HEWES : Heisenberg-Euler Weak-Field Expansion Simulator

Generated by Doxygen 1.9.4

Chapter 1

HEWES: Heisenberg-Euler Weak-Field Expansion Simulator

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

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

1.1 Contents

1.2 Preparing the Makefile

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

Additionally required software:

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

1.3 Short User Manual

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

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

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

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

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

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

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

1.3.1 Note on Simulation Settings

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

Example scenarios of colliding Gaussians are preconfigured for any dimension.

1.3.2 Note on Resource Occupation

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

1.3.3 Note on Output Analysis

The field data are either written in csv format to one file per MPI process, the ending of which (after an underscore) corresponds to the process number, as described above. This is not an elegant solution, but a portable way that also works fast and is straightforward to analyze.
Or, the option recommended for many larger write operations, in binary format with a single file per output step. Raw bytes are written to the files as they are in memory. This option is more performant and achieved with MPI IO. However, there is no guarantee of portability; postprocessing/conversion is required. The step number is the file name.
A `SimResults` folder is created in the chosen output directory if it does not exist and a folder named after the starting timestep of the simulation (in the form `yy-mm-dd_hh-MM-ss`) is created where the output files are written into. There are six columns in the csv files, corresponding to the six components of the electromagnetic

field: E_x, E_y, E_z, B_x, B_y, B_z. Each row corresponds to one lattice point.
Postprocessing is required to read-in the files in order. A Python module taking care of this is provided.
Likewise, another Python module is provided to read the binary data of a selected field component into a numpy array – its portability, however, cannot be guaranteed.
The process numbers first align along dimension 1 until the number of patches is that direction is reached, then continue on dimension two and finally fill dimension 3. For example, for a 3D simulation on 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 an open-access paper.
Some example Python analysis scripts can be found in the examples. The first steps demonstrate how the simulated data is accurately read-in from disk to numpy arrays using the provided get field data module. Harmonic generation in various forms is sketched as one application showing nonlinear quantum vacuum effects. Analyses of 3D simulations are more involved due to large volumes of data. Visualization requires tools like Paraview; examples are shown here. There is however *no simulation data provided* as it would make the repository size unnecessarily large.

1.4 Authors

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

Chapter 2

Hierarchical Index

2.1 Class Hierarchy

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

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 pulses in 1D

`#include <src/ICSetters.h>`

Public Member Functions

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

    construction with default parameters
- void addToSpace (sunrealtype x, sunrealtype y, sunrealtype z, sunrealtype *pTo6Space) const

    function for the actual implementation in space

Private Attributes

- sunrealtype kx

    wavenumber k_x
- sunrealtype ky

    wavenumber k_y
- sunrealtype kz

    wavenumber k_z
- sunrealtype px

    polarization & amplitude in x-direction, p_x
- sunrealtype py

    polarization & amplitude in y-direction, p_y
- sunrealtype pz

    polarization & amplitude in z-direction, p_z
- sunrealtype phix

    phase shift in x-direction, ϕ_x
- sunrealtype phiy

    phase shift in y-direction, ϕ_y
- sunrealtype phiz

phase shift in z-direction, ϕ_z

- sunrealtype x0x

    center of pulse in x-direction, x_0

- sunrealtype x0y

    center of pulse in y-direction, y_0

- sunrealtype x0z

    center of pulse in z-direction, z_0

- sunrealtype phig

    pulse width Φ_g

5.1.1 Detailed Description

class for Gaussian pulses in 1D

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

Definition at line 83 of file ICSetters.h.

5.1.2 Constructor & Destructor Documentation

5.1.2.1 Gauss1D()

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

construction with default parameters

Gauss1D construction with

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

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

Definition at line 125 of file ICSetters.cpp.

```
                              {
  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 144 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 *
      std::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 std::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 86 of file ICSetters.h.

Referenced by addToSpace(), and Gauss1D().

5.1.4.2 ky

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

wavenumber k_y

Definition at line 88 of file ICSetters.h.

Referenced by addToSpace(), and Gauss1D().

5.1.4.3 kz

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

wavenumber k_z

Definition at line 90 of file ICSetters.h.

Referenced by addToSpace(), and Gauss1D().

5.1.4.4 phig

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

pulse width Φ_g

Definition at line 110 of file ICSetters.h.

Referenced by addToSpace(), and Gauss1D().

5.1.4.5 phix

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

phase shift in x-direction, ϕ_x

Definition at line 98 of file ICSetters.h.

Referenced by addToSpace(), and Gauss1D().

5.1.4.6 phiy

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

phase shift in y-direction, ϕ_y

Definition at line 100 of file ICSetters.h.

Referenced by addToSpace(), and Gauss1D().

5.1.4.7 phiz

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

phase shift in z-direction, ϕ_z

Definition at line 102 of file ICSetters.h.

Referenced by addToSpace(), and Gauss1D().

5.1.4.8 px

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

polarization & amplitude in x-direction, p_x

Definition at line 92 of file ICSetters.h.

Referenced by addToSpace(), and Gauss1D().

5.1.4.9 py

`sunrealtype Gauss1D::py [private]`

polarization & amplitude in y-direction, p_y

Definition at line 94 of file ICSetters.h.

Referenced by addToSpace(), and Gauss1D().

5.1.4.10 pz

`sunrealtype Gauss1D::pz [private]`

polarization & amplitude in z-direction, p_z

Definition at line 96 of file ICSetters.h.

Referenced by addToSpace(), and Gauss1D().

5.1.4.11 x0x

`sunrealtype Gauss1D::x0x [private]`

center of pulse in x-direction, x_0

Definition at line 104 of file ICSetters.h.

Referenced by addToSpace(), and Gauss1D().

5.1.4.12 x0y

`sunrealtype Gauss1D::x0y [private]`

center of pulse in y-direction, y_0

Definition at line 106 of file ICSetters.h.

Referenced by addToSpace(), and Gauss1D().

5.1.4.13 x0z

`sunrealtype Gauss1D::x0z [private]`

center of pulse in z-direction, z_0

Definition at line 108 of file ICSetters.h.

Referenced by addToSpace(), and Gauss1D().

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

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

5.2 Gauss2D Class Reference

class for Gaussian pulses in 2D

`#include <src/ICSetters.h>`

Public Member Functions

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

  construction with default parameters
- void addToSpace (sunrealtype x, sunrealtype y, sunrealtype z, sunrealtype *pTo6Space) const

  function for the actual implementation in space

Private Attributes

- std::array< sunrealtype, 3 > dis

  distance maximum to origin
- std::array< sunrealtype, 3 > axis

  normalized propagation axis
- sunrealtype Amp

  amplitude A
- sunrealtype phip

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

  taille ω_0
- sunrealtype zr

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

  center of beam Φ_0
- sunrealtype PhA

  length of beam Φ_A
- sunrealtype A1

  amplitude projection on TE-mode
- sunrealtype A2

  amplitude projection on xy-plane
- sunrealtype lambda

  wavelength λ

5.2.1 Detailed Description

class for Gaussian pulses in 2D

They are given in the form $\vec{E} = A\,\vec{\epsilon}\,\sqrt{\frac{\omega_0}{\omega(z)}}\,\exp\left(-r/\omega(z)\right)^2\,\exp\left(-((z_g-\Phi_0)/\Phi_A)^2\right)\,\cos\left(\frac{k\,r^2}{2R(z)}+g(z)-k\,z_g\right)$ 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 139 of file ICSetters.h.

5.2.2 Constructor & Destructor Documentation

5.2.2.1 Gauss2D()

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

construction with default parameters

Gauss2D construction with

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

Definition at line 172 of file ICSetters.cpp.

```
                                                                 {
    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 = std::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 189 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("0x12345");  // unused parameter
    //  \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$
    // waist at position z
    const sunrealtype wz = w0 * sqrt(1 + (zg * zg / zr / zr));
    // \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("0x12345"); // beam curvature
    if (abs(zg) > 1e-15)
      Rz = zg * (1 + (zr * zr / zg / zg));
    else
      Rz = 1e308;
    // wavenumber \f$ k = 2\pi/\lambda \f$
    const sunrealtype k = 2 * std::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
  // no z-component for 2D propagation
  const sunrealtype sa = sqrt(1 - ca * ca);
  // 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 159 of file ICSetters.h.

Referenced by addToSpace(), and Gauss2D().

5.2.4.2 A2

`sunrealtype Gauss2D::A2 [private]`

amplitude projection on xy-plane

Definition at line 161 of file ICSetters.h.

Referenced by addToSpace(), and Gauss2D().

5.2.4.3 Amp

`sunrealtype Gauss2D::Amp [private]`

amplitude A

Definition at line 146 of file ICSetters.h.

Referenced by Gauss2D().

5.2.4.4 axis

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

normalized propagation axis

Definition at line 144 of file ICSetters.h.

Referenced by addToSpace(), and Gauss2D().

5.2.4.5 dis

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

distance maximum to origin

Definition at line 142 of file ICSetters.h.

Referenced by addToSpace(), and Gauss2D().

5.2.4.6 lambda

`sunrealtype Gauss2D::lambda [private]`

wavelength λ

Definition at line 163 of file ICSetters.h.

Referenced by addToSpace(), and Gauss2D().

5.2.4.7 Ph0

`sunrealtype Gauss2D::Ph0 [private]`

center of beam Φ_0

Definition at line 155 of file ICSetters.h.

Referenced by addToSpace(), and Gauss2D().

5.2.4.8 PhA

`sunrealtype Gauss2D::PhA [private]`

length of beam Φ_A

Definition at line 157 of file ICSetters.h.

Referenced by addToSpace(), and Gauss2D().

5.2.4.9 phip

`sunrealtype Gauss2D::phip [private]`

polarization rotation from TE-mode around propagation direction

Definition at line 149 of file ICSetters.h.

Referenced by Gauss2D().

5.2.4.10 w0

`sunrealtype Gauss2D::w0 [private]`

taille ω_0

Definition at line 151 of file ICSetters.h.

Referenced by addToSpace(), and Gauss2D().

5.2.4.11 zr

`sunrealtype Gauss2D::zr [private]`

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

Definition at line 153 of file ICSetters.h.

Referenced by addToSpace(), and Gauss2D().

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

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

5.3 Gauss3D Class Reference

class for Gaussian pulses in 3D

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

Public Member Functions

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

    construction with default parameters
- void addToSpace (sunrealtype x, sunrealtype y, sunrealtype z, sunrealtype ∗pTo6Space) const

    function for the actual implementation in space

Private Attributes

- std::array< sunrealtype, 3 > dis

    distance maximum to origin
- std::array< sunrealtype, 3 > axis

    normalized propagation axis
- sunrealtype Amp

    amplitude A
- sunrealtype phip

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

    taille ω_0
- sunrealtype zr

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

    center of beam Φ_0
- sunrealtype PhA

    length of beam Φ_A
- sunrealtype A1

    amplitude projection on TE-mode (z-axis)
- sunrealtype A2

    amplitude projection on xy-plane
- sunrealtype lambda

    wavelength λ

5.3.1 Detailed Description

class for Gaussian pulses in 3D

They are given in the form $\vec{E} = A\,\vec{\epsilon}\,\frac{\omega_0}{\omega(z)}\,\exp\left(-r/\omega(z)\right)^2\,\exp\left(-((z_g-\Phi_0)/\Phi_A)^2\right)\,\cos\left(\frac{k\,r^2}{2R(z)}+g(z)-k\,z_g\right)$ 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 193 of file ICSetters.h.

5.3.2 Constructor & Destructor Documentation

5.3.2.1 Gauss3D()

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

construction with default parameters

Gauss3D construction with

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

Definition at line 247 of file ICSetters.cpp.

```
                                                        {
    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 = std::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 268 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("0x12345");
    if (abs(zg) > 1e-15)
      Rz = zg * (1 + (zr * zr / zg / zg));
    else
      Rz = 1e308;
    const sunrealtype k = 2 * std::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 215 of file ICSetters.h.

Referenced by addToSpace(), and Gauss3D().

5.3.4.2 A2

`sunrealtype Gauss3D::A2 [private]`

amplitude projection on xy-plane

Definition at line 217 of file ICSetters.h.

Referenced by addToSpace(), and Gauss3D().

5.3.4.3 Amp

`sunrealtype Gauss3D::Amp [private]`

amplitude A

Definition at line 200 of file ICSetters.h.

Referenced by Gauss3D().

5.3.4.4 axis

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

normalized propagation axis

Definition at line 198 of file ICSetters.h.

Referenced by addToSpace(), and Gauss3D().

5.3.4.5 dis

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

distance maximum to origin

Definition at line 196 of file ICSetters.h.

Referenced by addToSpace(), and Gauss3D().

5.3.4.6 lambda

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

wavelength λ

Definition at line 219 of file ICSetters.h.

Referenced by addToSpace(), and Gauss3D().

5.3.4.7 Ph0

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

center of beam Φ_0

Definition at line 211 of file ICSetters.h.

Referenced by addToSpace(), and Gauss3D().

5.3.4.8 PhA

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

length of beam Φ_A

Definition at line 213 of file ICSetters.h.

Referenced by addToSpace(), and Gauss3D().

5.3.4.9 phip

```
sunrealtype Gauss3D::phip [private]
```

polarization rotation from TE-mode around propagation direction

Definition at line 203 of file ICSetters.h.

Referenced by Gauss3D().

5.3.4.10 w0

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

taille ω_0

Definition at line 207 of file ICSetters.h.

Referenced by addToSpace(), and Gauss3D().

5.3.4.11 zr

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

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

Definition at line 209 of file ICSetters.h.

Referenced by addToSpace(), and Gauss3D().

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

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

5.4 gaussian1D Struct Reference

1D Gaussian wave structure

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

Data Fields

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

5.4.1 Detailed Description

1D Gaussian wave structure

Definition at line 26 of file SimulationFunctions.h.

5.4.2 Field Documentation

5.4.2.1 k

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

wavevector (normalized to $1/\lambda$)

Definition at line 27 of file SimulationFunctions.h.

5.4.2.2 p

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

amplitude & polarization vector

Definition at line 28 of file SimulationFunctions.h.

5.4.2.3 phi

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

phase shift

Definition at line 31 of file SimulationFunctions.h.

5.4.2.4 phig

`sunrealtype gaussian1D::phig`

width

Definition at line 30 of file SimulationFunctions.h.

5.4.2.5 x0

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

shift from origin

Definition at line 29 of file SimulationFunctions.h.

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

- src/SimulationFunctions.h

5.5 gaussian2D Struct Reference

2D Gaussian wave structure

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

Data Fields

- std::array< sunrealtype, 3 > x0
- std::array< sunrealtype, 3 > axis
- sunrealtype amp
- sunrealtype phip
- sunrealtype w0
- sunrealtype zr
- sunrealtype ph0
- sunrealtype phA

5.5.1 Detailed Description

2D Gaussian wave structure

Definition at line 35 of file SimulationFunctions.h.

5.5.2 Field Documentation

5.5.2.1 amp

```
sunrealtype gaussian2D::amp
```

amplitude

Definition at line 38 of file SimulationFunctions.h.

5.5.2.2 axis

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

direction from where it comes

Definition at line 37 of file SimulationFunctions.h.

5.5.2.3 ph0

```
sunrealtype gaussian2D::ph0
```

beam center

Definition at line 42 of file SimulationFunctions.h.

5.5.2.4 phA

```
sunrealtype gaussian2D::phA
```

beam length

Definition at line 43 of file SimulationFunctions.h.

5.5.2.5 phip

```
sunrealtype gaussian2D::phip
```

polarization rotation

Definition at line 39 of file SimulationFunctions.h.

5.5.2.6 w0

```
sunrealtype gaussian2D::w0
```

taille

Definition at line 40 of file SimulationFunctions.h.

5.5.2.7 x0

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

center

Definition at line 36 of file SimulationFunctions.h.

5.5.2.8 zr

```
sunrealtype gaussian2D::zr
```

Rayleigh length

Definition at line 41 of file SimulationFunctions.h.

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

- src/SimulationFunctions.h

5.6 gaussian3D Struct Reference

3D Gaussian wave structure

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

Data Fields

- std::array< sunrealtype, 3 > x0
- std::array< sunrealtype, 3 > axis
- sunrealtype amp
- sunrealtype phip
- sunrealtype w0
- sunrealtype zr
- sunrealtype ph0
- sunrealtype phA

5.6.1 Detailed Description

3D Gaussian wave structure

Definition at line 47 of file SimulationFunctions.h.

5.6.2 Field Documentation

5.6.2.1 amp

```
sunrealtype gaussian3D::amp
```

amplitude

Definition at line 50 of file SimulationFunctions.h.

5.6.2.2 axis

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

direction from where it comes

Definition at line 49 of file SimulationFunctions.h.

5.6.2.3 ph0

```
sunrealtype gaussian3D::ph0
```

beam center

Definition at line 54 of file SimulationFunctions.h.

5.6.2.4 phA

```
sunrealtype gaussian3D::phA
```

beam length

Definition at line 55 of file SimulationFunctions.h.

5.6.2.5 phip

```
sunrealtype gaussian3D::phip
```

polarization rotation

Definition at line 51 of file SimulationFunctions.h.

5.6.2.6 w0

```
sunrealtype gaussian3D::w0
```

taille

Definition at line 52 of file SimulationFunctions.h.

5.6.2.7 x0

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

center

Definition at line 48 of file SimulationFunctions.h.

5.6.2.8 zr

```
sunrealtype gaussian3D::zr
```

Rayleigh length

Definition at line 53 of file SimulationFunctions.h.

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

- src/SimulationFunctions.h

5.7 ICSetter Class Reference

ICSetter class to initialize wave types with default parameters.

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

Public Member Functions

- void eval (sunrealtype x, sunrealtype y, sunrealtype z, sunrealtype ∗pTo6Space)

  function to set all coordinates to zero and then `add` *the field values*
- void add (sunrealtype x, sunrealtype y, sunrealtype z, sunrealtype ∗pTo6Space)

  function to fill the lattice space with initial field values
- void addPlaneWave1D (std::array< sunrealtype, 3 > k={1, 0, 0}, std::array< sunrealtype, 3 > p={0, 0, 1}, std::array< sunrealtype, 3 > phi={0, 0, 0})

  function to add plane waves in 1D to their container vector
- void addPlaneWave2D (std::array< sunrealtype, 3 > k={1, 0, 0}, std::array< sunrealtype, 3 > p={0, 0, 1}, std::array< sunrealtype, 3 > phi={0, 0, 0})

  function to add plane waves in 2D to their container vector
- void addPlaneWave3D (std::array< sunrealtype, 3 > k={1, 0, 0}, std::array< sunrealtype, 3 > p={0, 0, 1}, std::array< sunrealtype, 3 > phi={0, 0, 0})

  function to add plane waves in 3D to their container vector
- void addGauss1D (std::array< sunrealtype, 3 > k={1, 0, 0}, std::array< sunrealtype, 3 > p={0, 0, 1}, std↔ ::array< sunrealtype, 3 > xo={0, 0, 0}, sunrealtype phig_=1.0, std::array< sunrealtype, 3 > phi={0, 0, 0})

  function to add Gaussian wave packets in 1D to their container vector
- void addGauss2D (std::array< sunrealtype, 3 > dis_={0, 0, 0}, std::array< sunrealtype, 3 > axis_={1, 0, 0}, sunrealtype Amp_=1.0, sunrealtype phip_=0, sunrealtype w0_=1e-5, sunrealtype zr_=4e-5, sunrealtype Ph0_=2e-5, sunrealtype PhA_=0.45e-5)

  function to add Gaussian wave packets in 2D to their container vector
- void addGauss3D (std::array< sunrealtype, 3 > dis_={0, 0, 0}, std::array< sunrealtype, 3 > axis_={1, 0, 0}, sunrealtype Amp_=1.0, sunrealtype phip_=0, sunrealtype w0_=1e-5, sunrealtype zr_=4e-5, sunrealtype Ph0_=2e-5, sunrealtype PhA_=0.45e-5)

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

Private Attributes

- std::vector< PlaneWave1D > planeWaves1D

  container vector for plane waves in 1D
- std::vector< PlaneWave2D > planeWaves2D

  container vector for plane waves in 2D
- std::vector< PlaneWave3D > planeWaves3D

  container vector for plane waves in 3D
- std::vector< Gauss1D > gauss1Ds

  container vector for Gaussian wave packets in 1D
- std::vector< Gauss2D > gauss2Ds

  container vector for Gaussian wave packets in 2D
- std::vector< Gauss3D > gauss3Ds

  container vector for Gaussian wave packets in 3D

5.7.1 Detailed Description

ICSetter class to initialize wave types with default parameters.

Definition at line 238 of file ICSetters.h.

5.7.2 Member Function Documentation

5.7.2.1 add()

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

function to fill the lattice space with initial field values

Add all initial field values to the lattice space

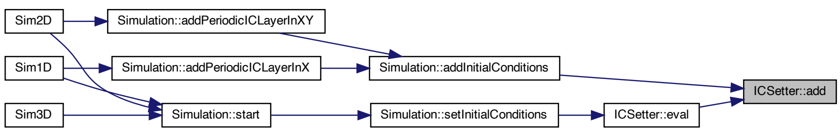
Definition at line 309 of file ICSetters.cpp.

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

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

Add Gaussian waves in 1D to their container vector

Definition at line 347 of file ICSetters.cpp.

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

```
00352 }
```

References gauss1Ds.

Referenced by Sim1D().

Here is the caller graph for this function:

5.7.2.3 addGauss2D()

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

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

Add Gaussian waves in 2D to their container vector

Definition at line 355 of file ICSetters.cpp.

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

References gauss2Ds.

Referenced by Sim2D().

Here is the caller graph for this function:

5.7.2.4 addGauss3D()

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

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

Add Gaussian waves in 3D to their container vector

Definition at line 365 of file ICSetters.cpp.

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

References gauss3Ds.

Referenced by Sim3D().

Here is the caller graph for this function:

5.7.2.5 addPlaneWave1D()

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

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

Add plane waves in 1D to their container vector

Definition at line 326 of file ICSetters.cpp.

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

References planeWaves1D.

Referenced by Sim1D().

Here is the caller graph for this function:

5.7.2.6 addPlaneWave2D()

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

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

Add plane waves in 2D to their container vector

Definition at line 333 of file ICSetters.cpp.

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

References planeWaves2D.

Referenced by Sim2D().

Here is the caller graph for this function:

5.7.2.7 addPlaneWave3D()

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

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

Add plane waves in 3D to their container vector

Definition at line 340 of file ICSetters.cpp.

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

References planeWaves3D.

Referenced by Sim3D().

Here is the caller graph for this function:

5.7.2.8 eval()

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

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

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

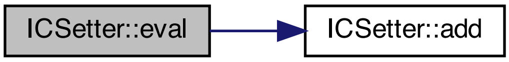
Definition at line 297 of file ICSetters.cpp.

