HEWES : Heisenberg-Euler Weak-Field Expansion Simulator

Generated by Doxygen 1.9.3

Chapter 1

HEWES: Heisenberg-Euler Weak-Field Expansion Simulator

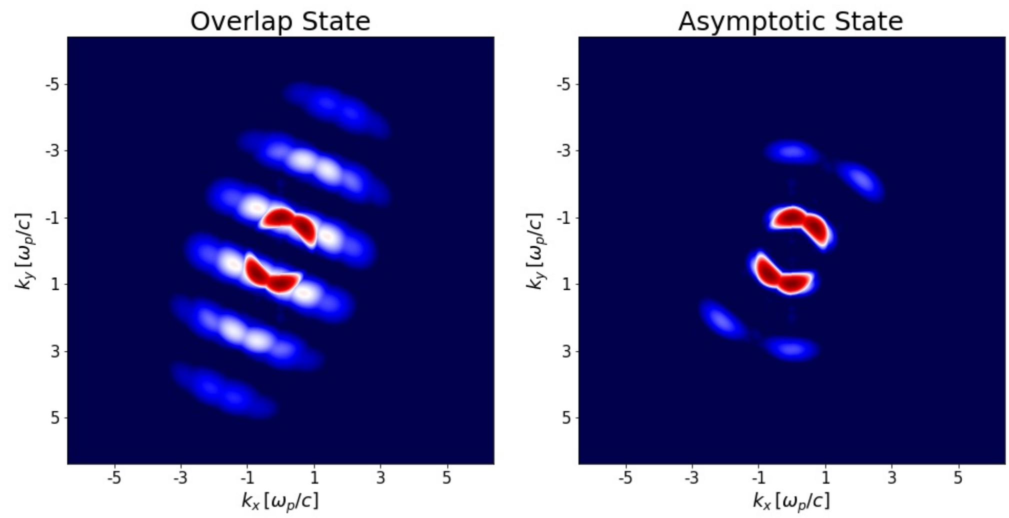


Figure 1.1 Harmonic Generation

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 CVode.
  In order to keep interpolation errors small do not choose this number too small.
- the multiple of steps at which you want the data to be written to disk.

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

A doxygen-generated complete code reference is provided with `ref.pdf`.

- Third, in the `src` directory, build the executable `Simulation` via the `make` command.
- Forth, run the simulation.
  Make sure to use `src` as working directory as the code uses a relative path to log the configuration in `main.cpp`.
  You determine the number of processes via the MPI execution command. Note that in 2D and 3D simulations this number has to coincide with the actual number of patches, as described above.
  Here, the simulation would be executed distributed over four processes:
  `mpirun -np 4 ./Simulation`
- Monitor stdout and stderr. The unique simulation identifier number (starting timestep = name of data directory), the process steps, and the used wall times per step are printed on stdout. Errors are printed on stderr.
  Note: Convergence of the employed CVode solver can not be guaranteed and issues of this kind can hardly be predicted. On top, they are even system dependent. Piece of advice: Only pass decimal numbers for the grid settings and initial conditions.
  CVode warnings and errors are reported on stdout and stderr.
  A `config.txt`file containing the 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 forth of the Nyquist frequency to avoid nonphysical dispersion effects. The wavelengths should neither be chosen too large (bulky wave) on a fine patchwork of narrow patches. Their communication might be problematic with too small halo layer depths. You would observe a blurring over time. The amplitudes need be below 1 – the critical field strength – for the weak-field expansion to be valid.
You can then investigate the arising of higher harmonics in frequency space via a Fourier analysis. The signals from the higher harmonics can be highlighted by subtracting the results of the same simulation in the linear Maxwell vacuum. You will be left with the nonlinear effects.
Choosing the probe pulse to be polarized with an angle to the polarization of the pump you may observe a fractional polarization flip of the probe due to their nonlinear interaction.
Decide beforehand which steps you need to be written to disk for your analysis.

Example scenarios of colliding Gaussians are preconfigured for any dimension.

1.3.2 Note on Resource Occupation

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

1.3.3 Note on Output Analysis

The field data are either written in csv format to one file per MPI process, the ending of which (after an underscore) corresponds to the process number, as described above. This is not an elegant solution, but a portable way that also works fast and is straightforward to analyze.
Or, the option recommended for many larger write operations, in binary format with a single file per output step. Raw bytes are written to the files as they are in memory. This option is more performant and achieved with MPI IO. However, there is no guarantee of portability; postprocessing/conversion is required. The step number is the file name.
A `SimResults` folder is created in the chosen output directory if it does not exist and a folder named after the starting timestep of the simulation (in the form `yy-mm-dd_hh-MM-ss`) is created where the output files are written into. There are six columns in the csv files, corresponding to the six components of the electromagnetic field: $`E_x`$, $`E_y`$, $`E_z`$, $`B_x`$, $`B_y`$, $`B_z`$. Each row corresponds to one lattice point.
Postprocessing is required to read-in the files in order. A Python `module` taking care of this is provided.
Likewise, 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 4x4x4=64 cores, the field data is divided over the patches as follows:

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

The axes denote the physical dimensions that are each divided into 4 sectors in this example. The numbers inside the 4x4 squares indicate the process number, which is the number of the patch and also the number at the end of the corresponding output csv file. The ordering of the array within a patch follows the standard C convention and can be reshaped in 2D and 3D to the actual size of the path.

More information describing settings and analysis procedures used for actual scientific results are given in the open-access `paper`.
Some example Python analysis scripts can be found in the examples. The `first steps` demonstrate how the simulated data is accurately read-in from disk to numpy arrays using the provided `get field data module`. `Harmonic generation` in various forms is sketched as one application showing nonlinear quantum vacuum effects. There is however *no simulation data provided* as it would make the repository size unnecessarily large.

1.4 Authors

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

Chapter 2

Hierarchical Index

2.1 Class Hierarchy

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

Chapter 3

Data Structure Index

3.1 Data Structures

Here are the data structures with brief descriptions:

Chapter 4

File Index

4.1 File List

Here is a list of all files with brief descriptions:

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.

```
                                            {
    kx = k[0];     /** - wavevectors \f$ k_x \f$ */
    ky = k[1];     /** - \f$ k_y \f$ */
    kz = k[2];     /** - \f$ k_z \f$ normalized to \f$ 1/\lambda \f$*/
    px = p[0];     /** - amplitude (polarization) in x-direction */
    py = p[1];     /** - amplitude (polarization) in y-direction */
    pz = p[2];     /** - amplitude (polarization) in z-direction */
    phix = phi[0]; /** - phase shift in x-direction */
    phiy = phi[1]; /** - phase shift in y-direction */
    phiz = phi[2]; /** - phase shift in z-direction */
    phig = phig_;  /** - width */
    x0x = xo[0];   /** - shift from origin in x-direction*/
    x0y = xo[1];   /** - shift from origin in y-direction*/
    x0z = xo[2];   /** - shift from origin in z-direction*/
}
```

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.

```
                                                          {
    const sunrealtype wavelength =
        sqrt(kx * kx + ky * ky + kz * kz); /* \f$ 1/\lambda \f$ */
    x = x - x0x; /* x-coordinate minus shift from origin */
    y = y - x0y; /* y-coordinate minus shift from origin */
    z = z - x0z; /* z-coordinate minus shift from origin */
    const sunrealtype kScalarX = (kx * x + ky * y + kz * z) * 2 *
                       numbers::pi; /* \f$ 2\pi \ \vec{k} \cdot \vec{x} \f$ */
    const sunrealtype envelopeAmp =
        exp(-(x * x + y * y + z * z) / phig / phig); /* enveloping Gauss shape */
    // Gaussian wave definition
    const array<sunrealtype, 3> E{
        {                                          /* E-field vector */
         px * cos(kScalarX - phix) * envelopeAmp,  /* \f$ E_x \f$ */
         py * cos(kScalarX - phiy) * envelopeAmp,  /* \f$ E_y \f$ */
         pz * cos(kScalarX - phiz) * envelopeAmp}}; /* \f$ E_z \f$ */
    // Put E-field into space
    pTo6Space[0] += E[0];
    pTo6Space[1] += E[1];
    pTo6Space[2] += E[2];
    // and B-field
    pTo6Space[3] += (ky * E[2] - kz * E[1]) / wavelength;
    pTo6Space[4] += (kz * E[0] - kx * E[2]) / wavelength;
    pTo6Space[5] += (kx * E[1] - ky * E[0]) / wavelength;
}
```

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

5.1.4 Field Documentation

5.1.4.1 kx

`sunrealtype Gauss1D::kx [private]`

wavenumber k_x

Definition at line 89 of file ICSetters.h.

Referenced by addToSpace(), and Gauss1D().

5.1.4.2 ky

`sunrealtype Gauss1D::ky [private]`

wavenumber k_y

Definition at line 91 of file ICSetters.h.

Referenced by addToSpace(), and Gauss1D().

5.1.4.3 kz

`sunrealtype Gauss1D::kz [private]`

wavenumber k_z

Definition at line 93 of file ICSetters.h.

Referenced by addToSpace(), and Gauss1D().

5.1.4.4 phig

`sunrealtype Gauss1D::phig [private]`

pulse width Φ_g

Definition at line 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 phip

  polarization rotation from TE-mode around propagation direction
- sunrealtype w0

  taille ω_0
- sunrealtype zr

  Rayleigh length $z_R = \pi\omega_0^2/\lambda$.
- sunrealtype Ph0

  center of beam Φ_0
- sunrealtype PhA

  length of beam Φ_A
- sunrealtype A1

  amplitude projection on TE-mode
- sunrealtype A2

  amplitude projection on xy-plane
- sunrealtype lambda

  wavelength λ

5.2.1 Detailed Description

class for Gaussian waves in 2D

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

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

Definition at line 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.

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

References A1, A2, Amp, axis, dis, lambda, Ph0, PhA, phip, w0, and zr.

5.2.3 Member Function Documentation

5.2.3.1 addToSpace()

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

function for the actual implementation in space

Definition at line 190 of file ICSetters.cpp.

```
                                                     {
  //\f$ \vec{x} = \vec{x}_0-\vec{dis} \f$ // coordinates minus distance to
  //origin
  x -= dis[0];
  y -= dis[1];
  // z-=dis[2];
  z = NAN;
  //  \f$ z_g = \vec{x}\cdot\vec{e}_g \f$ projection on propagation axis
  const sunrealtype zg =
      x * axis[0] + y * axis[1]; //+z*axis[2];  // =z-z0 -> propagation
                                       //direction, minus origin
  // \f$ r = \sqrt{\vec{x}^2 -z_g^2} \f$ -> pythagoras of radius minus
  // projection on prop axis
  const sunrealtype r = sqrt((x * x + y * y /*+z*z*/) -
                         zg * zg); // radial distance to propagation axis
  // \f$  w(z) = w0\sqrt{1+(z_g/z_R)^2} \f$
  const sunrealtype wz = w0 * sqrt(1 + (zg * zg / zr / zr)); // waist at position z
  // \f$ g(z) = atan(z_g/z_r) \f$
  const sunrealtype gz = atan(zg / zr); // Gouy phase
  // \f$ R(z) = z_g*(1+(z_r/z_g)^2) \f$
  sunrealtype Rz = NAN; // beam curvature
  if (zg != 0)
    Rz = zg * (1 + (zr * zr / zg / zg));
  else
    Rz = 1e308;
  // wavenumber \f$ k = 2\pi/\lambda \f$
  const sunrealtype k = 2 * numbers::pi / lambda;
  // \f$ \Phi_F = kr^2/(2*R(z))+g(z)-kz_g \f$
  const sunrealtype PhF =
      -k * r * r / (2 * Rz) + gz - k * zg; // to be inserted into cosine
  // \f$ G = \sqrt{w_0/w_z}\e^{-(r/w(z))^2}\e^{(zg-Ph0)^2/PhA^2}\cos(PhF) \f$
  // CVode is a diva, no chance to remove the square in the second exponential
  // -> h too small
  const sunrealtype G2D = sqrt(w0 / wz) * exp(-r * r / wz / wz) *
                       exp(-(zg - Ph0) * (zg - Ph0) / PhA / PhA) *
                       cos(PhF); // gauss shape
  // \f$ c_\alpha =\vec{e}_x\cdot\vec{axis} \f$
  // projection components; do like this for CVode convergence -> otherwise
  // results in machine error values for non-existant field components if
  // axis[0] and axis[1] are given
  const sunrealtype ca =
```

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

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↩_=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 $\vec{E} = A\,\vec{\epsilon}\,\frac{\omega_0}{\omega(z)}\,\exp\left(-r/\omega(z)\right)^2\,\exp\left(-((z_g-\Phi_0)/\Phi_A)^2\right)\,\cos\left(\frac{k\,r^2}{2R(z)}+g(z)-k\,z_g\right)$ with

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

Definition at line 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.

```
                                                          {
    dis = dis_;    /** - center it approaches */
    axis = axis_; /** - direction from where it comes */
    Amp = Amp_;    /** - amplitude */
    // pol=pol_;
    phip = phip_; /** - polarization rotation form TE-mode */
    w0 = w0_;      /** - taille */
    zr = zr_;      /** - Rayleigh length */
    Ph0 = Ph0_;    /** - beam center */
    PhA = PhA_;    /** - beam length */
    lambda = numbers::pi * w0 * w0 / zr;
    A1 = Amp * cos(phip);
    A2 = Amp * sin(phip);
}
```

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.

```
                                                            {
    x -= dis[0];
    y -= dis[1];
    z -= dis[2];
    const sunrealtype zg = x * axis[0] + y * axis[1] + z * axis[2];
    const sunrealtype r = sqrt((x * x + y * y + z * z) - zg * zg);
    const sunrealtype wz = w0 * sqrt(1 + (zg * zg / zr / zr));
    const sunrealtype gz = atan(zg / zr);
    sunrealtype Rz = NAN;
    if (zg != 0)
      Rz = zg * (1 + (zr * zr / zg / zg));
    else
      Rz = 1e308;
    const sunrealtype k = 2 * numbers::pi / lambda;
    const sunrealtype PhF = -k * r * r / (2 * Rz) + gz - k * zg;
    const sunrealtype G3D = (w0 / wz) * exp(-r * r / wz / wz) *
                            exp(-(zg - Ph0) * (zg - Ph0) / PhA / PhA) * cos(PhF);
    const sunrealtype ca = axis[0];
    const sunrealtype sa = sqrt(1 - ca * ca);
    pTo6Space[0] += sa * (G3D * A2);
    pTo6Space[1] += -ca * (G3D * A2);
    pTo6Space[2] += G3D * A1;
    pTo6Space[3] += -sa * (G3D * A1);
    pTo6Space[4] += ca * (G3D * A1);
    pTo6Space[5] += G3D * A2;
}
```

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::phip [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

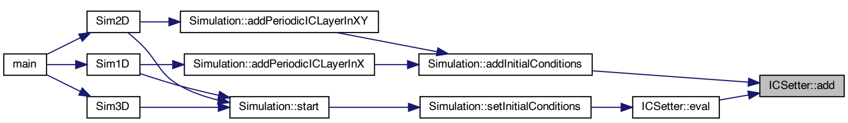
Definition at line 308 of file ICSetters.cpp.

```
                                                  {
  for (const auto &wave : planeWaves1D)
    wave.addToSpace(x, y, z, pTo6Space);
  for (const auto &wave : planeWaves2D)
    wave.addToSpace(x, y, z, pTo6Space);
  for (const auto &wave : planeWaves3D)
    wave.addToSpace(x, y, z, pTo6Space);
  for (const auto &wave : gauss1Ds)
    wave.addToSpace(x, y, z, pTo6Space);
  for (const auto &wave : gauss2Ds)
    wave.addToSpace(x, y, z, pTo6Space);
  for (const auto &wave : gauss3Ds)
    wave.addToSpace(x, y, z, pTo6Space);
}
```

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

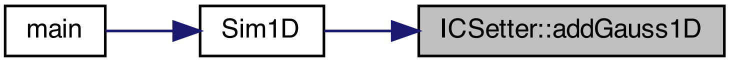
Definition at line 343 of file ICSetters.cpp.

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

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_ = 1.0l,
            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

Add Gaussian waves in 2D to their container vector

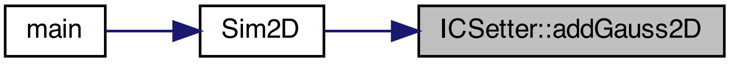
Definition at line 350 of file ICSetters.cpp.

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

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_ = 1.0l,
            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

Add Gaussian waves in 3D to their container vector

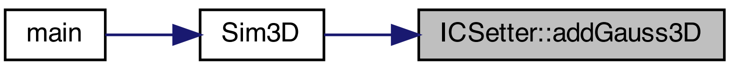
Definition at line 358 of file ICSetters.cpp.

```
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

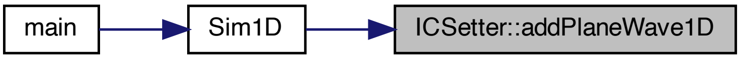
Definition at line 325 of file ICSetters.cpp.

```
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

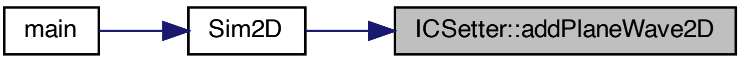
Definition at line 331 of file ICSetters.cpp.

```
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

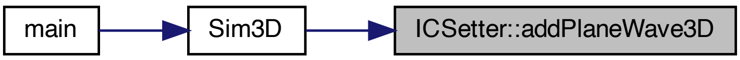
Definition at line 337 of file ICSetters.cpp.

```
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

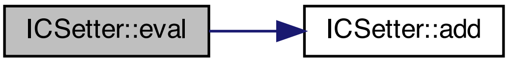
Definition at line 296 of file ICSetters.cpp.

```
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 }
```

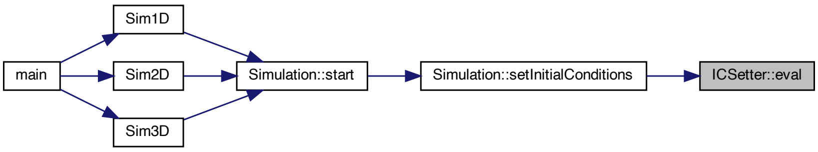
References add().

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

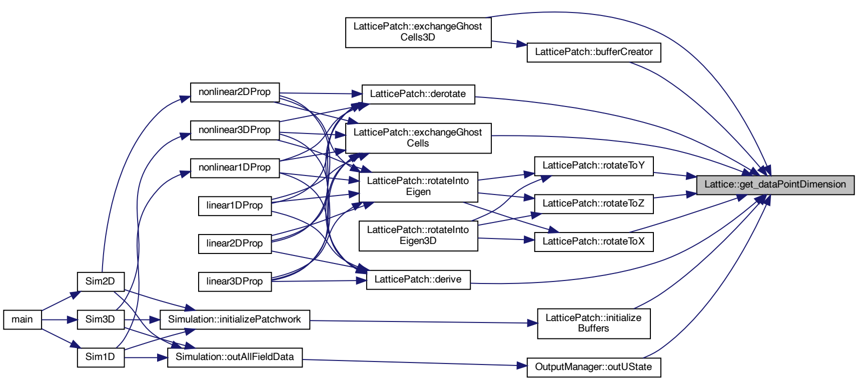
Definition at line 114 of file LatticePatch.h.

```
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

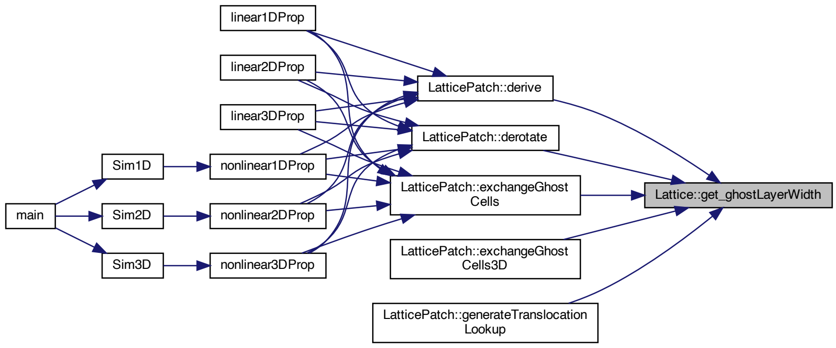
Definition at line 118 of file LatticePatch.h.

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

References ghostLayerWidth.

Referenced by LatticePatch::derive(), LatticePatch::derotate(), LatticePatch::exchangeGhostCells(), LatticePatch::exchangeGhostCell
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

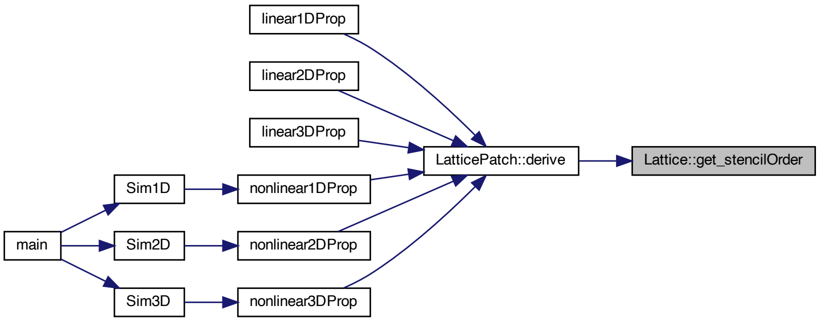
Definition at line 117 of file LatticePatch.h.

```
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

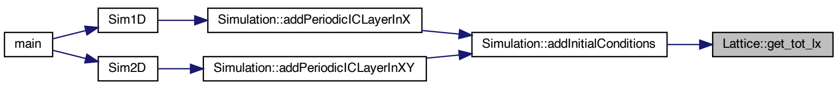
Definition at line 103 of file LatticePatch.h.

```
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

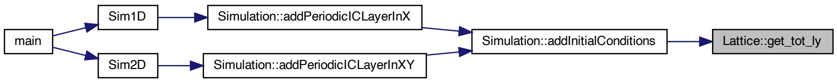
Definition at line 104 of file LatticePatch.h.

```
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

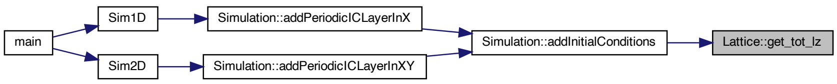
Definition at line 105 of file LatticePatch.h.

```
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.

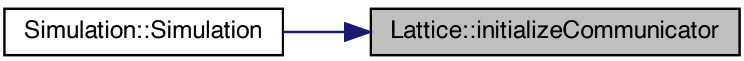
Definition at line 15 of file LatticePatch.cpp.

```
                                         {
    const int dims[3] = {nz, ny, nx};
    const int periods[3] = {static_cast<int>(per), static_cast<int>(per),
                      static_cast<int>(per)};
    // Create the cartesian communicator for MPI_COMM_WORLD
    MPI_Cart_create(MPI_COMM_WORLD, 3, dims, periods, 1, &comm);
    // Set MPI variables of the lattice
    MPI_Comm_size(comm, &(n_prc));
    MPI_Comm_rank(comm, &(my_prc));
    // Associate name to the communicator to identify it -> for debugging and
    // nicer error messages
    constexpr char lattice_comm_name[] = "Lattice";
    MPI_Comm_set_name(comm, lattice_comm_name);

    // Test if process naming is the same for both communicators
    /*
    int MYPRC;
    MPI_Comm_rank(MPI_COMM_WORLD,&MYPRC);
    cout«"\r"«my_prc«"\t"«MYPRC«endl;
    */
}
```

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.

```
{
  // copy the given data for number of points
  tot_nx = _nx;
  tot_ny = _ny;
  tot_nz = _nz;
  // compute the resulting number of points and datapoints
  tot_noP = tot_nx * tot_ny * tot_nz;
  tot_noDP = dataPointDimension * tot_noP;
  // compute the new Delta, the physical resolution
  dx = tot_lx / tot_nx;
  dy = tot_ly / tot_ny;
  dz = tot_lz / tot_nz;
}
```

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.

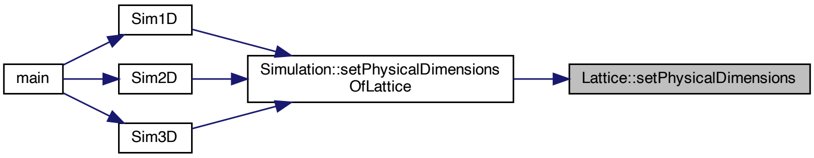
Definition at line 61 of file LatticePatch.cpp.

```
{
  tot_lx = _lx;
  tot_ly = _ly;
  tot_lz = _lz;
  // calculate physical distance between points
  dx = tot_lx / tot_nx;
  dy = tot_ly / tot_ny;
  dz = tot_lz / tot_nz;
  statusFlags |= FLatticeDimensionSet;
}
```

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:

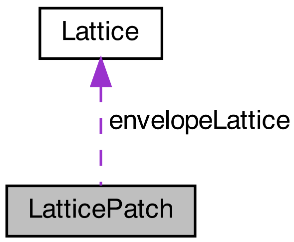
- src/LatticePatch.h
- src/LatticePatch.cpp

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 LIx

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

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

  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 lentgth 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.

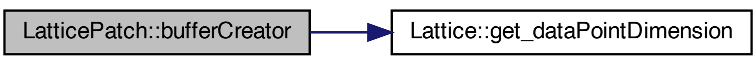
Definition at line 669 of file LatticePatch.cpp.

```
                                {
  const int dPD = envelopeLattice->get_dataPointDimension();
  // Initialize running index ui for to be transferred uData
  int ui = 0;

// #pragma omp parallel for reduction(+:ui) reduction(+:li) -> don't, probably
// bad idea to parallelize ghost cell exchange Loop over all planes and pick to
// be transferred points (uData indices)
#pragma distribute_point
  for (int iz = 0; iz < mz; iz++) {
    for (int iy = 0; iy < my; iy++) {
      // start index of uData vector halo data to be transferred
      // Here, in contrast to above, start at right boundary to send to left
      // s.t. updated left values are the first indices (bec of neighborhood
      // collective pattern) with each z-step add the whole xy-plane and with
      // y-step the x-range -> iterate all x-ranges
      ui = (iz * nx * ny + iy * nx + distToRight) * dPD;
      // copy from uData from right boundary (at each dimension) into buffer,
      // halo transfer size is given by x length at each step
      // memcpy(&ghostCellsToSend[li], \
                      &uData[ui], \
                      sizeof(sunrealtype)*mx*dPD);
      copy(&uData[ui], &uData[ui + mx * dPD], &ghostCellsToSend[li]);
      // increase li by transferred indices in this loop-step
      li += mx * dPD;
    }
  }

#pragma distribute_point
  for (int iz = 0; iz < mz; iz++) {
    for (int iy = 0; iy < my; iy++) {
      // Now copy from left boundary into buffer
      ui = (iz * nx * ny + iy * nx) * dPD;
      //memcpy(&ghostCellsToSend[li], \
                      &uData[ui], \
                      sizeof(sunrealtype)*mx*dPD);
      copy(&uData[ui], &uData[ui + mx * dPD], &ghostCellsToSend[li]);
      li += mx * dPD;
    }
  }
}
```

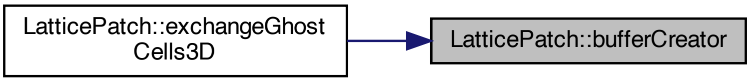
References envelopeLattice, Lattice::get_dataPointDimension(), ghostCellsToSend, nx, ny, and uData.

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.

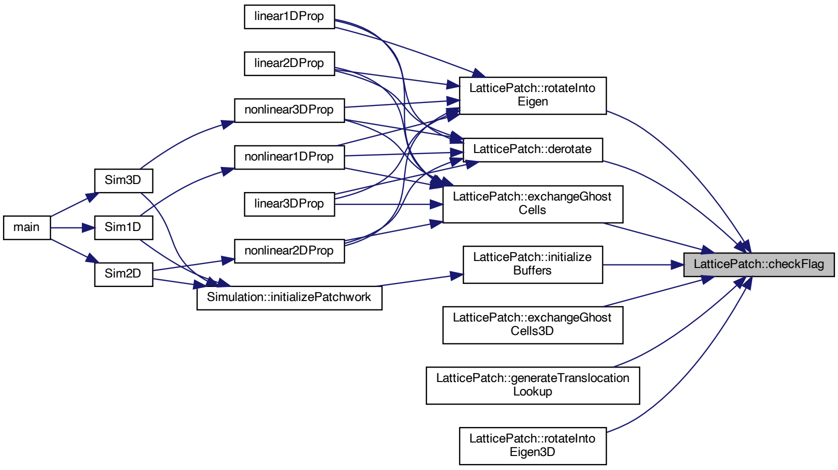
```
00713                                                      {
00714   if (!(statusFlags & flag)) {
00715     string errorMessage;
00716     switch (flag) {
00717     case FLatticePatchSetUp:
00718       errorMessage = "The Lattice patch was not set up please make sure to "
00719                      "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 dirWidthO = 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:
```

```
    for (int i = 0; i < perpPlainSize; i++) {
      for (int j = (i * dirWidth + gLW) * dPD;
           j < (i * dirWidth + gLW + dirWidthO) * dPD; j += dPD) {
        uAux[j + 0 - gLW * dPD] = s1b(&uAux[j + 0]) / dxi;
        uAux[j + 1 - gLW * dPD] = s1b(&uAux[j + 1]) / dxi;
        uAux[j + 2 - gLW * dPD] = s1f(&uAux[j + 2]) / dxi;
        uAux[j + 3 - gLW * dPD] = s1f(&uAux[j + 3]) / dxi;
        uAux[j + 4 - gLW * dPD] = s1f(&uAux[j + 4]) / dxi;
        uAux[j + 5 - gLW * dPD] = s1f(&uAux[j + 5]) / dxi;
      }
    }
    break;
  case 2:
    for (int i = 0; i < perpPlainSize; i++) {
      for (int j = (i * dirWidth + gLW) * dPD;
           j < (i * dirWidth + gLW + dirWidthO) * dPD; j += dPD) {
        uAux[j + 0 - gLW * dPD] = s2b(&uAux[j + 0]) / dxi;
        uAux[j + 1 - gLW * dPD] = s2b(&uAux[j + 1]) / dxi;
        uAux[j + 2 - gLW * dPD] = s2f(&uAux[j + 2]) / dxi;
        uAux[j + 3 - gLW * dPD] = s2f(&uAux[j + 3]) / dxi;
        uAux[j + 4 - gLW * dPD] = s2c(&uAux[j + 4]) / dxi;
        uAux[j + 5 - gLW * dPD] = s2c(&uAux[j + 5]) / dxi;
      }
    }
    break;
  case 3:
    for (int i = 0; i < perpPlainSize; i++) {
      for (int j = (i * dirWidth + gLW) * dPD;
           j < (i * dirWidth + gLW + dirWidthO) * dPD; j += dPD) {
        uAux[j + 0 - gLW * dPD] = s3b(&uAux[j + 0]) / dxi;
        uAux[j + 1 - gLW * dPD] = s3b(&uAux[j + 1]) / dxi;
        uAux[j + 2 - gLW * dPD] = s3f(&uAux[j + 2]) / dxi;
        uAux[j + 3 - gLW * dPD] = s3f(&uAux[j + 3]) / dxi;
        uAux[j + 4 - gLW * dPD] = s3f(&uAux[j + 4]) / dxi;
        uAux[j + 5 - gLW * dPD] = s3f(&uAux[j + 5]) / dxi;
      }
    }
    break;
  case 4:
    for (int i = 0; i < perpPlainSize; i++) {
      for (int j = (i * dirWidth + gLW) * dPD;
           j < (i * dirWidth + gLW + dirWidthO) * dPD; j += dPD) {
        uAux[j + 0 - gLW * dPD] = s4b(&uAux[j + 0]) / dxi;
        uAux[j + 1 - gLW * dPD] = s4b(&uAux[j + 1]) / dxi;
        uAux[j + 2 - gLW * dPD] = s4f(&uAux[j + 2]) / dxi;
        uAux[j + 3 - gLW * dPD] = s4f(&uAux[j + 3]) / dxi;
        uAux[j + 4 - gLW * dPD] = s4c(&uAux[j + 4]) / dxi;
        uAux[j + 5 - gLW * dPD] = s4c(&uAux[j + 5]) / dxi;
      }
    }
    break;
  case 5:
    for (int i = 0; i < perpPlainSize; i++) {
      for (int j = (i * dirWidth + gLW) * dPD;
           j < (i * dirWidth + gLW + dirWidthO) * dPD; j += dPD) {
        uAux[j + 0 - gLW * dPD] = s5b(&uAux[j + 0]) / dxi;
        uAux[j + 1 - gLW * dPD] = s5b(&uAux[j + 1]) / dxi;
        uAux[j + 2 - gLW * dPD] = s5f(&uAux[j + 2]) / dxi;
        uAux[j + 3 - gLW * dPD] = s5f(&uAux[j + 3]) / dxi;
        uAux[j + 4 - gLW * dPD] = s5f(&uAux[j + 4]) / dxi;
        uAux[j + 5 - gLW * dPD] = s5f(&uAux[j + 5]) / dxi;
      }
    }
    break;
  case 6:
    for (int i = 0; i < perpPlainSize; i++) {
      for (int j = (i * dirWidth + gLW) * dPD;
           j < (i * dirWidth + gLW + dirWidthO) * dPD; j += dPD) {
        uAux[j + 0 - gLW * dPD] = s6b(&uAux[j + 0]) / dxi;
        uAux[j + 1 - gLW * dPD] = s6b(&uAux[j + 1]) / dxi;
        uAux[j + 2 - gLW * dPD] = s6f(&uAux[j + 2]) / dxi;
        uAux[j + 3 - gLW * dPD] = s6f(&uAux[j + 3]) / dxi;
        uAux[j + 4 - gLW * dPD] = s6c(&uAux[j + 4]) / dxi;
        uAux[j + 5 - gLW * dPD] = s6c(&uAux[j + 5]) / dxi;
      }
    }
    break;
  case 7:
    for (int i = 0; i < perpPlainSize; i++) {
      for (int j = (i * dirWidth + gLW) * dPD;
           j < (i * dirWidth + gLW + dirWidthO) * dPD; j += dPD) {
        uAux[j + 0 - gLW * dPD] = s7b(&uAux[j + 0]) / dxi;
        uAux[j + 1 - gLW * dPD] = s7b(&uAux[j + 1]) / dxi;
        uAux[j + 2 - gLW * dPD] = s7f(&uAux[j + 2]) / dxi;
        uAux[j + 3 - gLW * dPD] = s7f(&uAux[j + 3]) / dxi;
        uAux[j + 4 - gLW * dPD] = s7f(&uAux[j + 4]) / dxi;
        uAux[j + 5 - gLW * dPD] = s7f(&uAux[j + 5]) / dxi;
```

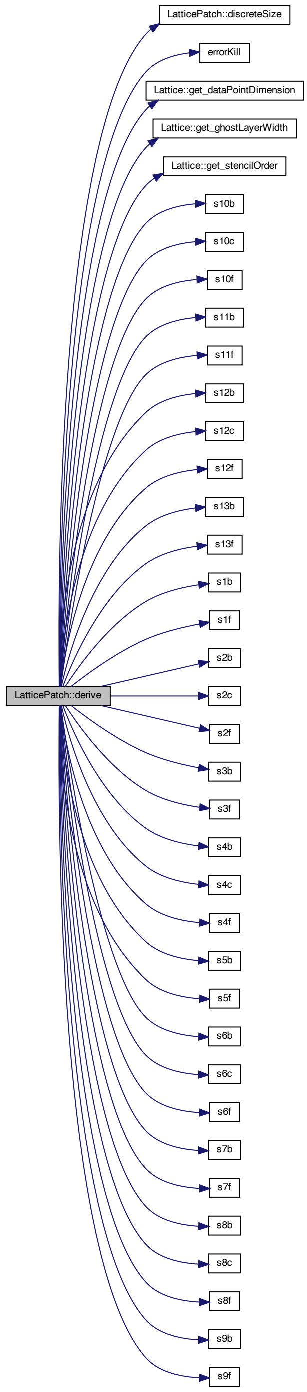
```
        }
      }
      break;
    case 8:
      for (int i = 0; i < perpPlainSize; i++) {
        for (int j = (i * dirWidth + gLW) * dPD;
             j < (i * dirWidth + gLW + dirWidthO) * dPD; j += dPD) {
          uAux[j + 0 - gLW * dPD] = s8b(&uAux[j + 0]) / dxi;
          uAux[j + 1 - gLW * dPD] = s8b(&uAux[j + 1]) / dxi;
          uAux[j + 2 - gLW * dPD] = s8f(&uAux[j + 2]) / dxi;
          uAux[j + 3 - gLW * dPD] = s8f(&uAux[j + 3]) / dxi;
          uAux[j + 4 - gLW * dPD] = s8c(&uAux[j + 4]) / dxi;
          uAux[j + 5 - gLW * dPD] = s8c(&uAux[j + 5]) / dxi;
        }
      }
      break;
    case 9:
      for (int i = 0; i < perpPlainSize; i++) {
        for (int j = (i * dirWidth + gLW) * dPD;
             j < (i * dirWidth + gLW + dirWidthO) * dPD; j += dPD) {
          uAux[j + 0 - gLW * dPD] = s9b(&uAux[j + 0]) / dxi;
          uAux[j + 1 - gLW * dPD] = s9b(&uAux[j + 1]) / dxi;
          uAux[j + 2 - gLW * dPD] = s9f(&uAux[j + 2]) / dxi;
          uAux[j + 3 - gLW * dPD] = s9f(&uAux[j + 3]) / dxi;
          uAux[j + 4 - gLW * dPD] = s9f(&uAux[j + 4]) / dxi;
          uAux[j + 5 - gLW * dPD] = s9f(&uAux[j + 5]) / dxi;
        }
      }
      break;
    case 10:
      for (int i = 0; i < perpPlainSize; i++) {
        for (int j = (i * dirWidth + gLW) * dPD;
             j < (i * dirWidth + gLW + dirWidthO) * dPD; j += dPD) {
          uAux[j + 0 - gLW * dPD] = s10b(&uAux[j + 0]) / dxi;
          uAux[j + 1 - gLW * dPD] = s10b(&uAux[j + 1]) / dxi;
          uAux[j + 2 - gLW * dPD] = s10f(&uAux[j + 2]) / dxi;
          uAux[j + 3 - gLW * dPD] = s10f(&uAux[j + 3]) / dxi;
          uAux[j + 4 - gLW * dPD] = s10c(&uAux[j + 4]) / dxi;
          uAux[j + 5 - gLW * dPD] = s10c(&uAux[j + 5]) / dxi;
        }
      }
      break;
    case 11:
      for (int i = 0; i < perpPlainSize; i++) {
        for (int j = (i * dirWidth + gLW) * dPD;
             j < (i * dirWidth + gLW + dirWidthO) * dPD; j += dPD) {
          uAux[j + 0 - gLW * dPD] = s11b(&uAux[j + 0]) / dxi;
          uAux[j + 1 - gLW * dPD] = s11b(&uAux[j + 1]) / dxi;
          uAux[j + 2 - gLW * dPD] = s11f(&uAux[j + 2]) / dxi;
          uAux[j + 3 - gLW * dPD] = s11f(&uAux[j + 3]) / dxi;
          uAux[j + 4 - gLW * dPD] = s11f(&uAux[j + 4]) / dxi;
          uAux[j + 5 - gLW * dPD] = s11f(&uAux[j + 5]) / dxi;
        }
      }
      break;
    case 12:
      for (int i = 0; i < perpPlainSize; i++) {
        for (int j = (i * dirWidth + gLW) * dPD;
             j < (i * dirWidth + gLW + dirWidthO) * dPD; j += dPD) {
          uAux[j + 0 - gLW * dPD] = s12b(&uAux[j + 0]) / dxi;
          uAux[j + 1 - gLW * dPD] = s12b(&uAux[j + 1]) / dxi;
          uAux[j + 2 - gLW * dPD] = s12f(&uAux[j + 2]) / dxi;
          uAux[j + 3 - gLW * dPD] = s12f(&uAux[j + 3]) / dxi;
          uAux[j + 4 - gLW * dPD] = s12c(&uAux[j + 4]) / dxi;
          uAux[j + 5 - gLW * dPD] = s12c(&uAux[j + 5]) / dxi;
        }
      }
      break;
    case 13:
      //#pragma omp parallel for default(none) firstprivate(uAux)
      //shared(dxi,dirWidth,dirWidth0,gLW,dPD) collapse(2) schedule(static,6)
      //#pragma ivdep
      //#pragma distribute_point -> No.
      //#pragma unroll_and_jam
      // Iterate through all points in the plane perpendicular to the given
      // direction
      for (int i = 0; i < perpPlainSize; i++) {
        // stencil functions range over 2*gLW+6 indices, attention to cache-line
        // false-sharing
        //#pragma omp simd safelen(2*gLW+dPD)
        // Iterate through the direction for each perpendicular plane point
        for (int j = (i * dirWidth + gLW /*to shift left by gLW below */) * dPD;
             j < (i * dirWidth + gLW + dirWidthO) * dPD; j += dPD) {
          /* Compute the stencil derivative for any of the six field components
           * with a ghostlayer width adjusted to the order of the finite
           * difference scheme */
          uAux[j + 0 - gLW * dPD] = s13b(&uAux[j + 0]) / dxi;
```

```
                uAux[j + 1 - gLW * dPD] = s13b(&uAux[j + 1]) / dxi;
                uAux[j + 2 - gLW * dPD] = s13f(&uAux[j + 2]) / dxi;
                uAux[j + 3 - gLW * dPD] = s13f(&uAux[j + 3]) / dxi;
                uAux[j + 4 - gLW * dPD] = s13f(&uAux[j + 4]) / dxi;
                uAux[j + 5 - gLW * dPD] = s13f(&uAux[j + 5]) / dxi;
            }
        }
        break;

    default:
        errorKill("Please set an existing stencil order");
        break;
    }
}
```

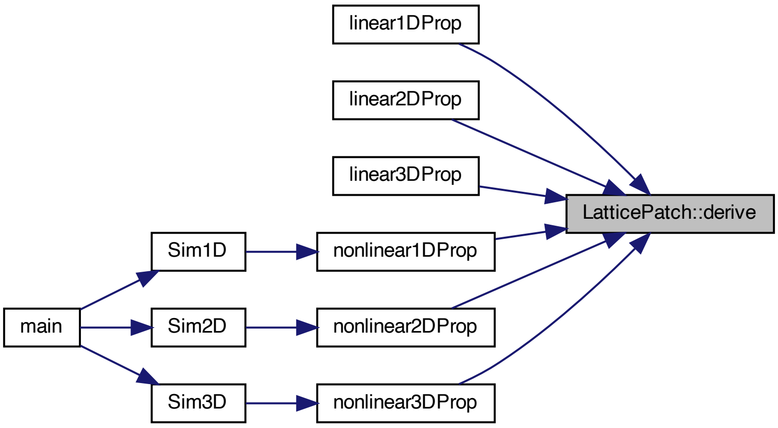
References discreteSize(), dx, dy, dz, envelopeLattice, errorKill(), Lattice::get_dataPointDimension(), Lattice::get_ghostLayerWidth(), Lattice::get_stencilOrder(), s10b(), s10c(), s10f(), s11b(), s11f(), s12b(), s12c(), s12f(), s13b(), s13f(), s1b(), s1f(), s2b(), s2c(), s2f(), s3b(), s3f(), s4b(), 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.

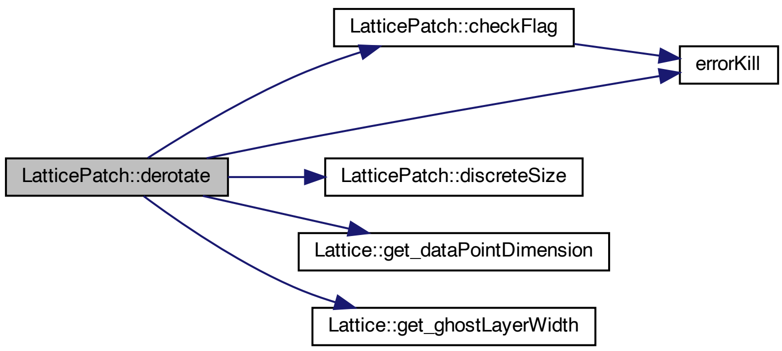
```
00426                                                             {
00427   // Check that the lattice as well as the translocation lookups have been set
00428   // up;
00429   checkFlag(FLatticePatchSetUp);
00430   checkFlag(TranslocationLookupSetUp);
00431   const int dPD = envelopeLattice->get_dataPointDimension();
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
```

```
    for (int i = 0; i < uSize; i++) {
      target = dPD * i;
      ii = dPD * (uToy[i] - gLW);
      buffOut[target + 0] = (uAux[ii] - uAux[2 + ii]) / 2.;
      buffOut[target + 1] = uAux[5 + ii];
      buffOut[target + 2] = (-uAux[1 + ii] + uAux[3 + ii]) / 2.;
      buffOut[target + 3] = (uAux[1 + ii] + uAux[3 + ii]) / 2.;
      buffOut[target + 4] = uAux[4 + ii];
      buffOut[target + 5] = (uAux[ii] + uAux[2 + ii]) / 2.;
    }
    break;
  case 3:
#pragma omp simd // safelen(6)
#pragma distribute_point
    for (int i = 0; i < uSize; i++) {
      target = dPD * i;
      ii = dPD * (uToz[i] - gLW);
      buffOut[target + 0] = (-uAux[ii] + uAux[2 + ii]) / 2.;
      buffOut[target + 1] = (uAux[1 + ii] - uAux[3 + ii]) / 2.;
      buffOut[target + 2] = uAux[5 + ii];
      buffOut[target + 3] = (uAux[1 + ii] + uAux[3 + ii]) / 2.;
      buffOut[target + 4] = (uAux[ii] + uAux[2 + ii]) / 2.;
      buffOut[target + 5] = uAux[4 + ii];
    }
    break;
  default:
    errorKill("Tried to derotate from the wrong direction");
    break;
  }
}
```

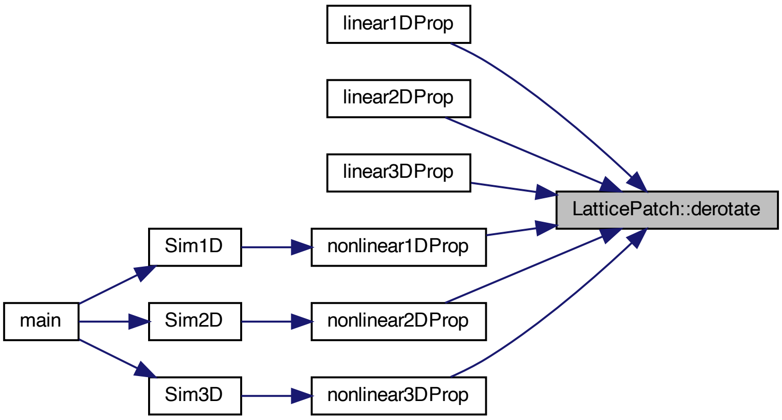
References checkFlag(), discreteSize(), envelopeLattice, errorKill(), FLatticePatchSetUp, Lattice::get_dataPointDimension(), Lattice::get_ghostLayerWidth(), TranslocationLookupSetUp, uAux, uTox, uToy, and uToz.

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

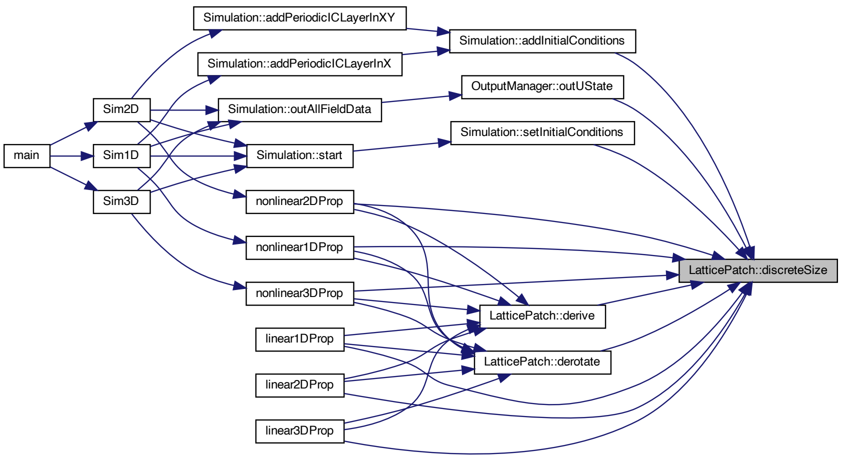
Definition at line 196 of file LatticePatch.cpp.

```
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.

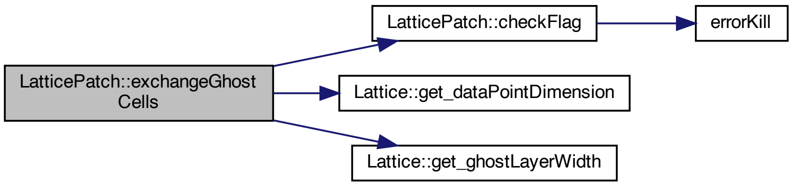
```
00502                                                   {
00503   // Check that the lattice has been set up
00504   checkFlag(FLatticeDimensionSet);
00505   checkFlag(FLatticePatchSetUp);
00506   // Variables to per dimension calculate the halo indices, and distance to
00507   // other side halo 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
```

```
  ghostCellLeft.resize(exchangeSize);
  ghostCellRight.resize(ghostCellLeft.size());
  ghostCellLeftToSend.resize(ghostCellLeft.size());
  ghostCellRightToSend.resize(ghostCellLeft.size());
  gCLData = &ghostCellLeft[0];
  gCRData = &ghostCellRight[0];
  statusFlags |= GhostLayersInitialized;

  // Initialize running index li for the halo buffers, and index ui of uData for
  // data transfer
  int li = 0, ui = 0;

// #pragma omp parallel for reduction(+:ui) reduction(+:li) -> don't probably
// bad idea to parallelize ghost cell exchange Loop over to be copied points in
// z and y direction
#pragma distribute_point
  for (int iz = 0; iz < mz; iz++) {
    for (int iy = 0; iy < my; iy++) {
      // uData vector start index of halo data to be transferred
      // with each z-step add the whole xy-plane and with y-step the x-range ->
      // iterate all x-ranges
      ui = (iz * nx * ny + iy * nx) * dPD;
      // copy left halo data from uData to buffer, transfer size is given by
      // x-length (not x-range) perhaps faster but more fragile C lib copy
      // operation (contained in cstring header)
      /*
      memcpy(&ghostCellLeftToSend[li],
             &uData[ui],
             sizeof(sunrealtype)*mx*dPD);
      // increase ui by the distance to vis-a-vis boundary and copy right halo
      data to buffer ui+=distToRight*dPD; memcpy(&ghostCellRightToSend[li],
             &uData[ui],
             sizeof(sunrealtype)*mx*dPD);
      */
      // perhaps more safe but slower copy operation (contained in algorithm
      // header) performance highly system dependent
      copy(&uData[ui], &uData[ui + mx * dPD], &ghostCellLeftToSend[li]);
      ui += distToRight * dPD;
      copy(&uData[ui], &uData[ui + mx * dPD], &ghostCellRightToSend[li]);

      // increase halo index by transferred items per y-iteration step
      // (x-length)
      li += mx * dPD;
    }
  }

  /* Send and receive the data to and from neighboring latticePatches */
  // Adjust direction to cartesian communicator
  int dim = 2; // default for dir==1
  if (dir == 2) {
    dim = 1;
  } else if (dir == 3) {
    dim = 0;
  }
  int rank_source = 0, rank_dest = 0;
  MPI_Cart_shift(envelopeLattice->comm, dim, -1, &rank_source,
                 &rank_dest); // s.t. rank_dest is left & v.v.

  // nonblocking Isend/Irecv

  MPI_Request requests[4];
  MPI_Irecv(&ghostCellRight[0], exchangeSize, MPI_SUNREALTYPE, rank_source, 1,
  envelopeLattice->comm, &requests[0]);
  MPI_Isend(&ghostCellLeftToSend[0], exchangeSize, MPI_SUNREALTYPE, rank_dest,
  1, envelopeLattice->comm, &requests[1]);
  MPI_Irecv(&ghostCellLeft[0], exchangeSize, MPI_SUNREALTYPE, rank_dest, 2,
  envelopeLattice->comm, &requests[2]);
  MPI_Isend(&ghostCellRightToSend[0], exchangeSize, MPI_SUNREALTYPE,
  rank_source, 2, envelopeLattice->comm, &requests[3]);
  MPI_Waitall(4, requests, MPI_STATUS_IGNORE);


  // blocking Sendrecv:
  /*
  MPI_Sendrecv(&ghostCellLeftToSend[0], exchangeSize, MPI_SUNREALTYPE,
               rank_dest, 1, &ghostCellRight[0], exchangeSize, MPI_SUNREALTYPE,
               rank_source, 1, envelopeLattice->comm, MPI_STATUS_IGNORE);
  MPI_Sendrecv(&ghostCellRightToSend[0], exchangeSize, MPI_SUNREALTYPE,
               rank_source, 2, &ghostCellLeft[0], exchangeSize, MPI_SUNREALTYPE,
               rank_dest, 2, envelopeLattice->comm, MPI_STATUS_IGNORE);
  */
}
```

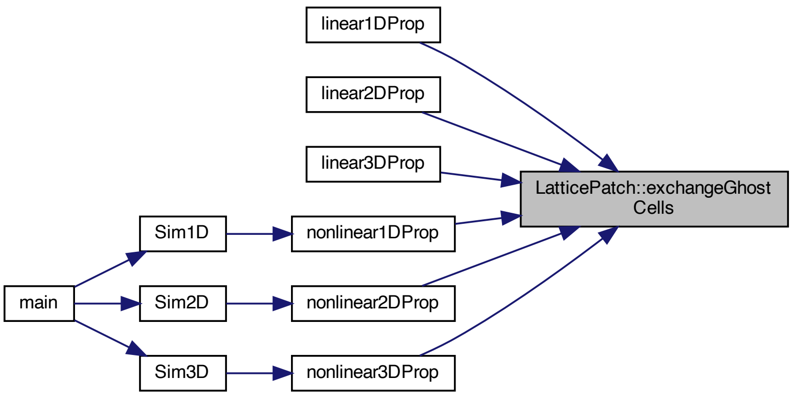
References checkFlag(), Lattice::comm, envelopeLattice, FLatticeDimensionSet, FLatticePatchSetUp, gCLData, gCRData, Lattice::get_dataPointDimension(), Lattice::get_ghostLayerWidth(), ghostCellLeft, ghostCellLeftToSend, ghostCellRight, ghostCellRightToSend, GhostLayersInitialized, nx, ny, nz, statusFlags,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

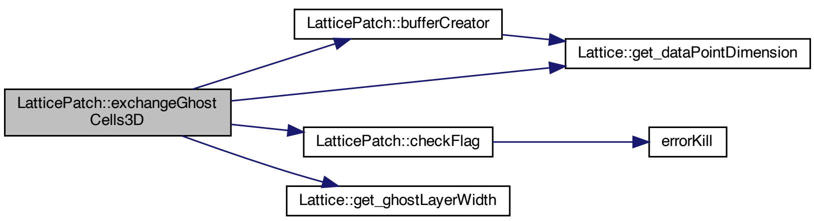
Definition at line 620 of file LatticePatch.cpp.

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

```
   const int exchangeSize = n * n * gLW * dPD;
   // Give ghostCells the total size of the ghost layers (the six halos)
   const int tot_exchangeSize = 6 * exchangeSize;
   ghostCells.resize(tot_exchangeSize);
   ghostCellsToSend.resize(ghostCells.size());
   // ghost cell data in all directions: left,right,bottom,top,front,abaft; but
   // with MPI dim order "lefts" are the first receivers and "rights" are the
   // first senders -> see buffer creator below
   gCFData = &ghostCells[0];
   gCAData = &ghostCells[exchangeSize];
   gCBData = &ghostCells[2 * exchangeSize];
   gCTData = &ghostCells[3 * exchangeSize];
   gCLData = &ghostCells[4 * exchangeSize];
   gCRData = &ghostCells[5 * exchangeSize];
   statusFlags |= GhostLayersInitialized;

   checkFlag(FLatticePatchSetUp);
   // variables to set to ghost layer width and point distance to next
   // communication point -> depends on direction

   int li = 0;          // running index for buffers
   int distToRight = 0; // distance to vis-a-vis halo data, varies per dim
   // filling buffers along the MPI dim order
   distToRight = n * n * (n - gLW);
   bufferCreator(li, n, n, gLW, distToRight);
   li += 2 * exchangeSize; // li increases by two exchange sizes per dim

   distToRight = n * (n - gLW);
   bufferCreator(li, n, gLW, n, distToRight);
   li += 2 * exchangeSize;

   distToRight = (n - gLW);
   bufferCreator(li, gLW, n, n, distToRight);

   MPI_Neighbor_alltoall(&ghostCellsToSend[0], exchangeSize, MPI_SUNREALTYPE,
                         &ghostCells[0], exchangeSize, MPI_SUNREALTYPE,
                         envelopeLattice->comm);
}
```

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

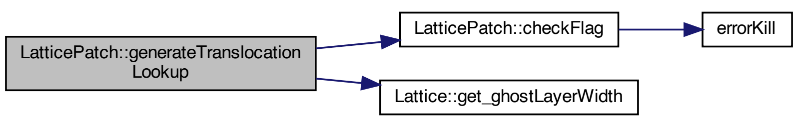
To avoid cache misses: create vectors to translate u vector into space coordinates and vice versa and same for left and right ghost layers to space

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     // calulate 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.

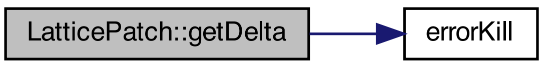
Definition at line 228 of file LatticePatch.cpp.

```
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 }
```

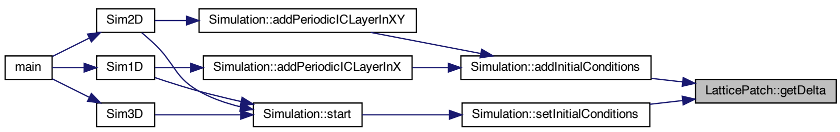
References dx, dy, dz, and errorKill().

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.

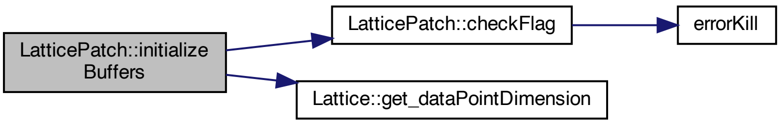
Definition at line 487 of file LatticePatch.cpp.

```
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 }
```

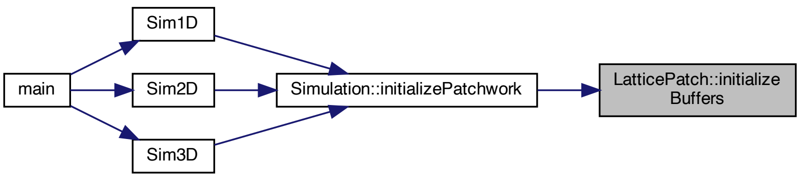
References buffData, BuffersInitialized, buffX, buffY, buffZ, checkFlag(), envelopeLattice, FLatticeDimensionSet, Lattice::get_dataPointDimension(), nx, ny, nz, and statusFlags.

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.

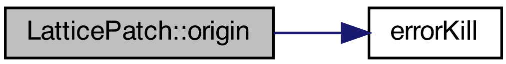
Definition at line 213 of file LatticePatch.cpp.

```
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 }
```

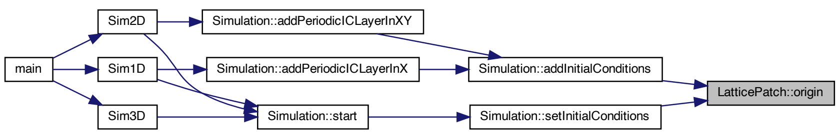
References errorKill(), x0, y0, and z0.

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

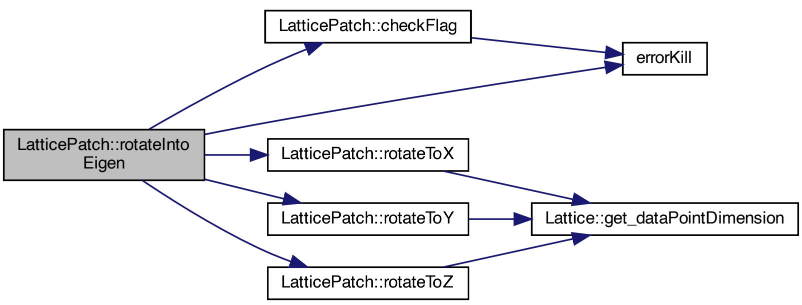
Definition at line 314 of file LatticePatch.cpp.

```
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 }
```

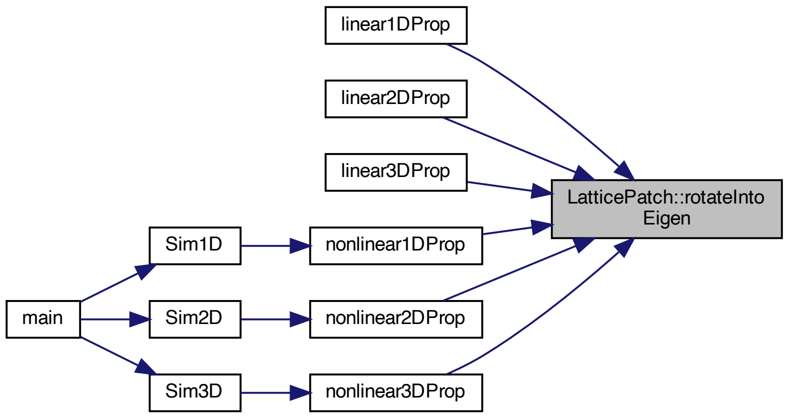
References checkFlag(), errorKill(), FLatticePatchSetUp, gCLData, gCRData, GhostLayersInitialized, lgcTox, lgcToy, lgcToz, rgcTox, rgcToy, rgcToz, rotateToX(), rotateToY(), rotateToZ(), TranslocationLookupSetUp, uAuxData, uData, uTox, uToy, and uToz.

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

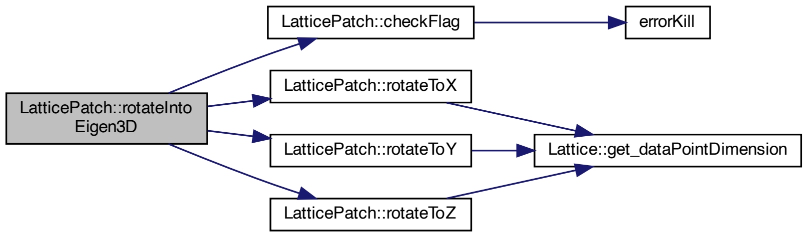
Same as `rotateIntoEigen` but for neighborhood 3D halo buffers.

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

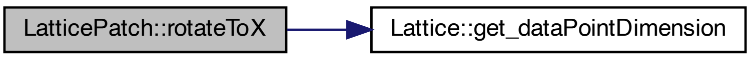
Definition at line 361 of file LatticePatch.cpp.

```
                                                                  {
  int ii = 0, target = 0;
#pragma ivdep
#pragma omp simd // safelen(6) - also good
#pragma distribute_point
  for (unsigned int i = 0; i < lookup.size(); i++) {
    // get correct u-vector and spatial indices along previously defined lookup
    // tables
    target = envelopeLattice->get_dataPointDimension() * lookup[i];
    ii = envelopeLattice->get_dataPointDimension() * i;
    outArray[target + 0] = -inArray[1 + ii] + inArray[5 + ii];
    outArray[target + 1] = inArray[2 + ii] + inArray[4 + ii];
    outArray[target + 2] = inArray[1 + ii] + inArray[5 + ii];
    outArray[target + 3] = -inArray[2 + ii] + inArray[4 + ii];
    outArray[target + 4] = inArray[3 + ii];
    outArray[target + 5] = inArray[ii];
  }
}
```

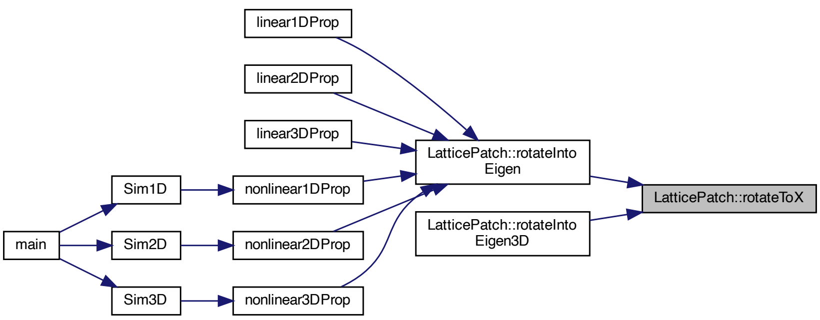
References envelopeLattice, and Lattice::get_dataPointDimension().

