Quad[2] +
00220         udata[5 + pp] * (-2 * lfg * udata[2 + pp] + lff * udata[5 + pp]);
00221     // #pragma unroll_and_jam
00222     // #pragma distribute_point
00223     for (int i = 0; i < 6; i++) {
00224         for (int j = i + 1; j < 6; j++) {
00225             JMM[i * 6 + j] = JMM[j * 6 + i];
00226         }
00227     }
00228     // 4. Final values for temporal derivatives of field values
00229     h[0] = 0;
00230     h[1] = dxData[pp] * JMM[30] + dxData[1 + pp] * JMM[31] +
00231         dxData[2 + pp] * JMM[32] + dxData[3 + pp] * JMM[33] +
00232         dxData[4 + pp] * JMM[34] + dxData[5 + pp] * (-1 + JMM[35]);
00233     h[2] = -(dxData[pp] * JMM[24]) - dxData[1 + pp] * JMM[25] -
00234         dxData[2 + pp] * JMM[26] - dxData[3 + pp] * JMM[27] +
00235         dxData[4 + pp] * (1 - JMM[28]) - dxData[5 + pp] * JMM[29];
00236     h[3] = 0;
00237     h[4] = dxData[2 + pp];
00238     h[5] = -dxData[1 + pp];
00239     h[0] -= h[3] * JMM[3] + h[4] * JMM[4] + h[5] * JMM[5];
00240     h[1] -= h[3] * JMM[9] + h[4] * JMM[10] + h[5] * JMM[11];

```

```

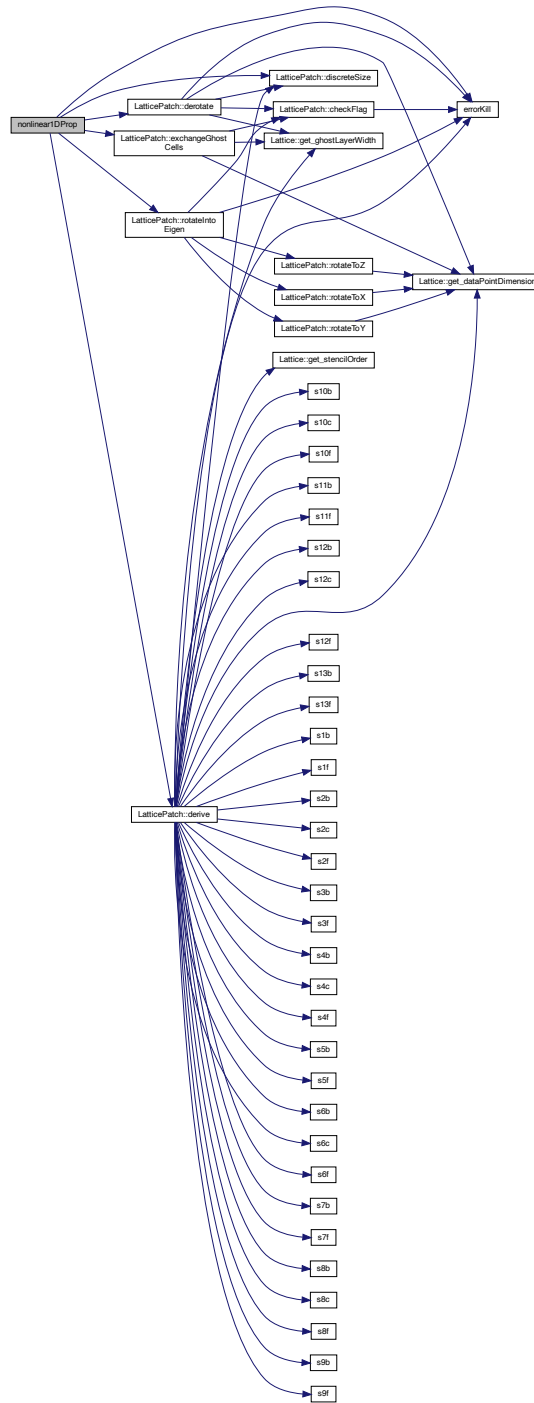
00241     h[2] -= h[3] * JMM[15] + h[4] * JMM[16] + h[5] * JMM[17];
00242     // (1+2)^-1 applies only to E components
00243     dudata[pp + 0] =
00244         h[2] * (-(JMM[2] * (1 + JMM[7])) + JMM[1] * JMM[8]) +
00245         h[1] * (JMM[2] * JMM[13] - JMM[1] * (1 + JMM[14])) +
00246         h[0] * (1 - JMM[8] * JMM[13] + JMM[14] + JMM[7] * (1 + JMM[14]));
00247     dudata[pp + 1] =
00248         h[2] * (JMM[2] * JMM[6] - (1 + JMM[0]) * JMM[8]) +
00249         h[1] * (1 - JMM[2] * JMM[12] + JMM[14] + JMM[0] * (1 + JMM[14])) +
00250         h[0] * (JMM[8] * JMM[12] - JMM[6] * (1 + JMM[14]));
00251     dudata[pp + 2] =
00252         h[2] * (1 - JMM[1] * JMM[6] + JMM[7] + JMM[0] * (1 + JMM[7])) +
00253         h[1] * (JMM[1] * JMM[12] - (1 + JMM[0]) * JMM[13]) +
00254         h[0] * (-(1 + JMM[7]) * JMM[12]) + JMM[6] * JMM[13]);
00255     pseudoDenom =
00256         -(1 + JMM[7]) * (-1 + JMM[2] * JMM[12]) +
00257         (JMM[2] * JMM[6] - JMM[8]) * JMM[13] + JMM[14] + JMM[7] * JMM[14] +
00258         JMM[0] * (1 + JMM[7] - JMM[8] * JMM[13] + (1 + JMM[7]) * JMM[14]) -
00259         JMM[1] * (-(JMM[8] * JMM[12]) + JMM[6] * (1 + JMM[14]));
00260     dudata[pp + 0] /= pseudoDenom;
00261     dudata[pp + 1] /= pseudoDenom;
00262     dudata[pp + 2] /= pseudoDenom;
00263     dudata[pp + 3] = h[3];
00264     dudata[pp + 4] = h[4];
00265     dudata[pp + 5] = h[5];
00266 }
00267 return;
00268 }

```

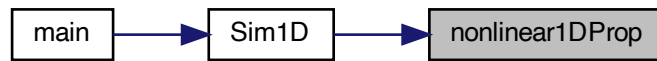
References [LatticePatch::buffData](#), [LatticePatch::derive\(\)](#), [LatticePatch::derotate\(\)](#), [LatticePatch::discreteSize\(\)](#), [errorKill\(\)](#), [LatticePatch::exchangeGhostCells\(\)](#), and [LatticePatch::rotateIntoEigen\(\)](#).

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

```

00303
00304
00305     sunrealtype *dxData = data->buffData[1 - 1];
00306     sunrealtype *dyData = data->buffData[2 - 1];
00307
00308     data->exchangeGhostCells(1);
00309     data->rotateIntoEigen(1);
00310     data->derive(1);
00311     data->derotate(1, dxData);
00312     data->exchangeGhostCells(2);
00313     data->rotateIntoEigen(2);
00314     data->derive(2);
00315     data->derotate(2, dyData);
00316
00317     sunrealtype f = NAN, g = NAN;
00318     sunrealtype lff = NAN, lff = NAN, lfg = NAN, lg = NAN, lgg = NAN;
00319     array<sunrealtype, 36> JMM;
00320     array<sunrealtype, 6> Quad;
00321     array<sunrealtype, 6> h;
00322     sunrealtype pseudoDenom = NAN;
00323     sunrealtype *udata = nullptr, *dudata = nullptr;
00324     udata = NV_DATA_P(u);
00325     dudata = NV_DATA_P(udot);
00326     int totalNP = data->discreteSize();
00327     //#pragma distribute_point
00328     //#pragma unroll_and_jam
00329     for (int pp = 0; pp < totalNP * 6; pp += 6) {
00330         // 1
00331         f = 0.5 * ((Quad[0] = udata[pp] * udata[pp]) +
00332                   (Quad[1] = udata[pp + 1] * udata[pp + 1]) +
00333                   (Quad[2] = udata[pp + 2] * udata[pp + 2]) -
00334                   (Quad[3] = udata[pp + 3] * udata[pp + 3]) -
00335                   (Quad[4] = udata[pp + 4] * udata[pp + 4]) -
00336                   (Quad[5] = udata[pp + 5] * udata[pp + 5]));
00337         g = udata[pp] * udata[pp + 3] + udata[pp + 1] * udata[pp + 4] +
00338             udata[pp + 2] * udata[pp + 5];
00339         // 2
00340         switch (*c) {
00341         case 0:
00342             lff = 0;
00343             lff = 0;
00344             lfg = 0;
00345             lg = 0;
  
```