```
                                     {
  pTo6Space[0] = 0;
  pTo6Space[1] = 0;
  pTo6Space[2] = 0;
  pTo6Space[3] = 0;
  pTo6Space[4] = 0;
  pTo6Space[5] = 0;
  add(x, y, z, pTo6Space);
}
```

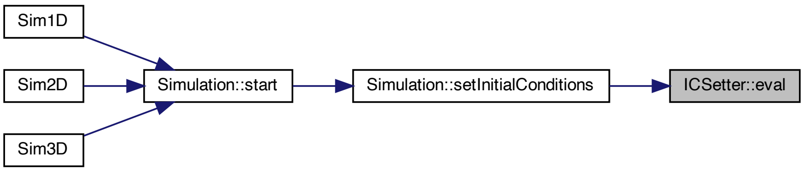
References add().

Referenced by Simulation::setInitialConditions().

Here is the call graph for this function:



Here is the caller graph for this function:



5.7.3 Field Documentation

5.7.3.1 gauss1Ds

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

container vector for Gaussian wave packets in 1D

Definition at line 247 of file ICSetters.h.

Referenced by add(), and addGauss1D().

5.7.3.2 gauss2Ds

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

container vector for Gaussian wave packets in 2D

Definition at line 249 of file ICSetters.h.

Referenced by add(), and addGauss2D().

5.7.3.3 gauss3Ds

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

container vector for Gaussian wave packets in 3D

Definition at line 251 of file ICSetters.h.

Referenced by add(), and addGauss3D().

5.7.3.4 planeWaves1D

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

container vector for plane waves in 1D

Definition at line 241 of file ICSetters.h.

Referenced by add(), and addPlaneWave1D().

5.7.3.5 planeWaves2D

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

container vector for plane waves in 2D

Definition at line 243 of file ICSetters.h.

Referenced by add(), and addPlaneWave2D().

5.7.3.6 planeWaves3D

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

container vector for plane waves in 3D

Definition at line 245 of file ICSetters.h.

Referenced by add(), and addPlaneWave3D().

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

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

5.8 Lattice Class Reference

Lattice class for the construction of the enveloping discrete simulation space.

`#include <src/LatticePatch.h>`

Public Member Functions

- void initializeCommunicator (const int Nx, const int Ny, const int Nz, const bool per)

  function to create and deploy the cartesian communicator
- Lattice (const int StO)

  default construction
- void setDiscreteDimensions (const sunindextype _nx, const sunindextype _ny, const sunindextype _nz)

  component function for resizing the discrete dimensions of the lattice
- void setPhysicalDimensions (const sunrealtype _lx, const sunrealtype _ly, const sunrealtype _lz)

  component function for resizing the physical size of the lattice

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

Data Fields

- int n_prc

  number of MPI processes
- int my_prc

  number of MPI process
- MPI_Comm comm

  personal communicator of the lattice
- SUNContext sunctx

  SUNContext object.

Private Attributes

- sunrealtype tot_lx

    physical size of the lattice in x-direction
- sunrealtype tot_ly

    physical size of the lattice in y-direction
- sunrealtype tot_lz

    physical size of the lattice in z-direction
- sunindextype tot_nx

    number of points in x-direction
- sunindextype tot_ny

    number of points in y-direction
- sunindextype tot_nz

    number of points in z-direction
- sunindextype tot_noP

    total number of lattice points
- sunindextype tot_noDP

    number of lattice points times data dimension of each point
- sunrealtype dx

    physical distance between lattice points in x-direction
- sunrealtype dy

    physical distance between lattice points in y-direction
- sunrealtype dz

    physical distance between lattice points in z-direction
- const int stencilOrder

    stencil order
- const int ghostLayerWidth

    required width of ghost layers (depends on the stencil order)
- unsigned int statusFlags

    lattice status flags

Static Private Attributes

- static constexpr int dataPointDimension = 6

    dimension of each data point set once and for all

5.8.1 Detailed Description

Lattice class for the construction of the enveloping discrete simulation space.

Definition at line 47 of file LatticePatch.h.

5.8.2 Constructor & Destructor Documentation

5.8.2.1 Lattice()

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

default construction

Construct the lattice and set the stencil order.

Definition at line 39 of file LatticePatch.cpp.

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

References statusFlags.

5.8.3 Member Function Documentation

5.8.3.1 get_dataPointDimension()

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

getter function

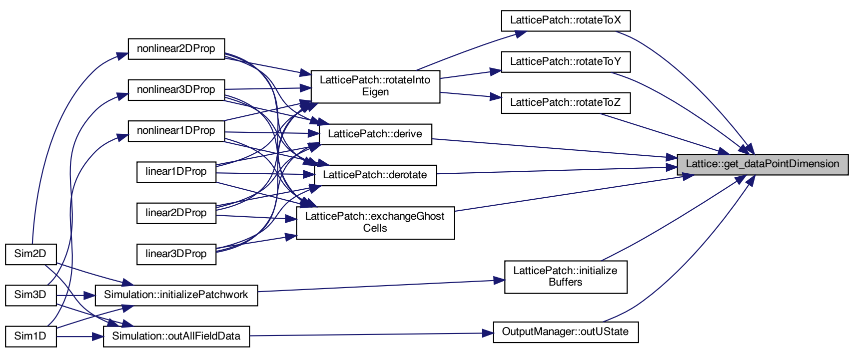
Definition at line 113 of file LatticePatch.h.

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

References dataPointDimension.

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

Here is the caller graph for this function:



5.8.3.2 get_dx()

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

getter function

Definition at line 110 of file LatticePatch.h.

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

References dx.

5.8.3.3 get_dy()

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

getter function

Definition at line 111 of file LatticePatch.h.

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

References dy.

5.8.3.4 get_dz()

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

getter function

Definition at line 112 of file LatticePatch.h.

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

References dz.

5.8.3.5 get_ghostLayerWidth()

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

getter function

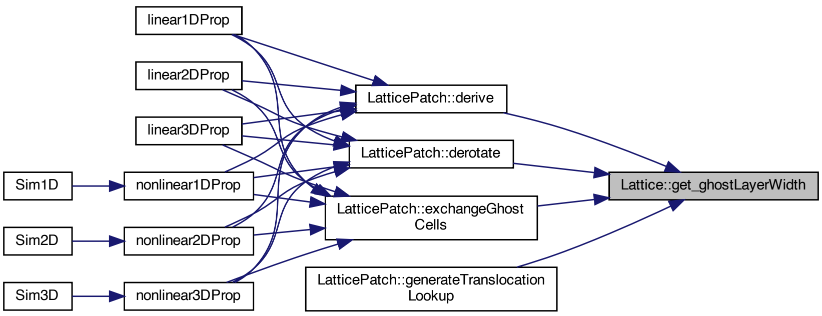
Definition at line 117 of file LatticePatch.h.

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

References ghostLayerWidth.

Referenced by LatticePatch::derive(), LatticePatch::derotate(), LatticePatch::exchangeGhostCells(), and LatticePatch::generateTranslocationLookup().

Here is the caller graph for this function:



5.8.3.6 get_stencilOrder()

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

getter function

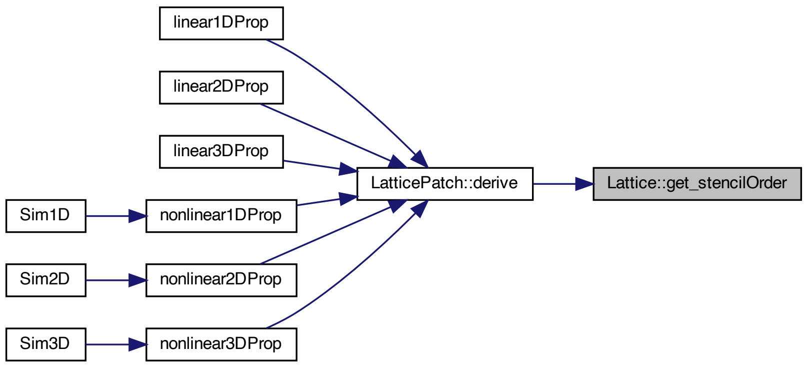
Definition at line 116 of file LatticePatch.h.

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

References stencilOrder.

Referenced by LatticePatch::derive().

Here is the caller graph for this function:



5.8.3.7 get_tot_lx()

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

getter function

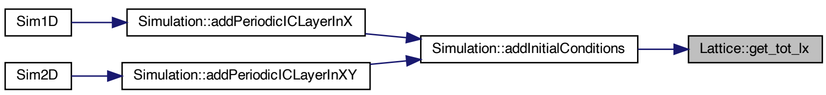
Definition at line 102 of file LatticePatch.h.

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

References tot_lx.

Referenced by Simulation::addInitialConditions().

Here is the caller graph for this function:



5.8.3.8 get_tot_ly()

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

getter function

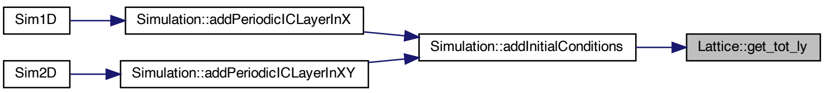
Definition at line 103 of file LatticePatch.h.

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

References tot_ly.

Referenced by Simulation::addInitialConditions().

Here is the caller graph for this function:



5.8.3.9 get_tot_lz()

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

getter function

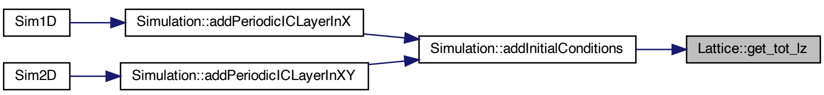
Definition at line 104 of file LatticePatch.h.

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

References tot_lz.

Referenced by Simulation::addInitialConditions().

Here is the caller graph for this function:



5.8.3.10 get_tot_noDP()

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

getter function

Definition at line 109 of file LatticePatch.h.
```
00109 { return tot_noDP; }
```

References tot_noDP.

5.8.3.11 get_tot_noP()

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

getter function

Definition at line 108 of file LatticePatch.h.
```
00108 { return tot_noP; }
```

References tot_noP.

5.8.3.12 get_tot_nx()

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

getter function

Definition at line 105 of file LatticePatch.h.
```
00105 { return tot_nx; }
```

References tot_nx.

5.8.3.13 get_tot_ny()

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

getter function

Definition at line 106 of file LatticePatch.h.
```
00106 { return tot_ny; }
```

References tot_ny.

5.8.3.14 get_tot_nz()

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

getter function

Definition at line 107 of file LatticePatch.h.

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

References tot_nz.

5.8.3.15 initializeCommunicator()

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

function to create and deploy the cartesian communicator

Initialize the cartesian communicator.

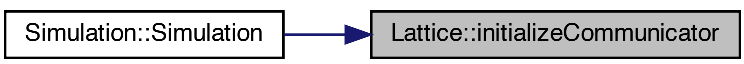
Definition at line 15 of file LatticePatch.cpp.

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

References comm, my_prc, and n_prc.

Referenced by Simulation::Simulation().

Here is the caller graph for this function:



5.8.3.16 setDiscreteDimensions()

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

component function for resizing the discrete dimensions of the lattice

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

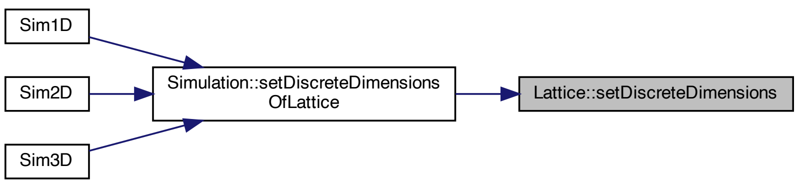
Definition at line 45 of file LatticePatch.cpp.

```
                                                    {
  // 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.

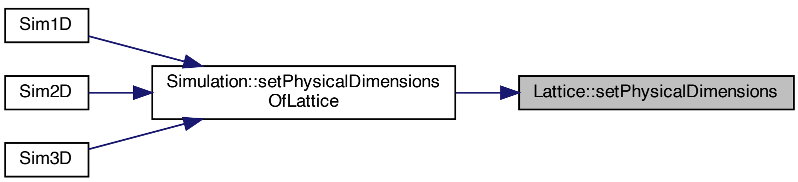
```
                                                  {
  tot_lx = _lx;
  tot_ly = _ly;
```

```
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 86 of file LatticePatch.h.

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

5.8.4.2 dataPointDimension

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

dimension of each data point set once and for all

Definition at line 64 of file LatticePatch.h.

Referenced by get_dataPointDimension(), and setDiscreteDimensions().

5.8.4.3 dx

`sunrealtype Lattice::dx [private]`

physical distance between lattice points in x-direction

Definition at line 68 of file LatticePatch.h.

Referenced by get_dx(), setDiscreteDimensions(), and setPhysicalDimensions().

5.8.4.4 dy

`sunrealtype Lattice::dy [private]`

physical distance between lattice points in y-direction

Definition at line 70 of file LatticePatch.h.

Referenced by get_dy(), setDiscreteDimensions(), and setPhysicalDimensions().

5.8.4.5 dz

`sunrealtype Lattice::dz [private]`

physical distance between lattice points in z-direction

Definition at line 72 of file LatticePatch.h.

Referenced by get_dz(), setDiscreteDimensions(), and setPhysicalDimensions().

5.8.4.6 ghostLayerWidth

`const int Lattice::ghostLayerWidth [private]`

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

Definition at line 76 of file LatticePatch.h.

Referenced by get_ghostLayerWidth().

5.8.4.7 my_prc

`int Lattice::my_prc`

number of MPI process

Definition at line 84 of file LatticePatch.h.

Referenced by Simulation::advanceToTime(), initializeCommunicator(), Simulation::initializeCVODEobject(), OutputManager::outUState(), and Simulation::Simulation().

5.8.4.8 n_prc

`int Lattice::n_prc`

number of MPI processes

Definition at line 82 of file LatticePatch.h.

Referenced by initializeCommunicator().

5.8.4.9 statusFlags

`unsigned int Lattice::statusFlags [private]`

lattice status flags

Definition at line 78 of file LatticePatch.h.

Referenced by Lattice(), and setPhysicalDimensions().

5.8.4.10 stencilOrder

`const int Lattice::stencilOrder [private]`

stencil order

Definition at line 74 of file LatticePatch.h.

Referenced by get_stencilOrder().

5.8.4.11 sunctx

`SUNContext Lattice::sunctx`

SUNContext object.

Definition at line 93 of file LatticePatch.h.

Referenced by Simulation::initializeCVODEobject(), Simulation::Simulation(), and Simulation::∼Simulation().

5.8.4.12 tot_lx

`sunrealtype Lattice::tot_lx [private]`

physical size of the lattice in x-direction

Definition at line 50 of file LatticePatch.h.

Referenced by get_tot_lx(), setDiscreteDimensions(), and setPhysicalDimensions().

5.8.4.13 tot_ly

`sunrealtype Lattice::tot_ly [private]`

physical size of the lattice in y-direction

Definition at line 52 of file LatticePatch.h.

Referenced by get_tot_ly(), setDiscreteDimensions(), and setPhysicalDimensions().

5.8.4.14 tot_lz

`sunrealtype Lattice::tot_lz [private]`

physical size of the lattice in z-direction

Definition at line 54 of file LatticePatch.h.

Referenced by get_tot_lz(), setDiscreteDimensions(), and setPhysicalDimensions().

5.8.4.15 tot_noDP

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

number of lattice points times data dimension of each point

Definition at line 66 of file LatticePatch.h.

Referenced by get_tot_noDP(), and setDiscreteDimensions().

5.8.4.16 tot_noP

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

total number of lattice points

Definition at line 62 of file LatticePatch.h.

Referenced by get_tot_noP(), and setDiscreteDimensions().

5.8.4.17 tot_nx

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

number of points in x-direction

Definition at line 56 of file LatticePatch.h.

Referenced by get_tot_nx(), setDiscreteDimensions(), and setPhysicalDimensions().

5.8.4.18 tot_ny

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

number of points in y-direction

Definition at line 58 of file LatticePatch.h.

Referenced by get_tot_ny(), setDiscreteDimensions(), and setPhysicalDimensions().

5.8.4.19 tot_nz

`sunindextype Lattice::tot_nz [private]`

number of points in z-direction

Definition at line 60 of file LatticePatch.h.

Referenced by get_tot_nz(), setDiscreteDimensions(), and setPhysicalDimensions().

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

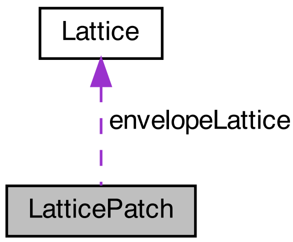
- src/LatticePatch.h
- src/LatticePatch.cpp

5.9 LatticePatch Class Reference

LatticePatch class for the construction of the patches in the enveloping lattice.

`#include <src/LatticePatch.h>`

Collaboration diagram for LatticePatch:



Public Member Functions

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

function to fill out the lookup tables for cache efficiency

- void rotateIntoEigen (const int dir)

  function to rotate u into Z-matrix eigenraum
- void derotate (int dir, sunrealtype ∗buffOut)

  function to derotate uAux into dudata lattice direction of x
- void initializeBuffers ()

  initialize buffers to save derivatives
- void exchangeGhostCells (const int dir)

  function to exchange ghost cells
- void derive (const int dir)

  function to derive the centered values in uAux and save them noncentered
- void checkFlag (unsigned int flag) const

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

Data Fields

- int ID

  ID of the LatticePatch, corresponds to process number (for debugging)
- N_Vector u

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

  N_Vector for saving temporal derivatives of the field data.
- sunrealtype ∗ uData

  pointer to field data
- sunrealtype ∗ uAuxData

  pointer to auxiliary data vector
- sunrealtype ∗ duData

  pointer to time-derivative data
- std::array< sunrealtype ∗, 3 > buffData

- sunrealtype ∗ gCLData
- sunrealtype ∗ gCRData

Private Member Functions

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

Private Attributes

- sunrealtype x0

  origin of the patch in physical space; x-coordinate
- sunrealtype y0

  origin of the patch in physical space; y-coordinate
- sunrealtype z0

  origin of the patch in physical space; z-coordinate
- sunindextype 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
- unsigned int statusFlags

  lattice patch status flags
- const Lattice ∗ envelopeLattice

  pointer to the enveloping lattice
- std::vector< sunrealtype > uAux

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

- std::vector< sunindextype > uTox
- std::vector< sunindextype > uToy
- std::vector< sunindextype > uToz
- std::vector< sunindextype > xTou
- std::vector< sunindextype > yTou
- std::vector< sunindextype > zTou

- std::vector< sunrealtype > buffX

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

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

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

Friends

- int generatePatchwork (const Lattice &envelopeLattice, LatticePatch &patchToMold, const int DLx, const int DLy, const int DLz)

  friend function for creating the patchwork slicing of the overall lattice

5.9.1 Detailed Description

LatticePatch class for the construction of the patches in the enveloping lattice.

Definition at line 125 of file LatticePatch.h.

5.9.2 Constructor & Destructor Documentation

5.9.2.1 LatticePatch()

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

constructor setting up a default first lattice patch

Construct the lattice patch.

Definition at line 78 of file LatticePatch.cpp.

```
00078                               {
00079   // set default origin coordinates to (0,0,0)
00080   x0 = y0 = z0 = 0;
00081   // set default position in Lattice-Patchwork to (0,0,0)
00082   LIx = LIy = LIz = 0;
00083   // set default physical 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 checkFlag()

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

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

Check if all flags are set.

Definition at line 615 of file LatticePatch.cpp.

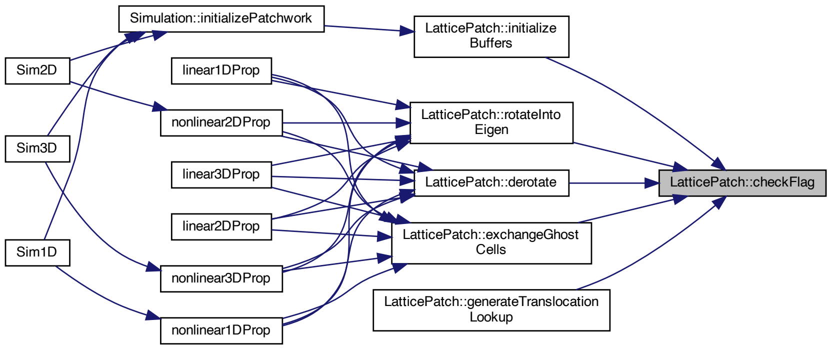
```
00615                                                  {
00616   if (!(statusFlags & flag)) {
00617     std::string errorMessage;
00618     switch (flag) {
00619     case FLatticePatchSetUp:
00620       errorMessage = "The Lattice patch was not set up please make sure to "
00621                 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

References BuffersInitialized, errorKill(), FLatticePatchSetUp, GhostLayersInitialized, statusFlags, and TranslocationLookupSetUp.

Referenced by derotate(), exchangeGhostCells(), generateTranslocationLookup(), initializeBuffers(), and rotateIntoEigen().

Here is the call graph for this function:

Here is the caller graph for this function:



5.9.3.2 derive()

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

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

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

Definition at line 646 of file LatticePatch.cpp.

```
00646                                           {
00647   // ghost layer width adjusted to the chosen stencil order
00648   const int gLW = envelopeLattice->get_ghostLayerWidth();
00649   // dimensionality of data points -> 6
00650   const int dPD = envelopeLattice->get_dataPointDimension();
00651   // total width of patch in given direction including ghost layers at ends
00652   const sunindextype dirWidth = discreteSize(dir) + 2 * gLW;
00653   // width of patch only in given direction
00654   const sunindextype dirWidthO = discreteSize(dir);
00655   // size of plane perpendicular to given dimension
00656   const sunindextype perpPlainSize = discreteSize() / discreteSize(dir);
00657   // physical distance between points in that direction
00658   sunrealtype dxi = nan("0x12345");
00659   switch (dir) {
00660   case 1:
00661     dxi = dx;
00662     break;
00663   case 2:
00664     dxi = dy;
00665     break;
00666   case 3:
00667     dxi = dz;
00668     break;
00669   default:
00670     dxi = 1;
00671     errorKill("Tried to derive in the wrong direction");
00672     break;
00673   }
00674   // Derive according to chosen stencil accuracy order
00675   const int order = envelopeLattice->get_stencilOrder();
00676   switch (order) {
00677   case 1:  // gLW=1
00678     #pragma omp parallel for default(none) \
00679     shared(perpPlainSize, dxi, dirWidth, dirWidthO, gLW, dPD, uAux)
00680     for (sunindextype i = 0; i < perpPlainSize; i++) {
00681       #pragma omp simd
00682       for (sunindextype j = (i * dirWidth + gLW) * dPD;
00683            j < (i * dirWidth + gLW + dirWidthO) * dPD; j += dPD) {
00684         uAux[j + 0 - gLW * dPD] = s1b(&uAux[j + 0]) / dxi;
```