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 Ry into eigenspace of Z matrix and write to auxiliary vector

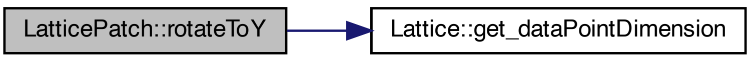
Definition at line 384 of file LatticePatch.cpp.

```
                                                                  {
  int ii = 0, target = 0;
#pragma ivdep
#pragma omp simd // safelen(6)
#pragma distribute_point
  for (unsigned int i = 0; i < lookup.size(); i++) {
    target = envelopeLattice->get_dataPointDimension() * lookup[i];
    ii = envelopeLattice->get_dataPointDimension() * i;
    outArray[target + 0] = inArray[ii] + inArray[5 + ii];
    outArray[target + 1] = -inArray[2 + ii] + inArray[3 + ii];
    outArray[target + 2] = -inArray[ii] + inArray[5 + ii];
    outArray[target + 3] = inArray[2 + ii] + inArray[3 + ii];
    outArray[target + 4] = inArray[4 + ii];
    outArray[target + 5] = inArray[1 + ii];
  }
}
```

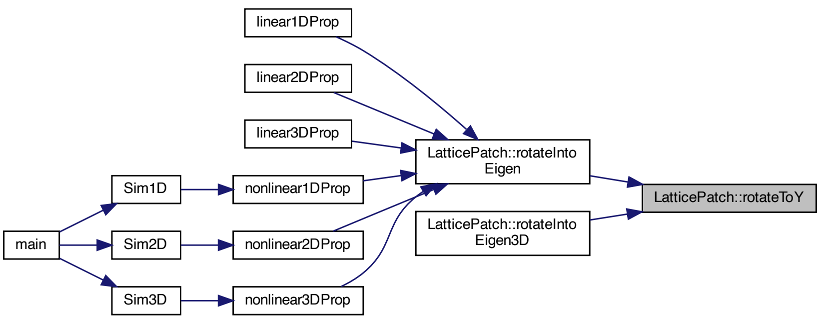
References envelopeLattice, and Lattice::get_dataPointDimension().

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 Rz into eigenspace of Z matrix and write to auxiliary vector

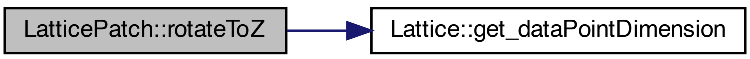
Definition at line 405 of file LatticePatch.cpp.

```
                                                           {
  int ii = 0, target = 0;
#pragma ivdep
#pragma omp simd // safelen(6)
#pragma distribute_point
  for (unsigned int i = 0; i < lookup.size(); i++) {
    target = envelopeLattice->get_dataPointDimension() * lookup[i];
    ii = envelopeLattice->get_dataPointDimension() * i;
    outArray[target + 0] = -inArray[ii] + inArray[4 + ii];
    outArray[target + 1] = inArray[1 + ii] + inArray[3 + ii];
    outArray[target + 2] = inArray[ii] + inArray[4 + ii];
    outArray[target + 3] = -inArray[1 + ii] + inArray[3 + ii];
    outArray[target + 4] = inArray[5 + ii];
    outArray[target + 5] = inArray[2 + ii];
  }
}
```

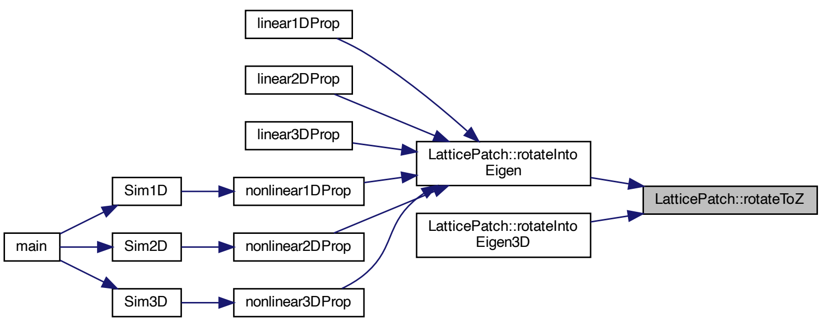
References envelopeLattice, and Lattice::get_dataPointDimension().

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.

```
                                                                          {
    // Retrieve the ghost layer depth
    const int gLW = envelopeLattice.get_ghostLayerWidth();
    // Retrieve the data point dimension
    const int dPD = envelopeLattice.get_dataPointDimension();
    // MPI process/patch
    const int my_prc = envelopeLattice.my_prc;
    // Determine thicknes of the slice
    const sunindextype tot_NOXP = envelopeLattice.get_tot_nx(); // total points of lattice
    const sunindextype tot_NOYP = envelopeLattice.get_tot_ny();
    const sunindextype tot_NOZP = envelopeLattice.get_tot_nz();
    // position of the patch in the lattice of patches -> process associated to
    // position
    const sunindextype LIx = my_prc % DLx;
    const sunindextype LIy = (my_prc / DLx) % DLy;
    const sunindextype LIz = (my_prc / DLx) / DLy;
    // Determine the number of points in the patch and first absolute points in
    // each dimension
    const sunindextype local_NOXP = tot_NOXP / DLx;
    const sunindextype local_NOYP = tot_NOYP / DLy;
    const sunindextype local_NOZP = tot_NOZP / DLz;
    // absolute positions of the first point in each dimension
    const sunindextype firstXPoint = local_NOXP * LIx;
    const sunindextype firstYPoint = local_NOYP * LIy;
    const sunindextype firstZPoint = local_NOZP * LIz;
    // total number of points in the patch
    const sunindextype local_NODP = dPD * local_NOXP * local_NOYP * local_NOZP;

    // Set patch up with above derived quantities
    /*
    // Experiment: Resolution can be adapted for each process/patch
    const int Scaler=4; // Better hand over as parameter via `initializePatchwork`
    if(my_prc==0){patchToMold.dx=envelopeLattice.get_dx();}
    else if(my_prc==1){patchToMold.dx=envelopeLattice.get_dx()*Scaler;}
    else{errorKill("Only do this resolution barrier test with 2 processes.");}
    */
    patchToMold.dx = envelopeLattice.get_dx();
    patchToMold.dy = envelopeLattice.get_dy();
    patchToMold.dz = envelopeLattice.get_dz();
    patchToMold.x0 = firstXPoint * patchToMold.dx;
    patchToMold.y0 = firstYPoint * patchToMold.dy;
    patchToMold.z0 = firstZPoint * patchToMold.dz;
    patchToMold.LIx = LIx;
    patchToMold.LIy = LIy;
    patchToMold.LIz = LIz;
    patchToMold.nx = local_NOXP;
    patchToMold.ny = local_NOYP;
    patchToMold.nz = local_NOZP;
    patchToMold.lx = patchToMold.nx * patchToMold.dx;
    patchToMold.ly = patchToMold.ny * patchToMold.dy;
    patchToMold.lz = patchToMold.nz * patchToMold.dz;
    /* // Check name of lattice communicator
    char lattice_comm_name[MPI_MAX_OBJECT_NAME];
    int lattice_namelen;
    MPI_Comm_get_name(envelopeLattice.comm, lattice_comm_name, &lattice_namelen);
    cout«"envelopeLattice.comm gives  "« lattice_comm_name « endl;
    */
    /* Create and allocate memory for parallel vectors with defined local and
     * global lenghts *
     * (-> CVode problem sizes Nlocal and N)
```

```
 * for field data and temporal derivatives and set extra pointers to them */
patchToMold.u =
    N_VNew_Parallel(envelopeLattice.comm, local_NODP,
                    envelopeLattice.get_tot_noDP(), envelopeLattice.sunctx);
patchToMold.uData = NV_DATA_P(patchToMold.u);
patchToMold.du =
    N_VNew_Parallel(envelopeLattice.comm, local_NODP,
                    envelopeLattice.get_tot_noDP(), envelopeLattice.sunctx);
patchToMold.duData = NV_DATA_P(patchToMold.du);
// Allocate space for auxiliary uAux so that the lattice and all possible
// directions of ghost Layers fit
const int s1 = patchToMold.nx, s2 = patchToMold.ny, s3 = patchToMold.nz;
const int s_min = min(s1, min(s2, s3));
patchToMold.uAux.resize(s1 * s2 * s3 / s_min * (s_min + 2 * gLW) * dPD);
patchToMold.uAuxData = &patchToMold.uAux[0];
patchToMold.envelopeLattice = &envelopeLattice;
// Set patch "name" to process number -> only for debugging
// patchToMold.ID=my_prc;
// set flag
patchToMold.statusFlags = FLatticePatchSetUp;
patchToMold.generateTranslocationLookup();
return 0;
}
```

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 LIx

```
sunindextype LatticePatch::LIx [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 LIy

`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 (auxilliarly) 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
  varible to safe 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

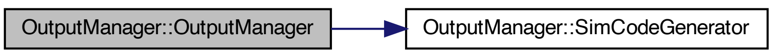
Directly generate the simCode at construction.

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 direcory "SimResults" and a directory with the simCode. The relevant part of the main file is written to a "config.txt" file in that directory to log the settings.

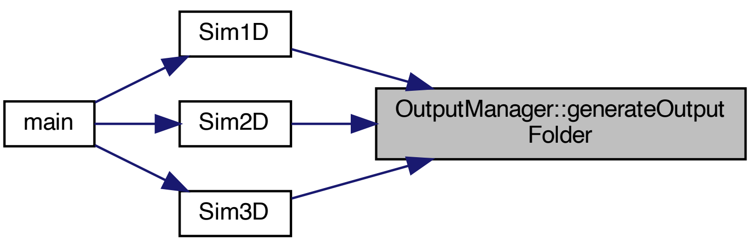
Definition at line 40 of file Outputters.cpp.

```
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

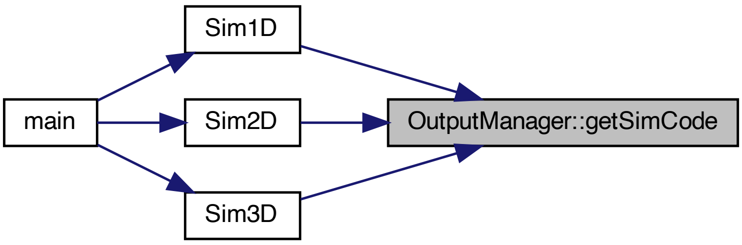
Definition at line 45 of file Outputters.h.

```
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.

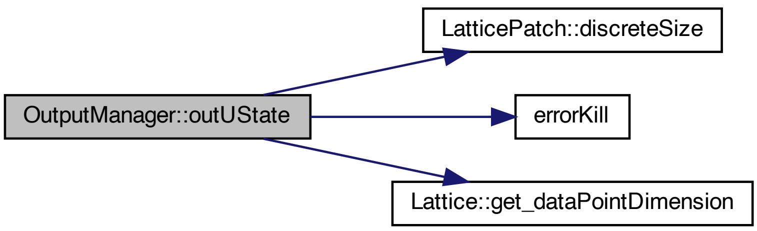
```
00086                                              {
00087   switch(outputStyle){
00088       case 'c': { // one csv file per process
00089   ofstream ofs;
00090   ofs.open(Path + to_string(state) + "_" + to_string(lattice.my_prc) + ".csv");
00091   // Precision of sunrealtype in significant decimal 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]
```

```
        « endl;
    }
    ofs.close();
    break;
                }

        case 'b': { // a single binary file
    // Open the output file
    MPI_File fh;
    const string filename = Path+to_string(state);
    MPI_File_open(lattice.comm,&filename[0],MPI_MODE_WRONLY|MPI_MODE_CREATE,
            MPI_INFO_NULL,&fh);
    // number of datapoints in the patch with process offset
    const int count = latticePatch.discreteSize()*lattice.get_dataPointDimension();
    MPI_Offset offset = lattice.my_prc*count*sizeof(MPI_SUNREALTYPE);
    // Go to offset in file and write data to it; maximal precision in
    // "native" representation
    MPI_File_set_view(fh,offset,MPI_SUNREALTYPE,MPI_SUNREALTYPE,"native",
            MPI_INFO_NULL);
    MPI_File_write_all(fh,latticePatch.uData,count,MPI_SUNREALTYPE,MPI_STATUS_IGNORE);
    /*
    MPI_Request write_request;
    MPI_File_iwrite_all(fh,latticePatch.uData,count,MPI_SUNREALTYPE,&write_request);
    MPI_Wait(&write_request,MPI_STATUS_IGNORE);
    */
    /*
    MPI_File_write_at_all(fh,offset,latticePatch.uData,count,MPI_SUNREALTYPE,
            MPI_STATUS_IGNORE);
            */
    break;
                }
        default: {
    errorKill("No valid output style defined.\
            Choose between (c): one csv file per process, (b) one binary file");
    break;
               }
    return;
    }
}
```

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

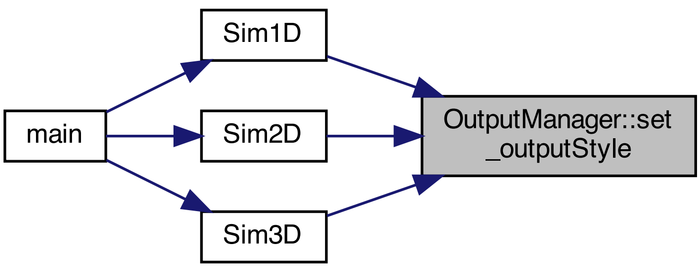
Definition at line 76 of file Outputters.cpp.

```
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

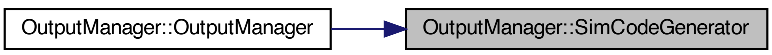
Generate the identifier number reverse from year to minute in the format yy-mm-dd_hh-MM-ss

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]
```

varible to safe 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:

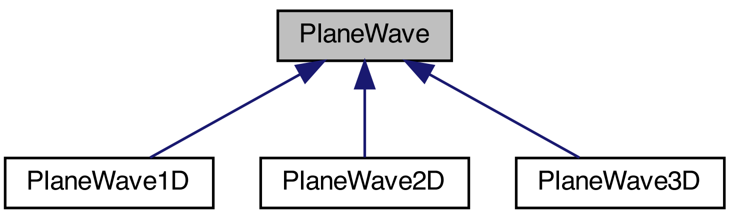
- src/Outputters.h
- src/Outputters.cpp

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 $\vec{E} = \vec{E}_0 \, \cos\left(\vec{k} \cdot \vec{x} - \vec{\phi}\right)$

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:

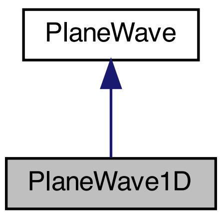
- src/SimulationFunctions.h

5.13 PlaneWave1D Class Reference

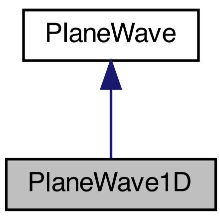
class for plane waves in 1D

```
#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.

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

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:

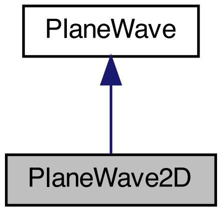
- src/ICSetters.h
- src/ICSetters.cpp

5.14 PlaneWave2D Class Reference

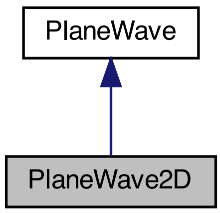
class for plane waves in 2D

```
#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.

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

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.

```
                                                              {
    const sunrealtype wavelength =
        sqrt(kx * kx + ky * ky + kz * kz); /* \f$ 1/\lambda \f$ */
    const sunrealtype kScalarX = (kx * x + ky * y + kz * z) * 2 *
                               numbers::pi; /* \f$ 2\pi \ \vec{k} \cdot \vec{x} \f$ */
    // Plane wave definition
    const array<sunrealtype, 3> E{{                             /* E-field vector */
                                 px * cos(kScalarX - phix),   /* \f$ E_x \f$ */
                                 py * cos(kScalarX - phiy),   /* \f$ E_y \f$ */
                                 pz * cos(kScalarX - phiz)}}; /* \f$ E_z \f$ */
    // Put E-field into space
    pTo6Space[0] += E[0];
    pTo6Space[1] += E[1];
    pTo6Space[2] += E[2];
    // and B-field
    pTo6Space[3] += (ky * E[2] - kz * E[1]) / wavelength;
    pTo6Space[4] += (kz * E[0] - kx * E[2]) / wavelength;
    pTo6Space[5] += (kx * E[1] - ky * E[0]) / wavelength;
}
```

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:

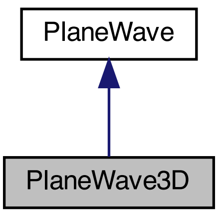
- src/ICSetters.h
- src/ICSetters.cpp

5.15 PlaneWave3D Class Reference

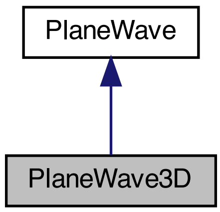
class for plane waves in 3D

`#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.

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

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.

```
                                                                            {
    const sunrealtype wavelength =
        sqrt(kx * kx + ky * ky + kz * kz); /* \f$ 1/\lambda \f$ */
    const sunrealtype kScalarX = (kx * x + ky * y + kz * z) * 2 *
                                 numbers::pi; /* \f$ 2\pi \ \vec{k} \cdot \vec{x} \f$ */
    // Plane wave definition
    const array<sunrealtype, 3> E{{/* E-field vector \f$ \vec{E}\f$*/
                                   px * cos(kScalarX - phix),    /* \f$ E_x \f$ */
                                   py * cos(kScalarX - phiy),    /* \f$ E_y \f$ */
                                   pz * cos(kScalarX - phiz)}}; /* \f$ E_z \f$ */
    // Put E-field into space
    pTo6Space[0] += E[0];
    pTo6Space[1] += E[1];
    pTo6Space[2] += E[2];
    // and B-field
    pTo6Space[3] += (ky * E[2] - kz * E[1]) / wavelength;
    pTo6Space[4] += (kz * E[0] - kx * E[2]) / wavelength;
    pTo6Space[5] += (kx * E[1] - ky * E[0]) / wavelength;
}
```

References PlaneWave::kx, PlaneWave::ky, PlaneWave::kz, PlaneWave::phix, PlaneWave::phiy, PlaneWave::phiz, PlaneWave::px, PlaneWave::py, and PlaneWave::pz.

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

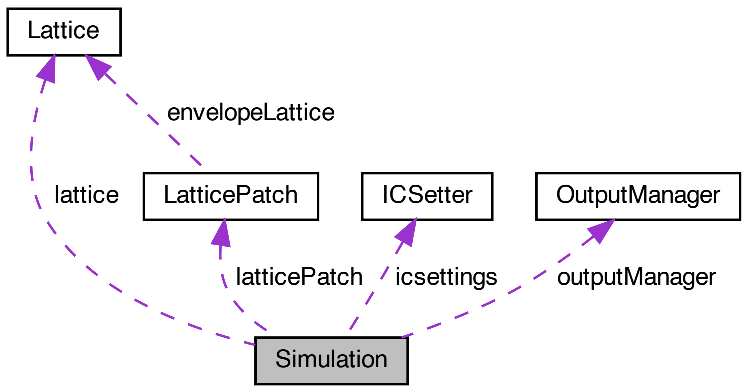
- src/ICSetters.h
- src/ICSetters.cpp

5.16 Simulation Class Reference

Simulation class to instantiate the whole walkthrough of a Simulation.

`#include <src/SimulationClass.h>`

Collaboration diagram for Simulation:



Public Member Functions

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

  constructor function for the creation of the cartesian communicator
- ∼Simulation ()

  destructor function freeing CVode memory and Sundials context
- MPI_Comm ∗ get_cart_comm ()

  Reference to the cartesian communicator of the lattice -> for debugging.
- void setDiscreteDimensionsOfLattice (const sunindextype _tot_nx, const sunindextype _tot_ny, const sunindextype _tot_nz)

  function to set discrete dimensions of the lattice
- void setPhysicalDimensionsOfLattice (const sunrealtype lx, const sunrealtype ly, const sunrealtype lz)

  function to set physical dimensions of the lattice
- void initializePatchwork (const int nx, const int ny, const int nz)

  function to initialize the Patchwork
- void initializeCVODEobject (const sunrealtype reltol, const sunrealtype abstol)

  function to initialize the CVODE object with all requirements
- void start ()

  function to start the simulation for time iteration
- void setInitialConditions ()

  functions to set the initial field configuration onto the lattice

- void addInitialConditions (const int xm, const int ym, const int zm=0)

    functions to add initial periodic field configurations
- void addPeriodicICLayerInX ()

    function to add a periodic IC Layer in one dimension
- void addPeriodicICLayerInXY ()

    function to add periodic IC Layers in two dimensions
- void advanceToTime (const sunrealtype &tEnd)

    function to advance solution in time with CVODE
- void outAllFieldData (const int &state)

    function to generate Output of the whole field at a given time
- void checkFlag (unsigned int flag) const

    function to check that a flag has been set and if not print an error
- void checkNoFlag (unsigned int flag) const

    function to check that if flag has not been set and if print an error

Data Fields

- ICSetter icsettings

    IC Setter object.
- OutputManager outputManager

    Output Manager object.
- void ∗ cvode_mem

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

Private Attributes

- Lattice lattice

    Lattice object.
- LatticePatch latticePatch

    LatticePatch object.
- sunrealtype t

    current time of the simulation
- unsigned char statusFlags

    char for checking simulation flags

5.16.1 Detailed Description

Simulation class to instantiate the whole walkthrough of a Simulation.

Definition at line 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

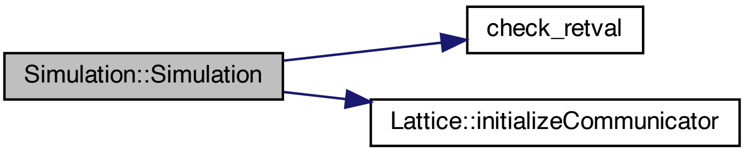
Along with the simulation object, create the cartesian communicator and SUNContext object

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 CVode memory and Sundials context

Free the CVode 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 & CvodeObjectSetUp) {
00037     // PrintFinalStats(cvode_mem); // TODO write this function as in cvodes
00038     // cvAdvDiff_bnd.c SUNDIALS_MARK_FUNCTION_END(lattice.profobj);
00039     CVodeFree(&cvode_mem);
00040     SUNContext_Free(&lattice.sunctx);
00041   }
00042 }
```

References cvode_mem, CvodeObjectSetUp, 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.

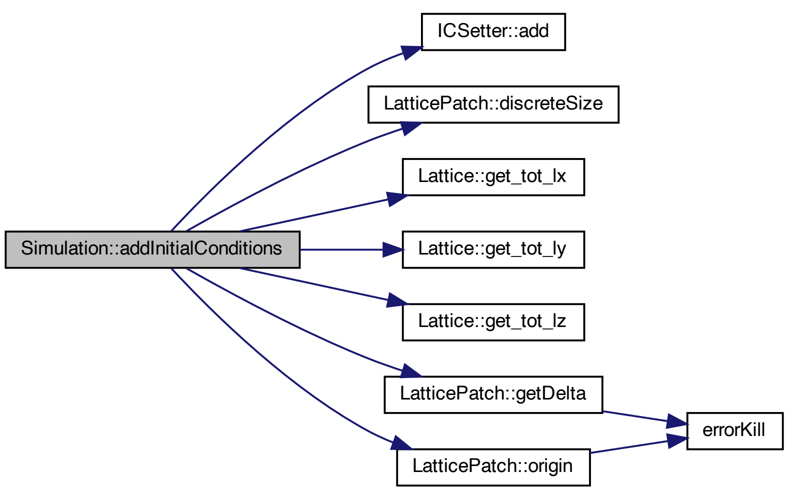
Definition at line 185 of file SimulationClass.cpp.

```
                                  {
  const sunrealtype dx = latticePatch.getDelta(1);
  const sunrealtype dy = latticePatch.getDelta(2);
  const sunrealtype dz = latticePatch.getDelta(3);
  const int nx = latticePatch.discreteSize(1);
  const int ny = latticePatch.discreteSize(2);
  // Correct for demanded displacement, rest as for setInitialConditions
  const sunrealtype x0 = latticePatch.origin(1) + xm*lattice.get_tot_lx();
  const sunrealtype y0 = latticePatch.origin(2) + ym*lattice.get_tot_ly();
  const sunrealtype z0 = latticePatch.origin(3) + zm*lattice.get_tot_lz();
  int px = 0, py = 0, pz = 0;
  for (int i = 0; i < latticePatch.discreteSize() * 6; i += 6) {
    px = (i / 6) % nx;
    py = ((i / 6) / nx) % ny;
    pz = ((i / 6) / nx) / ny;
    icsettings.add(static_cast<sunrealtype>(px) * dx + x0,
            static_cast<sunrealtype>(py) * dy + y0,
            static_cast<sunrealtype>(pz) * dz + z0, &latticePatch.uData[i]);
  }
  return;
}
```

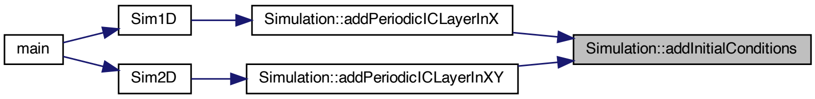
References ICSetter::add(), LatticePatch::discreteSize(), Lattice::get_tot_lx(), Lattice::get_tot_ly(), Lattice::get_tot_lz(), LatticePatch::getDelta(), icsettings, lattice, latticePatch, LatticePatch::origin(), and LatticePatch::uData.

Referenced by addPeriodicICLayerInX(), and addPeriodicICLayerInXY().

Here is the call graph for this function:



Here is the caller graph for this function:



5.16.3.2 addPeriodicICLayerInX()

```
void Simulation::addPeriodicICLayerInX ( )
```

function to add a periodic IC Layer in one dimension

Add initial conditions in one dimension.

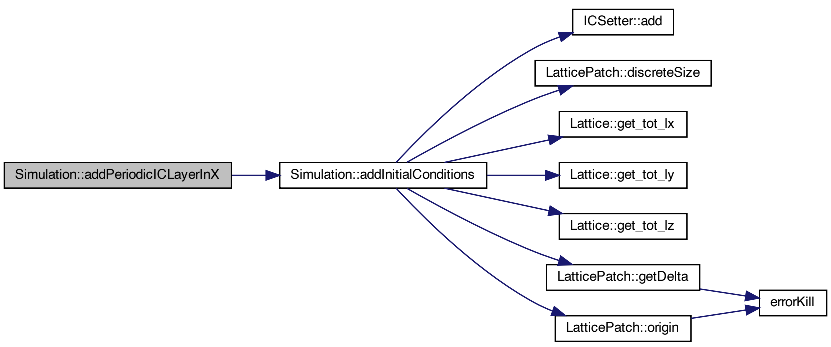
Definition at line 209 of file SimulationClass.cpp.

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

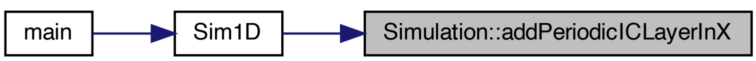
References addInitialConditions().

Referenced by Sim1D().

Here is the call graph for this function:



Here is the caller graph for this function:



5.16.3.3 addPeriodicICLayerInXY()

```
void Simulation::addPeriodicICLayerInXY ( )
```

function to add periodic IC Layers in two dimensions

Add initial conditions in two dimensions.

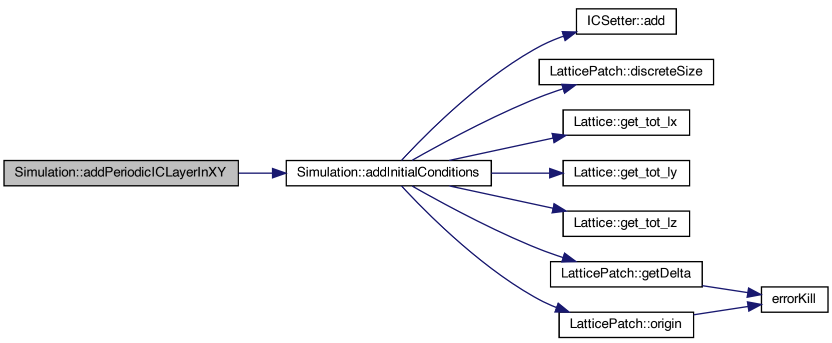
Definition at line 216 of file SimulationClass.cpp.

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

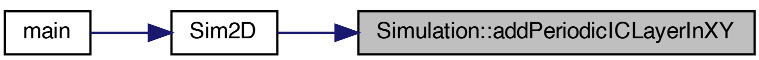
References addInitialConditions().

Referenced by Sim2D().

Here is the call graph for this function:



Here is the caller graph for this function:



5.16.3.4 advanceToTime()

```
void Simulation::advanceToTime (
            const sunrealtype & tEnd )
```

function to advance solution in time with CVODE

Advance the solution in time -> integrate the ODE over an interval t.

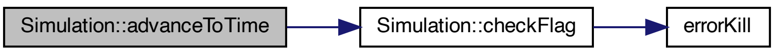
Definition at line 229 of file SimulationClass.cpp.

```
00229                                                     {
00230   checkFlag(SimulationStarted);
00231   int flag = 0;
00232   flag = 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 }
```

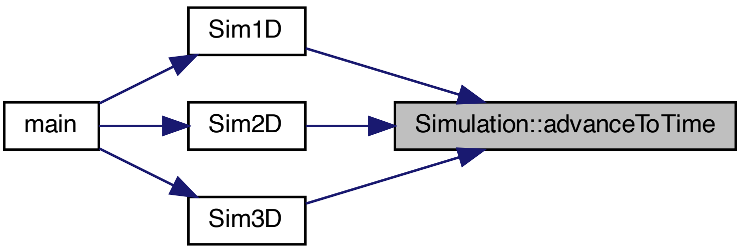
References checkFlag(), cvode_mem, latticePatch, SimulationStarted, t, and LatticePatch::u.

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.

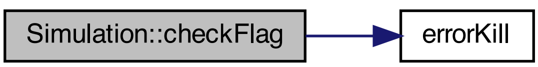
Definition at line 247 of file SimulationClass.cpp.

```
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 }
```

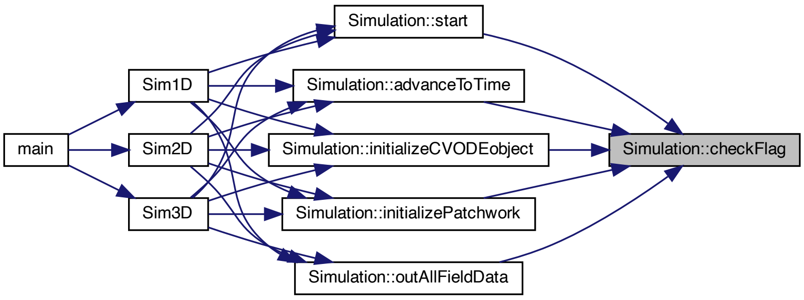
References CvodeObjectSetUp, errorKill(), LatticeDiscreteSetUp, LatticePatchworkSetUp, LatticePhysicalSetUp, SimulationStarted, and statusFlags.

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.

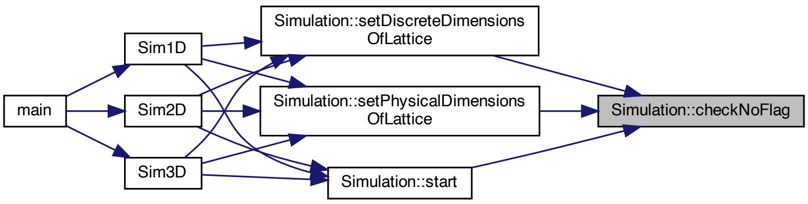
```
00277                                                     {
00278   if ((statusFlags & flag)) {
00279     string errorMessage;
00280     switch (flag) {
00281     case LatticeDiscreteSetUp:
00282       errorMessage =
00283           "The discrete size of the Simulation has already been set up";
00284       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.

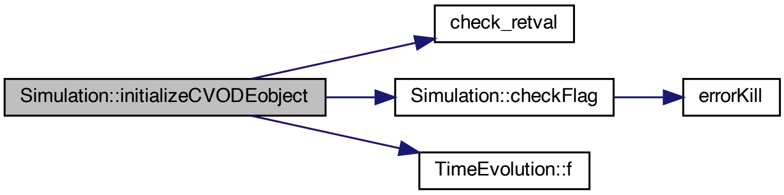
```
00075                                          {
00076   checkFlag(SimulationStarted);
```

```

  // CVode settings return value
  int retval = 0;

  // Set the profiler
  retval = SUNContext_GetProfiler(lattice.sunctx, &lattice.profobj);
  if (check_retval(&retval, "SUNContext_GetProfiler", 1, lattice.my_prc))
    MPI_Abort(lattice.comm, 1);
  // if (flag != CV_SUCCESS) { printf("SUNContext_GetProfiler failed,
  // flag=%d.\n", flag);
  //     MPI_Abort(lattice.comm, 1); }

  // SUNDIALS_MARK_FUNCTION_BEGIN(profobj);

  // Create CVODE object -- returns a pointer to the cvode memory structure
  // with Adams method (Adams-Moulton formula) solver chosen for non-stiff ODE
  cvode_mem = CVodeCreate(CV_ADAMS, lattice.sunctx);

  // Specify user data and attach it to the main cvode memory block
  retval = CVodeSetUserData(
      cvode_mem,
      &latticePatch); // patch contains the user data as used in CVRhsFn
  if (check_retval(&retval, "CVodeSetUserData", 1, lattice.my_prc))
    MPI_Abort(lattice.comm, 1);
  // if (flag != CV_SUCCESS) { printf("CVodeSetUserData failed, flag=%d.\n",
  // flag);
  //     MPI_Abort(lattice.comm, 1); }

  // Initialize CVODE solver -> can only be called after start of simulation to
  // have data ready Provide required problem and solution specifications,
  // allocate internal memory, and initialize cvode
  retval = CVodeInit(cvode_mem, TimeEvolution::f, 0,
                     latticePatch.u); // allocate memory, CVRhsFn f, t_i=0, u
                                      // contains the initial values
  if (check_retval(&retval, "CVodeInit", 1, lattice.my_prc))
    MPI_Abort(lattice.comm, 1);
  // if (flag != CV_SUCCESS) { printf("CVodeInit failed, flag=%d.\n", flag);
  //     MPI_Abort(lattice.comm, 1); }

  // Create fixed point nonlinear solver object (suitable for non-stiff ODE) and
  // attach it to CVode
  SUNNonlinearSolver NLS =
      SUNNonlinSol_FixedPoint(latticePatch.u, 0, lattice.sunctx);
  retval = CVodeSetNonlinearSolver(cvode_mem, NLS);
  if (check_retval(&retval, "CVodeSetNonlinearSolver", 1, lattice.my_prc))
    MPI_Abort(lattice.comm, 1);
  // if (flag != CV_SUCCESS) {printf("CVodeSetNonlinearSolver failed,
  // flag=%d.\n", flag);
  //     MPI_Abort(lattice.comm, 1); }

  // Specify the maximum number of steps to be taken by the solver in its
  // attempt to reach the next output time
  retval = CVodeSetMaxNumSteps(cvode_mem, 10000);
  if (check_retval(&retval, "CVodeSetMaxNumSteps", 1, lattice.my_prc))
    MPI_Abort(lattice.comm, 1);
  // if (flag != CV_SUCCESS) { printf("CVodeSetMaxNumSteps failed, flag=%d.\n",
  // flag);
  //     MPI_Abort(lattice.comm, 1); }

  // Specify integration tolerances -- a scalar relative tolerance and scalar
  // absolute tolerance
  retval = CVodeSStolerances(cvode_mem, reltol, abstol);
  if (check_retval(&retval, "CVodeSStolerances", 1, lattice.my_prc))
    MPI_Abort(lattice.comm, 1);
  // if (flag != CV_SUCCESS) { printf("CVodeSStolerances failed, flag=%d.\n",
  // flag);
  //     MPI_Abort(lattice.comm, 1); }

  statusFlags |= CvodeObjectSetUp;
}
```

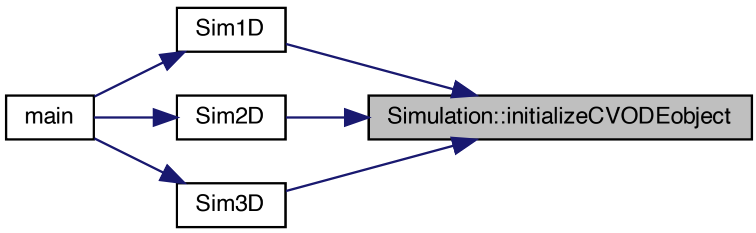
References check_retval(), checkFlag(), Lattice::comm, cvode_mem, CvodeObjectSetUp, TimeEvolution::f(), lattice, latticePatch, Lattice::my_prc, Lattice::profobj, SimulationStarted, statusFlags, Lattice::sunctx, and LatticePatch::u.

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.

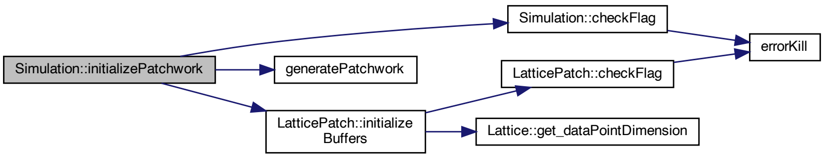
Definition at line 61 of file SimulationClass.cpp.

```
                        {
  checkFlag(LatticeDiscreteSetUp);
  checkFlag(LatticePhysicalSetUp);

  // Generate the patchwork
  generatePatchwork(lattice, latticePatch, nx, ny, nz);
  latticePatch.initializeBuffers();

  statusFlags |= LatticePatchworkSetUp;
}
```

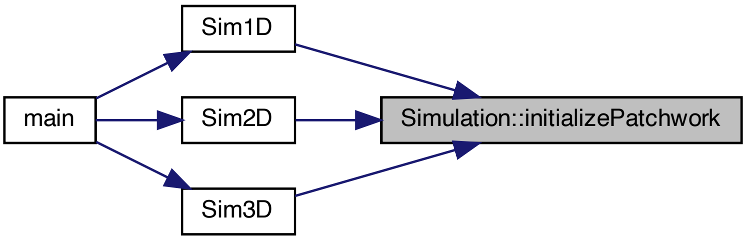
References checkFlag(), generatePatchwork(), LatticePatch::initializeBuffers(), lattice, LatticeDiscreteSetUp, latticePatch, LatticePatchworkSetUp, LatticePhysicalSetUp, and statusFlags.

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.

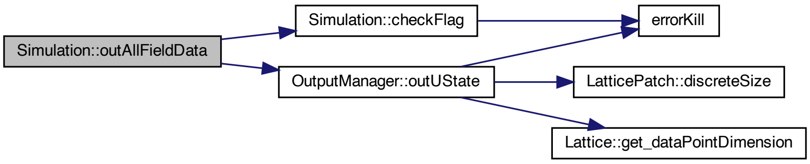
Definition at line 241 of file SimulationClass.cpp.

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

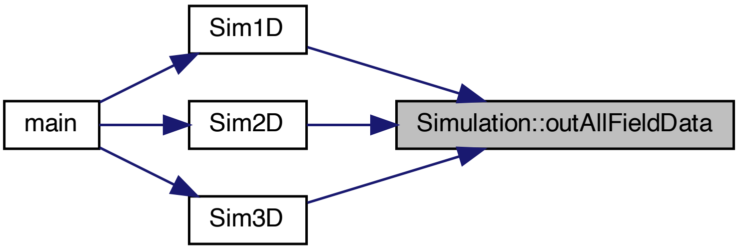
References checkFlag(), lattice, latticePatch, outputManager, OutputManager::outUState(), and SimulationStarted.

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.

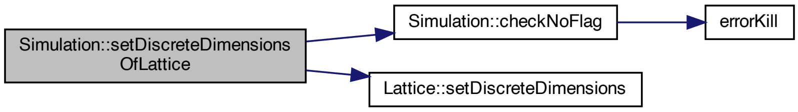
Definition at line 45 of file SimulationClass.cpp.

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

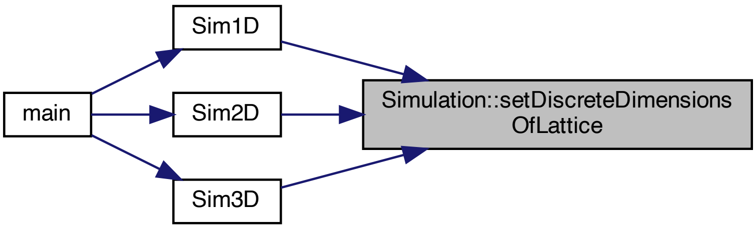
References checkNoFlag(), lattice, LatticeDiscreteSetUp, LatticePatchworkSetUp, Lattice::setDiscreteDimensions(), and statusFlags.

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

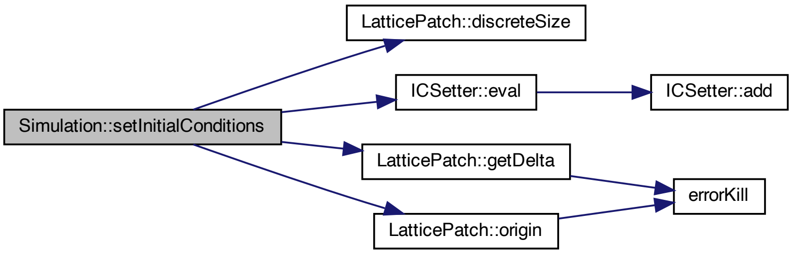
Definition at line 161 of file SimulationClass.cpp.

```
                                              {
  const sunrealtype dx = latticePatch.getDelta(1);
  const sunrealtype dy = latticePatch.getDelta(2);
  const sunrealtype dz = latticePatch.getDelta(3);
  const int nx = latticePatch.discreteSize(1);
  const int ny = latticePatch.discreteSize(2);
  const sunrealtype x0 = latticePatch.origin(1);
  const sunrealtype y0 = latticePatch.origin(2);
  const sunrealtype z0 = latticePatch.origin(3);
  int px = 0, py = 0, pz = 0;
  // space coordinates
  for (int i = 0; i < latticePatch.discreteSize() * 6; i += 6) {
    px = (i / 6) % nx;
    py = ((i / 6) / nx) % ny;
    pz = ((i / 6) / nx) / ny;
    // Call the `eval` function to fill the lattice points with the field data
    icsettings.eval(static_cast<sunrealtype>(px) * dx + x0,
            static_cast<sunrealtype>(py) * dy + y0,
            static_cast<sunrealtype>(pz) * dz + z0, &latticePatch.uData[i]);
  }
  return;
}
```

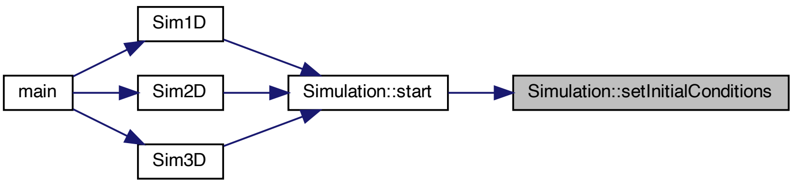
References LatticePatch::discreteSize(), ICSetter::eval(), LatticePatch::getDelta(), icsettings, latticePatch, LatticePatch::origin(), and LatticePatch::uData.

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 lenghts in micro meters.

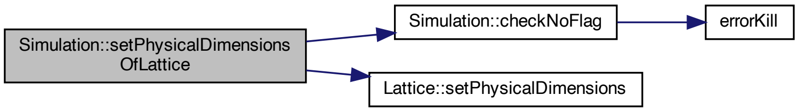
Definition at line 53 of file SimulationClass.cpp.

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

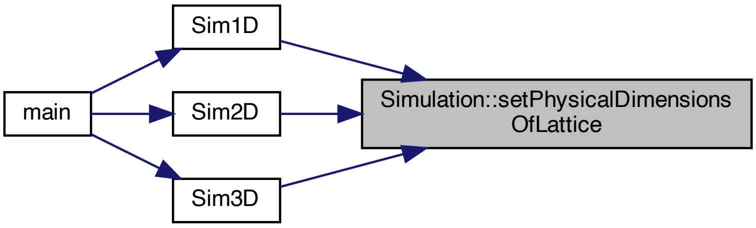
References checkNoFlag(), lattice, LatticePatchworkSetUp, LatticePhysicalSetUp, Lattice::setPhysicalDimensions(), and statusFlags.

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.

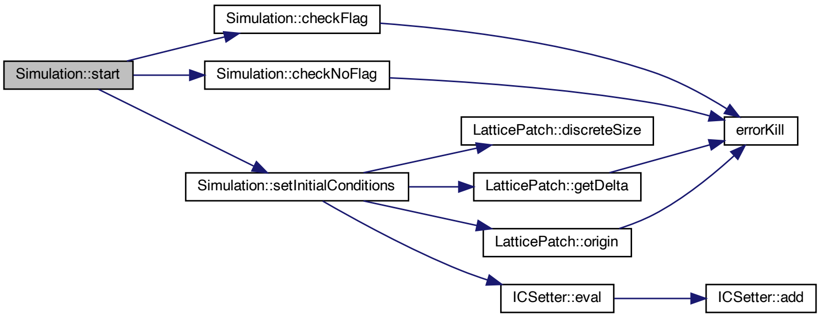
Definition at line 149 of file SimulationClass.cpp.

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

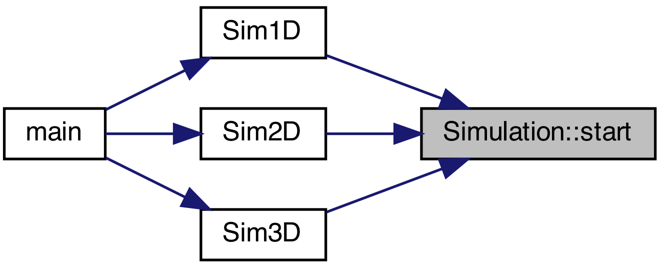
References checkFlag(), checkNoFlag(), CvodeObjectSetUp, LatticeDiscreteSetUp, LatticePatchworkSetUp, LatticePhysicalSetUp, setInitialConditions(), SimulationStarted, and statusFlags.

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)

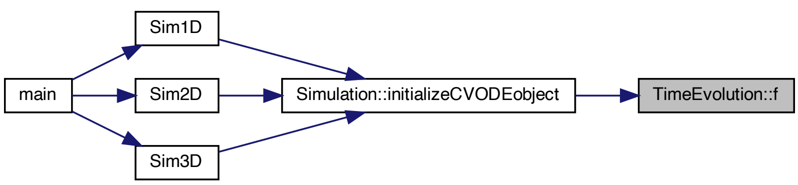
Definition at line 13 of file TimeEvolutionFunctions.cpp.

```
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

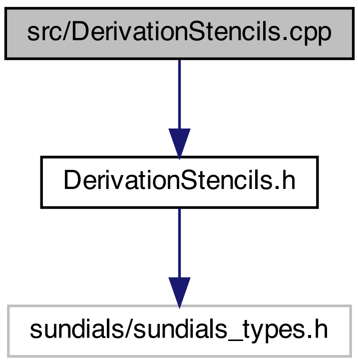
6.1 README.md File Reference

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.

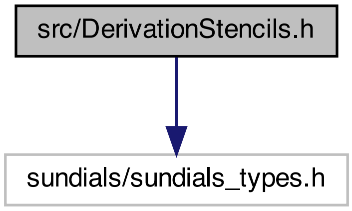
```
///////////////////////////////////////////////////
/// @file DerivationStencils.cpp
/// @brief Empty. All definitions in the header.
//////////////////////////////////////////////////
#include "DerivationStencils.h"
```

6.4 src/DerivationStencils.h File Reference

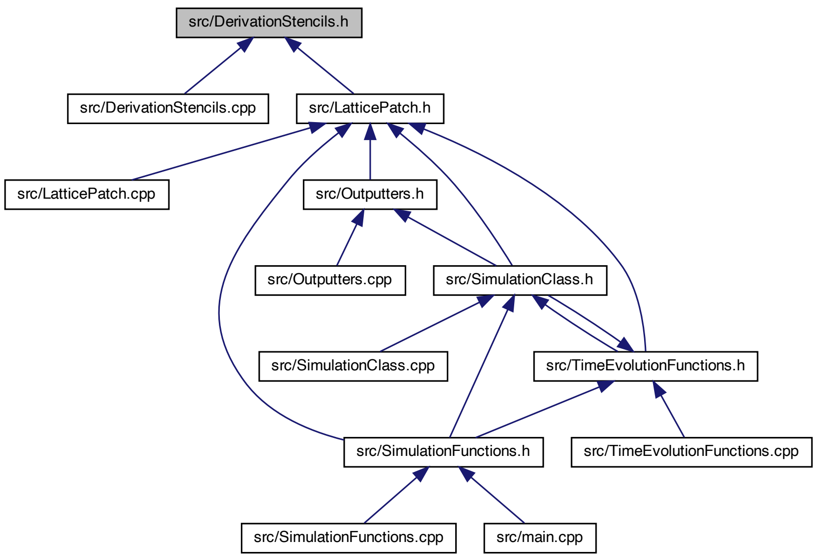
Definition of derivation stencils from order 1 to 13.

`#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]

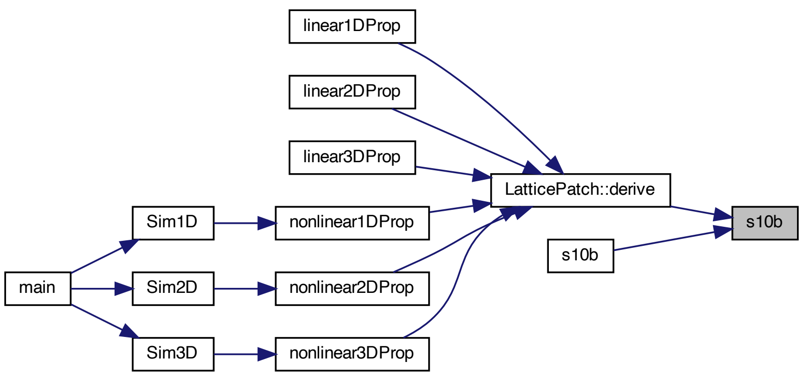
```
sunrealtype s10b (
            const sunrealtype * udata,
            int nx )  [inline]
```

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]

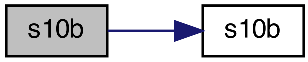
```
sunrealtype s10b (
             sunrealtype * udata ) [inline]
```

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]

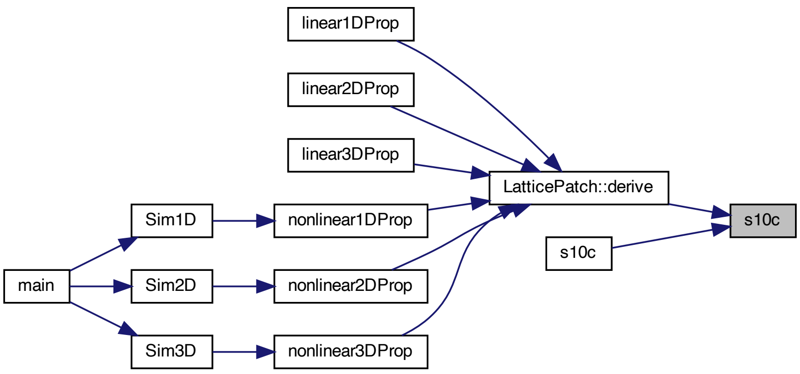
```
sunrealtype s10c (
             const sunrealtype * udata,
             int nx ) [inline]
```

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]

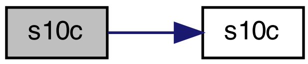
```
sunrealtype s10c (
            sunrealtype * udata )  [inline]
```

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]

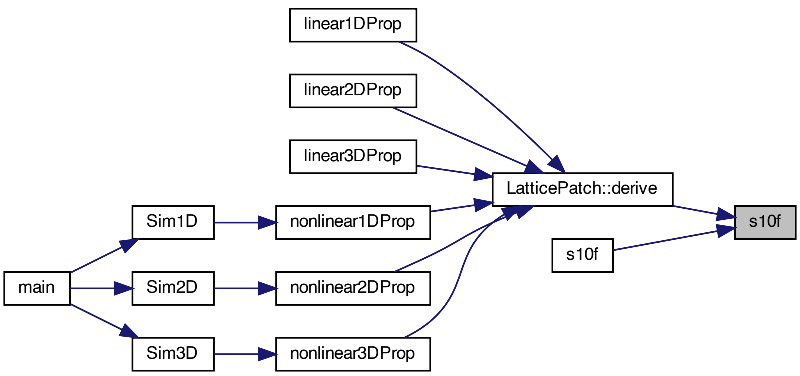
```
sunrealtype s10f (
            const sunrealtype * udata,
            int nx )  [inline]
```

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]

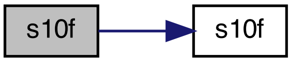
```
sunrealtype s10f (
            sunrealtype * udata ) [inline]
```

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]

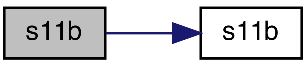
```
sunrealtype s11b (
            sunrealtype * udata ) [inline]
```

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]

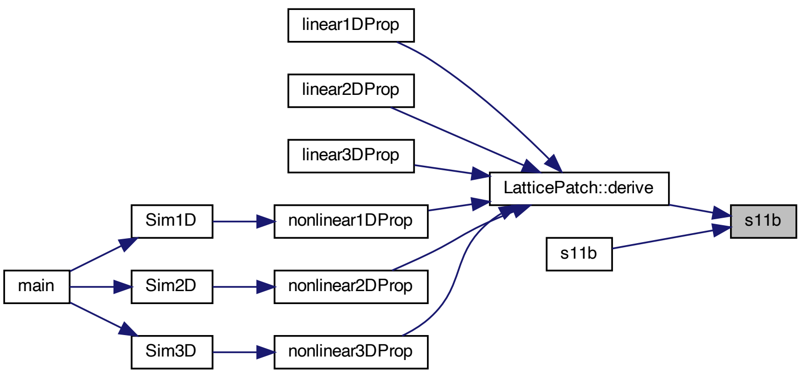
```
sunrealtype s11b (
            sunrealtype * udata,
            int nx ) [inline]
```

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]

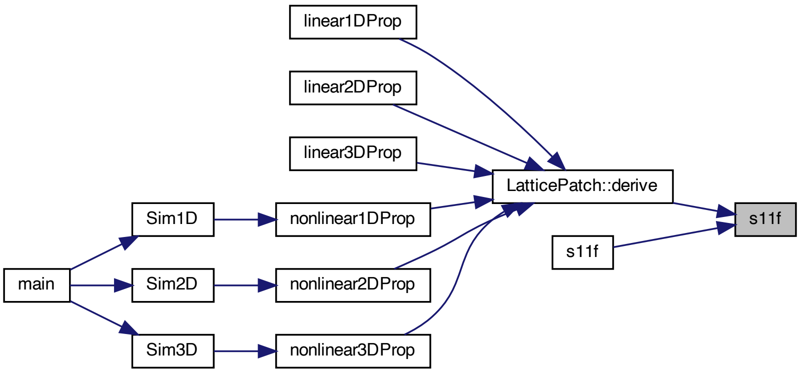
```
sunrealtype s11f (
            const sunrealtype * udata,
            int nx ) [inline]
```

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]

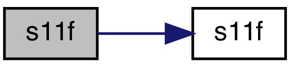
```
sunrealtype s11f (
            sunrealtype * udata )  [inline]
```

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]
```

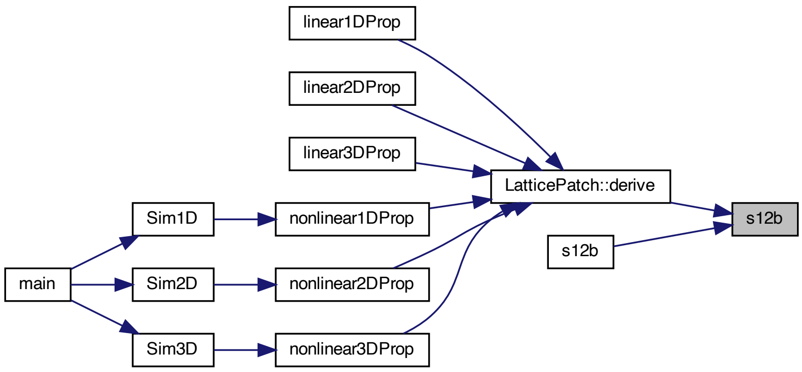
Definition at line 185 of file DerivationStencils.h.