```

00346     lgg = 0;
00347     break;
00348 case 2:
00349     lf = 0.000354046449700427580438254 * f * f +
00350         0.000191775160254398272737387 * g * g;
00351     lff = 0.0007080928994008551608765075 * f;
00352     lfg = 0.0003835503205087965454747749 * g;
00353     lg = 0.0003835503205087965454747749 * f * g;
00354     lgg = 0.0003835503205087965454747749 * f;
00355     break;
00356 case 1:
00357     lf = 0.000206527095658582755255648 * f;
00358     lff = 0.000206527095658582755255648;
00359     lfg = 0;
00360     lg = 0.0003614224174025198216973841 * g;
00361     lgg = 0.0003614224174025198216973841;
00362     break;
00363 case 3:
00364     lf = (0.000206527095658582755255648 + 0.000354046449700427580438254 * f) *
00365         f +
00366         0.000191775160254398272737387 * g * g;
00367     lff = 0.000206527095658582755255648 + 0.000708092899400855160876508 * f;
00368     lfg = 0.0003835503205087965454747749 * g;
00369     lg = (0.000361422417402519821697384 + 0.000383550320508796545474775 * f) *
00370         g;
00371     lgg = 0.000361422417402519821697384 + 0.000383550320508796545474775 * f;
00372     break;
00373 default:
00374     errorKill(
00375         "You need to specify a correct order in the weak-field expansion.");
00376 }
00377 // 3
00378 JMM[0] = lf + lff * Quad[0] +
00379     udata[3 + pp] * (2 * lfg * udata[pp] + lgg * udata[3 + pp]);
00380 JMM[6] =
00381     lff * udata[pp] * udata[1 + pp] + lfg * udata[1 + pp] * udata[3 + pp] +
00382     lfg * udata[pp] * udata[4 + pp] + lgg * udata[3 + pp] * udata[4 + pp];
00383 JMM[7] = lf + lff * Quad[1] +
00384     udata[4 + pp] * (2 * lfg * udata[1 + pp] + lgg * udata[4 + pp]);
00385 JMM[12] =
00386     lff * udata[pp] * udata[2 + pp] + lfg * udata[2 + pp] * udata[3 + pp] +
00387     lfg * udata[pp] * udata[5 + pp] + lgg * udata[3 + pp] * udata[5 + pp];
00388 JMM[13] = lff * udata[1 + pp] * udata[2 + pp] +
00389     lfg * udata[2 + pp] * udata[4 + pp] +
00390     lfg * udata[1 + pp] * udata[5 + pp] +
00391     lgg * udata[4 + pp] * udata[5 + pp];
00392 JMM[14] = lf + lff * Quad[2] +
00393     udata[5 + pp] * (2 * lfg * udata[2 + pp] + lgg * udata[5 + pp]);
00394 JMM[18] = lg + lfg * (Quad[0] - Quad[3 + 0]) +
00395     (-lff + lgg) * udata[pp] * udata[3 + pp];
00396 JMM[19] = -(udata[3 + pp] * (lff * udata[1 + pp] + lfg * udata[4 + pp])) +
00397     udata[pp] * (lfg * udata[1 + pp] + lgg * udata[4 + pp]);
00398 JMM[20] = -(udata[3 + pp] * (lff * udata[2 + pp] + lfg * udata[5 + pp])) +
00399     udata[pp] * (lfg * udata[2 + pp] + lgg * udata[5 + pp]);
00400 JMM[21] = -lf + lgg * Quad[0] +
00401     udata[3 + pp] * (-2 * lfg * udata[pp] + lff * udata[3 + pp]);
00402 JMM[24] = udata[1 + pp] * (lfg * udata[pp] + lgg * udata[3 + pp]) -
00403     (lff * udata[pp] + lfg * udata[3 + pp]) * udata[4 + pp];
00404 JMM[25] = lg + lfg * (Quad[1] - Quad[4 + 0]) +
00405     (-lff + lgg) * udata[1 + pp] * udata[4 + pp];
00406 JMM[26] = -(udata[4 + pp] * (lff * udata[2 + pp] + lfg * udata[5 + pp])) +
00407     udata[1 + pp] * (lfg * udata[2 + pp] + lgg * udata[5 + pp]);
00408 JMM[27] = lgg * udata[pp] * udata[1 + pp] +
00409     lff * udata[3 + pp] * udata[4 + pp] -
00410     lfg * (udata[1 + pp] * udata[3 + pp] + udata[pp] * udata[4 + pp]);
00411 JMM[28] = -lf + lgg * Quad[1] +
00412     udata[4 + pp] * (-2 * lfg * udata[1 + pp] + lff * udata[4 + pp]);
00413 JMM[30] = udata[2 + pp] * (lfg * udata[pp] + lgg * udata[3 + pp]) -
00414     (lff * udata[pp] + lfg * udata[3 + pp]) * udata[5 + pp];
00415 JMM[31] = udata[2 + pp] * (lfg * udata[1 + pp] + lgg * udata[4 + pp]) -
00416     (lff * udata[1 + pp] + lfg * udata[4 + pp]) * udata[5 + pp];
00417 JMM[32] = lg + lfg * (Quad[2] - Quad[5 + 0]) +
00418     (-lff + lgg) * udata[2 + pp] * udata[5 + pp];
00419 JMM[33] = lgg * udata[pp] * udata[2 + pp] +
00420     lff * udata[3 + pp] * udata[5 + pp] -
00421     lfg * (udata[2 + pp] * udata[3 + pp] + udata[pp] * udata[5 + pp]);
00422 JMM[34] =
00423     lgg * udata[1 + pp] * udata[2 + pp] +
00424     lff * udata[4 + pp] * udata[5 + pp] -
00425     lfg * (udata[2 + pp] * udata[4 + pp] + udata[1 + pp] * udata[5 + pp]);
00426 JMM[35] = -lf + lgg * Quad[2] +
00427     udata[5 + pp] * (-2 * lfg * udata[2 + pp] + lff * udata[5 + pp]);
00428 // 4
00429 // #pragma distribute_point
00430 // #pragma unroll_and_jam
00431 for (int i = 0; i < 6; i++) {
00432     for (int j = i + 1; j < 6; j++) {

```