```
          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:  // gLW=2
      #pragma omp parallel for default(none) \
      shared(perpPlainSize, dxi, dirWidth, dirWidthO, gLW, dPD, uAux)
      for (sunindextype i = 0; i < perpPlainSize; i++) {
        #pragma omp simd
        for (sunindextype 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:  // gLW=2
      #pragma omp parallel for default(none) \
      shared(perpPlainSize, dxi, dirWidth, dirWidthO, gLW, dPD, uAux)
      for (sunindextype i = 0; i < perpPlainSize; i++) {
        #pragma omp simd
        for (sunindextype 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:  // gLW=3
      #pragma omp parallel for default(none) \
      shared(perpPlainSize, dxi, dirWidth, dirWidthO, gLW, dPD, uAux)
      for (sunindextype i = 0; i < perpPlainSize; i++) {
        #pragma omp simd
        for (sunindextype 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:  // gLW=3
      #pragma omp parallel for default(none) \
      shared(perpPlainSize, dxi, dirWidth, dirWidthO, gLW, dPD, uAux)
      for (sunindextype i = 0; i < perpPlainSize; i++) {
        #pragma omp simd
        for (sunindextype 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:  // gLW=4
      #pragma omp parallel for default(none) \
      shared(perpPlainSize, dxi, dirWidth, dirWidthO, gLW, dPD, uAux)
      for (sunindextype i = 0; i < perpPlainSize; i++) {
        #pragma omp simd
        for (sunindextype 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;
        }
      }
```

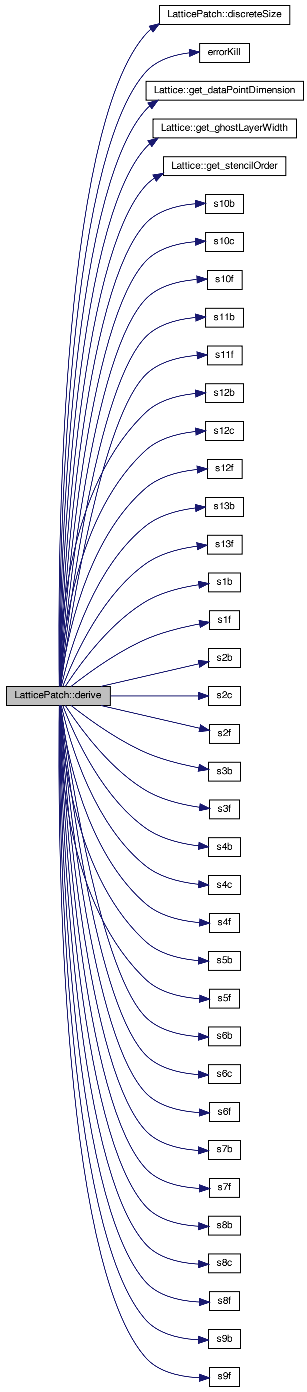
```
    break;
  case 7:  // gLW=4
    #pragma omp parallel for default(none) \
    shared(perpPlainSize, dxi, dirWidth, dirWidthO, gLW, dPD, uAux)
    for (sunindextype i = 0; i < perpPlainSize; i++) {
      #pragma omp simd
      for (sunindextype j = (i * dirWidth + gLW) * dPD;
           j < (i * 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:  // gLW=5
    #pragma omp parallel for default(none) \
    shared(perpPlainSize, dxi, dirWidth, dirWidthO, gLW, dPD, uAux)
    for (sunindextype i = 0; i < perpPlainSize; i++) {
      #pragma omp simd
      for (sunindextype 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:  // gLW=5
    #pragma omp parallel for default(none) \
    shared(perpPlainSize, dxi, dirWidth, dirWidthO, gLW, dPD, uAux)
    for (sunindextype i = 0; i < perpPlainSize; i++) {
      #pragma omp simd
      for (sunindextype 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:  // gLW=6
    #pragma omp parallel for default(none) \
    shared(perpPlainSize, dxi, dirWidth, dirWidthO, gLW, dPD, uAux)
    for (sunindextype i = 0; i < perpPlainSize; i++) {
      #pragma omp simd
      for (sunindextype 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:  // gLW=6
    #pragma omp parallel for default(none) \
    shared(perpPlainSize, dxi, dirWidth, dirWidthO, gLW, dPD, uAux)
    for (sunindextype i = 0; i < perpPlainSize; i++) {
      #pragma omp simd
      for (sunindextype 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:  // gLW=7
    #pragma omp parallel for default(none) \
    shared(perpPlainSize, dxi, dirWidth, dirWidthO, gLW, dPD, uAux)
    for (sunindextype i = 0; i < perpPlainSize; i++) {
      #pragma omp simd
      for (sunindextype 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:  // gLW=7
      // For all points in the plane perpendicular to the given direction
      #pragma omp parallel for default(none) \
      shared(perpPlainSize, dxi, dirWidth, dirWidthO, gLW, dPD, uAux)
      for (sunindextype i = 0; i < perpPlainSize; i++) {
        // iterate through the derivation direction
        #pragma omp simd
        for (sunindextype 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
          // and update position by ghost width shift
          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;
    }
}
```

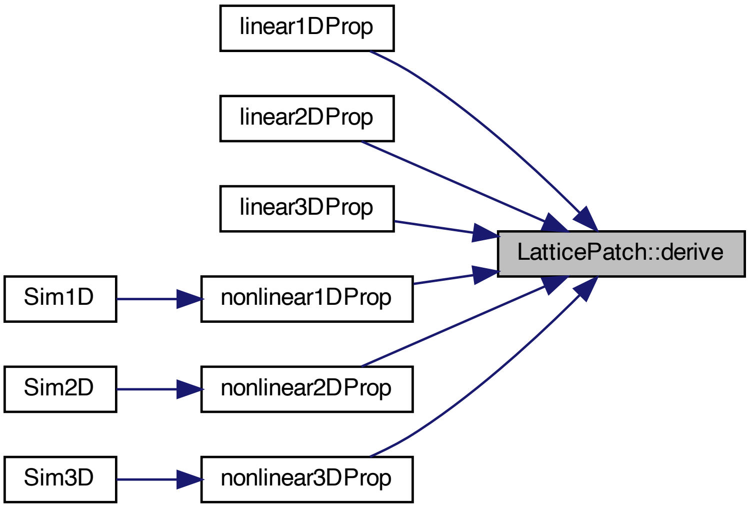
References discreteSize(), dx, dy, dz, envelopeLattice, errorKill(), Lattice::get_dataPointDimension(), Lattice::get_ghostLayerWidth(), Lattice::get_stencilOrder(), s10b(), s10c(), s10f(), s11b(), s11f(), s12b(), s12c(), s12f(), s13b(), s13f(), s1b(), s1f(), s2b(), s2c(), s2f(), s3b(), s3f(), s4b(),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



5.9.3.3 derotate()

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

function to derotate uAux into dudata lattice direction of x

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

Definition at line 428 of file LatticePatch.cpp.

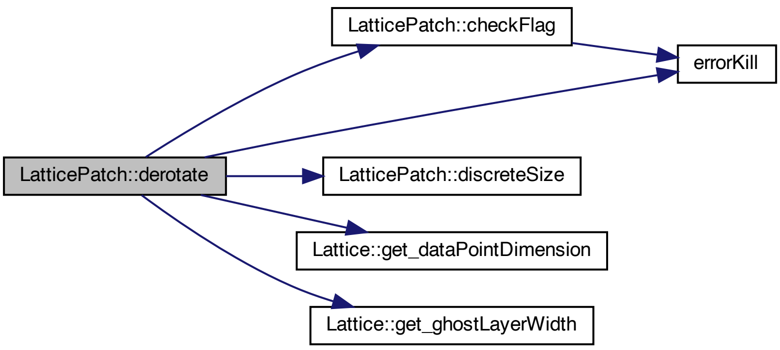
```
00428                                                              {
00429   // Check that the lattice as well as the translocation lookups have been set
00430   // up;
00431   checkFlag(FLatticePatchSetUp);
00432   checkFlag(TranslocationLookupSetUp);
00433   const int dPD = envelopeLattice->get_dataPointDimension();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

```
      buffOut[target + 5] = (uAux[ii] + uAux[2 + ii]) / 2.;
    }
    break;
  case 2:
    #pragma omp parallel for simd \
    private(ii, target) \
    shared(dPD, gLW, totalNP, uTox, uAux, buffOut) \
    schedule(static)
    for (sunindextype i = 0; i < totalNP; 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 parallel for simd \
    private(ii, target) \
    shared(dPD, gLW, totalNP, uTox, uAux, buffOut) \
    schedule(static)
    for (sunindextype i = 0; i < totalNP; 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;
  }
}
```

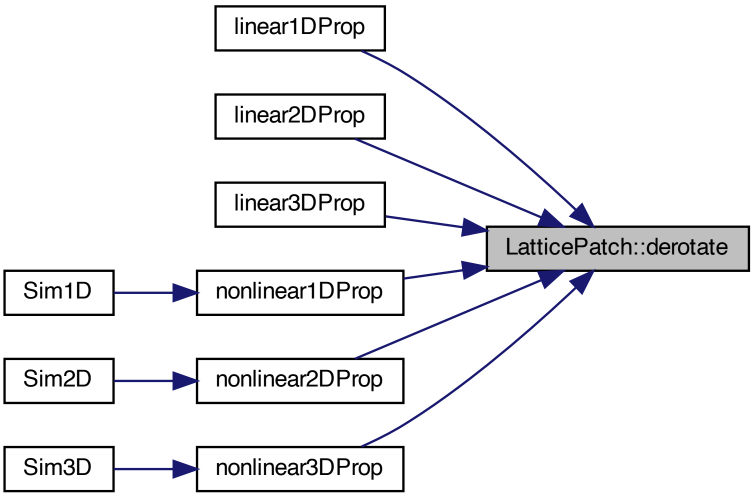
References checkFlag(), discreteSize(), envelopeLattice, errorKill(), FLatticePatchSetUp, Lattice::get_dataPointDimension(), Lattice::get_ghostLayerWidth(), TranslocationLookupSetUp, uAux, uTox, uToy, and uToz.

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:



5.9.3.4 discreteSize()

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

function to get the discrete size of the LatticePatch

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

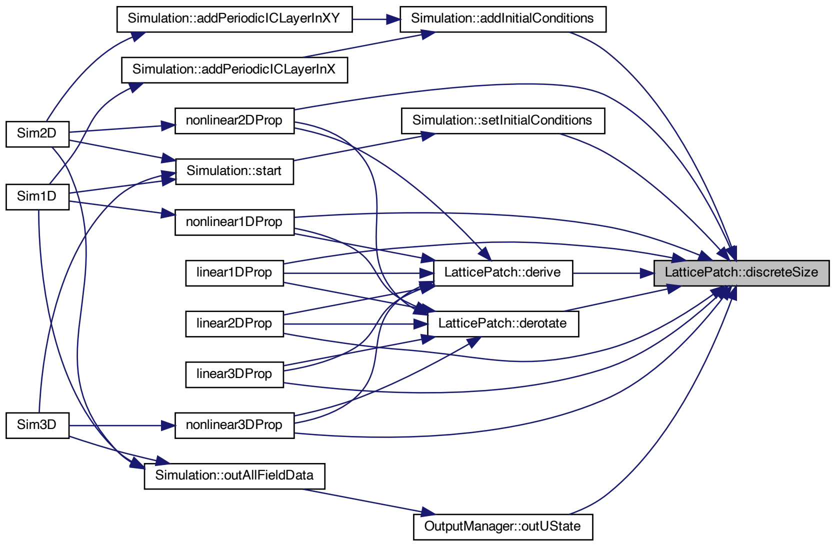
Definition at line 185 of file LatticePatch.cpp.

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

References nx, ny, and nz.

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

Here is the caller graph for this function:



5.9.3.5 exchangeGhostCells()

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

function to exchange ghost cells

Perform the ghost cell exchange in a specified direction.

Definition at line 509 of file LatticePatch.cpp.

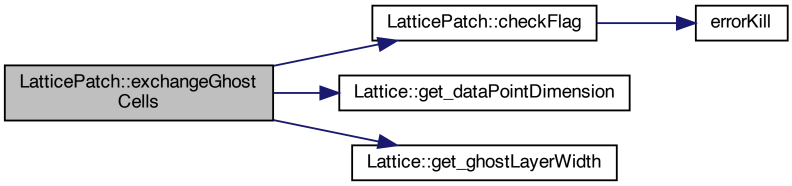
```
00509                                                       {
00510   // Check that the lattice has been set up
00511   checkFlag(FLatticeDimensionSet);
00512   checkFlag(FLatticePatchSetUp);
00513   // Variables to per dimension calculate the halo indices, and distance to
00514   // other side halo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

```
    // total number of exchanged points
    const int dPD = envelopeLattice->get_dataPointDimension();
    const sunindextype 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
    sunindextype li = 0, ui = 0;
    // Fill the halo buffers
    #pragma omp parallel for default(none) \
    private(ui) firstprivate(li) \
    shared(nx, ny, mx, my, mz, dPD, distToRight, uData, \
            ghostCellLeftToSend, ghostCellRightToSend)
    for (sunindextype iz = 0; iz < mz; iz++) {
      for (sunindextype 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)
        std::copy(&uData[ui], &uData[ui + mx * dPD], &ghostCellLeftToSend[li]);
        ui += distToRight * dPD;
        std::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 Irecv/Isend

    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);
    */
}
```

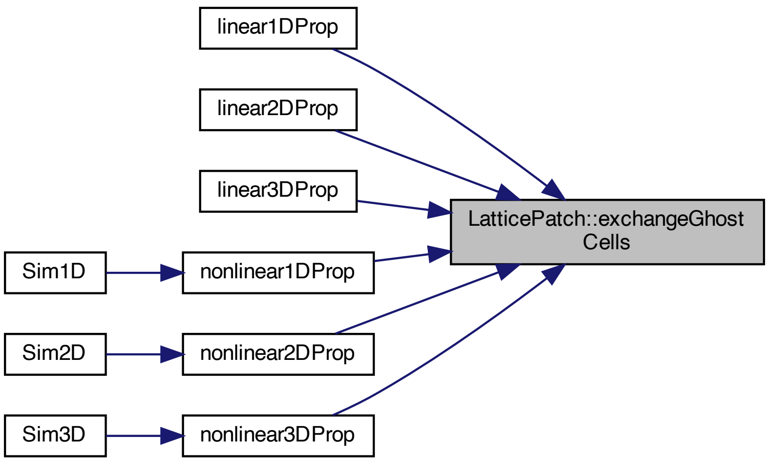
References checkFlag(), Lattice::comm, envelopeLattice, FLatticeDimensionSet, FLatticePatchSetUp, gCLData, gCRData, Lattice::get_dataPointDimension(), Lattice::get_ghostLayerWidth(), ghostCellLeft, ghostCellLeftToSend, ghostCellRight, ghostCellRightToSend, GhostLayersInitialized, nx, ny, nz, statusFlags,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



5.9.3.6 generateTranslocationLookup()

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

function to fill out the lookup tables for cache efficiency

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

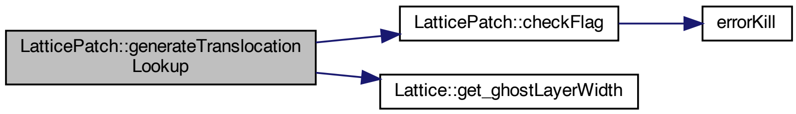
Definition at line 235 of file LatticePatch.cpp.

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

```
  // sizes for lookup vectors
  const sunindextype totalNP = nx * ny * nz;
  const sunindextype haloXSize = mx * ny * nz;
  const sunindextype haloYSize = nx * my * nz;
  const sunindextype haloZSize = nx * ny * mz;
  // generate u->uAux
  uTox.resize(totalNP);
  uToy.resize(totalNP);
  uToz.resize(totalNP);
  // generate uAux->u with length including halo
  xTou.resize(haloXSize);
  yTou.resize(haloYSize);
  zTou.resize(haloZSize);
  // same for ghost layer lookup tables
  const sunindextype ghostXSize = gLW * ny * nz;
  const sunindextype ghostYSize = gLW * nx * nz;
  const sunindextype ghostZSize = gLW * nx * ny;
  lgcTox.resize(ghostXSize);
  rgcTox.resize(ghostXSize);
  lgcToy.resize(ghostYSize);
  rgcToy.resize(ghostYSize);
  lgcToz.resize(ghostZSize);
  rgcToz.resize(ghostZSize);
  // variables for cartesian position in the 3D discrete lattice
  sunindextype px = 0, py = 0, pz = 0;
  // Fill the lookup tables
  #pragma omp parallel default(none) \
  private(px, py, pz) \
  shared(uTox, uToy, uToz, xTou, yTou, zTou, \
          nx, ny, mx, my, mz, gLW, totalNP, \
          lgcTox, rgcTox, lgcToy, rgcToy, lgcToz, rgcToz, \
          ghostXSize, ghostYSize, ghostZSize)
  {
  #pragma omp for simd schedule(static)
  for (sunindextype i = 0; i < totalNP; i++) { // loop over 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;
  }
  #pragma omp for simd schedule(static)
  for (sunindextype i = 0; i < ghostXSize; 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;
  }
  #pragma omp for simd schedule(static)
  for (sunindextype i = 0; i < ghostYSize; 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;
  }
  #pragma omp for simd schedule(static)
  for (sunindextype i = 0; i < ghostZSize; 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;
}
```

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

Here is the call graph for this function:



5.9.3.7 getDelta()

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

function to get distance between points

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

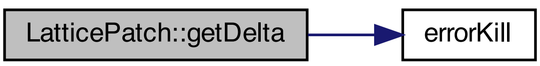
Definition at line 217 of file LatticePatch.cpp.

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

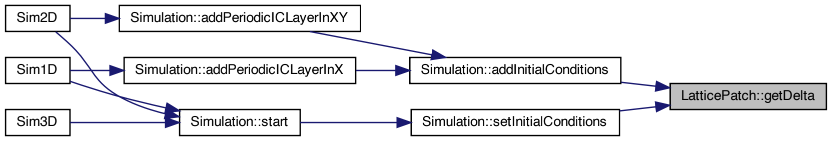
References dx, dy, dz, and errorKill().

Referenced by Simulation::addInitialConditions(), and Simulation::setInitialConditions().

Here is the call graph for this function:



Here is the caller graph for this function:



5.9.3.8 initializeBuffers()

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

initialize buffers to save derivatives

Create buffers to save derivative values, optimizing computational load.

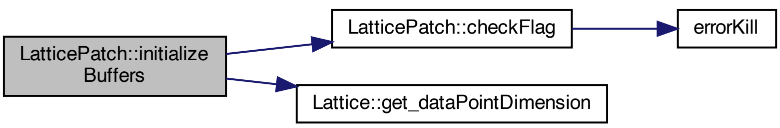
Definition at line 494 of file LatticePatch.cpp.

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

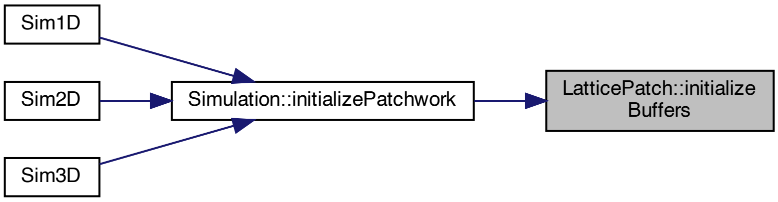
References buffData, BuffersInitialized, buffX, buffY, buffZ, checkFlag(), envelopeLattice, FLatticeDimensionSet, Lattice::get_dataPointDimension(), nx, ny, nz, and statusFlags.

Referenced by Simulation::initializePatchwork().

Here is the call graph for this function:



Here is the caller graph for this function:



5.9.3.9 origin()

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

function to get the origin of the patch

Return the physical origin of the patch in a dimension.

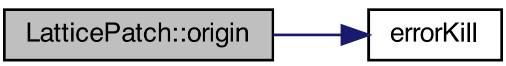
Definition at line 202 of file LatticePatch.cpp.

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

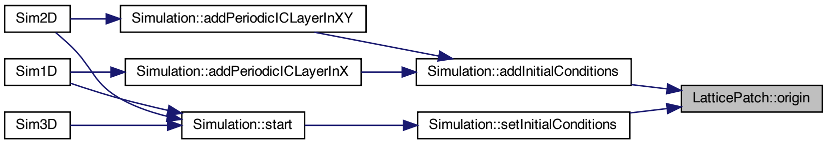
References errorKill(), x0, y0, and z0.

Referenced by Simulation::addInitialConditions(), and Simulation::setInitialConditions().

Here is the call graph for this function:



Here is the caller graph for this function:



5.9.3.10 rotateIntoEigen()

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

function to rotate u into Z-matrix eigenraum

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

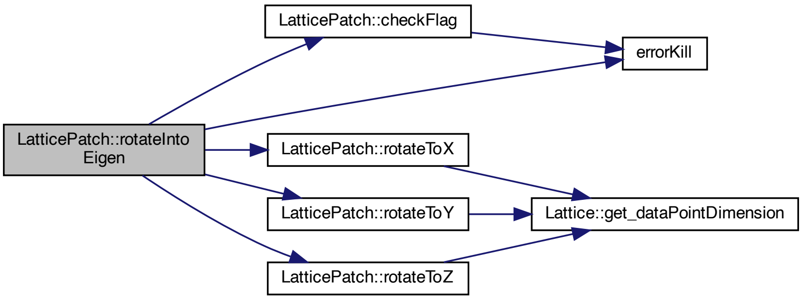
Definition at line 323 of file LatticePatch.cpp.

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

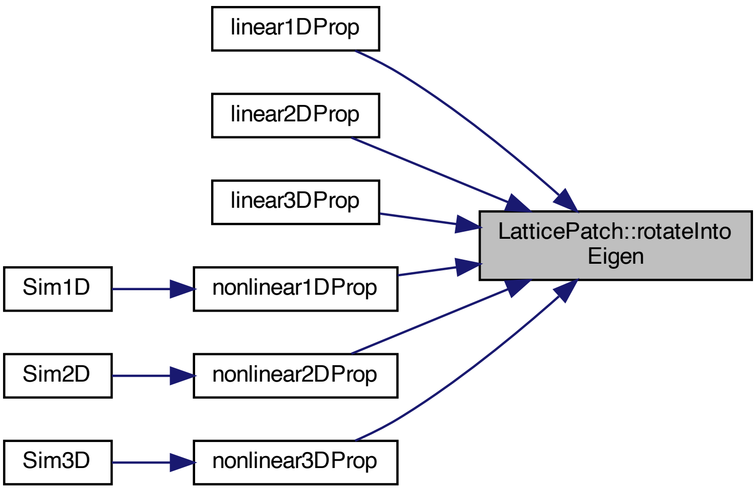
References checkFlag(), errorKill(), FLatticePatchSetUp, gCLData, gCRData, GhostLayersInitialized, lgcTox, lgcToy, lgcToz, rgcTox, rgcToy, rgcToz, rotateToX(), rotateToY(), rotateToZ(), TranslocationLookupSetUp, uAuxData, uData, uTox, uToy, and uToz.

Referenced by linear1DProp(), linear2DProp(), linear3DProp(), nonlinear1DProp(), nonlinear2DProp(), and nonlinear3DProp().

Here is the call graph for this function:



Here is the caller graph for this function:



5.9.3.11 rotateToX()

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

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

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

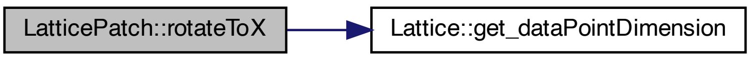
Definition at line 354 of file LatticePatch.cpp.

```
{
  sunindextype ii = 0, target = 0;
  const sunindextype size = lookup.size();
  const int dPD = envelopeLattice->get_dataPointDimension();
  #pragma omp parallel for simd \
  private(target, ii) \
  shared(lookup, outArray, inArray, size, dPD) \
  schedule(static)
  for (sunindextype i = 0; i < size; i++) {
    // get correct u-vector and spatial indices along previously defined lookup
    // tables
    target = dPD * lookup[i];
    ii = dPD * 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];
  }
}
```

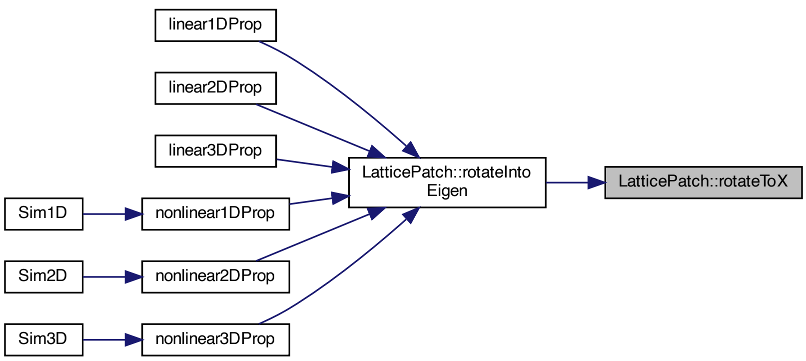
References envelopeLattice, and Lattice::get_dataPointDimension().

Referenced by rotateIntoEigen().

Here is the call graph for this function:



Here is the caller graph for this function:



5.9.3.12 rotateToY()

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

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

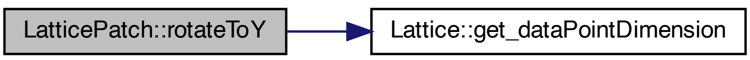
Definition at line 380 of file LatticePatch.cpp.