```
00185                                                         {
```

```
  return -1.0 / 3960.0 * udata[-5 * nx] + 1.0 / 264.0 * udata[-4 * nx] -
         1.0 / 36.0 * udata[-3 * nx] + 5.0 / 36.0 * udata[-2 * nx] -
         5.0 / 8.0 * udata[-1 * nx] - 13.0 / 42.0 * udata[0] +
         7.0 / 6.0 * udata[1 * nx] - 1.0 / 2.0 * udata[2 * nx] +
         5.0 / 24.0 * udata[3 * nx] - 5.0 / 72.0 * udata[4 * nx] +
         1.0 / 60.0 * udata[5 * nx] - 1.0 / 396.0 * udata[6 * nx] +
         1.0 / 5544.0 * udata[7 * nx];
}
```

Referenced by LatticePatch::derive(), and s12b().

Here is the caller graph for this function:



6.4.2.12 s12b() [2/2]

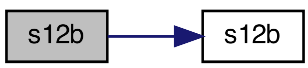
```
sunrealtype s12b (
            sunrealtype * udata )  [inline]
```

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]

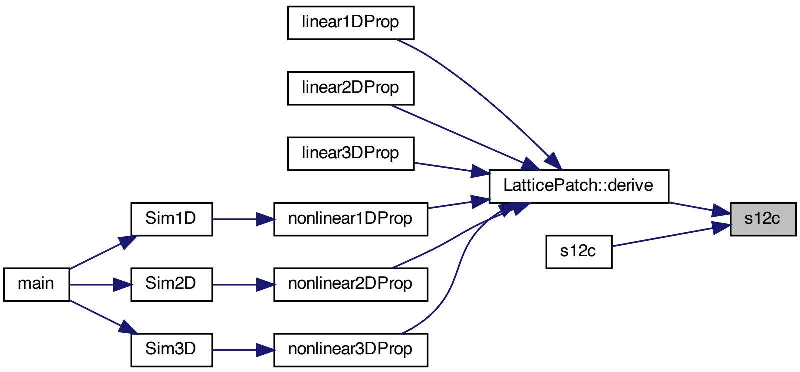
```
sunrealtype s12c (
            const sunrealtype * udata,
            int nx ) [inline]
```

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]

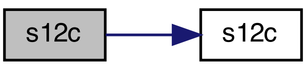
```
sunrealtype s12c (
            sunrealtype * udata ) [inline]
```

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]

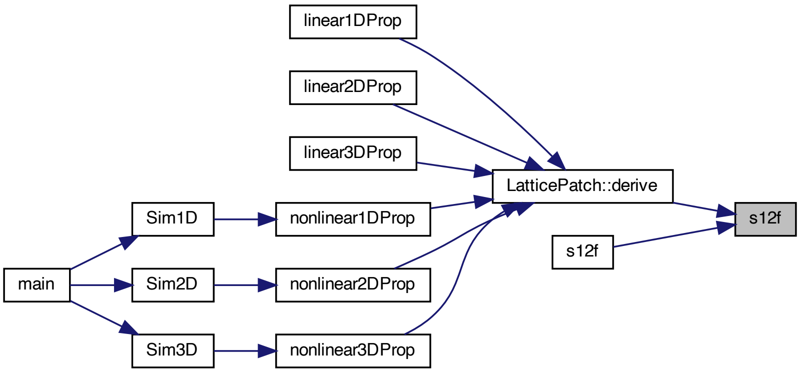
```
sunrealtype s12f (
            const sunrealtype * udata,
            int nx ) [inline]
```

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]

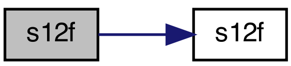
```
sunrealtype s12f (
            sunrealtype * udata ) [inline]
```

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]

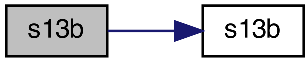
```
sunrealtype s13b (
            sunrealtype * udata )  [inline]
```

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]

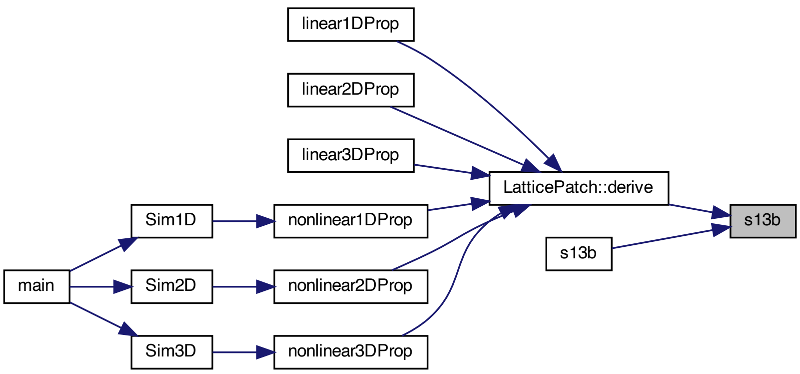
```
sunrealtype s13b (
            sunrealtype * udata,
            int nx )  [inline]
```

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]

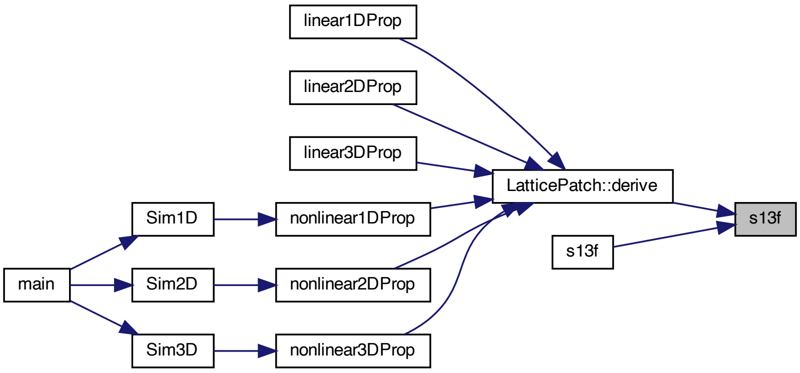
```
sunrealtype s13f (
            const sunrealtype * udata,
            int nx ) [inline]
```

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]

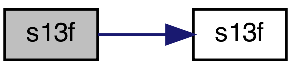
```
sunrealtype s13f (
            sunrealtype * udata ) [inline]
```

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]

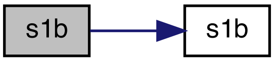
```
sunrealtype s1b (
            sunrealtype * udata )  [inline]
```

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]

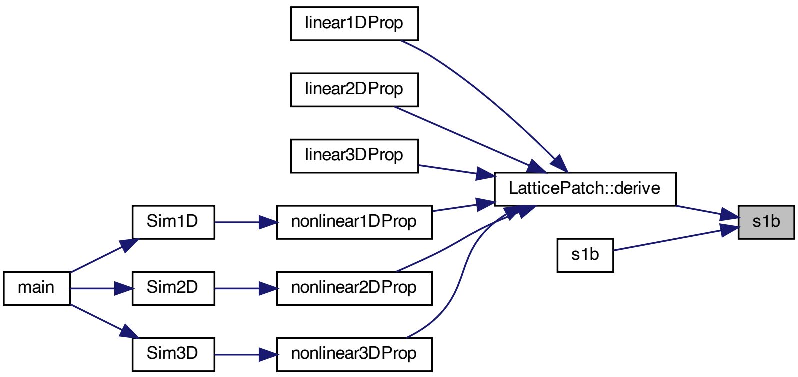
```
sunrealtype s1b (
            sunrealtype * udata,
            int nx )  [inline]
```

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]

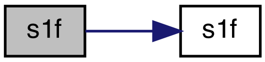
```
sunrealtype s1f (
            sunrealtype * udata )  [inline]
```

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]

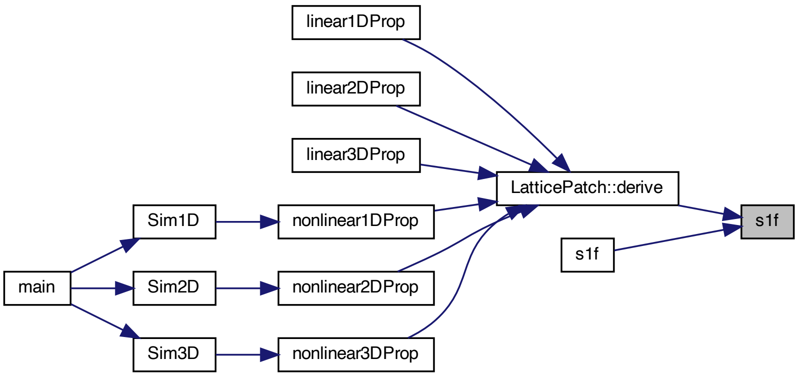
```
sunrealtype s1f (
            sunrealtype * udata,
            int nx )  [inline]
```

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]

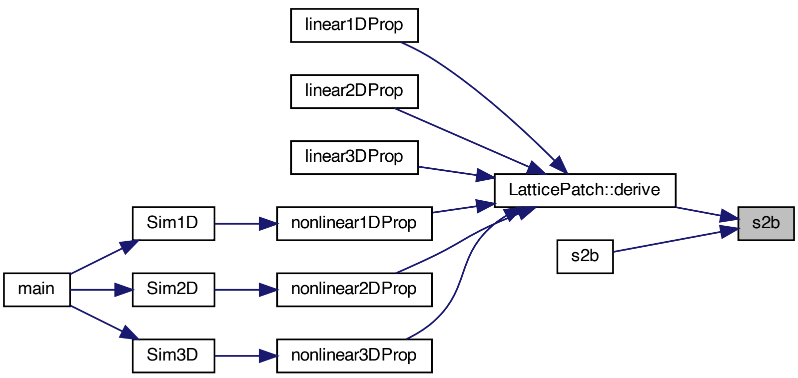
```
sunrealtype s2b (
            const sunrealtype * udata,
            int nx ) [inline]
```

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]

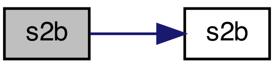
```
sunrealtype s2b (
            sunrealtype * udata ) [inline]
```

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]

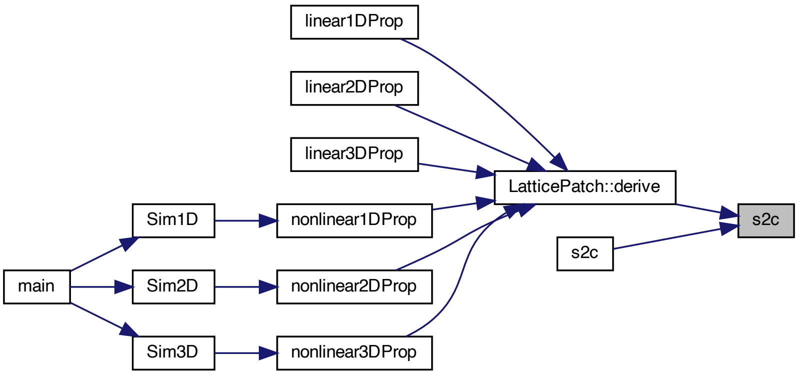
```
sunrealtype s2c (
            const sunrealtype * udata,
            int nx ) [inline]
```

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]

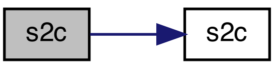
```
sunrealtype s2c (
            sunrealtype * udata ) [inline]
```

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]

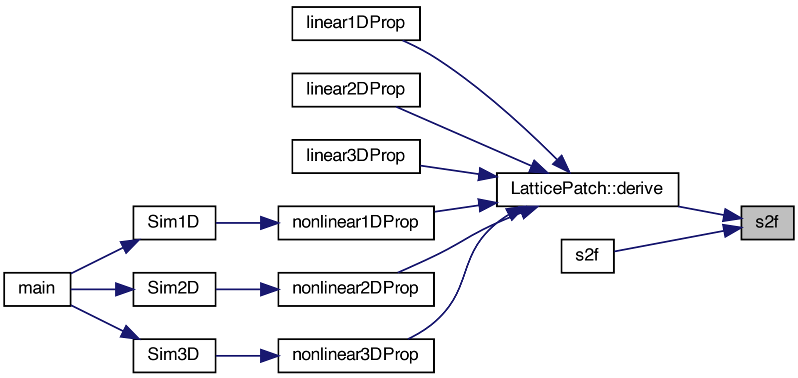
```
sunrealtype s2f (
            const sunrealtype * udata,
            int nx ) [inline]
```

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]

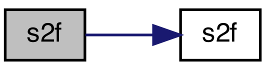
```
sunrealtype s2f (
            sunrealtype * udata ) [inline]
```

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]

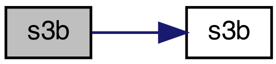
```
sunrealtype s3b (
              sunrealtype * udata )  [inline]
```

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]

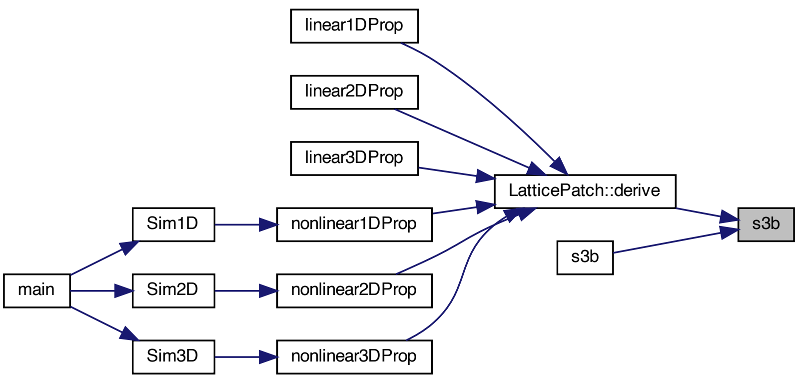
```
sunrealtype s3b (
              sunrealtype * udata,
              int nx )  [inline]
```

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]

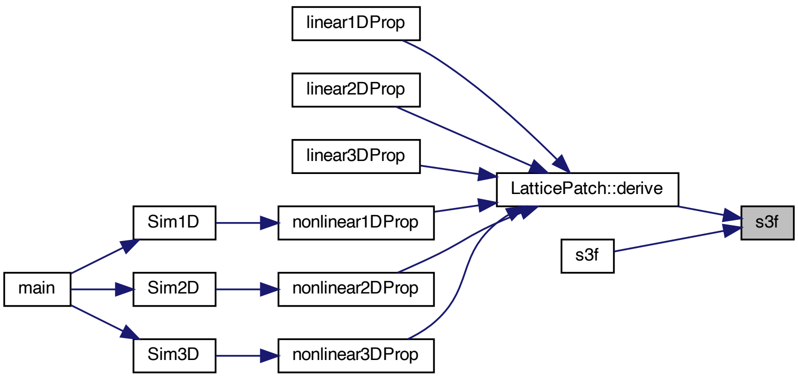
```
sunrealtype s3f (
            const sunrealtype * udata,
            int nx ) [inline]
```

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]

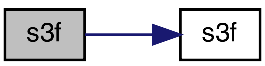
```
sunrealtype s3f (
            sunrealtype * udata ) [inline]
```

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]

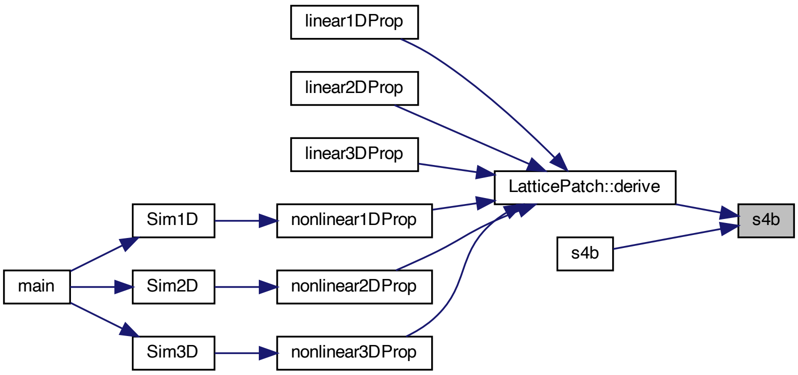
```
sunrealtype s4b (
            const sunrealtype * udata,
            int nx )  [inline]
```

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]

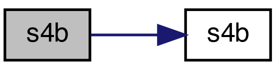
```
sunrealtype s4b (
            sunrealtype * udata )  [inline]
```

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]

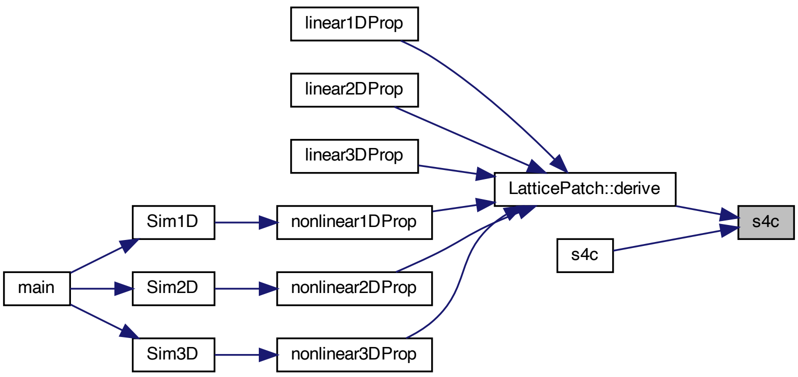
```
sunrealtype s4c (
            const sunrealtype * udata,
            int nx ) [inline]
```

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]

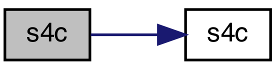
```
sunrealtype s4c (
            sunrealtype * udata ) [inline]
```

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]

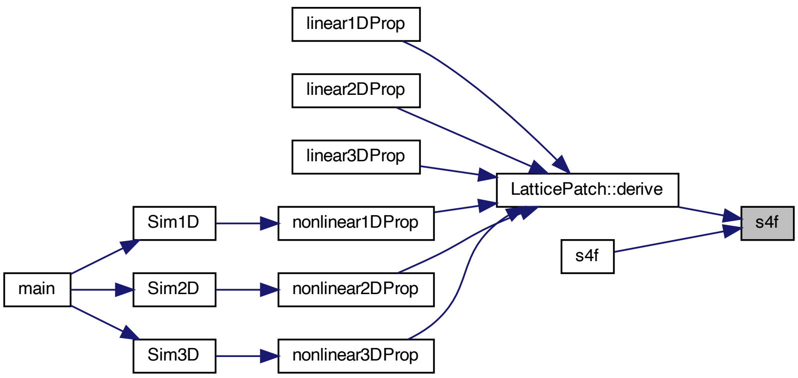
```
sunrealtype s4f (
            const sunrealtype * udata,
            int nx ) [inline]
```

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]

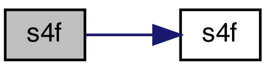
```
sunrealtype s4f (
            sunrealtype * udata ) [inline]
```

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]

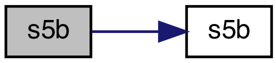
```
sunrealtype s5b (
            sunrealtype * udata )  [inline]
```

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]

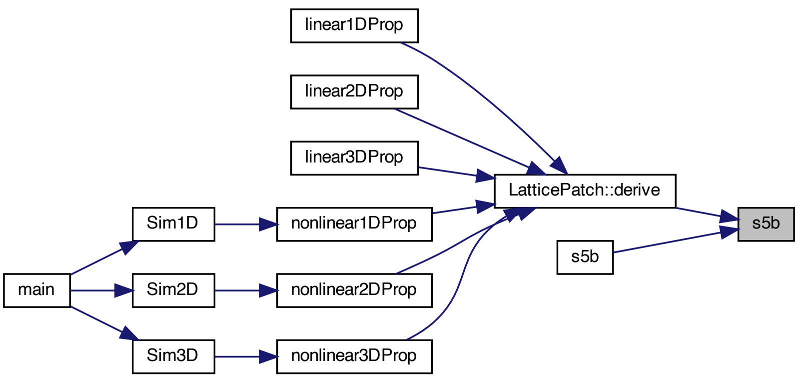
```
sunrealtype s5b (
            sunrealtype * udata,
            int nx )  [inline]
```

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]

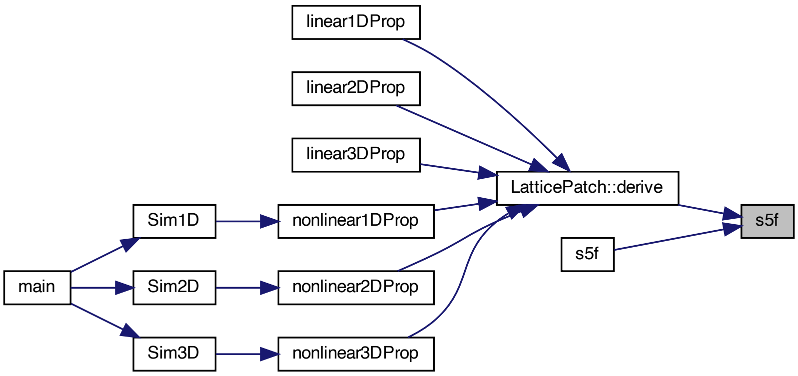
```
sunrealtype s5f (
            const sunrealtype * udata,
            int nx ) [inline]
```

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]

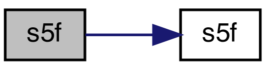
```
sunrealtype s5f (
            sunrealtype * udata ) [inline]
```

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]

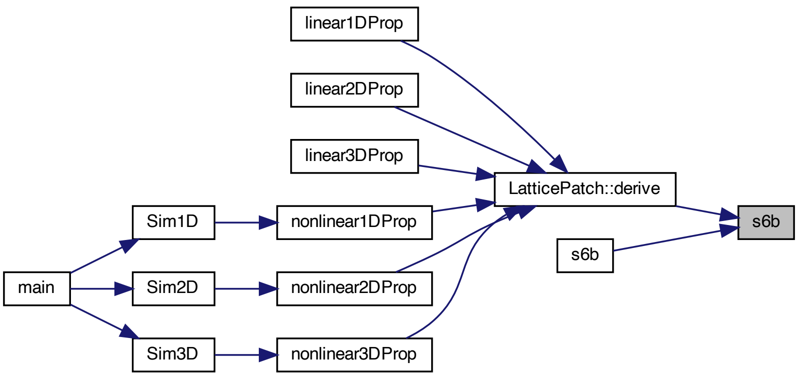
```
sunrealtype s6b (
            const sunrealtype * udata,
            int nx )  [inline]
```

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]

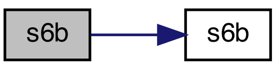
```
sunrealtype s6b (
            sunrealtype * udata )  [inline]
```

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]

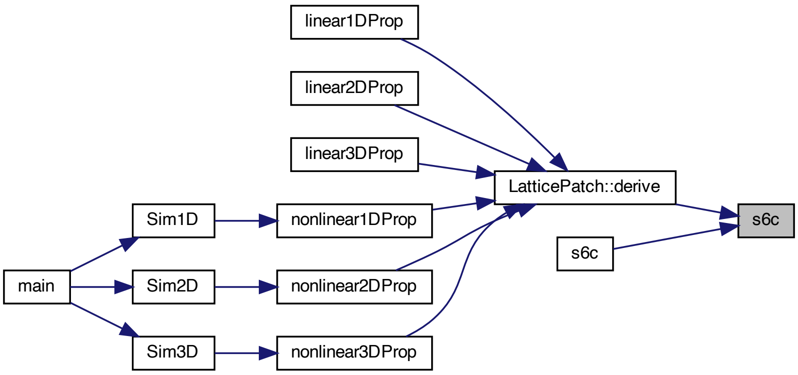
```
sunrealtype s6c (
            const sunrealtype * udata,
            int nx ) [inline]
```

Definition at line 72 of file DerivationStencils.h.

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

Referenced by LatticePatch::derive(), and s6c().

Here is the caller graph for this function:



6.4.2.48 s6c() [2/2]

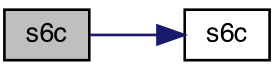
```
sunrealtype s6c (
            sunrealtype * udata ) [inline]
```

Definition at line 233 of file DerivationStencils.h.

```
{ return s6c(udata, 6); }
```

References s6c().

Here is the call graph for this function:



6.4.2.49 s6f() [1/2]

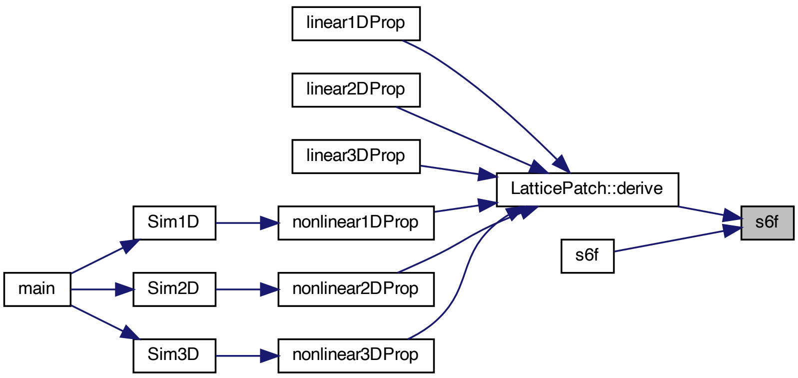
```
sunrealtype s6f (
            const sunrealtype * udata,
            int nx )  [inline]
```

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]

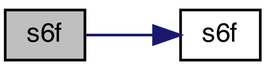
```
sunrealtype s6f (
            sunrealtype * udata )  [inline]
```

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]

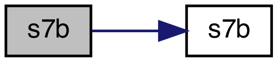
```
sunrealtype s7b (
            sunrealtype * udata ) [inline]
```

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]

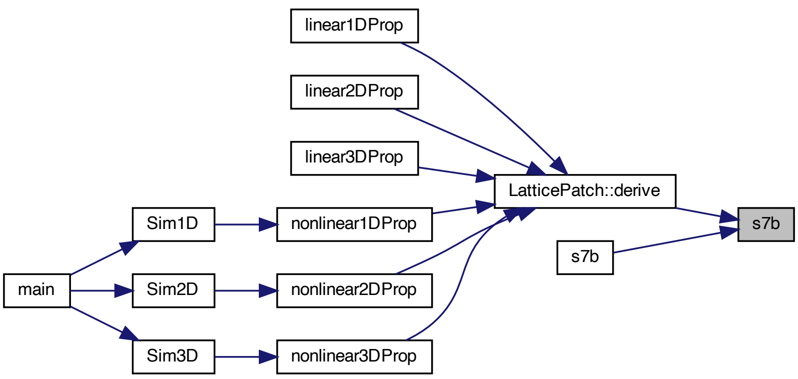
```
sunrealtype s7b (
            sunrealtype * udata,
            int nx ) [inline]
```

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]

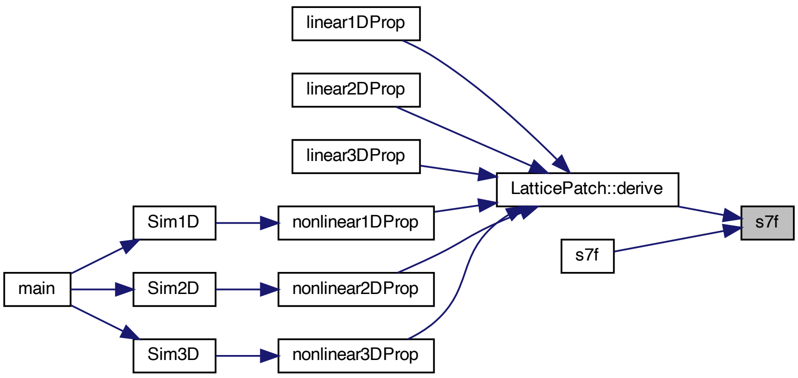
```
sunrealtype s7f (
            const sunrealtype * udata,
            int nx ) [inline]
```

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]

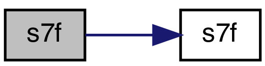
```
sunrealtype s7f (
            sunrealtype * udata ) [inline]
```

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]

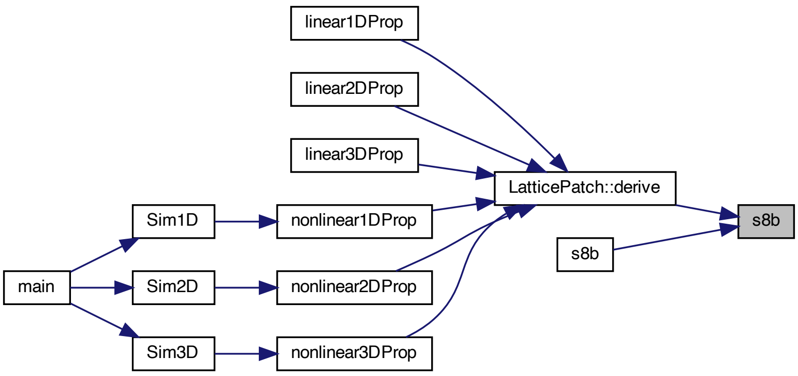
```
sunrealtype s8b (
            const sunrealtype * udata,
            int nx ) [inline]
```

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]

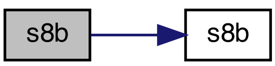
```
sunrealtype s8b (
            sunrealtype * udata ) [inline]
```

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]

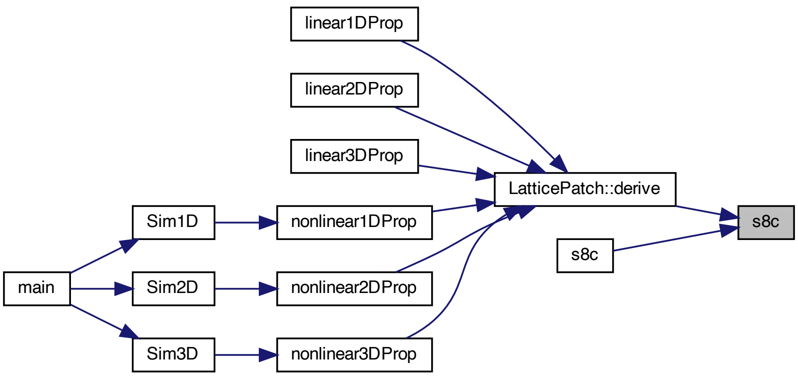
```
sunrealtype s8c (
              const sunrealtype * udata,
              int nx )  [inline]
```

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]

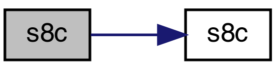
```
sunrealtype s8c (
              sunrealtype * udata )  [inline]
```

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]

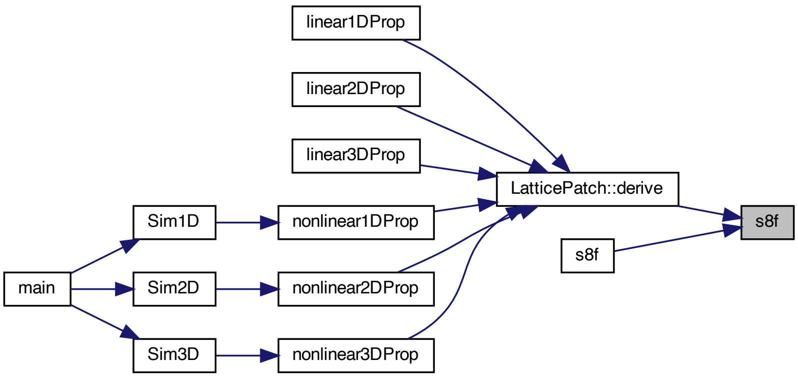
```
sunrealtype s8f (
            const sunrealtype * udata,
            int nx ) [inline]
```

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]

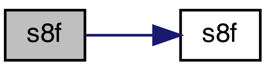
```
sunrealtype s8f (
            sunrealtype * udata ) [inline]
```

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]

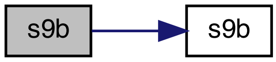
```
sunrealtype s9b (
            sunrealtype * udata ) [inline]
```

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]

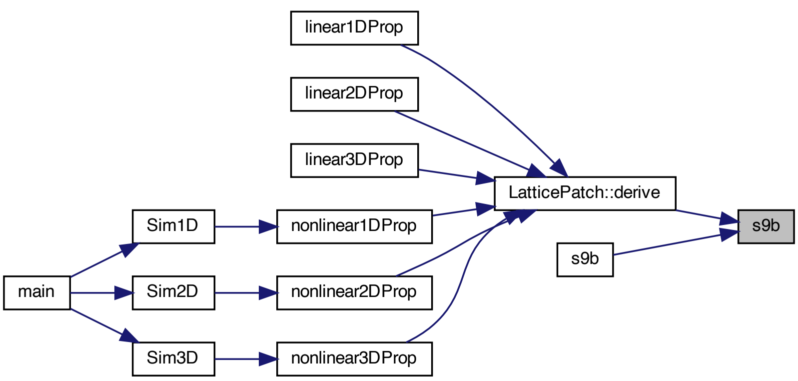
```
sunrealtype s9b (
            sunrealtype * udata,
            int nx ) [inline]
```

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]

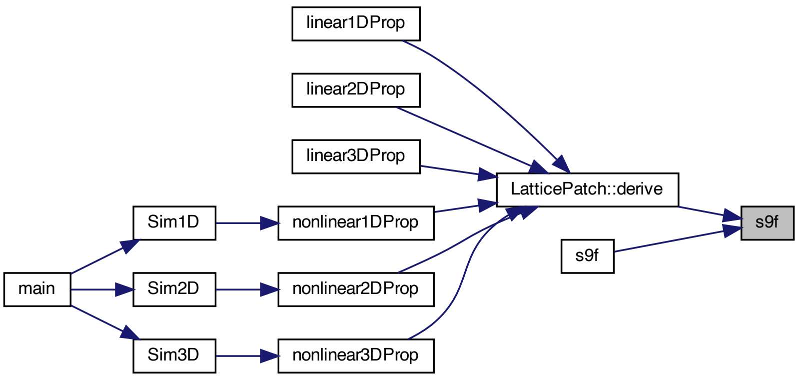
```
sunrealtype s9f (
            const sunrealtype * udata,
            int nx )  [inline]
```

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]

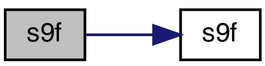
```
sunrealtype s9f (
            sunrealtype * udata )  [inline]
```

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.

```
///////////////////////////////////////////////////////////////////
/// @file DerivationStencils.h
/// @brief Definition of derivation stencils from order 1 to 13
///////////////////////////////////////////////////////////////////

#pragma once

#include <sundials/sundials_types.h> /* definition of type sunrealtype */

/////////////////////////////////////////////////////////
// Stencils with variable nx -- data point dimension //
/////////////////////////////////////////////////////////

// Downwind (forward) dfferentiating
inline sunrealtype s1f(sunrealtype *udata, int nx) {
  return -1.0 / 1.0 * udata[-1 * nx] + udata[0];
}
// Upwind (backward) differentiating
inline sunrealtype s1b(sunrealtype *udata, int nx) {
  return -1.0 / 1.0 * udata[0] + udata[1 * nx];
}

inline sunrealtype s2f(const sunrealtype *udata, int nx) {
  return 1.0 / 2.0 * udata[-2 * nx] - 2.0 / 1.0 * udata[-1 * nx] +
         3.0 / 2.0 * udata[0];
}
inline sunrealtype s2c(const sunrealtype *udata, int nx) {
  return -1.0 / 2.0 * udata[-1 * nx] + 0 + 1.0 / 2.0 * udata[1 * nx];
}
inline sunrealtype s2b(const sunrealtype *udata, int nx) {
  return -3.0 / 2.0 * udata[0] + 2.0 / 1.0 * udata[1 * nx] -
         1.0 / 2.0 * udata[2 * nx];
}
inline sunrealtype s3f(const sunrealtype *udata, int nx) {
  return 1.0 / 6.0 * udata[-2 * nx] - 1.0 / 1.0 * udata[-1 * nx] +
         1.0 / 2.0 * udata[0] + 1.0 / 3.0 * udata[1 * nx];
}
inline sunrealtype s3b(sunrealtype *udata, int nx) {
  return -1.0 / 3.0 * udata[-1 * nx] - 1.0 / 2.0 * udata[0] + udata[1 * nx] -
         1.0 / 6.0 * udata[2 * nx];
}
inline sunrealtype s4f(const sunrealtype *udata, int nx) {
  return -1.0 / 12.0 * udata[-3 * nx] + 1.0 / 2.0 * udata[-2 * nx] -
         3.0 / 2.0 * udata[-1 * nx] + 5.0 / 6.0 * udata[0] +
         1.0 / 4.0 * udata[1 * nx];
}
inline sunrealtype s4c(const sunrealtype *udata, int nx) {
  return 1.0 / 12.0 * udata[-2 * nx] - 2.0 / 3.0 * udata[-1 * nx] + 0 +
         2.0 / 3.0 * udata[1 * nx] - 1.0 / 12.0 * udata[2 * nx];
}
inline sunrealtype s4b(const sunrealtype *udata, int nx) {
  return -1.0 / 4.0 * udata[-1 * nx] - 5.0 / 6.0 * udata[0] +
         3.0 / 2.0 * udata[1 * nx] - 1.0 / 2.0 * udata[2 * nx] +
         1.0 / 12.0 * udata[3 * nx];
}
inline sunrealtype s5f(const sunrealtype *udata, int nx) {
  return -1.0 / 30.0 * udata[-3 * nx] + 1.0 / 4.0 * udata[-2 * nx] -
         1.0 / 1.0 * udata[-1 * nx] + 1.0 / 3.0 * udata[0] +
         1.0 / 2.0 * udata[1 * nx] - 1.0 / 20.0 * udata[2 * nx];
}
inline sunrealtype s5b(sunrealtype *udata, int nx) {
  return 1.0 / 20.0 * udata[-2 * nx] - 1.0 / 2.0 * udata[-1 * nx] -
         1.0 / 3.0 * udata[0] + udata[1 * nx] - 1.0 / 4.0 * udata[2 * nx] +
         1.0 / 30.0 * udata[3 * nx];
}
inline sunrealtype s6f(const sunrealtype *udata, int nx) {
  return 1.0 / 60.0 * udata[-4 * nx] - 2.0 / 15.0 * udata[-3 * nx] +
         1.0 / 2.0 * udata[-2 * nx] - 4.0 / 3.0 * udata[-1 * nx] +
         7.0 / 12.0 * udata[0] + 2.0 / 5.0 * udata[1 * nx] -
         1.0 / 30.0 * udata[2 * nx];
}
inline sunrealtype s6c(const sunrealtype *udata, int nx) {
  return -1.0 / 60.0 * udata[-3 * nx] + 3.0 / 20.0 * udata[-2 * nx] -
         3.0 / 4.0 * udata[-1 * nx] + 0 + 3.0 / 4.0 * udata[1 * nx] -
         3.0 / 20.0 * udata[2 * nx] + 1.0 / 60.0 * udata[3 * nx];
}
inline sunrealtype s6b(const sunrealtype *udata, int nx) {
  return 1.0 / 30.0 * udata[-2 * nx] - 2.0 / 5.0 * udata[-1 * nx] -
         7.0 / 12.0 * udata[0] + 4.0 / 3.0 * udata[1 * nx] -
         1.0 / 2.0 * udata[2 * nx] + 2.0 / 15.0 * udata[3 * nx] -
         1.0 / 60.0 * udata[4 * nx];
}
```

```
inline sunrealtype s7f(const sunrealtype *udata, int nx) {
  return 1.0 / 140.0 * udata[-4 * nx] - 1.0 / 15.0 * udata[-3 * nx] +
         3.0 / 10.0 * udata[-2 * nx] - 1.0 / 1.0 * udata[-1 * nx] +
         1.0 / 4.0 * udata[0] + 3.0 / 5.0 * udata[1 * nx] -
         1.0 / 10.0 * udata[2 * nx] + 1.0 / 105.0 * udata[3 * nx];
}
inline sunrealtype s7b(sunrealtype *udata, int nx) {
  return -1.0 / 105.0 * udata[-3 * nx] + 1.0 / 10.0 * udata[-2 * nx] -
         3.0 / 5.0 * udata[-1 * nx] - 1.0 / 4.0 * udata[0] + udata[1 * nx] -
         3.0 / 10.0 * udata[2 * nx] + 1.0 / 15.0 * udata[3 * nx] -
         1.0 / 140.0 * udata[4 * nx];
}
inline sunrealtype s8f(const sunrealtype *udata, int nx) {
  return -1.0 / 280.0 * udata[-5 * nx] + 1.0 / 28.0 * udata[-4 * nx] -
         1.0 / 6.0 * udata[-3 * nx] + 1.0 / 2.0 * udata[-2 * nx] -
         5.0 / 4.0 * udata[-1 * nx] + 9.0 / 20.0 * udata[0] +
         1.0 / 2.0 * udata[1 * nx] - 1.0 / 14.0 * udata[2 * nx] +
         1.0 / 168.0 * udata[3 * nx];
}
inline sunrealtype s8c(const sunrealtype *udata, int nx) {
  return 1.0 / 280.0 * udata[-4 * nx] - 4.0 / 105.0 * udata[-3 * nx] +
         1.0 / 5.0 * udata[-2 * nx] - 4.0 / 5.0 * udata[-1 * nx] + 0 +
         4.0 / 5.0 * udata[1 * nx] - 1.0 / 5.0 * udata[2 * nx] +
         4.0 / 105.0 * udata[3 * nx] - 1.0 / 280.0 * udata[4 * nx];
}
inline sunrealtype s8b(const sunrealtype *udata, int nx) {
  return -1.0 / 168.0 * udata[-3 * nx] + 1.0 / 14.0 * udata[-2 * nx] -
         1.0 / 2.0 * udata[-1 * nx] - 9.0 / 20.0 * udata[0] +
         5.0 / 4.0 * udata[1 * nx] - 1.0 / 2.0 * udata[2 * nx] +
         1.0 / 6.0 * udata[3 * nx] - 1.0 / 28.0 * udata[4 * nx] +
         1.0 / 280.0 * udata[5 * nx];
}
inline sunrealtype s9f(const sunrealtype *udata, int nx) {
  return -1.0 / 630.0 * udata[-5 * nx] + 1.0 / 56.0 * udata[-4 * nx] -
         2.0 / 21.0 * udata[-3 * nx] + 1.0 / 3.0 * udata[-2 * nx] -
         1.0 / 1.0 * udata[-1 * nx] + 1.0 / 5.0 * udata[0] +
         2.0 / 3.0 * udata[1 * nx] - 1.0 / 7.0 * udata[2 * nx] +
         1.0 / 42.0 * udata[3 * nx] - 1.0 / 504.0 * udata[4 * nx];
}
inline sunrealtype s9b(sunrealtype *udata, int nx) {
  return 1.0 / 504.0 * udata[-4 * nx] - 1.0 / 42.0 * udata[-3 * nx] +
         1.0 / 7.0 * udata[-2 * nx] - 2.0 / 3.0 * udata[-1 * nx] -
         1.0 / 5.0 * udata[0] + udata[1 * nx] - 1.0 / 3.0 * udata[2 * nx] +
         2.0 / 21.0 * udata[3 * nx] - 1.0 / 56.0 * udata[4 * nx] +
         1.0 / 630.0 * udata[5 * nx];
}
inline sunrealtype s10f(const sunrealtype *udata, int nx) {
  return 1.0 / 1260.0 * udata[-6 * nx] - 1.0 / 105.0 * udata[-5 * nx] +
         3.0 / 56.0 * udata[-4 * nx] - 4.0 / 21.0 * udata[-3 * nx] +
         1.0 / 2.0 * udata[-2 * nx] - 6.0 / 5.0 * udata[-1 * nx] +
         11.0 / 30.0 * udata[0] + 4.0 / 7.0 * udata[1 * nx] -
         3.0 / 28.0 * udata[2 * nx] + 1.0 / 63.0 * udata[3 * nx] -
         1.0 / 840.0 * udata[4 * nx];
}
inline sunrealtype s10c(const sunrealtype *udata, int nx) {
  return -1.0 / 1260.0 * udata[-5 * nx] + 5.0 / 504.0 * udata[-4 * nx] -
         5.0 / 84.0 * udata[-3 * nx] + 5.0 / 21.0 * udata[-2 * nx] -
         5.0 / 6.0 * udata[-1 * nx] + 0 + 5.0 / 6.0 * udata[1 * nx] -
         5.0 / 21.0 * udata[2 * nx] + 5.0 / 84.0 * udata[3 * nx] -
         5.0 / 504.0 * udata[4 * nx] + 1.0 / 1260.0 * udata[5 * nx];
}
inline sunrealtype s10b(const sunrealtype *udata, int nx) {
  return 1.0 / 840.0 * udata[-4 * nx] - 1.0 / 63.0 * udata[-3 * nx] +
         3.0 / 28.0 * udata[-2 * nx] - 4.0 / 7.0 * udata[-1 * nx] -
         11.0 / 30.0 * udata[0] + 6.0 / 5.0 * udata[1 * nx] -
         1.0 / 2.0 * udata[2 * nx] + 4.0 / 21.0 * udata[3 * nx] -
         3.0 / 56.0 * udata[4 * nx] + 1.0 / 105.0 * udata[5 * nx] -
         1.0 / 1260.0 * udata[6 * nx];
}
inline sunrealtype s11f(const sunrealtype *udata, int nx) {
  return 1.0 / 2772.0 * udata[-6 * nx] - 1.0 / 210.0 * udata[-5 * nx] +
         5.0 / 168.0 * udata[-4 * nx] - 5.0 / 42.0 * udata[-3 * nx] +
         5.0 / 14.0 * udata[-2 * nx] - 1.0 / 1.0 * udata[-1 * nx] +
         1.0 / 6.0 * udata[0] + 5.0 / 7.0 * udata[1 * nx] -
         5.0 / 28.0 * udata[2 * nx] + 5.0 / 126.0 * udata[3 * nx] -
         1.0 / 168.0 * udata[4 * nx] + 1.0 / 2310.0 * udata[5 * nx];
}
inline sunrealtype s11b(sunrealtype *udata, int nx) {
  return -1.0 / 2310.0 * udata[-5 * nx] + 1.0 / 168.0 * udata[-4 * nx] -
         5.0 / 126.0 * udata[-3 * nx] + 5.0 / 28.0 * udata[-2 * nx] -
         5.0 / 7.0 * udata[-1 * nx] - 1.0 / 6.0 * udata[0] + udata[1 * nx] -
         5.0 / 14.0 * udata[2 * nx] + 5.0 / 42.0 * udata[3 * nx] -
         5.0 / 168.0 * udata[4 * nx] + 1.0 / 210.0 * udata[5 * nx] -
         1.0 / 2772.0 * udata[6 * nx];
}
inline sunrealtype s12f(const sunrealtype *udata, int nx) {
  return -1.0 / 5544.0 * udata[-7 * nx] + 1.0 / 396.0 * udata[-6 * nx] -
```

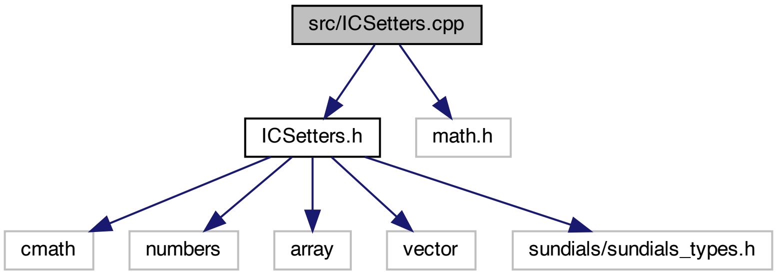
```
         1.0 / 60.0 * udata[-5 * nx] + 5.0 / 72.0 * udata[-4 * nx] -
         5.0 / 24.0 * udata[-3 * nx] + 1.0 / 2.0 * udata[-2 * nx] -
         7.0 / 6.0 * udata[-1 * nx] + 13.0 / 42.0 * udata[0] +
         5.0 / 8.0 * udata[1 * nx] - 5.0 / 36.0 * udata[2 * nx] +
         1.0 / 36.0 * udata[3 * nx] - 1.0 / 264.0 * udata[4 * nx] +
         1.0 / 3960.0 * udata[5 * nx];
}
inline sunrealtype s12c(const sunrealtype *udata, int nx) {
  return 1.0 / 5544.0 * udata[-6 * nx] - 1.0 / 385.0 * udata[-5 * nx] +
         1.0 / 56.0 * udata[-4 * nx] - 5.0 / 63.0 * udata[-3 * nx] +
         15.0 / 56.0 * udata[-2 * nx] - 6.0 / 7.0 * udata[-1 * nx] + 0 +
         6.0 / 7.0 * udata[1 * nx] - 15.0 / 56.0 * udata[2 * nx] +
         5.0 / 63.0 * udata[3 * nx] - 1.0 / 56.0 * udata[4 * nx] +
         1.0 / 385.0 * udata[5 * nx] - 1.0 / 5544.0 * udata[6 * nx];
}
inline sunrealtype s12b(const sunrealtype *udata, int nx) {
  return -1.0 / 3960.0 * udata[-5 * nx] + 1.0 / 264.0 * udata[-4 * nx] -
         1.0 / 36.0 * udata[-3 * nx] + 5.0 / 36.0 * udata[-2 * nx] -
         5.0 / 8.0 * udata[-1 * nx] - 13.0 / 42.0 * udata[0] +
         7.0 / 6.0 * udata[1 * nx] - 1.0 / 2.0 * udata[2 * nx] +
         5.0 / 24.0 * udata[3 * nx] - 5.0 / 72.0 * udata[4 * nx] +
         1.0 / 60.0 * udata[5 * nx] - 1.0 / 396.0 * udata[6 * nx] +
         1.0 / 5544.0 * udata[7 * nx];
}
//#pragma omp declare simd notinbranch uniform(nx) simdlen(13) // Is this safe
//here? Do I need critical or atomic?
inline sunrealtype s13f(const sunrealtype *udata, int nx) {
  return -1.0 / 12012.0 * udata[-7 * nx] + 1.0 / 792.0 * udata[-6 * nx] -
         1.0 / 110.0 * udata[-5 * nx] + 1.0 / 24.0 * udata[-4 * nx] -
         5.0 / 36.0 * udata[-3 * nx] + 3.0 / 8.0 * udata[-2 * nx] -
         1.0 / 1.0 * udata[-1 * nx] + 1.0 / 7.0 * udata[0] +
         3.0 / 4.0 * udata[1 * nx] - 5.0 / 24.0 * udata[2 * nx] +
         1.0 / 18.0 * udata[3 * nx] - 1.0 / 88.0 * udata[4 * nx] +
         1.0 / 660.0 * udata[5 * nx] - 1.0 / 10296.0 * udata[6 * nx];
}
//#pragma omp declare simd notinbranch uniform(nx) simdlen(13)
inline sunrealtype s13b(sunrealtype *udata, int nx) {
  return 1.0 / 10296.0 * udata[-6 * nx] - 1.0 / 660.0 * udata[-5 * nx] +
         1.0 / 88.0 * udata[-4 * nx] - 1.0 / 18.0 * udata[-3 * nx] +
         5.0 / 24.0 * udata[-2 * nx] - 3.0 / 4.0 * udata[-1 * nx] -
         1.0 / 7.0 * udata[0] + udata[1 * nx] - 3.0 / 8.0 * udata[2 * nx] +
         5.0 / 36.0 * udata[3 * nx] - 1.0 / 24.0 * udata[4 * nx] +
         1.0 / 110.0 * udata[5 * nx] - 1.0 / 792.0 * udata[6 * nx] +
         1.0 / 12012.0 * udata[7 * nx];
}

////////////////////////////////
// Stencils with nx fixed to 6//
////////////////////////////////

inline sunrealtype s1f(sunrealtype *udata) { return s1f(udata, 6); }
inline sunrealtype s1b(sunrealtype *udata) { return s1b(udata, 6); }
inline sunrealtype s2f(sunrealtype *udata) { return s2f(udata, 6); }
inline sunrealtype s2c(sunrealtype *udata) { return s2c(udata, 6); }
inline sunrealtype s2b(sunrealtype *udata) { return s2b(udata, 6); }
inline sunrealtype s3f(sunrealtype *udata) { return s3f(udata, 6); }
inline sunrealtype s3b(sunrealtype *udata) { return s3b(udata, 6); }
inline sunrealtype s4f(sunrealtype *udata) { return s4f(udata, 6); }
inline sunrealtype s4c(sunrealtype *udata) { return s4c(udata, 6); }
inline sunrealtype s4b(sunrealtype *udata) { return s4b(udata, 6); }
inline sunrealtype s5f(sunrealtype *udata) { return s5f(udata, 6); }
inline sunrealtype s5b(sunrealtype *udata) { return s5b(udata, 6); }
inline sunrealtype s6f(sunrealtype *udata) { return s6f(udata, 6); }
inline sunrealtype s6c(sunrealtype *udata) { return s6c(udata, 6); }
inline sunrealtype s6b(sunrealtype *udata) { return s6b(udata, 6); }
inline sunrealtype s7f(sunrealtype *udata) { return s7f(udata, 6); }
inline sunrealtype s7b(sunrealtype *udata) { return s7b(udata, 6); }
inline sunrealtype s8f(sunrealtype *udata) { return s8f(udata, 6); }
inline sunrealtype s8c(sunrealtype *udata) { return s8c(udata, 6); }
inline sunrealtype s8b(sunrealtype *udata) { return s8b(udata, 6); }
inline sunrealtype s9f(sunrealtype *udata) { return s9f(udata, 6); }
inline sunrealtype s9b(sunrealtype *udata) { return s9b(udata, 6); }
inline sunrealtype s10f(sunrealtype *udata) { return s10f(udata, 6); }
inline sunrealtype s10c(sunrealtype *udata) { return s10c(udata, 6); }
inline sunrealtype s10b(sunrealtype *udata) { return s10b(udata, 6); }
inline sunrealtype s11f(sunrealtype *udata) { return s11f(udata, 6); }
inline sunrealtype s11b(sunrealtype *udata) { return s11b(udata, 6); }
inline sunrealtype s12f(sunrealtype *udata) { return s12f(udata, 6); }
inline sunrealtype s12c(sunrealtype *udata) { return s12c(udata, 6); }
inline sunrealtype s12b(sunrealtype *udata) { return s12b(udata, 6); }
inline sunrealtype s13f(sunrealtype *udata) { return s13f(udata, 6); }
inline sunrealtype s13b(sunrealtype *udata) { return s13b(udata, 6); }
```

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.

```
///////////////////////////////////////////////////////////////////////////////////////
/// @file ICSetters.cpp
/// @brief Implementation of the plane wave and Gaussian wave packets in 1D, 2D,
/// 3D
///////////////////////////////////////////////////////////////////////////////////////

#include "ICSetters.h"

#include <math.h>

/** PlaneWave1D construction with */
PlaneWave1D::PlaneWave1D(vector<sunrealtype> k, vector<sunrealtype> p,
                         vector<sunrealtype> phi) {
  kx = k[0]; /** - wavevectors \f$ k_x \f$ */
  ky = k[1]; /** - \f$ k_y \f$ */
  kz = k[2]; /** - \f$ k_z \f$ normalized to \f$ 1/\lambda \f$ */
  // Amplitude bug: lower by factor 3
  px = p[0] / 3; /** - amplitude (polarization) in x-direction \f$ p_x \f$ */
  py = p[1] / 3; /** - amplitude (polarization) in y-direction \f$ p_y \f$ */
  pz = p[2] / 3; /** - amplitude (polarization) in z-direction \f$ p_z \f$ */
  phix = phi[0]; /** - phase shift in x-direction \f$ \phi_x \f$ */
  phiy = phi[1]; /** - phase shift in y-direction \f$ \phi_y \f$ */
  phiz = phi[2]; /** - phase shift in z-direction \f$ \phi_z \f$ */
}

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

```
  const sunrealtype kScalarX = (kx * x + ky * y + kz * z) * 2 *
                         numbers::pi; /* \f$ 2\pi \ \vec{k} \cdot \vec{x} \f$ */
  // Plane wave definition
  const array<sunrealtype, 3> E{{                                     /* E-field vector */
                           px * cos(kScalarX - phix),    /* \f$ E_x \f$ */
                           py * cos(kScalarX - phiy),    /* \f$ E_y \f$ */
                           pz * cos(kScalarX - phiz)}}; /* \f$ E_z \f$ */
  // Put E-field into space
  pTo6Space[0] += E[0];
  pTo6Space[1] += E[1];
  pTo6Space[2] += E[2];
  // and B-field
  pTo6Space[3] += (ky * E[2] - kz * E[1]) / wavelength;
  pTo6Space[4] += (kz * E[0] - kx * E[2]) / wavelength;
  pTo6Space[5] += (kx * E[1] - ky * E[0]) / wavelength;
}

/** PlaneWave2D construction with */
PlaneWave2D::PlaneWave2D(vector<sunrealtype> k, vector<sunrealtype> p,
                         vector<sunrealtype> phi) {
  kx = k[0]; /** - wavevectors \f$ k_x \f$ */
  ky = k[1]; /** - \f$ k_y \f$ */
  kz = k[2]; /** - \f$ k_z \f$ normalized to \f$ 1/\lambda \f$*/
  // Amplitude bug: lower by factor 9
  px = p[0] / 9; /** - amplitude (polarization) in x-direction \f$ p_x \f$ */
  py = p[1] / 9; /** - amplitude (polarization) in y-direction \f$ p_y \f$ */
  pz = p[2] / 9; /** - amplitude (polarization) in z-direction \f$ p_z \f$ */
  phix = phi[0]; /** - phase shift in x-direction \f$ \phi_x \f$ */
  phiy = phi[1]; /** - phase shift in y-direction \f$ \phi_y \f$ */
  phiz = phi[2]; /** - phase shift in z-direction \f$ \phi_z \f$ */
}

/** PlaneWave2D implementation in space */
//#pragma omp declare simd uniform(x,y,z) linear(pTo6Space:6)
void PlaneWave2D::addToSpace(const sunrealtype x, const sunrealtype y, const sunrealtype z,
                             sunrealtype *pTo6Space) const {
  const sunrealtype wavelength =
      sqrt(kx * kx + ky * ky + kz * kz); /* \f$ 1/\lambda \f$ */
  const sunrealtype kScalarX = (kx * x + ky * y + kz * z) * 2 *
                         numbers::pi; /* \f$ 2\pi \ \vec{k} \cdot \vec{x} \f$ */
  // Plane wave definition
  const array<sunrealtype, 3> E{{                                     /* E-field vector */
                           px * cos(kScalarX - phix),    /* \f$ E_x \f$ */
                           py * cos(kScalarX - phiy),    /* \f$ E_y \f$ */
                           pz * cos(kScalarX - phiz)}}; /* \f$ E_z \f$ */
  // Put E-field into space
  pTo6Space[0] += E[0];
  pTo6Space[1] += E[1];
  pTo6Space[2] += E[2];
  // and B-field
  pTo6Space[3] += (ky * E[2] - kz * E[1]) / wavelength;
  pTo6Space[4] += (kz * E[0] - kx * E[2]) / wavelength;
  pTo6Space[5] += (kx * E[1] - ky * E[0]) / wavelength;
}

/** PlaneWave3D construction with */
PlaneWave3D::PlaneWave3D(vector<sunrealtype> k, vector<sunrealtype> p,
                         vector<sunrealtype> phi) {
  kx = k[0];     /** - wavevectors \f$ k_x \f$ */
  ky = k[1];     /** - \f$ k_y \f$ */
  kz = k[2];     /** - \f$ k_z \f$ normalized to \f$ 1/\lambda \f$ */
  px = p[0];     /** - amplitude (polarization) in x-direction \f$ p_x \f$ */
  py = p[1];     /** - amplitude (polarization) in y-direction \f$ p_y \f$ */
  pz = p[2];     /** - amplitude (polarization) in z-direction \f$ p_z \f$ */
  phix = phi[0]; /** - phase shift in x-direction \f$ \phi_x \f$ */
  phiy = phi[1]; /** - phase shift in y-direction \f$ \phi_y \f$ */
  phiz = phi[2]; /** - phase shift in z-direction \f$ \phi_z \f$ */
}

/** PlaneWave3D implementation in space */
//#pragma omp declare simd uniform(x,y,z) linear(pTo6Space:6)
void PlaneWave3D::addToSpace(sunrealtype x, sunrealtype y, sunrealtype z,
                             sunrealtype *pTo6Space) const {
  const sunrealtype wavelength =
      sqrt(kx * kx + ky * ky + kz * kz); /* \f$ 1/\lambda \f$ */
  const sunrealtype kScalarX = (kx * x + ky * y + kz * z) * 2 *
                         numbers::pi; /* \f$ 2\pi \ \vec{k} \cdot \vec{x} \f$ */
  // Plane wave definition
  const array<sunrealtype, 3> E{{/* E-field vector \f$ \vec{E}\f$*/
                           px * cos(kScalarX - phix),    /* \f$ E_x \f$ */
                           py * cos(kScalarX - phiy),    /* \f$ E_y \f$ */
                           pz * cos(kScalarX - phiz)}}; /* \f$ E_z \f$ */
  // Put E-field into space
  pTo6Space[0] += E[0];
  pTo6Space[1] += E[1];
  pTo6Space[2] += E[2];
  // and B-field
```

```
  pTo6Space[3] += (ky * E[2] - kz * E[1]) / wavelength;
  pTo6Space[4] += (kz * E[0] - kx * E[2]) / wavelength;
  pTo6Space[5] += (kx * E[1] - ky * E[0]) / wavelength;
}

/** Gauss1D construction with */
Gauss1D::Gauss1D(vector<sunrealtype> k, vector<sunrealtype> p,
                 vector<sunrealtype> xo, sunrealtype phig_,
                 vector<sunrealtype> phi) {
  kx = k[0];      /** - wavevectors \f$ k_x \f$ */
  ky = k[1];      /** - \f$ k_y \f$ */
  kz = k[2];      /** - \f$ k_z \f$ normalized to \f$ 1/\lambda \f$*/
  px = p[0];      /** - amplitude (polarization) in x-direction */
  py = p[1];      /** - amplitude (polarization) in y-direction */
  pz = p[2];      /** - amplitude (polarization) in z-direction */
  phix = phi[0]; /** - phase shift in x-direction */
  phiy = phi[1]; /** - phase shift in y-direction */
  phiz = phi[2]; /** - phase shift in z-direction */
  phig = phig_;  /** - width */
  x0x = xo[0];   /** - shift from origin in x-direction*/
  x0y = xo[1];   /** - shift from origin in y-direction*/
  x0z = xo[2];   /** - shift from origin in z-direction*/
}

/** Gauss1D implementation in space */
//#pragma omp declare simd uniform(x,y,z) linear(pTo6Space:6)
void Gauss1D::addToSpace(sunrealtype x, sunrealtype y, sunrealtype z,
                         sunrealtype *pTo6Space) const {
  const sunrealtype wavelength =
      sqrt(kx * kx + ky * ky + kz * kz); /* \f$ 1/\lambda \f$ */
  x = x - x0x; /* x-coordinate minus shift from origin */
  y = y - x0y; /* y-coordinate minus shift from origin */
  z = z - x0z; /* z-coordinate minus shift from origin */
  const sunrealtype kScalarX = (kx * x + ky * y + kz * z) * 2 *
                         numbers::pi; /* \f$ 2\pi \ \vec{k} \cdot \vec{x} \f$ */
  const sunrealtype envelopeAmp =
      exp(-(x * x + y * y + z * z) / phig / phig); /* enveloping Gauss shape */
  // Gaussian wave definition
  const array<sunrealtype, 3> E{
      {                                           /* E-field vector */
       px * cos(kScalarX - phix) * envelopeAmp,   /* \f$ E_x \f$ */
       py * cos(kScalarX - phiy) * envelopeAmp,   /* \f$ E_y \f$ */
       pz * cos(kScalarX - phiz) * envelopeAmp}}; /* \f$ E_z \f$ */
  // Put E-field into space
  pTo6Space[0] += E[0];
  pTo6Space[1] += E[1];
  pTo6Space[2] += E[2];
  // and B-field
  pTo6Space[3] += (ky * E[2] - kz * E[1]) / wavelength;
  pTo6Space[4] += (kz * E[0] - kx * E[2]) / wavelength;
  pTo6Space[5] += (kx * E[1] - ky * E[0]) / wavelength;
}

/** Gauss2D construction with */
Gauss2D::Gauss2D(vector<sunrealtype> dis_, vector<sunrealtype> axis_,
                 sunrealtype Amp_, sunrealtype phip_, sunrealtype w0_,
                 sunrealtype zr_, sunrealtype Ph0_, sunrealtype PhA_) {
  dis = dis_;           /** - center it approaches */
  axis = axis_;         /** - direction form where it comes */
  Amp = Amp_;           /** - amplitude */
  phip = phip_;         /** - polarization rotation from TE-mode */
  w0 = w0_;             /** - taille */
  zr = zr_;             /** - Rayleigh length */
  Ph0 = Ph0_;           /** - beam center */
  PhA = PhA_;           /** - beam length */
  A1 = Amp * cos(phip); // amplitude in z-direction
  A2 = Amp * sin(phip); // amplitude on xy-plane
  lambda = numbers::pi * w0 * w0 / zr; // formula for wavelength
}

//#pragma omp declare simd uniform(x,y,z) linear(pTo6Space:6)
void Gauss2D::addToSpace(sunrealtype x, sunrealtype y, sunrealtype z,
                         sunrealtype *pTo6Space) const {
  //\f$ \vec{x} = \vec{x}_0-\vec{dis} \f$ // coordinates minus distance to
  //origin
  x -= dis[0];
  y -= dis[1];
  // z-=dis[2];
  z = NAN;
  //  \f$ z_g = \vec{x}\cdot\vec{e}_g \f$ projection on propagation axis
  const sunrealtype zg =
      x * axis[0] + y * axis[1]; //+z*axis[2];  // =z-z0 -> propagation
                                 //direction, minus origin
  // \f$ r = \sqrt{\vec{x}^2 -z_g^2} \f$ -> pythagoras of radius minus
  // projection on prop axis
  const sunrealtype r = sqrt((x * x + y * y /*+z*z*/) -
                        zg * zg); // radial distance to propagation axis
```

```
  // \f$  w(z) = w0\sqrt{1+(z_g/z_R)^2} \f$
  const sunrealtype wz = w0 * sqrt(1 + (zg * zg / zr / zr)); // waist at position z
  // \f$ g(z) = atan(z_g/z_r) \f$
  const sunrealtype gz = atan(zg / zr); // Gouy phase
  // \f$ R(z) = z_g*(1+(z_r/z_g)^2) \f$
  sunrealtype Rz = NAN; // beam curvature
  if (zg != 0)
    Rz = zg * (1 + (zr * zr / zg / zg));
  else
    Rz = 1e308;
  // wavenumber \f$ k = 2\pi/\lambda \f$
  const sunrealtype k = 2 * numbers::pi / lambda;
  // \f$ \Phi_F = kr^2/(2*R(z))+g(z)-kz_g \f$
  const sunrealtype PhF =
      -k * r * r / (2 * Rz) + gz - k * zg; // to be inserted into cosine
  // \f$ G = \sqrt{w_0/w_z}\e^{-(r/w(z))^2}\e^{(zg-Ph0)^2/PhA^2}\cos(PhF) \f$
  // CVode is a diva, no chance to remove the square in the second exponential
  // -> h too small
  const sunrealtype G2D = sqrt(w0 / wz) * exp(-r * r / wz / wz) *
                          exp(-(zg - Ph0) * (zg - Ph0) / PhA / PhA) *
                          cos(PhF); // gauss shape
  // \f$ c_\alpha =\vec{e}_x\cdot\vec{axis} \f$
  // projection components; do like this for CVode convergence -> otherwise
  // results in machine error values for non-existant field components if
  // axis[0] and axis[1] are given
  const sunrealtype ca =
      axis[0]; // x-component of propagation axis which is given as parameter
  const sunrealtype sa = sqrt(1 - ca * ca); // no z-component for 2D propagation
  // E-field to space: polarization in xy-plane (A2) is projection of
  // z-polarization (A1) on x- and y-directions
  pTo6Space[0] += sa * (G2D * A2);
  pTo6Space[1] += -ca * (G2D * A2);
  pTo6Space[2] += G2D * A1;
  // B-field -> negative derivative wrt polarization shift of E-field
  pTo6Space[3] += -sa * (G2D * A1);
  pTo6Space[4] += ca * (G2D * A1);
  pTo6Space[5] += G2D * A2;
}

/** Gauss3D construction with */
Gauss3D::Gauss3D(vector<sunrealtype> dis_, vector<sunrealtype> axis_,
                 sunrealtype Amp_,
                 // vector<sunrealtype> pol_,
                 sunrealtype phip_, sunrealtype w0_, sunrealtype zr_,
                 sunrealtype Ph0_, sunrealtype PhA_) {
  dis = dis_;   /** - center it approaches */
  axis = axis_; /** - direction from where it comes */
  Amp = Amp_;   /** - amplitude */
  // pol=pol_;
  phip = phip_; /** - polarization rotation form TE-mode */
  w0 = w0_;     /** - taille */
  zr = zr_;     /** - Rayleigh length */
  Ph0 = Ph0_;   /** - beam center */
  PhA = PhA_;   /** - beam length */
  lambda = numbers::pi * w0 * w0 / zr;
  A1 = Amp * cos(phip);
  A2 = Amp * sin(phip);
}

/** Gauss3D implementation in space */
void Gauss3D::addToSpace(sunrealtype x, sunrealtype y, sunrealtype z,
                         sunrealtype *pTo6Space) const {
  x -= dis[0];
  y -= dis[1];
  z -= dis[2];
  const sunrealtype zg = x * axis[0] + y * axis[1] + z * axis[2];
  const sunrealtype r = sqrt((x * x + y * y + z * z) - zg * zg);
  const sunrealtype wz = w0 * sqrt(1 + (zg * zg / zr / zr));
  const sunrealtype gz = atan(zg / zr);
  sunrealtype Rz = NAN;
  if (zg != 0)
    Rz = zg * (1 + (zr * zr / zg / zg));
  else
    Rz = 1e308;
  const sunrealtype k = 2 * numbers::pi / lambda;
  const sunrealtype PhF = -k * r * r / (2 * Rz) + gz - k * zg;
  const sunrealtype G3D = (w0 / wz) * exp(-r * r / wz / wz) *
                          exp(-(zg - Ph0) * (zg - Ph0) / PhA / PhA) * cos(PhF);
  const sunrealtype ca = axis[0];
  const sunrealtype sa = sqrt(1 - ca * ca);
  pTo6Space[0] += sa * (G3D * A2);
  pTo6Space[1] += -ca * (G3D * A2);
  pTo6Space[2] += G3D * A1;
  pTo6Space[3] += -sa * (G3D * A1);
  pTo6Space[4] += ca * (G3D * A1);
  pTo6Space[5] += G3D * A2;
}
```

```
/** Evaluate lattice point values to zero and add field values */
//# pragma omp declare simd uniform(x,y,z) linear(pTo6Space:6)
void ICSetter::eval(sunrealtype x, sunrealtype y, sunrealtype z,
                    sunrealtype *pTo6Space) {
  pTo6Space[0] = 0;
  pTo6Space[1] = 0;
  pTo6Space[2] = 0;
  pTo6Space[3] = 0;
  pTo6Space[4] = 0;
  pTo6Space[5] = 0;
  add(x, y, z, pTo6Space);
}

/** Add all initial field values to the lattice space */
void ICSetter::add(sunrealtype x, sunrealtype y, sunrealtype z,
                   sunrealtype *pTo6Space) {
  for (const auto &wave : planeWaves1D)
    wave.addToSpace(x, y, z, pTo6Space);
  for (const auto &wave : planeWaves2D)
    wave.addToSpace(x, y, z, pTo6Space);
  for (const auto &wave : planeWaves3D)
    wave.addToSpace(x, y, z, pTo6Space);
  for (const auto &wave : gauss1Ds)
    wave.addToSpace(x, y, z, pTo6Space);
  for (const auto &wave : gauss2Ds)
    wave.addToSpace(x, y, z, pTo6Space);
  for (const auto &wave : gauss3Ds)
    wave.addToSpace(x, y, z, pTo6Space);
}

/** Add plane waves in 1D to their container vector */
void ICSetter::addPlaneWave1D(vector<sunrealtype> k, vector<sunrealtype> p,
                              vector<sunrealtype> phi) {
  planeWaves1D.emplace_back(PlaneWave1D(k, p, phi));
}

/** Add plane waves in 2D to their container vector */
void ICSetter::addPlaneWave2D(vector<sunrealtype> k, vector<sunrealtype> p,
                              vector<sunrealtype> phi) {
  planeWaves2D.emplace_back(PlaneWave2D(k, p, phi));
}

/** Add plane waves in 3D to their container vector */
void ICSetter::addPlaneWave3D(vector<sunrealtype> k, vector<sunrealtype> p,
                              vector<sunrealtype> phi) {
  planeWaves3D.emplace_back(PlaneWave3D(k, p, phi));
}

/** Add Gaussian waves in 1D to their container vector */
void ICSetter::addGauss1D(vector<sunrealtype> k, vector<sunrealtype> p,
                          vector<sunrealtype> xo, sunrealtype phig_,
                          vector<sunrealtype> phi) {
  gauss1Ds.emplace_back(Gauss1D(k, p, xo, phig_, phi));
}

/** Add Gaussian waves in 2D to their container vector */
void ICSetter::addGauss2D(vector<sunrealtype> dis_, vector<sunrealtype> axis_,
                          sunrealtype Amp_, sunrealtype phip_, sunrealtype w0_,
                          sunrealtype zr_, sunrealtype Ph0_, sunrealtype PhA_) {
  gauss2Ds.emplace_back(
      Gauss2D(dis_, axis_, Amp_, phip_, w0_, zr_, Ph0_, PhA_));
}

/** Add Gaussian waves in 3D to their container vector */
void ICSetter::addGauss3D(vector<sunrealtype> dis_, vector<sunrealtype> axis_,
                          sunrealtype Amp_, sunrealtype phip_, sunrealtype w0_,
                          sunrealtype zr_, sunrealtype Ph0_, sunrealtype PhA_) {
  gauss3Ds.emplace_back(
      Gauss3D(dis_, axis_, Amp_, phip_, w0_, zr_, Ph0_, PhA_));
}
```

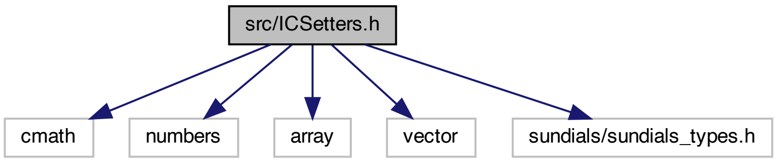
6.8 src/ICSetters.h File Reference

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

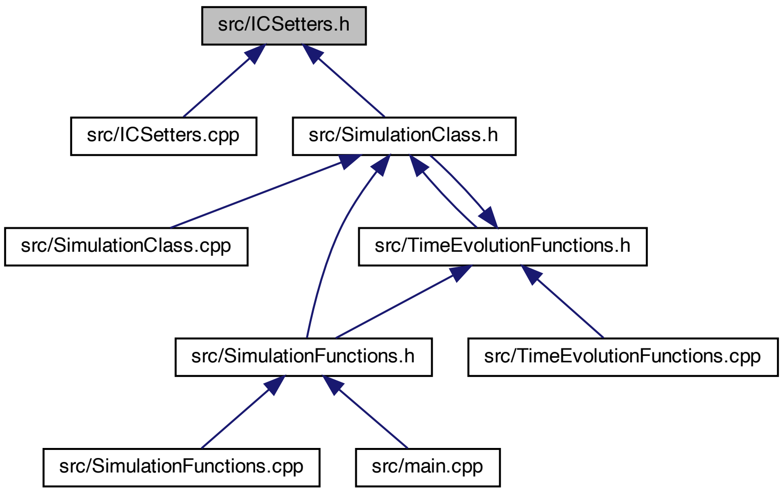
```
#include <cmath>
#include <numbers>
#include <array>
```

```
#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.