```

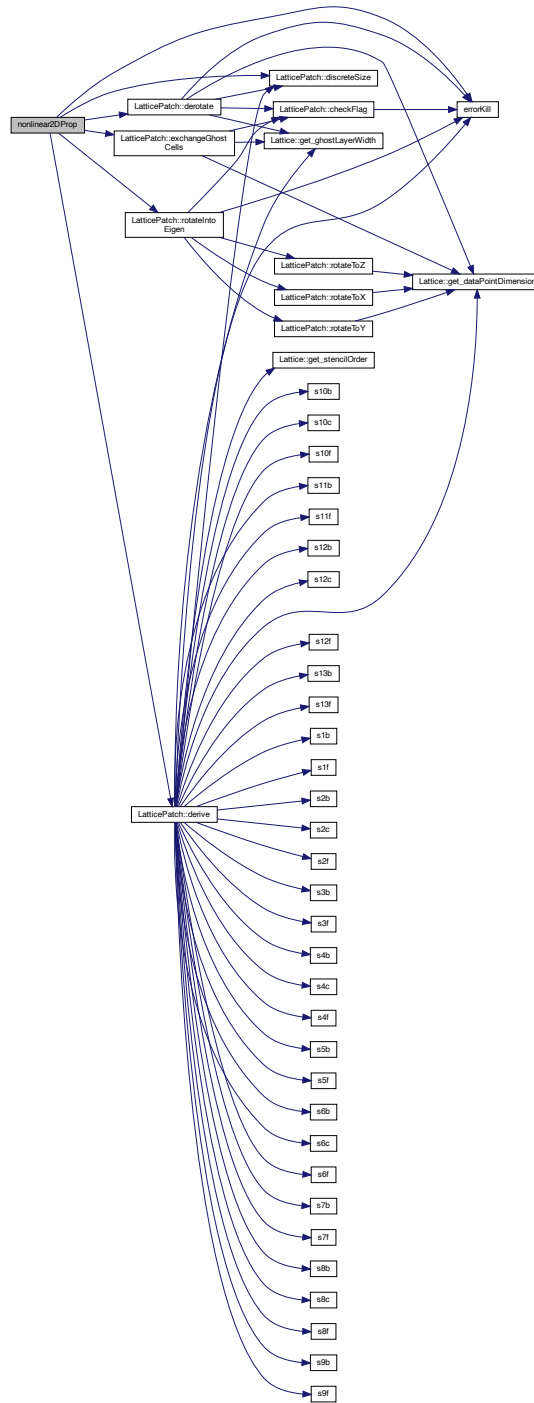
00433     JMM[i * 6 + j] = JMM[j * 6 + i];
00434 }
00435 }
00436 h[0] = 0;
00437 h[1] = dxData[pp] * JMM[30] + dxData[1 + pp] * JMM[31] +
00438     dxData[2 + pp] * JMM[32] + dxData[3 + pp] * JMM[33] +
00439     dxData[4 + pp] * JMM[34] + dxData[5 + pp] * (-1 + JMM[35]);
00440 h[2] = -(dxData[pp] * JMM[24]) - dxData[1 + pp] * JMM[25] -
00441     dxData[2 + pp] * JMM[26] - dxData[3 + pp] * JMM[27] +
00442     dxData[4 + pp] * (1 - JMM[28]) - dxData[5 + pp] * JMM[29];
00443 h[3] = 0;
00444 h[4] = dxData[2 + pp];
00445 h[5] = -dxData[1 + pp];
00446 h[0] += -(dyData[pp] * JMM[30]) - dyData[1 + pp] * JMM[31] -
00447     dyData[2 + pp] * JMM[32] - dyData[3 + pp] * JMM[33] -
00448     dyData[4 + pp] * JMM[34] + dyData[5 + pp] * (1 - JMM[35]);
00449 h[1] += 0;
00450 h[2] += dyData[pp] * JMM[18] + dyData[1 + pp] * JMM[19] +
00451     dyData[2 + pp] * JMM[20] + dyData[3 + pp] * (-1 + JMM[21]) +
00452     dyData[4 + pp] * JMM[22] + dyData[5 + pp] * JMM[23];
00453 h[3] += -dyData[2 + pp];
00454 h[4] += 0;
00455 h[5] += dyData[pp];
00456 h[0] -= h[3] * JMM[3] + h[4] * JMM[4] + h[5] * JMM[5];
00457 h[1] -= h[3] * JMM[9] + h[4] * JMM[10] + h[5] * JMM[11];
00458 h[2] -= h[3] * JMM[15] + h[4] * JMM[16] + h[5] * JMM[17];
00459 dudata[pp + 0] =
00460     h[2] * (-(JMM[2] * (1 + JMM[7])) + JMM[1] * JMM[8]) +
00461     h[1] * (JMM[2] * JMM[13] - JMM[1] * (1 + JMM[14])) +
00462     h[0] * (1 - JMM[8] * JMM[13] + JMM[14] + JMM[7] * (1 + JMM[14]));
00463 dudata[pp + 1] =
00464     h[2] * (JMM[2] * JMM[6] - (1 + JMM[0]) * JMM[8]) +
00465     h[1] * (1 - JMM[2] * JMM[12] + JMM[14] + JMM[0] * (1 + JMM[14])) +
00466     h[0] * (JMM[8] * JMM[12] - JMM[6] * (1 + JMM[14]));
00467 dudata[pp + 2] =
00468     h[2] * (1 - JMM[1] * JMM[6] + JMM[7] + JMM[0] * (1 + JMM[7])) +
00469     h[1] * (JMM[1] * JMM[12] - (1 + JMM[0]) * JMM[13]) +
00470     h[0] * (-(1 + JMM[7]) * JMM[12]) + JMM[6] * JMM[13];
00471 pseudoDenom =
00472     -(1 + JMM[7]) * (-1 + JMM[2] * JMM[12]) +
00473     (JMM[2] * JMM[6] - JMM[8]) * JMM[13] + JMM[14] + JMM[7] * JMM[14] +
00474     JMM[0] * (1 + JMM[7] - JMM[8] * JMM[13] + (1 + JMM[7]) * JMM[14]) -
00475     JMM[1] * (-(JMM[8] * JMM[12]) + JMM[6] * (1 + JMM[14]));
00476 dudata[pp + 0] /= pseudoDenom;
00477 dudata[pp + 1] /= pseudoDenom;
00478 dudata[pp + 2] /= pseudoDenom;
00479 dudata[pp + 3] = h[3];
00480 dudata[pp + 4] = h[4];
00481 dudata[pp + 5] = h[5];
00482 }
00483 return;
00484 }

```

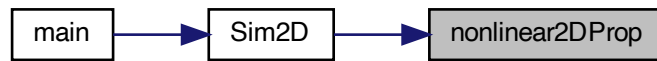
References [LatticePatch::buffData](#), [LatticePatch::derive\(\)](#), [LatticePatch::derotate\(\)](#), [LatticePatch::discreteSize\(\)](#), [errorKill\(\)](#), [LatticePatch::exchangeGhostCells\(\)](#), and [LatticePatch::rotateIntoEigen\(\)](#).

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

```

00538                                     {
00539
00540     sunrealtype *dxData = data->buffData[1 - 1];
00541     sunrealtype *dyData = data->buffData[2 - 1];
00542     sunrealtype *dzData = data->buffData[3 - 1];
00543
00544     /* Under the hood call of point-to-point or collective communication */
00545     // Point-to-Point:
00546
00547     data->exchangeGhostCells(1);
00548     data->rotateIntoEigen(1);
00549     data->derive(1);
00550     data->derotate(1, dxData);
00551     data->exchangeGhostCells(2);
00552     data->rotateIntoEigen(2);
00553     data->derive(2);
00554     data->derotate(2, dyData);
00555     data->exchangeGhostCells(3);
00556     data->rotateIntoEigen(3);
00557     data->derive(3);
00558     data->derotate(3, dzData);
00559
00560     // Collective:
00561     /*
00562     data->exchangeGhostCells3D();
00563     data->rotateIntoEigen3D();
00564     data->derive(1);
00565     data->derotate(1, dxData);
00566     data->derive(2);
00567     data->derotate(2, dyData);
00568     data->derive(3);
00569     data->derotate(3, dzData);
00570     */
00571
00572     sunrealtype f = NAN, g = NAN;
00573     sunrealtype lff = NAN, lffg = NAN, lfg = NAN, lg = NAN, lgg = NAN;
00574     array<sunrealtype, 36> JMM;
00575     array<sunrealtype, 6> Quad;
00576     array<sunrealtype, 6> h;
00577     sunrealtype pseudoDenom = NAN;
00578     sunrealtype *udata = nullptr, *dudata = nullptr;
00579     udata = NV_DATA_P(u);
00580     dudata = NV_DATA_P(udot);
  
```