```
                                                                        {
  sunindextype ii = 0, target = 0;
  const int dPD = envelopeLattice->get_dataPointDimension();
  const sunindextype size = lookup.size();
  #pragma omp parallel for simd \
  private(target, ii) \
  shared(lookup, outArray, inArray, size, dPD) \
  schedule(static)
  for (sunindextype i = 0; i < size; i++) {
    target = dPD * lookup[i];
    ii = dPD * 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];
  }
}
```

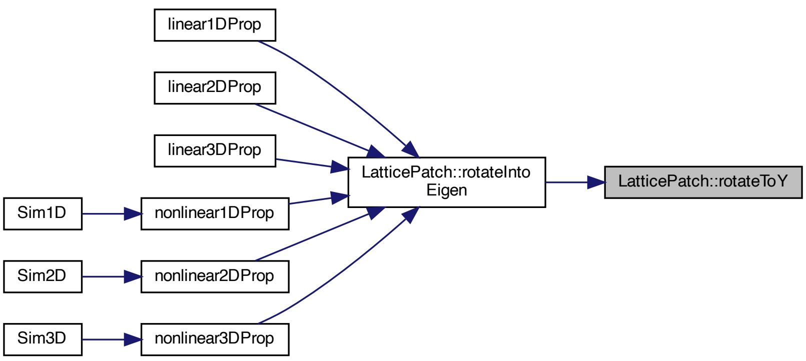
References envelopeLattice, and Lattice::get_dataPointDimension().

Referenced by rotateIntoEigen().

Here is the call graph for this function:



Here is the caller graph for this function:



5.9.3.13 rotateToZ()

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

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

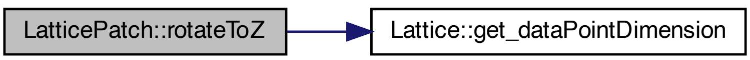
Definition at line 404 of file LatticePatch.cpp.

```
                                                                        {
  sunindextype ii = 0, target = 0;
  const sunindextype size = lookup.size();
  const int dPD = envelopeLattice->get_dataPointDimension();
  #pragma omp parallel for simd \
  private(target, ii) \
  shared(lookup, outArray, inArray, size, dPD) \
  schedule(static)
  for (sunindextype i = 0; i < size; i++) {
    target = dPD * lookup[i];
    ii = dPD * 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];
  }
}
```

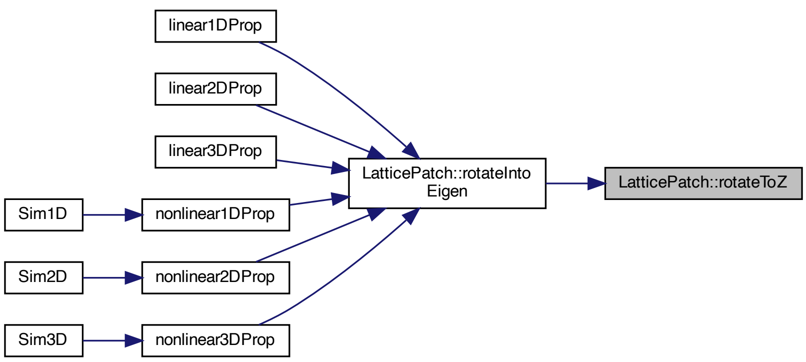
References envelopeLattice, and Lattice::get_dataPointDimension().

Referenced by rotateIntoEigen().

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();
    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
    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;
    /* 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 sunindextype s1 = patchToMold.nx, s2 = patchToMold.ny,
          s3 = patchToMold.nz;
```

```
const sunindextype s_min = std::min(s1, std::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

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

pointer to spatial derivative data buffers

Definition at line 208 of file LatticePatch.h.

Referenced by initializeBuffers(), linear1DProp(), linear2DProp(), linear3DProp(), nonlinear1DProp(), nonlinear2DProp(), and nonlinear3DProp().

5.9.5.2 buffX

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

buffer to save spatial derivative values

Definition at line 169 of file LatticePatch.h.

Referenced by initializeBuffers().

5.9.5.3 buffY

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

buffer to save spatial derivative values

Definition at line 169 of file LatticePatch.h.

Referenced by initializeBuffers().

5.9.5.4 buffZ

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

buffer to save spatial derivative values

Definition at line 169 of file LatticePatch.h.

Referenced by initializeBuffers().

5.9.5.5 du

```
N_Vector LatticePatch::du
```

N_Vector for saving temporal derivatives of the field data.

Definition at line 196 of file LatticePatch.h.

Referenced by ∼LatticePatch().

5.9.5.6 duData

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

pointer to time-derivative data

Definition at line 202 of file LatticePatch.h.

Referenced by TimeEvolution::f(), linear1DProp(), linear2DProp(), and linear3DProp().

5.9.5.7 dx

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

physical distance between lattice points in x-direction

Definition at line 152 of file LatticePatch.h.

Referenced by derive(), and getDelta().

5.9.5.8 dy

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

physical distance between lattice points in y-direction

Definition at line 154 of file LatticePatch.h.

Referenced by derive(), and getDelta().

5.9.5.9 dz

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

physical distance between lattice points in z-direction

Definition at line 156 of file LatticePatch.h.

Referenced by derive(), and getDelta().

5.9.5.10 envelopeLattice

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

pointer to the enveloping lattice

Definition at line 160 of file LatticePatch.h.

Referenced by derive(), derotate(), exchangeGhostCells(), generateTranslocationLookup(), initializeBuffers(), rotateToX(), rotateToY(), and rotateToZ().

5.9.5.11 gCLData

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

pointer to halo data

Definition at line 205 of file LatticePatch.h.

Referenced by exchangeGhostCells(), and rotateIntoEigen().

5.9.5.12 gCRData

`sunrealtype * LatticePatch::gCRData`

pointer to halo data

Definition at line 205 of file LatticePatch.h.

Referenced by exchangeGhostCells(), and rotateIntoEigen().

5.9.5.13 ghostCellLeft

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

buffer for passing ghost cell data

Definition at line 173 of file LatticePatch.h.

Referenced by exchangeGhostCells().

5.9.5.14 ghostCellLeftToSend

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

buffer for passing ghost cell data

Definition at line 173 of file LatticePatch.h.

Referenced by exchangeGhostCells().

5.9.5.15 ghostCellRight

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

buffer for passing ghost cell data

Definition at line 173 of file LatticePatch.h.

Referenced by exchangeGhostCells().

5.9.5.16 ghostCellRightToSend

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

buffer for passing ghost cell data

Definition at line 174 of file LatticePatch.h.

Referenced by exchangeGhostCells().

5.9.5.17 ghostCells

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

buffer for passing ghost cell data

Definition at line 174 of file LatticePatch.h.

5.9.5.18 ghostCellsToSend

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

buffer for passing ghost cell data

Definition at line 174 of file LatticePatch.h.

5.9.5.19 ID

```
int LatticePatch::ID
```

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

Definition at line 192 of file LatticePatch.h.

5.9.5.20 lgcTox

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

ghost cell translocation lookup table

Definition at line 178 of file LatticePatch.h.

Referenced by generateTranslocationLookup(), and rotateIntoEigen().

5.9.5.21 lgcToy

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

ghost cell translocation lookup table

Definition at line 178 of file LatticePatch.h.

Referenced by generateTranslocationLookup(), and rotateIntoEigen().

5.9.5.22 lgcToz

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

ghost cell translocation lookup table

Definition at line 178 of file LatticePatch.h.

Referenced by generateTranslocationLookup(), and rotateIntoEigen().

5.9.5.23 LIx

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

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

Definition at line 134 of file LatticePatch.h.

Referenced by LatticePatch().

5.9.5.24 LIy

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

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

Definition at line 136 of file LatticePatch.h.

Referenced by LatticePatch().

5.9.5.25 LIz

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

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

Definition at line 138 of file LatticePatch.h.

Referenced by LatticePatch().

5.9.5.26 lx

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

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

Definition at line 140 of file LatticePatch.h.

Referenced by LatticePatch().

5.9.5.27 ly

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

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

Definition at line 142 of file LatticePatch.h.

Referenced by LatticePatch().

5.9.5.28 lz

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

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

Definition at line 144 of file LatticePatch.h.

Referenced by LatticePatch().

5.9.5.29 nx

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

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

Definition at line 146 of file LatticePatch.h.

Referenced by discreteSize(), exchangeGhostCells(), generateTranslocationLookup(), initializeBuffers(), and LatticePatch().

5.9.5.30 ny

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

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

Definition at line 148 of file LatticePatch.h.

Referenced by discreteSize(), exchangeGhostCells(), generateTranslocationLookup(), initializeBuffers(), and LatticePatch().

5.9.5.31 nz

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

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

Definition at line 150 of file LatticePatch.h.

Referenced by discreteSize(), exchangeGhostCells(), generateTranslocationLookup(), initializeBuffers(), and LatticePatch().

5.9.5.32 rgcTox

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

ghost cell translocation lookup table

Definition at line 178 of file LatticePatch.h.

Referenced by generateTranslocationLookup(), and rotateIntoEigen().

5.9.5.33 rgcToy

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

ghost cell translocation lookup table

Definition at line 178 of file LatticePatch.h.

Referenced by generateTranslocationLookup(), and rotateIntoEigen().

5.9.5.34 rgcToz

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

ghost cell translocation lookup table

Definition at line 178 of file LatticePatch.h.

Referenced by generateTranslocationLookup(), and rotateIntoEigen().

5.9.5.35 statusFlags

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

lattice patch status flags

Definition at line 158 of file LatticePatch.h.

Referenced by checkFlag(), exchangeGhostCells(), generateTranslocationLookup(), initializeBuffers(), LatticePatch(), and ∼LatticePatch().

5.9.5.36 u

```
N_Vector LatticePatch::u
```

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

Definition at line 194 of file LatticePatch.h.

Referenced by Simulation::advanceToTime(), Simulation::initializeCVODEobject(), and ∼LatticePatch().

5.9.5.37 uAux

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

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

Definition at line 162 of file LatticePatch.h.

Referenced by derive(), and derotate().

5.9.5.38 uAuxData

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

pointer to auxiliary data vector

Definition at line 200 of file LatticePatch.h.

Referenced by rotateIntoEigen().

5.9.5.39 uData

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

pointer to field data

Definition at line 198 of file LatticePatch.h.

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

5.9.5.40 uTox

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

translocation lookup table

Definition at line 165 of file LatticePatch.h.

Referenced by derotate(), generateTranslocationLookup(), and rotateIntoEigen().

5.9.5.41 uToy

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

translocation lookup table

Definition at line 165 of file LatticePatch.h.

Referenced by derotate(), generateTranslocationLookup(), and rotateIntoEigen().

5.9.5.42 uToz

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

translocation lookup table

Definition at line 165 of file LatticePatch.h.

Referenced by derotate(), generateTranslocationLookup(), and rotateIntoEigen().

5.9.5.43 x0

`sunrealtype LatticePatch::x0 [private]`

origin of the patch in physical space; x-coordinate

Definition at line 128 of file LatticePatch.h.

Referenced by LatticePatch(), and origin().

5.9.5.44 xTou

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

translocation lookup table

Definition at line 165 of file LatticePatch.h.

Referenced by generateTranslocationLookup().

5.9.5.45 y0

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

origin of the patch in physical space; y-coordinate

Definition at line 130 of file LatticePatch.h.

Referenced by LatticePatch(), and origin().

5.9.5.46 yTou

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

translocation lookup table

Definition at line 165 of file LatticePatch.h.

Referenced by generateTranslocationLookup().

5.9.5.47 z0

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

origin of the patch in physical space; z-coordinate

Definition at line 132 of file LatticePatch.h.

Referenced by LatticePatch(), and origin().

5.9.5.48 zTou

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

translocation lookup table

Definition at line 165 of file LatticePatch.h.

Referenced by generateTranslocationLookup().

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

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

5.10 OutputManager Class Reference

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

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

Public Member Functions

- OutputManager ()

  default constructor
- void generateOutputFolder (const std::string &dir)

  function that creates folder to save simulation data
- void set_outputStyle (const char _outputStyle)

  set the output style
- void outUState (const int &state, const Lattice &lattice, const LatticePatch &latticePatch)

  function to write data to disk in specified way
- const std::string & getSimCode () const

  simCode getter function

Static Private Member Functions

- static std::string SimCodeGenerator ()

  function to create the Code of the Simulations

Private Attributes

- std::string simCode

  varible to safe the SimCode generated at execution
- std::string Path

  variable for the path to the output folder
- char outputStyle

  output style; csv or binary

5.10.1 Detailed Description

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

Definition at line 21 of file Outputters.h.

5.10.2 Constructor & Destructor Documentation

5.10.2.1 OutputManager()

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

default constructor

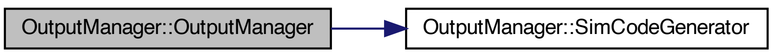
Directly generate the simCode at construction.

Definition at line 12 of file Outputters.cpp.

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

References outputStyle, simCode, and SimCodeGenerator().

Here is the call graph for this function:



5.10.3 Member Function Documentation

5.10.3.1 generateOutputFolder()

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

function that creates folder to save simulation data

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

Definition at line 47 of file Outputters.cpp.

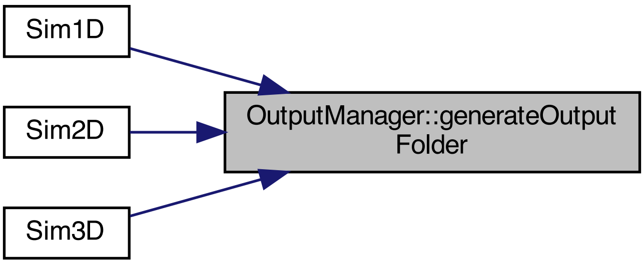
```
00047                                                              {
00048   // Do this only once for the first process
00049   int myPrc;
00050   MPI_Comm_rank(MPI_COMM_WORLD, &myPrc);
00051   if (myPrc == 0) {
00052     if (!fs::is_directory(dir))
00053       fs::create_directory(dir);
00054  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

```
    for (int i = 1; !fin.eof(); i++) {
      getline(fin, line);
      if (line.starts_with("    //------------ B")) {
          begin=i;
      }
      if (i < begin) {
        continue;
      }
      fout « line « std::endl;
      if (line.starts_with("    //------------- E")) {
          break;
      }
    }
    return;
}
```

References Path, and simCode.

Referenced by Sim1D(), Sim2D(), and Sim3D().

Here is the caller graph for this function:



5.10.3.2 getSimCode()

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

simCode getter function

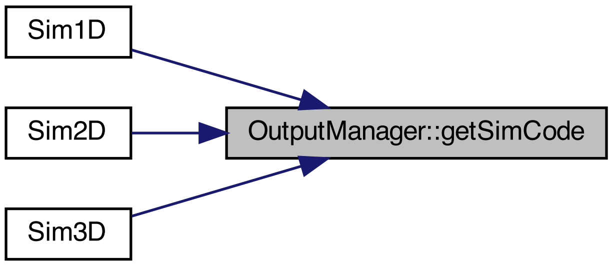
Definition at line 42 of file Outputters.h.

```
{ return simCode; }
```

References simCode.

Referenced by Sim1D(), Sim2D(), and Sim3D().

Here is the caller graph for this function:



5.10.3.3 outUState()

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

function to write data to disk in specified way

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

Definition at line 92 of file Outputters.cpp.

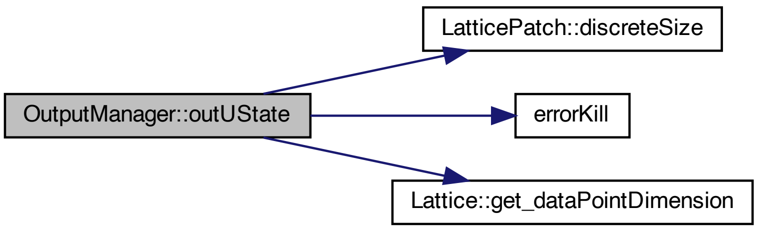
```
                                                {
switch(outputStyle){
    case 'c': { // one csv file per process
                  std::ofstream ofs;
ofs.open(Path + std::to_string(state) + "_"
        + std::to_string(lattice.my_prc) + ".csv");
// Precision of sunrealtype in significant decimal digits; 15 for IEEE double
ofs « std::setprecision(std::numeric_limits<sunrealtype>::digits10);

// Walk through each lattice point
const sunindextype totalNP = latticePatch.discreteSize();
for (sunindextype i = 0; i < totalNP * 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]
      « std::endl;
}
ofs.close();
break;
              }

    case 'b': { // a single binary file
// Open the output file
MPI_File fh;
const std::string filename = Path+std::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 sunindextype 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_Request write_request;
    MPI_File_iwrite_all(fh,latticePatch.uData,count,MPI_SUNREALTYPE,
            &write_request);
    MPI_Wait(&write_request,MPI_STATUS_IGNORE);
    MPI_File_close(&fh);
    break;
                  }
        default: {
    errorKill("No valid output style defined."
            " Choose between (c): one csv file per process,"
            " (b) one binary file");
    break;
                 }
    }
  }
```

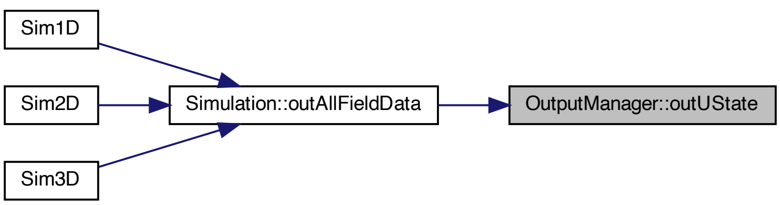
References Lattice::comm, LatticePatch::discreteSize(), errorKill(), Lattice::get_dataPointDimension(), Lattice::my_prc, outputStyle, Path, and LatticePatch::uData.

Referenced by Simulation::outAllFieldData().

Here is the call graph for this function:



Here is the caller graph for this function:



5.10.3.4 set_outputStyle()

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

set the output style

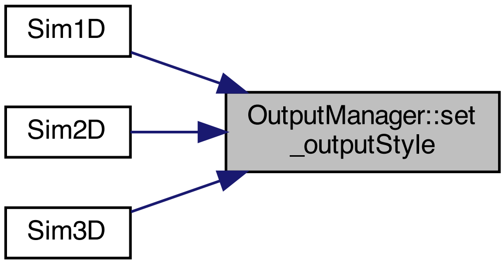
Definition at line 83 of file Outputters.cpp.

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

References outputStyle.

Referenced by Sim1D(), Sim2D(), and Sim3D().

Here is the caller graph for this function:



5.10.3.5 SimCodeGenerator()

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

function to create the Code of the Simulations

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

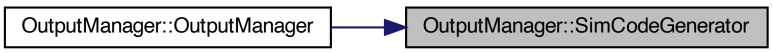
Definition at line 19 of file Outputters.cpp.

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

```
        « std::setw(2) « hms.minutes().count() « "-"
        « std::setfill('0') « std::setw(2)
        « hms.seconds().count();
        //« "_" « hms.subseconds().count(); // subseconds render the filename
        // too large
    return temp.str();
}
```

Referenced by OutputManager().

Here is the caller graph for this function:



5.10.4 Field Documentation

5.10.4.1 outputStyle

`char OutputManager::outputStyle [private]`

output style; csv or binary

Definition at line 30 of file Outputters.h.

Referenced by OutputManager(), outUState(), and set_outputStyle().

5.10.4.2 Path

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

variable for the path to the output folder

Definition at line 28 of file Outputters.h.

Referenced by generateOutputFolder(), and outUState().

5.10.4.3 simCode

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

varible to safe the SimCode generated at execution

Definition at line 26 of file Outputters.h.

Referenced by generateOutputFolder(), getSimCode(), and OutputManager().

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

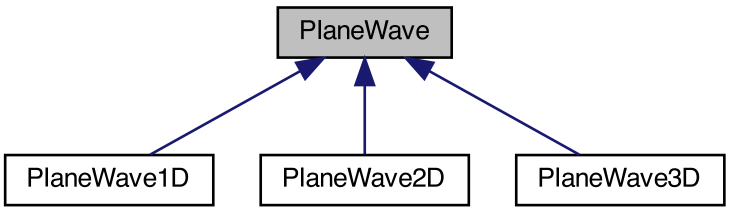
- src/Outputters.h
- src/Outputters.cpp

5.11 PlaneWave Class Reference

super-class for plane waves

`#include <src/ICSetters.h>`

Inheritance diagram for PlaneWave:



Protected Attributes

- sunrealtype kx

  wavenumber k_x
- sunrealtype ky

  wavenumber k_y
- sunrealtype kz

  wavenumber k_z
- sunrealtype px

  polarization & amplitude in x-direction, p_x
- sunrealtype py

  polarization & amplitude in y-direction, p_y
- sunrealtype pz

  polarization & amplitude in z-direction, p_z
- sunrealtype phix

  phase shift in x-direction, ϕ_x
- sunrealtype phiy

  phase shift in y-direction, ϕ_y
- sunrealtype phiz

  phase shift in z-direction, ϕ_z

5.11.1 Detailed Description

super-class for plane waves

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

Definition at line 20 of file ICSetters.h.

5.11.2 Field Documentation

5.11.2.1 kx

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

wavenumber k_x

Definition at line 23 of file ICSetters.h.

Referenced by PlaneWave1D::addToSpace(), PlaneWave2D::addToSpace(), PlaneWave3D::addToSpace(), PlaneWave1D::PlaneWave1D(), PlaneWave2D::PlaneWave2D(), and PlaneWave3D::PlaneWave3D().

5.11.2.2 ky

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

wavenumber k_y

Definition at line 25 of file ICSetters.h.

Referenced by PlaneWave1D::addToSpace(), PlaneWave2D::addToSpace(), PlaneWave3D::addToSpace(), PlaneWave1D::PlaneWave1D(), PlaneWave2D::PlaneWave2D(), and PlaneWave3D::PlaneWave3D().

5.11.2.3 kz

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

wavenumber k_z

Definition at line 27 of file ICSetters.h.

Referenced by PlaneWave1D::addToSpace(), PlaneWave2D::addToSpace(), PlaneWave3D::addToSpace(), PlaneWave1D::PlaneWave1D(), PlaneWave2D::PlaneWave2D(), and PlaneWave3D::PlaneWave3D().

5.11.2.4 phix

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

phase shift in x-direction, ϕ_x

Definition at line 35 of file ICSetters.h.

Referenced by PlaneWave1D::addToSpace(), PlaneWave2D::addToSpace(), PlaneWave3D::addToSpace(), PlaneWave1D::PlaneWave1D(), PlaneWave2D::PlaneWave2D(), and PlaneWave3D::PlaneWave3D().

5.11.2.5 phiy

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

phase shift in y-direction, ϕ_y

Definition at line 37 of file ICSetters.h.

Referenced by PlaneWave1D::addToSpace(), PlaneWave2D::addToSpace(), PlaneWave3D::addToSpace(), PlaneWave1D::PlaneWave1D(), PlaneWave2D::PlaneWave2D(), and PlaneWave3D::PlaneWave3D().

5.11.2.6 phiz

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

phase shift in z-direction, ϕ_z

Definition at line 39 of file ICSetters.h.

Referenced by PlaneWave1D::addToSpace(), PlaneWave2D::addToSpace(), PlaneWave3D::addToSpace(), PlaneWave1D::PlaneWave1D(), PlaneWave2D::PlaneWave2D(), and PlaneWave3D::PlaneWave3D().

5.11.2.7 px

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

polarization & amplitude in x-direction, p_x

Definition at line 29 of file ICSetters.h.

Referenced by PlaneWave1D::addToSpace(), PlaneWave2D::addToSpace(), PlaneWave3D::addToSpace(), PlaneWave1D::PlaneWave1D(), PlaneWave2D::PlaneWave2D(), and PlaneWave3D::PlaneWave3D().

5.11.2.8 py

`sunrealtype PlaneWave::py [protected]`

polarization & amplitude in y-direction, p_y

Definition at line 31 of file ICSetters.h.

Referenced by PlaneWave1D::addToSpace(), PlaneWave2D::addToSpace(), PlaneWave3D::addToSpace(), PlaneWave1D::PlaneWave1D(), PlaneWave2D::PlaneWave2D(), and PlaneWave3D::PlaneWave3D().