```
////////////////////////////////////////////////////////////////////////////////////////
/// @file ICSetters.h
/// @brief Declaration of the plane wave and Gaussian wave packets in 1D, 2D, 3D
////////////////////////////////////////////////////////////////////////////////////////

#pragma once

// math, constants, vector, and array
#include <cmath>
//#include <mathimf.h>
#include <numbers>
#include <array>
#include <vector>

#include <sundials/sundials_types.h> /* definition of type sunrealtype */

using namespace std;

/** @brief super-class for plane waves
 *
 * They are given in the form \f$ \vec{E} = \vec{E}_0 \ \cos \left( \vec{k}
 * \cdot \vec{x} - \vec{\phi} \right) \f$ */
class PlaneWave {
protected:
  /// wavenumber \f$ k_x \f$
  sunrealtype kx;
  /// wavenumber \f$ k_y \f$
  sunrealtype ky;
  /// wavenumber \f$ k_z \f$
  sunrealtype kz;
  /// polarization & amplitude in x-direction, \f$ p_x \f$
  sunrealtype px;
  /// polarization & amplitude in y-direction, \f$ p_y \f$
  sunrealtype py;
  /// polarization & amplitude in z-direction, \f$ p_z \f$
  sunrealtype pz;
  /// phase shift in x-direction, \f$ \phi_x \f$
  sunrealtype phix;
  /// phase shift in y-direction, \f$ \phi_y \f$
  sunrealtype phiy;
  /// phase shift in z-direction, \f$ \phi_z \f$
  sunrealtype phiz;
};

/** @brief class for plane waves in 1D */
class PlaneWave1D : public PlaneWave {
public:
  /// construction with default parameters
  PlaneWave1D(vector<sunrealtype> k = {1, 0, 0},
              vector<sunrealtype> p = {0, 0, 1},
              vector<sunrealtype> phi = {0, 0, 0});
  /// function for the actual implementation in the lattice
  void addToSpace(sunrealtype x, sunrealtype y, sunrealtype z,
                  sunrealtype *pTo6Space) const;
};

/** @brief class for plane waves in 2D */
class PlaneWave2D : public PlaneWave {
public:
```

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

/** @brief class for plane waves in 3D */
class PlaneWave3D : public PlaneWave {
public:
  /// construction with default parameters
  PlaneWave3D(vector<sunrealtype> k = {1, 0, 0},
              vector<sunrealtype> p = {0, 0, 1},
              vector<sunrealtype> phi = {0, 0, 0});
  /// function for the actual implementation in space
  void addToSpace(sunrealtype x, sunrealtype y, sunrealtype z,
                  sunrealtype *pTo6Space) const;
};

/** @brief class for Gaussian waves in 1D
 *
 * They are given in the form \f$ \vec{E}=\vec{p} \, \exp \left(
 * -(\vec{x}-\vec{x}_0)^2 / \Phi_g^2 \right) \, \cos(\vec{k} \cdot \vec{x}) \f$
 */
class Gauss1D {
private:
  /// wavenumber \f$ k_x \f$
  sunrealtype kx;
  /// wavenumber \f$ k_y \f$
  sunrealtype ky;
  /// wavenumber \f$ k_z \f$
  sunrealtype kz;
  /// polarization & amplitude in x-direction, \f$ p_x \f$
  sunrealtype px;
  /// polarization & amplitude in y-direction, \f$ p_y \f$
  sunrealtype py;
  /// polarization & amplitude in z-direction, \f$ p_z \f$
  sunrealtype pz;
  /// phase shift in x-direction, \f$ \phi_x \f$
  sunrealtype phix;
  /// phase shift in y-direction, \f$ \phi_y \f$
  sunrealtype phiy;
  /// phase shift in z-direction, \f$ \phi_z \f$
  sunrealtype phiz;
  /// center of pulse in x-direction, \f$ x_0 \f$
  sunrealtype x0x;
  /// center of pulse in y-direction, \f$ y_0 \f$
  sunrealtype x0y;
  /// center of pulse in z-direction, \f$ z_0 \f$
  sunrealtype x0z;
  /// pulse width \f$ \Phi_g \f$
  sunrealtype phig;

public:
  /// construction with default parameters
  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});
  /// function for the actual implementation in space
  void addToSpace(sunrealtype x, sunrealtype y, sunrealtype z,
                  sunrealtype *pTo6Space) const;

public:
};

/** @brief class for Gaussian waves in 2D
 *
 * They are given in the form
 * \f$ \vec{E}= A \, \vec{\epsilon} \ \sqrt{\frac{\omega_0}{\omega(z)}} \, \exp
 * \left(-r/\omega(z) \right)^2 \, \exp \left(-((z_g-\Phi_0)/\Phi_A)^2 \right)
 * \, \cos \left( \frac{k \, r^2}{2R(z)} + g(z) - k\, z_g \right)  \f$ with
 * - propagation direction (subtracted distance to origin) \f$ z_g \f$
 * - radial distance to propagation axis \f$ r = \sqrt{\vec{x}^2 -z_g^2} \f$
 * - \f$ k = 2\pi / \lambda \f$
 * - waist at position z, \f$ \omega(z) = w_0 \, \sqrt{1+(z_g/z_R)^2} \f$
 * - Gouy phase \f$ g(z) = \tan^{-1}(z_g/z_r) \f$
 * - beam curvature \f$ R(z) = z_g \, (1+(z_r/z_g)^2) \f$
 * obtained via the chosen parameters */
class Gauss2D {
private:
  /// distance maximum to origin
  vector<sunrealtype> dis;
  /// normalized propagation axis
  vector<sunrealtype> axis;
  /// amplitude \f$ A\f$
```

```
  sunrealtype Amp;
  /// polarization rotation from TE-mode around propagation direction
  // that determines \f$ \vec{\epsilon}\f$ above
  sunrealtype phip;
  /// taille \f$ \omega_0 \f$
  sunrealtype w0;
  /// Rayleigh length \f$ z_R = \pi \omega_0^2 / \lambda \f$
  sunrealtype zr;
  /// center of beam \f$ \Phi_0 \f$
  sunrealtype Ph0;
  /// length of beam \f$ \Phi_A \f$
  sunrealtype PhA;
  /// amplitude projection on TE-mode
  sunrealtype A1;
  /// amplitude projection on xy-plane
  sunrealtype A2;
  /// wavelength \f$ \lambda \f$
  sunrealtype lambda;

public:
  /// construction with default parameters
  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);
  /// function for the actual implementation in space
  void addToSpace(sunrealtype x, sunrealtype y, sunrealtype z,
                  sunrealtype *pTo6Space) const;

public:
};

/** @brief class for Gaussian waves in 3D
 *
 * They are given in the form
 * \f$ \vec{E}= A \, \vec{\epsilon} \ \frac{\omega_0}{\omega(z)} \, \exp
 * \left(-r/\omega(z) \right)^2 \, \exp \left(-((z_g-\Phi_0)/\Phi_A)^2 \right)
 * \, \cos \left( \frac{k \, r^2}{2R(z)} + g(z) - k\, z_g \right)  \f$ with
 * - propagation direction (subtracted distance to origin) \f$ z_g \f$
 * - radial distance to propagation axis \f$ r = \sqrt{\vec{x}^2 -z_g^2} \f$
 * - \f$ k = 2\pi / \lambda \f$
 * - waist at position z, \f$ \omega(z) = w_0 \, \sqrt{1+(z_g/z_R)^2} \f$
 * - Gouy phase \f$ g(z) = \tan^{-1}(z_g/z_r) \f$
 * - beam curvature \f$ R(z) = z_g \, (1+(z_r/z_g)^2) \f$
 * obtained via the chosen parameters */
class Gauss3D {
private:
  /// distance maximum to origin
  vector<sunrealtype> dis;
  /// normalized propagation axis
  vector<sunrealtype> axis;
  /// amplitude \f$ A\f$
  sunrealtype Amp;
  /// polarization rotation from TE-mode around propagation direction
  // that determines \f$ \vec{\epsilon}\f$ above
  sunrealtype phip;
  // polarization
  // vector<sunrealtype> pol;
  /// taille \f$ \omega_0 \f$
  sunrealtype w0;
  /// Rayleigh length \f$ z_R = \pi \omega_0^2 / \lambda \f$
  sunrealtype zr;
  /// center of beam \f$ \Phi_0 \f$
  sunrealtype Ph0;
  /// length of beam \f$ \Phi_A \f$
  sunrealtype PhA;
  /// amplitude projection on TE-mode (z-axis)
  sunrealtype A1;
  /// amplitude projection on xy-plane
  sunrealtype A2;
  /// wavelength \f$ \lambda \f$
  sunrealtype lambda;

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

public:
};
```

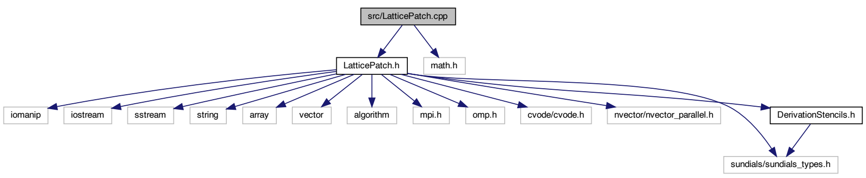
```
/** @brief ICSetter class to initialize wave types with default parameters */
class ICSetter {
private:
  /// container vector for plane waves in 1D
  vector<PlaneWave1D> planeWaves1D;
  /// container vector for plane waves in 2D
  vector<PlaneWave2D> planeWaves2D;
  /// container vector for plane waves in 3D
  vector<PlaneWave3D> planeWaves3D;
  /// container vector for Gaussian waves in 1D
  vector<Gauss1D> gauss1Ds;
  /// container vector for Gaussian waves in 2D
  vector<Gauss2D> gauss2Ds;
  /// container vector for Gaussian waves in 3D
  vector<Gauss3D> gauss3Ds;

public:
  /// function to set all coordinates to zero and then `add` the field values
  void eval(sunrealtype x, sunrealtype y, sunrealtype z,
            sunrealtype *pTo6Space);
  /// function to fill the lattice space with initial field values
  // of all field vector containers
  void add(sunrealtype x, sunrealtype y, sunrealtype z, sunrealtype *pTo6Space);
  /// function to add plane waves in 1D to their container vector
  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 2D 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 3D 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 Gaussian waves in 1D 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_ = 1.01,
                  vector<sunrealtype> phi = {0, 0, 0});
  /// function to add Gaussian waves in 2D to their container vector
  void 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 3D to their container vector
  void 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);
};

```

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 )
```

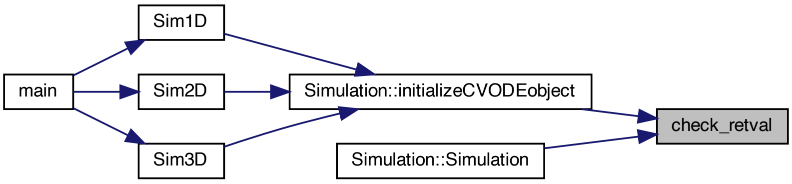
Check function return value. From CVode examples. opt == 0 means SUNDIALS function allocates memory so check if returned NULL pointer opt == 1 means SUNDIALS function returns an integer value so check if retval <0 opt == 2 means function allocates memory so check if returned NULL pointer

Definition at line 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 )
```

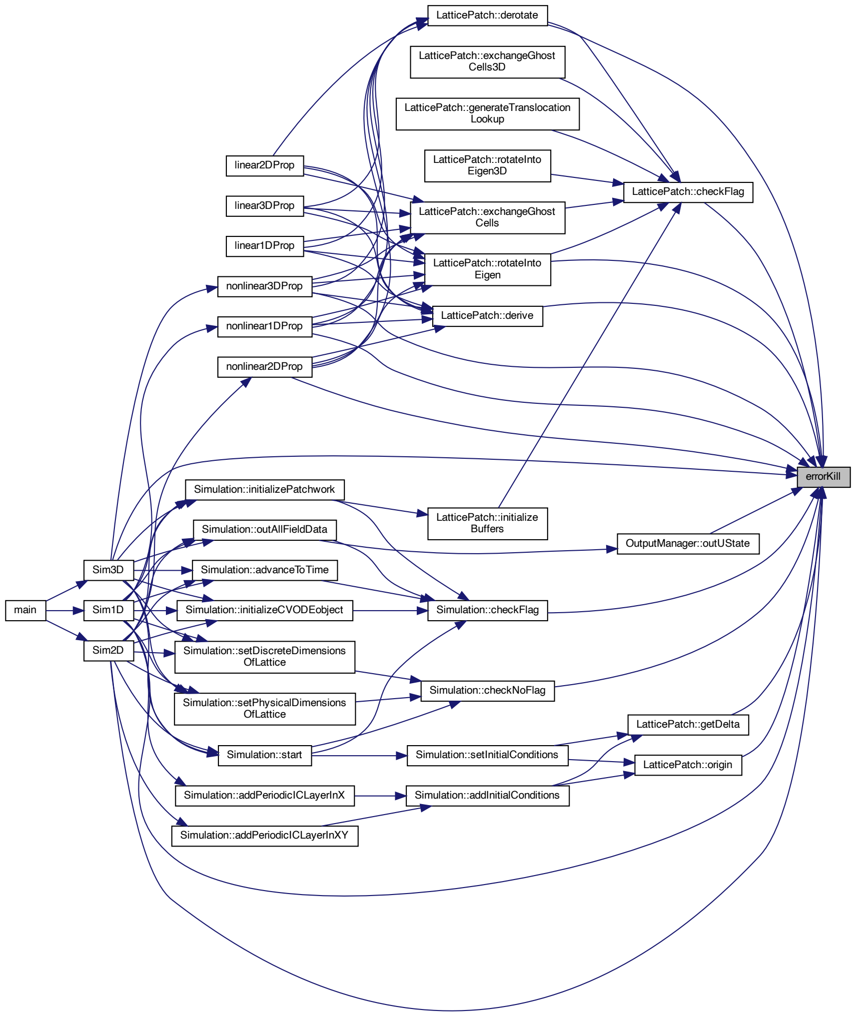
Print a specific error message to stdout.

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

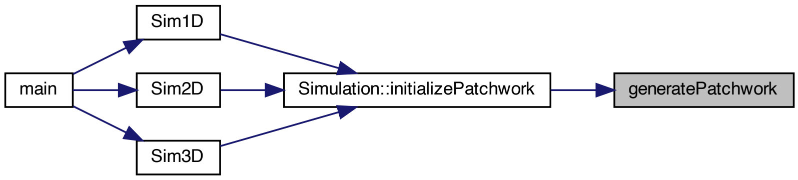
Definition at line 109 of file LatticePatch.cpp.

```
00110                                                                       {
00111     // Retrieve the ghost layer depth
```

```
    const int gLW = envelopeLattice.get_ghostLayerWidth();
    // Retrieve the data point dimension
    const int dPD = envelopeLattice.get_dataPointDimension();
    // MPI process/patch
    const int my_prc = envelopeLattice.my_prc;
    // Determine thicknes of the slice
    const sunindextype tot_NOXP = envelopeLattice.get_tot_nx(); // total points of lattice
    const sunindextype tot_NOYP = envelopeLattice.get_tot_ny();
    const sunindextype tot_NOZP = envelopeLattice.get_tot_nz();
    // position of the patch in the lattice of patches -> process associated to
    // position
    const sunindextype LIx = my_prc % DLx;
    const sunindextype LIy = (my_prc / DLx) % DLy;
    const sunindextype LIz = (my_prc / DLx) / DLy;
    // Determine the number of points in the patch and first absolute points in
    // each dimension
    const sunindextype local_NOXP = tot_NOXP / DLx;
    const sunindextype local_NOYP = tot_NOYP / DLy;
    const sunindextype local_NOZP = tot_NOZP / DLz;
    // absolute positions of the first point in each dimension
    const sunindextype firstXPoint = local_NOXP * LIx;
    const sunindextype firstYPoint = local_NOYP * LIy;
    const sunindextype firstZPoint = local_NOZP * LIz;
    // total number of points in the patch
    const sunindextype local_NODP = dPD * local_NOXP * local_NOYP * local_NOZP;

    // Set patch up with above derived quantities
    /*
    // Experiment: Resolution can be adapted for each process/patch
    const int Scaler=4; // Better hand over as parameter via `initializePatchwork`
    if(my_prc==0){patchToMold.dx=envelopeLattice.get_dx();}
    else if(my_prc==1){patchToMold.dx=envelopeLattice.get_dx()*Scaler;}
    else{errorKill("Only do this resolution barrier test with 2 processes.");}
    */
    patchToMold.dx = envelopeLattice.get_dx();
    patchToMold.dy = envelopeLattice.get_dy();
    patchToMold.dz = envelopeLattice.get_dz();
    patchToMold.x0 = firstXPoint * patchToMold.dx;
    patchToMold.y0 = firstYPoint * patchToMold.dy;
    patchToMold.z0 = firstZPoint * patchToMold.dz;
    patchToMold.LIx = LIx;
    patchToMold.LIy = LIy;
    patchToMold.LIz = LIz;
    patchToMold.nx = local_NOXP;
    patchToMold.ny = local_NOYP;
    patchToMold.nz = local_NOZP;
    patchToMold.lx = patchToMold.nx * patchToMold.dx;
    patchToMold.ly = patchToMold.ny * patchToMold.dy;
    patchToMold.lz = patchToMold.nz * patchToMold.dz;
    /* // Check name of lattice communicator
    char lattice_comm_name[MPI_MAX_OBJECT_NAME];
    int lattice_namelen;
    MPI_Comm_get_name(envelopeLattice.comm, lattice_comm_name, &lattice_namelen);
    cout«"envelopeLattice.comm gives  "« lattice_comm_name « endl;
    */
    /* Create and allocate memory for parallel vectors with defined local and
     * global lenghts *
     * (-> CVode problem sizes Nlocal and N)
     * for field data and temporal derivatives and set extra pointers to them */
    patchToMold.u =
        N_VNew_Parallel(envelopeLattice.comm, local_NODP,
                        envelopeLattice.get_tot_noDP(), envelopeLattice.sunctx);
    patchToMold.uData = NV_DATA_P(patchToMold.u);
    patchToMold.du =
        N_VNew_Parallel(envelopeLattice.comm, local_NODP,
                        envelopeLattice.get_tot_noDP(), envelopeLattice.sunctx);
    patchToMold.duData = NV_DATA_P(patchToMold.du);
    // Allocate space for auxiliary uAux so that the lattice and all possible
    // directions of ghost Layers fit
    const int s1 = patchToMold.nx, s2 = patchToMold.ny, s3 = patchToMold.nz;
    const int s_min = min(s1, min(s2, s3));
    patchToMold.uAux.resize(s1 * s2 * s3 / s_min * (s_min + 2 * gLW) * dPD);
    patchToMold.uAuxData = &patchToMold.uAux[0];
    patchToMold.envelopeLattice = &envelopeLattice;
    // Set patch "name" to process number -> only for debugging
    // patchToMold.ID=my_prc;
    // set flag
    patchToMold.statusFlags = FLatticePatchSetUp;
    patchToMold.generateTranslocationLookup();
    return 0;
}
```

Referenced by Simulation::initializePatchwork().

Here is the caller graph for this function:



6.11 LatticePatch.cpp

Go to the documentation of this file.

```
///////////////////////////////////////////////////////////////////////////////
/// @file LatticePatch.cpp
/// @brief Costruction of the overall envelope lattice and the lattice patches
///////////////////////////////////////////////////////////////////////////////

#include "LatticePatch.h"

#include <math.h>

///////////////////////////////////////////////////////
//// Implementation of Lattice component functions ////
///////////////////////////////////////////////////////

/// Initialize the cartesian communicator
void Lattice::initializeCommunicator(const int nx, const int ny,
        const int nz, const bool per) {
  const int dims[3] = {nz, ny, nx};
  const int periods[3] = {static_cast<int>(per), static_cast<int>(per),
                  static_cast<int>(per)};
  // Create the cartesian communicator for MPI_COMM_WORLD
  MPI_Cart_create(MPI_COMM_WORLD, 3, dims, periods, 1, &comm);
  // Set MPI variables of the lattice
  MPI_Comm_size(comm, &(n_prc));
  MPI_Comm_rank(comm, &(my_prc));
  // Associate name to the communicator to identify it -> for debugging and
  // nicer error messages
  constexpr char lattice_comm_name[] = "Lattice";
  MPI_Comm_set_name(comm, lattice_comm_name);

  // Test if process naming is the same for both communicators
  /*
  int MYPRC;
  MPI_Comm_rank(MPI_COMM_WORLD,&MYPRC);
  cout«"\r"«my_prc«"\t"«MYPRC«endl;
  */
}

/// Construct the lattice and set the stencil order
Lattice::Lattice(const int StO) : stencilOrder(StO),
    ghostLayerWidth(StO/2+1) {
  statusFlags = 0;
}

/// Set the number of points in each dimension of the lattice
void Lattice::setDiscreteDimensions(const sunindextype _nx,
        const sunindextype _ny, const sunindextype _nz) {
  // copy the given data for number of points
  tot_nx = _nx;
  tot_ny = _ny;
  tot_nz = _nz;
  // compute the resulting number of points and datapoints
  tot_noP = tot_nx * tot_ny * tot_nz;
  tot_noDP = dataPointDimension * tot_noP;
  // compute the new Delta, the physical resolution
  dx = tot_lx / tot_nx;
  dy = tot_ly / tot_ny;
  dz = tot_lz / tot_nz;
}
```

```

/// Set the physical size of the lattice
void Lattice::setPhysicalDimensions(const sunrealtype _lx,
        const sunrealtype _ly, const sunrealtype _lz) {
  tot_lx = _lx;
  tot_ly = _ly;
  tot_lz = _lz;
  // calculate physical distance between points
  dx = tot_lx / tot_nx;
  dy = tot_ly / tot_ny;
  dz = tot_lz / tot_nz;
  statusFlags |= FLatticeDimensionSet;
}

////////////////////////////////////////////////////////////
//// Implementation of LatticePatch component functions ////
////////////////////////////////////////////////////////////

/// Construct the lattice patch
LatticePatch::LatticePatch() {
  // set default origin coordinates to (0,0,0)
  x0 = y0 = z0 = 0;
  // set default position in Lattice-Patchwork to (0,0,0)
  LIx = LIy = LIz = 0;
  // set default physical lentgth for lattice patch to (0,0,0)
  lx = ly = lz = 0;
  // set default discrete length for lattice patch to (0,1,1)
  /* This is done in this manner as even in 1D simulations require a 1 point
   * width */
  nx = 0;
  ny = nz = 1;

  // u is not initialized as it wouldn't make any sense before the dimensions
  // are set idem for the enveloping lattice

  // set default statusFlags to non set
  statusFlags = 0;
}

/// Destruct the patch and thereby destroy the NVectors
LatticePatch::~LatticePatch() {
  // Deallocate memory for solution vector
  if (statusFlags & FLatticePatchSetUp) {
    // Destroy data vectors
    N_VDestroy_Parallel(u);
    N_VDestroy_Parallel(du);
  }
}

/// Set up the patchwork
int generatePatchwork(const Lattice &envelopeLattice, LatticePatch &patchToMold,
                      const int DLx, const int DLy, const int DLz) {
  // Retrieve the ghost layer depth
  const int gLW = envelopeLattice.get_ghostLayerWidth();
  // Retrieve the data point dimension
  const int dPD = envelopeLattice.get_dataPointDimension();
  // MPI process/patch
  const int my_prc = envelopeLattice.my_prc;
  // Determine thicknes of the slice
  const sunindextype tot_NOXP = envelopeLattice.get_tot_nx(); // total points of lattice
  const sunindextype tot_NOYP = envelopeLattice.get_tot_ny();
  const sunindextype tot_NOZP = envelopeLattice.get_tot_nz();
  // position of the patch in the lattice of patches -> process associated to
  // position
  const sunindextype LIx = my_prc % DLx;
  const sunindextype LIy = (my_prc / DLx) % DLy;
  const sunindextype LIz = (my_prc / DLx) / DLy;
  // Determine the number of points in the patch and first absolute points in
  // each dimension
  const sunindextype local_NOXP = tot_NOXP / DLx;
  const sunindextype local_NOYP = tot_NOYP / DLy;
  const sunindextype local_NOZP = tot_NOZP / DLz;
  // absolute positions of the first point in each dimension
  const sunindextype firstXPoint = local_NOXP * LIx;
  const sunindextype firstYPoint = local_NOYP * LIy;
  const sunindextype firstZPoint = local_NOZP * LIz;
  // total number of points in the patch
  const sunindextype local_NODP = dPD * local_NOXP * local_NOYP * local_NOZP;

  // Set patch up with above derived quantities
  /*
  // Experiment: Resolution can be adapted for each process/patch
  const int Scaler=4; // Better hand over as parameter via `initializePatchwork'
  if(my_prc==0){patchToMold.dx=envelopeLattice.get_dx();}
  else if(my_prc==1){patchToMold.dx=envelopeLattice.get_dx()*Scaler;}
  else{errorKill("Only do this resolution barrier test with 2 processes.");}
  */
```

```
  patchToMold.dx = envelopeLattice.get_dx();
  patchToMold.dy = envelopeLattice.get_dy();
  patchToMold.dz = envelopeLattice.get_dz();
  patchToMold.x0 = firstXPoint * patchToMold.dx;
  patchToMold.y0 = firstYPoint * patchToMold.dy;
  patchToMold.z0 = firstZPoint * patchToMold.dz;
  patchToMold.LIx = LIx;
  patchToMold.LIy = LIy;
  patchToMold.LIz = LIz;
  patchToMold.nx = local_NOXP;
  patchToMold.ny = local_NOYP;
  patchToMold.nz = local_NOZP;
  patchToMold.lx = patchToMold.nx * patchToMold.dx;
  patchToMold.ly = patchToMold.ny * patchToMold.dy;
  patchToMold.lz = patchToMold.nz * patchToMold.dz;
  /* // Check name of lattice communicator
  char lattice_comm_name[MPI_MAX_OBJECT_NAME];
  int lattice_namelen;
  MPI_Comm_get_name(envelopeLattice.comm, lattice_comm_name, &lattice_namelen);
  cout«"envelopeLattice.comm gives  "« lattice_comm_name « endl;
  */
  /* Create and allocate memory for parallel vectors with defined local and
   * global lenghts *
   * (-> CVode problem sizes Nlocal and N)
   * for field data and temporal derivatives and set extra pointers to them */
  patchToMold.u =
      N_VNew_Parallel(envelopeLattice.comm, local_NODP,
                      envelopeLattice.get_tot_noDP(), envelopeLattice.sunctx);
  patchToMold.uData = NV_DATA_P(patchToMold.u);
  patchToMold.du =
      N_VNew_Parallel(envelopeLattice.comm, local_NODP,
                      envelopeLattice.get_tot_noDP(), envelopeLattice.sunctx);
  patchToMold.duData = NV_DATA_P(patchToMold.du);
  // Allocate space for auxiliary uAux so that the lattice and all possible
  // directions of ghost Layers fit
  const int s1 = patchToMold.nx, s2 = patchToMold.ny, s3 = patchToMold.nz;
  const int s_min = min(s1, min(s2, s3));
  patchToMold.uAux.resize(s1 * s2 * s3 / s_min * (s_min + 2 * gLW) * dPD);
  patchToMold.uAuxData = &patchToMold.uAux[0];
  patchToMold.envelopeLattice = &envelopeLattice;
  // Set patch "name" to process number -> only for debugging
  // patchToMold.ID=my_prc;
  // set flag
  patchToMold.statusFlags = FLatticePatchSetUp;
  patchToMold.generateTranslocationLookup();
  return 0;
}

/// Return the discrete size of the patch: number of lattice patch points in
/// specified dimension
int LatticePatch::discreteSize(int dir) const {
  switch (dir) {
  case 0:
    return nx * ny * nz;
  case 1:
    return nx;
  case 2:
    return ny;
  case 3:
    return nz;
  // case 4: return uAux.size(); // for debugging
  default:
    return -1;
  }
}

/// Return the physical origin of the patch in a dimension
sunrealtype LatticePatch::origin(const int dir) const {
  switch (dir) {
  case 1:
    return x0;
  case 2:
    return y0;
  case 3:
    return z0;
  default:
    errorKill("LatticePatch::origin function called with wrong dir parameter");
    return -1;
  }
}

/// Return the distance between points in the patch in a dimension
sunrealtype LatticePatch::getDelta(const int dir) const {
  switch (dir) {
  case 1:
    return dx;
  case 2:
```

```
      return dy;
    case 3:
      return dz;
    default:
      errorKill(
          "LatticePatch::getDelta function called with wrong dir parameter");
      return -1;
    }
}

/** 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 */
void LatticePatch::generateTranslocationLookup() {
    // Check that the lattice has been set up
    checkFlag(FLatticeDimensionSet);
    // lenghts for auxilliary layers, including ghost layers
    const int gLW = envelopeLattice->get_ghostLayerWidth();
    const int mx = nx + 2 * gLW;
    const int my = ny + 2 * gLW;
    const int mz = nz + 2 * gLW;
    // sizes for lookup vectors
    // generate u->uAux
    uTox.resize(nx * ny * nz);
    uToy.resize(nx * ny * nz);
    uToz.resize(nx * ny * nz);
    // generate uAux->u with length including halo
    xTou.resize(mx * ny * nz);
    yTou.resize(nx * my * nz);
    zTou.resize(nx * ny * mz);
    // variables for cartesian position in the 3D discrete lattice
    int px = 0, py = 0, pz = 0;
    for (unsigned int i = 0; i < uToy.size(); i++) { // loop over all points in the patch
      // calulate cartesian coordinates
      px = i % nx;
      py = (i / nx) % ny;
      pz = (i / nx) / ny;
      // fill lookups extended by halos (useful for y and z direction)
      uTox[i] = (px + gLW) + py * mx +
                pz * mx * ny; // unroll (de-flatten) cartesian dimension
      xTou[px + py * mx + pz * mx * ny] =
          i; // match cartesian point to u location
      uToy[i] = (py + gLW) + pz * my + px * my * nz;
      yTou[py + pz * my + px * my * nz] = i;
      uToz[i] = (pz + gLW) + px * mz + py * mz * nx;
      zTou[pz + px * mz + py * mz * nx] = i;
    }
    // same for ghost layer lookup tables
    lgcTox.resize(gLW * ny * nz);
    rgcTox.resize(gLW * ny * nz);
    for (unsigned int i = 0; i < lgcTox.size(); i++) {
      px = i % gLW;
      py = (i / gLW) % ny;
      pz = (i / gLW) / ny;
      lgcTox[i] = px + py * mx + pz * mx * ny;
      rgcTox[i] = px + nx + gLW + py * mx + pz * mx * ny;
    }
    lgcToy.resize(gLW * nx * nz);
    rgcToy.resize(gLW * nx * nz);
    for (unsigned int i = 0; i < lgcToy.size(); i++) {
      px = i % nx;
      py = (i / nx) % gLW;
      pz = (i / nx) / gLW;
      lgcToy[i] = py + pz * my + px * my * nz;
      rgcToy[i] = py + ny + gLW + pz * my + px * my * nz;
    }
    lgcToz.resize(gLW * nx * ny);
    rgcToz.resize(gLW * nx * ny);
    for (unsigned int i = 0; i < lgcToz.size(); i++) {
      px = i % nx;
      py = (i / nx) % ny;
      pz = (i / nx) / ny;
      lgcToz[i] = pz + px * mz + py * mz * nx;
      rgcToz[i] = pz + nz + gLW + px * mz + py * mz * nx;
    }
    statusFlags |= TranslocationLookupSetUp;
}

/** Rotate into eigenraum along R matrices of paper using below rotation
 * functions
 * -> uAuxData gets the rotated left-halo-, inner-patch-, right-halo-data */
void LatticePatch::rotateIntoEigen(const int dir) {
    // Check that the lattice, ghost layers as well as the translocation lookups
    // have been set up;
    checkFlag(FLatticePatchSetUp);
    checkFlag(TranslocationLookupSetUp);
    checkFlag(GhostLayersInitialized); // this check is only after call to
```

```
                                    // exchange ghost cells
  switch (dir) {
  case 1:
    rotateToX(uAuxData, gCLData, lgcTox);
    rotateToX(uAuxData, uData, uTox);
    rotateToX(uAuxData, gCRData, rgcTox);
    break;
  case 2:
    rotateToY(uAuxData, gCLData, lgcToy);
    rotateToY(uAuxData, uData, uToy);
    rotateToY(uAuxData, gCRData, rgcToy);
    break;
  case 3:
    rotateToZ(uAuxData, gCLData, lgcToz);
    rotateToZ(uAuxData, uData, uToz);
    rotateToZ(uAuxData, gCRData, rgcToz);
    break;
  default:
    errorKill("Tried to rotate into the wrong direction");
    break;
  }
}

/// Same as `rotateIntoEigen` but for neighborhood 3D halo buffers
void LatticePatch::rotateIntoEigen3D() {
  checkFlag(FLatticePatchSetUp);
  checkFlag(TranslocationLookupSetUp);
  checkFlag(GhostLayersInitialized);
  rotateToX(uAuxData, gCLData, lgcTox);
  rotateToX(uAuxData, uData, uTox);
  rotateToX(uAuxData, gCRData, rgcTox);
  rotateToY(uAuxData, gCBData, lgcToy);
  rotateToY(uAuxData, uData, uToy);
  rotateToY(uAuxData, gCTData, rgcToy);
  rotateToZ(uAuxData, gCFData, lgcToz);
  rotateToZ(uAuxData, uData, uToz);
  rotateToZ(uAuxData, gCAData, rgcToz);
}

/// Rotate halo and inner-patch data vectors with rotation matrix Rx into
/// eigenspace of Z matrix and write to auxiliary vector
inline void LatticePatch::rotateToX(sunrealtype *outArray,
                                    const sunrealtype *inArray,
                                    const vector<int> &lookup) {
  int ii = 0, target = 0;
#pragma ivdep
#pragma omp simd // safelen(6) - also good
#pragma distribute_point
  for (unsigned int i = 0; i < lookup.size(); i++) {
    // get correct u-vector and spatial indices along previously defined lookup
    // tables
    target = envelopeLattice->get_dataPointDimension() * lookup[i];
    ii = envelopeLattice->get_dataPointDimension() * i;
    outArray[target + 0] = -inArray[1 + ii] + inArray[5 + ii];
    outArray[target + 1] = inArray[2 + ii] + inArray[4 + ii];
    outArray[target + 2] = inArray[1 + ii] + inArray[5 + ii];
    outArray[target + 3] = -inArray[2 + ii] + inArray[4 + ii];
    outArray[target + 4] = inArray[3 + ii];
    outArray[target + 5] = inArray[ii];
  }
}

/// Rotate halo and inner-patch data vectors with rotation matrix Ry into
/// eigenspace of Z matrix and write to auxiliary vector
inline void LatticePatch::rotateToY(sunrealtype *outArray,
                                    const sunrealtype *inArray,
                                    const vector<int> &lookup) {
  int ii = 0, target = 0;
#pragma ivdep
#pragma omp simd // safelen(6)
#pragma distribute_point
  for (unsigned int i = 0; i < lookup.size(); i++) {
    target = envelopeLattice->get_dataPointDimension() * lookup[i];
    ii = envelopeLattice->get_dataPointDimension() * i;
    outArray[target + 0] = inArray[ii] + inArray[5 + ii];
    outArray[target + 1] = -inArray[2 + ii] + inArray[3 + ii];
    outArray[target + 2] = -inArray[ii] + inArray[5 + ii];
    outArray[target + 3] = inArray[2 + ii] + inArray[3 + ii];
    outArray[target + 4] = inArray[4 + ii];
    outArray[target + 5] = inArray[1 + ii];
  }
}

/// Rotate halo and inner-patch data vectors with rotation matrix Rz into
/// eigenspace of Z matrix and write to auxiliary vector
inline void LatticePatch::rotateToZ(sunrealtype *outArray,
                                    const sunrealtype *inArray,
```

```
                                     const vector<int> &lookup) {
  int ii = 0, target = 0;
#pragma ivdep
#pragma omp simd // safelen(6)
#pragma distribute_point
  for (unsigned int i = 0; i < lookup.size(); i++) {
    target = envelopeLattice->get_dataPointDimension() * lookup[i];
    ii = envelopeLattice->get_dataPointDimension() * i;
    outArray[target + 0] = -inArray[ii] + inArray[4 + ii];
    outArray[target + 1] = inArray[1 + ii] + inArray[3 + ii];
    outArray[target + 2] = inArray[ii] + inArray[4 + ii];
    outArray[target + 3] = -inArray[1 + ii] + inArray[3 + ii];
    outArray[target + 4] = inArray[5 + ii];
    outArray[target + 5] = inArray[2 + ii];
  }
}

/// Derotate uAux with transposed rotation matrices and write to derivative
/// buffer -- normalization is done here by the factor 1/2
void LatticePatch::derotate(int dir, sunrealtype *buffOut) {
  // Check that the lattice as well as the translocation lookups have been set
  // up;
  checkFlag(FLatticePatchSetUp);
  checkFlag(TranslocationLookupSetUp);
  const int dPD = envelopeLattice->get_dataPointDimension();
  const int gLW = envelopeLattice->get_ghostLayerWidth();
  const int uSize = discreteSize();
  int ii = 0, target = 0;
  switch (dir) {
  case 1:
#pragma ivdep
#pragma omp simd // safelen(6) - also good
#pragma distribute_point
    for (int i = 0; i < uSize; i++) {
      // get correct indices in u and rotation space
      target = dPD * i;
      ii = dPD * (uTox[i] - gLW);
      buffOut[target + 0] = uAux[5 + ii];
      buffOut[target + 1] = (-uAux[ii] + uAux[2 + ii]) / 2.;
      buffOut[target + 2] = (uAux[1 + ii] - uAux[3 + ii]) / 2.;
      buffOut[target + 3] = uAux[4 + ii];
      buffOut[target + 4] = (uAux[1 + ii] + uAux[3 + ii]) / 2.;
      buffOut[target + 5] = (uAux[ii] + uAux[2 + ii]) / 2.;
    }
    break;
  case 2:
#pragma omp simd // safelen(6)
#pragma distribute_point
    for (int i = 0; i < uSize; i++) {
      target = dPD * i;
      ii = dPD * (uToy[i] - gLW);
      buffOut[target + 0] = (uAux[ii] - uAux[2 + ii]) / 2.;
      buffOut[target + 1] = uAux[5 + ii];
      buffOut[target + 2] = (-uAux[1 + ii] + uAux[3 + ii]) / 2.;
      buffOut[target + 3] = (uAux[1 + ii] + uAux[3 + ii]) / 2.;
      buffOut[target + 4] = uAux[4 + ii];
      buffOut[target + 5] = (uAux[ii] + uAux[2 + ii]) / 2.;
    }
    break;
  case 3:
#pragma omp simd // safelen(6)
#pragma distribute_point
    for (int i = 0; i < uSize; i++) {
      target = dPD * i;
      ii = dPD * (uToz[i] - gLW);
      buffOut[target + 0] = (-uAux[ii] + uAux[2 + ii]) / 2.;
      buffOut[target + 1] = (uAux[1 + ii] - uAux[3 + ii]) / 2.;
      buffOut[target + 2] = uAux[5 + ii];
      buffOut[target + 3] = (uAux[1 + ii] + uAux[3 + ii]) / 2.;
      buffOut[target + 4] = (uAux[ii] + uAux[2 + ii]) / 2.;
      buffOut[target + 5] = uAux[4 + ii];
    }
    break;
  default:
    errorKill("Tried to derotate from the wrong direction");
    break;
  }
}

/// Create buffers to save derivative values, optimizing computational load
void LatticePatch::initializeBuffers() {
  // Check that the lattice has been set up
  checkFlag(FLatticeDimensionSet);
  const int dPD = envelopeLattice->get_dataPointDimension();
  buffX.resize(nx * ny * nz * dPD);
  buffY.resize(nx * ny * nz * dPD);
  buffZ.resize(nx * ny * nz * dPD);
```

```
  // Set pointers used for propagation functions
  buffData[0] = &buffX[0];
  buffData[1] = &buffY[0];
  buffData[2] = &buffZ[0];
  statusFlags |= BuffersInitialized;
}

/// Perform the ghost cell exchange in a specified direction
void LatticePatch::exchangeGhostCells(const int dir) {
  // Check that the lattice has been set up
  checkFlag(FLatticeDimensionSet);
  checkFlag(FLatticePatchSetUp);
  // Variables to per dimension calculate the halo indices, and distance to
  // other side halo boundary
  int mx = 1, my = 1, mz = 1, distToRight = 1;
  const int gLW = envelopeLattice->get_ghostLayerWidth();
  // In the chosen direction m is set to ghost layer width while the others
  // remain to form the plane
  switch (dir) {
  case 1:
    mx = gLW;
    my = ny;
    mz = nz;
    distToRight = (nx - gLW);
    break;
  case 2:
    mx = nx;
    my = gLW;
    mz = nz;
    distToRight = nx * (ny - gLW);
    break;
  case 3:
    mx = nx;
    my = ny;
    mz = gLW;
    distToRight = nx * ny * (nz - gLW);
    break;
  }
  // total number of exchanged points
  const int dPD = envelopeLattice->get_dataPointDimension();
  const int exchangeSize = mx * my * mz * dPD;
  // provide size of the halos for ghost cells
  ghostCellLeft.resize(exchangeSize);
  ghostCellRight.resize(ghostCellLeft.size());
  ghostCellLeftToSend.resize(ghostCellLeft.size());
  ghostCellRightToSend.resize(ghostCellLeft.size());
  gCLData = &ghostCellLeft[0];
  gCRData = &ghostCellRight[0];
  statusFlags |= GhostLayersInitialized;

  // Initialize running index li for the halo buffers, and index ui of uData for
  // data transfer
  int li = 0, ui = 0;

// #pragma omp parallel for reduction(+:ui) reduction(+:li) -> don't probably
// bad idea to parallelize ghost cell exchange Loop over to be copied points in
// z and y direction
#pragma distribute_point
  for (int iz = 0; iz < mz; iz++) {
    for (int iy = 0; iy < my; iy++) {
      // uData vector start index of halo data to be transferred
      // with each z-step add the whole xy-plane and with y-step the x-range ->
      // iterate all x-ranges
      ui = (iz * nx * ny + iy * nx) * dPD;
      // copy left halo data from uData to buffer, transfer size is given by
      // x-length (not x-range) perhaps faster but more fragile C lib copy
      // operation (contained in cstring header)
      /*
      memcpy(&ghostCellLeftToSend[li],
             &uData[ui],
             sizeof(sunrealtype)*mx*dPD);
      // increase ui by the distance to vis-a-vis boundary and copy right halo
      data to buffer ui+=distToRight*dPD; memcpy(&ghostCellRightToSend[li],
             &uData[ui],
             sizeof(sunrealtype)*mx*dPD);
      */
      // perhaps more safe but slower copy operation (contained in algorithm
      // header) performance highly system dependent
      copy(&uData[ui], &uData[ui + mx * dPD], &ghostCellLeftToSend[li]);
      ui += distToRight * dPD;
      copy(&uData[ui], &uData[ui + mx * dPD], &ghostCellRightToSend[li]);

      // increase halo index by transferred items per y-iteration step
      // (x-length)
      li += mx * dPD;
    }
  }
```

```

  /* Send and receive the data to and from neighboring latticePatches */
  // Adjust direction to cartesian communicator
  int dim = 2; // default for dir==1
  if (dir == 2) {
    dim = 1;
  } else if (dir == 3) {
    dim = 0;
  }
  int rank_source = 0, rank_dest = 0;
  MPI_Cart_shift(envelopeLattice->comm, dim, -1, &rank_source,
                 &rank_dest); // s.t. rank_dest is left & v.v.

  // nonblocking Isend/Irecv

  MPI_Request requests[4];
  MPI_Irecv(&ghostCellRight[0], exchangeSize, MPI_SUNREALTYPE, rank_source, 1,
  envelopeLattice->comm, &requests[0]);
  MPI_Isend(&ghostCellLeftToSend[0], exchangeSize, MPI_SUNREALTYPE, rank_dest,
  1, envelopeLattice->comm, &requests[1]);
  MPI_Irecv(&ghostCellLeft[0], exchangeSize, MPI_SUNREALTYPE, rank_dest, 2,
  envelopeLattice->comm, &requests[2]);
  MPI_Isend(&ghostCellRightToSend[0], exchangeSize, MPI_SUNREALTYPE,
  rank_source, 2, envelopeLattice->comm, &requests[3]);
  MPI_Waitall(4, requests, MPI_STATUS_IGNORE);


  // blocking Sendrecv:
  /*
  MPI_Sendrecv(&ghostCellLeftToSend[0], exchangeSize, MPI_SUNREALTYPE,
               rank_dest, 1, &ghostCellRight[0], exchangeSize, MPI_SUNREALTYPE,
               rank_source, 1, envelopeLattice->comm, MPI_STATUS_IGNORE);
  MPI_Sendrecv(&ghostCellRightToSend[0], exchangeSize, MPI_SUNREALTYPE,
               rank_source, 2, &ghostCellLeft[0], exchangeSize, MPI_SUNREALTYPE,
               rank_dest, 2, envelopeLattice->comm, MPI_STATUS_IGNORE);
  */
}

/// Exchange ghost cells with a neighborhood collective operation
void LatticePatch::exchangeGhostCells3D() {
  // Check that the lattice has been set up
  checkFlag(FLatticeDimensionSet);
  // ghostlayerwidth
  const int gLW = envelopeLattice->get_ghostLayerWidth();
  // datapoint dimension
  const int dPD = envelopeLattice->get_dataPointDimension();
  // total number of exchanged points per halo
  const int n = nx; // only cubic patches allowed -> use general length n
  const int exchangeSize = n * n * gLW * dPD;
  // Give ghostCells the total size of the ghost layers (the six halos)
  const int tot_exchangeSize = 6 * exchangeSize;
  ghostCells.resize(tot_exchangeSize);
  ghostCellsToSend.resize(ghostCells.size());
  // ghost cell data in all directions: left,right,bottom,top,front,abaft; but
  // with MPI dim order "lefts" are the first receivers and "rights" are the
  // first senders -> see buffer creator below
  gCFData = &ghostCells[0];
  gCAData = &ghostCells[exchangeSize];
  gCBData = &ghostCells[2 * exchangeSize];
  gCTData = &ghostCells[3 * exchangeSize];
  gCLData = &ghostCells[4 * exchangeSize];
  gCRData = &ghostCells[5 * exchangeSize];
  statusFlags |= GhostLayersInitialized;

  checkFlag(FLatticePatchSetUp);
  // variables to set to ghost layer width and point distance to next
  // communication point -> depends on direction

  int li = 0;          // running index for buffers
  int distToRight = 0; // distance to vis-a-vis halo data, varies per dim
  // filling buffers along the MPI dim order
  distToRight = n * n * (n - gLW);
  bufferCreator(li, n, n, gLW, distToRight);
  li += 2 * exchangeSize; // li increases by two exchange sizes per dim

  distToRight = n * (n - gLW);
  bufferCreator(li, n, gLW, n, distToRight);
  li += 2 * exchangeSize;

  distToRight = (n - gLW);
  bufferCreator(li, gLW, n, n, distToRight);

  MPI_Neighbor_alltoall(&ghostCellsToSend[0], exchangeSize, MPI_SUNREALTYPE,
                        &ghostCells[0], exchangeSize, MPI_SUNREALTYPE,
                        envelopeLattice->comm);
}

```

```
/// Fill the halo buffers for neighborhood collectives
void LatticePatch::bufferCreator(int li, int mx, int my, int mz,
        int distToRight) {
  const int dPD = envelopeLattice->get_dataPointDimension();
  // Initialize running index ui for to be transferred uData
  int ui = 0;

// #pragma omp parallel for reduction(+:ui) reduction(+:li) -> don't, probably
// bad idea to parallelize ghost cell exchange Loop over all planes and pick to
// be transferred points (uData indices)
#pragma distribute_point
  for (int iz = 0; iz < mz; iz++) {
    for (int iy = 0; iy < my; iy++) {
      // start index of uData vector halo data to be transferred
      // Here, in contrast to above, start at right boundary to send to left
      // s.t. updated left values are the first indices (bec of neighborhood
      // collective pattern) with each z-step add the whole xy-plane and with
      // y-step the x-range -> iterate all x-ranges
      ui = (iz * nx * ny + iy * nx + distToRight) * dPD;
      // copy from uData from right boundary (at each dimension) into buffer,
      // halo transfer size is given by x length at each step
      // memcpy(&ghostCellsToSend[li], \
                   &uData[ui], \
                   sizeof(sunrealtype)*mx*dPD);
      copy(&uData[ui], &uData[ui + mx * dPD], &ghostCellsToSend[li]);
      // increase li by transferred indices in this loop-step
      li += mx * dPD;
    }
  }

#pragma distribute_point
  for (int iz = 0; iz < mz; iz++) {
    for (int iy = 0; iy < my; iy++) {
      // Now copy from left boundary into buffer
      ui = (iz * nx * ny + iy * nx) * dPD;
      //memcpy(&ghostCellsToSend[li], \
                   &uData[ui], \
                   sizeof(sunrealtype)*mx*dPD);
      copy(&uData[ui], &uData[ui + mx * dPD], &ghostCellsToSend[li]);
      li += mx * dPD;
    }
  }
}

/// Check if all flags are set
void LatticePatch::checkFlag(unsigned int flag) const {
  if (!(statusFlags & flag)) {
    string errorMessage;
    switch (flag) {
    case FLatticePatchSetUp:
      errorMessage = "The Lattice patch was not set up please make sure to "
                     "initilize a Lattice topology";
      break;
    case TranslocationLookupSetUp:
      errorMessage = "The translocation lookup tables have not been generated, "
                     "please be sure to run generateTranslocationLookup()";
      break;
    case GhostLayersInitialized:
      errorMessage = "The space for the ghost layers has not been allocated, "
                     "please be sure to run initializeGhostLayer()";
      break;
    case BuffersInitialized:
      errorMessage = "The space for the buffers has not been allocated, please "
                     "be sure to run initializeBuffers()";
      break;
    default:
      errorMessage = "Uppss, you've made a non-standard error, sadly I can't "
                     "help you there";
      break;
    }
    errorKill(errorMessage);
  }
  return;
}

/// Calculate derivatives in the patch (uAux) in the specified direction
void LatticePatch::derive(const int dir) {
  // ghost layer width
  const int gLW = envelopeLattice->get_ghostLayerWidth();
  // dimensionality of data points -> 6
  const int dPD = envelopeLattice->get_dataPointDimension();
  // total width of patch in given direction including ghost layers at ends
  const int dirWidth = discreteSize(dir) + 2 * gLW;
  // width of patch only in given direction
  const int dirWidthO = discreteSize(dir);
  // size of plane perpendicular to given dimension
  const int perpPlainSize = discreteSize() / discreteSize(dir);
```

```
// physical distance between points in that direction
sunrealtype dxi = NAN;
switch (dir) {
case 1:
  dxi = dx;
  break;
case 2:
  dxi = dy;
  break;
case 3:
  dxi = dz;
  break;
default:
  dxi = 1;
  errorKill("Tried to derive in the wrong direction");
  break;
}
// Derive according to chosen stencil accuracy order (which determines also
// gLW)
const int order = envelopeLattice->get_stencilOrder();
switch (order) {
case 1:
  for (int i = 0; i < perpPlainSize; i++) {
    for (int j = (i * dirWidth + gLW) * dPD;
         j < (i * dirWidth + gLW + dirWidthO) * dPD; j += dPD) {
      uAux[j + 0 - gLW * dPD] = s1b(&uAux[j + 0]) / dxi;
      uAux[j + 1 - gLW * dPD] = s1b(&uAux[j + 1]) / dxi;
      uAux[j + 2 - gLW * dPD] = s1f(&uAux[j + 2]) / dxi;
      uAux[j + 3 - gLW * dPD] = s1f(&uAux[j + 3]) / dxi;
      uAux[j + 4 - gLW * dPD] = s1f(&uAux[j + 4]) / dxi;
      uAux[j + 5 - gLW * dPD] = s1f(&uAux[j + 5]) / dxi;
    }
  }
  break;
case 2:
  for (int i = 0; i < perpPlainSize; i++) {
    for (int j = (i * dirWidth + gLW) * dPD;
         j < (i * dirWidth + gLW + dirWidthO) * dPD; j += dPD) {
      uAux[j + 0 - gLW * dPD] = s2b(&uAux[j + 0]) / dxi;
      uAux[j + 1 - gLW * dPD] = s2b(&uAux[j + 1]) / dxi;
      uAux[j + 2 - gLW * dPD] = s2f(&uAux[j + 2]) / dxi;
      uAux[j + 3 - gLW * dPD] = s2f(&uAux[j + 3]) / dxi;
      uAux[j + 4 - gLW * dPD] = s2c(&uAux[j + 4]) / dxi;
      uAux[j + 5 - gLW * dPD] = s2c(&uAux[j + 5]) / dxi;
    }
  }
  break;
case 3:
  for (int i = 0; i < perpPlainSize; i++) {
    for (int j = (i * dirWidth + gLW) * dPD;
         j < (i * dirWidth + gLW + dirWidthO) * dPD; j += dPD) {
      uAux[j + 0 - gLW * dPD] = s3b(&uAux[j + 0]) / dxi;
      uAux[j + 1 - gLW * dPD] = s3b(&uAux[j + 1]) / dxi;
      uAux[j + 2 - gLW * dPD] = s3f(&uAux[j + 2]) / dxi;
      uAux[j + 3 - gLW * dPD] = s3f(&uAux[j + 3]) / dxi;
      uAux[j + 4 - gLW * dPD] = s3f(&uAux[j + 4]) / dxi;
      uAux[j + 5 - gLW * dPD] = s3f(&uAux[j + 5]) / dxi;
    }
  }
  break;
case 4:
  for (int i = 0; i < perpPlainSize; i++) {
    for (int j = (i * dirWidth + gLW) * dPD;
         j < (i * dirWidth + gLW + dirWidthO) * dPD; j += dPD) {
      uAux[j + 0 - gLW * dPD] = s4b(&uAux[j + 0]) / dxi;
      uAux[j + 1 - gLW * dPD] = s4b(&uAux[j + 1]) / dxi;
      uAux[j + 2 - gLW * dPD] = s4f(&uAux[j + 2]) / dxi;
      uAux[j + 3 - gLW * dPD] = s4f(&uAux[j + 3]) / dxi;
      uAux[j + 4 - gLW * dPD] = s4c(&uAux[j + 4]) / dxi;
      uAux[j + 5 - gLW * dPD] = s4c(&uAux[j + 5]) / dxi;
    }
  }
  break;
case 5:
  for (int i = 0; i < perpPlainSize; i++) {
    for (int j = (i * dirWidth + gLW) * dPD;
         j < (i * dirWidth + gLW + dirWidthO) * dPD; j += dPD) {
      uAux[j + 0 - gLW * dPD] = s5b(&uAux[j + 0]) / dxi;
      uAux[j + 1 - gLW * dPD] = s5b(&uAux[j + 1]) / dxi;
      uAux[j + 2 - gLW * dPD] = s5f(&uAux[j + 2]) / dxi;
      uAux[j + 3 - gLW * dPD] = s5f(&uAux[j + 3]) / dxi;
      uAux[j + 4 - gLW * dPD] = s5f(&uAux[j + 4]) / dxi;
      uAux[j + 5 - gLW * dPD] = s5f(&uAux[j + 5]) / dxi;
    }
  }
  break;
case 6:
```

```
      for (int i = 0; i < perpPlainSize; i++) {
        for (int j = (i * dirWidth + gLW) * dPD;
             j < (i * dirWidth + gLW + dirWidthO) * dPD; j += dPD) {
          uAux[j + 0 - gLW * dPD] = s6b(&uAux[j + 0]) / dxi;
          uAux[j + 1 - gLW * dPD] = s6b(&uAux[j + 1]) / dxi;
          uAux[j + 2 - gLW * dPD] = s6f(&uAux[j + 2]) / dxi;
          uAux[j + 3 - gLW * dPD] = s6f(&uAux[j + 3]) / dxi;
          uAux[j + 4 - gLW * dPD] = s6c(&uAux[j + 4]) / dxi;
          uAux[j + 5 - gLW * dPD] = s6c(&uAux[j + 5]) / dxi;
        }
      }
      break;
    case 7:
      for (int i = 0; i < perpPlainSize; i++) {
        for (int j = (i * dirWidth + gLW) * dPD;
             j < (i * dirWidth + gLW + dirWidthO) * dPD; j += dPD) {
          uAux[j + 0 - gLW * dPD] = s7b(&uAux[j + 0]) / dxi;
          uAux[j + 1 - gLW * dPD] = s7b(&uAux[j + 1]) / dxi;
          uAux[j + 2 - gLW * dPD] = s7f(&uAux[j + 2]) / dxi;
          uAux[j + 3 - gLW * dPD] = s7f(&uAux[j + 3]) / dxi;
          uAux[j + 4 - gLW * dPD] = s7f(&uAux[j + 4]) / dxi;
          uAux[j + 5 - gLW * dPD] = s7f(&uAux[j + 5]) / dxi;
        }
      }
      break;
    case 8:
      for (int i = 0; i < perpPlainSize; i++) {
        for (int j = (i * dirWidth + gLW) * dPD;
             j < (i * dirWidth + gLW + dirWidthO) * dPD; j += dPD) {
          uAux[j + 0 - gLW * dPD] = s8b(&uAux[j + 0]) / dxi;
          uAux[j + 1 - gLW * dPD] = s8b(&uAux[j + 1]) / dxi;
          uAux[j + 2 - gLW * dPD] = s8f(&uAux[j + 2]) / dxi;
          uAux[j + 3 - gLW * dPD] = s8f(&uAux[j + 3]) / dxi;
          uAux[j + 4 - gLW * dPD] = s8c(&uAux[j + 4]) / dxi;
          uAux[j + 5 - gLW * dPD] = s8c(&uAux[j + 5]) / dxi;
        }
      }
      break;
    case 9:
      for (int i = 0; i < perpPlainSize; i++) {
        for (int j = (i * dirWidth + gLW) * dPD;
             j < (i * dirWidth + gLW + dirWidthO) * dPD; j += dPD) {
          uAux[j + 0 - gLW * dPD] = s9b(&uAux[j + 0]) / dxi;
          uAux[j + 1 - gLW * dPD] = s9b(&uAux[j + 1]) / dxi;
          uAux[j + 2 - gLW * dPD] = s9f(&uAux[j + 2]) / dxi;
          uAux[j + 3 - gLW * dPD] = s9f(&uAux[j + 3]) / dxi;
          uAux[j + 4 - gLW * dPD] = s9f(&uAux[j + 4]) / dxi;
          uAux[j + 5 - gLW * dPD] = s9f(&uAux[j + 5]) / dxi;
        }
      }
      break;
    case 10:
      for (int i = 0; i < perpPlainSize; i++) {
        for (int j = (i * dirWidth + gLW) * dPD;
             j < (i * dirWidth + gLW + dirWidthO) * dPD; j += dPD) {
          uAux[j + 0 - gLW * dPD] = s10b(&uAux[j + 0]) / dxi;
          uAux[j + 1 - gLW * dPD] = s10b(&uAux[j + 1]) / dxi;
          uAux[j + 2 - gLW * dPD] = s10f(&uAux[j + 2]) / dxi;
          uAux[j + 3 - gLW * dPD] = s10f(&uAux[j + 3]) / dxi;
          uAux[j + 4 - gLW * dPD] = s10c(&uAux[j + 4]) / dxi;
          uAux[j + 5 - gLW * dPD] = s10c(&uAux[j + 5]) / dxi;
        }
      }
      break;
    case 11:
      for (int i = 0; i < perpPlainSize; i++) {
        for (int j = (i * dirWidth + gLW) * dPD;
             j < (i * dirWidth + gLW + dirWidthO) * dPD; j += dPD) {
          uAux[j + 0 - gLW * dPD] = s11b(&uAux[j + 0]) / dxi;
          uAux[j + 1 - gLW * dPD] = s11b(&uAux[j + 1]) / dxi;
          uAux[j + 2 - gLW * dPD] = s11f(&uAux[j + 2]) / dxi;
          uAux[j + 3 - gLW * dPD] = s11f(&uAux[j + 3]) / dxi;
          uAux[j + 4 - gLW * dPD] = s11f(&uAux[j + 4]) / dxi;
          uAux[j + 5 - gLW * dPD] = s11f(&uAux[j + 5]) / dxi;
        }
      }
      break;
    case 12:
      for (int i = 0; i < perpPlainSize; i++) {
        for (int j = (i * dirWidth + gLW) * dPD;
             j < (i * dirWidth + gLW + dirWidthO) * dPD; j += dPD) {
          uAux[j + 0 - gLW * dPD] = s12b(&uAux[j + 0]) / dxi;
          uAux[j + 1 - gLW * dPD] = s12b(&uAux[j + 1]) / dxi;
          uAux[j + 2 - gLW * dPD] = s12f(&uAux[j + 2]) / dxi;
          uAux[j + 3 - gLW * dPD] = s12f(&uAux[j + 3]) / dxi;
          uAux[j + 4 - gLW * dPD] = s12c(&uAux[j + 4]) / dxi;
          uAux[j + 5 - gLW * dPD] = s12c(&uAux[j + 5]) / dxi;
```

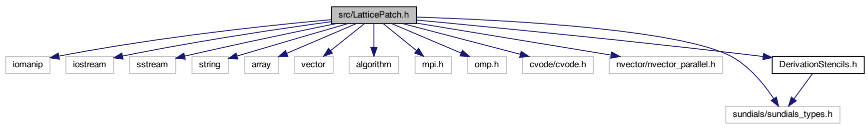
```
      }
    }
    break;
  case 13:
    //#pragma omp parallel for default(none) firstprivate(uAux)
    //shared(dxi,dirWidth,dirWidth0,gLW,dPD) collapse(2) schedule(static,6)
    //#pragma ivdep
    //#pragma distribute_point -> No.
    //#pragma unroll_and_jam
    // Iterate through all points in the plane perpendicular to the given
    // direction
    for (int i = 0; i < perpPlainSize; i++) {
      // stencil functions range over 2*gLW+6 indices, attention to cache-line
      // false-sharing
      //#pragma omp simd safelen(2*gLW+dPD)
      // Iterate through the direction for each perpendicular plane point
      for (int j = (i * dirWidth + gLW /*to shift left by gLW below */) * dPD;
           j < (i * dirWidth + gLW + dirWidth0) * dPD; j += dPD) {
        /* Compute the stencil derivative for any of the six field components
         * with a ghostlayer width adjusted to the order of the finite
         * difference scheme */
        uAux[j + 0 - gLW * dPD] = s13b(&uAux[j + 0]) / dxi;
        uAux[j + 1 - gLW * dPD] = s13b(&uAux[j + 1]) / dxi;
        uAux[j + 2 - gLW * dPD] = s13f(&uAux[j + 2]) / dxi;
        uAux[j + 3 - gLW * dPD] = s13f(&uAux[j + 3]) / dxi;
        uAux[j + 4 - gLW * dPD] = s13f(&uAux[j + 4]) / dxi;
        uAux[j + 5 - gLW * dPD] = s13f(&uAux[j + 5]) / dxi;
      }
    }
    break;

  default:
    errorKill("Please set an existing stencil order");
    break;
  }
}

///////// Helper functions /////////

/// Print a specific error message to stdout
void errorKill(const string & errorMessage) {
  cerr « endl « "Error: " « errorMessage « " Aborting..." « endl;
  MPI_Abort(MPI_COMM_WORLD, 1);
  return;
}

/** Check function return value. From CVode examples.
     opt == 0 means SUNDIALS function allocates memory so check if
              returned NULL pointer
     opt == 1 means SUNDIALS function returns an integer value so check if
              retval < 0
     opt == 2 means function allocates memory so check if returned
              NULL pointer */
int check_retval(void *returnvalue, const char *funcname, int opt, int id) {
  int *retval = nullptr;

  /* Check if SUNDIALS function returned NULL pointer - no memory allocated */
  if (opt == 0 && returnvalue == nullptr) {
    fprintf(stderr,
            "\nSUNDIALS_ERROR(%d): %s() failed - returned NULL pointer\n\n", id,
            funcname);
    return (1);
  }

  /* Check if retval < 0 */
  else if (opt == 1) {
    retval = (int *)returnvalue;
    if (*retval < 0) {
      fprintf(stderr, "\nSUNDIALS_ERROR(%d): %s() failed with retval = %d\n\n",
              id, funcname, *retval);
      return (1);
    }
  }

  /* Check if function returned NULL pointer - no memory allocated */
  else if (opt == 2 && returnvalue == nullptr) {
    fprintf(stderr,
            "\nMEMORY_ERROR(%d): %s() failed - returned NULL pointer\n\n", id,
            funcname);
    return (1);
  }

  return (0);
}
```

6.12 src/LatticePatch.h File Reference

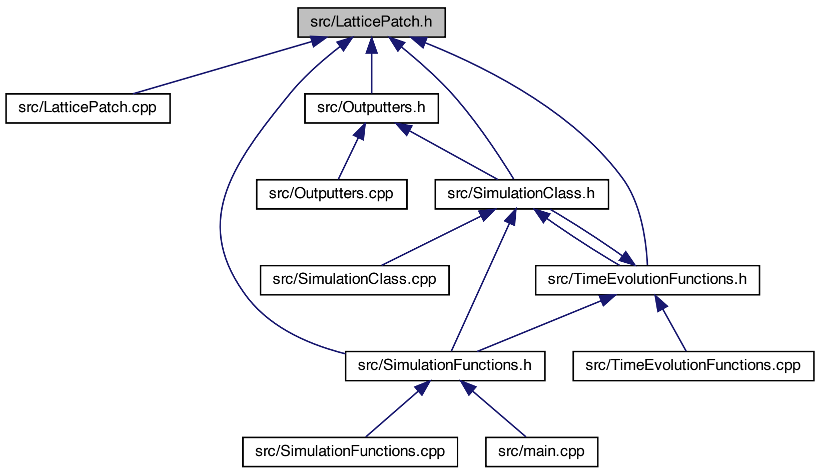
Declaration of the lattice and lattice patches.

```
#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 )
```

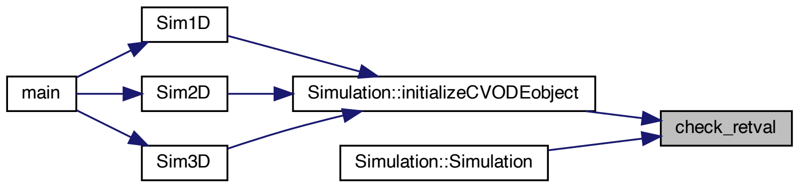
Check function return value. From CVode examples. opt == 0 means SUNDIALS function allocates memory so check if returned NULL pointer opt == 1 means SUNDIALS function returns an integer value so check if retval $<$ 0 opt == 2 means function allocates memory so check if returned NULL pointer

Definition at line 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 )
```

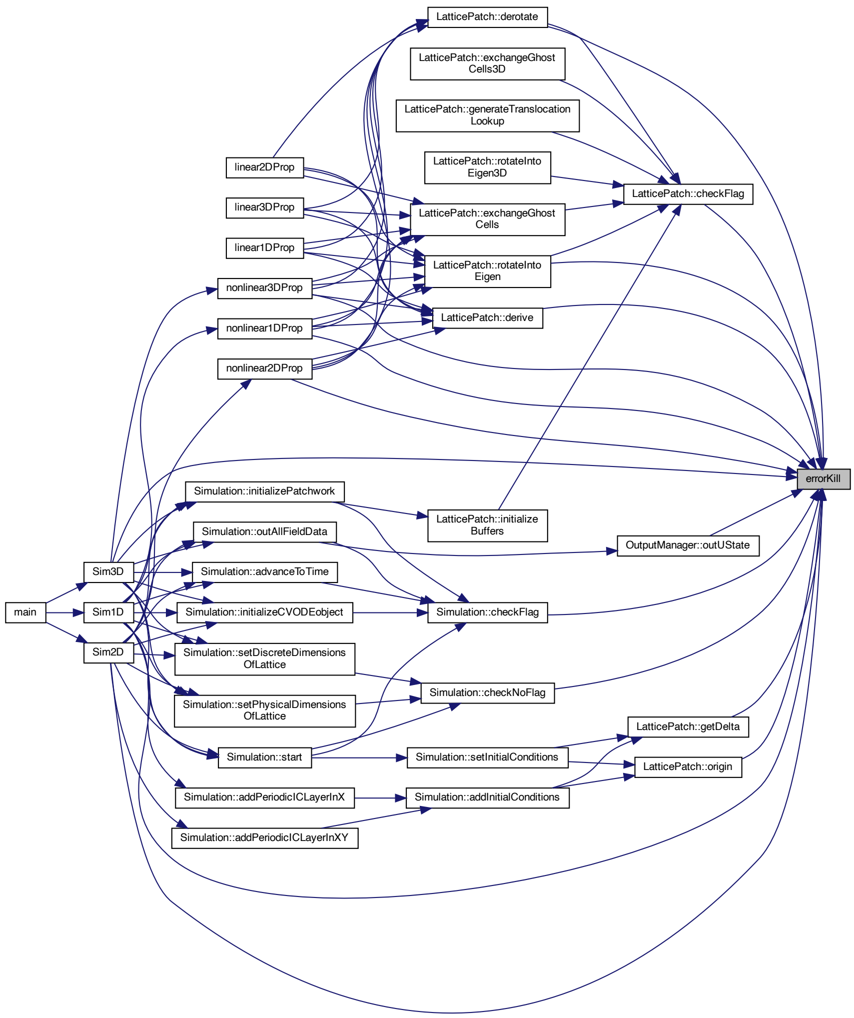
Print a specific error message to stdout.

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.