```

00581 int totalNP = data->discreteSize();
00582 ///pragma distribute_point
00583 ///pragma unroll_and_jam
00584 for (int pp = 0; pp < totalNP * 6; pp += 6) {
00585     // 1
00586     f = 0.5 * ((Quad[0] = udata[pp] * udata[pp]) +
00587               (Quad[1] = udata[pp + 1] * udata[pp + 1]) +
00588               (Quad[2] = udata[pp + 2] * udata[pp + 2]) -
00589               (Quad[3] = udata[pp + 3] * udata[pp + 3]) -
00590               (Quad[4] = udata[pp + 4] * udata[pp + 4]) -
00591               (Quad[5] = udata[pp + 5] * udata[pp + 5]));
00592     g = udata[pp] * udata[pp + 3] + udata[pp + 1] * udata[pp + 4] +
00593         udata[pp + 2] * udata[pp + 5];
00594     // 2
00595     switch (*c) {
00596     case 0:
00597         lff = 0;
00598         lff = 0;
00599         lfg = 0;
00600         lg = 0;
00601         lgg = 0;
00602         break;
00603     case 2:
00604         lf = 0.000354046449700427580438254 * f * f +
00605             0.000191775160254398272737387 * g * g;
00606         lff = 0.0007080928994008551608765075 * f;
00607         lfg = 0.0003835503205087965454747749 * g;
00608         lg = 0.0003835503205087965454747749 * f * g;
00609         lgg = 0.0003835503205087965454747749 * f;
00610         break;
00611     case 1:
00612         lf = 0.000206527095658582755255648 * f;
00613         lff = 0.000206527095658582755255648;
00614         lfg = 0;
00615         lg = 0.0003614224174025198216973841 * g;
00616         lgg = 0.0003614224174025198216973841;
00617         break;
00618     case 3:
00619         lf = (0.000206527095658582755255648 + 0.000354046449700427580438254 * f) *
00620             f +
00621             0.000191775160254398272737387 * g * g;
00622         lff = 0.000206527095658582755255648 + 0.000708092899400855160876508 * f;
00623         lfg = 0.0003835503205087965454747749 * g;
00624         lg = (0.000361422417402519821697384 + 0.000383550320508796545474775 * f) *
00625             g;
00626         lgg = 0.000361422417402519821697384 + 0.000383550320508796545474775 * f;
00627         break;
00628     default:
00629         errorKill(
00630             "You need to specify a correct order in the weak-field expansion.");
00631     }
00632     // 3
00633     JMM[0] = lf + lff * Quad[0] +
00634         udata[3 + pp] * (2 * lfg * udata[pp] + lgg * udata[3 + pp]);
00635     JMM[6] =
00636         lff * udata[pp] * udata[1 + pp] + lfg * udata[1 + pp] * udata[3 + pp] +
00637         lfg * udata[pp] * udata[4 + pp] + lgg * udata[3 + pp] * udata[4 + pp];
00638     JMM[7] = lf + lff * Quad[1] +
00639         udata[4 + pp] * (2 * lfg * udata[1 + pp] + lgg * udata[4 + pp]);
00640     JMM[12] =
00641         lff * udata[pp] * udata[2 + pp] + lfg * udata[2 + pp] * udata[3 + pp] +
00642         lfg * udata[pp] * udata[5 + pp] + lgg * udata[3 + pp] * udata[5 + pp];
00643     JMM[13] = lff * udata[1 + pp] * udata[2 + pp] +
00644         lfg * udata[2 + pp] * udata[4 + pp] +
00645         lfg * udata[1 + pp] * udata[5 + pp] +
00646         lgg * udata[4 + pp] * udata[5 + pp];
00647     JMM[14] = lf + lff * Quad[2] +
00648         udata[5 + pp] * (2 * lfg * udata[2 + pp] + lgg * udata[5 + pp]);
00649     JMM[18] = lg + lfg * (Quad[0] - Quad[3 + 0]) +
00650         (-lff + lgg) * udata[pp] * udata[3 + pp];
00651     JMM[19] = -(udata[3 + pp] * (lff * udata[1 + pp] + lfg * udata[4 + pp])) +
00652         udata[pp] * (lfg * udata[1 + pp] + lgg * udata[4 + pp]);
00653     JMM[20] = -(udata[3 + pp] * (lff * udata[2 + pp] + lfg * udata[5 + pp])) +
00654         udata[pp] * (lfg * udata[2 + pp] + lgg * udata[5 + pp]);
00655     JMM[21] = -lf + lgg * Quad[0] +
00656         udata[3 + pp] * (-2 * lfg * udata[pp] + lff * udata[3 + pp]);
00657     JMM[24] = udata[1 + pp] * (lfg * udata[pp] + lgg * udata[3 + pp]) -
00658         (lff * udata[pp] + lfg * udata[3 + pp]) * udata[4 + pp];
00659     JMM[25] = lg + lfg * (Quad[1] - Quad[4 + 0]) +
00660         (-lff + lgg) * udata[1 + pp] * udata[4 + pp];
00661     JMM[26] = -(udata[4 + pp] * (lff * udata[2 + pp] + lfg * udata[5 + pp])) +
00662         udata[1 + pp] * (lfg * udata[2 + pp] + lgg * udata[5 + pp]);
00663     JMM[27] = lgg * udata[pp] * udata[1 + pp] +
00664         lff * udata[3 + pp] * udata[4 + pp] -
00665         lfg * (udata[1 + pp] * udata[3 + pp] + udata[pp] * udata[4 + pp]);
00666     JMM[28] = -lf + lgg * Quad[1] +
00667         udata[4 + pp] * (-2 * lfg * udata[1 + pp] + lff * udata[4 + pp]);

```