5.11.2.9 pz

`sunrealtype PlaneWave::pz [protected]`

polarization & amplitude in z-direction, p_z

Definition at line 33 of file ICSetters.h.

Referenced by PlaneWave1D::addToSpace(), PlaneWave2D::addToSpace(), PlaneWave3D::addToSpace(), PlaneWave1D::PlaneWave1D(), PlaneWave2D::PlaneWave2D(), and PlaneWave3D::PlaneWave3D().

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

- src/ICSetters.h

5.12 planewave Struct Reference

plane wave structure

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

Data Fields

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

5.12.1 Detailed Description

plane wave structure

Definition at line 19 of file SimulationFunctions.h.

5.12.2 Field Documentation

5.12.2.1 k

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

wavevector (normalized to $1/\lambda$)

Definition at line 20 of file SimulationFunctions.h.

5.12.2.2 p

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

amplitde & polarization vector

Definition at line 21 of file SimulationFunctions.h.

5.12.2.3 phi

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

phase shift

Definition at line 22 of file SimulationFunctions.h.

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

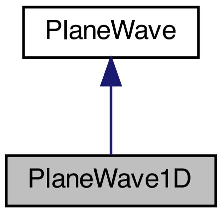
- src/SimulationFunctions.h

5.13 PlaneWave1D Class Reference

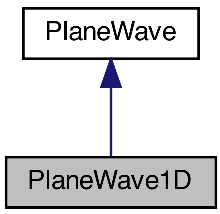
class for plane waves in 1D

`#include <src/ICSetters.h>`

Inheritance diagram for PlaneWave1D:



Collaboration diagram for PlaneWave1D:



Public Member Functions

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

    construction with default parameters
- void addToSpace (sunrealtype x, sunrealtype y, sunrealtype z, sunrealtype ∗pTo6Space) const

    function for the actual implementation in the lattice

Additional Inherited Members

5.13.1 Detailed Description

class for plane waves in 1D

Definition at line 43 of file ICSetters.h.

5.13.2 Constructor & Destructor Documentation

5.13.2.1 PlaneWave1D()

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

construction with default parameters

PlaneWave1D construction with

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

Definition at line 11 of file ICSetters.cpp.

```
                                      {
  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 27 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 *
        std::numbers::pi; /* \f$ 2\pi \ \vec{k} \cdot \vec{x} \f$ */
    // Plane wave definition
    const std::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:

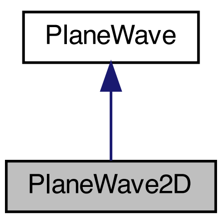
- src/ICSetters.h
- src/ICSetters.cpp

5.14 PlaneWave2D Class Reference

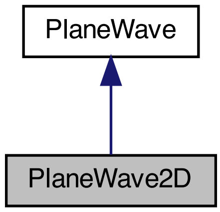
class for plane waves in 2D

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

Inheritance diagram for PlaneWave2D:



Collaboration diagram for PlaneWave2D:



Public Member Functions

- PlaneWave2D (std::array< sunrealtype, 3 > k={1, 0, 0}, std::array< sunrealtype, 3 > p={0, 0, 1}, std::array< sunrealtype, 3 > phi={0, 0, 0})

    construction with default parameters
- void addToSpace (sunrealtype x, sunrealtype y, sunrealtype z, sunrealtype ∗pTo6Space) const

    function for the actual implementation in the lattice

Additional Inherited Members

5.14.1 Detailed Description

class for plane waves in 2D

Definition at line 55 of file ICSetters.h.

5.14.2 Constructor & Destructor Documentation

5.14.2.1 PlaneWave2D()

```
PlaneWave2D::PlaneWave2D (
            std::array< sunrealtype, 3 > k = {1, 0, 0},
            std::array< sunrealtype, 3 > p = {0, 0, 1},
            std::array< sunrealtype, 3 > phi = {0, 0, 0} )
```

construction with default parameters

PlaneWave2D construction with

- wavevectors k_x

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

Definition at line 50 of file ICSetters.cpp.

```
                                            {
    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 *
        std::numbers::pi; /* \f$ 2\pi \ \vec{k} \cdot \vec{x} \f$ */
    // Plane wave definition
    const std::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;
```

```
00083   pTo6Space[4] += (kz * E[0] - kx * E[2]) / wavelength;
00084   pTo6Space[5] += (kx * E[1] - ky * E[0]) / wavelength;
00085 }
```

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

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

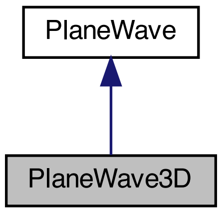
- src/ICSetters.h
- src/ICSetters.cpp

5.15 PlaneWave3D Class Reference

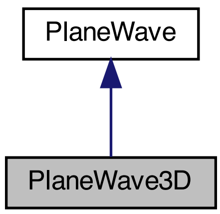
class for plane waves in 3D

`#include <src/ICSetters.h>`

Inheritance diagram for PlaneWave3D:



Collaboration diagram for PlaneWave3D:



Public Member Functions

- PlaneWave3D (std::array< sunrealtype, 3 > k={1, 0, 0}, std::array< sunrealtype, 3 > p={0, 0, 1}, std::array< sunrealtype, 3 > phi={0, 0, 0})

    construction with default parameters
- void addToSpace (sunrealtype x, sunrealtype y, sunrealtype z, sunrealtype ∗pTo6Space) const

    function for the actual implementation in space

Additional Inherited Members

5.15.1 Detailed Description

class for plane waves in 3D

Definition at line 67 of file ICSetters.h.

5.15.2 Constructor & Destructor Documentation

5.15.2.1 PlaneWave3D()

```
PlaneWave3D::PlaneWave3D (
            std::array< sunrealtype, 3 > k = {1, 0, 0},
            std::array< sunrealtype, 3 > p = {0, 0, 1},
            std::array< sunrealtype, 3 > phi = {0, 0, 0} )
```

construction with default parameters

PlaneWave3D construction with

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

Definition at line 88 of file ICSetters.cpp.

```
                                                    {
  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 *
        std::numbers::pi; /* \f$ 2\pi \ \vec{k} \cdot \vec{x} \f$ */
    // Plane wave definition
    const std::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:

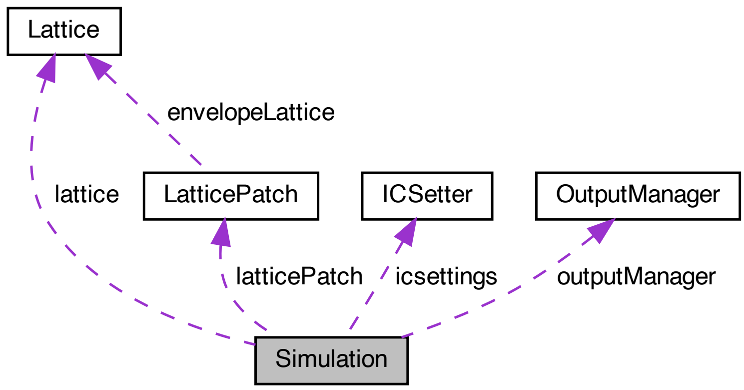
- src/ICSetters.h
- src/ICSetters.cpp

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 sunindextype xm, const sunindextype ym, const sunindextype zm=0)

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

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

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

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

  function to write field data to disk

- void checkFlag (unsigned int flag) const

  function to check if flag has been set
- void checkNoFlag (unsigned int flag) const

  function to check if flag has not been set

Data Fields

- ICSetter icsettings

  IC Setter object.
- OutputManager outputManager

  Output Manager object.
- void ∗ cvode_mem

  pointer to CVode memory object
- SUNNonlinearSolver NLS

  nonlinear solver object

Private Attributes

- Lattice lattice

  Lattice object.
- LatticePatch latticePatch

  LatticePatch object.
- sunrealtype t

  current time of the simulation
- unsigned int statusFlags

  simulation status flags

5.16.1 Detailed Description

Simulation class to instantiate the whole walkthrough of a Simulation.

Definition at line 30 of file SimulationClass.h.

5.16.2 Constructor & Destructor Documentation

5.16.2.1 Simulation()

```
Simulation::Simulation (
            const int Nx,
            const int Ny,
            const int Nz,
            const int StencilOrder,
            const bool periodicity )
```

constructor function for the creation of the cartesian communicator

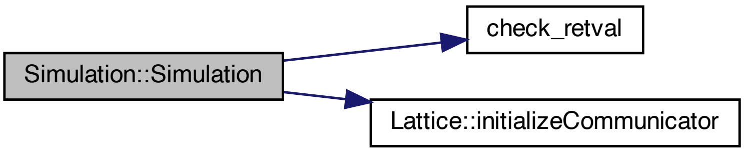
Along with the simulation object, create the cartesian communicator and SUNContext object

Definition at line 14 of file SimulationClass.cpp.

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

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

Here is the call graph for this function:



5.16.2.2 ∼Simulation()

```
Simulation::∼Simulation ( )
```

destructor function freeing CVode memory and Sundials context

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

Definition at line 31 of file SimulationClass.cpp.

```
00031                          {
00032   // Free solver memory
00033   if (statusFlags & CvodeObjectSetUp) {
00034     CVodeFree(&cvode_mem);
00035     SUNNonlinSolFree(NLS);
00036     SUNContext_Free(&lattice.sunctx);
00037   }
00038 }
```

References cvode_mem, CvodeObjectSetUp, lattice, NLS, statusFlags, and Lattice::sunctx.

5.16.3 Member Function Documentation

5.16.3.1 addInitialConditions()

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

functions to add initial periodic field configurations

Use parameters to add periodic IC layers.

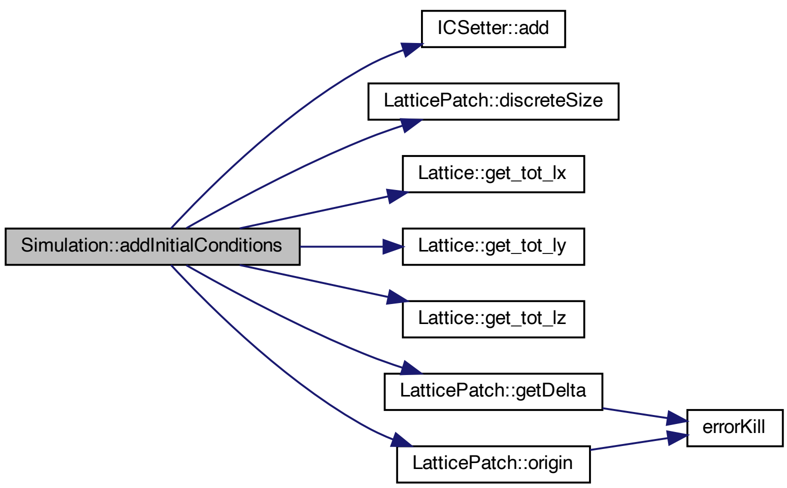
Definition at line 167 of file SimulationClass.cpp.

```
                                    {
  const sunrealtype dx = latticePatch.getDelta(1);
  const sunrealtype dy = latticePatch.getDelta(2);
  const sunrealtype dz = latticePatch.getDelta(3);
  const sunindextype nx = latticePatch.discreteSize(1);
  const sunindextype ny = latticePatch.discreteSize(2);
  const sunindextype totalNP = latticePatch.discreteSize();
  // 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();
  sunindextype px = 0, py = 0, pz = 0;
  for (sunindextype i = 0; i < totalNP * 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.

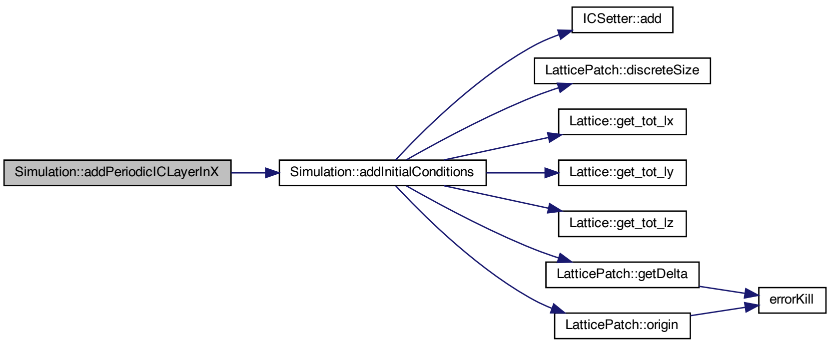
Definition at line 193 of file SimulationClass.cpp.

```
00193                                            {
00194   addInitialConditions(-1, 0, 0);
00195   addInitialConditions(1, 0, 0);
00196   return;
00197 }
```

References addInitialConditions().

Referenced by Sim1D().

Here is the call graph for this function:



Here is the caller graph for this function:

5.16.3.3 addPeriodicICLayerInXY()

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

function to add periodic IC layers in two dimensions

Add initial conditions in two dimensions.

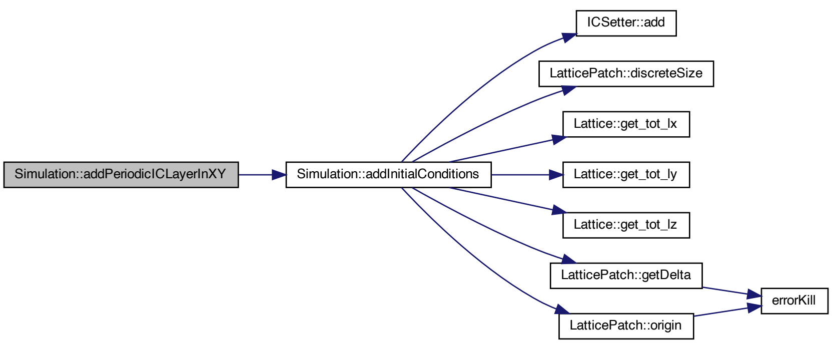
Definition at line 200 of file SimulationClass.cpp.

```
                                        {
    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;
}
```

References addInitialConditions().

Referenced by Sim2D().

Here is the call graph for this function:



Here is the caller graph for this function:

5.16.3.4 advanceToTime()

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

function to advance solution in time with CVODE

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

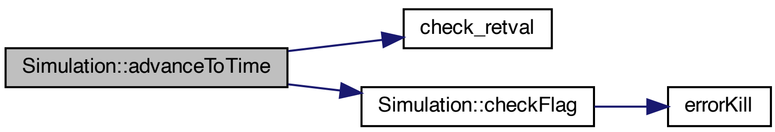
Definition at line 213 of file SimulationClass.cpp.

```
00213                                                    {
00214   checkFlag(SimulationStarted);
00215   int retval = 0;
00216   retval = CVode(cvode_mem, tEnd, latticePatch.u, &t,
00217                  CV_NORMAL); // CV_NORMAL: internal steps to reach tEnd, then
00218                              // interpolate to return latticePatch.u, return time
00219                              // reached by the solver as t
00220   if (check_retval(&retval, "CVode", 1, lattice.my_prc))
00221     MPI_Abort(lattice.comm, 1);
00222 }
```

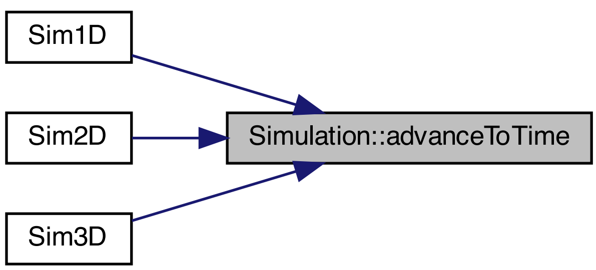
References check_retval(), checkFlag(), Lattice::comm, cvode_mem, lattice, latticePatch, Lattice::my_prc, SimulationStarted, t, and LatticePatch::u.

Referenced by Sim1D(), Sim2D(), and Sim3D().

Here is the call graph for this function:



Here is the caller graph for this function:



5.16.3.5 checkFlag()

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

function to check if flag has been set

Check presence of configuration flags.

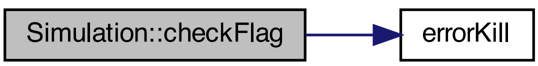
Definition at line 231 of file SimulationClass.cpp.

```
00231                                                  {
00232   if (!(statusFlags & flag)) {
00233     std::string errorMessage;
00234     switch (flag) {
00235     case LatticeDiscreteSetUp:
00236       errorMessage = "The discrete size of the Simulation has not been set up";
00237       break;
00238     case LatticePhysicalSetUp:
00239       errorMessage = "The physical size of the Simulation has not been set up";
00240       break;
00241     case LatticePatchworkSetUp:
00242       errorMessage = "The patchwork for the Simulation has not been set up";
00243       break;
00244     case CvodeObjectSetUp:
00245       errorMessage = "The CVODE object has not been initialized";
00246       break;
00247     case SimulationStarted:
00248       errorMessage = "The Simulation has not been started";
00249       break;
00250     default:
00251       errorMessage = "Uppss, you've made a non-standard error, sadly I can't "
00252                      "help you there";
00253       break;
00254     }
00255     errorKill(errorMessage);
00256   }
00257   return;
00258 }
```

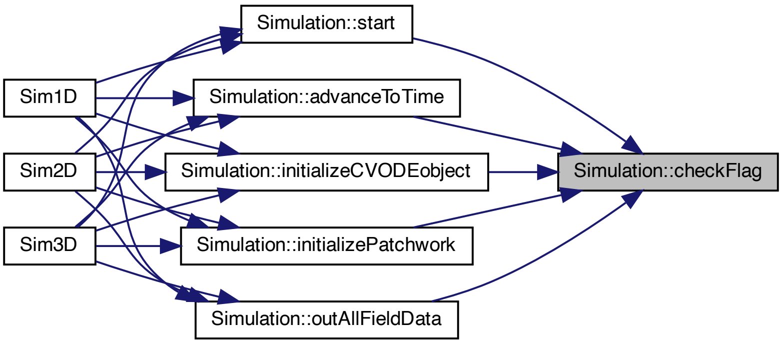
References CvodeObjectSetUp, errorKill(), LatticeDiscreteSetUp, LatticePatchworkSetUp, LatticePhysicalSetUp, SimulationStarted, and statusFlags.

Referenced by advanceToTime(), initializeCVODEobject(), initializePatchwork(), outAllFieldData(), and start().

Here is the call graph for this function:



Here is the caller graph for this function:



5.16.3.6 checkNoFlag()

```
void Simulation::checkNoFlag (
            unsigned int flag ) const
```

function to check if flag has not been set

Check absence of configuration flags.

Definition at line 261 of file SimulationClass.cpp.

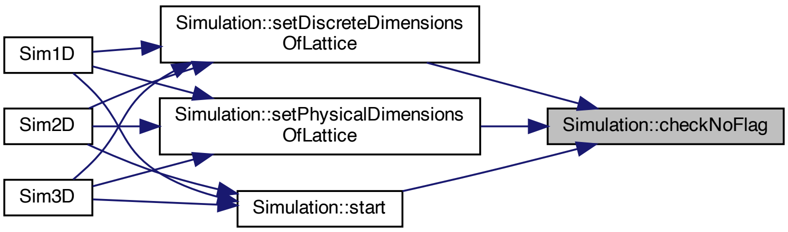
```
00261                                                           {
00262   if ((statusFlags & flag)) {
00263     std::string errorMessage;
00264     switch (flag) {
00265     case LatticeDiscreteSetUp:
00266       errorMessage =
00267           "The discrete size of the Simulation has already been set up";
00268       break;
00269     case LatticePhysicalSetUp:
00270       errorMessage =
00271           "The physical size of the Simulation has already been set up";
00272       break;
00273     case LatticePatchworkSetUp:
00274       errorMessage = "The patchwork for the Simulation has already been set up";
00275       break;
00276     case CvodeObjectSetUp:
00277       errorMessage = "The CVODE object has already been initialized";
00278       break;
00279     case SimulationStarted:
00280       errorMessage = "The simulation has already started, some changes are no "
00281                      "longer possible";
00282       break;
00283     default:
00284       errorMessage = "Uppss, you've made a non-standard error, sadly I can't "
00285                      "help you there";
00286       break;
00287     }
00288     errorKill(errorMessage);
00289   }
00290   return;
00291 }
```

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

Referenced by setDiscreteDimensionsOfLattice(), setPhysicalDimensionsOfLattice(), and start().

Here is the call graph for this function:

Here is the caller graph for this function:



5.16.3.7 get_cart_comm()

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

reference to the cartesian communicator of the lattice (for debugging)

Definition at line 56 of file SimulationClass.h.

```
00056 { return &lattice.comm; }
```

References Lattice::comm, and lattice.

5.16.3.8 initializeCVODEobject()

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

function to initialize the CVODE object with all requirements

Configure CVODE.

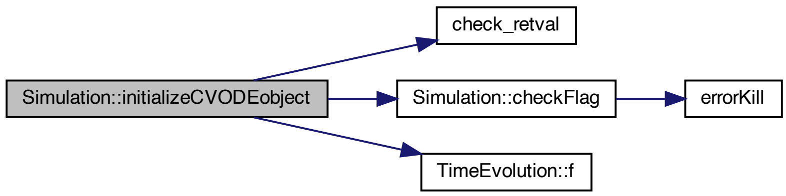
Definition at line 70 of file SimulationClass.cpp.

```
                                {
  checkFlag(SimulationStarted);

  // CVode settings return value
  int retval = 0;

  // 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);

  // Initialize CVODE solver
  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);

  // Create fixed point nonlinear solver object (suitable for non-stiff ODE) and
  // attach it to CVode
  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);

  // Anderson damping factor
  retval = SUNNonlinSolSetDamping_FixedPoint(NLS,1);
  if (check_retval(&retval, "SUNNonlinSolSetDamping_FixedPoint", 1,
              lattice.my_prc)) 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);

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

  // maximum number of warnings for too small h
  retval = CVodeSetMaxHnilWarns(cvode_mem,3);
  if (check_retval(&retval, "CVodeSetMaxHnilWarns", 1, lattice.my_prc))
    MPI_Abort(lattice.comm, 1);

  statusFlags |= CvodeObjectSetUp;
}
```

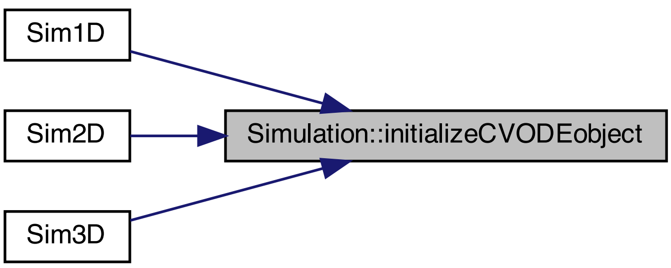
References check_retval(), checkFlag(), Lattice::comm, cvode_mem, CvodeObjectSetUp, TimeEvolution::f(), lattice, latticePatch, Lattice::my_prc, NLS, SimulationStarted, statusFlags, Lattice::sunctx, and LatticePatch::u.

Referenced by Sim1D(), Sim2D(), and Sim3D().

Here is the call graph for this function:



Here is the caller graph for this function:



5.16.3.9 initializePatchwork()

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

function to initialize the Patchwork

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

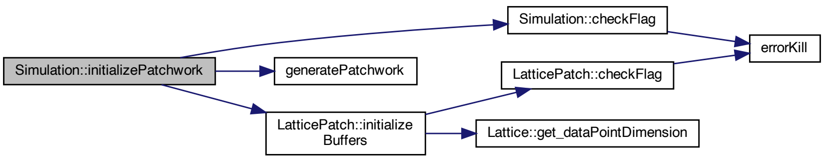
Definition at line 57 of file SimulationClass.cpp.