```
////////////////////////////////////////////////////////////////////////
/// @file LatticePatch.h
/// @brief Declaration of the lattice and lattice patches
////////////////////////////////////////////////////////////////////////

#pragma once

// IO
#include <iomanip>
#include <iostream>
#include <sstream>

// string, container, algorithm
#include <string>
//#include <string_view>
#include <array>
#include <vector>
#include <algorithm>

// MPI & OpenMP
#include <mpi.h>
#include <omp.h>
```

```
// Sundials
#include <cvode/cvode.h>                /* prototypes for CVODE fcts. */
#include <nvector/nvector_parallel.h> /* definition of N_Vector and macros */
#include <sundials/sundials_types.h>  /* definition of type sunrealtype */

// stencils
#include "DerivationStencils.h"

using namespace std;

// lattice construction checking flags
enum LatticeOptions {
    FLatticeDimensionSet = 0x01,  // 1
    /*OPT_B = 0x02,  // 2
    OPT_C = 0x04,  // 4
    OPT_D = 0x08,  // 8
    OPT_E = 0x10,  // 16
    OPT_F = 0x20,*/  // 32
};

/** @brief Lattice class for the construction of the enveloping discrete
 * simulation space */
class Lattice {
private:
  /// physical size of the lattice in x-direction
  sunrealtype tot_lx;
  /// physical size of the lattice in y-direction
  sunrealtype tot_ly;
  /// physical size of the lattice in z-direction
  sunrealtype tot_lz;
  /// number of points in x-direction
  sunindextype tot_nx;
  /// number of points in y-direction
  sunindextype tot_ny;
  /// number of points in z-direction
  sunindextype tot_nz;
  /// total number of lattice points
  sunindextype tot_noP;
  /// dimension of each data point -> set once and for all
  static constexpr int dataPointDimension = 6;
  /// number of lattice points times data dimension of each point
  sunindextype tot_noDP;
  /// physical distance between lattice points in x-direction
  sunrealtype dx;
  /// physical distance between lattice points in y-direction
  sunrealtype dy;
  /// physical distance between lattice points in z-direction
  sunrealtype dz;
  /// stencil order
  const int stencilOrder;
  /// required width of ghost layers (depends on the stencil order)
  const int ghostLayerWidth;
  /// char for checking if lattice flags are set
  unsigned char statusFlags;

public:
  /// number of MPI processes
  int n_prc;
  /// number of MPI process
  int my_prc;
  /// personal communicator of the lattice
  MPI_Comm comm;
  /// function to create and deploy the cartesian communicator
  void initializeCommunicator(const int nx, const int ny,
          const int nz, const bool per);
  /// default construction
  Lattice(const int StO);
  /// SUNContext object
  SUNContext sunctx;
  /// SUNProfiler object
  SUNProfiler profobj;
  /// component function for resizing the discrete dimensions of the lattice
  void setDiscreteDimensions(const sunindextype _nx,
          const sunindextype _ny, const sunindextype _nz);
  /// component function for resizing the physical size of the lattice
  void setPhysicalDimensions(const sunrealtype _lx,
          const sunrealtype _ly, const sunrealtype _lz);
  ///@{
  /** getter function */
  [[nodiscard]] const sunrealtype &get_tot_lx() const { return tot_lx; }
  [[nodiscard]] const sunrealtype &get_tot_ly() const { return tot_ly; }
  [[nodiscard]] const sunrealtype &get_tot_lz() const { return tot_lz; }
  [[nodiscard]] const sunindextype &get_tot_nx() const { return tot_nx; }
  [[nodiscard]] const sunindextype &get_tot_ny() const { return tot_ny; }
  [[nodiscard]] const sunindextype &get_tot_nz() const { return tot_nz; }
  [[nodiscard]] const sunindextype &get_tot_noP() const { return tot_noP; }
```

```
  [[nodiscard]] const sunindextype &get_tot_noDP() const { return tot_noDP; }
  [[nodiscard]] const sunrealtype &get_dx() const { return dx; }
  [[nodiscard]] const sunrealtype &get_dy() const { return dy; }
  [[nodiscard]] const sunrealtype &get_dz() const { return dz; }
  [[nodiscard]] constexpr int get_dataPointDimension() const {
    return dataPointDimension;
  }
  [[nodiscard]] const int &get_stencilOrder() const { return stencilOrder; }
  [[nodiscard]] const int &get_ghostLayerWidth() const {
    return ghostLayerWidth;
  }
  ///@}
};

/// lattice patch construction checking flags
enum LatticePatchOptions {
  FLatticePatchSetUp = 0x01,
  TranslocationLookupSetUp = 0x02,
  GhostLayersInitialized = 0x04,
  BuffersInitialized = 0x08
  /*OPT_D = 0x08,
  OPT_E = 0x10,
  OPT_F = 0x20,*/
};

/** @brief LatticePatch class for the construction of the patches in the
 * enveloping lattice */
class LatticePatch {
private:
  /// origin of the patch in physical space; x-coordinate
  sunrealtype x0;
  /// origin of the patch in physical space; y-coordinate
  sunrealtype y0;
  /// origin of the patch in physical space; z-coordinate
  sunrealtype z0;
  /// inner position of lattice-patch in the lattice patchwork; x-points
  sunindextype LIx;
  /// inner position of lattice-patch in the lattice patchwork; y-points
  sunindextype LIy;
  /// inner position of lattice-patch in the lattice patchwork; z-points
  sunindextype LIz;
  /// physical size of the lattice-patch in the x-dimension
  sunrealtype lx;
  /// physical size of the lattice-patch in the y-dimension
  sunrealtype ly;
  /// physical size of the lattice-patch in the z-dimension
  sunrealtype lz;
  /// number of points in the lattice patch in the x-dimension
  sunindextype nx;
  /// number of points in the lattice patch in the y-dimension
  sunindextype ny;
  /// number of points in the lattice patch in the z-dimension
  sunindextype nz;
  /// physical distance between lattice points in x-direction
  sunrealtype dx;
  /// physical distance between lattice points in y-direction
  sunrealtype dy;
  /// physical distance between lattice points in z-direction
  sunrealtype dz;
  /// pointer to the enveloping lattice
  const Lattice *envelopeLattice;
  ///@{
  /** translocation lookup table */
  vector<int> uTox, uToy, uToz, xTou, yTou, zTou;
  ///@}
  /// aid (auxilliarly) vector including ghost cells to compute the derivatives
  vector<sunrealtype> uAux;
  ///@{
  /** buffer to save spatial derivative values */
  vector<sunrealtype> buffX, buffY, buffZ;
  ///@}
  ///@{
  /** buffer for passing ghost cell data */
  vector<sunrealtype> ghostCellLeft, ghostCellRight, ghostCellLeftToSend,
      ghostCellRightToSend, ghostCellsToSend, ghostCells;
  ///@}
  ///@{
  /** ghost cell translocation lookup table */
  vector<int> lgcTox, rgcTox, lgcToy, rgcToy, lgcToz, rgcToz;
  ///@}
  /** char for checking flags */
  unsigned char statusFlags;
  ///@{
  /** rotate and translocate an input array according to a lookup into an output
   * array */
  inline void rotateToX(sunrealtype *outArray, const sunrealtype *inArray,
                        const vector<int> &lookup);
```

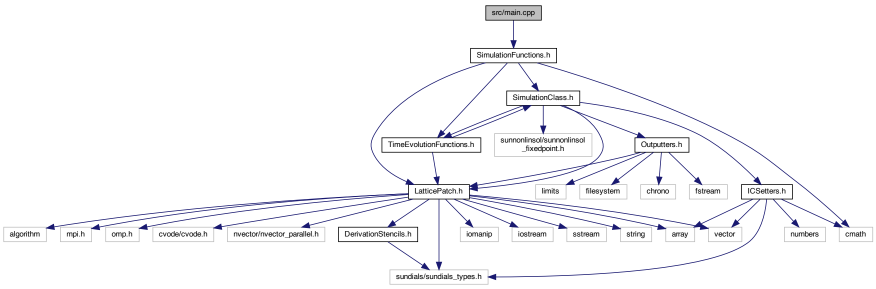
```
  inline void rotateToY(sunrealtype *outArray, const sunrealtype *inArray,
                        const vector<int> &lookup);
  inline void rotateToZ(sunrealtype *outArray, const sunrealtype *inArray,
                        const vector<int> &lookup);
  ///@}
public:
  /// ID of the LatticePatch, corresponds to process number
  // (required solely for  debugging)
  int ID;
  /// N_Vector for saving field components u=(E,B) in lattice points
  N_Vector u;
  /// N_Vector for saving temporal derivatives of the field data
  N_Vector du;
  /// pointer to field data
  sunrealtype *uData;
  /// pointer to auxiliary data vector
  sunrealtype *uAuxData;
  /// pointer to time-derivative data
  sunrealtype *duData;
  ///@{
  /** pointer to halo data */
  sunrealtype *gCLData, *gCRData, *gCBData, *gCTData, *gCFData, *gCAData;
  ///@}
  /// pointer to spatial derivative data buffers
  array<sunrealtype *, 3> buffData;
  /// constructor setting up a default first lattice patch
  LatticePatch();
  /// destructor freeing parallel vectors
  ~LatticePatch();
  /// friend function for creating the patchwork slicing of the overall lattice
  friend int generatePatchwork(const Lattice &envelopeLattice,
                               LatticePatch &patchToMold, const int DLx,
                               const int DLy, const int DLz);
  /// function to get the discrete size of the LatticePatch
  // (0 direction corresponds to total)
  int discreteSize(int dir=0) const;
  /// function to get the origin of the patch
  sunrealtype origin(const int dir) const;
  /// function to get distance between points
  sunrealtype getDelta(const int dir) const;
  /// function to fill out the lookup tables for translocation
  // and de-translocation of data point
  void generateTranslocationLookup();
  /// function to rotate u into Z-matrix eigenraum
  // and make it the primary lattice direction of dir
  void rotateIntoEigen(const int dir);
  /// function to rotate as in `rotateIntoEigen` with special 3D halo buffers
  void rotateIntoEigen3D();
  /// function to derotate uAux into dudata lattice direction of x
  void derotate(int dir, sunrealtype *buffOut);
  /// initialize ghost cells for halo exchange
  void initializeGhostLayer();
  /// initialize buffers to save derivatives
  void initializeBuffers();
  /// function to exchange ghost cells in uAux for the derivative
  void exchangeGhostCells(const int dir);
  /// function to exchange ghost cells using a neighborhood collective operation
  // for 3D simulations; requires cubic patches
  void exchangeGhostCells3D();
  /// outsourced convenience function to fill halo buffers with uData for 3D
  void bufferCreator(int li, int mx, int my, int mz, int distToRight);
  /// function to derive the centered values in uAux and save them noncentered
  void derive(const int dir);
  /// function to check if a flag has been set and if not abort
  void checkFlag(unsigned int flag) const;
};

// helper function for error messages
void errorKill(const string & errorMessage);

// helper function to check for CVode success
int check_retval(void *returnvalue, const char *funcname, int opt, int id);

```

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 fom TE-mode (z-axis)
- taille
- Rayleigh length
- beam center
- beam length

A 3D simulation with specified

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

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.

```
{
    // Initialize MPI environment
    MPI_Init (&argc, &argv);
    MPI_Comm comm = MPI_COMM_WORLD;
    // Prepare MPI for Master-only threading
    //int provided;
    //MPI_Init_thread(&argc, &argv, MPI_THREAD_FUNNELED, &provided);

    int rank = 0;
    MPI_Comm_rank(comm,&rank);
    double ti=MPI_Wtime(); // Overall start time

    /** 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      */
    //constexpr auto outputDirectory = "/gpfs/scratch/uh3o1/ru68dab/ru68dab/HE/outputs";
    //constexpr auto outputDirectory = "/home/andi/Documents/";
    constexpr auto outputDirectory = "/Users/andi/Documents/";

    if(rank==0 && !filesystem::exists(outputDirectory)) {
        cerr«"\nOutput directory nonexistent.\n";
        MPI_Abort(comm,1);
    }


    //------------ BEGIN OF CONFIGURATION ------------//

    //////////////////// -- 1D -- /////////////////////
    /** A 1D simulation with specified */

    //// Specify your settings here ////
    constexpr array <sunrealtype,2> CVodeTolerances={1.0e-16,1.0e-16}; /// - relative and absolute
tolerances of the CVode solver
```

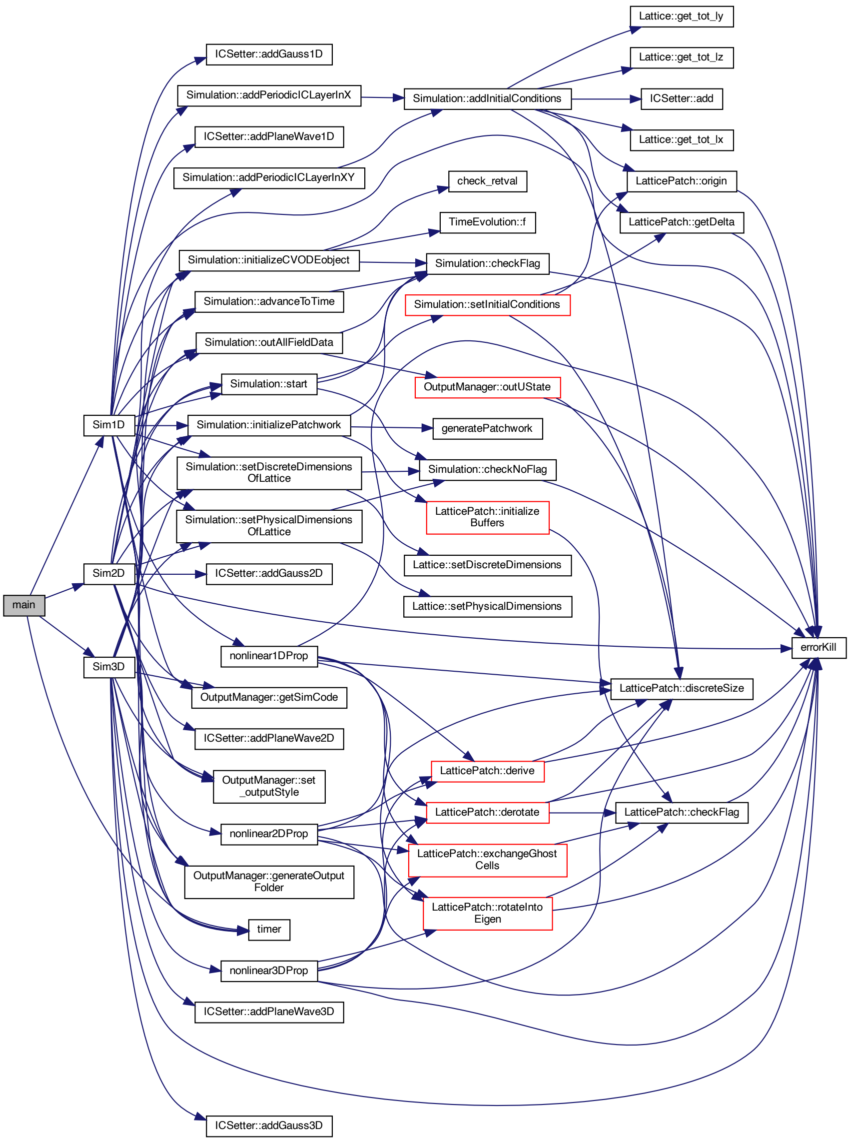
```
    constexpr int StencilOrder=13;                              /// - accuracy order of the stencils in the range 1-13
    constexpr sunrealtype physical_sidelength=300e-6;           /// - physical length of the lattice in meters
    constexpr sunindextype latticepoints=6e3;                   /// - number of lattice points
    constexpr bool periodic=true;                               /// - periodic or vanishing boundary values
    int processOrder=3;                                         /// - included processes of the weak-field expansion, see README.md
    constexpr sunrealtype simulationTime=100.0e-61;             /// -  physical total simulation time
    constexpr int numberOfSteps=100;                            /// -  discrete time steps
    constexpr int outputStep=1;                               /// -  output step multiples
    constexpr char outputStyle='c';                           /// - output in csv (c) or binary (b)

    /// Add electromagnetic waves.
    planewave plane1;              /// A plane wave with
    plane1.k = {1e5,0,0};          /// - wavevector (normalized to \f$ 1/\lambda \f$)
    plane1.p = {0,0,0.1};          /// - amplitude/polarization
    plane1.phi = {0,0,0};          /// - phase shift
    planewave plane2;              /// Another plane wave with
    plane2.k = {-1e6,0,0};         /// - wavevector (normalized to \f$ 1/\lambda \f$)
    plane2.p = {0,0,0.5};          /// - amplitude/polarization
    plane2.phi = {0,0,0};          /// - phase shift
    // Do not comment out this vector, even if no plane wave is used. But if, emplace used plane waves.
    vector<planewave> planewaves;
    //planewaves.emplace_back(plane1);
    //planewaves.emplace_back(plane2);

    gaussian1D gauss1;             /// A Gaussian wave with
    gauss1.k = {1.0e6,0,0};        /// - wavevector (normalized to \f$ 1/\lambda \f$)
    gauss1.p = {0,0,0.1};          /// - polarization/amplitude
    gauss1.x0 = {100e-6,0,0};      /// - shift from origin
    gauss1.phig = 5e-6;            /// - width
    gauss1.phi = {0,0,0};          /// - phase shift
    gaussian1D gauss2;             /// Another Gaussian with
    gauss2.k = {-0.2e6,0,0};       /// - wavevector (normalized to \f$ 1/\lambda \f$)
    gauss2.p = {0,0,0.5};          /// - polarization/amplitude
    gauss2.x0 = {200e-6,0,0};      /// - shift from origin
    gauss2.phig = 15e-6;           /// - width
    gauss2.phi = {0,0,0};          /// - phase shift
    // Do not comment out this vector, even if no Gaussian wave is used. But if, emplace used Gaussian waves.
    vector<gaussian1D> Gaussians1D;
    Gaussians1D.emplace_back(gauss1);
    Gaussians1D.emplace_back(gauss2);

    //// Do not change this below ////
    int *interactions = &processOrder;
    Sim1D(CVodeTolerances,StencilOrder,physical_sidelength,latticepoints,
            periodic,interactions,simulationTime,numberOfSteps,
            outputDirectory,outputStep,outputStyle,
            planewaves,Gaussians1D);

    ///////////////////////////////////////////////////


    //////////////////// -- 2D -- //////////////////////
    /** A 2D simulation with specified */

    //// Specify your settings here ////
    constexpr array<sunrealtype,2> CVodeTolerances={1.0e-12,1.0e-12};  /// - relative and absolute tolerances of the CVode solver
    constexpr int StencilOrder=13;                              /// - accuracy order of the stencils in the range 1-13
    constexpr array<sunrealtype,2> physical_sidelengths={80e-6,80e-6}; /// - physical length of the lattice in the given dimensions in meters
    constexpr array<sunindextype,2> latticepoints_per_dim={800,800};   /// - number of lattice points per dimension
    constexpr array<int,2> patches_per_dim={2,2};               /// - slicing of discrete dimensions into patches
    constexpr bool periodic=true;                               /// - periodic or vanishing boundary values
    int processOrder=3;                                         /// - included processes of the weak-field expansion, see README.md
    constexpr sunrealtype simulationTime=4e-61;                 /// - physical total simulation time
    constexpr int numberOfSteps=10;                             /// - discrete time steps
    constexpr int outputStep=1;                                /// - output step multiples
    constexpr char outputStyle='c';                           /// - output in csv (c) or binary (b)

    /// Add electromagnetic waves.
    planewave plane1;                /// A plane wave with
    plane1.k = {1e5,0,0};            /// - wavevector (normalized to \f$ 1/\lambda \f$)
```

```
    plane1.p = {0,0,0.1};              /// - amplitude/polarization
    plane1.phi = {0,0,0};              /// - phase shift
    planewave plane2;                  /// Another plane wave with
    plane2.k = {-1e6,0,0};             /// - wavevector
    plane2.p = {0,0,0.5};              /// - amplitude/polarization
    plane2.phi = {0,0,0};              /// - phase shift
    // Do not comment out this vector, even if no plane wave is used. But if, emplace used plane waves.
    vector<planewave> planewaves;
    //planewaves.emplace_back(plane1);
    //planewaves.emplace_back(plane2);

    gaussian2D gauss1;                 /// A Gaussian wave with
    gauss1.x0 = {40e-6,40e-6};         /// - center it approaches
    gauss1.axis = {1,0};               /// - normalized direction _from_ which the wave approaches the center
    gauss1.amp = 0.5;                  /// - amplitude
    gauss1.phip = 2*atan(0);           /// - polarization rotation from TE-mode (z-axis)
    gauss1.w0 = 2.3e-6;                /// - taille
    gauss1.zr = 16.619e-6;             /// - Rayleigh length
    /// the wavelength is determined by the relation \f$ \lambda = \pi*w_0^2/z_R \f$
    gauss1.ph0 = 2e-5;                 /// - beam center
    gauss1.phA = 0.45e-5;              /// - beam length
    gaussian2D gauss2;                 /// Another Gaussian wave with
    gauss2.x0 = {40e-6,40e-6};         /// - center it approaches
    gauss2.axis = {-0.7071,0.7071};    /// - normalized direction from which the wave approaches the center
    gauss2.amp = 0.5;                  /// - amplitude
    gauss2.phip = 2*atan(0);           /// - polarization rotation fom TE-mode (z-axis)
    gauss2.w0 = 2.3e-6;                /// - taille
    gauss2.zr = 16.619e-6;             /// - Rayleigh length
    gauss2.ph0 = 2e-5;                 /// - beam center
    gauss2.phA = 0.45e-5;              /// - beam length
    // Do not comment out this vector, even if no Gaussian wave is used. But if, emplace used Gaussian waves.
    vector<gaussian2D> Gaussians2D;
    Gaussians2D.emplace_back(gauss1);
    Gaussians2D.emplace_back(gauss2);

    //// Do not change this below ////
    static_assert(latticepoints_per_dim[0]%patches_per_dim[0]==0 &&
            latticepoints_per_dim[1]%patches_per_dim[1]==0,
            "The number of lattice points in each dimension must be "
            "divisible by the number of patches in that direction.");
    int * interactions = &processOrder;
    Sim2D(CVodeTolerances,StencilOrder,physical_sidelengths,
            latticepoints_per_dim,patches_per_dim,periodic,interactions,
            simulationTime,numberOfSteps,outputDirectory,outputStep,
            outputStyle,planewaves,Gaussians2D);

    ////////////////////////////////////////////////////


    //////////////////// -- 3D -- //////////////////////
    /** A 3D simulation with specified */

    //// Specify your settings here ////
    constexpr array<sunrealtype,2> CVodeTolerances={1.0e-12,1.0e-12};        /// - relative and absolute tolerances of the CVode solver
    constexpr int StencilOrder=4;                                            /// - accuracy order of the stencils in the range 1-13
    constexpr array<sunrealtype,3> physical_sidelengths={80e-6,80e-6,20e-6}; /// - physical dimensions in meters
    constexpr array<sunindextype,3> latticepoints_per_dim={160,160,40};      /// - number of lattice points in any dimension
    constexpr array<int,3> patches_per_dim= {2,2,1};                         /// - slicing of discrete dimensions into patches
    constexpr bool periodic=false;                                           /// - perodic or non-periodic boundaries
    int processOrder=3;                                                      /// - processes of the weak-field expansion, see README.md
    constexpr sunrealtype simulationTime=2e-6;                               /// - physical total simulation time
    constexpr int numberOfSteps=5;                                           /// - discrete time steps

    constexpr int outputStep=1;                                              /// - output step multiples
    constexpr char outputStyle='b';                                  /// - output in csv (c) or binary (b)
                                                                     //
    /// Add electromagnetic waves.
    planewave plane1;                  /// A plane wave with
    plane1.k = {1e5,0,0};              /// - wavevector (normalized to \f$ 1/\lambda \f$)
    plane1.p = {0,0,0.1};              /// - amplitude/polarization
    plane1.phi = {0,0,0};              /// - phase shift
    planewave plane2;                  /// Another plane wave with
    plane2.k = {-1e6,0,0};             /// - wavevector (normalized to \f$ 1/\lambda \f$)
```

```
    plane2.p = {0,0,0.5};               /// - amplitude/polarization
    plane2.phi = {0,0,0};               /// - phase shift
    // Do not comment out this vector, even if no plane wave is used. But if, emplace used plane
  waves.
    vector<planewave> planewaves;
    //planewaves.emplace_back(plane1);
    //planewaves.emplace_back(plane2);

    gaussian3D gauss1;                  /// A Gaussian wave with
    gauss1.x0 = {40e-6,40e-6,10e-6};    /// - center it approaches
    gauss1.axis = {1,0,0};              /// - normalized direction _from_ which the wave approaches
  the center
    gauss1.amp = 0.05;                  /// - amplitude
    gauss1.phip = 2*atan(0);            /// - polarization rotation from TE-mode (z-axis)
    gauss1.w0 = 3.5e-6;                 /// - taille
    gauss1.zr = 19.242e-6;              /// - Rayleigh length
    /// the wavelength is determined by the relation \f$ \lambda = \pi*w_0^2/z_R \f$
    gauss1.ph0 = 2e-5;                  /// - beam center
    gauss1.phA = 0.45e-5;               /// - beam length
    gaussian3D gauss2;                  /// Another Gaussian wave with
    gauss2.x0 = {40e-6,40e-6,10e-6};    /// - center it approaches
    gauss2.axis = {0,1,0};              /// - normalized direction from which the wave approaches the
  center
    gauss2.amp = 0.05;                  /// - amplitude
    gauss2.phip = 2*atan(0);            /// - polarization rotation from TE-mode (z-axis)
    gauss2.w0 = 3.5e-6;                 /// - taille
    gauss2.zr = 19.242e-6;              /// - Rayleigh length
    gauss2.ph0 = 2e-5;                  /// - beam center
    gauss2.phA = 0.45e-5;               /// - beam length
    // Do not comment out this vector, even if no Gaussian wave is used. But if, emplace used Gaussian
  waves.
    vector<gaussian3D> Gaussians3D;
    Gaussians3D.emplace_back(gauss1);
    Gaussians3D.emplace_back(gauss2);

    //// Do not change this below ////
    static_assert(latticepoints_per_dim[0]%patches_per_dim[0]==0 &&
            latticepoints_per_dim[1]%patches_per_dim[1]==0 &&
            latticepoints_per_dim[2]%patches_per_dim[2]==0,
            "The number of lattice points in each dimension must be "
            "divisible by the number of patches in that direction.");
    int *interactions = &processOrder;
    Sim3D(CVodeTolerances,StencilOrder,physical_sidelengths,
            latticepoints_per_dim,patches_per_dim,periodic,interactions,
            simulationTime,numberOfSteps,outputDirectory,outputStep,
            outputStyle,planewaves,Gaussians3D);

    /////////////////////////////////////////////////////

    //------------- END OF CONFIGURATION -------------//

    double tf=MPI_Wtime(); // Overall finish time
    if(rank==0) {cout«endl; timer(ti,tf);} // Print the elapsed time

    // Finalize MPI environment
    MPI_Finalize();

    return 0;
}
```

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.

```
/// @file main.cpp
/// @brief Main function to configure the user’s simulation settings
#include "SimulationFunctions.h"   /* complete simulation functions and all headers */
int main(int argc, char *argv[])
{
    // Initialize MPI environment
    MPI_Init (&argc, &argv);
    MPI_Comm comm = MPI_COMM_WORLD;
    // Prepare MPI for Master-only threading
    //int provided;
    //MPI_Init_thread(&argc, &argv, MPI_THREAD_FUNNELED, &provided);
```

```
    int rank = 0;
    MPI_Comm_rank(comm,&rank);
    double ti=MPI_Wtime(); // Overall start time

    /** 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      */
    //constexpr auto outputDirectory = "/gpfs/scratch/uh3o1/ru68dab/ru68dab/HE/outputs";
    //constexpr auto outputDirectory = "/home/andi/Documents/";
    constexpr auto outputDirectory = "/Users/andi/Documents/";

    if(rank==0 && !filesystem::exists(outputDirectory)) {
        cerr«"\nOutput directory nonexistent.\n";
        MPI_Abort(comm,1);
    }


    //------------ BEGIN OF CONFIGURATION ------------//

    //////////////////// -- 1D -- //////////////////////
    /** A 1D simulation with specified */

    //// Specify your settings here ////
    constexpr array <sunrealtype,2> CVodeTolerances={1.0e-16,1.0e-16}; /// - relative and absolute tolerances of the CVode solver
    constexpr int StencilOrder=13;                                     /// - accuracy order of the stencils in the range 1-13
    constexpr sunrealtype physical_sidelength=300e-6;                  /// - physical length of the lattice in meters
    constexpr sunindextype latticepoints=6e3;                          /// - number of lattice points
    constexpr bool periodic=true;                                      /// - periodic or vanishing boundary values
    int processOrder=3;                                                /// - included processes of the weak-field expansion, see README.md
    constexpr sunrealtype simulationTime=100.0e-6l;                    /// -  physical total simulation time
    constexpr int numberOfSteps=100;                                   /// -  discrete time steps
    constexpr int outputStep=1;                                      /// -  output step multiples
    constexpr char outputStyle='c';                                  /// - output in csv (c) or binary (b)

    /// Add electromagnetic waves.
    planewave plane1;               /// A plane wave with
    plane1.k = {1e5,0,0};           /// - wavevector (normalized to \f$ 1/\lambda \f$)
    plane1.p = {0,0,0.1};           /// - amplitude/polarization
    plane1.phi = {0,0,0};           /// - phase shift
    planewave plane2;               /// Another plane wave with
    plane2.k = {-1e6,0,0};          /// - wavevector (normalized to \f$ 1/\lambda \f$)
    plane2.p = {0,0,0.5};           /// - amplitude/polarization
    plane2.phi = {0,0,0};           /// - phase shift
    // Do not comment out this vector, even if no plane wave is used. But if, emplace used plane waves.
    vector<planewave> planewaves;
    //planewaves.emplace_back(plane1);
    //planewaves.emplace_back(plane2);

    gaussian1D gauss1;              /// A Gaussian wave with
    gauss1.k = {1.0e6,0,0};         /// - wavevector (normalized to \f$ 1/\lambda \f$)
    gauss1.p = {0,0,0.1};           /// - polarization/amplitude
    gauss1.x0 = {100e-6,0,0};       /// - shift from origin
    gauss1.phig = 5e-6;             /// - width
    gauss1.phi = {0,0,0};           /// - phase shift
    gaussian1D gauss2;              /// Another Gaussian with
    gauss2.k = {-0.2e6,0,0};        /// - wavevector (normalized to \f$ 1/\lambda \f$)
    gauss2.p = {0,0,0.5};           /// - polarization/amplitude
    gauss2.x0 = {200e-6,0,0};       /// - shift from origin
    gauss2.phig = 15e-6;            /// - width
    gauss2.phi = {0,0,0};           /// - phase shift
    // Do not comment out this vector, even if no Gaussian wave is used. But if, emplace used Gaussian waves.
    vector<gaussian1D> Gaussians1D;
    Gaussians1D.emplace_back(gauss1);
    Gaussians1D.emplace_back(gauss2);

    //// Do not change this below ////
    int *interactions = &processOrder;
    Sim1D(CVodeTolerances,StencilOrder,physical_sidelength,latticepoints,
            periodic,interactions,simulationTime,numberOfSteps,
            outputDirectory,outputStep,outputStyle,
            planewaves,Gaussians1D);

    ///////////////////////////////////////////////////


    //////////////////// -- 2D -- //////////////////////
    /** A 2D simulation with specified */
```

```

    //// Specify your settings here ////
    constexpr array<sunrealtype,2> CVodeTolerances={1.0e-12,1.0e-12};  /// - relative and absolute tolerances of the CVode solver
    constexpr int StencilOrder=13;                                     /// - accuracy order of the stencils in the range 1-13
    constexpr array<sunrealtype,2> physical_sidelengths={80e-6,80e-6}; /// - physical length of the lattice in the given dimensions in meters
    constexpr array<sunindextype,2> latticepoints_per_dim={800,800};   /// - number of lattice points per dimension
    constexpr array<int,2> patches_per_dim={2,2};                      /// - slicing of discrete dimensions into patches
    constexpr bool periodic=true;                                      /// - periodic or vanishing boundary values
    int processOrder=3;                                                /// - included processes of the weak-field expansion, see README.md
    constexpr sunrealtype simulationTime=4e-61;                        /// - physical total simulation time
    constexpr int numberOfSteps=10;                                    /// - discrete time steps
    constexpr int outputStep=1;                                       /// - output step multiples
    constexpr char outputStyle='c';                                  /// - output in csv (c) or binary (b)

    /// Add electromagnetic waves.
    planewave plane1;                  /// A plane wave with
    plane1.k = {1e5,0,0};              /// - wavevector (normalized to \f$ 1/\lambda \f$)
    plane1.p = {0,0,0.1};              /// - amplitude/polarization
    plane1.phi = {0,0,0};              /// - phase shift
    planewave plane2;                  /// Another plane wave with
    plane2.k = {-1e6,0,0};             /// - wavevector
    plane2.p = {0,0,0.5};              /// - amplitude/polarization
    plane2.phi = {0,0,0};              /// - phase shift
    // Do not comment out this vector, even if no plane wave is used. But if, emplace used plane waves.
    vector<planewave> planewaves;
    //planewaves.emplace_back(plane1);
    //planewaves.emplace_back(plane2);

    gaussian2D gauss1;                 /// A Gaussian wave with
    gauss1.x0 = {40e-6,40e-6};         /// - center it approaches
    gauss1.axis = {1,0};               /// - normalized direction _from_ which the wave approaches the center
    gauss1.amp = 0.5;                  /// - amplitude
    gauss1.phip = 2*atan(0);           /// - polarization rotation from TE-mode (z-axis)
    gauss1.w0 = 2.3e-6;                /// - taille
    gauss1.zr = 16.619e-6;             /// - Rayleigh length
    /// the wavelength is determined by the relation \f$ \lambda = \pi*w_0^2/z_R \f$
    gauss1.ph0 = 2e-5;                 /// - beam center
    gauss1.phA = 0.45e-5;              /// - beam length
    gaussian2D gauss2;                 /// Another Gaussian wave with
    gauss2.x0 = {40e-6,40e-6};         /// - center it approaches
    gauss2.axis = {-0.7071,0.7071};    /// - normalized direction from which the wave approaches the center
    gauss2.amp = 0.5;                  /// - amplitude
    gauss2.phip = 2*atan(0);           /// - polarization rotation fom TE-mode (z-axis)
    gauss2.w0 = 2.3e-6;                /// - taille
    gauss2.zr = 16.619e-6;             /// - Rayleigh length
    gauss2.ph0 = 2e-5;                 /// - beam center
    gauss2.phA = 0.45e-5;              /// - beam length
    // Do not comment out this vector, even if no Gaussian wave is used. But if, emplace used Gaussian waves.
    vector<gaussian2D> Gaussians2D;
    Gaussians2D.emplace_back(gauss1);
    Gaussians2D.emplace_back(gauss2);

    //// Do not change this below ////
    static_assert(latticepoints_per_dim[0]%patches_per_dim[0]==0 &&
            latticepoints_per_dim[1]%patches_per_dim[1]==0,
            "The number of lattice points in each dimension must be "
            "divisible by the number of patches in that direction.");
    int * interactions = &processOrder;
    Sim2D(CVodeTolerances,StencilOrder,physical_sidelengths,
            latticepoints_per_dim,patches_per_dim,periodic,interactions,
            simulationTime,numberOfSteps,outputDirectory,outputStep,
            outputStyle,planewaves,Gaussians2D);

    ////////////////////////////////////////////////////


    ///////////////////// -- 3D -- /////////////////////
    /** A 3D simulation with specified */

    //// Specify your settings here ////
    constexpr array<sunrealtype,2> CVodeTolerances={1.0e-12,1.0e-12};        /// - relative and absolute tolerances of the CVode solver
    constexpr int StencilOrder=4;                                            /// - accuracy order of the stencils in the range 1-13
```

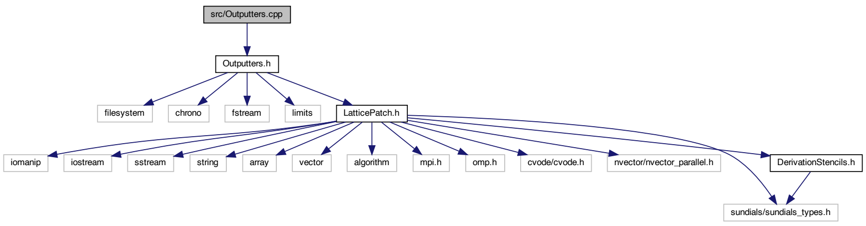
```
    constexpr array<sunrealtype,3> physical_sidelengths={80e-6,80e-6,20e-6}; /// - physical dimensions in meters
    constexpr array<sunindextype,3> latticepoints_per_dim={160,160,40};      /// - number of lattice points in any dimension
    constexpr array<int,3> patches_per_dim= {2,2,1};                         /// - slicing of discrete dimensions into patches
    constexpr bool periodic=false;                                           /// - perodic or non-periodic boundaries
    int processOrder=3;                                                      /// - processes of the weak-field expansion, see README.md
    constexpr sunrealtype simulationTime=2e-6;                               /// - physical total simulation time
    constexpr int numberOfSteps=5;                                           /// - discrete time steps
    constexpr int outputStep=1;                                              /// - output step multiples
    constexpr char outputStyle='b';                                  /// - output in csv (c) or binary (b)
                                                                     //
    /// Add electromagnetic waves.
    planewave plane1;                   /// A plane wave with
    plane1.k = {1e5,0,0};               /// - wavevector (normalized to \f$ 1/\lambda \f$)
    plane1.p = {0,0,0.1};               /// - amplitude/polarization
    plane1.phi = {0,0,0};               /// - phase shift
    planewave plane2;                   /// Another plane wave with
    plane2.k = {-1e6,0,0};              /// - wavevector (normalized to \f$ 1/\lambda \f$)
    plane2.p = {0,0,0.5};               /// - amplitude/polarization
    plane2.phi = {0,0,0};               /// - phase shift
    // Do not comment out this vector, even if no plane wave is used. But if, emplace used plane waves.
    vector<planewave> planewaves;
    //planewaves.emplace_back(plane1);
    //planewaves.emplace_back(plane2);

    gaussian3D gauss1;                  /// A Gaussian wave with
    gauss1.x0 = {40e-6,40e-6,10e-6};    /// - center it approaches
    gauss1.axis = {1,0,0};              /// - normalized direction _from_ which the wave approaches the center
    gauss1.amp = 0.05;                  /// - amplitude
    gauss1.phip = 2*atan(0);            /// - polarization rotation from TE-mode (z-axis)
    gauss1.w0 = 3.5e-6;                 /// - taille
    gauss1.zr = 19.242e-6;              /// - Rayleigh length
    /// the wavelength is determined by the relation \f$ \lambda = \pi*w_0^2/z_R \f$
    gauss1.ph0 = 2e-5;                  /// - beam center
    gauss1.phA = 0.45e-5;               /// - beam length
    gaussian3D gauss2;                  /// Another Gaussian wave with
    gauss2.x0 = {40e-6,40e-6,10e-6};    /// - center it approaches
    gauss2.axis = {0,1,0};              /// - normalized direction from which the wave approaches the center
    gauss2.amp = 0.05;                  /// - amplitude
    gauss2.phip = 2*atan(0);            /// - polarization rotation from TE-mode (z-axis)
    gauss2.w0 = 3.5e-6;                 /// - taille
    gauss2.zr = 19.242e-6;              /// - Rayleigh length
    gauss2.ph0 = 2e-5;                  /// - beam center
    gauss2.phA = 0.45e-5;               /// - beam length
    // Do not comment out this vector, even if no Gaussian wave is used. But if, emplace used Gaussian waves.
    vector<gaussian3D> Gaussians3D;
    Gaussians3D.emplace_back(gauss1);
    Gaussians3D.emplace_back(gauss2);

    //// Do not change this below ////
    static_assert(latticepoints_per_dim[0]%patches_per_dim[0]==0 &&
            latticepoints_per_dim[1]%patches_per_dim[1]==0 &&
            latticepoints_per_dim[2]%patches_per_dim[2]==0,
            "The number of lattice points in each dimension must be "
            "divisible by the number of patches in that direction.");
    int *interactions = &processOrder;
    Sim3D(CVodeTolerances,StencilOrder,physical_sidelengths,
            latticepoints_per_dim,patches_per_dim,periodic,interactions,
            simulationTime,numberOfSteps,outputDirectory,outputStep,
            outputStyle,planewaves,Gaussians3D);

    ////////////////////////////////////////////////////

    //------------- END OF CONFIGURATION -------------//

    double tf=MPI_Wtime(); // Overall finish time
    if(rank==0) {cout«endl; timer(ti,tf);} // Print the elapsed time

    // Finalize MPI environment
    MPI_Finalize();

    return 0;
}
```

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.

```
////////////////////////////////////////////////////////////////$
/// @file Outputters.cpp
/// @brief Generation of output writing to disk
////////////////////////////////////////////////////////////////$

#include "Outputters.h"

/// Directly generate the simCode at construction
OutputManager::OutputManager() {
  simCode = SimCodeGenerator();
  outputStyle = 'c';
}

/// Generate the identifier number reverse from year to minute in the format
/// yy-mm-dd_hh-MM-ss
string OutputManager::SimCodeGenerator() {
  const chrono::time_point<chrono::system_clock> now{
      chrono::system_clock::now()};
  const chrono::year_month_day ymd{chrono::floor<chrono::days>(now)};
  const auto tod = now - chrono::floor<chrono::days>(now);
  const chrono::hh_mm_ss hms{tod};

  stringstream temp;
  temp « setfill('0') « setw(2)
       « static_cast<int>(ymd.year() - chrono::years(2000)) « "-"
       « setfill('0') « setw(2) « static_cast<unsigned>(ymd.month()) « "-"
       « setfill('0') « setw(2) « static_cast<unsigned>(ymd.day()) « "_"
       « setfill('0') « setw(2) « hms.hours().count() « "-" « setfill('0')
       « setw(2) « hms.minutes().count() « "-" « setfill('0') « setw(2)
       « hms.seconds().count();
       //« "_" « hms.subseconds().count(); // subseconds render the filename
       // too large
  return temp.str();
}

/** Generate the folder to save the data to by one process:
```

```
 * In the given directory it creates a direcory "SimResults" and a directory
 * with the simCode. The relevant part of the main file is written to a
 * "config.txt" file in that directory to log the settings. */
void OutputManager::generateOutputFolder(const string &dir) {
  // Do this only once for the first process
  int myPrc;
  MPI_Comm_rank(MPI_COMM_WORLD, &myPrc);
  if (myPrc == 0) {
    if (!fs::is_directory(dir))
      fs::create_directory(dir);
    if (!fs::is_directory(dir + "/SimResults"))
      fs::create_directory(dir + "/SimResults");
    if (!fs::is_directory(dir + "/SimResults/" + simCode))
      fs::create_directory(dir + "/SimResults/" + simCode);
  }
  // path variable for the output generation
  Path = dir + "/SimResults/" + simCode + "/";

  // Logging configurations from main.cpp
  ifstream fin("main.cpp");
  ofstream fout(Path + "config.txt");
  string line;
  int begin=1000;
  for (int i = 1; !fin.eof(); i++) {
    getline(fin, line);
    if (line.starts_with("    //------------ B")) {
        begin=i;
    }
    if (i < begin) {
      continue;
    }
    fout « line « endl;
    if (line.starts_with("    //------------- E")) {
        break;
    }
  }
  return;
}

void OutputManager::set_outputStyle(const char _outputStyle){
    outputStyle = _outputStyle;
}

/** 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. */
void OutputManager::outUState(const int &state, const Lattice &lattice,
        const LatticePatch &latticePatch) {
  switch(outputStyle){
      case 'c': { // one csv file per process
  ofstream ofs;
  ofs.open(Path + to_string(state) + "_" + to_string(lattice.my_prc) + ".csv");
  // Precision of sunrealtype in significant decimal digits; 15 for IEEE double
  ofs « setprecision(numeric_limits<sunrealtype>::digits10);

  // Walk through each lattice point
  for (int i = 0; i < latticePatch.discreteSize() * 6; i += 6) {
    // Six columns to contain the field data: Ex,Ey,Ez,Bx,By,Bz
    ofs « latticePatch.uData[i + 0] « "," « latticePatch.uData[i + 1] « ","
        « latticePatch.uData[i + 2] « "," « latticePatch.uData[i + 3] « ","
        « latticePatch.uData[i + 4] « "," « latticePatch.uData[i + 5]
        « endl;
  }
  ofs.close();
  break;
                  }

      case 'b': { // a single binary file
  // Open the output file
  MPI_File fh;
  const string filename = Path+to_string(state);
  MPI_File_open(lattice.comm,&filename[0],MPI_MODE_WRONLY|MPI_MODE_CREATE,
          MPI_INFO_NULL,&fh);
  // number of datapoints in the patch with process offset
  const int count = latticePatch.discreteSize()*lattice.get_dataPointDimension();
  MPI_Offset offset = lattice.my_prc*count*sizeof(MPI_SUNREALTYPE);
  // Go to offset in file and write data to it; maximal precision in
  // "native" representation
  MPI_File_set_view(fh,offset,MPI_SUNREALTYPE,MPI_SUNREALTYPE,"native",
          MPI_INFO_NULL);
  MPI_File_write_all(fh,latticePatch.uData,count,MPI_SUNREALTYPE,MPI_STATUS_IGNORE);
  /*
  MPI_Request write_request;
  MPI_File_iwrite_all(fh,latticePatch.uData,count,MPI_SUNREALTYPE,&write_request);
  MPI_Wait(&write_request,MPI_STATUS_IGNORE);
```

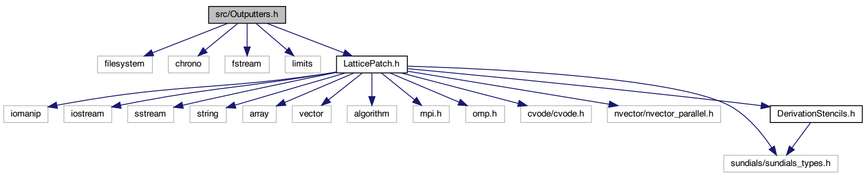
```
    */
    /*
    MPI_File_write_at_all(fh,offset,latticePatch.uData,count,MPI_SUNREALTYPE,
            MPI_STATUS_IGNORE);
            */
    break;
                    }
        default: {
    errorKill("No valid output style defined.\
            Choose between (c): one csv file per process, (b) one binary file");
    break;
                    }
    return;
    }
}

```

6.18 src/Outputters.h File Reference

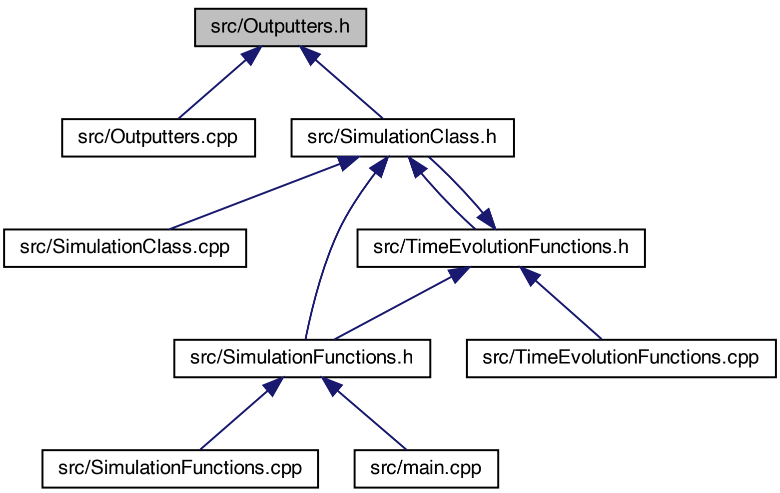
OutputManager class to outstream simulation data.

```
#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.

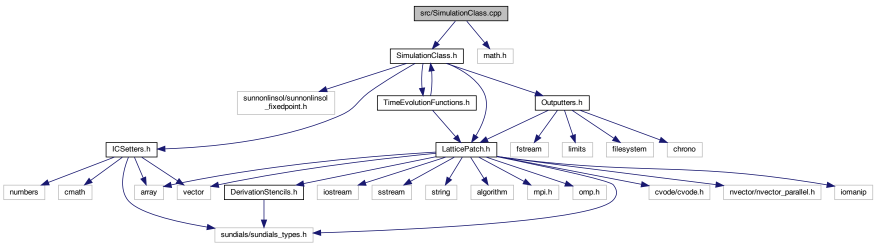
```
////////////////////////////////////////////////////////////////
/// @file Outputters.h
/// @brief OutputManager class to outstream simulation data
////////////////////////////////////////////////////////////////

#pragma once

// perform operations on the filesystem
#include <filesystem>

// output controlling with limits and timestep
#include <chrono>
#include <fstream>
#include <limits>

// project subfile header
#include "LatticePatch.h"

using namespace std;
namespace fs = std::filesystem;
namespace chrono = std::chrono;

/** @brief Output Manager class to generate and coordinate output writing to
 * disk */
class OutputManager {
private:
  /// function to create the Code of the Simulations
  static string SimCodeGenerator();
  /// varible to safe the SimCode generated at execution
  string simCode;
  /// variable for the path to the output folder
  string Path;
  /// output style; csv or binary
  char outputStyle;
public:
  /// default constructor
  OutputManager();
  /// function that creates folder to save simulation data
  void generateOutputFolder(const string &dir);
  /// set the output style
  void set_outputStyle(const char _outputStyle);
  /// function to write data to disk in specified way
  void outUState(const int &state, const Lattice &lattice, const LatticePatch &latticePatch);
  /// simCode getter function
  [[nodiscard]] const string &getSimCode() const { return simCode; }
};

```

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.

```
////////////////////////////////////////////////////////////////////////////////
/// @file SimulationClass.cpp
/// @brief 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>

/// Along with the simulation object, create the cartesian communicator and
/// SUNContext object
Simulation::Simulation(const int nx, const int ny, const int nz,
        const int StencilOrder, const bool periodicity) :
    lattice(StencilOrder){
  statusFlags = 0;
  t = 0;
  // Initialize the cartesian communicator
  lattice.initializeCommunicator(nx, ny, nz, periodicity);

  // Create the SUNContext object associated with the thread of execution
  int retval = 0;
  retval = SUNContext_Create(&lattice.comm, &lattice.sunctx);
  if (check_retval(&retval, "SUNContext_Create", 1, lattice.my_prc))
    MPI_Abort(lattice.comm, 1);
  // if (flag != CV_SUCCESS) { printf("SUNContext_Create failed, flag=%d.\n",
  // flag);
  //     MPI_Abort(lattice.comm, 1); }
}

/// Free the CVode solver memory and Sundials context object with the finish of
/// the simulation
Simulation::~Simulation() {
  // Free solver memory
  if (statusFlags & CvodeObjectSetUp) {
    // PrintFinalStats(cvode_mem); // TODO write this function as in cvodes
    // cvAdvDiff_bnd.c SUNDIALS_MARK_FUNCTION_END(lattice.profobj);
    CVodeFree(&cvode_mem);
    SUNContext_Free(&lattice.sunctx);
  }
}

```

```
/// Set the discrete dimensions, the number of points per dimension
void Simulation::setDiscreteDimensionsOfLattice(const sunindextype nx,
        const sunindextype ny, const sunindextype nz) {
  checkNoFlag(LatticePatchworkSetUp);
  lattice.setDiscreteDimensions(nx, ny, nz);
  statusFlags |= LatticeDiscreteSetUp;
}

/// Set the physical dimensions with lenghts in micro meters
void Simulation::setPhysicalDimensionsOfLattice(const sunrealtype lx,
        const sunrealtype ly, const sunrealtype lz) {
  checkNoFlag(LatticePatchworkSetUp);
  lattice.setPhysicalDimensions(lx, ly, lz);
  statusFlags |= LatticePhysicalSetUp;
}

/// Check that the lattice dimensions are set up and generate the patchwork
void Simulation::initializePatchwork(const int nx, const int ny,
        const int nz) {
  checkFlag(LatticeDiscreteSetUp);
  checkFlag(LatticePhysicalSetUp);

  // Generate the patchwork
  generatePatchwork(lattice, latticePatch, nx, ny, nz);
  latticePatch.initializeBuffers();

  statusFlags |= LatticePatchworkSetUp;
}

/// Configure CVODE
void Simulation::initializeCVODEobject(const sunrealtype reltol,
        const sunrealtype abstol) {
  checkFlag(SimulationStarted);

  // CVode settings return value
  int retval = 0;

  // Set the profiler
  retval = SUNContext_GetProfiler(lattice.sunctx, &lattice.profobj);
  if (check_retval(&retval, "SUNContext_GetProfiler", 1, lattice.my_prc))
    MPI_Abort(lattice.comm, 1);
  // if (flag != CV_SUCCESS) { printf("SUNContext_GetProfiler failed,
  // flag=%d.\n", flag);
  //     MPI_Abort(lattice.comm, 1); }

  // SUNDIALS_MARK_FUNCTION_BEGIN(profobj);

  // Create CVODE object -- returns a pointer to the cvode memory structure
  // with Adams method (Adams-Moulton formula) solver chosen for non-stiff ODE
  cvode_mem = CVodeCreate(CV_ADAMS, lattice.sunctx);

  // Specify user data and attach it to the main cvode memory block
  retval = CVodeSetUserData(
      cvode_mem,
      &latticePatch); // patch contains the user data as used in CVRhsFn
  if (check_retval(&retval, "CVodeSetUserData", 1, lattice.my_prc))
    MPI_Abort(lattice.comm, 1);
  // if (flag != CV_SUCCESS) { printf("CVodeSetUserData failed, flag=%d.\n",
  // flag);
  //     MPI_Abort(lattice.comm, 1); }

  // Initialize CVODE solver -> can only be called after start of simulation to
  // have data ready Provide required problem and solution specifications,
  // allocate internal memory, and initialize cvode
  retval = CVodeInit(cvode_mem, TimeEvolution::f, 0,
                     latticePatch.u); // allocate memory, CVRhsFn f, t_i=0, u
                                      // contains the initial values
  if (check_retval(&retval, "CVodeInit", 1, lattice.my_prc))
    MPI_Abort(lattice.comm, 1);
  // if (flag != CV_SUCCESS) { printf("CVodeInit failed, flag=%d.\n", flag);
  //     MPI_Abort(lattice.comm, 1); }

  // Create fixed point nonlinear solver object (suitable for non-stiff ODE) and
  // attach it to CVode
  SUNNonlinearSolver NLS =
      SUNNonlinSol_FixedPoint(latticePatch.u, 0, lattice.sunctx);
  retval = CVodeSetNonlinearSolver(cvode_mem, NLS);
  if (check_retval(&retval, "CVodeSetNonlinearSolver", 1, lattice.my_prc))
    MPI_Abort(lattice.comm, 1);
  // if (flag != CV_SUCCESS) {printf("CVodeSetNonlinearSolver failed,
  // flag=%d.\n", flag);
  //     MPI_Abort(lattice.comm, 1); }

  // Specify the maximum number of steps to be taken by the solver in its
  // attempt to reach the next output time
  retval = CVodeSetMaxNumSteps(cvode_mem, 10000);
  if (check_retval(&retval, "CVodeSetMaxNumSteps", 1, lattice.my_prc))
```

```
    MPI_Abort(lattice.comm, 1);
  // if (flag != CV_SUCCESS) { printf("CVodeSetMaxNumSteps failed, flag=%d.\n",
  // flag);
  //     MPI_Abort(lattice.comm, 1); }

  // Specify integration tolerances -- a scalar relative tolerance and scalar
  // absolute tolerance
  retval = CVodeSStolerances(cvode_mem, reltol, abstol);
  if (check_retval(&retval, "CVodeSStolerances", 1, lattice.my_prc))
    MPI_Abort(lattice.comm, 1);
  // if (flag != CV_SUCCESS) { printf("CVodeSStolerances failed, flag=%d.\n",
  // flag);
  //     MPI_Abort(lattice.comm, 1); }

  statusFlags |= CvodeObjectSetUp;
}

/// Check if the lattice patchwork is set up and set the initial conditions
void Simulation::start() {
  checkFlag(LatticeDiscreteSetUp);
  checkFlag(LatticePhysicalSetUp);
  checkFlag(LatticePatchworkSetUp);
  checkNoFlag(SimulationStarted);
  checkNoFlag(CvodeObjectSetUp);
  setInitialConditions();
  statusFlags |= SimulationStarted;
}

/// Set initial conditions: Fill the lattice points with the initial field
/// values
void Simulation::setInitialConditions() {
  const sunrealtype dx = latticePatch.getDelta(1);
  const sunrealtype dy = latticePatch.getDelta(2);
  const sunrealtype dz = latticePatch.getDelta(3);
  const int nx = latticePatch.discreteSize(1);
  const int ny = latticePatch.discreteSize(2);
  const sunrealtype x0 = latticePatch.origin(1);
  const sunrealtype y0 = latticePatch.origin(2);
  const sunrealtype z0 = latticePatch.origin(3);
  int px = 0, py = 0, pz = 0;
  // space coordinates
  for (int i = 0; i < latticePatch.discreteSize() * 6; i += 6) {
    px = (i / 6) % nx;
    py = ((i / 6) / nx) % ny;
    pz = ((i / 6) / nx) / ny;
    // Call the `eval` function to fill the lattice points with the field data
    icsettings.eval(static_cast<sunrealtype>(px) * dx + x0,
            static_cast<sunrealtype>(py) * dy + y0,
            static_cast<sunrealtype>(pz) * dz + z0, &latticePatch.uData[i]);
  }
  return;
}

/// Use parameters to add periodic IC layers
void Simulation::addInitialConditions(const int xm, const int ym,
        const int zm /* zm=0 always */ ) {
  const sunrealtype dx = latticePatch.getDelta(1);
  const sunrealtype dy = latticePatch.getDelta(2);
  const sunrealtype dz = latticePatch.getDelta(3);
  const int nx = latticePatch.discreteSize(1);
  const int ny = latticePatch.discreteSize(2);
  // Correct for demanded displacement, rest as for setInitialConditions
  const sunrealtype x0 = latticePatch.origin(1) + xm*lattice.get_tot_lx();
  const sunrealtype y0 = latticePatch.origin(2) + ym*lattice.get_tot_ly();
  const sunrealtype z0 = latticePatch.origin(3) + zm*lattice.get_tot_lz();
  int px = 0, py = 0, pz = 0;
  for (int i = 0; i < latticePatch.discreteSize() * 6; i += 6) {
    px = (i / 6) % nx;
    py = ((i / 6) / nx) % ny;
    pz = ((i / 6) / nx) / ny;
    icsettings.add(static_cast<sunrealtype>(px) * dx + x0,
            static_cast<sunrealtype>(py) * dy + y0,
            static_cast<sunrealtype>(pz) * dz + z0, &latticePatch.uData[i]);
  }
  return;
}

/// Add initial conditions in one dimension
void Simulation::addPeriodicICLayerInX() {
  addInitialConditions(-1, 0, 0);
  addInitialConditions(1, 0, 0);
  return;
}

/// Add initial conditions in two dimensions
void Simulation::addPeriodicICLayerInXY() {
  addInitialConditions(-1, -1, 0);
```

```
  addInitialConditions(-1, 0, 0);
  addInitialConditions(-1, 1, 0);
  addInitialConditions(0, 1, 0);
  addInitialConditions(0, -1, 0);
  addInitialConditions(1, -1, 0);
  addInitialConditions(1, 0, 0);
  addInitialConditions(1, 1, 0);
  return;
}

/// Advance the solution in time -> integrate the ODE over an interval t
void Simulation::advanceToTime(const sunrealtype &tEnd) {
  checkFlag(SimulationStarted);
  int flag = 0;
  flag = CVode(cvode_mem, tEnd, latticePatch.u, &t,
               CV_NORMAL); // CV_NORMAL: internal steps to reach tEnd, then
                           // interpolate to return latticePatch.u, return time
                           // reached by the solver as t
  if (flag != CV_SUCCESS)
    printf("CVode failed, flag=%d.\n", flag);
}

/// Write specified simulations steps to disk
void Simulation::outAllFieldData(const int & state) {
  checkFlag(SimulationStarted);
  outputManager.outUState(state, lattice, latticePatch);
}

/// Check the presence configuration flags
void Simulation::checkFlag(unsigned int flag) const {
  if (!(statusFlags & flag)) {
    string errorMessage;
    switch (flag) {
    case LatticeDiscreteSetUp:
      errorMessage = "The discrete size of the Simulation has not been set up";
      break;
    case LatticePhysicalSetUp:
      errorMessage = "The physical size of the Simulation has not been set up";
      break;
    case LatticePatchworkSetUp:
      errorMessage = "The patchwork for the Simulation has not been set up";
      break;
    case CvodeObjectSetUp:
      errorMessage = "The CVODE object has not been initialized";
      break;
    case SimulationStarted:
      errorMessage = "The Simulation has not been started";
      break;
    default:
      errorMessage = "Uppss, you've made a non-standard error, sadly I can't "
                     "help you there";
      break;
    }
    errorKill(errorMessage);
  }
  return;
}

/// Check the absence of configuration flags
void Simulation::checkNoFlag(unsigned int flag) const {
  if ((statusFlags & flag)) {
    string errorMessage;
    switch (flag) {
    case LatticeDiscreteSetUp:
      errorMessage =
          "The discrete size of the Simulation has already been set up";
      break;
    case LatticePhysicalSetUp:
      errorMessage =
          "The physical size of the Simulation has already been set up";
      break;
    case LatticePatchworkSetUp:
      errorMessage = "The patchwork for the Simulation has already been set up";
      break;
    case CvodeObjectSetUp:
      errorMessage = "The CVODE object has already been initialized";
      break;
    case SimulationStarted:
      errorMessage = "The simulation has already started, some changes are no "
                     "longer possible";
      break;
    default:
      errorMessage = "Uppss, you've made a non-standard error, sadly I can't "
                     "help you there";
      break;
    }
    errorKill(errorMessage);
```

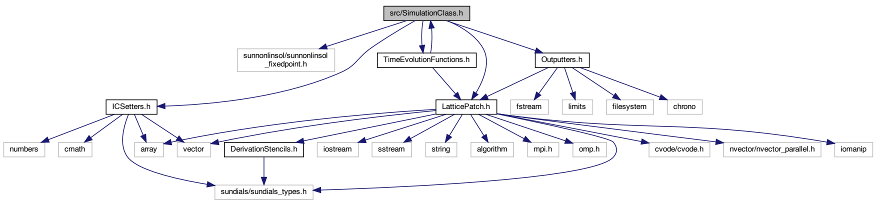
```
00305   }
00306   return;
00307 }
```

6.22 src/SimulationClass.h File Reference

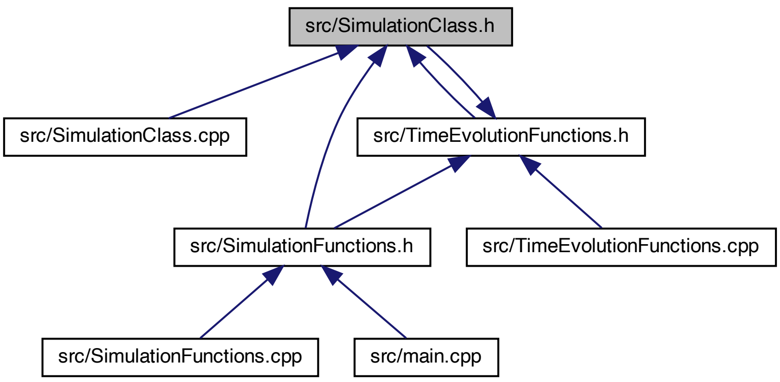
Class for the Simulation object calling all functionality: from wave settings over lattice construction, time evolution and outputs initialization of the CVode object.