```

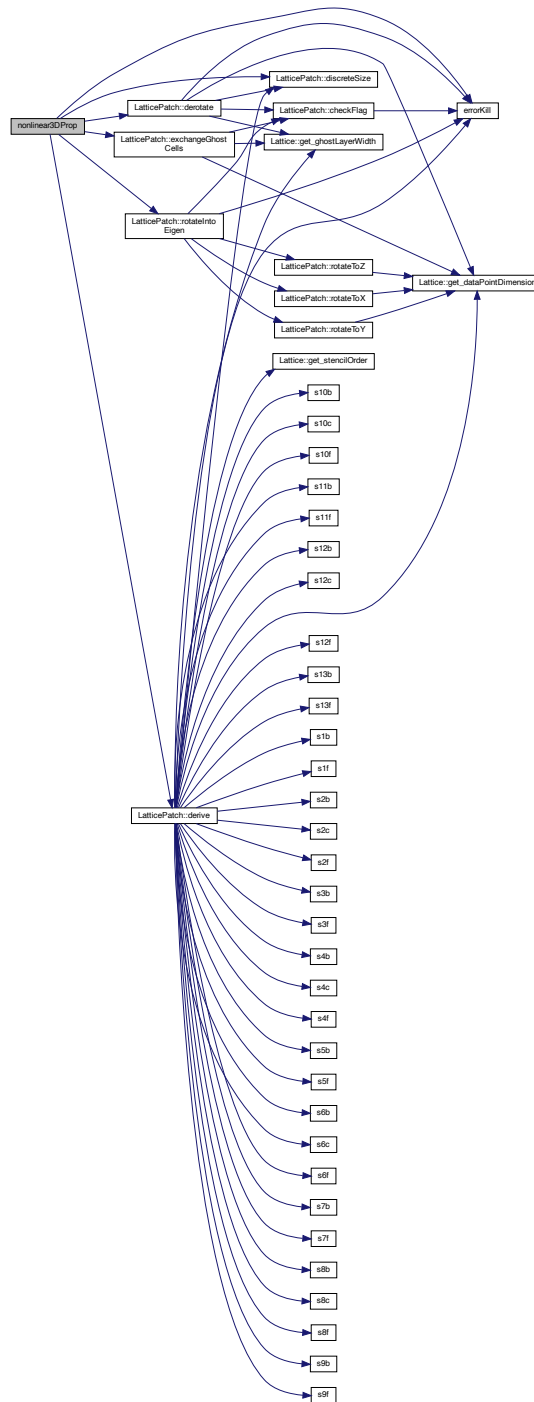
00668 JMM[30] = udata[2 + pp] * (lfg * udata[pp] + lgg * udata[3 + pp]) -
00669         (lff * udata[pp] + lfg * udata[3 + pp]) * udata[5 + pp];
00670 JMM[31] = udata[2 + pp] * (lfg * udata[1 + pp] + lgg * udata[4 + pp]) -
00671         (lff * udata[1 + pp] + lfg * udata[4 + pp]) * udata[5 + pp];
00672 JMM[32] = lg + lfg * (Quad[2] - Quad[5 + 0]) +
00673         (-lff + lgg) * udata[2 + pp] * udata[5 + pp];
00674 JMM[33] = lgg * udata[pp] * udata[2 + pp] +
00675         lff * udata[3 + pp] * udata[5 + pp] -
00676         lfg * (udata[2 + pp] * udata[3 + pp] + udata[pp] * udata[5 + pp]);
00677 JMM[34] =
00678         lgg * udata[1 + pp] * udata[2 + pp] +
00679         lff * udata[4 + pp] * udata[5 + pp] -
00680         lfg * (udata[2 + pp] * udata[4 + pp] + udata[1 + pp] * udata[5 + pp]);
00681 JMM[35] = -lf + lgg * Quad[2] +
00682         udata[5 + pp] * (-2 * lfg * udata[2 + pp] + lff * udata[5 + pp]);
00683 // 4
00684 // #pragma distribute_point
00685 // #pragma unroll_and_jam
00686 for (int i = 0; i < 6; i++) {
00687     for (int j = i + 1; j < 6; j++) {
00688         JMM[i * 6 + j] = JMM[j * 6 + i];
00689     }
00690 }
00691 h[0] = 0;
00692 h[1] = dxData[pp] * JMM[30] + dxData[1 + pp] * JMM[31] +
00693         dxData[2 + pp] * JMM[32] + dxData[3 + pp] * JMM[33] +
00694         dxData[4 + pp] * JMM[34] + dxData[5 + pp] * (-1 + JMM[35]);
00695 h[2] = -(dxData[pp] * JMM[24]) - dxData[1 + pp] * JMM[25] -
00696         dxData[2 + pp] * JMM[26] - dxData[3 + pp] * JMM[27] +
00697         dxData[4 + pp] * (1 - JMM[28]) - dxData[5 + pp] * JMM[29];
00698 h[3] = 0;
00699 h[4] = dxData[2 + pp];
00700 h[5] = -dxData[1 + pp];
00701 h[0] += -(dyData[pp] * JMM[30]) - dyData[1 + pp] * JMM[31] -
00702         dyData[2 + pp] * JMM[32] - dyData[3 + pp] * JMM[33] -
00703         dyData[4 + pp] * JMM[34] + dyData[5 + pp] * (1 - JMM[35]);
00704 h[1] += 0;
00705 h[2] += dyData[pp] * JMM[18] + dyData[1 + pp] * JMM[19] +
00706         dzData[2 + pp] * JMM[20] + dyData[3 + pp] * (-1 + JMM[21]) +
00707         dyData[4 + pp] * JMM[22] + dyData[5 + pp] * JMM[23];
00708 h[3] += -dyData[2 + pp];
00709 h[4] += 0;
00710 h[5] += dyData[pp];
00711 h[0] += dzData[pp] * JMM[24] + dzData[1 + pp] * JMM[25] +
00712         dzData[2 + pp] * JMM[26] + dzData[3 + pp] * JMM[27] +
00713         dzData[4 + pp] * (-1 + JMM[28]) + dzData[5 + pp] * JMM[29];
00714 h[1] += -(dzData[pp] * JMM[18]) - dzData[1 + pp] * JMM[19] -
00715         dzData[2 + pp] * JMM[20] + dzData[3 + pp] * (1 - JMM[21]) -
00716         dzData[4 + pp] * JMM[22] - dzData[5 + pp] * JMM[23];
00717 h[2] += 0;
00718 h[3] += dzData[1 + pp];
00719 h[4] += -dzData[pp];
00720 h[5] += 0;
00721 h[0] -= h[3] * JMM[3] + h[4] * JMM[4] + h[5] * JMM[5];
00722 h[1] -= h[3] * JMM[9] + h[4] * JMM[10] + h[5] * JMM[11];
00723 h[2] -= h[3] * JMM[15] + h[4] * JMM[16] + h[5] * JMM[17];
00724 dudata[pp + 0] =
00725         h[2] * (-(JMM[2] * (1 + JMM[7])) + JMM[1] * JMM[8]) +
00726         h[1] * (JMM[2] * JMM[13] - JMM[1] * (1 + JMM[14])) +
00727         h[0] * (1 - JMM[8] * JMM[13] + JMM[14] + JMM[7] * (1 + JMM[14]));
00728 dudata[pp + 1] =
00729         h[2] * (JMM[2] * JMM[6] - (1 + JMM[0]) * JMM[8]) +
00730         h[1] * (1 - JMM[2] * JMM[12] + JMM[14] + JMM[0] * (1 + JMM[14])) +
00731         h[0] * (JMM[8] * JMM[12] - JMM[6] * (1 + JMM[14]));
00732 dudata[pp + 2] =
00733         h[2] * (1 - JMM[1] * JMM[6] + JMM[7] + JMM[0] * (1 + JMM[7])) +
00734         h[1] * (JMM[1] * JMM[12] - (1 + JMM[0]) * JMM[13]) +
00735         h[0] * (-(1 + JMM[7]) * JMM[12]) + JMM[6] * JMM[13];
00736 pseudoDenom =
00737         -(1 + JMM[7]) * (-1 + JMM[2] * JMM[12]) +
00738         (JMM[2] * JMM[6] - JMM[8]) * JMM[13] + JMM[14] + JMM[7] * JMM[14] +
00739         JMM[0] * (1 + JMM[7] - JMM[8] * JMM[13] + (1 + JMM[7]) * JMM[14]) -
00740         JMM[1] * (-(JMM[8] * JMM[12]) + JMM[6] * (1 + JMM[14]));
00741 dudata[pp + 0] /= pseudoDenom;
00742 dudata[pp + 1] /= pseudoDenom;
00743 dudata[pp + 2] /= pseudoDenom;
00744 dudata[pp + 3] = h[3];
00745 dudata[pp + 4] = h[4];
00746 dudata[pp + 5] = h[5];
00747 }
00748 return;
00749 }

```

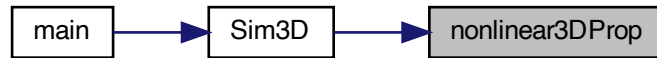
References [LatticePatch::buffData](#), [LatticePatch::derive\(\)](#), [LatticePatch::derotate\(\)](#), [LatticePatch::discreteSize\(\)](#), [errorKill\(\)](#), [LatticePatch::exchangeGhostCells\(\)](#), and [LatticePatch::rotateIntoEigen\(\)](#).

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

Here is the call graph for this function:



Here is the caller graph for this function:



6.31 TimeEvolutionFunctions.h

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