```
00058                         {
00059   checkFlag(LatticeDiscreteSetUp);
00060   checkFlag(LatticePhysicalSetUp);
00061
00062   // Generate the patchwork
00063   generatePatchwork(lattice, latticePatch, nx, ny, nz);
00064   latticePatch.initializeBuffers();
00065
00066   statusFlags |= LatticePatchworkSetUp;
00067 }
```

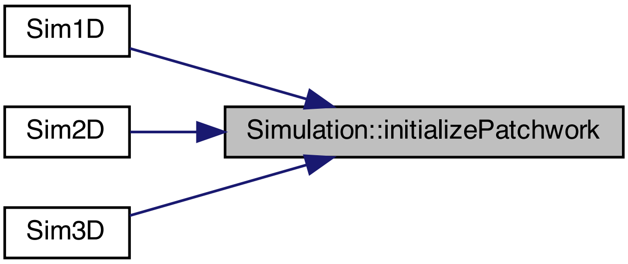
References checkFlag(), generatePatchwork(), LatticePatch::initializeBuffers(), lattice, LatticeDiscreteSetUp, latticePatch, LatticePatchworkSetUp, LatticePhysicalSetUp, and statusFlags.

Referenced by Sim1D(), Sim2D(), and Sim3D().

Here is the call graph for this function:



Here is the caller graph for this function:



5.16.3.10 outAllFieldData()

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

function to write field data to disk

Write specified simulation steps to disk.

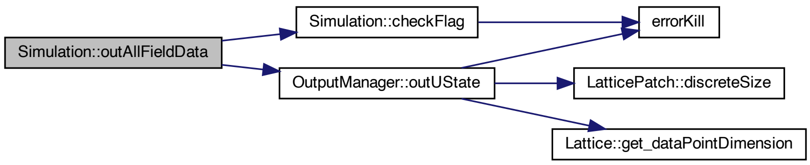
Definition at line 225 of file SimulationClass.cpp.

```
                                                    {
  checkFlag(SimulationStarted);
  outputManager.outUState(state, lattice, latticePatch);
}
```

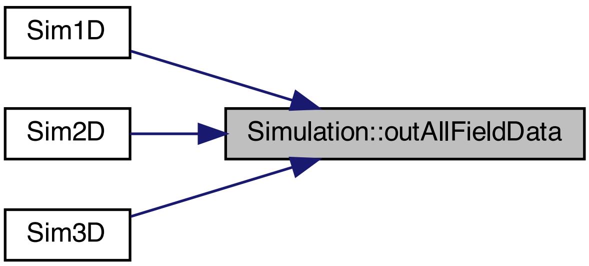
References checkFlag(), lattice, latticePatch, outputManager, OutputManager::outUState(), and SimulationStarted.

Referenced by Sim1D(), Sim2D(), and Sim3D().

Here is the call graph for this function:



Here is the caller graph for this function:



5.16.3.11 setDiscreteDimensionsOfLattice()

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

function to set discrete dimensions of the lattice

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

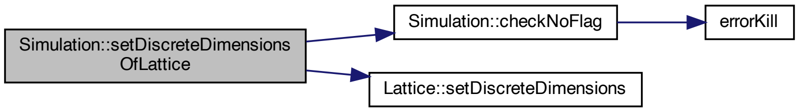
Definition at line 41 of file SimulationClass.cpp.

```
00042                                                           {
00043   checkNoFlag(LatticePatchworkSetUp);
00044   lattice.setDiscreteDimensions(nx, ny, nz);
00045   statusFlags |= LatticeDiscreteSetUp;
00046 }
```

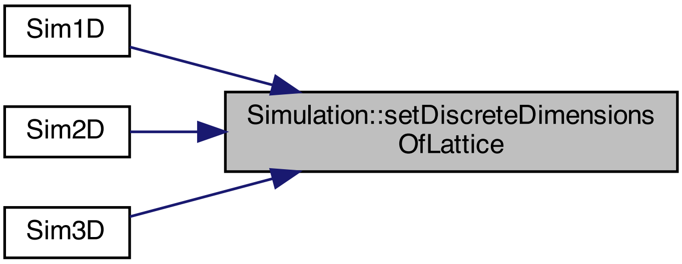
References checkNoFlag(), lattice, LatticeDiscreteSetUp, LatticePatchworkSetUp, Lattice::setDiscreteDimensions(), and statusFlags.

Referenced by Sim1D(), Sim2D(), and Sim3D().

Here is the call graph for this function:



Here is the caller graph for this function:



5.16.3.12 setInitialConditions()

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

functions to set the initial field configuration onto the lattice

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

Definition at line 140 of file SimulationClass.cpp.

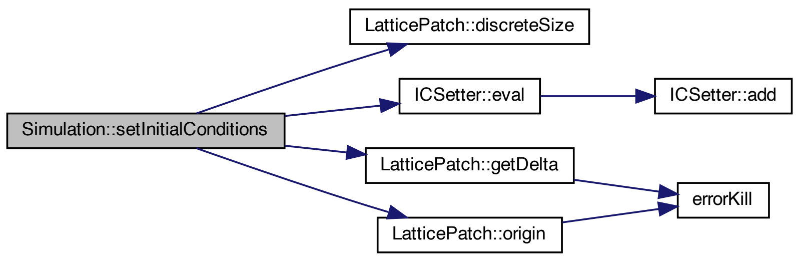
```
00140                                        {
00141   const sunrealtype dx = latticePatch.getDelta(1);
00142   const sunrealtype dy = latticePatch.getDelta(2);
00143   const sunrealtype dz = latticePatch.getDelta(3);
00144   const sunindextype nx = latticePatch.discreteSize(1);
00145   const sunindextype ny = latticePatch.discreteSize(2);
00146   const sunindextype totalNP = latticePatch.discreteSize();
00147   const sunrealtype x0 = latticePatch.origin(1);
00148   const sunrealtype y0 = latticePatch.origin(2);
00149   const sunrealtype z0 = latticePatch.origin(3);
00150   sunindextype px = 0, py = 0, pz = 0;
00151   #pragma omp parallel for default(none) \
00152   shared(nx, ny, totalNP, dx, dy, dz, x0, y0, z0) \
00153   firstprivate(px, py, pz) schedule(static)
00154   for (sunindextype i = 0; i < totalNP * 6; i += 6) {
00155     px = (i / 6) % nx;
00156     py = ((i / 6) / nx) % ny;
00157     pz = ((i / 6) / nx) / ny;
00158     // Call the `eval` function to fill the lattice points with the field data
00159     icsettings.eval(static_cast<sunrealtype>(px) * dx + x0,
```

```
            static_cast<sunrealtype>(py) * dy + y0,
            static_cast<sunrealtype>(pz) * dz + z0, &latticePatch.uData[i]);
  }
  return;
}
```

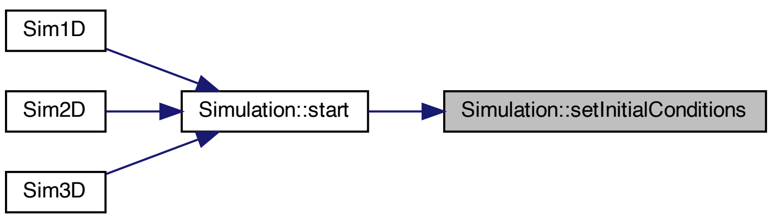
References LatticePatch::discreteSize(), ICSetter::eval(), LatticePatch::getDelta(), icsettings, latticePatch, LatticePatch::origin(), and LatticePatch::uData.

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.

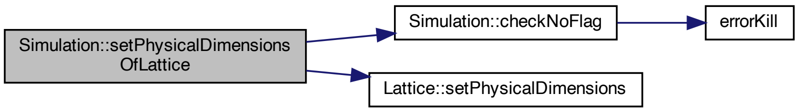
Definition at line 49 of file SimulationClass.cpp.

```
00050                                                      {
00051   checkNoFlag(LatticePatchworkSetUp);
00052   lattice.setPhysicalDimensions(lx, ly, lz);
00053   statusFlags |= LatticePhysicalSetUp;
00054 }
```

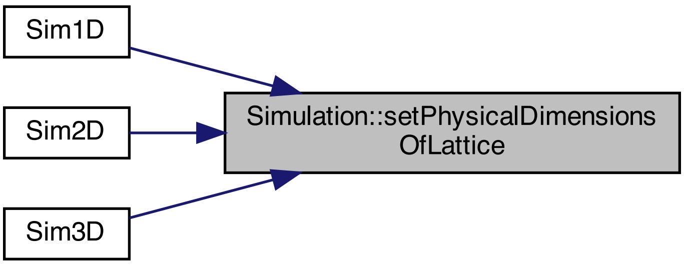
References checkNoFlag(), lattice, LatticePatchworkSetUp, LatticePhysicalSetUp, Lattice::setPhysicalDimensions(), and statusFlags.

Referenced by Sim1D(), Sim2D(), and Sim3D().

Here is the call graph for this function:



Here is the caller graph for this function:



5.16.3.14 start()

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

function to start the simulation for time iteration

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

Definition at line 128 of file SimulationClass.cpp.

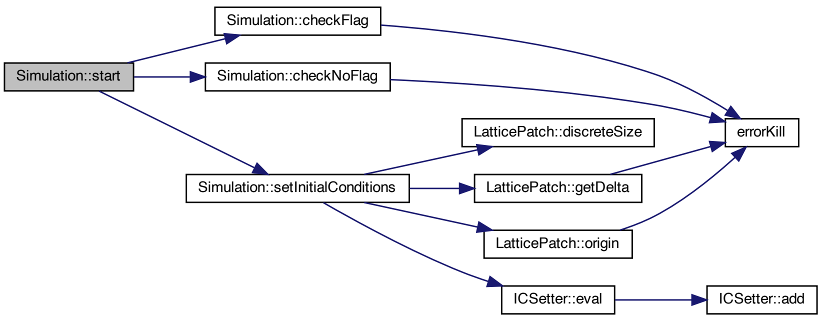
```
00128                            {
00129   checkFlag(LatticeDiscreteSetUp);
00130   checkFlag(LatticePhysicalSetUp);
00131   checkFlag(LatticePatchworkSetUp);
00132   checkNoFlag(SimulationStarted);
```

```
  checkNoFlag(CvodeObjectSetUp);
  setInitialConditions();
  statusFlags |= SimulationStarted;
}
```

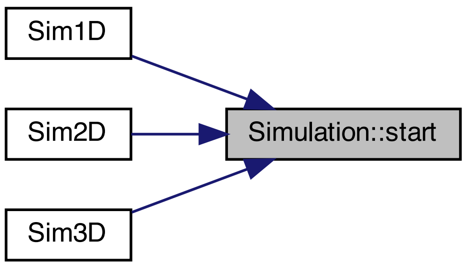
References checkFlag(), checkNoFlag(), CvodeObjectSetUp, LatticeDiscreteSetUp, LatticePatchworkSetUp, LatticePhysicalSetUp, setInitialConditions(), SimulationStarted, and statusFlags.

Referenced by Sim1D(), Sim2D(), and Sim3D().

Here is the call graph for this function:



Here is the caller graph for this function:



5.16.4 Field Documentation

5.16.4.1 cvode_mem

`void* Simulation::cvode_mem`

pointer to CVode memory object

Definition at line 47 of file SimulationClass.h.

Referenced by advanceToTime(), initializeCVODEobject(), and ∼Simulation().

5.16.4.2 icsettings

```
ICSetter Simulation::icsettings
```

IC Setter object.

Definition at line 43 of file SimulationClass.h.

Referenced by addInitialConditions(), setInitialConditions(), Sim1D(), Sim2D(), and Sim3D().

5.16.4.3 lattice

```
Lattice Simulation::lattice [private]
```

Lattice object.

Definition at line 33 of file SimulationClass.h.

Referenced by addInitialConditions(), advanceToTime(), get_cart_comm(), initializeCVODEobject(), initializePatchwork(), outAllFieldData(), setDiscreteDimensionsOfLattice(), setPhysicalDimensionsOfLattice(), Simulation(), and ∼Simulation().

5.16.4.4 latticePatch

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

LatticePatch object.

Definition at line 35 of file SimulationClass.h.

Referenced by addInitialConditions(), advanceToTime(), initializeCVODEobject(), initializePatchwork(), outAllFieldData(), and setInitialConditions().

5.16.4.5 NLS

```
SUNNonlinearSolver Simulation::NLS
```

nonlinear solver object

Definition at line 49 of file SimulationClass.h.

Referenced by initializeCVODEobject(), and ∼Simulation().

5.16.4.6 outputManager

`OutputManager Simulation::outputManager`

Output Manager object.

Definition at line 45 of file SimulationClass.h.

Referenced by outAllFieldData(), Sim1D(), Sim2D(), and Sim3D().

5.16.4.7 statusFlags

`unsigned int Simulation::statusFlags [private]`

simulation status flags

Definition at line 39 of file SimulationClass.h.

Referenced by checkFlag(), checkNoFlag(), initializeCVODEobject(), initializePatchwork(), setDiscreteDimensionsOfLattice(), setPhysicalDimensionsOfLattice(), Simulation(), start(), and ∼Simulation().

5.16.4.8 t

`sunrealtype Simulation::t [private]`

current time of the simulation

Definition at line 37 of file SimulationClass.h.

Referenced by advanceToTime(), and Simulation().

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

- src/SimulationClass.h
- src/SimulationClass.cpp

5.17 TimeEvolution Class Reference

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

`#include <src/TimeEvolutionFunctions.h>`

Static Public Member Functions

- static int f (sunrealtype t, N_Vector u, N_Vector udot, void ∗data_loc)

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

Static Public Attributes

- static int * c = nullptr

    choice which processes of the weak field expansion are included
- static void(* TimeEvolver)(LatticePatch *, N_Vector, N_Vector, int *) = nonlinear1DProp

    Pointer to functions for differentiation and time evolution.

5.17.1 Detailed Description

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

Definition at line 15 of file TimeEvolutionFunctions.h.

5.17.2 Member Function Documentation

5.17.2.1 f()

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

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

CVode right-hand-side function (CVRhsFn)

Definition at line 11 of file TimeEvolutionFunctions.cpp.

```
00011                                                                           {
00012
00013   // Set recover pointer to provided lattice patch where the field data resides
00014   LatticePatch *data = static_cast<LatticePatch *>(data_loc);
00015
00016   // update circle
00017   // Access provided field values and temp. derivatieves with NVector pointers
00018   sunrealtype *udata = NV_DATA_P(u),
00019               *dudata = NV_DATA_P(udot);
00020
00021   // Store original data location of the patch
00022   sunrealtype *originaluData = data->uData,
00023               *originalduData = data->duData;
00024
00025   // Point patch data to arguments of f
00026   data->uData = udata;
00027   data->duData = dudata;
00028
00029   // Time-evolve these arguments (the field data) with specific propagator below
00030   TimeEvolver(data, u, udot, c);
00031
00032   // Refer patch data back to original location
00033   data->uData = originaluData;
00034   data->duData = originalduData;
00035
00036   return (0);
00037 }
```

References c, LatticePatch::duData, TimeEvolver, and LatticePatch::uData.

Referenced by Simulation::initializeCVODEobject().

Here is the caller graph for this function:

5.17.3 Field Documentation

5.17.3.1 c

```
int * TimeEvolution::c = nullptr  [static]
```

choice which processes of the weak field expansion are included

Definition at line 18 of file TimeEvolutionFunctions.h.

Referenced by f(), Sim1D(), Sim2D(), and Sim3D().

5.17.3.2 TimeEvolver

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

Pointer to functions for differentiation and time evolution.

Definition at line 21 of file TimeEvolutionFunctions.h.

Referenced by f(), Sim1D(), Sim2D(), and Sim3D().

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

- src/TimeEvolutionFunctions.h
- src/SimulationFunctions.cpp
- src/TimeEvolutionFunctions.cpp

Chapter 6

File Documentation

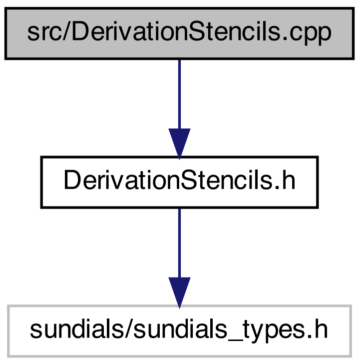
6.1 README.md File Reference

6.2 src/DerivationStencils.cpp File Reference

Empty. All definitions in the header.

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

Include dependency graph for DerivationStencils.cpp:



6.2.1 Detailed Description

Empty. All definitions in the header.

Definition in file DerivationStencils.cpp.

6.3 DerivationStencils.cpp

Go to the documentation of this file.

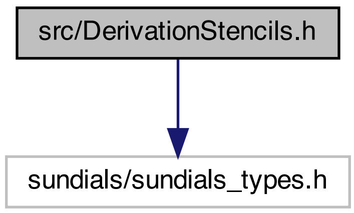
```
///////////////////////////////////////////////////
/// @file DerivationStencils.cpp
/// @brief Empty. All definitions in the header.
//////////////////////////////////////////////////
#include "DerivationStencils.h"
```

6.4 src/DerivationStencils.h File Reference

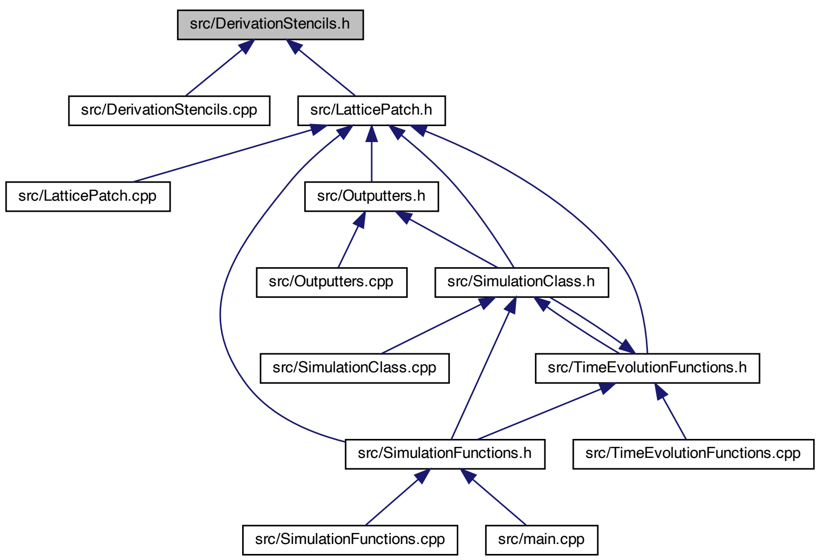
Definition of derivation stencils from order 1 to 13.

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

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

6.4.1 Detailed Description

Definition of derivation stencils from order 1 to 13.

Definition in file DerivationStencils.h.

6.4.2 Function Documentation

6.4.2.1 s10b() [1/2]

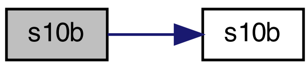
```
sunrealtype s10b (
            sunrealtype const * udata )  [inline]
```

Definition at line 275 of file DerivationStencils.h.

```
00276 { return s10b(udata, 6); }
```

References s10b().

Here is the call graph for this function:



6.4.2.2 s10b() [2/2]

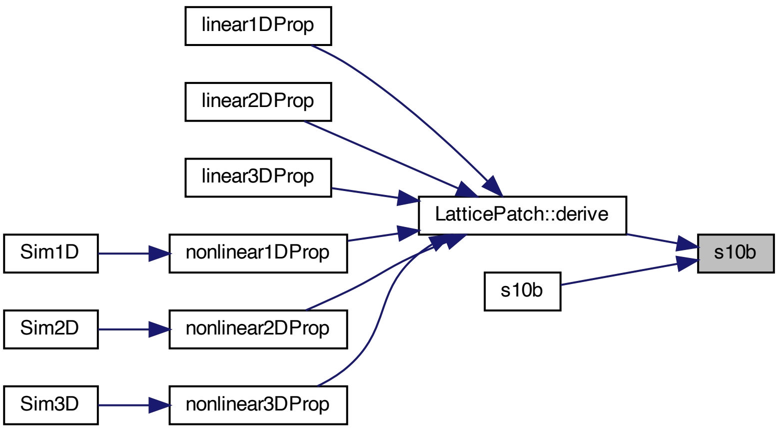
```
sunrealtype s10b (
            sunrealtype const * udata,
            const int dPD ) [inline]
```

Definition at line 169 of file DerivationStencils.h.

```
00169                                                                  {
00170   return 1.0 / 840.0 * udata[-4 * dPD] - 1.0 / 63.0 * udata[-3 * dPD] +
00171          3.0 / 28.0 * udata[-2 * dPD] - 4.0 / 7.0 * udata[-1 * dPD] -
00172          11.0 / 30.0 * udata[0] + 6.0 / 5.0 * udata[1 * dPD] -
00173          1.0 / 2.0 * udata[2 * dPD] + 4.0 / 21.0 * udata[3 * dPD] -
00174          3.0 / 56.0 * udata[4 * dPD] + 1.0 / 105.0 * udata[5 * dPD] -
00175          1.0 / 1260.0 * udata[6 * dPD];
00176 }
```

Referenced by LatticePatch::derive(), and s10b().

Here is the caller graph for this function:



6.4.2.3 s10c() [1/2]

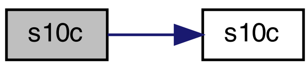
```
sunrealtype s10c (
            sunrealtype const * udata ) [inline]
```

Definition at line 273 of file DerivationStencils.h.

```
00274 { return s10c(udata, 6); }
```

References s10c().

Here is the call graph for this function:



6.4.2.4 s10c() [2/2]

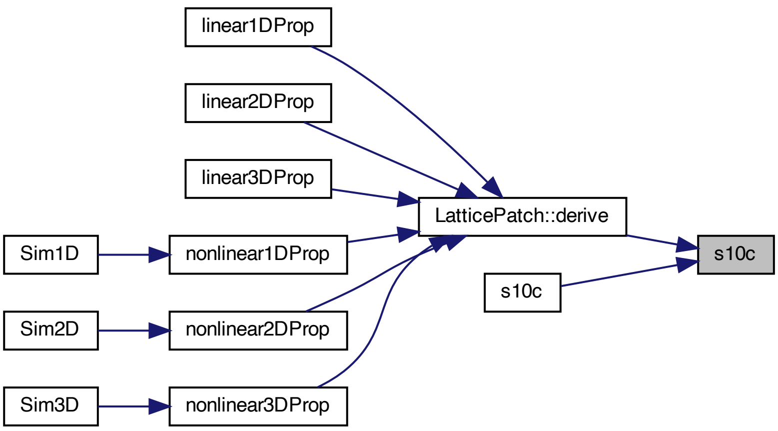
```
sunrealtype s10c (
            sunrealtype const * udata,
            const int dPD )  [inline]
```

Definition at line 161 of file DerivationStencils.h.

```
{
  return -1.0 / 1260.0 * udata[-5 * dPD] + 5.0 / 504.0 * udata[-4 * dPD] -
         5.0 / 84.0 * udata[-3 * dPD] + 5.0 / 21.0 * udata[-2 * dPD] -
         5.0 / 6.0 * udata[-1 * dPD] + 0 + 5.0 / 6.0 * udata[1 * dPD] -
         5.0 / 21.0 * udata[2 * dPD] + 5.0 / 84.0 * udata[3 * dPD] -
         5.0 / 504.0 * udata[4 * dPD] + 1.0 / 1260.0 * udata[5 * dPD];
}
```

Referenced by LatticePatch::derive(), and s10c().

Here is the caller graph for this function:



6.4.2.5 s10f() [1/2]

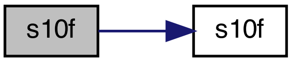
```
sunrealtype s10f (
            sunrealtype const * udata )  [inline]
```

Definition at line 271 of file DerivationStencils.h.

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

References s10f().

Here is the call graph for this function:



6.4.2.6 s10f() [2/2]

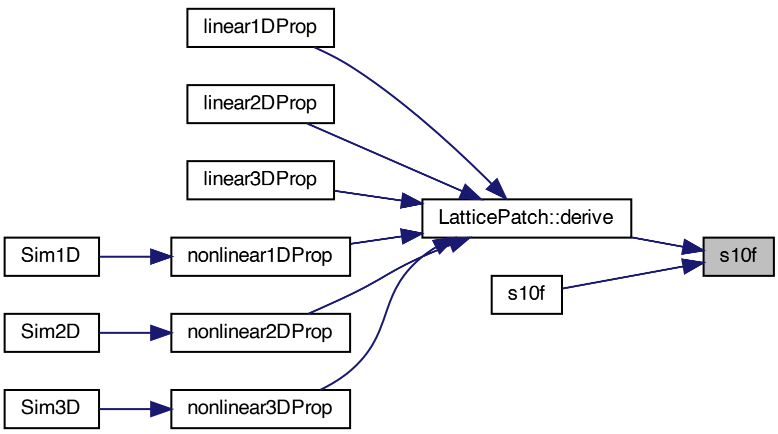
```
sunrealtype s10f (
            sunrealtype const * udata,
            const int dPD )  [inline]
```

Definition at line 152 of file DerivationStencils.h.