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

Include dependency graph for SimulationClass.h:



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



Data Structures

- class Simulation

  Simulation class to instantiate the whole walkthrough of a Simulation.

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 CVode 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 ///////////////////////////////////////////////////////////////////////////////
```

```
/// @file SimulationClass.h
/// @brief Class for the Simulation object calling all functionality:
/// from wave settings over lattice construction, time evolution and outputs
/// initialization of the CVode object
////////////////////////////////////////////////////////////////////////////

#pragma once

/* access to the fixed point SUNNonlinearSolver */
#include "sunnonlinsol/sunnonlinsol_fixedpoint.h"

// project subfile headers
#include "ICSetters.h"
#include "LatticePatch.h"
#include "Outputters.h"
#include "TimeEvolutionFunctions.h"

using namespace std;

/// simulation checking flags
enum SimulationOptions {
  LatticeDiscreteSetUp = 0x01,
  LatticePhysicalSetUp = 0x02,
  LatticePatchworkSetUp = 0x04, // not used anymore
  CvodeObjectSetUp = 0x08,
  SimulationStarted = 0x10
  /*OPT_B = 0x02,
  OPT_C = 0x04,
  OPT_D = 0x08,
  OPT_E = 0x10,
  OPT_F = 0x20,*/
};

/** @brief Simulation class to instantiate the whole walkthrough of a Simulation
 */
class Simulation {
private:
  /// Lattice object
  Lattice lattice;
  /// LatticePatch object
  LatticePatch latticePatch;
  /// current time of the simulation
  sunrealtype t;
  /// char for checking simulation flags
  unsigned char statusFlags;

public:
  /// IC Setter object
  ICSetter icsettings;
  /// Output Manager object
  OutputManager outputManager;
  /// Pointer to CVode memory object -- public to avoid cross library errors
  void *cvode_mem;
  /// constructor function for the creation of the cartesian communicator
  Simulation(const int nx, const int ny, const int nz, const int StencilOrder,
          const bool periodicity);
  /// destructor function freeing CVode memory and Sundials context
  ~Simulation();
  /// Reference to the cartesian communicator of the lattice -> for debugging
  MPI_Comm *get_cart_comm() { return &lattice.comm; };
  /// function to set discrete dimensions of the lattice
  void setDiscreteDimensionsOfLattice(const sunindextype _tot_nx,
          const sunindextype _tot_ny, const sunindextype _tot_nz);
  /// function to set physical dimensions of the lattice
  void setPhysicalDimensionsOfLattice(const sunrealtype lx, const sunrealtype ly,
                                      const sunrealtype lz);
  /// function to initialize the Patchwork
  void initializePatchwork(const int nx, const int ny, const int nz);
  /// function to initialize the CVODE object with all requirements
  void initializeCVODEobject(const sunrealtype reltol,
                             const sunrealtype abstol);
  /// function to start the simulation for time iteration
  void start();
  /// functions to set the initial field configuration onto the lattice
  void setInitialConditions();
  /// functions to add initial periodic field configurations
  void addInitialConditions(const int xm, const int ym, const int zm = 0);
  /// function to add a periodic IC Layer in one dimension
  void addPeriodicICLayerInX();
  /// function to add periodic IC Layers in two dimensions
  void addPeriodicICLayerInXY();
  /// function to advance solution in time with CVODE
  void advanceToTime(const sunrealtype &tEnd);
  /// function to generate Output of the whole field at a given time
  void outAllFieldData(const int & state);
  /// function to check that a flag has been set and if not print an error
  // message and cause an abort on all ranks
```

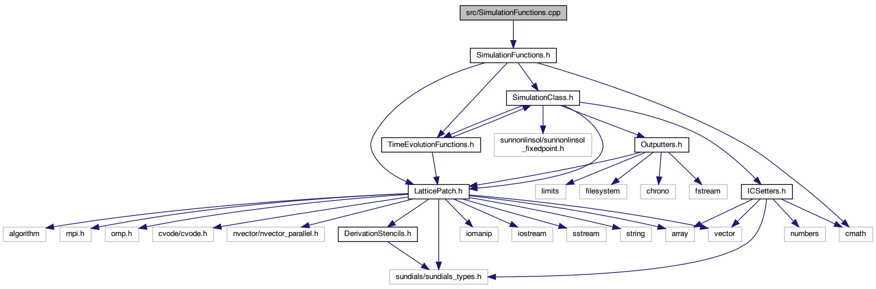
```
  void checkFlag(unsigned int flag) const;
  /// function to check that if flag has not been set and if print an error
  // message and cause an abort on all ranks
  void checkNoFlag(unsigned int flag) const;
};

```

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 > 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
- void Sim2D (const array< sunrealtype, 2 > CVodeTol, 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 > CVodeTol, 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 > 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.

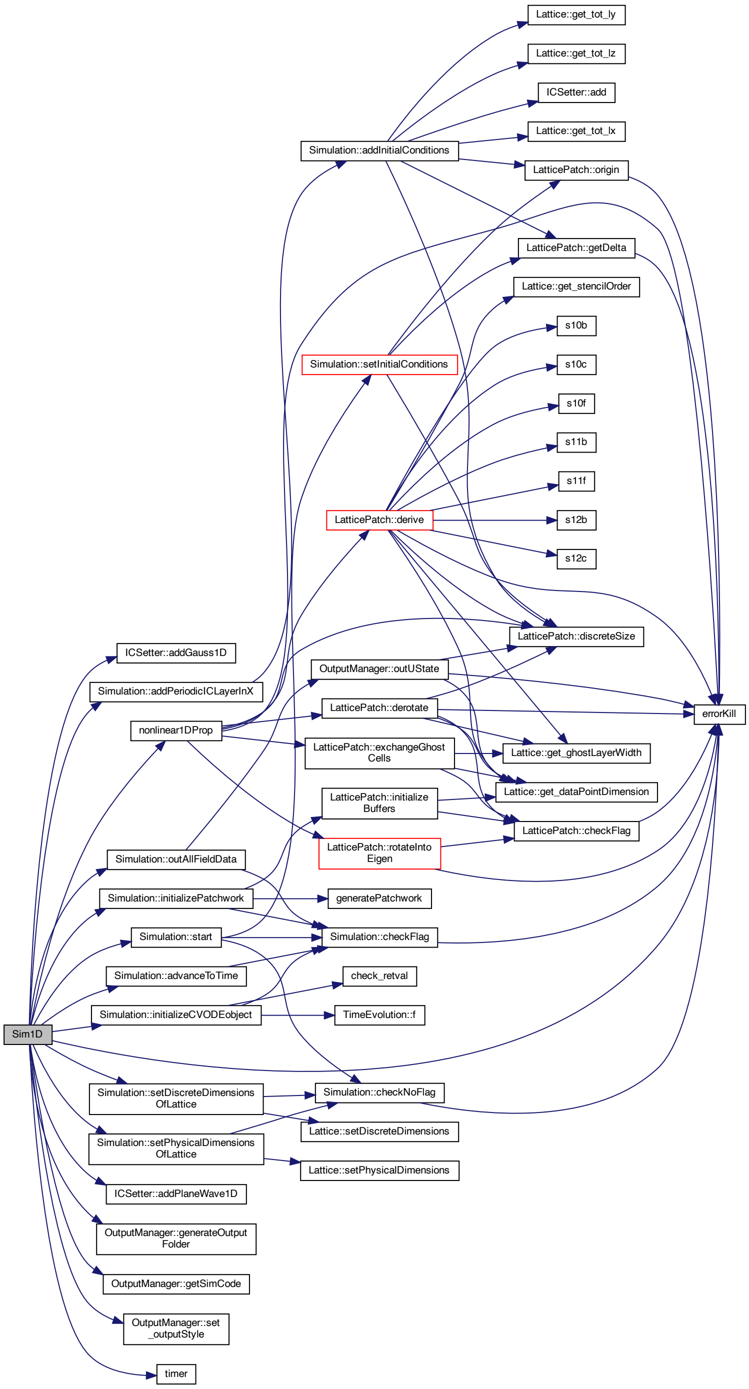
```
                                               {

  // MPI data
  int myPrc = 0, nprc = 0;
  MPI_Comm_size(MPI_COMM_WORLD, &nprc);
  MPI_Comm_rank(MPI_COMM_WORLD, &myPrc);

  // Check feasibility of the patchwork decomposition
  if (myPrc == 0) {
      if (disc_dim % nprc != 0) {
        errorKill("The number of lattice points must be "
                  "divisible by the number of processes.");
      }
  }

  // Initialize the simulation, set up the cartesian communicator
  array<int, 3> patches = {nprc, 1, 1};
  Simulation sim(patches[0], patches[1], patches[2], StencilOrder, periodic);

  // Configure the patchwork
  sim.setPhysicalDimensionsOfLattice(phys_dim,1,1);
  sim.setDiscreteDimensionsOfLattice(disc_dim,1,1);
  sim.initializePatchwork(patches[0], patches[1], patches[2]);

  // Add em-waves
  for (const auto &gauss : gaussians)
    sim.icsettings.addGauss1D(gauss.k, gauss.p, gauss.x0, gauss.phig,
                              gauss.phi);
  for (const auto &plane : planes)
    sim.icsettings.addPlaneWave1D(plane.k, plane.p, plane.phi);

  // Check that the patchwork is ready and set the initial conditions
  sim.start();
  sim.addPeriodicICLayerInX();

  // Initialize CVode with abs and rel tolerances
  sim.initializeCVODEobject(CVodeTol[0], CVodeTol[1]);

  // Configure the time evolution function
  TimeEvolution::c = interactions;
  TimeEvolution::TimeEvolver = nonlinear1DProp;

  // Configure the output
  sim.outputManager.generateOutputFolder(outputDirectory);
  if (!myPrc) {
```

```
    cout « "Simulation code: " « sim.outputManager.getSimCode() « endl;
  }
  sim.outputManager.set_outputStyle(outputStyle);

  double ts = MPI_Wtime();

  //sim.outAllFieldData(0);  // output of initial state
  // Conduct the propagation in space and time
  for (int step = 1; step <= numberOfSteps; step++) {
    sim.advanceToTime(endTime / numberOfSteps * step);
    if (step % outputStep == 0) {
      sim.outAllFieldData(step);
    }
    double tn = MPI_Wtime();
    if (!myPrc) {
      cout « "\rStep " « step « "\t\t" « flush;
      timer(ts, tn);
    }
  }

  return;
}
```

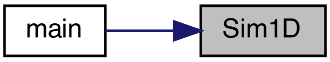
References ICSetter::addGauss1D(), Simulation::addPeriodicICLayerInX(), ICSetter::addPlaneWave1D(), Simulation::advanceToTime(), TimeEvolution::c, errorKill(), OutputManager::generateOutputFolder(), OutputManager::getSimCode(), Simulation::icsettings, Simulation::initializeCVODEobject(), Simulation::initializePatchwork(), nonlinear1DProp(), Simulation::outAllFieldData(), Simulation::outputManager, OutputManager::set_outputStyle(), Simulation::setDiscreteDimensionsOfLa 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 > CVodeTol,
            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.

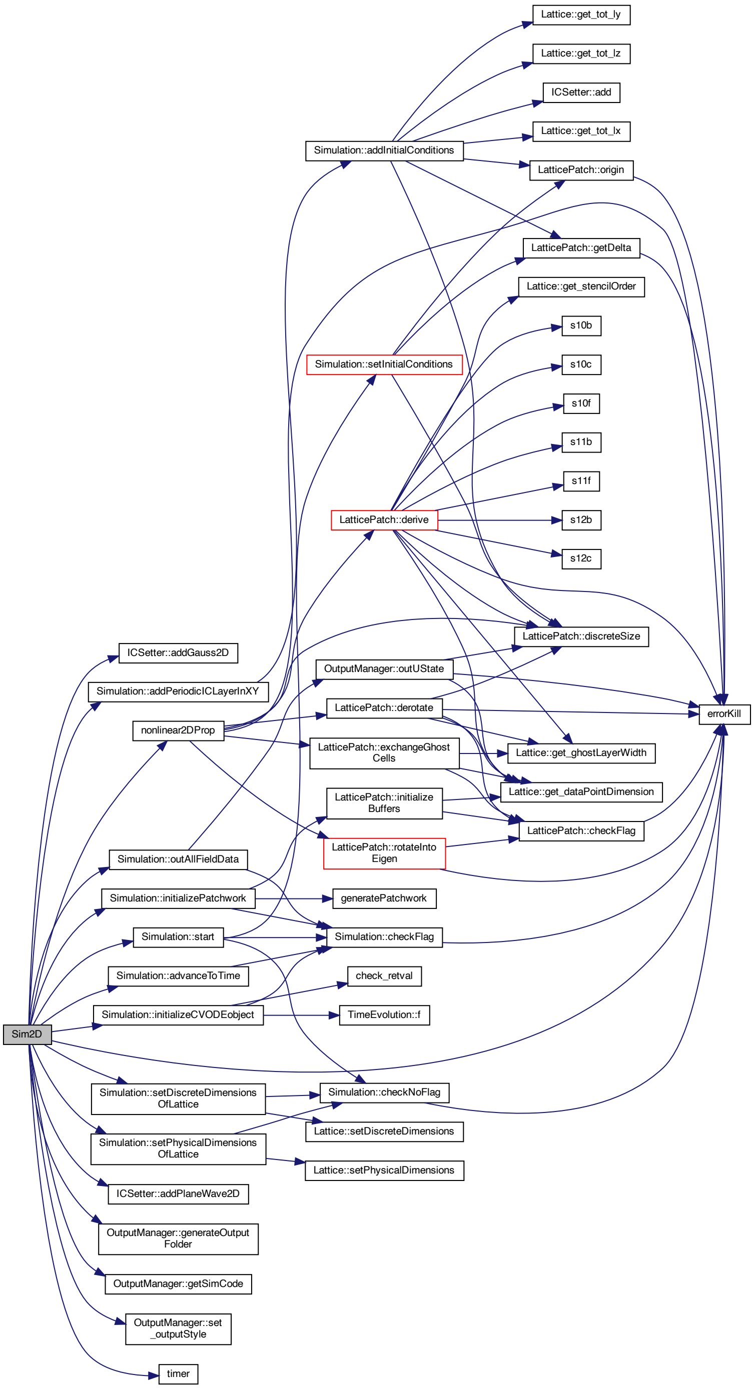
```
                                                                          {

  // MPI data
  int myPrc = 0, nprc = 0; // Get process rank and number of processes
  MPI_Comm_rank(MPI_COMM_WORLD,
                &myPrc); // Return process rank, number \in [1,nprc]
  MPI_Comm_size(MPI_COMM_WORLD,
                &nprc); // Return number of processes (communicator size)

  // Check feasibility of the patchwork decomposition
  if (myPrc == 0) {
    if (nprc != patches[0] * patches[1]) {
      errorKill(
          "The number of MPI processes must match the number of patches.");
    }
  }

  // Initialize the simulation, set up the cartesian communicator
  Simulation sim(patches[0], patches[1], 1, StencilOrder, periodic);

  /* // Check that lattice communicator is unique; same as used in patchwork
  generation char cart_comm_name[MPI_MAX_OBJECT_NAME]; int cart_namelen;
  MPI_Comm_get_name(*sim.get_cart_comm(), cart_comm_name, &cart_namelen);
  printf("sim.get_cart_comm gives %s \n", cart_comm_name);
  */

  // Configure the patchwork
  sim.setPhysicalDimensionsOfLattice(phys_dims[0],
```

```
                                     phys_dims[1],
                                     1); // spacing of the lattice
  sim.setDiscreteDimensionsOfLattice(
      disc_dims[0], disc_dims[1], 1); // Spacing equivalence to points
  sim.initializePatchwork(patches[0], patches[1], 1);

  // Add em-waves
  for (const auto &gauss : gaussians)
    sim.icsettings.addGauss2D(gauss.x0, gauss.axis, gauss.amp, gauss.phip,
                              gauss.w0, gauss.zr, gauss.ph0, gauss.phA);
  for (const auto &plane : planes)
    sim.icsettings.addPlaneWave2D(plane.k, plane.p, plane.phi);

  // Check that the patchwork is ready and set the initial conditions
  sim.start(); // Check if the lattice is set up, set initial field
               // configuration
  sim.addPeriodicICLayerInXY(); // insure periodicity in propagation directions

  // Initialize CVode with rel and abs tolerances
  sim.initializeCVODEobject(CVodeTol[0], CVodeTol[1]);

  // Configure the time evolution function
  TimeEvolution::c = interactions;
  TimeEvolution::TimeEvolver = nonlinear2DProp;

  // Configure the output
  sim.outputManager.generateOutputFolder(outputDirectory);
  if (!myPrc) {
    cout « "Simulation code: " « sim.outputManager.getSimCode() « endl;
  }
  sim.outputManager.set_outputStyle(outputStyle);

  double ts = MPI_Wtime();

  //sim.outAllFieldData(0);  // output of initial state
  // Conduct the propagation in space and time
  for (int step = 1; step <= numberOfSteps; step++) {
    sim.advanceToTime(endTime / numberOfSteps * step);
    if (step % outputStep == 0) {
      sim.outAllFieldData(step);
    }
    double tn = MPI_Wtime();
    if (!myPrc) {
      cout « "\rStep " « step « "\t\t" « flush;
      timer(ts, tn);
    }
  }

  return;
}
```

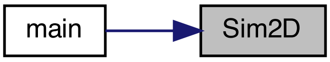
References ICSetter::addGauss2D(), Simulation::addPeriodicICLayerInXY(), ICSetter::addPlaneWave2D(), Simulation::advanceToTime(), TimeEvolution::c, errorKill(), OutputManager::generateOutputFolder(), OutputManager::getSimCode(), Simulation::icsettings, Simulation::initializeCVODEobject(), Simulation::initializePatchwork(), nonlinear2DProp(), Simulation::outAllFieldData(), Simulation::outputManager, OutputManager::set_outputStyle(), Simulation::setDiscreteDimensionsOfLa 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 > CVodeTol,
            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.

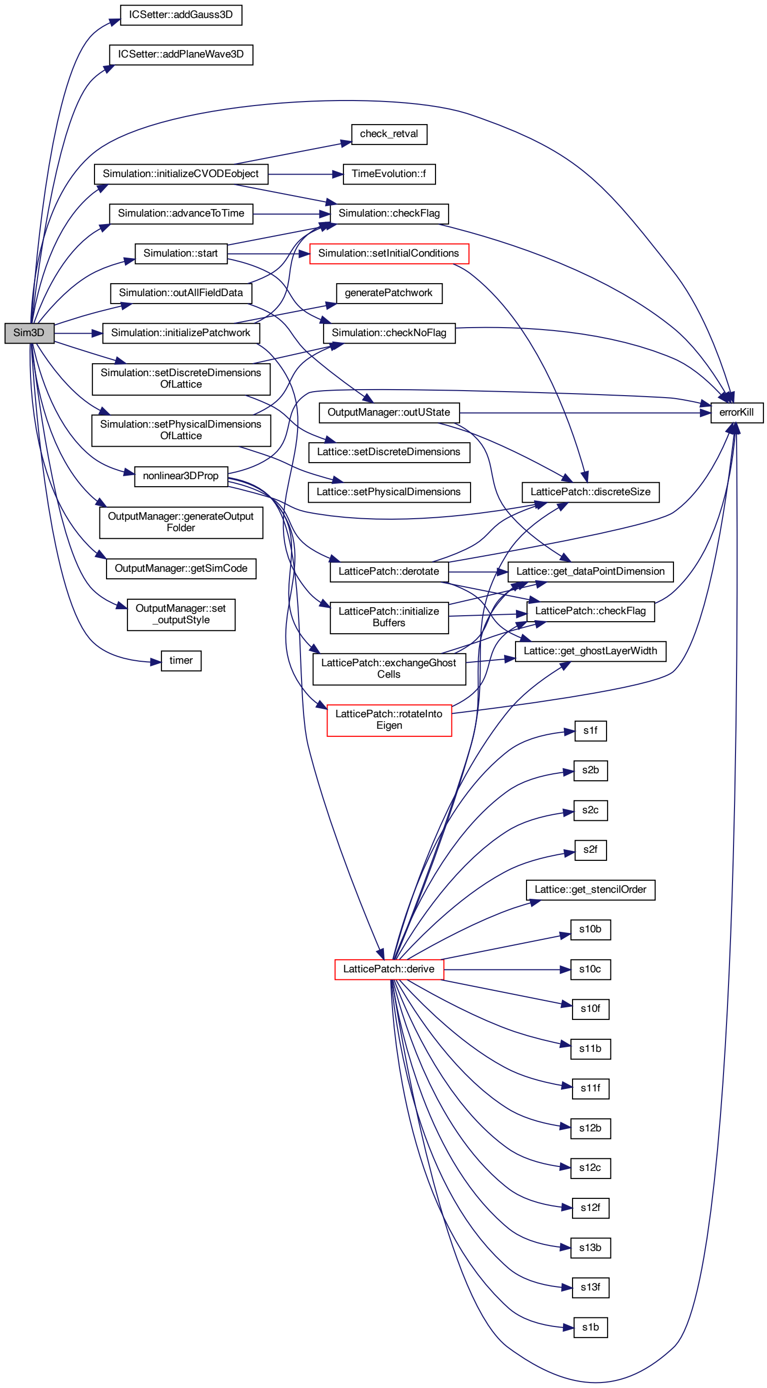
```
                                                                              {

  // MPI data
  int myPrc = 0, nprc = 0; // Get process rank and numer of process
  MPI_Comm_rank(MPI_COMM_WORLD,
                &myPrc); // rank of the process inside the world communicator
  MPI_Comm_size(MPI_COMM_WORLD,
               &nprc); // Size of the communicator is the number of processes

  // Check feasibility of the patchwork decomposition
  if (myPrc == 0) {
    if (nprc != patches[0] * patches[1] * patches[2]) {
      errorKill(
          "The number of MPI processes must match the number of patches.");
    }
    if ( ( disc_dims[0] / patches[0] != disc_dims[1] / patches[1] ) |
         ( disc_dims[0] / patches[0] != disc_dims[2] / patches[2] ) ) {
      clog
          « "\nWarning: Patches should be cubic in terms of the lattice "
             "points for the computational efficiency of larger simulations.\n";
    }
  }

  // Initialize the simulation, set up the cartesian communicator
  Simulation sim(patches[0], patches[1], patches[2],
                 StencilOrder, periodic); // Simulation object with slicing

  // Create the SUNContext object associated with the thread of execution
```

```
  sim.setPhysicalDimensionsOfLattice(phys_dims[0], phys_dims[1],
                                     phys_dims[2]); // spacing of the box
  sim.setDiscreteDimensionsOfLattice(
      disc_dims[0], disc_dims[1],
      disc_dims[2]); // Spacing equivalence to points
  sim.initializePatchwork(patches[0], patches[1], patches[2]);

  // Add em-waves
  for (const auto &plane : planes)
    sim.icsettings.addPlaneWave3D(plane.k, plane.p, plane.phi);
  for (const auto &gauss : gaussians)
    sim.icsettings.addGauss3D(gauss.x0, gauss.axis, gauss.amp, gauss.phip,
                              gauss.w0, gauss.zr, gauss.ph0, gauss.phA);

  // Check that the patchwork is ready and set the initial conditions
  sim.start();

  // Initialize CVode with abs and rel tolerances
  sim.initializeCVODEobject(CVodeTol[0], CVodeTol[1]);

  // Configure the time evolution function
  TimeEvolution::c = interactions;
  TimeEvolution::TimeEvolver = nonlinear3DProp;

  // Configure the output
  sim.outputManager.generateOutputFolder(outputDirectory);
  if (!myPrc) {
    cout « "Simulation code: " « sim.outputManager.getSimCode() « endl;
  }
  sim.outputManager.set_outputStyle(outputStyle);

  double ts = MPI_Wtime();

  //sim.outAllFieldData(0);  // output of initial state
  // Conduct the propagation in space and time
  for (int step = 1; step <= numberOfSteps; step++) {
    sim.advanceToTime(endTime / numberOfSteps * step);
    if (step % outputStep == 0) {
      sim.outAllFieldData(step);
    }
    double tn = MPI_Wtime();
    if (!myPrc) {
      cout « "\rStep " « step « "\t\t" « flush;
      timer(ts, tn);
    }
  }
  return;
}
```

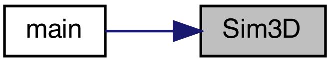
References ICSetter::addGauss3D(), ICSetter::addPlaneWave3D(), Simulation::advanceToTime(), TimeEvolution::c, errorKill(), OutputManager::generateOutputFolder(), OutputManager::getSimCode(), Simulation::icsettings, Simulation::initializeCVODEobject(), Simulation::initializePatchwork(), nonlinear3DProp(),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

Referenced by main().

Here is the call graph for this function:



Here is the caller graph for this function:



6.24.2.4 timer()

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

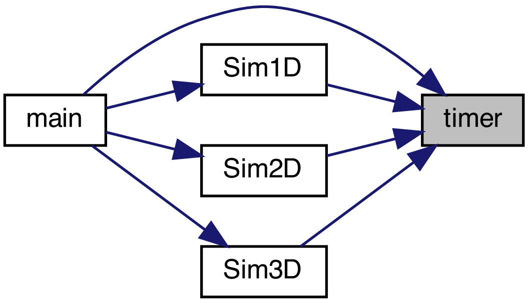
Calculate and print the total simulation time

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.

```
///////////////////////////////////////////////////////////////////////
/// @file SimulationFunctions.cpp
/// @brief Implementation of the complete simulation functions for
/// 1D, 2D, and 3D, as called in the main function
///////////////////////////////////////////////////////////////////////

#include "SimulationFunctions.h"

using namespace std;

/** Calculate and print the total simulation time */
void timer(double &t1, double &t2) {
  printf("Elapsed time:  %fs\n", (t2 - t1));
}

// Instantiate and preliminarily initialize the time evolver
// non-const statics to be defined in actual simulation process
int *TimeEvolution::c = nullptr;
void (*TimeEvolution::TimeEvolver)(LatticePatch *, N_Vector, N_Vector,
                                   int *) = nonlinear1DProp;

/** Conduct the complete 1D simulation process */
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) {

  // MPI data
  int myPrc = 0, nprc = 0;
  MPI_Comm_size(MPI_COMM_WORLD, &nprc);
  MPI_Comm_rank(MPI_COMM_WORLD, &myPrc);

  // Check feasibility of the patchwork decomposition
  if (myPrc == 0) {
      if (disc_dim % nprc != 0) {
        errorKill("The number of lattice points must be "
                  "divisible by the number of processes.");
      }
  }

  // Initialize the simulation, set up the cartesian communicator
  array<int, 3> patches = {nprc, 1, 1};
  Simulation sim(patches[0], patches[1], patches[2], StencilOrder, periodic);

  // Configure the patchwork
  sim.setPhysicalDimensionsOfLattice(phys_dim,1,1);
  sim.setDiscreteDimensionsOfLattice(disc_dim,1,1);
  sim.initializePatchwork(patches[0], patches[1], patches[2]);

  // Add em-waves
  for (const auto &gauss : gaussians)
    sim.icsettings.addGauss1D(gauss.k, gauss.p, gauss.x0, gauss.phig,
                              gauss.phi);
  for (const auto &plane : planes)
    sim.icsettings.addPlaneWave1D(plane.k, plane.p, plane.phi);

  // Check that the patchwork is ready and set the initial conditions
  sim.start();
  sim.addPeriodicICLayerInX();

  // Initialize CVode with abs and rel tolerances
  sim.initializeCVODEobject(CVodeTol[0], CVodeTol[1]);

  // Configure the time evolution function
  TimeEvolution::c = interactions;
  TimeEvolution::TimeEvolver = nonlinear1DProp;

  // Configure the output
  sim.outputManager.generateOutputFolder(outputDirectory);
  if (!myPrc) {
    cout « "Simulation code: " « sim.outputManager.getSimCode() « endl;
  }
  sim.outputManager.set_outputStyle(outputStyle);

  double ts = MPI_Wtime();

  //sim.outAllFieldData(0);  // output of initial state
  // Conduct the propagation in space and time
```

```
  for (int step = 1; step <= numberOfSteps; step++) {
    sim.advanceToTime(endTime / numberOfSteps * step);
    if (step % outputStep == 0) {
      sim.outAllFieldData(step);
    }
    double tn = MPI_Wtime();
    if (!myPrc) {
      cout « "\rStep " « step « "\t\t" « flush;
      timer(ts, tn);
    }
  }

  return;
}

/** Conduct the complete 2D simulation process */
void Sim2D(const array<sunrealtype,2> CVodeTol, 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) {

  // MPI data
  int myPrc = 0, nprc = 0; // Get process rank and number of processes
  MPI_Comm_rank(MPI_COMM_WORLD,
                &myPrc); // Return process rank, number \in [1,nprc]
  MPI_Comm_size(MPI_COMM_WORLD,
                &nprc); // Return number of processes (communicator size)

  // Check feasibility of the patchwork decomposition
  if (myPrc == 0) {
    if (nprc != patches[0] * patches[1]) {
      errorKill(
          "The number of MPI processes must match the number of patches.");
    }
  }

  // Initialize the simulation, set up the cartesian communicator
  Simulation sim(patches[0], patches[1], 1, StencilOrder, periodic);

  /* // Check that lattice communicator is unique; same as used in patchwork
  generation char cart_comm_name[MPI_MAX_OBJECT_NAME]; int cart_namelen;
  MPI_Comm_get_name(*sim.get_cart_comm(), cart_comm_name, &cart_namelen);
  printf("sim.get_cart_comm gives %s \n", cart_comm_name);
  */

  // Configure the patchwork
  sim.setPhysicalDimensionsOfLattice(phys_dims[0],
                                     phys_dims[1],
                                     1); // spacing of the lattice
  sim.setDiscreteDimensionsOfLattice(
      disc_dims[0], disc_dims[1], 1); // Spacing equivalence to points
  sim.initializePatchwork(patches[0], patches[1], 1);

  // Add em-waves
  for (const auto &gauss : gaussians)
    sim.icsettings.addGauss2D(gauss.x0, gauss.axis, gauss.amp, gauss.phip,
                              gauss.w0, gauss.zr, gauss.ph0, gauss.phA);
  for (const auto &plane : planes)
    sim.icsettings.addPlaneWave2D(plane.k, plane.p, plane.phi);

  // Check that the patchwork is ready and set the initial conditions
  sim.start(); // Check if the lattice is set up, set initial field
               // configuration
  sim.addPeriodicICLayerInXY(); // insure periodicity in propagation directions

  // Initialize CVode with rel and abs tolerances
  sim.initializeCVODEobject(CVodeTol[0], CVodeTol[1]);

  // Configure the time evolution function
  TimeEvolution::c = interactions;
  TimeEvolution::TimeEvolver = nonlinear2DProp;

  // Configure the output
  sim.outputManager.generateOutputFolder(outputDirectory);
  if (!myPrc) {
    cout « "Simulation code: " « sim.outputManager.getSimCode() « endl;
  }
  sim.outputManager.set_outputStyle(outputStyle);

  double ts = MPI_Wtime();

  //sim.outAllFieldData(0);  // output of initial state
  // Conduct the propagation in space and time
  for (int step = 1; step <= numberOfSteps; step++) {
    sim.advanceToTime(endTime / numberOfSteps * step);
```

```
    if (step % outputStep == 0) {
      sim.outAllFieldData(step);
    }
    double tn = MPI_Wtime();
    if (!myPrc) {
      cout « "\rStep " « step « "\t\t" « flush;
      timer(ts, tn);
    }
  }

  return;
}

/** Conduct the complete 3D simulation process */
void Sim3D(const array<sunrealtype,2> CVodeTol, 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) {

  // MPI data
  int myPrc = 0, nprc = 0; // Get process rank and numer of process
  MPI_Comm_rank(MPI_COMM_WORLD,
                &myPrc); // rank of the process inside the world communicator
  MPI_Comm_size(MPI_COMM_WORLD,
                &nprc); // Size of the communicator is the number of processes

  // Check feasibility of the patchwork decomposition
  if (myPrc == 0) {
    if (nprc != patches[0] * patches[1] * patches[2]) {
      errorKill(
          "The number of MPI processes must match the number of patches.");
    }
    if ( ( disc_dims[0] / patches[0] != disc_dims[1] / patches[1] ) |
         ( disc_dims[0] / patches[0] != disc_dims[2] / patches[2] ) ) {
      clog
          « "\nWarning: Patches should be cubic in terms of the lattice "
             "points for the computational efficiency of larger simulations.\n";
    }
  }

  // Initialize the simulation, set up the cartesian communicator
  Simulation sim(patches[0], patches[1], patches[2],
                 StencilOrder, periodic); // Simulation object with slicing

  // Create the SUNContext object associated with the thread of execution
  sim.setPhysicalDimensionsOfLattice(phys_dims[0], phys_dims[1],
                                     phys_dims[2]); // spacing of the box
  sim.setDiscreteDimensionsOfLattice(
      disc_dims[0], disc_dims[1],
      disc_dims[2]); // Spacing equivalence to points
  sim.initializePatchwork(patches[0], patches[1], patches[2]);

  // Add em-waves
  for (const auto &plane : planes)
    sim.icsettings.addPlaneWave3D(plane.k, plane.p, plane.phi);
  for (const auto &gauss : gaussians)
    sim.icsettings.addGauss3D(gauss.x0, gauss.axis, gauss.amp, gauss.phip,
                              gauss.w0, gauss.zr, gauss.ph0, gauss.phA);

  // Check that the patchwork is ready and set the initial conditions
  sim.start();

  // Initialize CVode with abs and rel tolerances
  sim.initializeCVODEobject(CVodeTol[0], CVodeTol[1]);

  // Configure the time evolution function
  TimeEvolution::c = interactions;
  TimeEvolution::TimeEvolver = nonlinear3DProp;

  // Configure the output
  sim.outputManager.generateOutputFolder(outputDirectory);
  if (!myPrc) {
    cout « "Simulation code: " « sim.outputManager.getSimCode() « endl;
  }
  sim.outputManager.set_outputStyle(outputStyle);

  double ts = MPI_Wtime();

  //sim.outAllFieldData(0);  // output of initial state
  // Conduct the propagation in space and time
  for (int step = 1; step <= numberOfSteps; step++) {
    sim.advanceToTime(endTime / numberOfSteps * step);
    if (step % outputStep == 0) {
      sim.outAllFieldData(step);
```

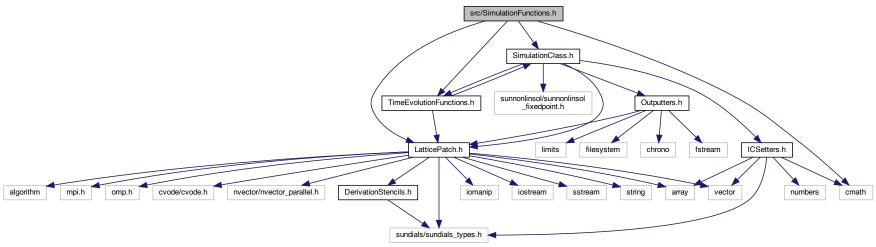
```
    }
    double tn = MPI_Wtime();
    if (!myPrc) {
      cout « "\rStep " « step « "\t\t" « flush;
      timer(ts, tn);
    }
  }
  return;
}
```

6.26 src/SimulationFunctions.h File Reference

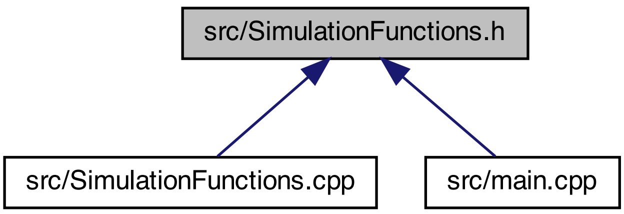
Full simulation functions for 1D, 2D, and 3D used in main.cpp.

```
#include <cmath>
#include "LatticePatch.h"
#include "SimulationClass.h"
#include "TimeEvolutionFunctions.h"
```

Include dependency graph for SimulationFunctions.h:



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



Data Structures

- struct planewave

    plane wave structure
- struct gaussian1D

    1D Gaussian wave structure
- struct gaussian2D

    2D Gaussian wave structure
- struct gaussian3D

    3D Gaussian wave structure

Functions

- void Sim1D (const array< sunrealtype, 2 >, const int, const sunrealtype, const sunindextype, const bool, int ∗, const sunrealtype, const int, const string, const int, const 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.

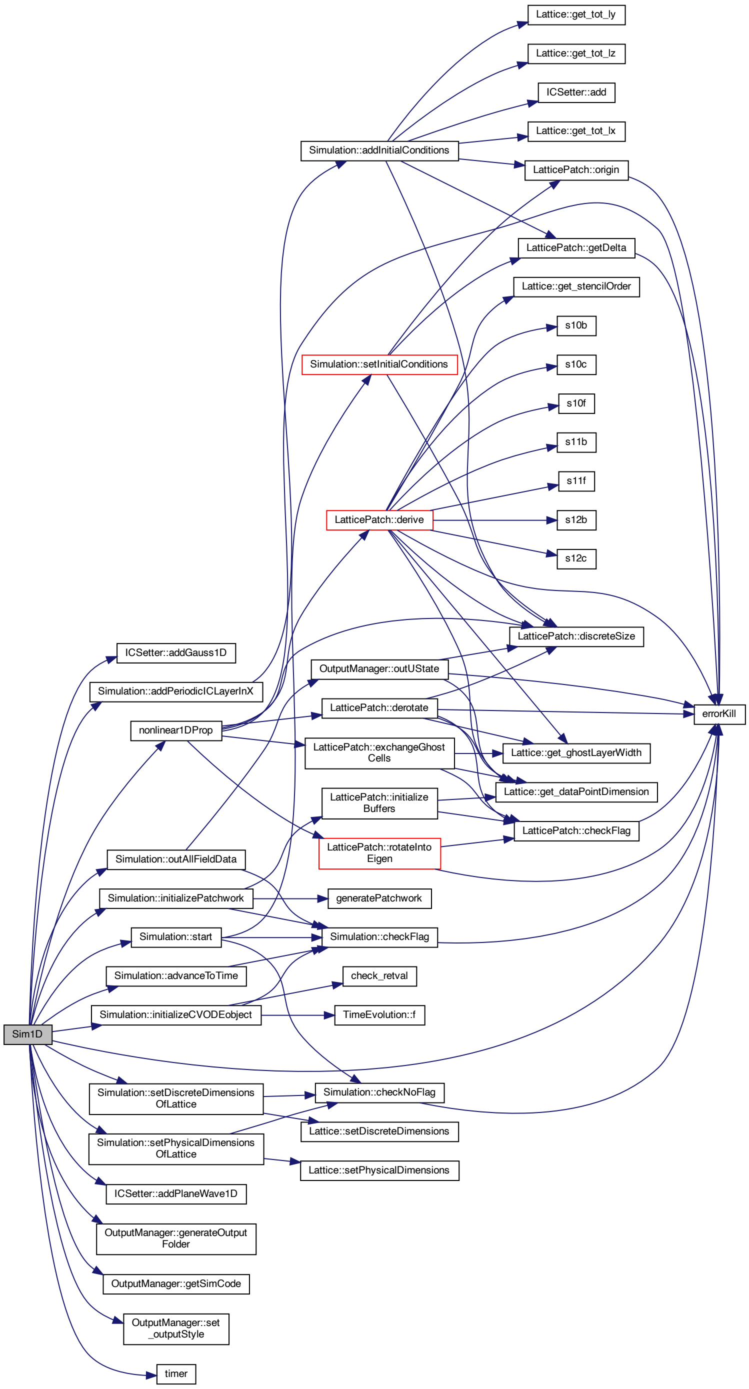
```
                                            {

  // MPI data
  int myPrc = 0, nprc = 0;
  MPI_Comm_size(MPI_COMM_WORLD, &nprc);
  MPI_Comm_rank(MPI_COMM_WORLD, &myPrc);
```

```

  // Check feasibility of the patchwork decomposition
  if (myPrc == 0) {
      if (disc_dim % nprc != 0) {
        errorKill("The number of lattice points must be "
                  "divisible by the number of processes.");
      }
  }

  // Initialize the simulation, set up the cartesian communicator
  array<int, 3> patches = {nprc, 1, 1};
  Simulation sim(patches[0], patches[1], patches[2], StencilOrder, periodic);

  // Configure the patchwork
  sim.setPhysicalDimensionsOfLattice(phys_dim,1,1);
  sim.setDiscreteDimensionsOfLattice(disc_dim,1,1);
  sim.initializePatchwork(patches[0], patches[1], patches[2]);

  // Add em-waves
  for (const auto &gauss : gaussians)
    sim.icsettings.addGauss1D(gauss.k, gauss.p, gauss.x0, gauss.phig,
                              gauss.phi);
  for (const auto &plane : planes)
    sim.icsettings.addPlaneWave1D(plane.k, plane.p, plane.phi);

  // Check that the patchwork is ready and set the initial conditions
  sim.start();
  sim.addPeriodicICLayerInX();

  // Initialize CVode with abs and rel tolerances
  sim.initializeCVODEobject(CVodeTol[0], CVodeTol[1]);

  // Configure the time evolution function
  TimeEvolution::c = interactions;
  TimeEvolution::TimeEvolver = nonlinear1DProp;

  // Configure the output
  sim.outputManager.generateOutputFolder(outputDirectory);
  if (!myPrc) {
    cout « "Simulation code: " « sim.outputManager.getSimCode() « endl;
  }
  sim.outputManager.set_outputStyle(outputStyle);

  double ts = MPI_Wtime();

  //sim.outAllFieldData(0);  // output of initial state
  // Conduct the propagation in space and time
  for (int step = 1; step <= numberOfSteps; step++) {
    sim.advanceToTime(endTime / numberOfSteps * step);
    if (step % outputStep == 0) {
      sim.outAllFieldData(step);
    }
    double tn = MPI_Wtime();
    if (!myPrc) {
      cout « "\rStep " « step « "\t\t" « flush;
      timer(ts, tn);
    }
  }

  return;
}
```

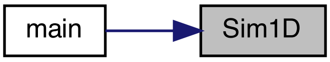
References ICSetter::addGauss1D(), Simulation::addPeriodicICLayerInX(), ICSetter::addPlaneWave1D(), Simulation::advanceToTime(), TimeEvolution::c, errorKill(), OutputManager::generateOutputFolder(), OutputManager::getSimCode(), Simulation::icsettings, Simulation::initializeCVODEobject(), Simulation::initializePatchwork(), nonlinear1DProp(), Simulation::outAllFieldData(), Simulation::outputManager, OutputManager::set_outputStyle(), Simulation::setDiscreteDimensionsOfLa 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 > CVodeTol,
        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.

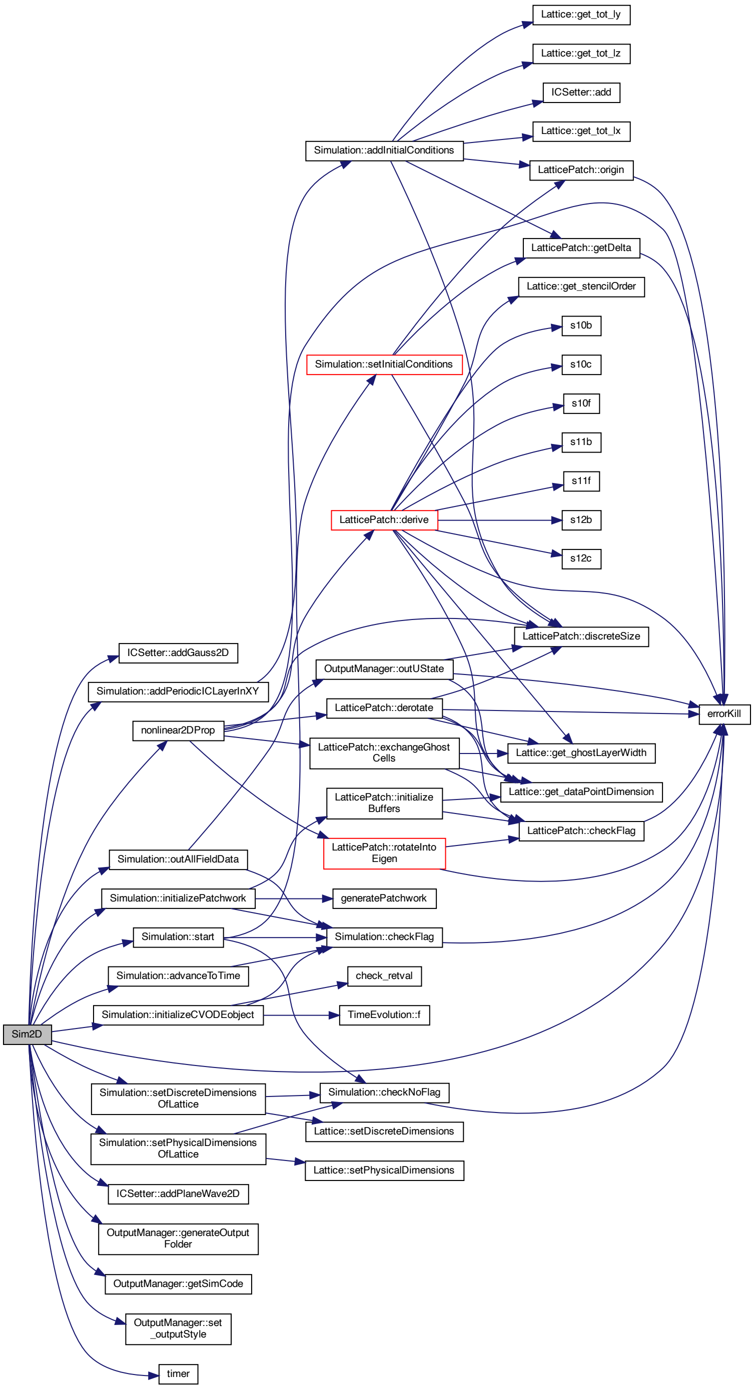
```
                                                                                   {

  // MPI data
  int myPrc = 0, nprc = 0; // Get process rank and number of processes
  MPI_Comm_rank(MPI_COMM_WORLD,
                &myPrc); // Return process rank, number \in [1,nprc]
  MPI_Comm_size(MPI_COMM_WORLD,
                &nprc); // Return number of processes (communicator size)

  // Check feasibility of the patchwork decomposition
  if (myPrc == 0) {
    if (nprc != patches[0] * patches[1]) {
      errorKill(
          "The number of MPI processes must match the number of patches.");
    }
  }

  // Initialize the simulation, set up the cartesian communicator
  Simulation sim(patches[0], patches[1], 1, StencilOrder, periodic);

  /* // Check that lattice communicator is unique; same as used in patchwork
  generation char cart_comm_name[MPI_MAX_OBJECT_NAME]; int cart_namelen;
  MPI_Comm_get_name(*sim.get_cart_comm(), cart_comm_name, &cart_namelen);
  printf("sim.get_cart_comm gives %s \n", cart_comm_name);
  */

  // Configure the patchwork
  sim.setPhysicalDimensionsOfLattice(phys_dims[0],
```

```
                                     phys_dims[1],
                                     1); // spacing of the lattice
  sim.setDiscreteDimensionsOfLattice(
      disc_dims[0], disc_dims[1], 1); // Spacing equivalence to points
  sim.initializePatchwork(patches[0], patches[1], 1);

  // Add em-waves
  for (const auto &gauss : gaussians)
    sim.icsettings.addGauss2D(gauss.x0, gauss.axis, gauss.amp, gauss.phip,
                              gauss.w0, gauss.zr, gauss.ph0, gauss.phA);
  for (const auto &plane : planes)
    sim.icsettings.addPlaneWave2D(plane.k, plane.p, plane.phi);

  // Check that the patchwork is ready and set the initial conditions
  sim.start(); // Check if the lattice is set up, set initial field
               // configuration
  sim.addPeriodicICLayerInXY(); // insure periodicity in propagation directions

  // Initialize CVode with rel and abs tolerances
  sim.initializeCVODEobject(CVodeTol[0], CVodeTol[1]);

  // Configure the time evolution function
  TimeEvolution::c = interactions;
  TimeEvolution::TimeEvolver = nonlinear2DProp;

  // Configure the output
  sim.outputManager.generateOutputFolder(outputDirectory);
  if (!myPrc) {
    cout « "Simulation code: " « sim.outputManager.getSimCode() « endl;
  }
  sim.outputManager.set_outputStyle(outputStyle);

  double ts = MPI_Wtime();

  //sim.outAllFieldData(0);  // output of initial state
  // Conduct the propagation in space and time
  for (int step = 1; step <= numberOfSteps; step++) {
    sim.advanceToTime(endTime / numberOfSteps * step);
    if (step % outputStep == 0) {
      sim.outAllFieldData(step);
    }
    double tn = MPI_Wtime();
    if (!myPrc) {
      cout « "\rStep " « step « "\t\t" « flush;
      timer(ts, tn);
    }
  }

  return;
}
```

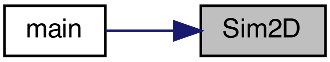
References ICSetter::addGauss2D(), Simulation::addPeriodicICLayerInXY(), ICSetter::addPlaneWave2D(), Simulation::advanceToTime(), TimeEvolution::c, errorKill(), OutputManager::generateOutputFolder(), OutputManager::getSimCode(), Simulation::icsettings, Simulation::initializeCVODEobject(), Simulation::initializePatchwork(), nonlinear2DProp(), Simulation::outAllFieldData(), Simulation::outputManager, OutputManager::set_outputStyle(), Simulation::setDiscreteDimensionsOfLa 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 > CVodeTol,
            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.

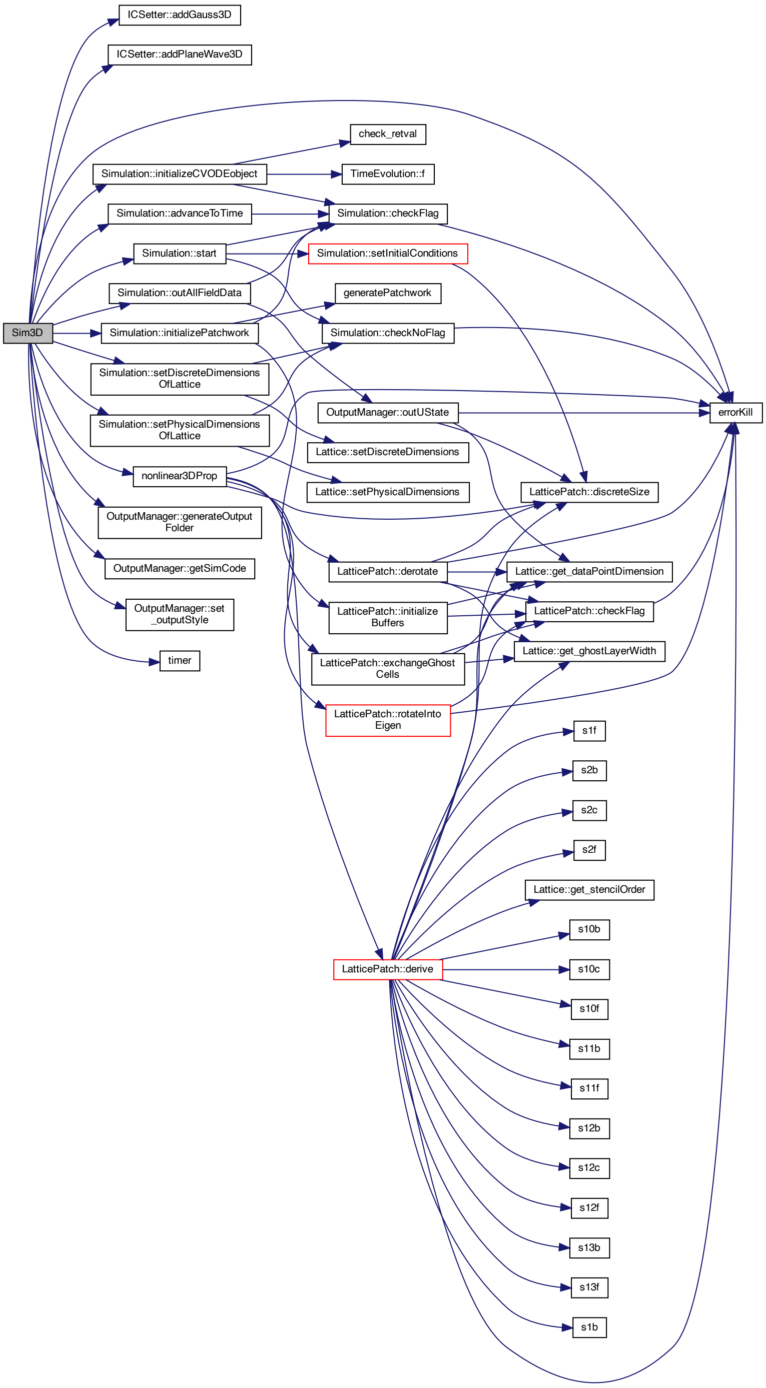
```
                                                                            {

  // MPI data
  int myPrc = 0, nprc = 0; // Get process rank and numer of process
  MPI_Comm_rank(MPI_COMM_WORLD,
                &myPrc); // rank of the process inside the world communicator
  MPI_Comm_size(MPI_COMM_WORLD,
                &nprc); // Size of the communicator is the number of processes

  // Check feasibility of the patchwork decomposition
  if (myPrc == 0) {
    if (nprc != patches[0] * patches[1] * patches[2]) {
      errorKill(
          "The number of MPI processes must match the number of patches.");
    }
    if ( ( disc_dims[0] / patches[0] != disc_dims[1] / patches[1] ) |
         ( disc_dims[0] / patches[0] != disc_dims[2] / patches[2] ) ) {
      clog
          « "\nWarning: Patches should be cubic in terms of the lattice "
             "points for the computational efficiency of larger simulations.\n";
    }
  }

  // Initialize the simulation, set up the cartesian communicator
  Simulation sim(patches[0], patches[1], patches[2],
                 StencilOrder, periodic); // Simulation object with slicing

  // Create the SUNContext object associated with the thread of execution
```

```
  sim.setPhysicalDimensionsOfLattice(phys_dims[0], phys_dims[1],
                                     phys_dims[2]); // spacing of the box
  sim.setDiscreteDimensionsOfLattice(
      disc_dims[0], disc_dims[1],
      disc_dims[2]); // Spacing equivalence to points
  sim.initializePatchwork(patches[0], patches[1], patches[2]);

  // Add em-waves
  for (const auto &plane : planes)
    sim.icsettings.addPlaneWave3D(plane.k, plane.p, plane.phi);
  for (const auto &gauss : gaussians)
    sim.icsettings.addGauss3D(gauss.x0, gauss.axis, gauss.amp, gauss.phip,
                              gauss.w0, gauss.zr, gauss.ph0, gauss.phA);

  // Check that the patchwork is ready and set the initial conditions
  sim.start();

  // Initialize CVode with abs and rel tolerances
  sim.initializeCVODEobject(CVodeTol[0], CVodeTol[1]);

  // Configure the time evolution function
  TimeEvolution::c = interactions;
  TimeEvolution::TimeEvolver = nonlinear3DProp;

  // Configure the output
  sim.outputManager.generateOutputFolder(outputDirectory);
  if (!myPrc) {
    cout « "Simulation code: " « sim.outputManager.getSimCode() « endl;
  }
  sim.outputManager.set_outputStyle(outputStyle);

  double ts = MPI_Wtime();

  //sim.outAllFieldData(0);  // output of initial state
  // Conduct the propagation in space and time
  for (int step = 1; step <= numberOfSteps; step++) {
    sim.advanceToTime(endTime / numberOfSteps * step);
    if (step % outputStep == 0) {
      sim.outAllFieldData(step);
    }
    double tn = MPI_Wtime();
    if (!myPrc) {
      cout « "\rStep " « step « "\t\t" « flush;
      timer(ts, tn);
    }
  }
  return;
}
```

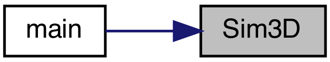
References ICSetter::addGauss3D(), ICSetter::addPlaneWave3D(), Simulation::advanceToTime(), TimeEvolution::c, errorKill(), OutputManager::generateOutputFolder(), OutputManager::getSimCode(), Simulation::icsettings, Simulation::initializeCVODEobject(), Simulation::initializePatchwork(), nonlinear3DProp(),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

Referenced by main().

Here is the call graph for this function:



Here is the caller graph for this function:



6.26.2.4 timer()

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

MPI timer function

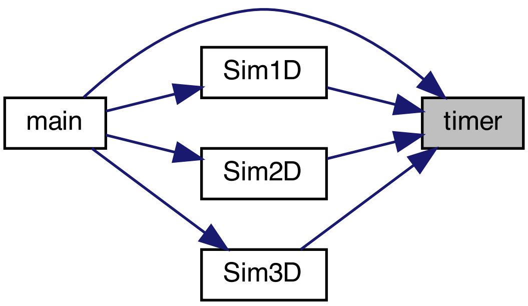
Calculate and print the total simulation time

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.

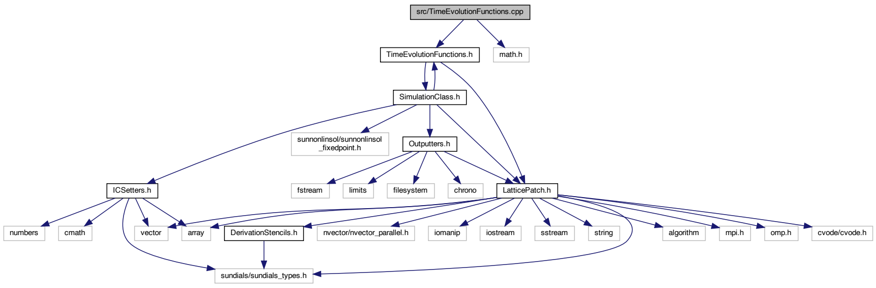
```
///////////////////////////////////////////////////////////////////////////
/// @file SimulationFunctions.h
/// @brief Full simulation functions for 1D, 2D, and 3D used in main.cpp
///////////////////////////////////////////////////////////////////////////

#pragma once

// math
#include <cmath>
//#include <mathimf.h>

// project subfile headers
#include "LatticePatch.h"
#include "SimulationClass.h"
#include "TimeEvolutionFunctions.h"

/****** EM-wave structures ******/

/// plane wave structure
struct planewave {
  vector<sunrealtype> k;   /**< wavevector (normalized to \f$ 1/\lambda \f$) */
  vector<sunrealtype> p;   /**< amplitde & polarization vector */
  vector<sunrealtype> phi; /**< phase shift */
};

/// 1D Gaussian wave structure
struct gaussian1D {
  vector<sunrealtype> k;   /**< wavevector (normalized to \f$ 1/\lambda \f$) */
  vector<sunrealtype> p;   /**< amplitude & polarization vector */
  vector<sunrealtype> x0;  /**< shift from origin */
  sunrealtype phig;        /**< width */
  vector<sunrealtype> phi; /**< phase shift */
};

/// 2D Gaussian wave structure
struct gaussian2D {
  vector<sunrealtype> x0;   /**< center */
  vector<sunrealtype> axis; /**< direction to center */
  sunrealtype amp;          /**< amplitude */
  sunrealtype phip;         /**< polarization rotation */
  sunrealtype w0;           /**< taille */
  sunrealtype zr;           /**< Rayleigh length */
  sunrealtype ph0;          /**< beam center */
  sunrealtype phA;          /**< beam length */
};

/// 3D Gaussian wave structure
struct gaussian3D {
  vector<sunrealtype> x0;   /**< center */
  vector<sunrealtype> axis; /**< direction to center */
  sunrealtype amp;          /**< amplitude */
  sunrealtype phip;         /**< polarization rotation */
  sunrealtype w0;           /**< taille */
  sunrealtype zr;           /**< Rayleigh length */
  sunrealtype ph0;          /**< beam center */
  sunrealtype phA;          /**< beam length */
};

/****** simulation function declarations ******/

/// complete 1D Simulation function
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 2D 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 3D 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> &);

/** MPI timer function */
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

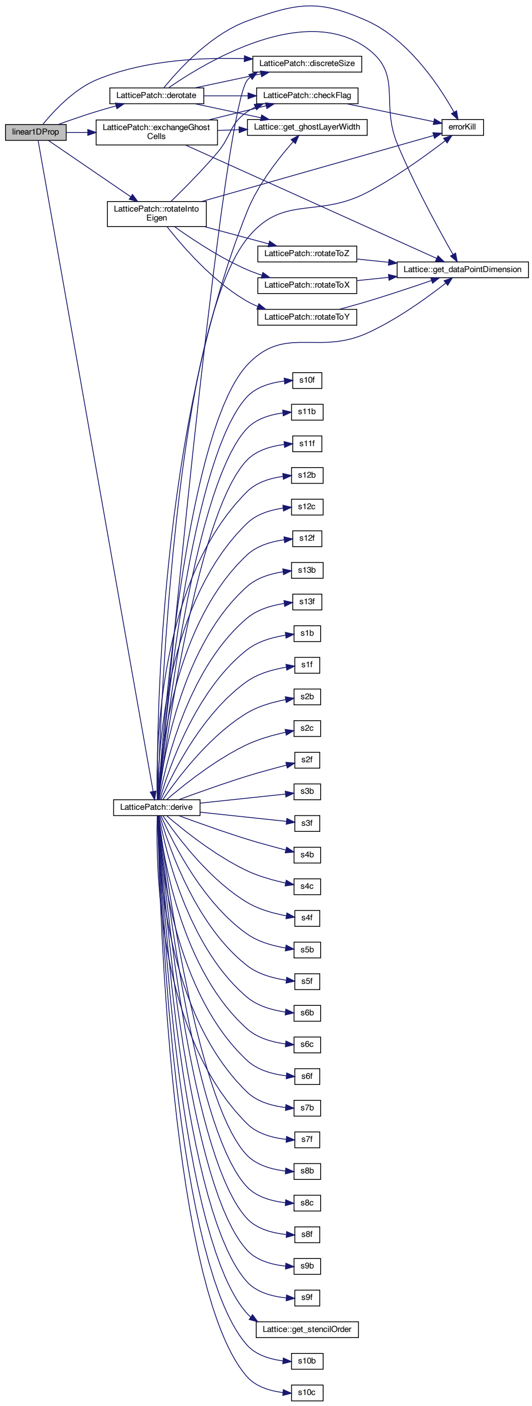
Maxwell propagation function for 1D – only for reference.

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 deriative 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

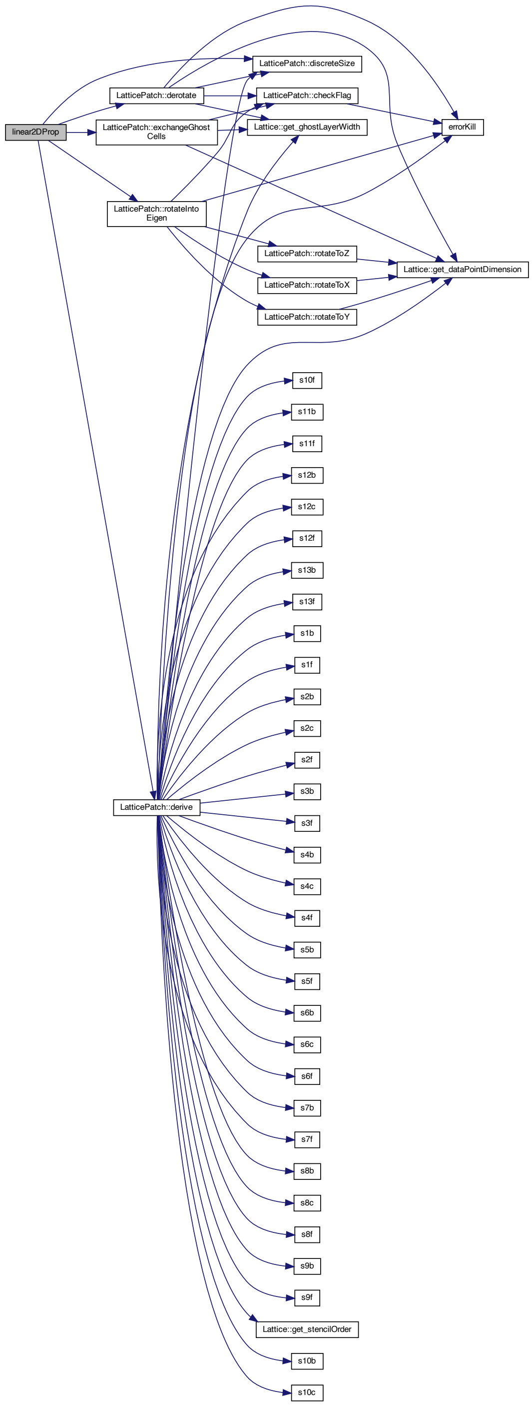
Maxwell propagation function for 2D – only for reference.