```

00001 ///////////////////////////////////////////////////////////////////
00002 /// @file TimeEvolutionFunctions.h
00003 /// @brief Functions to propagate data vectors in time
00004 /// according to Maxwell's equations, and various
00005 /// orders in the HE weak-field expansion
00006 ///////////////////////////////////////////////////////////////////
00007
00008 #pragma once
00009
00010 #include "LatticePatch.h"
00011 #include "SimulationClass.h"
00012
00013 /** @brief monostate TimeEvolution Class to propagate the field data in time in
00014  * a given order of the HE weak-field expansion */
00015 class TimeEvolution {
00016 public:
00017     /// choice which processes of the weak field expansion are included
00018     static int *c;
00019
00020     /// Pointer to functions for differentiation and time evolution
00021     static void (*TimeEvolver)(LatticePatch *, N_Vector, N_Vector, int *);
00022
00023     /// CVODE right hand side function (CVRhsFn) to provide IVP of the ODE
00024     static int f(sunrealtype t, N_Vector u, N_Vector udot, void *data_loc);
00025 };
00026
00027 /// Maxwell propagation function for 1D -- only for reference
00028 void linear1DProp(LatticePatch *data, N_Vector u, N_Vector udot, int *c);
00029 /// HE propagation function for 1D
00030 void nonlinear1DProp(LatticePatch *data, N_Vector u, N_Vector udot, int *c);
00031 /// Maxwell propagation function for 2D -- only for reference
00032 void linear2DProp(LatticePatch *data, N_Vector u, N_Vector udot, int *c);
00033 /// HE propagation function for 2D
00034 void nonlinear2DProp(LatticePatch *data, N_Vector u, N_Vector udot, int *c);
00035 /// Maxwell propagation function for 3D -- only for reference
00036 void linear3DProp(LatticePatch *data, N_Vector u, N_Vector udot, int *c);
00037 /// HE propagation function for 3D
00038 void nonlinear3DProp(LatticePatch *data, N_Vector u, N_Vector udot, int *c);
00039

```


Index

~LatticePatch
 LatticePatch, [64](#)
~Simulation
 Simulation, [124](#)

A1
 Gauss2D, [22](#)
 Gauss3D, [27](#)
A2
 Gauss2D, [22](#)
 Gauss3D, [28](#)
add
 ICSetter, [37](#)
addGauss1D
 ICSetter, [38](#)
addGauss2D
 ICSetter, [39](#)
addGauss3D
 ICSetter, [39](#)
addInitialConditions
 Simulation, [125](#)
addPeriodicICLayerInX
 Simulation, [126](#)
addPeriodicICLayerInXY
 Simulation, [126](#)
addPlaneWave1D
 ICSetter, [40](#)
addPlaneWave2D
 ICSetter, [41](#)
addPlaneWave3D
 ICSetter, [41](#)
addToSpace
 Gauss1D, [15](#)
 Gauss2D, [21](#)
 Gauss3D, [27](#)
 PlaneWave1D, [116](#)
 PlaneWave2D, [118](#)
 PlaneWave3D, [121](#)
advanceToTime
 Simulation, [127](#)
Amp
 Gauss2D, [22](#)
 Gauss3D, [28](#)
amp
 gaussian2D, [33](#)
 gaussian3D, [35](#)
axis
 Gauss2D, [22](#)
 Gauss3D, [28](#)
 gaussian2D, [33](#)

gaussian3D, [35](#)

buffData
 LatticePatch, [89](#)
bufferCreator
 LatticePatch, [64](#)
BuffersInitialized
 LatticePatch.h, [212](#)
buffX
 LatticePatch, [89](#)
buffY
 LatticePatch, [89](#)
buffZ
 LatticePatch, [89](#)
c
 TimeEvolution, [143](#)
check_retval
 LatticePatch.cpp, [194](#)
 LatticePatch.h, [212](#)
checkFlag
 LatticePatch, [66](#)
 Simulation, [128](#)
checkNoFlag
 Simulation, [130](#)
comm
 Lattice, [55](#)
cnode_mem
 Simulation, [139](#)
CnodeObjectSetUp
 SimulationClass.h, [239](#)
dataPointDimension
 Lattice, [55](#)
DerivationStencils.h
 s10b, [148](#)
 s10c, [149](#)
 s10f, [150](#)
 s11b, [151](#)
 s11f, [152](#), [153](#)
 s12b, [153](#), [154](#)
 s12c, [154](#), [155](#)
 s12f, [155](#), [156](#)
 s13b, [156](#), [157](#)
 s13f, [157](#), [158](#)
 s1b, [158](#), [159](#)
 s1f, [159](#), [160](#)
 s2b, [160](#), [161](#)
 s2c, [161](#), [162](#)
 s2f, [162](#), [163](#)

- s3b, [163](#), [164](#)
- s3f, [164](#), [165](#)
- s4b, [165](#), [166](#)
- s4c, [166](#), [167](#)
- s4f, [167](#), [168](#)
- s5b, [168](#), [169](#)
- s5f, [169](#), [170](#)
- s6b, [170](#), [171](#)
- s6c, [171](#), [172](#)
- s6f, [172](#), [173](#)
- s7b, [173](#), [174](#)
- s7f, [174](#), [175](#)
- s8b, [175](#), [176](#)
- s8c, [176](#), [177](#)
- s8f, [177](#), [178](#)
- s9b, [178](#), [179](#)
- s9f, [179](#), [180](#)
- derive
 - LatticePatch, [67](#)
- derotate
 - LatticePatch, [72](#)
- dis
 - Gauss2D, [23](#)
 - Gauss3D, [28](#)
- discreteSize
 - LatticePatch, [74](#)
- du
 - LatticePatch, [90](#)
- duData
 - LatticePatch, [90](#)
- dx
 - Lattice, [55](#)
 - LatticePatch, [90](#)
- dy
 - Lattice, [55](#)
 - LatticePatch, [90](#)
- dz
 - Lattice, [56](#)
 - LatticePatch, [91](#)
- envelopeLattice
 - LatticePatch, [91](#)
- errorKill
 - LatticePatch.cpp, [195](#)
 - LatticePatch.h, [213](#)
- eval
 - ICSetter, [42](#)
- exchangeGhostCells
 - LatticePatch, [75](#)
- exchangeGhostCells3D
 - LatticePatch, [77](#)
- f
 - TimeEvolution, [142](#)
- FLatticeDimensionSet
 - LatticePatch.h, [211](#)
- FLatticePatchSetUp
 - LatticePatch.h, [212](#)
- 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
- gCAData
 - LatticePatch, 91
- gCBData
 - LatticePatch, 91
- gCFData
 - LatticePatch, 92
- gCLData
 - LatticePatch, 92
- gCRData
 - LatticePatch, 92
- gCTData
 - LatticePatch, 92
- generateOutputFolder
 - OutputManager, 103
- generatePatchwork
 - LatticePatch, 88
 - LatticePatch.cpp, 196
- generateTranslocationLookup
 - LatticePatch, 78
- get_cart_comm
 - Simulation, 131
- 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, 50
- get_tot_ly
 - Lattice, 50
- get_tot_lz
 - Lattice, 51
- get_tot_noDP
 - Lattice, 51
- get_tot_noP
 - Lattice, 51
- get_tot_nx
 - Lattice, 52
- get_tot_ny
 - Lattice, 52
- get_tot_nz
 - Lattice, 52
- getDelta
 - LatticePatch, 80
- getSimCode
 - OutputManager, 104
- ghostCellLeft
 - LatticePatch, 93
- ghostCellLeftToSend
 - LatticePatch, 93
- ghostCellRight
 - LatticePatch, 93
- ghostCellRightToSend
 - LatticePatch, 93
- ghostCells