```
00152                                                                       {
00153   return 1.0 / 1260.0 * udata[-6 * dPD] - 1.0 / 105.0 * udata[-5 * dPD] +
00154          3.0 / 56.0 * udata[-4 * dPD] - 4.0 / 21.0 * udata[-3 * dPD] +
00155          1.0 / 2.0 * udata[-2 * dPD] - 6.0 / 5.0 * udata[-1 * dPD] +
00156          11.0 / 30.0 * udata[0] + 4.0 / 7.0 * udata[1 * dPD] -
00157          3.0 / 28.0 * udata[2 * dPD] + 1.0 / 63.0 * udata[3 * dPD] -
00158          1.0 / 840.0 * udata[4 * dPD];
00159 }
```

Referenced by LatticePatch::derive(), and s10f().

Here is the caller graph for this function:



6.4.2.7 s11b() [1/2]

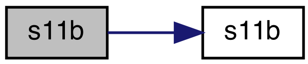
```
sunrealtype s11b (
            sunrealtype const * udata )  [inline]
```

Definition at line 279 of file DerivationStencils.h.

```
00280 { return s11b(udata, 6); }
```

References s11b().

Here is the call graph for this function:



6.4.2.8 s11b() [2/2]

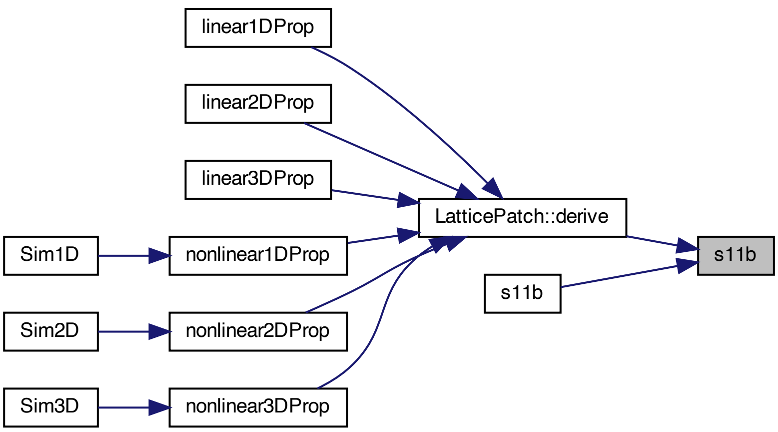
```
sunrealtype s11b (
            sunrealtype const * udata,
            const int dPD )  [inline]
```

Definition at line 187 of file DerivationStencils.h.

```
00187                                                        {
00188   return -1.0 / 2310.0 * udata[-5 * dPD] + 1.0 / 168.0 * udata[-4 * dPD] -
00189          5.0 / 126.0 * udata[-3 * dPD] + 5.0 / 28.0 * udata[-2 * dPD] -
00190          5.0 / 7.0 * udata[-1 * dPD] - 1.0 / 6.0 * udata[0] + udata[1 * dPD] -
00191          5.0 / 14.0 * udata[2 * dPD] + 5.0 / 42.0 * udata[3 * dPD] -
00192          5.0 / 168.0 * udata[4 * dPD] + 1.0 / 210.0 * udata[5 * dPD] -
00193          1.0 / 2772.0 * udata[6 * dPD];
00194 }
```

Referenced by LatticePatch::derive(), and s11b().

Here is the caller graph for this function:



6.4.2.9 s11f() [1/2]

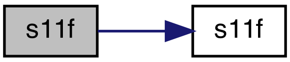
```
sunrealtype s11f (
            sunrealtype const * udata )  [inline]
```

Definition at line 277 of file DerivationStencils.h.

```
00278 { return s11f(udata, 6); }
```

References s11f().

Here is the call graph for this function:



6.4.2.10 s11f() [2/2]

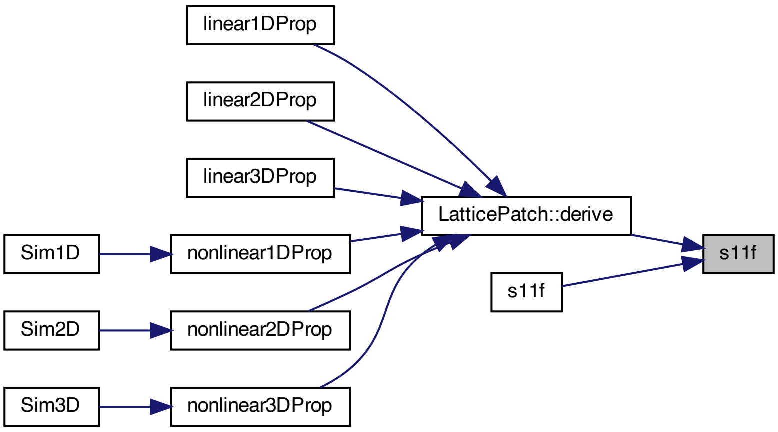
```
sunrealtype s11f (
            sunrealtype const * udata,
            const int dPD )  [inline]
```

Definition at line 178 of file DerivationStencils.h.

```
00178                                                                  {
00179   return 1.0 / 2772.0 * udata[-6 * dPD] - 1.0 / 210.0 * udata[-5 * dPD] +
00180          5.0 / 168.0 * udata[-4 * dPD] - 5.0 / 42.0 * udata[-3 * dPD] +
00181          5.0 / 14.0 * udata[-2 * dPD] - 1.0 / 1.0 * udata[-1 * dPD] +
00182          1.0 / 6.0 * udata[0] + 5.0 / 7.0 * udata[1 * dPD] -
00183          5.0 / 28.0 * udata[2 * dPD] + 5.0 / 126.0 * udata[3 * dPD] -
00184          1.0 / 168.0 * udata[4 * dPD] + 1.0 / 2310.0 * udata[5 * dPD];
00185 }
```

Referenced by LatticePatch::derive(), and s11f().

Here is the caller graph for this function:



6.4.2.11 s12b() [1/2]

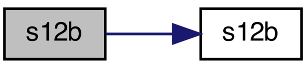
```
sunrealtype s12b (
            sunrealtype const * udata )  [inline]
```

Definition at line 285 of file DerivationStencils.h.

```
00286 { return s12b(udata, 6); }
```

References s12b().

Here is the call graph for this function:



6.4.2.12 s12b() [2/2]

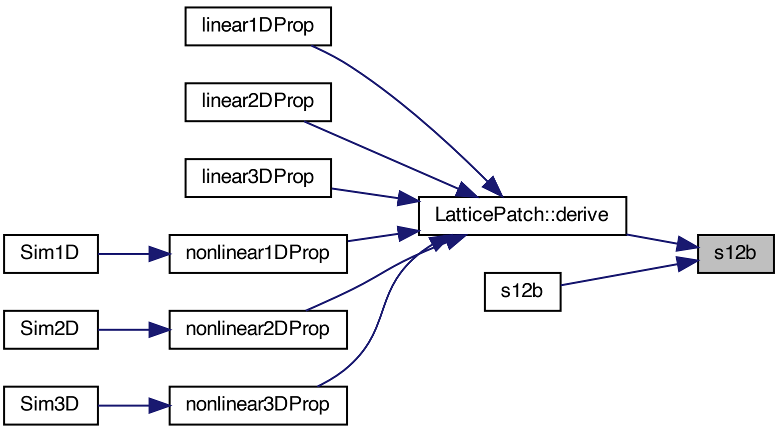
```
sunrealtype s12b (
            sunrealtype const * udata,
            const int dPD )  [inline]
```

Definition at line 215 of file DerivationStencils.h.

```
00215                                                                  {
00216   return -1.0 / 3960.0 * udata[-5 * dPD] + 1.0 / 264.0 * udata[-4 * dPD] -
00217          1.0 / 36.0 * udata[-3 * dPD] + 5.0 / 36.0 * udata[-2 * dPD] -
00218          5.0 / 8.0 * udata[-1 * dPD] - 13.0 / 42.0 * udata[0] +
00219          7.0 / 6.0 * udata[1 * dPD] - 1.0 / 2.0 * udata[2 * dPD] +
00220          5.0 / 24.0 * udata[3 * dPD] - 5.0 / 72.0 * udata[4 * dPD] +
00221          1.0 / 60.0 * udata[5 * dPD] - 1.0 / 396.0 * udata[6 * dPD] +
00222          1.0 / 5544.0 * udata[7 * dPD];
00223 }
```

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

Here is the caller graph for this function:



6.4.2.13 s12c() [1/2]

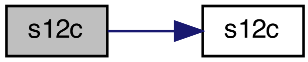
```
sunrealtype s12c (
            sunrealtype const * udata )  [inline]
```

Definition at line 283 of file DerivationStencils.h.

```
00284 { return s12c(udata, 6); }
```

References s12c().

Here is the call graph for this function:



6.4.2.14 s12c() [2/2]

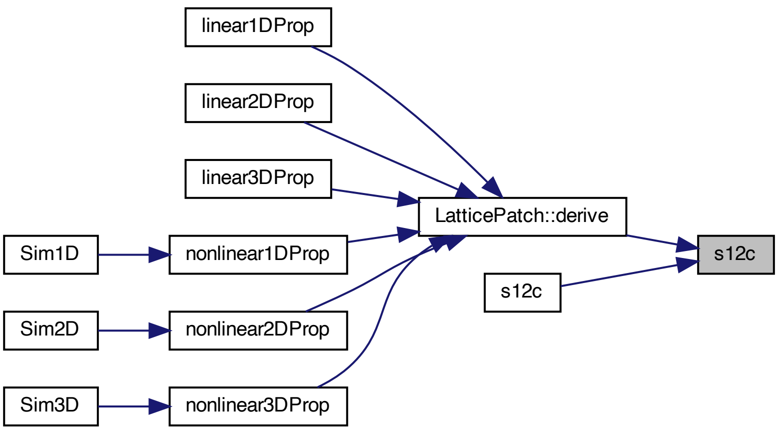
```
sunrealtype s12c (
            sunrealtype const * udata,
            const int dPD ) [inline]
```

Definition at line 206 of file DerivationStencils.h.

```
00206                                                                   {
00207   return 1.0 / 5544.0 * udata[-6 * dPD] - 1.0 / 385.0 * udata[-5 * dPD] +
00208          1.0 / 56.0 * udata[-4 * dPD] - 5.0 / 63.0 * udata[-3 * dPD] +
00209          15.0 / 56.0 * udata[-2 * dPD] - 6.0 / 7.0 * udata[-1 * dPD] + 0 +
00210          6.0 / 7.0 * udata[1 * dPD] - 15.0 / 56.0 * udata[2 * dPD] +
00211          5.0 / 63.0 * udata[3 * dPD] - 1.0 / 56.0 * udata[4 * dPD] +
00212          1.0 / 385.0 * udata[5 * dPD] - 1.0 / 5544.0 * udata[6 * dPD];
00213 }
```

Referenced by LatticePatch::derive(), and s12c().

Here is the caller graph for this function:



6.4.2.15 s12f() [1/2]

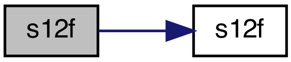
```
sunrealtype s12f (
            sunrealtype const * udata )  [inline]
```

Definition at line 281 of file DerivationStencils.h.

```
00282 { return s12f(udata, 6); }
```

References s12f().

Here is the call graph for this function:



6.4.2.16 s12f() [2/2]

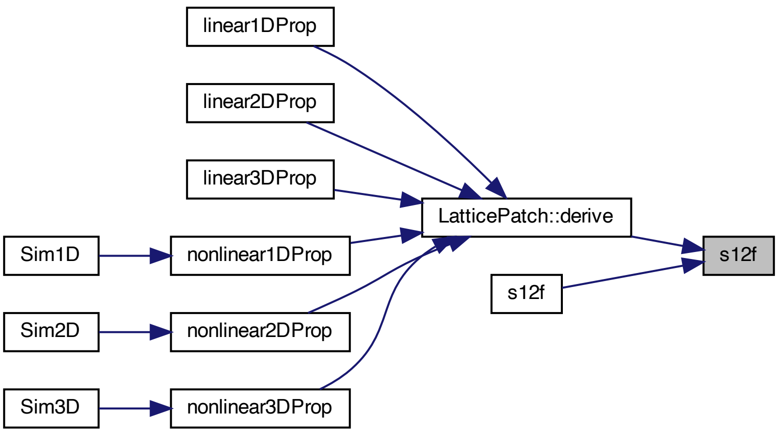
```
sunrealtype s12f (
            sunrealtype const * udata,
            const int dPD )  [inline]
```

Definition at line 196 of file DerivationStencils.h.

```
00196                                                                         {
00197   return -1.0 / 5544.0 * udata[-7 * dPD] + 1.0 / 396.0 * udata[-6 * dPD] -
00198          1.0 / 60.0 * udata[-5 * dPD] + 5.0 / 72.0 * udata[-4 * dPD] -
00199          5.0 / 24.0 * udata[-3 * dPD] + 1.0 / 2.0 * udata[-2 * dPD] -
00200          7.0 / 6.0 * udata[-1 * dPD] + 13.0 / 42.0 * udata[0] +
00201          5.0 / 8.0 * udata[1 * dPD] - 5.0 / 36.0 * udata[2 * dPD] +
00202          1.0 / 36.0 * udata[3 * dPD] - 1.0 / 264.0 * udata[4 * dPD] +
00203          1.0 / 3960.0 * udata[5 * dPD];
00204 }
```

Referenced by LatticePatch::derive(), and s12f().

Here is the caller graph for this function:



6.4.2.17 s13b() [1/2]

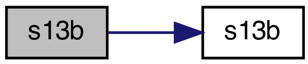
```
sunrealtype s13b (
            sunrealtype const * udata ) [inline]
```

Definition at line 289 of file DerivationStencils.h.

```
00290 { return s13b(udata, 6); }
```

References s13b().

Here is the call graph for this function:



6.4.2.18 s13b() [2/2]

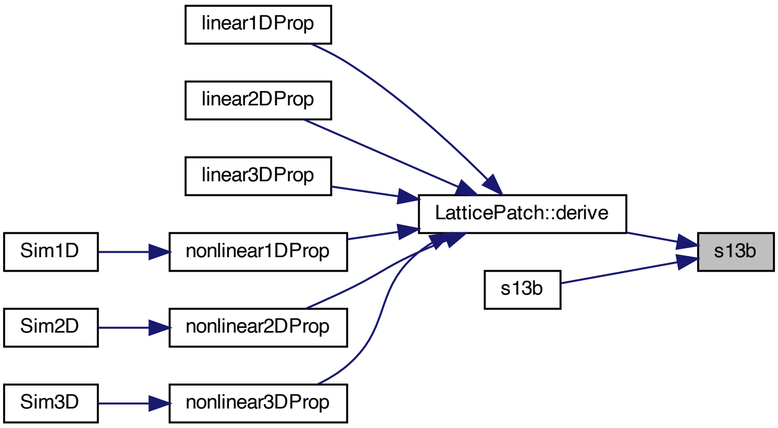
```
sunrealtype s13b (
            sunrealtype const * udata,
            const int dPD ) [inline]
```

Definition at line 235 of file DerivationStencils.h.

```
00235                                                                    {
00236   return 1.0 / 10296.0 * udata[-6 * dPD] - 1.0 / 660.0 * udata[-5 * dPD] +
00237          1.0 / 88.0 * udata[-4 * dPD] - 1.0 / 18.0 * udata[-3 * dPD] +
00238          5.0 / 24.0 * udata[-2 * dPD] - 3.0 / 4.0 * udata[-1 * dPD] -
00239          1.0 / 7.0 * udata[0] + udata[1 * dPD] - 3.0 / 8.0 * udata[2 * dPD] +
00240          5.0 / 36.0 * udata[3 * dPD] - 1.0 / 24.0 * udata[4 * dPD] +
00241          1.0 / 110.0 * udata[5 * dPD] - 1.0 / 792.0 * udata[6 * dPD] +
00242          1.0 / 12012.0 * udata[7 * dPD];
00243 }
```

Referenced by LatticePatch::derive(), and s13b().

Here is the caller graph for this function:



6.4.2.19 s13f() [1/2]

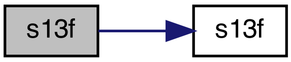
```
sunrealtype s13f (
            sunrealtype const * udata )  [inline]
```

Definition at line 287 of file DerivationStencils.h.

```
00288 { return s13f(udata, 6); }
```

References s13f().

Here is the call graph for this function:



6.4.2.20 s13f() [2/2]

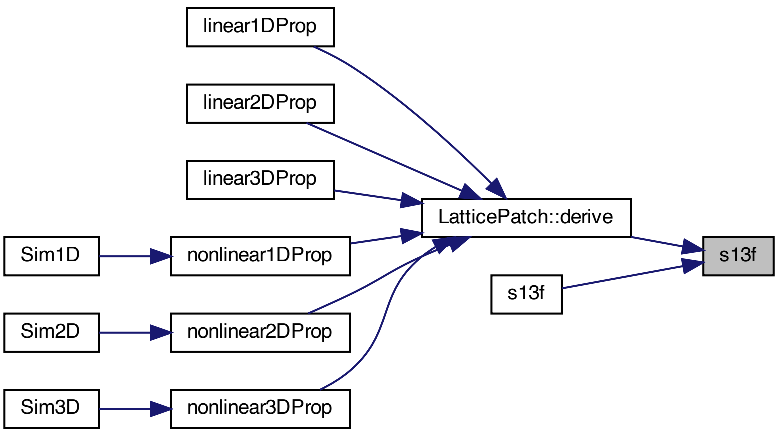
```
sunrealtype s13f (
            sunrealtype const * udata,
            const int dPD )  [inline]
```

Definition at line 225 of file DerivationStencils.h.

```
00225                                                            {
00226   return -1.0 / 12012.0 * udata[-7 * dPD] + 1.0 / 792.0 * udata[-6 * dPD] -
00227          1.0 / 110.0 * udata[-5 * dPD] + 1.0 / 24.0 * udata[-4 * dPD] -
00228          5.0 / 36.0 * udata[-3 * dPD] + 3.0 / 8.0 * udata[-2 * dPD] -
00229          1.0 / 1.0 * udata[-1 * dPD] + 1.0 / 7.0 * udata[0] +
00230          3.0 / 4.0 * udata[1 * dPD] - 5.0 / 24.0 * udata[2 * dPD] +
00231          1.0 / 18.0 * udata[3 * dPD] - 1.0 / 88.0 * udata[4 * dPD] +
00232          1.0 / 660.0 * udata[5 * dPD] - 1.0 / 10296.0 * udata[6 * dPD];
00233 }
```

Referenced by LatticePatch::derive(), and s13f().

Here is the caller graph for this function:



6.4.2.21 s1b() [1/2]

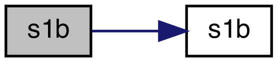
```
sunrealtype s1b (
            sunrealtype const * udata )  [inline]
```

Definition at line 250 of file DerivationStencils.h.

```
00250 { return s1b(udata, 6); }
```

References s1b().

Here is the call graph for this function:



6.4.2.22 s1b() [2/2]

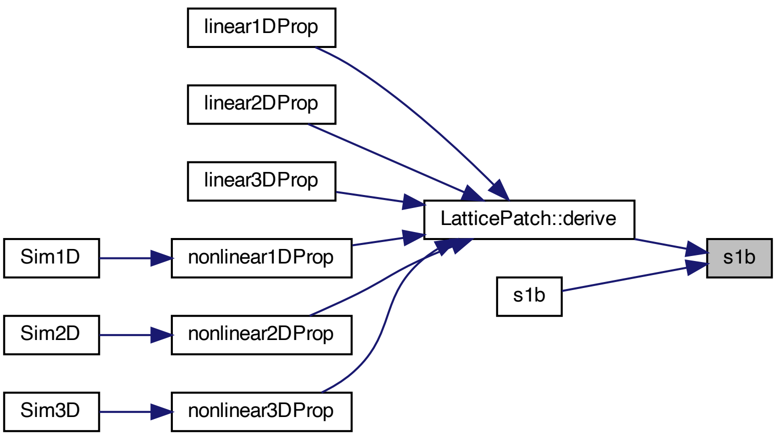
```
sunrealtype s1b (
            sunrealtype const * udata,
            const int dPD )  [inline]
```

Definition at line 21 of file DerivationStencils.h.

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

Referenced by LatticePatch::derive(), and s1b().

Here is the caller graph for this function:



6.4.2.23 s1f() [1/2]

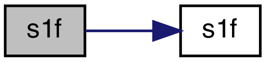
```
sunrealtype s1f (
            sunrealtype const * udata )  [inline]
```

Definition at line 249 of file DerivationStencils.h.

```
00249 { return s1f(udata, 6); }
```

References s1f().

Here is the call graph for this function:



6.4.2.24 s1f() [2/2]

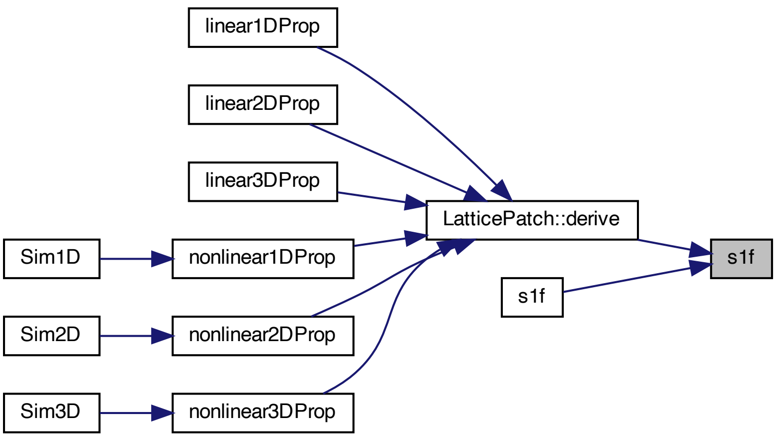
```
sunrealtype s1f (
            sunrealtype const * udata,
            const int dPD )  [inline]
```

Definition at line 16 of file DerivationStencils.h.

```
00016                                                          {
00017   return -1.0 / 1.0 * udata[-1 * dPD] + udata[0];
00018 }
```

Referenced by LatticePatch::derive(), and s1f().

Here is the caller graph for this function:



6.4.2.25 s2b() [1/2]

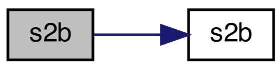
```
sunrealtype s2b (
            sunrealtype const * udata )  [inline]
```

Definition at line 253 of file DerivationStencils.h.