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

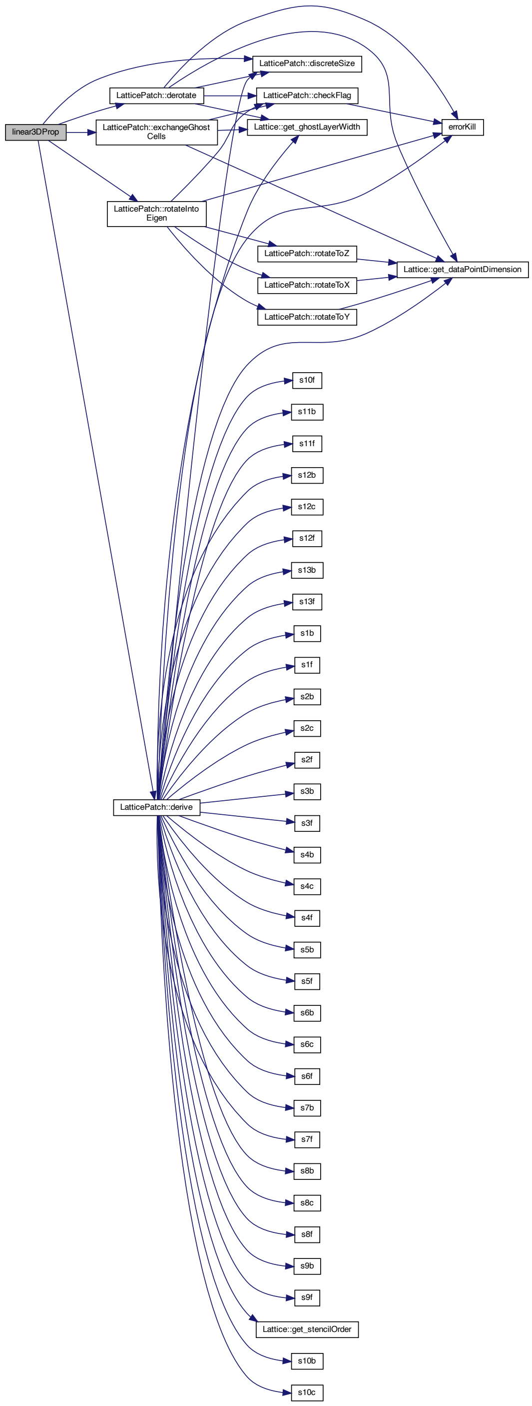
Maxwell propagation function for 3D – only for reference.

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;
```

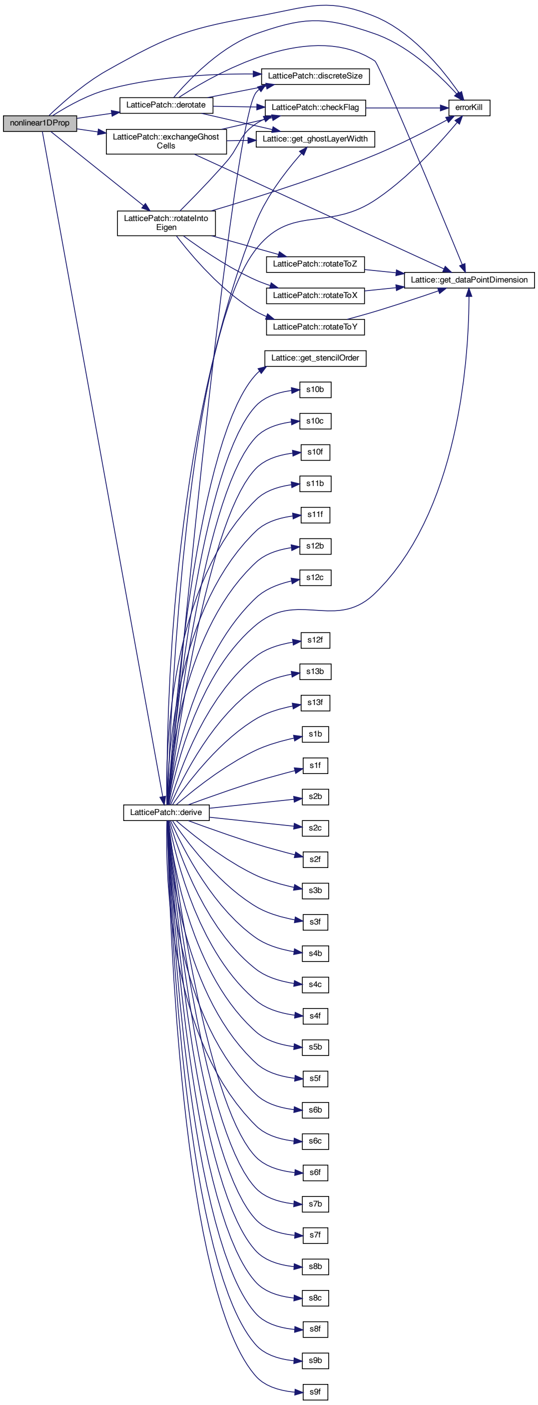
```
      break;
    case 3:
      lf = (0.000206527095658582755255648 + 0.000354046449700427580438254 * f) *
               f +
           0.000191775160254398272737387 * g * g;
      lff = 0.000206527095658582755255648 + 0.0007080928994008551608765075 * f;
      lfg = 0.0003835503205087965454747749 * g;
      lg = (0.0003614224174025198216973841 +
            0.0003835503205087965454747749 * f) *
           g;
      lgg = 0.0003614224174025198216973841 + 0.0003835503205087965454747749 * f;
      break;
    default:
      errorKill(
          "You need to specify a correct order in the weak-field expansion.");
    }
    // 3. Assign Jacobi components
    JMM[0] = lf + lff * Quad[0] +
             udata[3 + pp] * (2 * lfg * udata[pp] + lgg * udata[3 + pp]);
    JMM[6] =
        lff * udata[pp] * udata[1 + pp] + lfg * udata[1 + pp] * udata[3 + pp] +
        lfg * udata[pp] * udata[4 + pp] + lgg * udata[3 + pp] * udata[4 + pp];
    JMM[7] = lf + lff * Quad[1] +
             udata[4 + pp] * (2 * lfg * udata[1 + pp] + lgg * udata[4 + pp]);
    JMM[12] =
        lff * udata[pp] * udata[2 + pp] + lfg * udata[2 + pp] * udata[3 + pp] +
        lfg * udata[pp] * udata[5 + pp] + lgg * udata[3 + pp] * udata[5 + pp];
    JMM[13] = lff * udata[1 + pp] * udata[2 + pp] +
              lfg * udata[2 + pp] * udata[4 + pp] +
              lfg * udata[1 + pp] * udata[5 + pp] +
              lgg * udata[4 + pp] * udata[5 + pp];
    JMM[14] = lf + lff * Quad[2] +
              udata[5 + pp] * (2 * lfg * udata[2 + pp] + lgg * udata[5 + pp]);
    JMM[18] = lg + lfg * (Quad[0] - Quad[3 + 0]) +
              (-lff + lgg) * udata[pp] * udata[3 + pp];
    JMM[19] = -(udata[3 + pp] * (lff * udata[1 + pp] + lfg * udata[4 + pp])) +
              udata[pp] * (lfg * udata[1 + pp] + lgg * udata[4 + pp]);
    JMM[20] = -(udata[3 + pp] * (lff * udata[2 + pp] + lfg * udata[5 + pp])) +
              udata[pp] * (lfg * udata[2 + pp] + lgg * udata[5 + pp]);
    JMM[21] = -lf + lgg * Quad[0] +
              udata[3 + pp] * (-2 * lfg * udata[pp] + lff * udata[3 + pp]);
    JMM[24] = udata[1 + pp] * (lfg * udata[pp] + lgg * udata[3 + pp]) -
              (lff * udata[pp] + lfg * udata[3 + pp]) * udata[4 + pp];
    JMM[25] = lg + lfg * (Quad[1] - Quad[4 + 0]) +
              (-lff + lgg) * udata[1 + pp] * udata[4 + pp];
    JMM[26] = -(udata[4 + pp] * (lff * udata[2 + pp] + lfg * udata[5 + pp])) +
              udata[1 + pp] * (lfg * udata[2 + pp] + lgg * udata[5 + pp]);
    JMM[27] = lgg * udata[pp] * udata[1 + pp] +
              lff * udata[3 + pp] * udata[4 + pp] -
              lfg * (udata[1 + pp] * udata[3 + pp] + udata[pp] * udata[4 + pp]);
    JMM[28] = -lf + lgg * Quad[1] +
              udata[4 + pp] * (-2 * lfg * udata[1 + pp] + lff * udata[4 + pp]);
    JMM[30] = udata[2 + pp] * (lfg * udata[pp] + lgg * udata[3 + pp]) -
              (lff * udata[pp] + lfg * udata[3 + pp]) * udata[5 + pp];
    JMM[31] = udata[2 + pp] * (lfg * udata[1 + pp] + lgg * udata[4 + pp]) -
              (lff * udata[1 + pp] + lfg * udata[4 + pp]) * udata[5 + pp];
    JMM[32] = lg + lfg * (Quad[2] - Quad[5 + 0]) +
              (-lff + lgg) * udata[2 + pp] * udata[5 + pp];
    JMM[33] = lgg * udata[pp] * udata[2 + pp] +
              lff * udata[3 + pp] * udata[5 + pp] -
              lfg * (udata[2 + pp] * udata[3 + pp] + udata[pp] * udata[5 + pp]);
    JMM[34] =
        lgg * udata[1 + pp] * udata[2 + pp] +
        lff * udata[4 + pp] * udata[5 + pp] -
        lfg * (udata[2 + pp] * udata[4 + pp] + udata[1 + pp] * udata[5 + pp]);
    JMM[35] = -lf + lgg * Quad[2] +
              udata[5 + pp] * (-2 * lfg * udata[2 + pp] + lff * udata[5 + pp]);
    //#pragma unroll_and_jam
    //#pragma distribute_point
    for (int i = 0; i < 6; i++) {
      for (int j = i + 1; j < 6; j++) {
        JMM[i * 6 + j] = JMM[j * 6 + i];
      }
    }
    // 4. Final values for temporal derivatives of field values
    h[0] = 0;
    h[1] = dxData[pp] * JMM[30] + dxData[1 + pp] * JMM[31] +
           dxData[2 + pp] * JMM[32] + dxData[3 + pp] * JMM[33] +
           dxData[4 + pp] * JMM[34] + dxData[5 + pp] * (-1 + JMM[35]);
    h[2] = -(dxData[pp] * JMM[24]) - dxData[1 + pp] * JMM[25] -
           dxData[2 + pp] * JMM[26] - dxData[3 + pp] * JMM[27] +
           dxData[4 + pp] * (1 - JMM[28]) - dxData[5 + pp] * JMM[29];
    h[3] = 0;
    h[4] = dxData[2 + pp];
    h[5] = -dxData[1 + pp];
    h[0] -= h[3] * JMM[3] + h[4] * JMM[4] + h[5] * JMM[5];
    h[1] -= h[3] * JMM[9] + h[4] * JMM[10] + h[5] * JMM[11];
```

```
    h[2] -= h[3] * JMM[15] + h[4] * JMM[16] + h[5] * JMM[17];
    // (1+Z)^-1 applies only to E components
    dudata[pp + 0] =
        h[2] * (-(JMM[2] * (1 + JMM[7])) + JMM[1] * JMM[8]) +
        h[1] * (JMM[2] * JMM[13] - JMM[1] * (1 + JMM[14])) +
        h[0] * (1 - JMM[8] * JMM[13] + JMM[14] + JMM[7] * (1 + JMM[14]));
    dudata[pp + 1] =
        h[2] * (JMM[2] * JMM[6] - (1 + JMM[0]) * JMM[8]) +
        h[1] * (1 - JMM[2] * JMM[12] + JMM[14] + JMM[0] * (1 + JMM[14])) +
        h[0] * (JMM[8] * JMM[12] - JMM[6] * (1 + JMM[14]));
    dudata[pp + 2] =
        h[2] * (1 - JMM[1] * JMM[6] + JMM[7] + JMM[0] * (1 + JMM[7])) +
        h[1] * (JMM[1] * JMM[12] - (1 + JMM[0]) * JMM[13]) +
        h[0] * (-((1 + JMM[7]) * JMM[12]) + JMM[6] * JMM[13]);
    pseudoDenom =
        -((1 + JMM[7]) * (-1 + JMM[2] * JMM[12])) +
        (JMM[2] * JMM[6] - JMM[8]) * JMM[13] + JMM[14] + JMM[7] * JMM[14] +
        JMM[0] * (1 + JMM[7] - JMM[8] * JMM[13] + (1 + JMM[7]) * JMM[14]) -
        JMM[1] * (-(JMM[8] * JMM[12]) + JMM[6] * (1 + JMM[14]));
    dudata[pp + 0] /= pseudoDenom;
    dudata[pp + 1] /= pseudoDenom;
    dudata[pp + 2] /= pseudoDenom;
    dudata[pp + 3] = h[3];
    dudata[pp + 4] = h[4];
    dudata[pp + 5] = h[5];
  }
  return;
}
```

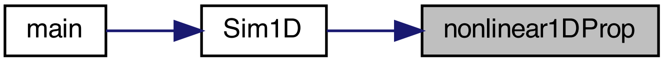
References LatticePatch::buffData, LatticePatch::derive(), LatticePatch::derotate(), LatticePatch::discreteSize(), errorKill(), LatticePatch::exchangeGhostCells(), and LatticePatch::rotateIntoEigen().

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 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;
```

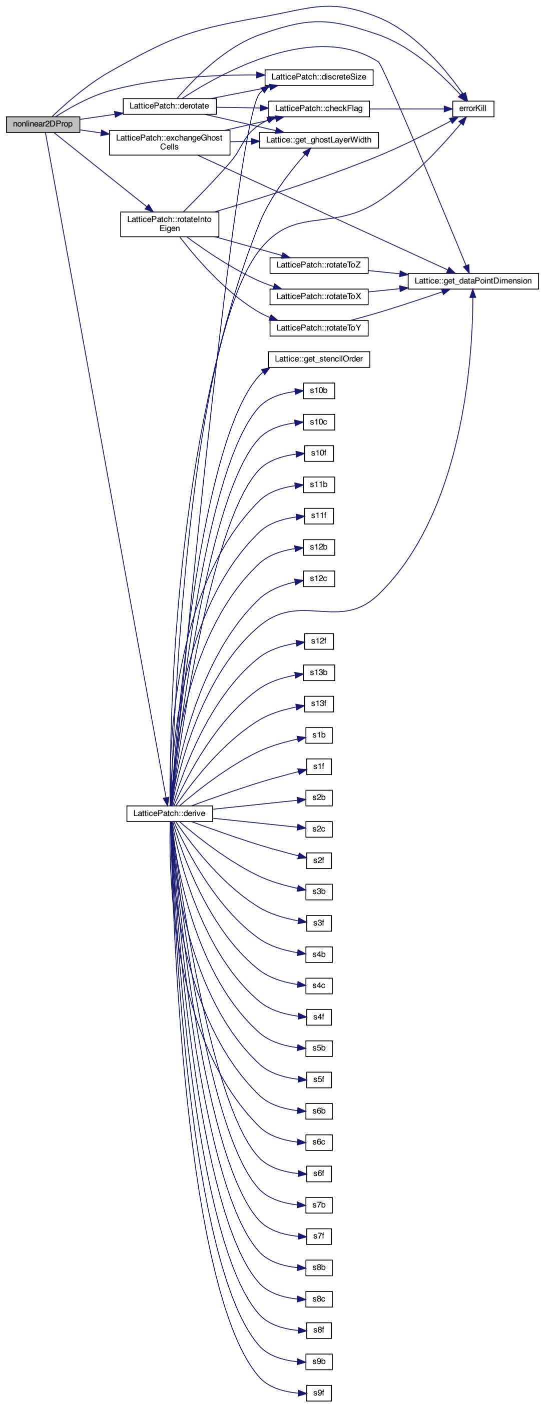
```
      lgg = 0;
      break;
    case 2:
      lf = 0.000354046449700427580438254 * f * f +
           0.000191775160254398272737387 * g * g;
      lff = 0.000708092899400855160876507 * f;
      lfg = 0.000383550320508796545474775 * g;
      lg = 0.000383550320508796545474775 * f * g;
      lgg = 0.000383550320508796545474775 * f;
      break;
    case 1:
      lf = 0.000206527095658582755255648 * f;
      lff = 0.000206527095658582755255648;
      lfg = 0;
      lg = 0.0003614224174025198216973841 * g;
      lgg = 0.0003614224174025198216973841;
      break;
    case 3:
      lf = (0.000206527095658582755255648 + 0.000354046449700427580438254 * f) *
               f +
           0.000191775160254398272737387 * g * g;
      lff = 0.000206527095658582755255648 + 0.000708092899400855160876508 * f;
      lfg = 0.000383550320508796545474775 * g;
      lg = (0.000361422417402519821697384 + 0.000383550320508796545474775 * f) *
           g;
      lgg = 0.000361422417402519821697384 + 0.000383550320508796545474775 * f;
      break;
    default:
      errorKill(
          "You need to specify a correct order in the weak-field expansion.");
    }
    // 3
    JMM[0] = lf + lff * Quad[0] +
             udata[3 + pp] * (2 * lfg * udata[pp] + lgg * udata[3 + pp]);
    JMM[6] =
        lff * udata[pp] * udata[1 + pp] + lfg * udata[1 + pp] * udata[3 + pp] +
        lfg * udata[pp] * udata[4 + pp] + lgg * udata[3 + pp] * udata[4 + pp];
    JMM[7] = lf + lff * Quad[1] +
             udata[4 + pp] * (2 * lfg * udata[1 + pp] + lgg * udata[4 + pp]);
    JMM[12] =
        lff * udata[pp] * udata[2 + pp] + lfg * udata[2 + pp] * udata[3 + pp] +
        lfg * udata[pp] * udata[5 + pp] + lgg * udata[3 + pp] * udata[5 + pp];
    JMM[13] = lff * udata[1 + pp] * udata[2 + pp] +
              lfg * udata[2 + pp] * udata[4 + pp] +
              lfg * udata[1 + pp] * udata[5 + pp] +
              lgg * udata[4 + pp] * udata[5 + pp];
    JMM[14] = lf + lff * Quad[2] +
              udata[5 + pp] * (2 * lfg * udata[2 + pp] + lgg * udata[5 + pp]);
    JMM[18] = lg + lfg * (Quad[0] - Quad[3 + 0]) +
              (-lff + lgg) * udata[pp] * udata[3 + pp];
    JMM[19] = -(udata[3 + pp] * (lff * udata[1 + pp] + lfg * udata[4 + pp])) +
              udata[pp] * (lfg * udata[1 + pp] + lgg * udata[4 + pp]);
    JMM[20] = -(udata[3 + pp] * (lff * udata[2 + pp] + lfg * udata[5 + pp])) +
              udata[pp] * (lfg * udata[2 + pp] + lgg * udata[5 + pp]);
    JMM[21] = -lf + lgg * Quad[0] +
              udata[3 + pp] * (-2 * lfg * udata[pp] + lff * udata[3 + pp]);
    JMM[24] = udata[1 + pp] * (lfg * udata[pp] + lgg * udata[3 + pp]) -
              (lff * udata[pp] + lfg * udata[3 + pp]) * udata[4 + pp];
    JMM[25] = lg + lfg * (Quad[1] - Quad[4 + 0]) +
              (-lff + lgg) * udata[1 + pp] * udata[4 + pp];
    JMM[26] = -(udata[4 + pp] * (lff * udata[2 + pp] + lfg * udata[5 + pp])) +
              udata[1 + pp] * (lfg * udata[2 + pp] + lgg * udata[5 + pp]);
    JMM[27] = lgg * udata[pp] * udata[1 + pp] +
              lff * udata[3 + pp] * udata[4 + pp] -
              lfg * (udata[1 + pp] * udata[3 + pp] + udata[pp] * udata[4 + pp]);
    JMM[28] = -lf + lgg * Quad[1] +
              udata[4 + pp] * (-2 * lfg * udata[1 + pp] + lff * udata[4 + pp]);
    JMM[30] = udata[2 + pp] * (lfg * udata[pp] + lgg * udata[3 + pp]) -
              (lff * udata[pp] + lfg * udata[3 + pp]) * udata[5 + pp];
    JMM[31] = udata[2 + pp] * (lfg * udata[1 + pp] + lgg * udata[4 + pp]) -
              (lff * udata[1 + pp] + lfg * udata[4 + pp]) * udata[5 + pp];
    JMM[32] = lg + lfg * (Quad[2] - Quad[5 + 0]) +
              (-lff + lgg) * udata[2 + pp] * udata[5 + pp];
    JMM[33] = lgg * udata[pp] * udata[2 + pp] +
              lff * udata[3 + pp] * udata[5 + pp] -
              lfg * (udata[2 + pp] * udata[3 + pp] + udata[pp] * udata[5 + pp]);
    JMM[34] =
        lgg * udata[1 + pp] * udata[2 + pp] +
        lff * udata[4 + pp] * udata[5 + pp] -
        lfg * (udata[2 + pp] * udata[4 + pp] + udata[1 + pp] * udata[5 + pp]);
    JMM[35] = -lf + lgg * Quad[2] +
              udata[5 + pp] * (-2 * lfg * udata[2 + pp] + lff * udata[5 + pp]);
    // 4
    //#pragma distribute_point
    //#pragma unroll_and_jam
    for (int i = 0; i < 6; i++) {
      for (int j = i + 1; j < 6; j++) {
```

```
          JMM[i * 6 + j] = JMM[j * 6 + i];
        }
      }
      h[0] = 0;
      h[1] = dxData[pp] * JMM[30] + dxData[1 + pp] * JMM[31] +
             dxData[2 + pp] * JMM[32] + dxData[3 + pp] * JMM[33] +
             dxData[4 + pp] * JMM[34] + dxData[5 + pp] * (-1 + JMM[35]);
      h[2] = -(dxData[pp] * JMM[24]) - dxData[1 + pp] * JMM[25] -
             dxData[2 + pp] * JMM[26] - dxData[3 + pp] * JMM[27] +
             dxData[4 + pp] * (1 - JMM[28]) - dxData[5 + pp] * JMM[29];
      h[3] = 0;
      h[4] = dxData[2 + pp];
      h[5] = -dxData[1 + pp];
      h[0] += -(dyData[pp] * JMM[30]) - dyData[1 + pp] * JMM[31] -
              dyData[2 + pp] * JMM[32] - dyData[3 + pp] * JMM[33] -
              dyData[4 + pp] * JMM[34] + dyData[5 + pp] * (1 - JMM[35]);
      h[1] += 0;
      h[2] += dyData[pp] * JMM[18] + dyData[1 + pp] * JMM[19] +
              dyData[2 + pp] * JMM[20] + dyData[3 + pp] * (-1 + JMM[21]) +
              dyData[4 + pp] * JMM[22] + dyData[5 + pp] * JMM[23];
      h[3] += -dyData[2 + pp];
      h[4] += 0;
      h[5] += dyData[pp];
      h[0] -= h[3] * JMM[3] + h[4] * JMM[4] + h[5] * JMM[5];
      h[1] -= h[3] * JMM[9] + h[4] * JMM[10] + h[5] * JMM[11];
      h[2] -= h[3] * JMM[15] + h[4] * JMM[16] + h[5] * JMM[17];
      dudata[pp + 0] =
          h[2] * (-(JMM[2] * (1 + JMM[7])) + JMM[1] * JMM[8]) +
          h[1] * (JMM[2] * JMM[13] - JMM[1] * (1 + JMM[14])) +
          h[0] * (1 - JMM[8] * JMM[13] + JMM[14] + JMM[7] * (1 + JMM[14]));
      dudata[pp + 1] =
          h[2] * (JMM[2] * JMM[6] - (1 + JMM[0]) * JMM[8]) +
          h[1] * (1 - JMM[2] * JMM[12] + JMM[14] + JMM[0] * (1 + JMM[14])) +
          h[0] * (JMM[8] * JMM[12] - JMM[6] * (1 + JMM[14]));
      dudata[pp + 2] =
          h[2] * (1 - JMM[1] * JMM[6] + JMM[7] + JMM[0] * (1 + JMM[7])) +
          h[1] * (JMM[1] * JMM[12] - (1 + JMM[0]) * JMM[13]) +
          h[0] * (-((1 + JMM[7]) * JMM[12]) + JMM[6] * JMM[13]);
      pseudoDenom =
          -((1 + JMM[7]) * (-1 + JMM[2] * JMM[12])) +
          (JMM[2] * JMM[6] - JMM[8]) * JMM[13] + JMM[14] + JMM[7] * JMM[14] +
          JMM[0] * (1 + JMM[7] - JMM[8] * JMM[13] + (1 + JMM[7]) * JMM[14]) -
          JMM[1] * (-(JMM[8] * JMM[12]) + JMM[6] * (1 + JMM[14]));
      dudata[pp + 0] /= pseudoDenom;
      dudata[pp + 1] /= pseudoDenom;
      dudata[pp + 2] /= pseudoDenom;
      dudata[pp + 3] = h[3];
      dudata[pp + 4] = h[4];
      dudata[pp + 5] = h[5];
    }
    return;
}
```

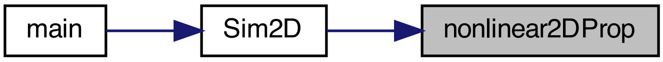
References LatticePatch::buffData, LatticePatch::derive(), LatticePatch::derotate(), LatticePatch::discreteSize(), errorKill(), LatticePatch::exchangeGhostCells(), and LatticePatch::rotateIntoEigen().

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 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);
```

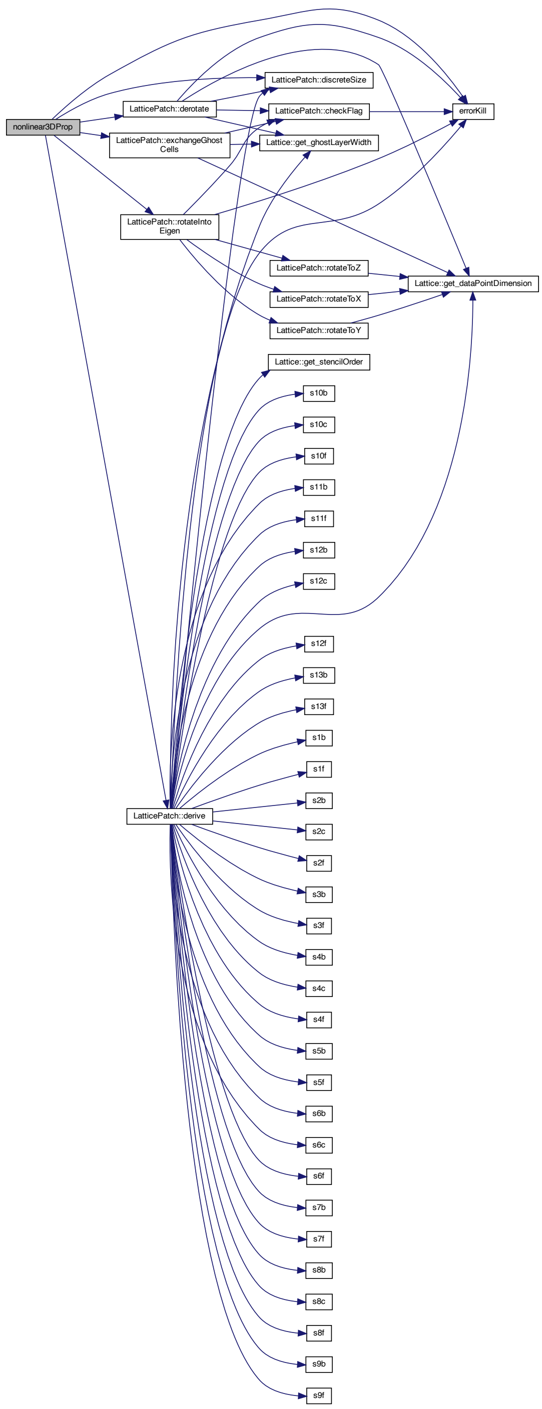
```
int totalNP = data->discreteSize();
//#pragma distribute_point
//#pragma unroll_and_jam
for (int pp = 0; pp < totalNP * 6; pp += 6) {
  // 1
  f = 0.5 * ((Quad[0] = udata[pp] * udata[pp]) +
             (Quad[1] = udata[pp + 1] * udata[pp + 1]) +
             (Quad[2] = udata[pp + 2] * udata[pp + 2]) -
             (Quad[3] = udata[pp + 3] * udata[pp + 3]) -
             (Quad[4] = udata[pp + 4] * udata[pp + 4]) -
             (Quad[5] = udata[pp + 5] * udata[pp + 5]));
  g = udata[pp] * udata[pp + 3] + udata[pp + 1] * udata[pp + 4] +
      udata[pp + 2] * udata[pp + 5];
  // 2
  switch (*c) {
  case 0:
    lf = 0;
    lff = 0;
    lfg = 0;
    lg = 0;
    lgg = 0;
    break;
  case 2:
    lf = 0.000354046449700427580438254 * f * f +
         0.000191775160254398272737387 * g * g;
    lff = 0.000708092899400855160876 5075 * f;
    lfg = 0.000383550320508796545474 7749 * g;
    lg = 0.000383550320508796545474 7749 * f * g;
    lgg = 0.000383550320508796545474 7749 * f;
    break;
  case 1:
    lf = 0.000206527095658582755255648 * f;
    lff = 0.000206527095658582755255648;
    lfg = 0;
    lg = 0.000361422417402519821697 3841 * g;
    lgg = 0.000361422417402519821697 3841;
    break;
  case 3:
    lf = (0.000206527095658582755255648 + 0.000354046449700427580438254 * f) *
             f +
         0.000191775160254398272737387 * g * g;
    lff = 0.000206527095658582755255648 + 0.00070809289940085516087 6508 * f;
    lfg = 0.000383550320508796545474 7749 * g;
    lg = (0.00036142241740251982169 7384 + 0.00038355032050879654547 4775 * f) *
         g;
    lgg = 0.00036142241740251982169 7384 + 0.00038355032050879654547 4775 * f;
    break;
  default:
    errorKill(
        "You need to specify a correct order in the weak-field expansion.");
  }
  // 3
  JMM[0] = lf + lff * Quad[0] +
           udata[3 + pp] * (2 * lfg * udata[pp] + lgg * udata[3 + pp]);
  JMM[6] =
      lff * udata[pp] * udata[1 + pp] + lfg * udata[1 + pp] * udata[3 + pp] +
      lfg * udata[pp] * udata[4 + pp] + lgg * udata[3 + pp] * udata[4 + pp];
  JMM[7] = lf + lff * Quad[1] +
           udata[4 + pp] * (2 * lfg * udata[1 + pp] + lgg * udata[4 + pp]);
  JMM[12] =
      lff * udata[pp] * udata[2 + pp] + lfg * udata[2 + pp] * udata[3 + pp] +
      lfg * udata[pp] * udata[5 + pp] + lgg * udata[3 + pp] * udata[5 + pp];
  JMM[13] = lff * udata[1 + pp] * udata[2 + pp] +
            lfg * udata[2 + pp] * udata[4 + pp] +
            lfg * udata[1 + pp] * udata[5 + pp] +
            lgg * udata[4 + pp] * udata[5 + pp];
  JMM[14] = lf + lff * Quad[2] +
            udata[5 + pp] * (2 * lfg * udata[2 + pp] + lgg * udata[5 + pp]);
  JMM[18] = lg + lfg * (Quad[0] - Quad[3 + 0]) +
            (-lff + lgg) * udata[pp] * udata[3 + pp];
  JMM[19] = -(udata[3 + pp] * (lff * udata[1 + pp] + lfg * udata[4 + pp])) +
            udata[pp] * (lfg * udata[1 + pp] + lgg * udata[4 + pp]);
  JMM[20] = -(udata[3 + pp] * (lff * udata[2 + pp] + lfg * udata[5 + pp])) +
            udata[pp] * (lfg * udata[2 + pp] + lgg * udata[5 + pp]);
  JMM[21] = -lf + lgg * Quad[0] +
            udata[3 + pp] * (-2 * lfg * udata[pp] + lff * udata[3 + pp]);
  JMM[24] = udata[1 + pp] * (lfg * udata[pp] + lgg * udata[3 + pp]) -
            (lff * udata[pp] + lfg * udata[3 + pp]) * udata[4 + pp];
  JMM[25] = lg + lfg * (Quad[1] - Quad[4 + 0]) +
            (-lff + lgg) * udata[1 + pp] * udata[4 + pp];
  JMM[26] = -(udata[4 + pp] * (lff * udata[2 + pp] + lfg * udata[5 + pp])) +
            udata[1 + pp] * (lfg * udata[2 + pp] + lgg * udata[5 + pp]);
  JMM[27] = lgg * udata[pp] * udata[1 + pp] +
            lff * udata[3 + pp] * udata[4 + pp] -
            lfg * (udata[1 + pp] * udata[3 + pp] + udata[pp] * udata[4 + pp]);
  JMM[28] = -lf + lgg * Quad[1] +
            udata[4 + pp] * (-2 * lfg * udata[1 + pp] + lff * udata[4 + pp]);
```

```
    JMM[30] = udata[2 + pp] * (lfg * udata[pp] + lgg * udata[3 + pp]) -
              (lff * udata[pp] + lfg * udata[3 + pp]) * udata[5 + pp];
    JMM[31] = udata[2 + pp] * (lfg * udata[1 + pp] + lgg * udata[4 + pp]) -
              (lff * udata[1 + pp] + lfg * udata[4 + pp]) * udata[5 + pp];
    JMM[32] = lg + lfg * (Quad[2] - Quad[5 + 0]) +
              (-lff + lgg) * udata[2 + pp] * udata[5 + pp];
    JMM[33] = lgg * udata[pp] * udata[2 + pp] +
              lff * udata[3 + pp] * udata[5 + pp] -
              lfg * (udata[2 + pp] * udata[3 + pp] + udata[pp] * udata[5 + pp]);
    JMM[34] =
        lgg * udata[1 + pp] * udata[2 + pp] +
        lff * udata[4 + pp] * udata[5 + pp] -
        lfg * (udata[2 + pp] * udata[4 + pp] + udata[1 + pp] * udata[5 + pp]);
    JMM[35] = -lf + lgg * Quad[2] +
              udata[5 + pp] * (-2 * lfg * udata[2 + pp] + lff * udata[5 + pp]);
    // 4
    //#pragma distribute_point
    //#pragma unroll_and_jam
    for (int i = 0; i < 6; i++) {
      for (int j = i + 1; j < 6; j++) {
        JMM[i * 6 + j] = JMM[j * 6 + i];
      }
    }
    h[0] = 0;
    h[1] = dxData[pp] * JMM[30] + dxData[1 + pp] * JMM[31] +
           dxData[2 + pp] * JMM[32] + dxData[3 + pp] * JMM[33] +
           dxData[4 + pp] * JMM[34] + dxData[5 + pp] * (-1 + JMM[35]);
    h[2] = -(dxData[pp] * JMM[24]) - dxData[1 + pp] * JMM[25] -
           dxData[2 + pp] * JMM[26] - dxData[3 + pp] * JMM[27] +
           dxData[4 + pp] * (1 - JMM[28]) - dxData[5 + pp] * JMM[29];
    h[3] = 0;
    h[4] = dxData[2 + pp];
    h[5] = -dxData[1 + pp];
    h[0] += -(dyData[pp] * JMM[30]) - dyData[1 + pp] * JMM[31] -
            dyData[2 + pp] * JMM[32] - dyData[3 + pp] * JMM[33] -
            dyData[4 + pp] * JMM[34] - dyData[5 + pp] * (1 - JMM[35]);
    h[1] += 0;
    h[2] += dyData[pp] * JMM[18] + dyData[1 + pp] * JMM[19] +
            dyData[2 + pp] * JMM[20] + dyData[3 + pp] * (-1 + JMM[21]) +
            dyData[4 + pp] * JMM[22] + dyData[5 + pp] * JMM[23];
    h[3] += -dyData[2 + pp];
    h[4] += 0;
    h[5] += dyData[pp];
    h[0] += dzData[pp] * JMM[24] + dzData[1 + pp] * JMM[25] +
            dzData[2 + pp] * JMM[26] + dzData[3 + pp] * JMM[27] +
            dzData[4 + pp] * (-1 + JMM[28]) + dzData[5 + pp] * JMM[29];
    h[1] += -(dzData[pp] * JMM[18]) - dzData[1 + pp] * JMM[19] -
            dzData[2 + pp] * JMM[20] + dzData[3 + pp] * (1 - JMM[21]) -
            dzData[4 + pp] * JMM[22] - dzData[5 + pp] * JMM[23];
    h[2] += 0;
    h[3] += dzData[1 + pp];
    h[4] += -dzData[pp];
    h[5] += 0;
    h[0] -= h[3] * JMM[3] + h[4] * JMM[4] + h[5] * JMM[5];
    h[1] -= h[3] * JMM[9] + h[4] * JMM[10] + h[5] * JMM[11];
    h[2] -= h[3] * JMM[15] + h[4] * JMM[16] + h[5] * JMM[17];
    dudata[pp + 0] =
        h[2] * (-(JMM[2] * (1 + JMM[7])) + JMM[1] * JMM[8]) +
        h[1] * (JMM[2] * JMM[13] - JMM[1] * (1 + JMM[14])) +
        h[0] * (1 - JMM[8] * JMM[13] + JMM[14] + JMM[7] * (1 + JMM[14]));
    dudata[pp + 1] =
        h[2] * (JMM[2] * JMM[6] - (1 + JMM[0]) * JMM[8]) +
        h[1] * (1 - JMM[2] * JMM[12] + JMM[14] + JMM[0] * (1 + JMM[14])) +
        h[0] * (JMM[8] * JMM[12] - JMM[6] * (1 + JMM[14]));
    dudata[pp + 2] =
        h[2] * (1 - JMM[1] * JMM[6] + JMM[7] + JMM[0] * (1 + JMM[7])) +
        h[1] * (JMM[1] * JMM[12] - (1 + JMM[0]) * JMM[13]) +
        h[0] * (-((1 + JMM[7]) * JMM[12]) + JMM[6] * JMM[13]);
    pseudoDenom =
        -((1 + JMM[7]) * (-1 + JMM[2] * JMM[12])) +
        (JMM[2] * JMM[6] - JMM[8]) * JMM[13] + JMM[14] + JMM[7] * JMM[14] +
        JMM[0] * (1 + JMM[7] - JMM[8] * JMM[13] + (1 + JMM[7]) * JMM[14]) -
        JMM[1] * (-(JMM[8] * JMM[12]) + JMM[6] * (1 + JMM[14]));
    dudata[pp + 0] /= pseudoDenom;
    dudata[pp + 1] /= pseudoDenom;
    dudata[pp + 2] /= pseudoDenom;
    dudata[pp + 3] = h[3];
    dudata[pp + 4] = h[4];
    dudata[pp + 5] = h[5];
  }
  return;
}
```

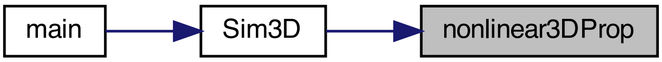
References LatticePatch::buffData, LatticePatch::derive(), LatticePatch::derotate(), LatticePatch::discreteSize(), errorKill(), LatticePatch::exchangeGhostCells(), and LatticePatch::rotateIntoEigen().

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.

```
////////////////////////////////////////////////////////////
/// @file TimeEvolutionFunctions.cpp
/// @brief 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>

/// CVode right-hand-side function (CVRhsFn)
int TimeEvolution::f(sunrealtype t, N_Vector u, N_Vector udot, void *data_loc) {
  // Set recover pointer to provided lattice patch where the data resides
  LatticePatch *data = nullptr;
  data = static_cast<LatticePatch *>(data_loc);

  // pointers for update circle
  sunrealtype *udata = nullptr, *dudata = nullptr;
  sunrealtype *originaluData = nullptr, *originalduData = nullptr;

  // Access NVECTOR_PARALLEL argument data with pointers
  udata = NV_DATA_P(u);
  dudata = NV_DATA_P(udot);

  // Store original data location of the patch
  originaluData = data->uData;
  originalduData = data->duData;
  // Point patch data to arguments of f
  data->uData = udata;
  data->duData = dudata;

  // Time-evolve these arguments (the field data) with specific propagator below
  TimeEvolver(data, u, udot, c);

  // Refer patch data back to original location
  data->uData = originaluData;
  data->duData = originalduData;

  return (0);
}

/// only under-the-hood-callable Maxwell propagation in 1D
// unused parameters 2-4 for compliance with CVRhsFn
// same as the respective nonlinear function without nonlinear terms
void linear1DProp(LatticePatch *data, N_Vector u, N_Vector udot, int *c) {

  // pointers to temporal and spatial derivative data
  sunrealtype *duData = data->duData;
  sunrealtype *dxData = data->buffData[1 - 1];

  // sequence along any dimension:
  data->exchangeGhostCells(1); // exchange halos
  data->rotateIntoEigen(
      1);           // -> rotate all data to prepare derivative operation
  data->derive(1); // -> perform derivative on it
  data->derotate(
      1, dxData); // -> derotate derivative data to x-space for further use

  int totalNP = data->discreteSize();
  int pp = 0;
```

```
#pragma distribute_point
  for (int i = 0; i < totalNP; i++) {
    pp = i * 6;
    /*
     simple vacuum Maxwell equations for spatial deriative only in x-direction
     temporal derivative is approximated by spatial derivative according to the
     numerical scheme with Jacobi=0 -> no polarization or magnetization terms
    */
    duData[pp + 0] = 0;
    duData[pp + 1] = -dxData[pp + 5];
    duData[pp + 2] = dxData[pp + 4];
    duData[pp + 3] = 0;
    duData[pp + 4] = dxData[pp + 2];
    duData[pp + 5] = -dxData[pp + 1];
  }
}

/// nonlinear 1D HE propagation
void nonlinear1DProp(LatticePatch *data, N_Vector u, N_Vector udot, int *c) {

  // pointer to spatial derivative data sufficient, temporal derivative data
  // provided with udot
  sunrealtype *dxData = data->buffData[1 - 1];

  // same sequence as in the linear case
  data->exchangeGhostCells(1);
  data->rotateIntoEigen(1);
  data->derive(1);
  data->derotate(1, dxData);

  /*
  F and G are nonzero in the nonlinear case,
  polarization and magnetization contributions in Jacobi matrix style
  with derivatives of polarization and magnetization
  w.r.t. E- and B-field
  */
  sunrealtype f = NAN, g = NAN; // em field invariants F, G
  sunrealtype lf = NAN, lff = NAN, lfg = NAN, lg = NAN,
              lgg = NAN; // derivatives of Lagrangian w.r.t. field invariants
  array<sunrealtype, 36> JMM; // Jacobi matrix
  array<sunrealtype, 6> Quad; // array to hold E^2 and B^2 components
  array<sunrealtype, 6> h; // holding temporal derivatives of E and B components
                           // before operating (1+Z)^-1
  sunrealtype pseudoDenom = NAN; // needed for inversion of 1+Z
  sunrealtype *udata = nullptr,
              *dudata = nullptr; // pointers to data and temp. derivative data
  udata = NV_DATA_P(u);
  dudata = NV_DATA_P(udot);
  int totalNP = data->discreteSize(); // number of points in the patch
  // #pragma omp parallel for private(...) reduction(...) -> unsafe due to
  // reductions and many variables, how to deal with / reduction?
  //#pragma block_loop
  //#pragma unroll_and_jam
  //#pragma distribute_point
  for (int pp = 0; pp < totalNP * 6;
       pp += 6) { // loops through all 6dim points in the patch
                  //     for(int ppB=0;ppB<totalNP*6;ppB+=6*6){
                  //       for(int pp=ppB;pp<min(totalNP*6,ppB+6*6);pp+=6){
    /// Calculation of the Jacobi matrix
    // 1. Calculate F and G
    f = 0.5 * ((Quad[0] = udata[pp] * udata[pp]) +
               (Quad[1] = udata[pp + 1] * udata[pp + 1]) +
               (Quad[2] = udata[pp + 2] * udata[pp + 2]) -
               (Quad[3] = udata[pp + 3] * udata[pp + 3]) -
               (Quad[4] = udata[pp + 4] * udata[pp + 4]) -
               (Quad[5] = udata[pp + 5] * udata[pp + 5]));
    g = udata[pp] * udata[pp + 3] + udata[pp + 1] * udata[pp + 4] +
        udata[pp + 2] * udata[pp + 5];
    // 2. Choose process/expansion order and assign derivative values of L
    // w.r.t. F, G
    switch (*c) {
    case 0:
      lf = 0;
      lff = 0;
      lfg = 0;
      lg = 0;
      lgg = 0;
      break;
    case 2:
      lf = 0.000354046449700427580438254 * f * f +
           0.000191775160254398272737387 * g * g;
      lff = 0.000708092899400855160876507 5 * f;
      lfg = 0.000383550320508796545474774 9 * g;
      lg = 0.000383550320508796545474774 9 * f * g;
      lgg = 0.000383550320508796545474774 9 * f;
      break;
    case 1:
```

```
    lf = 0.000206527095658582755255648 * f;
    lff = 0.000206527095658582755255648;
    lfg = 0;
    lg = 0.0003614224174025198216973841 * g;
    lgg = 0.0003614224174025198216973841;
    break;
  case 3:
    lf = (0.000206527095658582755255648 + 0.000354046449700427580438254 * f) *
             f +
         0.000191775160254398272737387 * g * g;
    lff = 0.000206527095658582755255648 + 0.0007080928994008551608765075 * f;
    lfg = 0.0003835503205087965454747749 * g;
    lg = (0.0003614224174025198216973841 +
          0.0003835503205087965454747749 * f) *
         g;
    lgg = 0.0003614224174025198216973841 + 0.0003835503205087965454747749 * f;
    break;
  default:
    errorKill(
        "You need to specify a correct order in the weak-field expansion.");
  }
  // 3. Assign Jacobi components
  JMM[0] = lf + lff * Quad[0] +
           udata[3 + pp] * (2 * lfg * udata[pp] + lgg * udata[3 + pp]);
  JMM[6] =
      lff * udata[pp] * udata[1 + pp] + lfg * udata[1 + pp] * udata[3 + pp] +
      lfg * udata[pp] * udata[4 + pp] + lgg * udata[3 + pp] * udata[4 + pp];
  JMM[7] = lf + lff * Quad[1] +
           udata[4 + pp] * (2 * lfg * udata[1 + pp] + lgg * udata[4 + pp]);
  JMM[12] =
      lff * udata[pp] * udata[2 + pp] + lfg * udata[2 + pp] * udata[3 + pp] +
      lfg * udata[pp] * udata[5 + pp] + lgg * udata[3 + pp] * udata[5 + pp];
  JMM[13] = lff * udata[1 + pp] * udata[2 + pp] +
            lfg * udata[2 + pp] * udata[4 + pp] +
            lfg * udata[1 + pp] * udata[5 + pp] +
            lgg * udata[4 + pp] * udata[5 + pp];
  JMM[14] = lf + lff * Quad[2] +
            udata[5 + pp] * (2 * lfg * udata[2 + pp] + lgg * udata[5 + pp]);
  JMM[18] = lg + lfg * (Quad[0] - Quad[3 + 0]) +
            (-lff + lgg) * udata[pp] * udata[3 + pp];
  JMM[19] = -(udata[3 + pp] * (lff * udata[1 + pp] + lfg * udata[4 + pp])) +
            udata[pp] * (lfg * udata[1 + pp] + lgg * udata[4 + pp]);
  JMM[20] = -(udata[3 + pp] * (lff * udata[2 + pp] + lfg * udata[5 + pp])) +
            udata[pp] * (lfg * udata[2 + pp] + lgg * udata[5 + pp]);
  JMM[21] = -lf + lgg * Quad[0] +
            udata[3 + pp] * (-2 * lfg * udata[pp] + lff * udata[3 + pp]);
  JMM[24] = udata[1 + pp] * (lfg * udata[pp] + lgg * udata[3 + pp]) -
            (lff * udata[pp] + lfg * udata[3 + pp]) * udata[4 + pp];
  JMM[25] = lg + lfg * (Quad[1] - Quad[4 + 0]) +
            (-lff + lgg) * udata[1 + pp] * udata[4 + pp];
  JMM[26] = -(udata[4 + pp] * (lff * udata[2 + pp] + lfg * udata[5 + pp])) +
            udata[1 + pp] * (lfg * udata[2 + pp] + lgg * udata[5 + pp]);
  JMM[27] = lgg * udata[pp] * udata[1 + pp] +
            lff * udata[3 + pp] * udata[4 + pp] -
            lfg * (udata[1 + pp] * udata[3 + pp] + udata[pp] * udata[4 + pp]);
  JMM[28] = -lf + lgg * Quad[1] +
            udata[4 + pp] * (-2 * lfg * udata[1 + pp] + lff * udata[4 + pp]);
  JMM[30] = udata[2 + pp] * (lfg * udata[pp] + lgg * udata[3 + pp]) -
            (lff * udata[pp] + lfg * udata[3 + pp]) * udata[5 + pp];
  JMM[31] = udata[2 + pp] * (lfg * udata[1 + pp] + lgg * udata[4 + pp]) -
            (lff * udata[1 + pp] + lfg * udata[4 + pp]) * udata[5 + pp];
  JMM[32] = lg + lfg * (Quad[2] - Quad[5 + 0]) +
            (-lff + lgg) * udata[2 + pp] * udata[5 + pp];
  JMM[33] = lgg * udata[pp] * udata[2 + pp] +
            lff * udata[3 + pp] * udata[5 + pp] -
            lfg * (udata[2 + pp] * udata[3 + pp] + udata[pp] * udata[5 + pp]);
  JMM[34] =
      lgg * udata[1 + pp] * udata[2 + pp] +
      lff * udata[4 + pp] * udata[5 + pp] -
      lfg * (udata[2 + pp] * udata[4 + pp] + udata[1 + pp] * udata[5 + pp]);
  JMM[35] = -lf + lgg * Quad[2] +
            udata[5 + pp] * (-2 * lfg * udata[2 + pp] + lff * udata[5 + pp]);
  //#pragma unroll_and_jam
  //#pragma distribute_point
  for (int i = 0; i < 6; i++) {
    for (int j = i + 1; j < 6; j++) {
      JMM[i * 6 + j] = JMM[j * 6 + i];
    }
  }
  // 4. Final values for temporal derivatives of field values
  h[0] = 0;
  h[1] = dxData[pp] * JMM[30] + dxData[1 + pp] * JMM[31] +
         dxData[2 + pp] * JMM[32] + dxData[3 + pp] * JMM[33] +
         dxData[4 + pp] * JMM[34] + dxData[5 + pp] * (-1 + JMM[35]);
  h[2] = -(dxData[pp] * JMM[24]) - dxData[1 + pp] * JMM[25] -
         dxData[2 + pp] * JMM[26] - dxData[3 + pp] * JMM[27] +
         dxData[4 + pp] * (1 - JMM[28]) - dxData[5 + pp] * JMM[29];
```

```
    h[3] = 0;
    h[4] = dxData[2 + pp];
    h[5] = -dxData[1 + pp];
    h[0] -= h[3] * JMM[3] + h[4] * JMM[4] + h[5] * JMM[5];
    h[1] -= h[3] * JMM[9] + h[4] * JMM[10] + h[5] * JMM[11];
    h[2] -= h[3] * JMM[15] + h[4] * JMM[16] + h[5] * JMM[17];
    // (1+Z)^-1 applies only to E components
    dudata[pp + 0] =
        h[2] * (-(JMM[2] * (1 + JMM[7])) + JMM[1] * JMM[8]) +
        h[1] * (JMM[2] * JMM[13] - JMM[1] * (1 + JMM[14])) +
        h[0] * (1 - JMM[8] * JMM[13] + JMM[14] + JMM[7] * (1 + JMM[14]));
    dudata[pp + 1] =
        h[2] * (JMM[2] * JMM[6] - (1 + JMM[0]) * JMM[8]) +
        h[1] * (1 - JMM[2] * JMM[12] + JMM[14] + JMM[0] * (1 + JMM[14])) +
        h[0] * (JMM[8] * JMM[12] - JMM[6] * (1 + JMM[14]));
    dudata[pp + 2] =
        h[2] * (1 - JMM[1] * JMM[6] + JMM[7] + JMM[0] * (1 + JMM[7])) +
        h[1] * (JMM[1] * JMM[12] - (1 + JMM[0]) * JMM[13]) +
        h[0] * (-((1 + JMM[7]) * JMM[12]) + JMM[6] * JMM[13]);
    pseudoDenom =
        -((1 + JMM[7]) * (-1 + JMM[2] * JMM[12])) +
        (JMM[2] * JMM[6] - JMM[8]) * JMM[13] + JMM[14] + JMM[7] * JMM[14] +
        JMM[0] * (1 + JMM[7] - JMM[8] * JMM[13] + (1 + JMM[7]) * JMM[14]) -
        JMM[1] * (-(JMM[8] * JMM[12]) + JMM[6] * (1 + JMM[14]));
    dudata[pp + 0] /= pseudoDenom;
    dudata[pp + 1] /= pseudoDenom;
    dudata[pp + 2] /= pseudoDenom;
    dudata[pp + 3] = h[3];
    dudata[pp + 4] = h[4];
    dudata[pp + 5] = h[5];
  }
  return;
}

/// only under-the-hood-callable Maxwell propagation in 2D
// unused parameters 2-4 for compliance with CVRhsFn
// same as the respective nonlinear function without nonlinear terms
void linear2DProp(LatticePatch *data, N_Vector u, N_Vector udot, int *c) {

  sunrealtype *duData = data->duData;
  sunrealtype *dxData = data->buffData[1 - 1];
  sunrealtype *dyData = data->buffData[2 - 1];

  data->exchangeGhostCells(1);
  data->rotateIntoEigen(1);
  data->derive(1);
  data->derotate(1, dxData);
  data->exchangeGhostCells(2);
  data->rotateIntoEigen(2);
  data->derive(2);
  data->derotate(2, dyData);

  int totalNP = data->discreteSize();
  int pp = 0;
#pragma distribute_point
  for (int i = 0; i < totalNP; i++) {
    pp = i * 6;
    duData[pp + 0] = dyData[pp + 5];
    duData[pp + 1] = -dxData[pp + 5];
    duData[pp + 2] = -dyData[pp + 3] + dxData[pp + 4];
    duData[pp + 3] = -dyData[pp + 2];
    duData[pp + 4] = dxData[pp + 2];
    duData[pp + 5] = dyData[pp + 0] - dxData[pp + 1];
  }
}

/// nonlinear 2D HE propagation
void nonlinear2DProp(LatticePatch *data, N_Vector u, N_Vector udot, int *c) {

  sunrealtype *dxData = data->buffData[1 - 1];
  sunrealtype *dyData = data->buffData[2 - 1];

  data->exchangeGhostCells(1);
  data->rotateIntoEigen(1);
  data->derive(1);
  data->derotate(1, dxData);
  data->exchangeGhostCells(2);
  data->rotateIntoEigen(2);
  data->derive(2);
  data->derotate(2, dyData);

  sunrealtype f = NAN, g = NAN;
  sunrealtype lf = NAN, lff = NAN, lfg = NAN, lg = NAN, lgg = NAN;
  array<sunrealtype, 36> JMM;
  array<sunrealtype, 6> Quad;
  array<sunrealtype, 6> h;
  sunrealtype pseudoDenom = NAN;
```

```
sunrealtype *udata = nullptr, *dudata = nullptr;
udata = NV_DATA_P(u);
dudata = NV_DATA_P(udot);
int totalNP = data->discreteSize();
//#pragma distribute_point
//#pragma unroll_and_jam
for (int pp = 0; pp < totalNP * 6; pp += 6) {
  // 1
  f = 0.5 * ((Quad[0] = udata[pp] * udata[pp]) +
             (Quad[1] = udata[pp + 1] * udata[pp + 1]) +
             (Quad[2] = udata[pp + 2] * udata[pp + 2]) -
             (Quad[3] = udata[pp + 3] * udata[pp + 3]) -
             (Quad[4] = udata[pp + 4] * udata[pp + 4]) -
             (Quad[5] = udata[pp + 5] * udata[pp + 5]));
  g = udata[pp] * udata[pp + 3] + udata[pp + 1] * udata[pp + 4] +
      udata[pp + 2] * udata[pp + 5];
  // 2
  switch (*c) {
  case 0:
    lf = 0;
    lff = 0;
    lfg = 0;
    lg = 0;
    lgg = 0;
    break;
  case 2:
    lf = 0.000354046449700427580438254 * f * f +
         0.000191775160254398272737387 * g * g;
    lff = 0.0007080928994008551608765075 * f;
    lfg = 0.000383550320508796545474749 * g;
    lg = 0.000383550320508796545474749 * f * g;
    lgg = 0.000383550320508796545474749 * f;
    break;
  case 1:
    lf = 0.000206527095658582755255648 * f;
    lff = 0.000206527095658582755255648;
    lfg = 0;
    lg = 0.0003614224174025198216973841 * g;
    lgg = 0.0003614224174025198216973841;
    break;
  case 3:
    lf = (0.000206527095658582755255648 + 0.000354046449700427580438254 * f) *
             f +
         0.000191775160254398272737387 * g * g;
    lff = 0.000206527095658582755255648 + 0.000708092899400855160876508 * f;
    lfg = 0.000383550320508796545474749 * g;
    lg = (0.000361422417402519821697384 + 0.000383550320508796545474775 * f) *
         g;
    lgg = 0.000361422417402519821697384 + 0.000383550320508796545474775 * f;
    break;
  default:
    errorKill(
        "You need to specify a correct order in the weak-field expansion.");
  }
  // 3
  JMM[0] = lf + lff * Quad[0] +
           udata[3 + pp] * (2 * lfg * udata[pp] + lgg * udata[3 + pp]);
  JMM[6] =
      lff * udata[pp] * udata[1 + pp] + lfg * udata[1 + pp] * udata[3 + pp] +
      lfg * udata[pp] * udata[4 + pp] + lgg * udata[3 + pp] * udata[4 + pp];
  JMM[7] = lf + lff * Quad[1] +
           udata[4 + pp] * (2 * lfg * udata[1 + pp] + lgg * udata[4 + pp]);
  JMM[12] =
      lff * udata[pp] * udata[2 + pp] + lfg * udata[2 + pp] * udata[3 + pp] +
      lfg * udata[pp] * udata[5 + pp] + lgg * udata[3 + pp] * udata[5 + pp];
  JMM[13] = lff * udata[1 + pp] * udata[2 + pp] +
            lfg * udata[2 + pp] * udata[4 + pp] +
            lfg * udata[1 + pp] * udata[5 + pp] +
            lgg * udata[4 + pp] * udata[5 + pp];
  JMM[14] = lf + lff * Quad[2] +
            udata[5 + pp] * (2 * lfg * udata[2 + pp] + lgg * udata[5 + pp]);
  JMM[18] = lg + lfg * (Quad[0] - Quad[3 + 0]) +
            (-lff + lgg) * udata[pp] * udata[3 + pp];
  JMM[19] = -(udata[3 + pp] * (lff * udata[1 + pp] + lfg * udata[4 + pp])) +
            udata[pp] * (lfg * udata[1 + pp] + lgg * udata[4 + pp]);
  JMM[20] = -(udata[3 + pp] * (lff * udata[2 + pp] + lfg * udata[5 + pp])) +
            udata[pp] * (lfg * udata[2 + pp] + lgg * udata[5 + pp]);
  JMM[21] = -lf + lgg * Quad[0] +
            udata[3 + pp] * (-2 * lfg * udata[pp] + lff * udata[3 + pp]);
  JMM[24] = udata[1 + pp] * (lfg * udata[pp] + lgg * udata[3 + pp]) -
            (lff * udata[pp] + lfg * udata[3 + pp]) * udata[4 + pp];
  JMM[25] = lg + lfg * (Quad[1] - Quad[4 + 0]) +
            (-lff + lgg) * udata[1 + pp] * udata[4 + pp];
  JMM[26] = -(udata[4 + pp] * (lff * udata[2 + pp] + lfg * udata[5 + pp])) +
            udata[1 + pp] * (lfg * udata[2 + pp] + lgg * udata[5 + pp]);
  JMM[27] = lgg * udata[pp] * udata[1 + pp] +
            lff * udata[3 + pp] * udata[4 + pp] -
```

```
              lfg * (udata[1 + pp] * udata[3 + pp] + udata[pp] * udata[4 + pp]);
    JMM[28] = -lf + lgg * Quad[1] +
              udata[4 + pp] * (-2 * lfg * udata[1 + pp] + lff * udata[4 + pp]);
    JMM[30] = udata[2 + pp] * (lfg * udata[pp] + lgg * udata[3 + pp]) -
              (lff * udata[pp] + lfg * udata[3 + pp]) * udata[5 + pp];
    JMM[31] = udata[2 + pp] * (lfg * udata[1 + pp] + lgg * udata[4 + pp]) -
              (lff * udata[1 + pp] + lfg * udata[4 + pp]) * udata[5 + pp];
    JMM[32] = lg + lfg * (Quad[2] - Quad[5 + 0]) +
              (-lff + lgg) * udata[2 + pp] * udata[5 + pp];
    JMM[33] = lgg * udata[pp] * udata[2 + pp] +
              lff * udata[3 + pp] * udata[5 + pp] -
              lfg * (udata[2 + pp] * udata[3 + pp] + udata[pp] * udata[5 + pp]);
    JMM[34] =
        lgg * udata[1 + pp] * udata[2 + pp] +
        lff * udata[4 + pp] * udata[5 + pp] -
        lfg * (udata[2 + pp] * udata[4 + pp] + udata[1 + pp] * udata[5 + pp]);
    JMM[35] = -lf + lgg * Quad[2] +
              udata[5 + pp] * (-2 * lfg * udata[2 + pp] + lff * udata[5 + pp]);
    // 4
    //#pragma distribute_point
    //#pragma unroll_and_jam
    for (int i = 0; i < 6; i++) {
      for (int j = i + 1; j < 6; j++) {
        JMM[i * 6 + j] = JMM[j * 6 + i];
      }
    }
    h[0] = 0;
    h[1] = dxData[pp] * JMM[30] + dxData[1 + pp] * JMM[31] +
           dxData[2 + pp] * JMM[32] + dxData[3 + pp] * JMM[33] +
           dxData[4 + pp] * JMM[34] + dxData[5 + pp] * (-1 + JMM[35]);
    h[2] = -(dxData[pp] * JMM[24]) - dxData[1 + pp] * JMM[25] -
           dxData[2 + pp] * JMM[26] - dxData[3 + pp] * JMM[27] +
           dxData[4 + pp] * (1 - JMM[28]) - dxData[5 + pp] * JMM[29];
    h[3] = 0;
    h[4] = dxData[2 + pp];
    h[5] = -dxData[1 + pp];
    h[0] += -(dyData[pp] * JMM[30]) - dyData[1 + pp] * JMM[31] -
            dyData[2 + pp] * JMM[32] - dyData[3 + pp] * JMM[33] -
            dyData[4 + pp] * JMM[34] + dyData[5 + pp] * (1 - JMM[35]);
    h[1] += 0;
    h[2] += dyData[pp] * JMM[18] + dyData[1 + pp] * JMM[19] +
            dyData[2 + pp] * JMM[20] + dyData[3 + pp] * (-1 + JMM[21]) +
            dyData[4 + pp] * JMM[22] + dyData[5 + pp] * JMM[23];
    h[3] += -dyData[2 + pp];
    h[4] += 0;
    h[5] += dyData[pp];
    h[0] -= h[3] * JMM[3] + h[4] * JMM[4] + h[5] * JMM[5];
    h[1] -= h[3] * JMM[9] + h[4] * JMM[10] + h[5] * JMM[11];
    h[2] -= h[3] * JMM[15] + h[4] * JMM[16] + h[5] * JMM[17];
    dudata[pp + 0] =
        h[2] * (-(JMM[2] * (1 + JMM[7])) + JMM[1] * JMM[8]) +
        h[1] * (JMM[2] * JMM[13] - JMM[1] * (1 + JMM[14])) +
        h[0] * (1 - JMM[8] * JMM[13] + JMM[14] + JMM[7] * (1 + JMM[14]));
    dudata[pp + 1] =
        h[2] * (JMM[2] * JMM[6] - (1 + JMM[0]) * JMM[8]) +
        h[1] * (1 - JMM[2] * JMM[12] + JMM[14] + JMM[0] * (1 + JMM[14])) +
        h[0] * (JMM[8] * JMM[12] - JMM[6] * (1 + JMM[14]));
    dudata[pp + 2] =
        h[2] * (1 - JMM[1] * JMM[6] + JMM[7] + JMM[0] * (1 + JMM[7])) +
        h[1] * (JMM[1] * JMM[12] - (1 + JMM[0]) * JMM[13]) +
        h[0] * (-((1 + JMM[7]) * JMM[12]) + JMM[6] * JMM[13]);
    pseudoDenom =
        -((1 + JMM[7]) * (-1 + JMM[2] * JMM[12])) +
        (JMM[2] * JMM[6] - JMM[8]) * JMM[13] + JMM[14] + JMM[7] * JMM[14] +
        JMM[0] * (1 + JMM[7] - JMM[8] * JMM[13] + (1 + JMM[7]) * JMM[14]) -
        JMM[1] * (-(JMM[8] * JMM[12]) + JMM[6] * (1 + JMM[14]));
    dudata[pp + 0] /= pseudoDenom;
    dudata[pp + 1] /= pseudoDenom;
    dudata[pp + 2] /= pseudoDenom;
    dudata[pp + 3] = h[3];
    dudata[pp + 4] = h[4];
    dudata[pp + 5] = h[5];
  }
  return;
}

/// only under-the-hood-callable Maxwell propagation in 3D
// unused parameters 2-4 for compliance with CVRhsFn
// same as the respective nonlinear function without nonlinear terms
void linear3DProp(LatticePatch *data, N_Vector u, N_Vector udot, int *c) {

  sunrealtype *duData = data->duData;
  sunrealtype *dxData = data->buffData[1 - 1];
  sunrealtype *dyData = data->buffData[2 - 1];
  sunrealtype *dzData = data->buffData[3 - 1];