 - LatticePatch, 94
- ghostCellsToSend
 - LatticePatch, 94
- GhostLayersInitialized
 - LatticePatch.h, 212
- 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, 139
- ID
 - LatticePatch, 94
- initializeBuffers
 - LatticePatch, 80
- initializeCommunicator
 - Lattice, 52
- initializeCVODEobject
 - Simulation, 131
- initializeGhostLayer
 - LatticePatch, 81
- initializePatchwork
 - Simulation, 133
- k

- gaussian1D, 31
- planewave, 113
- kx
 - Gauss1D, 16
 - PlaneWave, 110
- ky
 - Gauss1D, 16
 - PlaneWave, 110
- kz
 - Gauss1D, 16
 - PlaneWave, 111
- 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, 50
 - get_tot_ly, 50
 - get_tot_lz, 51
 - get_tot_noDP, 51
 - get_tot_noP, 51
 - get_tot_nx, 52
 - 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, 140
- LatticeDiscreteSetUp
 - SimulationClass.h, 239
- LatticeOptions
 - LatticePatch.h, 211
- LatticePatch, 60
 - ~LatticePatch, 64
 - buffData, 89
 - bufferCreator, 64
 - buffX, 89
 - buffY, 89
 - buffZ, 89
 - checkFlag, 66
 - derive, 67
 - derotate, 72
 - discreteSize, 74
 - du, 90
 - duData, 90
 - dx, 90
 - dy, 90
 - dz, 91
 - envelopeLattice, 91
 - exchangeGhostCells, 75
 - exchangeGhostCells3D, 77
 - gCAData, 91
 - gCBData, 91
 - gCFData, 92
 - gCLData, 92
 - gCRData, 92
 - gCTData, 92
 - generatePatchwork, 88
 - generateTranslocationLookup, 78
 - getDelta, 80
 - ghostCellLeft, 93
 - ghostCellLeftToSend, 93
 - ghostCellRight, 93
 - ghostCellRightToSend, 93
 - ghostCells, 94
 - ghostCellsToSend, 94
 - ID, 94
 - initializeBuffers, 80
 - initializeGhostLayer, 81
 - LatticePatch, 63
 - lgcTox, 94
 - lgcToy, 95
 - lgcToz, 95
 - Llx, 95
 - Lly, 95
 - Llz, 96
 - lx, 96
 - ly, 96
 - lz, 96
 - nx, 97
 - ny, 97
 - nz, 97
 - origin, 81
 - rgcTox, 97
 - rgcToy, 98
 - rgcToz, 98
 - rotateIntoEigen, 82
 - rotateIntoEigen3D, 84
 - rotateToX, 85

- rotateToY, [85](#)
- rotateToZ, [86](#)
- statusFlags, [98](#)
- u, [98](#)
- uAux, [99](#)
- uAuxData, [99](#)
- uData, [99](#)
- uTox, [99](#)
- uToy, [100](#)
- uToz, [100](#)
- x0, [100](#)
- xTou, [100](#)
- y0, [101](#)
- yTou, [101](#)
- z0, [101](#)
- zTou, [101](#)
- latticePatch
 - Simulation, [140](#)
- LatticePatch.cpp
 - check_retval, [194](#)
 - errorKill, [195](#)
 - generatePatchwork, [196](#)
- LatticePatch.h
 - BuffersInitialized, [212](#)
 - check_retval, [212](#)
 - errorKill, [213](#)
 - FLatticeDimensionSet, [211](#)
 - FLatticePatchSetUp, [212](#)
 - GhostLayersInitialized, [212](#)
 - LatticeOptions, [211](#)
 - LatticePatchOptions, [211](#)
 - TranslocationLookupSetUp, [212](#)
- LatticePatchOptions
 - LatticePatch.h, [211](#)
- LatticePatchworkSetUp
 - SimulationClass.h, [239](#)
- LatticePhysicalSetUp
 - SimulationClass.h, [239](#)
- lgcTox
 - LatticePatch, [94](#)
- lgcToy
 - LatticePatch, [95](#)
- lgcToz
 - LatticePatch, [95](#)
- linear1DProp
 - TimeEvolutionFunctions.cpp, [267](#)
 - TimeEvolutionFunctions.h, [296](#)
- linear2DProp
 - TimeEvolutionFunctions.cpp, [269](#)
 - TimeEvolutionFunctions.h, [297](#)
- linear3DProp
 - TimeEvolutionFunctions.cpp, [271](#)
 - TimeEvolutionFunctions.h, [299](#)
- Llx
 - LatticePatch, [95](#)
- Lly
 - LatticePatch, [95](#)
- Llz
 - LatticePatch, [96](#)
- lx
 - LatticePatch, [96](#)
- ly
 - LatticePatch, [96](#)
- lz
 - LatticePatch, [96](#)
- main
 - main.cpp, [218](#)
- main.cpp
 - main, [218](#)
- my_prc
 - Lattice, [56](#)
- n_prc
 - Lattice, [56](#)
- nonlinear1DProp
 - TimeEvolutionFunctions.cpp, [273](#)
 - TimeEvolutionFunctions.h, [301](#)
- nonlinear2DProp
 - TimeEvolutionFunctions.cpp, [278](#)
 - TimeEvolutionFunctions.h, [306](#)
- nonlinear3DProp
 - TimeEvolutionFunctions.cpp, [282](#)
 - TimeEvolutionFunctions.h, [310](#)
- nx
 - LatticePatch, [97](#)
- ny
 - LatticePatch, [97](#)
- nz
 - LatticePatch, [97](#)
- origin
 - LatticePatch, [81](#)
- outAllFieldData
 - Simulation, [134](#)
- OutputManager, [102](#)
 - generateOutputFolder, [103](#)
 - getSimCode, [104](#)
 - OutputManager, [103](#)
 - outputStyle, [108](#)
 - outUState, [105](#)
 - Path, [108](#)
 - set_outputStyle, [107](#)
 - simCode, [109](#)
 - SimCodeGenerator, [107](#)
- outputManager
 - Simulation, [140](#)
- outputStyle
 - OutputManager, [108](#)
- outUState
 - OutputManager, [105](#)
- p
 - gaussian1D, [31](#)
 - planewave, [113](#)
- Path
 - OutputManager, [108](#)

- Ph0
 - Gauss2D, [23](#)
 - Gauss3D, [29](#)
- ph0
 - gaussian2D, [33](#)
 - gaussian3D, [35](#)
- PhA
 - Gauss2D, [23](#)
 - Gauss3D, [29](#)
- phA
 - gaussian2D, [33](#)
 - gaussian3D, [35](#)
- phi
 - gaussian1D, [31](#)
 - planewave, [113](#)
- phig
 - Gauss1D, [16](#)
 - gaussian1D, [31](#)
- phip
 - Gauss2D, [24](#)
 - Gauss3D, [29](#)
 - gaussian2D, [33](#)
 - gaussian3D, [35](#)
- phix
 - Gauss1D, [16](#)
 - PlaneWave, [111](#)
- phiy
 - Gauss1D, [17](#)
 - PlaneWave, [111](#)
- phiz
 - Gauss1D, [17](#)
 - PlaneWave, [111](#)
- PlaneWave, [109](#)
 - kx, [110](#)
 - ky, [110](#)
 - kz, [111](#)
 - phix, [111](#)
 - phiy, [111](#)
 - phiz, [111](#)
 - px, [112](#)
 - py, [112](#)
 - pz, [112](#)
- planewave, [113](#)
 - k, [113](#)
 - p, [113](#)
 - phi, [113](#)
- PlaneWave1D, [114](#)
 - addToSpace, [116](#)
 - PlaneWave1D, [115](#)
- PlaneWave2D, [116](#)
 - addToSpace, [118](#)
 - PlaneWave2D, [117](#)
- PlaneWave3D, [119](#)
 - addToSpace, [121](#)
 - PlaneWave3D, [120](#)
- planeWaves1D
 - ICSetter, [44](#)
- planeWaves2D
 - ICSetter, [44](#)
- planeWaves3D
 - ICSetter, [44](#)
- profobj
 - Lattice, [57](#)
- px
 - Gauss1D, [17](#)
 - PlaneWave, [112](#)
- py
 - Gauss1D, [17](#)
 - PlaneWave, [112](#)
- pz
 - Gauss1D, [18](#)
 - PlaneWave, [112](#)
- README.md, [145](#)
- rgcTox
 - LatticePatch, [97](#)
- rgcToy
 - LatticePatch, [98](#)
- rgcToz
 - LatticePatch, [98](#)
- rotateIntoEigen
 - LatticePatch, [82](#)
- rotateIntoEigen3D