  /* Under the hood call of point-to-point or collective communication */
```

```
  // Point-to-Point:
  data->exchangeGhostCells(1);
  data->rotateIntoEigen(1);
  data->derive(1);
  data->derotate(1, dxData);
  data->exchangeGhostCells(2);
  data->rotateIntoEigen(2);
  data->derive(2);
  data->derotate(2, dyData);
  data->exchangeGhostCells(3);
  data->rotateIntoEigen(3);
  data->derive(3);
  data->derotate(3, dzData);

  // Collective:
  /*
      data->exchangeGhostCells3D();
      data->rotateIntoEigen3D();
      data->derive(1);
      data->derotate(1,dxData);
      data->derive(2);
      data->derotate(2,dyData);
      data->derive(3);
      data->derotate(3,dzData);
  */

  int totalNP = data->discreteSize();
  int pp = 0;
#pragma distribute_point
  for (int i = 0; i < totalNP; i++) {
    pp = i * 6;
    duData[pp + 0] = dyData[pp + 5] - dzData[pp + 4];
    duData[pp + 1] = dzData[pp + 3] - dxData[pp + 5];
    duData[pp + 2] = dxData[pp + 4] - dyData[pp + 3];
    duData[pp + 3] = -dyData[pp + 2] + dzData[pp + 1];
    duData[pp + 4] = -dzData[pp + 0] + dxData[pp + 2];
    duData[pp + 5] = -dxData[pp + 1] + dyData[pp + 0];
  }
}

/// nonlinear 3D HE propagation
void nonlinear3DProp(LatticePatch *data, N_Vector u, N_Vector udot, int *c) {

  sunrealtype *dxData = data->buffData[1 - 1];
  sunrealtype *dyData = data->buffData[2 - 1];
  sunrealtype *dzData = data->buffData[3 - 1];

  /* Under the hood call of point-to-point or collective communication */
  // Point-to-Point:

      data->exchangeGhostCells(1);
      data->rotateIntoEigen(1);
      data->derive(1);
      data->derotate(1,dxData);
      data->exchangeGhostCells(2);
      data->rotateIntoEigen(2);
      data->derive(2);
      data->derotate(2,dyData);
      data->exchangeGhostCells(3);
      data->rotateIntoEigen(3);
      data->derive(3);
      data->derotate(3,dzData);

  // Collective:
  /*
  data->exchangeGhostCells3D();
  data->rotateIntoEigen3D();
  data->derive(1);
  data->derotate(1, dxData);
  data->derive(2);
  data->derotate(2, dyData);
  data->derive(3);
  data->derotate(3, dzData);
  */

  sunrealtype f = NAN, g = NAN;
  sunrealtype lf = NAN, lff = NAN, lfg = NAN, lg = NAN, lgg = NAN;
  array<sunrealtype, 36> JMM;
  array<sunrealtype, 6> Quad;
  array<sunrealtype, 6> h;
  sunrealtype pseudoDenom = NAN;
  sunrealtype *udata = nullptr, *dudata = nullptr;
  udata = NV_DATA_P(u);
  dudata = NV_DATA_P(udot);
  int totalNP = data->discreteSize();
  //#pragma distribute_point
  //#pragma unroll_and_jam
```

```
  for (int pp = 0; pp < totalNP * 6; pp += 6) {
    // 1
    f = 0.5 * ((Quad[0] = udata[pp] * udata[pp]) +
               (Quad[1] = udata[pp + 1] * udata[pp + 1]) +
               (Quad[2] = udata[pp + 2] * udata[pp + 2]) -
               (Quad[3] = udata[pp + 3] * udata[pp + 3]) -
               (Quad[4] = udata[pp + 4] * udata[pp + 4]) -
               (Quad[5] = udata[pp + 5] * udata[pp + 5]));
    g = udata[pp] * udata[pp + 3] + udata[pp + 1] * udata[pp + 4] +
        udata[pp + 2] * udata[pp + 5];
    // 2
    switch (*c) {
    case 0:
      lf = 0;
      lff = 0;
      lfg = 0;
      lg = 0;
      lgg = 0;
      break;
    case 2:
      lf = 0.000354046449700427580438254 * f * f +
           0.000191775160254398272737387 * g * g;
      lff = 0.00070809289940085516087650 75 * f;
      lfg = 0.000383550320508796545474774 9 * g;
      lg = 0.000383550320508796545474774 9 * f * g;
      lgg = 0.000383550320508796545474774 9 * f;
      break;
    case 1:
      lf = 0.000206527095658582755255648 * f;
      lff = 0.000206527095658582755255648;
      lfg = 0;
      lg = 0.000361422417402519821697384 1 * g;
      lgg = 0.000361422417402519821697384 1;
      break;
    case 3:
      lf = (0.000206527095658582755255648 + 0.000354046449700427580438254 * f) *
               f +
           0.000191775160254398272737387 * g * g;
      lff = 0.000206527095658582755255648 + 0.000708092899400855160876508 * f;
      lfg = 0.000383550320508796545474774 9 * g;
      lg = (0.000361422417402519821697384 + 0.000383550320508796545474775 * f) *
           g;
      lgg = 0.000361422417402519821697384 + 0.000383550320508796545474775 * f;
      break;
    default:
      errorKill(
          "You need to specify a correct order in the weak-field expansion.");
    }
    // 3
    JMM[0] = lf + lff * Quad[0] +
             udata[3 + pp] * (2 * lfg * udata[pp] + lgg * udata[3 + pp]);
    JMM[6] =
        lff * udata[pp] * udata[1 + pp] + lfg * udata[1 + pp] * udata[3 + pp] +
        lfg * udata[pp] * udata[4 + pp] + lgg * udata[3 + pp] * udata[4 + pp];
    JMM[7] = lf + lff * Quad[1] +
             udata[4 + pp] * (2 * lfg * udata[1 + pp] + lgg * udata[4 + pp]);
    JMM[12] =
        lff * udata[pp] * udata[2 + pp] + lfg * udata[2 + pp] * udata[3 + pp] +
        lfg * udata[pp] * udata[5 + pp] + lgg * udata[3 + pp] * udata[5 + pp];
    JMM[13] = lff * udata[1 + pp] * udata[2 + pp] +
              lfg * udata[2 + pp] * udata[4 + pp] +
              lfg * udata[1 + pp] * udata[5 + pp] +
              lgg * udata[4 + pp] * udata[5 + pp];
    JMM[14] = lf + lff * Quad[2] +
              udata[5 + pp] * (2 * lfg * udata[2 + pp] + lgg * udata[5 + pp]);
    JMM[18] = lg + lfg * (Quad[0] - Quad[3 + 0]) +
              (-lff + lgg) * udata[pp] * udata[3 + pp];
    JMM[19] = -(udata[3 + pp] * (lff * udata[1 + pp] + lfg * udata[4 + pp])) +
              udata[pp] * (lfg * udata[1 + pp] + lgg * udata[4 + pp]);
    JMM[20] = -(udata[3 + pp] * (lff * udata[2 + pp] + lfg * udata[5 + pp])) +
              udata[pp] * (lfg * udata[2 + pp] + lgg * udata[5 + pp]);
    JMM[21] = -lf + lgg * Quad[0] +
              udata[3 + pp] * (-2 * lfg * udata[pp] + lff * udata[3 + pp]);
    JMM[24] = udata[1 + pp] * (lfg * udata[pp] + lgg * udata[3 + pp]) -
              (lff * udata[pp] + lfg * udata[3 + pp]) * udata[4 + pp];
    JMM[25] = lg + lfg * (Quad[1] - Quad[4 + 0]) +
              (-lff + lgg) * udata[1 + pp] * udata[4 + pp];
    JMM[26] = -(udata[4 + pp] * (lff * udata[2 + pp] + lfg * udata[5 + pp])) +
              udata[1 + pp] * (lfg * udata[2 + pp] + lgg * udata[5 + pp]);
    JMM[27] = lgg * udata[pp] * udata[1 + pp] +
              lff * udata[3 + pp] * udata[4 + pp] -
              lfg * (udata[1 + pp] * udata[3 + pp] + udata[pp] * udata[4 + pp]);
    JMM[28] = -lf + lgg * Quad[1] +
              udata[4 + pp] * (-2 * lfg * udata[1 + pp] + lff * udata[4 + pp]);
    JMM[30] = udata[2 + pp] * (lfg * udata[pp] + lgg * udata[3 + pp]) -
              (lff * udata[pp] + lfg * udata[3 + pp]) * udata[5 + pp];
    JMM[31] = udata[2 + pp] * (lfg * udata[1 + pp] + lgg * udata[4 + pp]) -
```

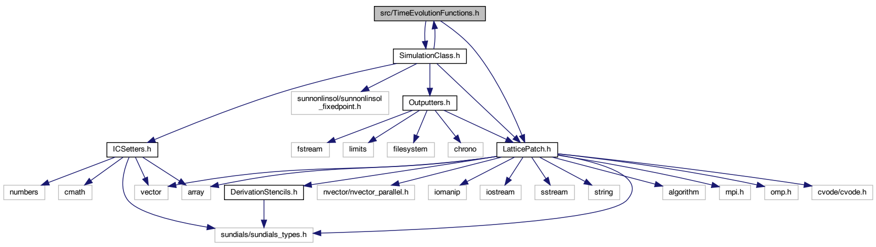
```
              (lff * udata[1 + pp] + lfg * udata[4 + pp]) * udata[5 + pp];
    JMM[32] = lg + lfg * (Quad[2] - Quad[5 + 0]) +
              (-lff + lgg) * udata[2 + pp] * udata[5 + pp];
    JMM[33] = lgg * udata[pp] * udata[2 + pp] +
              lff * udata[3 + pp] * udata[5 + pp] -
              lfg * (udata[2 + pp] * udata[3 + pp] + udata[pp] * udata[5 + pp]);
    JMM[34] =
        lgg * udata[1 + pp] * udata[2 + pp] +
        lff * udata[4 + pp] * udata[5 + pp] -
        lfg * (udata[2 + pp] * udata[4 + pp] + udata[1 + pp] * udata[5 + pp]);
    JMM[35] = -lf + lgg * Quad[2] +
              udata[5 + pp] * (-2 * lfg * udata[2 + pp] + lff * udata[5 + pp]);
    // 4
    //#pragma distribute_point
    //#pragma unroll_and_jam
    for (int i = 0; i < 6; i++) {
      for (int j = i + 1; j < 6; j++) {
        JMM[i * 6 + j] = JMM[j * 6 + i];
      }
    }
    h[0] = 0;
    h[1] = dxData[pp] * JMM[30] + dxData[1 + pp] * JMM[31] +
           dxData[2 + pp] * JMM[32] + dxData[3 + pp] * JMM[33] +
           dxData[4 + pp] * JMM[34] + dxData[5 + pp] * (-1 + JMM[35]);
    h[2] = -(dxData[pp] * JMM[24]) - dxData[1 + pp] * JMM[25] -
           dxData[2 + pp] * JMM[26] - dxData[3 + pp] * JMM[27] +
           dxData[4 + pp] * (1 - JMM[28]) - dxData[5 + pp] * JMM[29];
    h[3] = 0;
    h[4] = dxData[2 + pp];
    h[5] = -dxData[1 + pp];
    h[0] += -(dyData[pp] * JMM[30]) - dyData[1 + pp] * JMM[31] -
            dyData[2 + pp] * JMM[32] - dyData[3 + pp] * JMM[33] -
            dyData[4 + pp] * JMM[34] + dyData[5 + pp] * (1 - JMM[35]);
    h[1] += 0;
    h[2] += dyData[pp] * JMM[18] + dyData[1 + pp] * JMM[19] +
            dyData[2 + pp] * JMM[20] + dyData[3 + pp] * (-1 + JMM[21]) +
            dyData[4 + pp] * JMM[22] + dyData[5 + pp] * JMM[23];
    h[3] += -dyData[2 + pp];
    h[4] += 0;
    h[5] += dyData[pp];
    h[0] += dzData[pp] * JMM[24] + dzData[1 + pp] * JMM[25] +
            dzData[2 + pp] * JMM[26] + dzData[3 + pp] * JMM[27] +
            dzData[4 + pp] * (-1 + JMM[28]) + dzData[5 + pp] * JMM[29];
    h[1] += -(dzData[pp] * JMM[18]) - dzData[1 + pp] * JMM[19] -
            dzData[2 + pp] * JMM[20] + dzData[3 + pp] * (1 - JMM[21]) -
            dzData[4 + pp] * JMM[22] - dzData[5 + pp] * JMM[23];
    h[2] += 0;
    h[3] += dzData[1 + pp];
    h[4] += -dzData[pp];
    h[5] += 0;
    h[0] -= h[3] * JMM[3] + h[4] * JMM[4] + h[5] * JMM[5];
    h[1] -= h[3] * JMM[9] + h[4] * JMM[10] + h[5] * JMM[11];
    h[2] -= h[3] * JMM[15] + h[4] * JMM[16] + h[5] * JMM[17];
    dudata[pp + 0] =
        h[2] * (-(JMM[2] * (1 + JMM[7])) + JMM[1] * JMM[8]) +
        h[1] * (JMM[2] * JMM[13] - JMM[1] * (1 + JMM[14])) +
        h[0] * (1 - JMM[8] * JMM[13] + JMM[14] + JMM[7] * (1 + JMM[14]));
    dudata[pp + 1] =
        h[2] * (JMM[2] * JMM[6] - (1 + JMM[0]) * JMM[8]) +
        h[1] * (1 - JMM[2] * JMM[12] + JMM[14] + JMM[0] * (1 + JMM[14])) +
        h[0] * (JMM[8] * JMM[12] - JMM[6] * (1 + JMM[14]));
    dudata[pp + 2] =
        h[2] * (1 - JMM[1] * JMM[6] + JMM[7] + JMM[0] * (1 + JMM[7])) +
        h[1] * (JMM[1] * JMM[12] - (1 + JMM[0]) * JMM[13]) +
        h[0] * (-((1 + JMM[7]) * JMM[12]) + JMM[6] * JMM[13]);
    pseudoDenom =
        -((1 + JMM[7]) * (-1 + JMM[2] * JMM[12])) +
        (JMM[2] * JMM[6] - JMM[8]) * JMM[13] + JMM[14] + JMM[7] * JMM[14] +
        JMM[0] * (1 + JMM[7] - JMM[8] * JMM[13] + (1 + JMM[7]) * JMM[14]) -
        JMM[1] * (-(JMM[8] * JMM[12]) + JMM[6] * (1 + JMM[14]));
    dudata[pp + 0] /= pseudoDenom;
    dudata[pp + 1] /= pseudoDenom;
    dudata[pp + 2] /= pseudoDenom;
    dudata[pp + 3] = h[3];
    dudata[pp + 4] = h[4];
    dudata[pp + 5] = h[5];
  }
  return;
}
```

6.30 src/TimeEvolutionFunctions.h File Reference

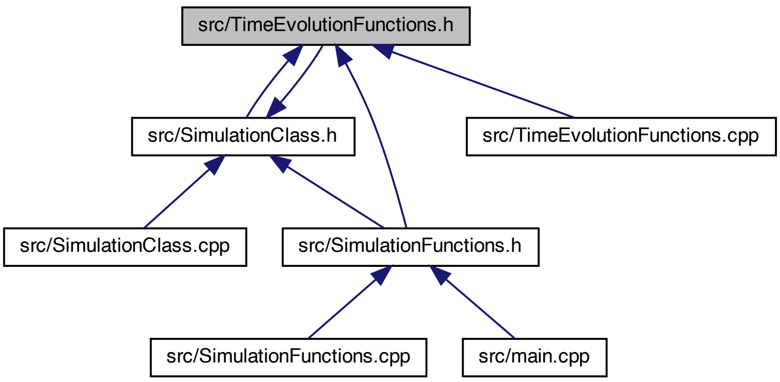
Functions to propagate data vectors in time according to Maxwell's equations, and various orders in the HE weak-field expansion.

```
#include "LatticePatch.h"
#include "SimulationClass.h"
```

Include dependency graph for TimeEvolutionFunctions.h:



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



Data Structures

- class TimeEvolution

  monostate TimeEvolution Class to propagate the field data in time in a given order of the HE weak-field expansion

Functions

- void linear1DProp (LatticePatch ∗data, N_Vector u, N_Vector udot, int ∗c)

  Maxwell propagation function for 1D – only for reference.
- void nonlinear1DProp (LatticePatch ∗data, N_Vector u, N_Vector udot, int ∗c)

  HE propagation function for 1D.
- void linear2DProp (LatticePatch ∗data, N_Vector u, N_Vector udot, int ∗c)

Maxwell propagation function for 2D – only for reference.

- void nonlinear2DProp (LatticePatch ∗data, N_Vector u, N_Vector udot, int ∗c)

  HE propagation function for 2D.

- void linear3DProp (LatticePatch ∗data, N_Vector u, N_Vector udot, int ∗c)

  Maxwell propagation function for 3D – only for reference.

- void nonlinear3DProp (LatticePatch ∗data, N_Vector u, N_Vector udot, int ∗c)

  HE propagation function for 3D.

6.30.1 Detailed Description

Functions to propagate data vectors in time according to Maxwell's equations, and various orders in the HE weak-field expansion.

Definition in file TimeEvolutionFunctions.h.

6.30.2 Function Documentation

6.30.2.1 linear1DProp()

```
void linear1DProp (
            LatticePatch * data,
            N_Vector u,
            N_Vector udot,
            int * c )
```

Maxwell propagation function for 1D – only for reference.

Maxwell propagation function for 1D – only for reference.

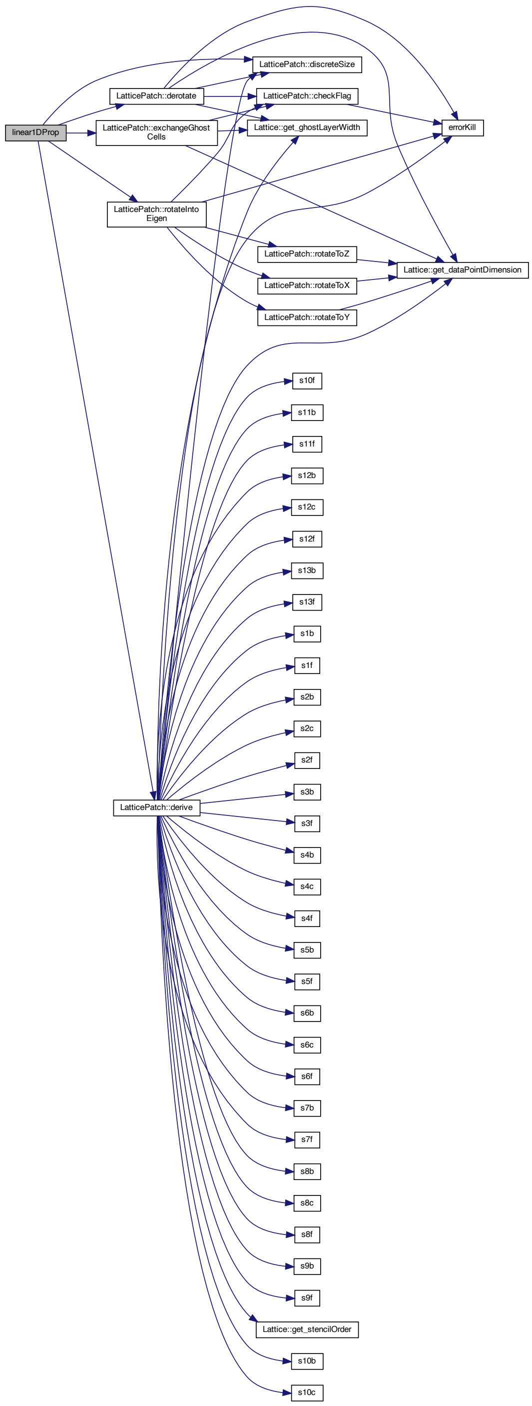
Definition at line 46 of file TimeEvolutionFunctions.cpp.

```
                                                                       {

  // pointers to temporal and spatial derivative data
  sunrealtype *duData = data->duData;
  sunrealtype *dxData = data->buffData[1 - 1];

  // sequence along any dimension:
  data->exchangeGhostCells(1); // exchange halos
  data->rotateIntoEigen(
      1);           // -> rotate all data to prepare derivative operation
  data->derive(1); // -> perform derivative on it
  data->derotate(
      1, dxData); // -> derotate derivative data to x-space for further use

  int totalNP = data->discreteSize();
  int pp = 0;
#pragma distribute_point
  for (int i = 0; i < totalNP; i++) {
    pp = i * 6;
    /*
     simple vacuum Maxwell equations for spatial deriative only in x-direction
     temporal derivative is approximated by spatial derivative according to the
     numerical scheme with Jacobi=0 -> no polarization or magnetization terms
    */
    duData[pp + 0] = 0;
    duData[pp + 1] = -dxData[pp + 5];
    duData[pp + 2] = dxData[pp + 4];
    duData[pp + 3] = 0;
    duData[pp + 4] = dxData[pp + 2];
    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.

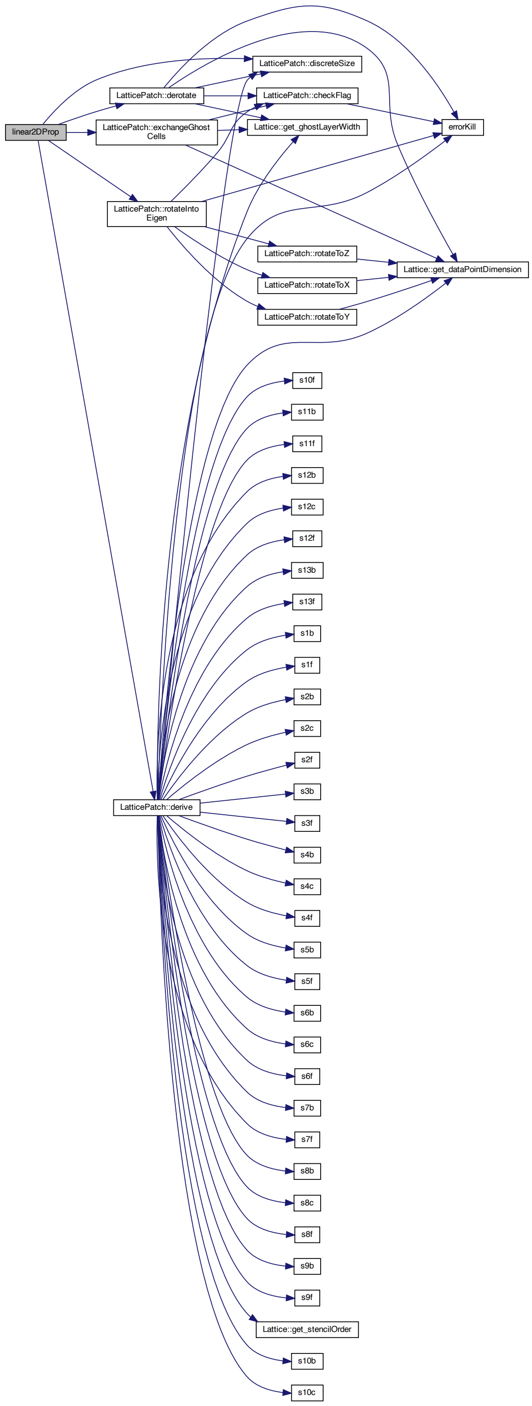
Maxwell propagation function for 2D – only for reference.

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.

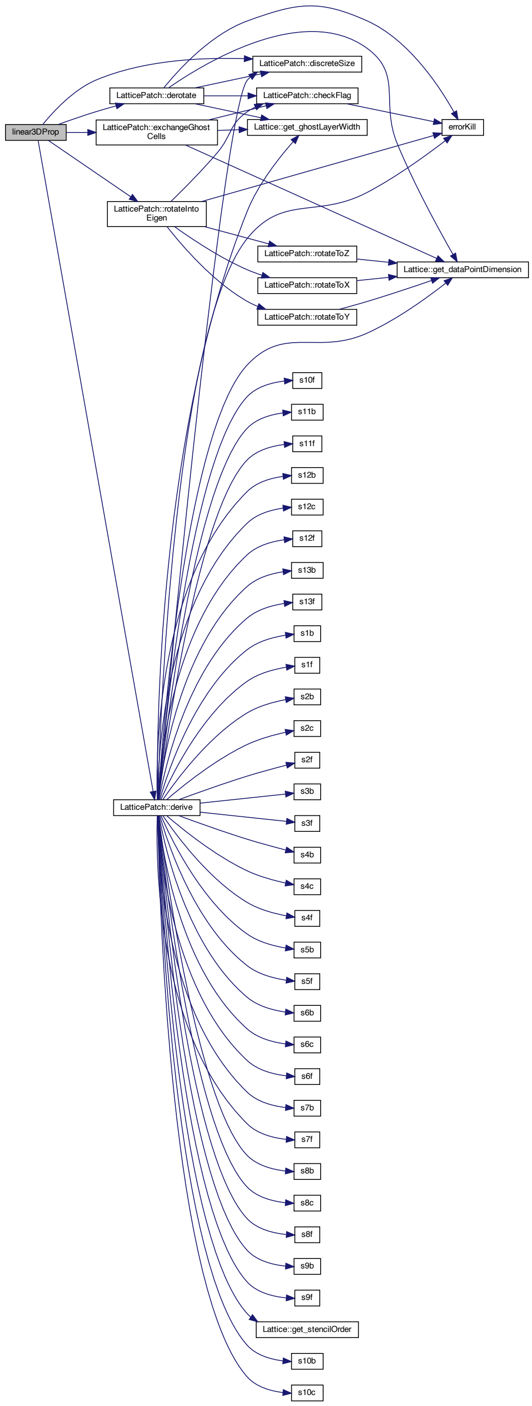
Maxwell propagation function for 3D – only for reference.

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 = 0.000708092899400855160876507 5 * f;
00144       lfg = 0.000383550320508796545474774 9 * g;
00145       lg = 0.000383550320508796545474774 9 * f * g;
00146       lgg = 0.000383550320508796545474774 9 * f;
00147       break;
00148     case 1:
00149       lf = 0.000206527095658582755255648 * f;
00150       lff = 0.000206527095658582755255648;
00151       lfg = 0;
00152       lg = 0.000361422417402519821697384 1 * g;
00153       lgg = 0.000361422417402519821697384 1;
```

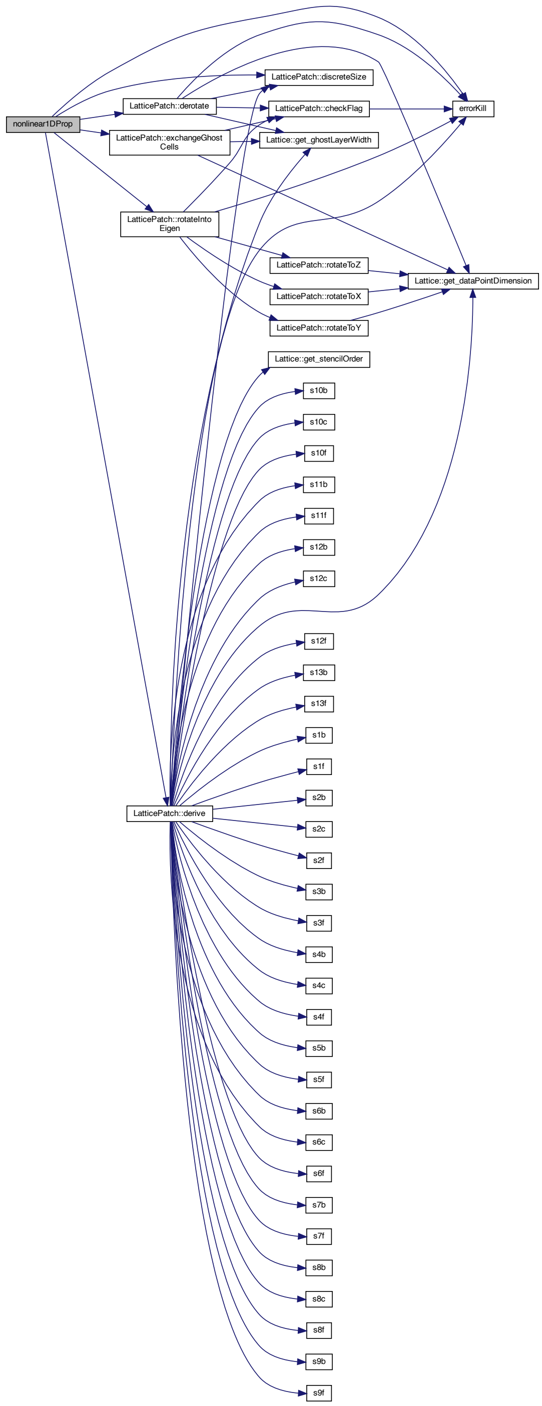
```
      break;
    case 3:
      lf = (0.00020652709565858275525564 8 + 0.0003540464497004275804382 54 * f) *
               f +
           0.00019177516025439827273738 7 * g * g;
      lff = 0.000206527095658582755255648 + 0.000708092899400855160876507 5 * f;
      lfg = 0.000383550320508796545474774 9 * g;
      lg = (0.000361422417402519821697384 1 +
            0.000383550320508796545474774 9 * f) *
           g;
      lgg = 0.000361422417402519821697384 1 + 0.000383550320508796545474774 9 * f;
      break;
    default:
      errorKill(
          "You need to specify a correct order in the weak-field expansion.");
    }
    // 3. Assign Jacobi components
    JMM[0] = lf + lff * Quad[0] +
             udata[3 + pp] * (2 * lfg * udata[pp] + lgg * udata[3 + pp]);
    JMM[6] =
        lff * udata[pp] * udata[1 + pp] + lfg * udata[1 + pp] * udata[3 + pp] +
        lfg * udata[pp] * udata[4 + pp] + lgg * udata[3 + pp] * udata[4 + pp];
    JMM[7] = lf + lff * Quad[1] +
             udata[4 + pp] * (2 * lfg * udata[1 + pp] + lgg * udata[4 + pp]);
    JMM[12] =
        lff * udata[pp] * udata[2 + pp] + lfg * udata[2 + pp] * udata[3 + pp] +
        lfg * udata[pp] * udata[5 + pp] + lgg * udata[3 + pp] * udata[5 + pp];
    JMM[13] = lff * udata[1 + pp] * udata[2 + pp] +
              lfg * udata[2 + pp] * udata[4 + pp] +
              lfg * udata[1 + pp] * udata[5 + pp] +
              lgg * udata[4 + pp] * udata[5 + pp];
    JMM[14] = lf + lff * Quad[2] +
              udata[5 + pp] * (2 * lfg * udata[2 + pp] + lgg * udata[5 + pp]);
    JMM[18] = lg + lfg * (Quad[0] - Quad[3 + 0]) +
              (-lff + lgg) * udata[pp] * udata[3 + pp];
    JMM[19] = -(udata[3 + pp] * (lff * udata[1 + pp] + lfg * udata[4 + pp])) +
              udata[pp] * (lfg * udata[1 + pp] + lgg * udata[4 + pp]);
    JMM[20] = -(udata[3 + pp] * (lff * udata[2 + pp] + lfg * udata[5 + pp])) +
              udata[pp] * (lfg * udata[2 + pp] + lgg * udata[5 + pp]);
    JMM[21] = -lf + lgg * Quad[0] +
              udata[3 + pp] * (-2 * lfg * udata[pp] + lff * udata[3 + pp]);
    JMM[24] = udata[1 + pp] * (lfg * udata[pp] + lgg * udata[3 + pp]) -
              (lff * udata[pp] + lfg * udata[3 + pp]) * udata[4 + pp];
    JMM[25] = lg + lfg * (Quad[1] - Quad[4 + 0]) +
              (-lff + lgg) * udata[1 + pp] * udata[4 + pp];
    JMM[26] = -(udata[4 + pp] * (lff * udata[2 + pp] + lfg * udata[5 + pp])) +
              udata[1 + pp] * (lfg * udata[2 + pp] + lgg * udata[5 + pp]);
    JMM[27] = lgg * udata[pp] * udata[1 + pp] +
              lff * udata[3 + pp] * udata[4 + pp] -
              lfg * (udata[1 + pp] * udata[3 + pp] + udata[pp] * udata[4 + pp]);
    JMM[28] = -lf + lgg * Quad[1] +
              udata[4 + pp] * (-2 * lfg * udata[1 + pp] + lff * udata[4 + pp]);
    JMM[30] = udata[2 + pp] * (lfg * udata[pp] + lgg * udata[3 + pp]) -
              (lff * udata[pp] + lfg * udata[3 + pp]) * udata[5 + pp];
    JMM[31] = udata[2 + pp] * (lfg * udata[1 + pp] + lgg * udata[4 + pp]) -
              (lff * udata[1 + pp] + lfg * udata[4 + pp]) * udata[5 + pp];
    JMM[32] = lg + lfg * (Quad[2] - Quad[5 + 0]) +
              (-lff + lgg) * udata[2 + pp] * udata[5 + pp];
    JMM[33] = lgg * udata[pp] * udata[2 + pp] +
              lff * udata[3 + pp] * udata[5 + pp] -
              lfg * (udata[2 + pp] * udata[3 + pp] + udata[pp] * udata[5 + pp]);
    JMM[34] =
        lgg * udata[1 + pp] * udata[2 + pp] +
        lff * udata[4 + pp] * udata[5 + pp] -
        lfg * (udata[2 + pp] * udata[4 + pp] + udata[1 + pp] * udata[5 + pp]);
    JMM[35] = -lf + lgg * Quad[2] +
              udata[5 + pp] * (-2 * lfg * udata[2 + pp] + lff * udata[5 + pp]);
    //#pragma unroll_and_jam
    //#pragma distribute_point
    for (int i = 0; i < 6; i++) {
      for (int j = i + 1; j < 6; j++) {
        JMM[i * 6 + j] = JMM[j * 6 + i];
      }
    }
    // 4. Final values for temporal derivatives of field values
    h[0] = 0;
    h[1] = dxData[pp] * JMM[30] + dxData[1 + pp] * JMM[31] +
           dxData[2 + pp] * JMM[32] + dxData[3 + pp] * JMM[33] +
           dxData[4 + pp] * JMM[34] + dxData[5 + pp] * (-1 + JMM[35]);
    h[2] = -(dxData[pp] * JMM[24]) - dxData[1 + pp] * JMM[25] -
           dxData[2 + pp] * JMM[26] - dxData[3 + pp] * JMM[27] +
           dxData[4 + pp] * (1 - JMM[28]) - dxData[5 + pp] * JMM[29];
    h[3] = 0;
    h[4] = dxData[2 + pp];
    h[5] = -dxData[1 + pp];
    h[0] -= h[3] * JMM[3] + h[4] * JMM[4] + h[5] * JMM[5];
    h[1] -= h[3] * JMM[9] + h[4] * JMM[10] + h[5] * JMM[11];
```

```
    h[2] -= h[3] * JMM[15] + h[4] * JMM[16] + h[5] * JMM[17];
    // (1+Z)^-1 applies only to E components
    dudata[pp + 0] =
        h[2] * (-(JMM[2] * (1 + JMM[7])) + JMM[1] * JMM[8]) +
        h[1] * (JMM[2] * JMM[13] - JMM[1] * (1 + JMM[14])) +
        h[0] * (1 - JMM[8] * JMM[13] + JMM[14] + JMM[7] * (1 + JMM[14]));
    dudata[pp + 1] =
        h[2] * (JMM[2] * JMM[6] - (1 + JMM[0]) * JMM[8]) +
        h[1] * (1 - JMM[2] * JMM[12] + JMM[14] + JMM[0] * (1 + JMM[14])) +
        h[0] * (JMM[8] * JMM[12] - JMM[6] * (1 + JMM[14]));
    dudata[pp + 2] =
        h[2] * (1 - JMM[1] * JMM[6] + JMM[7] + JMM[0] * (1 + JMM[7])) +
        h[1] * (JMM[1] * JMM[12] - (1 + JMM[0]) * JMM[13]) +
        h[0] * (-((1 + JMM[7]) * JMM[12]) + JMM[6] * JMM[13]);
    pseudoDenom =
        -((1 + JMM[7]) * (-1 + JMM[2] * JMM[12])) +
        (JMM[2] * JMM[6] - JMM[8]) * JMM[13] + JMM[14] + JMM[7] * JMM[14] +
        JMM[0] * (1 + JMM[7] - JMM[8] * JMM[13] + (1 + JMM[7]) * JMM[14]) -
        JMM[1] * (-(JMM[8] * JMM[12]) + JMM[6] * (1 + JMM[14]));
    dudata[pp + 0] /= pseudoDenom;
    dudata[pp + 1] /= pseudoDenom;
    dudata[pp + 2] /= pseudoDenom;
    dudata[pp + 3] = h[3];
    dudata[pp + 4] = h[4];
    dudata[pp + 5] = h[5];
  }
  return;
}
```

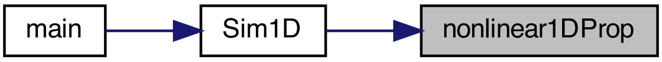
References LatticePatch::buffData, LatticePatch::derive(), LatticePatch::derotate(), LatticePatch::discreteSize(), errorKill(), LatticePatch::exchangeGhostCells(), and LatticePatch::rotateIntoEigen().

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 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;
```

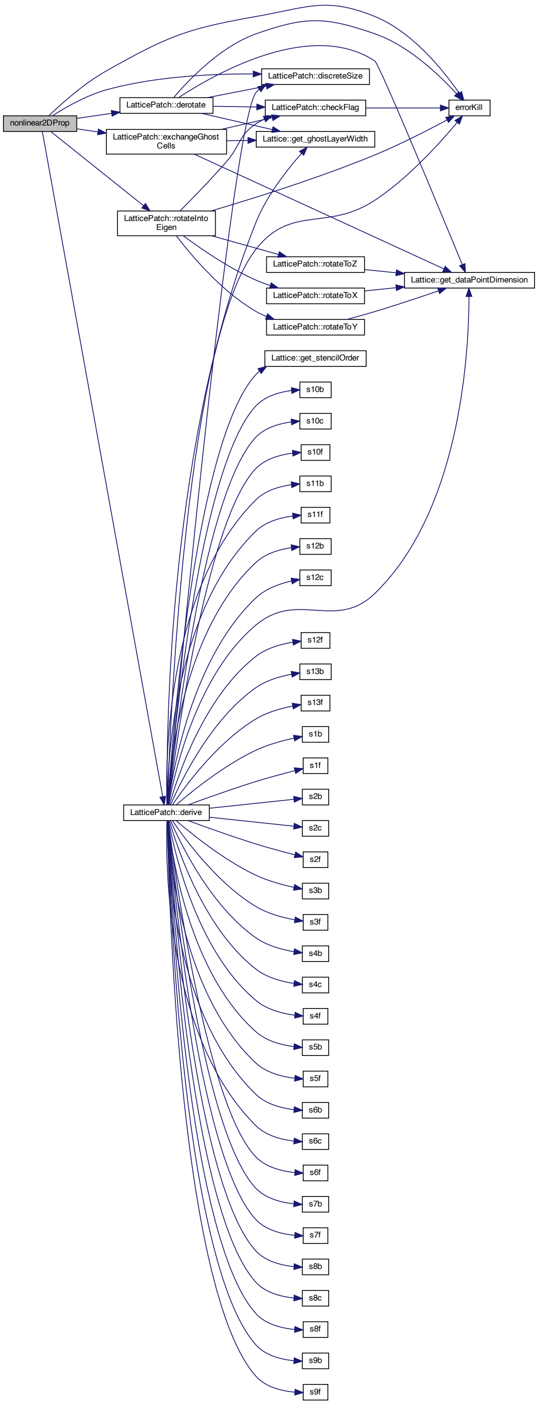
```
      lgg = 0;
      break;
    case 2:
      lf = 0.000354046449700427580438254 * f * f +
           0.000191775160254398272737387 * g * g;
      lff = 0.000708092899400855160876508 * f;
      lfg = 0.000383550320508796545474775 * g;
      lg = 0.000383550320508796545474775 * f * g;
      lgg = 0.000383550320508796545474775 * f;
      break;
    case 1:
      lf = 0.000206527095658582755255648 * f;
      lff = 0.000206527095658582755255648;
      lfg = 0;
      lg = 0.000361422417402519821697384 * g;
      lgg = 0.000361422417402519821697384;
      break;
    case 3:
      lf = (0.000206527095658582755255648 + 0.000354046449700427580438254 * f) *
               f +
           0.000191775160254398272737387 * g * g;
      lff = 0.000206527095658582755255648 + 0.000708092899400855160876508 * f;
      lfg = 0.000383550320508796545474775 * g;
      lg = (0.000361422417402519821697384 + 0.000383550320508796545474775 * f) *
           g;
      lgg = 0.000361422417402519821697384 + 0.000383550320508796545474775 * f;
      break;
    default:
      errorKill(
          "You need to specify a correct order in the weak-field expansion.");
    }
    // 3
    JMM[0] = lf + lff * Quad[0] +
             udata[3 + pp] * (2 * lfg * udata[pp] + lgg * udata[3 + pp]);
    JMM[6] =
        lff * udata[pp] * udata[1 + pp] + lfg * udata[1 + pp] * udata[3 + pp] +
        lfg * udata[pp] * udata[4 + pp] + lgg * udata[3 + pp] * udata[4 + pp];
    JMM[7] = lf + lff * Quad[1] +
             udata[4 + pp] * (2 * lfg * udata[1 + pp] + lgg * udata[4 + pp]);
    JMM[12] =
        lff * udata[pp] * udata[2 + pp] + lfg * udata[2 + pp] * udata[3 + pp] +
        lfg * udata[pp] * udata[5 + pp] + lgg * udata[3 + pp] * udata[5 + pp];
    JMM[13] = lff * udata[1 + pp] * udata[2 + pp] +
              lfg * udata[2 + pp] * udata[4 + pp] +
              lfg * udata[1 + pp] * udata[5 + pp] +
              lgg * udata[4 + pp] * udata[5 + pp];
    JMM[14] = lf + lff * Quad[2] +
              udata[5 + pp] * (2 * lfg * udata[2 + pp] + lgg * udata[5 + pp]);
    JMM[18] = lg + lfg * (Quad[0] - Quad[3 + 0]) +
              (-lff + lgg) * udata[pp] * udata[3 + pp];
    JMM[19] = -(udata[3 + pp] * (lff * udata[1 + pp] + lfg * udata[4 + pp])) +
              udata[pp] * (lfg * udata[1 + pp] + lgg * udata[4 + pp]);
    JMM[20] = -(udata[3 + pp] * (lff * udata[2 + pp] + lfg * udata[5 + pp])) +
              udata[pp] * (lfg * udata[2 + pp] + lgg * udata[5 + pp]);
    JMM[21] = -lf + lgg * Quad[0] +
              udata[3 + pp] * (-2 * lfg * udata[pp] + lff * udata[3 + pp]);
    JMM[24] = udata[1 + pp] * (lfg * udata[pp] + lgg * udata[3 + pp]) -
              (lff * udata[pp] + lfg * udata[3 + pp]) * udata[4 + pp];
    JMM[25] = lg + lfg * (Quad[1] - Quad[4 + 0]) +
              (-lff + lgg) * udata[1 + pp] * udata[4 + pp];
    JMM[26] = -(udata[4 + pp] * (lff * udata[2 + pp] + lfg * udata[5 + pp])) +
              udata[1 + pp] * (lfg * udata[2 + pp] + lgg * udata[5 + pp]);
    JMM[27] = lgg * udata[pp] * udata[1 + pp] +
              lff * udata[3 + pp] * udata[4 + pp] -
              lfg * (udata[1 + pp] * udata[3 + pp] + udata[pp] * udata[4 + pp]);
    JMM[28] = -lf + lgg * Quad[1] +
              udata[4 + pp] * (-2 * lfg * udata[1 + pp] + lff * udata[4 + pp]);
    JMM[30] = udata[2 + pp] * (lfg * udata[pp] + lgg * udata[3 + pp]) -
              (lff * udata[pp] + lfg * udata[3 + pp]) * udata[5 + pp];
    JMM[31] = udata[2 + pp] * (lfg * udata[1 + pp] + lgg * udata[4 + pp]) -
              (lff * udata[1 + pp] + lfg * udata[4 + pp]) * udata[5 + pp];
    JMM[32] = lg + lfg * (Quad[2] - Quad[5 + 0]) +
              (-lff + lgg) * udata[2 + pp] * udata[5 + pp];
    JMM[33] = lgg * udata[pp] * udata[2 + pp] +
              lff * udata[3 + pp] * udata[5 + pp] -
              lfg * (udata[2 + pp] * udata[3 + pp] + udata[pp] * udata[5 + pp]);
    JMM[34] =
        lgg * udata[1 + pp] * udata[2 + pp] +
        lff * udata[4 + pp] * udata[5 + pp] -
        lfg * (udata[2 + pp] * udata[4 + pp] + udata[1 + pp] * udata[5 + pp]);
    JMM[35] = -lf + lgg * Quad[2] +
              udata[5 + pp] * (-2 * lfg * udata[2 + pp] + lff * udata[5 + pp]);
    // 4
    //#pragma distribute_point
    //#pragma unroll_and_jam
    for (int i = 0; i < 6; i++) {
      for (int j = i + 1; j < 6; j++) {
```

```
        JMM[i * 6 + j] = JMM[j * 6 + i];
      }
    }
    h[0] = 0;
    h[1] = dxData[pp] * JMM[30] + dxData[1 + pp] * JMM[31] +
           dxData[2 + pp] * JMM[32] + dxData[3 + pp] * JMM[33] +
           dxData[4 + pp] * JMM[34] + dxData[5 + pp] * (-1 + JMM[35]);
    h[2] = -(dxData[pp] * JMM[24]) - dxData[1 + pp] * JMM[25] -
           dxData[2 + pp] * JMM[26] - dxData[3 + pp] * JMM[27] +
           dxData[4 + pp] * (1 - JMM[28]) - dxData[5 + pp] * JMM[29];
    h[3] = 0;
    h[4] = dxData[2 + pp];
    h[5] = -dxData[1 + pp];
    h[0] += -(dyData[pp] * JMM[30]) - dyData[1 + pp] * JMM[31] -
            dyData[2 + pp] * JMM[32] - dyData[3 + pp] * JMM[33] -
            dyData[4 + pp] * JMM[34] + dyData[5 + pp] * (1 - JMM[35]);
    h[1] += 0;
    h[2] += dyData[pp] * JMM[18] + dyData[1 + pp] * JMM[19] +
            dyData[2 + pp] * JMM[20] + dyData[3 + pp] * (-1 + JMM[21]) +
            dyData[4 + pp] * JMM[22] + dyData[5 + pp] * JMM[23];
    h[3] += -dyData[2 + pp];
    h[4] += 0;
    h[5] += dyData[pp];
    h[0] -= h[3] * JMM[3] + h[4] * JMM[4] + h[5] * JMM[5];
    h[1] -= h[3] * JMM[9] + h[4] * JMM[10] + h[5] * JMM[11];
    h[2] -= h[3] * JMM[15] + h[4] * JMM[16] + h[5] * JMM[17];
    dudata[pp + 0] =
        h[2] * (-(JMM[2] * (1 + JMM[7])) + JMM[1] * JMM[8]) +
        h[1] * (JMM[2] * JMM[13] - JMM[1] * (1 + JMM[14])) +
        h[0] * (1 - JMM[8] * JMM[13] + JMM[14] + JMM[7] * (1 + JMM[14]));
    dudata[pp + 1] =
        h[2] * (JMM[2] * JMM[6] - (1 + JMM[0]) * JMM[8]) +
        h[1] * (1 - JMM[2] * JMM[12] + JMM[14] + JMM[0] * (1 + JMM[14])) +
        h[0] * (JMM[8] * JMM[12] - JMM[6] * (1 + JMM[14]));
    dudata[pp + 2] =
        h[2] * (1 - JMM[1] * JMM[6] + JMM[7] + JMM[0] * (1 + JMM[7])) +
        h[1] * (JMM[1] * JMM[12] - (1 + JMM[0]) * JMM[13]) +
        h[0] * (-((1 + JMM[7]) * JMM[12]) + JMM[6] * JMM[13]);
    pseudoDenom =
        -((1 + JMM[7]) * (-1 + JMM[2] * JMM[12])) +
        (JMM[2] * JMM[6] - JMM[8]) * JMM[13] + JMM[14] + JMM[7] * JMM[14] +
        JMM[0] * (1 + JMM[7] - JMM[8] * JMM[13] + (1 + JMM[7]) * JMM[14]) -
        JMM[1] * (-(JMM[8] * JMM[12]) + JMM[6] * (1 + JMM[14]));
    dudata[pp + 0] /= pseudoDenom;
    dudata[pp + 1] /= pseudoDenom;
    dudata[pp + 2] /= pseudoDenom;
    dudata[pp + 3] = h[3];
    dudata[pp + 4] = h[4];
    dudata[pp + 5] = h[5];
  }
  return;
}
```

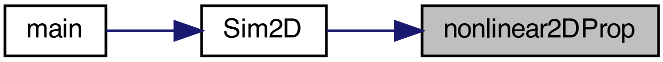
References LatticePatch::buffData, LatticePatch::derive(), LatticePatch::derotate(), LatticePatch::discreteSize(), errorKill(), LatticePatch::exchangeGhostCells(), and LatticePatch::rotateIntoEigen().

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.

```
                                                                                {

  sunrealtype *dxData = data->buffData[1 - 1];
  sunrealtype *dyData = data->buffData[2 - 1];
  sunrealtype *dzData = data->buffData[3 - 1];

  /* Under the hood call of point-to-point or collective communication */
  // Point-to-Point:

      data->exchangeGhostCells(1);
      data->rotateIntoEigen(1);
      data->derive(1);
      data->derotate(1,dxData);
      data->exchangeGhostCells(2);
      data->rotateIntoEigen(2);
      data->derive(2);
      data->derotate(2,dyData);
      data->exchangeGhostCells(3);
      data->rotateIntoEigen(3);
      data->derive(3);
      data->derotate(3,dzData);

  // Collective:
  /*
  data->exchangeGhostCells3D();
  data->rotateIntoEigen3D();
  data->derive(1);
  data->derotate(1, dxData);
  data->derive(2);
  data->derotate(2, dyData);
  data->derive(3);
  data->derotate(3, dzData);
  */

  sunrealtype f = NAN, g = NAN;
  sunrealtype lf = NAN, lff = NAN, lfg = NAN, lg = NAN, lgg = NAN;
  array<sunrealtype, 36> JMM;
  array<sunrealtype, 6> Quad;
  array<sunrealtype, 6> h;
  sunrealtype pseudoDenom = NAN;
  sunrealtype *udata = nullptr, *dudata = nullptr;
  udata = NV_DATA_P(u);
  dudata = NV_DATA_P(udot);
```

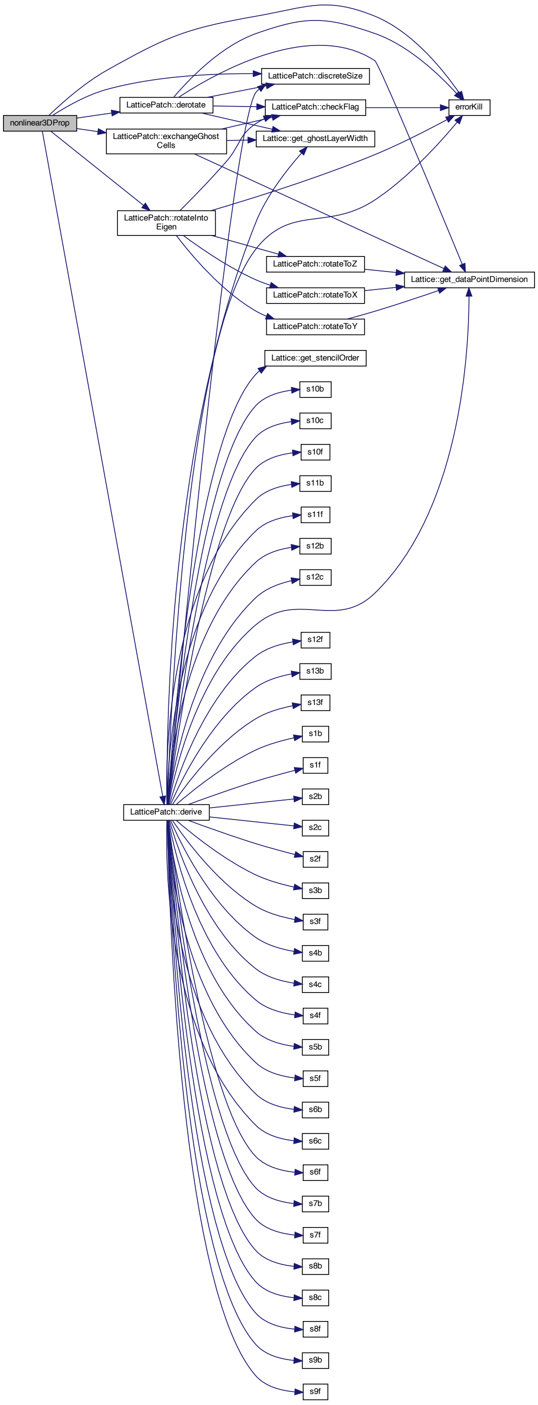
```
  int totalNP = data->discreteSize();
  //#pragma distribute_point
  //#pragma unroll_and_jam
  for (int pp = 0; pp < totalNP * 6; pp += 6) {
    // 1
    f = 0.5 * ((Quad[0] = udata[pp] * udata[pp]) +
               (Quad[1] = udata[pp + 1] * udata[pp + 1]) +
               (Quad[2] = udata[pp + 2] * udata[pp + 2]) -
               (Quad[3] = udata[pp + 3] * udata[pp + 3]) -
               (Quad[4] = udata[pp + 4] * udata[pp + 4]) -
               (Quad[5] = udata[pp + 5] * udata[pp + 5]));
    g = udata[pp] * udata[pp + 3] + udata[pp + 1] * udata[pp + 4] +
        udata[pp + 2] * udata[pp + 5];
    // 2
    switch (*c) {
    case 0:
      lf = 0;
      lff = 0;
      lfg = 0;
      lg = 0;
      lgg = 0;
      break;
    case 2:
      lf = 0.000354046449700427580438254 * f * f +
           0.000191775160254398272737387 * g * g;
      lff = 0.000708092899400855160876507 * f;
      lfg = 0.000383550320508796545474774 * g;
      lg = 0.000383550320508796545474774 * f * g;
      lgg = 0.000383550320508796545474774 * f;
      break;
    case 1:
      lf = 0.000206527095658582755255648 * f;
      lff = 0.000206527095658582755255648;
      lfg = 0;
      lg = 0.000361422417402519821697384 * g;
      lgg = 0.000361422417402519821697384;
      break;
    case 3:
      lf = (0.000206527095658582755255648 + 0.000354046449700427580438254 * f) *
               f +
           0.000191775160254398272737387 * g * g;
      lff = 0.000206527095658582755255648 + 0.000708092899400855160876508 * f;
      lfg = 0.000383550320508796545474774 * g;
      lg = (0.000361422417402519821697384 + 0.000383550320508796545474775 * f) *
           g;
      lgg = 0.000361422417402519821697384 + 0.000383550320508796545474775 * f;
      break;
    default:
      errorKill(
          "You need to specify a correct order in the weak-field expansion.");
    }
    // 3
    JMM[0] = lf + lff * Quad[0] +
             udata[3 + pp] * (2 * lfg * udata[pp] + lgg * udata[3 + pp]);
    JMM[6] =
        lff * udata[pp] * udata[1 + pp] + lfg * udata[1 + pp] * udata[3 + pp] +
        lfg * udata[pp] * udata[4 + pp] + lgg * udata[3 + pp] * udata[4 + pp];
    JMM[7] = lf + lff * Quad[1] +
             udata[4 + pp] * (2 * lfg * udata[1 + pp] + lgg * udata[4 + pp]);
    JMM[12] =
        lff * udata[pp] * udata[2 + pp] + lfg * udata[2 + pp] * udata[3 + pp] +
        lfg * udata[pp] * udata[5 + pp] + lgg * udata[3 + pp] * udata[5 + pp];
    JMM[13] = lff * udata[1 + pp] * udata[2 + pp] +
              lfg * udata[2 + pp] * udata[4 + pp] +
              lfg * udata[1 + pp] * udata[5 + pp] +
              lgg * udata[4 + pp] * udata[5 + pp];
    JMM[14] = lf + lff * Quad[2] +
              udata[5 + pp] * (2 * lfg * udata[2 + pp] + lgg * udata[5 + pp]);
    JMM[18] = lg + lfg * (Quad[0] - Quad[3 + 0]) +
              (-lff + lgg) * udata[pp] * udata[3 + pp];
    JMM[19] = -(udata[3 + pp] * (lff * udata[1 + pp] + lfg * udata[4 + pp])) +
              udata[pp] * (lfg * udata[1 + pp] + lgg * udata[4 + pp]);
    JMM[20] = -(udata[3 + pp] * (lff * udata[2 + pp] + lfg * udata[5 + pp])) +
              udata[pp] * (lfg * udata[2 + pp] + lgg * udata[5 + pp]);
    JMM[21] = -lf + lgg * Quad[0] +
              udata[3 + pp] * (-2 * lfg * udata[pp] + lff * udata[3 + pp]);
    JMM[24] = udata[1 + pp] * (lfg * udata[pp] + lgg * udata[3 + pp]) -
              (lff * udata[pp] + lfg * udata[3 + pp]) * udata[4 + pp];
    JMM[25] = lg + lfg * (Quad[1] - Quad[4 + 0]) +
              (-lff + lgg) * udata[1 + pp] * udata[4 + pp];
    JMM[26] = -(udata[4 + pp] * (lff * udata[2 + pp] + lfg * udata[5 + pp])) +
              udata[1 + pp] * (lfg * udata[2 + pp] + lgg * udata[5 + pp]);
    JMM[27] = lgg * udata[pp] * udata[1 + pp] +
              lff * udata[3 + pp] * udata[4 + pp] -
              lfg * (udata[1 + pp] * udata[3 + pp] + udata[pp] * udata[4 + pp]);
    JMM[28] = -lf + lgg * Quad[1] +
              udata[4 + pp] * (-2 * lfg * udata[1 + pp] + lff * udata[4 + pp]);
```

```
    JMM[30] = udata[2 + pp] * (lfg * udata[pp] + lgg * udata[3 + pp]) -
              (lff * udata[pp] + lfg * udata[3 + pp]) * udata[5 + pp];
    JMM[31] = udata[2 + pp] * (lfg * udata[1 + pp] + lgg * udata[4 + pp]) -
              (lff * udata[1 + pp] + lfg * udata[4 + pp]) * udata[5 + pp];
    JMM[32] = lg + lfg * (Quad[2] - Quad[5 + 0]) +
              (-lff + lgg) * udata[2 + pp] * udata[5 + pp];
    JMM[33] = lgg * udata[pp] * udata[2 + pp] +
              lff * udata[3 + pp] * udata[5 + pp] -
              lfg * (udata[2 + pp] * udata[3 + pp] + udata[pp] * udata[5 + pp]);
    JMM[34] =
        lgg * udata[1 + pp] * udata[2 + pp] +
        lff * udata[4 + pp] * udata[5 + pp] -
        lfg * (udata[2 + pp] * udata[4 + pp] + udata[1 + pp] * udata[5 + pp]);
    JMM[35] = -lf + lgg * Quad[2] +
              udata[5 + pp] * (-2 * lfg * udata[2 + pp] + lff * udata[5 + pp]);
    // 4
    //#pragma distribute_point
    //#pragma unroll_and_jam
    for (int i = 0; i < 6; i++) {
      for (int j = i + 1; j < 6; j++) {
        JMM[i * 6 + j] = JMM[j * 6 + i];
      }
    }
    h[0] = 0;
    h[1] = dxData[pp] * JMM[30] + dxData[1 + pp] * JMM[31] +
           dxData[2 + pp] * JMM[32] + dxData[3 + pp] * JMM[33] +
           dxData[4 + pp] * JMM[34] + dxData[5 + pp] * (-1 + JMM[35]);
    h[2] = -(dxData[pp] * JMM[24]) - dxData[1 + pp] * JMM[25] -
           dxData[2 + pp] * JMM[26] - dxData[3 + pp] * JMM[27] +
           dxData[4 + pp] * (1 - JMM[28]) - dxData[5 + pp] * JMM[29];
    h[3] = 0;
    h[4] = dxData[2 + pp];
    h[5] = -dxData[1 + pp];
    h[0] += -(dyData[pp] * JMM[30]) - dyData[1 + pp] * JMM[31] -
            dyData[2 + pp] * JMM[32] - dyData[3 + pp] * JMM[33] -
            dyData[4 + pp] * JMM[34] + dyData[5 + pp] * (1 - JMM[35]);
    h[1] += 0;
    h[2] += dyData[pp] * JMM[18] + dyData[1 + pp] * JMM[19] +
            dyData[2 + pp] * JMM[20] + dyData[3 + pp] * (-1 + JMM[21]) +
            dyData[4 + pp] * JMM[22] + dyData[5 + pp] * JMM[23];
    h[3] += -dyData[2 + pp];
    h[4] += 0;
    h[5] += dyData[pp];
    h[0] += dzData[pp] * JMM[24] + dzData[1 + pp] * JMM[25] +
            dzData[2 + pp] * JMM[26] + dzData[3 + pp] * JMM[27] +
            dzData[4 + pp] * (-1 + JMM[28]) + dzData[5 + pp] * JMM[29];
    h[1] += -(dzData[pp] * JMM[18]) - dzData[1 + pp] * JMM[19] -
            dzData[2 + pp] * JMM[20] + dzData[3 + pp] * (1 - JMM[21]) -
            dzData[4 + pp] * JMM[22] - dzData[5 + pp] * JMM[23];
    h[2] += 0;
    h[3] += dzData[1 + pp];
    h[4] += -dzData[pp];
    h[5] += 0;
    h[0] -= h[3] * JMM[3] + h[4] * JMM[4] + h[5] * JMM[5];
    h[1] -= h[3] * JMM[9] + h[4] * JMM[10] + h[5] * JMM[11];
    h[2] -= h[3] * JMM[15] + h[4] * JMM[16] + h[5] * JMM[17];
    dudata[pp + 0] =
        h[2] * (-(JMM[2] * (1 + JMM[7])) + JMM[1] * JMM[8]) +
        h[1] * (JMM[2] * JMM[13] - JMM[1] * (1 + JMM[14])) +
        h[0] * (1 - JMM[8] * JMM[13] + JMM[14] + JMM[7] * (1 + JMM[14]));
    dudata[pp + 1] =
        h[2] * (JMM[2] * JMM[6] - (1 + JMM[0]) * JMM[8]) +
        h[1] * (1 - JMM[2] * JMM[12] + JMM[14] + JMM[0] * (1 + JMM[14])) +
        h[0] * (JMM[8] * JMM[12] - JMM[6] * (1 + JMM[14]));
    dudata[pp + 2] =
        h[2] * (1 - JMM[1] * JMM[6] + JMM[7] + JMM[0] * (1 + JMM[7])) +
        h[1] * (JMM[1] * JMM[12] - (1 + JMM[0]) * JMM[13]) +
        h[0] * (-((1 + JMM[7]) * JMM[12]) + JMM[6] * JMM[13]);
    pseudoDenom =
        -((1 + JMM[7]) * (-1 + JMM[2] * JMM[12])) +
        (JMM[2] * JMM[6] - JMM[8]) * JMM[13] + JMM[14] + JMM[7] * JMM[14] +
        JMM[0] * (1 + JMM[7] - JMM[8] * JMM[13] + (1 + JMM[7]) * JMM[14]) -
        JMM[1] * (-(JMM[8] * JMM[12]) + JMM[6] * (1 + JMM[14]));
    dudata[pp + 0] /= pseudoDenom;
    dudata[pp + 1] /= pseudoDenom;
    dudata[pp + 2] /= pseudoDenom;
    dudata[pp + 3] = h[3];
    dudata[pp + 4] = h[4];
    dudata[pp + 5] = h[5];
  }
  return;
}
```

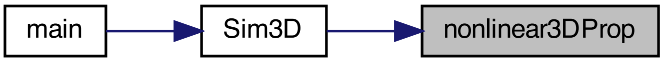
References LatticePatch::buffData, LatticePatch::derive(), LatticePatch::derotate(), LatticePatch::discreteSize(), errorKill(), LatticePatch::exchangeGhostCells(), and LatticePatch::rotateIntoEigen().

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.

```
////////////////////////////////////////////////////////
/// @file TimeEvolutionFunctions.h
/// @brief Functions to propagate data vectors in time
/// according to Maxwell's equations, and various
/// orders in the HE weak-field expansion
////////////////////////////////////////////////////////

#pragma once

#include "LatticePatch.h"
#include "SimulationClass.h"

/** @brief monostate TimeEvolution Class to propagate the field data in time in
 * a given order of the HE weak-field expansion */
class TimeEvolution {
public:
  /// choice which processes of the weak field expansion are included
  static int *c;

  /// Pointer to functions for differentiation and time evolution
  static void (*TimeEvolver)(LatticePatch *, N_Vector, N_Vector, int *);

  /// CVODE right hand side function (CVRhsFn) to provide IVP of the ODE
  static int f(sunrealtype t, N_Vector u, N_Vector udot, void *data_loc);
};

/// Maxwell propagation function for 1D -- only for reference
void linear1DProp(LatticePatch *data, N_Vector u, N_Vector udot, int *c);
/// HE propagation function for 1D
void nonlinear1DProp(LatticePatch *data, N_Vector u, N_Vector udot, int *c);
/// Maxwell propagation function for 2D -- only for reference
void linear2DProp(LatticePatch *data, N_Vector u, N_Vector udot, int *c);
/// HE propagation function for 2D
void nonlinear2DProp(LatticePatch *data, N_Vector u, N_Vector udot, int *c);
/// Maxwell propagation function for 3D -- only for reference
void linear3DProp(LatticePatch *data, N_Vector u, N_Vector udot, int *c);
/// HE propagation function for 3D
void nonlinear3DProp(LatticePatch *data, N_Vector u, N_Vector udot, int *c);
```

Index