 - LatticePatch, [84](#)
- rotateToX
 - LatticePatch, [85](#)
- rotateToY
 - LatticePatch, [85](#)
- rotateToZ
 - LatticePatch, [86](#)
- s10b
 - DerivationStencils.h, [148](#)
- s10c
 - DerivationStencils.h, [149](#)
- s10f
 - DerivationStencils.h, [150](#)
- s11b
 - DerivationStencils.h, [151](#)
- s11f
 - DerivationStencils.h, [152](#), [153](#)
- s12b
 - DerivationStencils.h, [153](#), [154](#)
- s12c
 - DerivationStencils.h, [154](#), [155](#)
- s12f
 - DerivationStencils.h, [155](#), [156](#)
- s13b
 - DerivationStencils.h, [156](#), [157](#)
- s13f
 - DerivationStencils.h, [157](#), [158](#)
- s1b
 - DerivationStencils.h, [158](#), [159](#)
- s1f
 - DerivationStencils.h, [159](#), [160](#)
- s2b
 - DerivationStencils.h, [160](#), [161](#)

- s2c
 - DerivationStencils.h, [161](#), [162](#)
- s2f
 - DerivationStencils.h, [162](#), [163](#)
- s3b
 - DerivationStencils.h, [163](#), [164](#)
- s3f
 - DerivationStencils.h, [164](#), [165](#)
- s4b
 - DerivationStencils.h, [165](#), [166](#)
- s4c
 - DerivationStencils.h, [166](#), [167](#)
- s4f
 - DerivationStencils.h, [167](#), [168](#)
- s5b
 - DerivationStencils.h, [168](#), [169](#)
- s5f
 - DerivationStencils.h, [169](#), [170](#)
- s6b
 - DerivationStencils.h, [170](#), [171](#)
- s6c
 - DerivationStencils.h, [171](#), [172](#)
- s6f
 - DerivationStencils.h, [172](#), [173](#)
- s7b
 - DerivationStencils.h, [173](#), [174](#)
- s7f
 - DerivationStencils.h, [174](#), [175](#)
- s8b
 - DerivationStencils.h, [175](#), [176](#)
- s8c
 - DerivationStencils.h, [176](#), [177](#)
- s8f
 - DerivationStencils.h, [177](#), [178](#)
- s9b
 - DerivationStencils.h, [178](#), [179](#)
- s9f
 - DerivationStencils.h, [179](#), [180](#)
- set_outputStyle
 - OutputManager, [107](#)
- setDiscreteDimensions
 - Lattice, [53](#)
- setDiscreteDimensionsOfLattice
 - Simulation, [135](#)
- setInitialConditions
 - Simulation, [136](#)
- setPhysicalDimensions
 - Lattice, [54](#)
- setPhysicalDimensionsOfLattice
 - Simulation, [137](#)
- Sim1D
 - SimulationFunctions.cpp, [242](#)
 - SimulationFunctions.h, [256](#)
- Sim2D
 - SimulationFunctions.cpp, [245](#)
 - SimulationFunctions.h, [259](#)
- Sim3D
 - SimulationFunctions.cpp, [248](#)
 - SimulationFunctions.h, [262](#)
- simCode
 - OutputManager, [109](#)
- SimCodeGenerator
 - OutputManager, [107](#)
- Simulation, [122](#)
 - ~Simulation, [124](#)
 - addInitialConditions, [125](#)
 - addPeriodicCLayerInX, [126](#)
 - addPeriodicCLayerInXY, [126](#)
 - advanceToTime, [127](#)
 - checkFlag, [128](#)
 - checkNoFlag, [130](#)
 - ccode_mem, [139](#)
 - get_cart_comm, [131](#)
 - icsettings, [139](#)
 - initializeCVODEobject, [131](#)
 - initializePatchwork, [133](#)
 - lattice, [140](#)
 - latticePatch, [140](#)
 - outAllFieldData, [134](#)
 - outputManager, [140](#)
 - setDiscreteDimensionsOfLattice, [135](#)
 - setInitialConditions, [136](#)
 - setPhysicalDimensionsOfLattice, [137](#)
 - Simulation, [123](#)
 - start, [138](#)
 - statusFlags, [140](#)
 - t, [141](#)
- SimulationClass.h
 - CcodeObjectSetUp, [239](#)
 - LatticeDiscreteSetUp, [239](#)
 - LatticePatchworkSetUp, [239](#)
 - LatticePhysicalSetUp, [239](#)
 - SimulationOptions, [239](#)
 - SimulationStarted, [239](#)
- SimulationFunctions.cpp
 - Sim1D, [242](#)
 - Sim2D, [245](#)
 - Sim3D, [248](#)
 - timer, [251](#)
- SimulationFunctions.h
 - Sim1D, [256](#)
 - Sim2D, [259](#)
 - Sim3D, [262](#)
 - timer, [265](#)
- SimulationOptions
 - SimulationClass.h, [239](#)
- SimulationStarted
 - SimulationClass.h, [239](#)
- src/DerivationStencils.cpp, [145](#), [146](#)
- src/DerivationStencils.h, [146](#), [181](#)
- src/ICSetters.cpp, [184](#)
- src/ICSetters.h, [188](#), [190](#)
- src/LatticePatch.cpp, [193](#), [198](#)
- src/LatticePatch.h, [210](#), [214](#)
- src/main.cpp, [217](#), [226](#)
- src/Outputters.cpp, [230](#)

- src/Outputters.h, [232](#), [233](#)
- src/SimulationClass.cpp, [233](#), [234](#)
- src/SimulationClass.h, [238](#), [239](#)
- src/SimulationFunctions.cpp, [241](#), [252](#)
- src/SimulationFunctions.h, [255](#), [266](#)
- src/TimeEvolutionFunctions.cpp, [267](#), [286](#)
- src/TimeEvolutionFunctions.h, [295](#), [314](#)
- start
 - Simulation, [138](#)
- statusFlags
 - Lattice, [57](#)
 - LatticePatch, [98](#)
 - Simulation, [140](#)
- stencilOrder
 - Lattice, [57](#)
- sunctx
 - Lattice, [57](#)
- t
 - Simulation, [141](#)
- TimeEvolution, [141](#)
 - c, [143](#)
 - f, [142](#)
 - TimeEvolver, [143](#)
- TimeEvolutionFunctions.cpp
 - linear1DProp, [267](#)
 - linear2DProp, [269](#)
 - linear3DProp, [271](#)
 - nonlinear1DProp, [273](#)
 - nonlinear2DProp, [278](#)
 - nonlinear3DProp, [282](#)
- TimeEvolutionFunctions.h
 - linear1DProp, [296](#)
 - linear2DProp, [297](#)
 - linear3DProp, [299](#)
 - nonlinear1DProp, [301](#)
 - nonlinear2DProp, [306](#)
 - nonlinear3DProp, [310](#)
- TimeEvolver
 - TimeEvolution, [143](#)
- timer
 - SimulationFunctions.cpp, [251](#)
 - SimulationFunctions.h, [265](#)
- 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, [212](#)
- u
 - LatticePatch, [98](#)
- uAux
 - LatticePatch, [99](#)
- uAuxData
 - LatticePatch, [99](#)
- uData
 - LatticePatch, [99](#)
- uTox
 - LatticePatch, [99](#)
- uToy
 - LatticePatch, [100](#)
- uToz
 - LatticePatch, [100](#)
- w0
 - Gauss2D, [24](#)
 - Gauss3D, [30](#)
 - gaussian2D, [33](#)
 - gaussian3D, [36](#)
- x0
 - gaussian1D, [32](#)
 - gaussian2D, [34](#)
 - gaussian3D, [36](#)
 - LatticePatch, [100](#)
- x0x
 - Gauss1D, [18](#)
- x0y
 - Gauss1D, [18](#)
- x0z
 - Gauss1D, [18](#)
- xTou
 - LatticePatch, [100](#)
- y0
 - LatticePatch, [101](#)
- yTou
 - LatticePatch, [101](#)
- z0
 - LatticePatch, [101](#)
- zr
 - Gauss2D, [24](#)
 - Gauss3D, [30](#)
 - gaussian2D, [34](#)
 - gaussian3D, [36](#)
- zTou
 - LatticePatch, [101](#)