```
00253 { return s2b(udata, 6); }
```

References s2b().

Here is the call graph for this function:



6.4.2.26 s2b() [2/2]

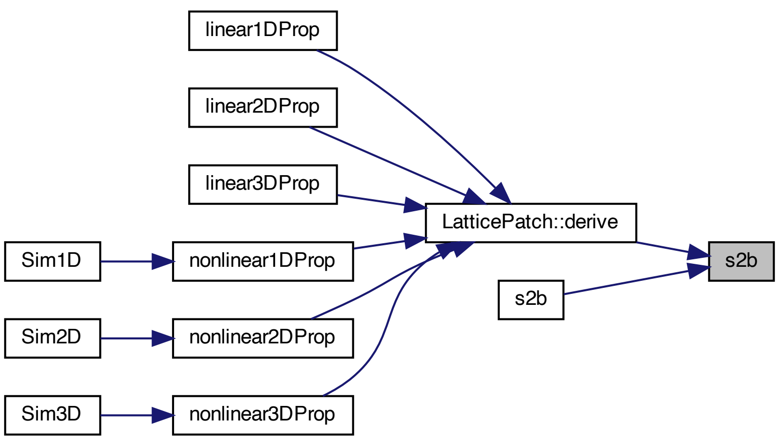
```
sunrealtype s2b (
            sunrealtype const * udata,
            const int dPD )  [inline]
```

Definition at line 35 of file DerivationStencils.h.

```
00035                                                              {
00036   return -3.0 / 2.0 * udata[0] + 2.0 / 1.0 * udata[1 * dPD] -
00037          1.0 / 2.0 * udata[2 * dPD];
00038 }
```

Referenced by LatticePatch::derive(), and s2b().

Here is the caller graph for this function:



6.4.2.27 s2c() [1/2]

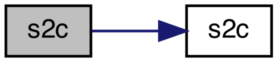
```
sunrealtype s2c (
            sunrealtype const * udata ) [inline]
```

Definition at line 252 of file DerivationStencils.h.

```
00252 { return s2c(udata, 6); }
```

References s2c().

Here is the call graph for this function:



6.4.2.28 s2c() [2/2]

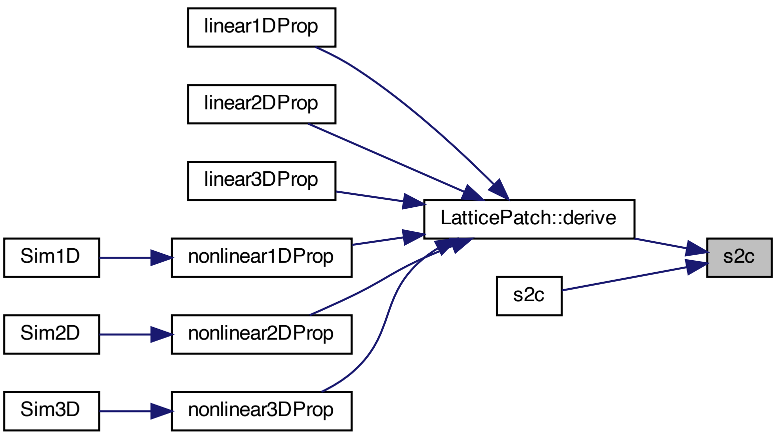
```
sunrealtype s2c (
            sunrealtype const * udata,
            const int dPD ) [inline]
```

Definition at line 31 of file DerivationStencils.h.

```
00031                                                         {
00032   return -1.0 / 2.0 * udata[-1 * dPD] + 0 + 1.0 / 2.0 * udata[1 * dPD];
00033 }
```

Referenced by LatticePatch::derive(), and s2c().

Here is the caller graph for this function:



6.4.2.29 s2f() [1/2]

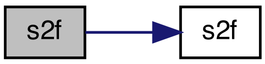
```
sunrealtype s2f (
            sunrealtype const * udata )  [inline]
```

Definition at line 251 of file DerivationStencils.h.

```
00251 { return s2f(udata, 6); }
```

References s2f().

Here is the call graph for this function:



6.4.2.30 s2f() [2/2]

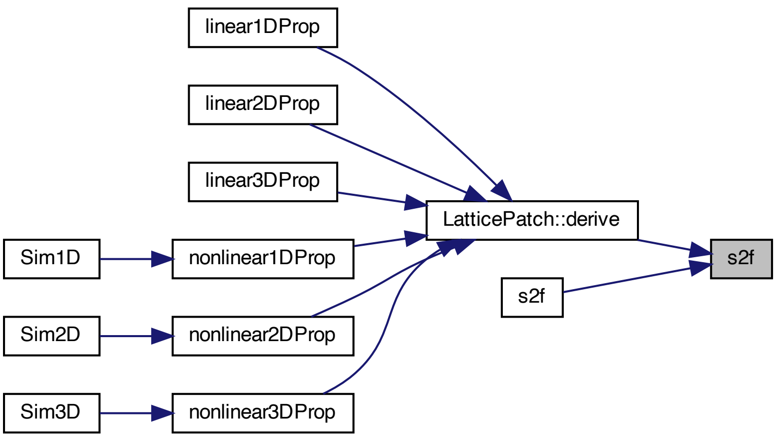
```
sunrealtype s2f (
            sunrealtype const * udata,
            const int dPD )  [inline]
```

Definition at line 26 of file DerivationStencils.h.

```
00026                                                       {
00027   return 1.0 / 2.0 * udata[-2 * dPD] - 2.0 / 1.0 * udata[-1 * dPD] +
00028          3.0 / 2.0 * udata[0];
00029 }
```

Referenced by LatticePatch::derive(), and s2f().

Here is the caller graph for this function:



6.4.2.31 s3b() [1/2]

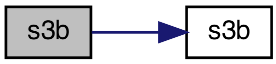
```
sunrealtype s3b (
            sunrealtype const * udata )  [inline]
```

Definition at line 255 of file DerivationStencils.h.

```
00255 { return s3b(udata, 6); }
```

References s3b().

Here is the call graph for this function:



6.4.2.32 s3b() [2/2]

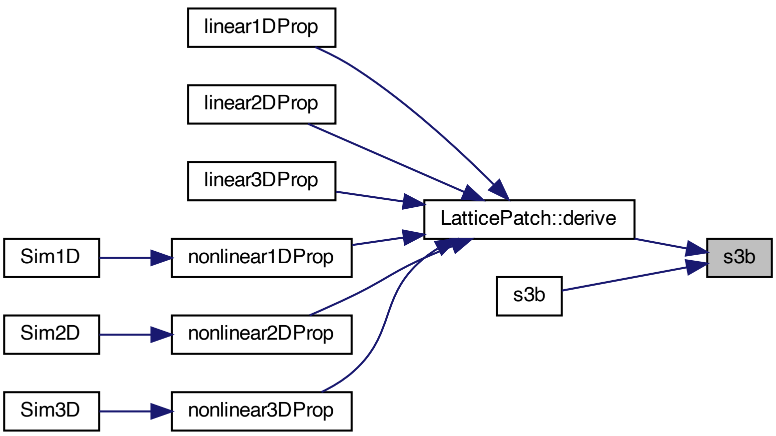
```
sunrealtype s3b (
            sunrealtype const * udata,
            const int dPD )  [inline]
```

Definition at line 45 of file DerivationStencils.h.

```
00045                                                      {
00046   return -1.0 / 3.0 * udata[-1 * dPD] - 1.0 / 2.0 * udata[0] + udata[1 * dPD] -
00047          1.0 / 6.0 * udata[2 * dPD];
00048 }
```

Referenced by LatticePatch::derive(), and s3b().

Here is the caller graph for this function:



6.4.2.33 s3f() [1/2]

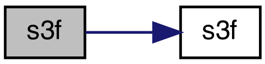
```
sunrealtype s3f (
             sunrealtype const * udata )  [inline]
```

Definition at line 254 of file DerivationStencils.h.

```
00254 { return s3f(udata, 6); }
```

References s3f().

Here is the call graph for this function:



6.4.2.34 s3f() [2/2]

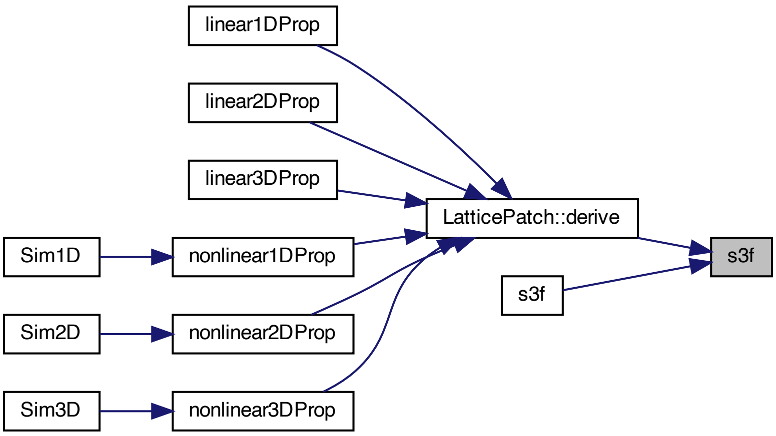
```
sunrealtype s3f (
             sunrealtype const * udata,
             const int dPD )  [inline]
```

Definition at line 40 of file DerivationStencils.h.

```
00040                                                        {
00041   return 1.0 / 6.0 * udata[-2 * dPD] - 1.0 / 1.0 * udata[-1 * dPD] +
00042          1.0 / 2.0 * udata[0] + 1.0 / 3.0 * udata[1 * dPD];
00043 }
```

Referenced by LatticePatch::derive(), and s3f().

Here is the caller graph for this function:



6.4.2.35 s4b() [1/2]

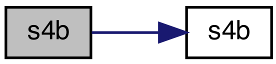
```
sunrealtype s4b (
            sunrealtype const * udata )  [inline]
```

Definition at line 258 of file DerivationStencils.h.

```
00258 { return s4b(udata, 6); }
```

References s4b().

Here is the call graph for this function:



6.4.2.36 s4b() [2/2]

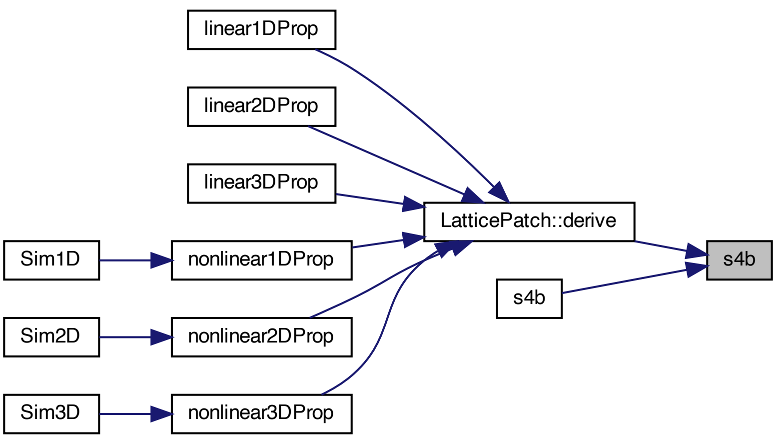
```
sunrealtype s4b (
            sunrealtype const * udata,
            const int dPD )  [inline]
```

Definition at line 61 of file DerivationStencils.h.

```
00061                                                          {
00062   return -1.0 / 4.0 * udata[-1 * dPD] - 5.0 / 6.0 * udata[0] +
00063          3.0 / 2.0 * udata[1 * dPD] - 1.0 / 2.0 * udata[2 * dPD] +
00064          1.0 / 12.0 * udata[3 * dPD];
00065 }
```

Referenced by LatticePatch::derive(), and s4b().

Here is the caller graph for this function:



6.4.2.37 s4c() [1/2]

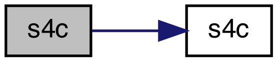
```
sunrealtype s4c (
            sunrealtype const * udata )  [inline]
```

Definition at line 257 of file DerivationStencils.h.

```
00257 { return s4c(udata, 6); }
```

References s4c().

Here is the call graph for this function:



6.4.2.38 s4c() [2/2]

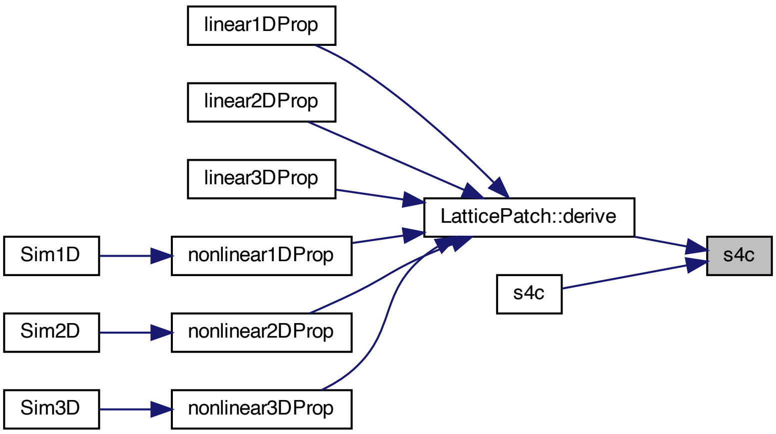
```
sunrealtype s4c (
            sunrealtype const * udata,
            const int dPD )  [inline]
```

Definition at line 56 of file DerivationStencils.h.

```
00056                                                        {
00057   return 1.0 / 12.0 * udata[-2 * dPD] - 2.0 / 3.0 * udata[-1 * dPD] + 0 +
00058          2.0 / 3.0 * udata[1 * dPD] - 1.0 / 12.0 * udata[2 * dPD];
00059 }
```

Referenced by LatticePatch::derive(), and s4c().

Here is the caller graph for this function:



6.4.2.39 s4f() [1/2]

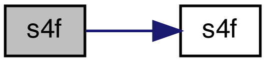
```
sunrealtype s4f (
            sunrealtype const * udata )  [inline]
```

Definition at line 256 of file DerivationStencils.h.

```
00256 { return s4f(udata, 6); }
```

References s4f().

Here is the call graph for this function:



6.4.2.40 s4f() [2/2]

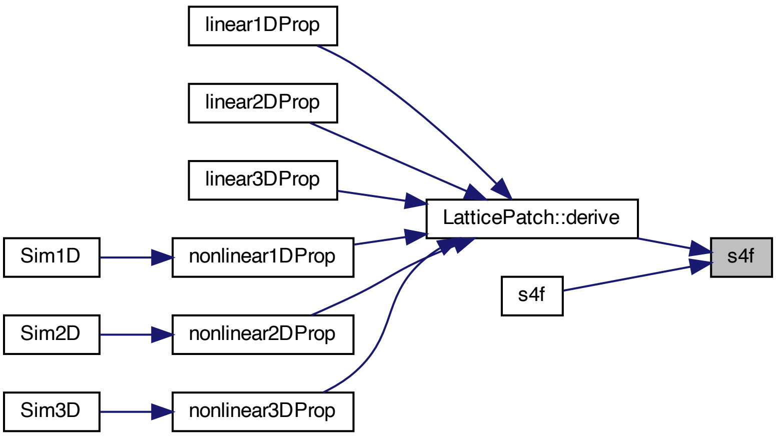
```
sunrealtype s4f (
            sunrealtype const * udata,
            const int dPD )  [inline]
```

Definition at line 50 of file DerivationStencils.h.

```
00050                                                        {
00051   return -1.0 / 12.0 * udata[-3 * dPD] + 1.0 / 2.0 * udata[-2 * dPD] -
00052          3.0 / 2.0 * udata[-1 * dPD] + 5.0 / 6.0 * udata[0] +
00053          1.0 / 4.0 * udata[1 * dPD];
00054 }
```

Referenced by LatticePatch::derive(), and s4f().

Here is the caller graph for this function:



6.4.2.41 s5b() [1/2]

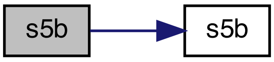
```
sunrealtype s5b (
            sunrealtype const * udata )  [inline]
```

Definition at line 260 of file DerivationStencils.h.

```
00260 { return s5b(udata, 6); }
```

References s5b().

Here is the call graph for this function:



6.4.2.42 s5b() [2/2]

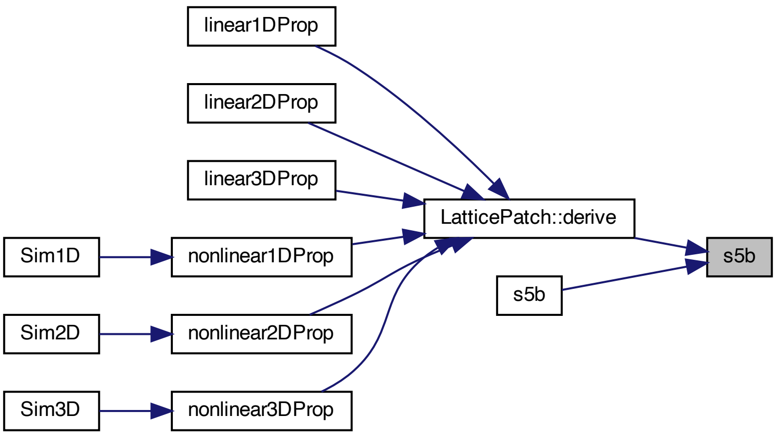
```
sunrealtype s5b (
            sunrealtype const * udata,
            const int dPD )  [inline]
```

Definition at line 73 of file DerivationStencils.h.

```
00073                                                            {
00074   return 1.0 / 20.0 * udata[-2 * dPD] - 1.0 / 2.0 * udata[-1 * dPD] -
00075          1.0 / 3.0 * udata[0] + udata[1 * dPD] - 1.0 / 4.0 * udata[2 * dPD] +
00076          1.0 / 30.0 * udata[3 * dPD];
00077 }
```

Referenced by LatticePatch::derive(), and s5b().

Here is the caller graph for this function:



6.4.2.43 s5f() [1/2]

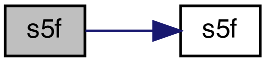
```
sunrealtype s5f (
            sunrealtype const * udata )  [inline]
```

Definition at line 259 of file DerivationStencils.h.

```
00259 { return s5f(udata, 6); }
```

References s5f().

Here is the call graph for this function:



6.4.2.44 s5f() [2/2]

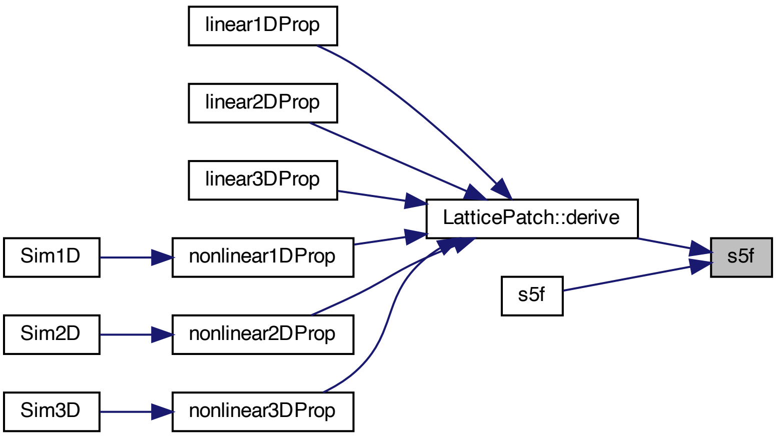
```
sunrealtype s5f (
            sunrealtype const * udata,
            const int dPD )  [inline]
```

Definition at line 67 of file DerivationStencils.h.

```
00067                                                          {
00068   return -1.0 / 30.0 * udata[-3 * dPD] + 1.0 / 4.0 * udata[-2 * dPD] -
00069          1.0 / 1.0 * udata[-1 * dPD] + 1.0 / 3.0 * udata[0] +
00070          1.0 / 2.0 * udata[1 * dPD] - 1.0 / 20.0 * udata[2 * dPD];
00071 }
```

Referenced by LatticePatch::derive(), and s5f().

Here is the caller graph for this function:



6.4.2.45 s6b() [1/2]

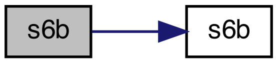
```
sunrealtype s6b (
            sunrealtype const * udata )  [inline]
```

Definition at line 263 of file DerivationStencils.h.

```
00263 { return s6b(udata, 6); }
```

References s6b().

Here is the call graph for this function:



6.4.2.46 s6b() [2/2]

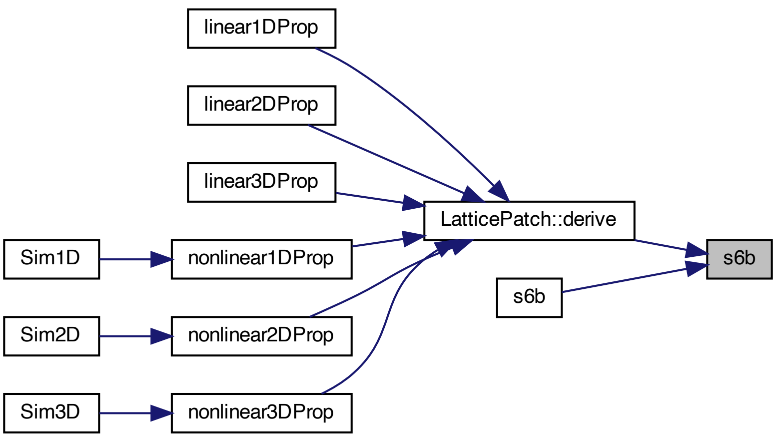
```
sunrealtype s6b (
            sunrealtype const * udata,
            const int dPD )  [inline]
```

Definition at line 92 of file DerivationStencils.h.

```
00092                                                   {
00093   return 1.0 / 30.0 * udata[-2 * dPD] - 2.0 / 5.0 * udata[-1 * dPD] -
00094          7.0 / 12.0 * udata[0] + 4.0 / 3.0 * udata[1 * dPD] -
00095          1.0 / 2.0 * udata[2 * dPD] + 2.0 / 15.0 * udata[3 * dPD] -
00096          1.0 / 60.0 * udata[4 * dPD];
00097 }
```

Referenced by LatticePatch::derive(), and s6b().

Here is the caller graph for this function:



6.4.2.47 s6c() [1/2]

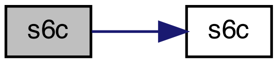
```
sunrealtype s6c (
            sunrealtype const * udata )  [inline]
```

Definition at line 262 of file DerivationStencils.h.

```
00262 { return s6c(udata, 6); }
```

References s6c().

Here is the call graph for this function:



6.4.2.48 s6c() [2/2]

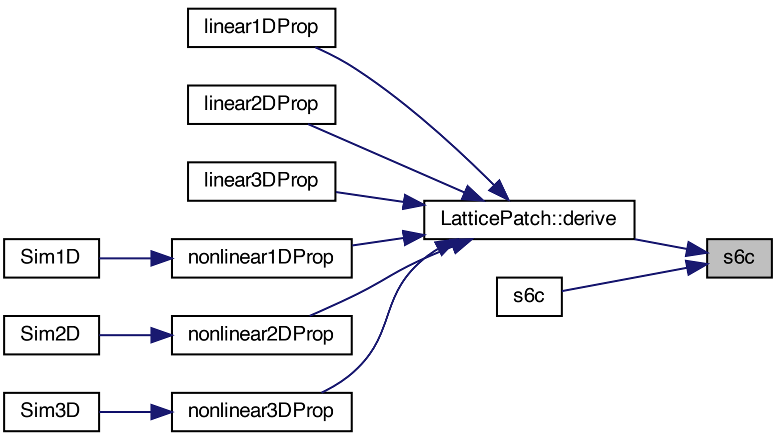
```
sunrealtype s6c (
            sunrealtype const * udata,
            const int dPD )  [inline]
```

Definition at line 86 of file DerivationStencils.h.

```
00086                                                                {
00087   return -1.0 / 60.0 * udata[-3 * dPD] + 3.0 / 20.0 * udata[-2 * dPD] -
00088          3.0 / 4.0 * udata[-1 * dPD] + 0 + 3.0 / 4.0 * udata[1 * dPD] -
00089          3.0 / 20.0 * udata[2 * dPD] + 1.0 / 60.0 * udata[3 * dPD];
00090 }
```

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

Here is the caller graph for this function:



6.4.2.49 s6f() [1/2]

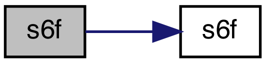
```
sunrealtype s6f (
            sunrealtype const * udata )  [inline]
```

Definition at line 261 of file DerivationStencils.h.

```
00261 { return s6f(udata, 6); }
```

References s6f().

Here is the call graph for this function:



6.4.2.50 s6f() [2/2]

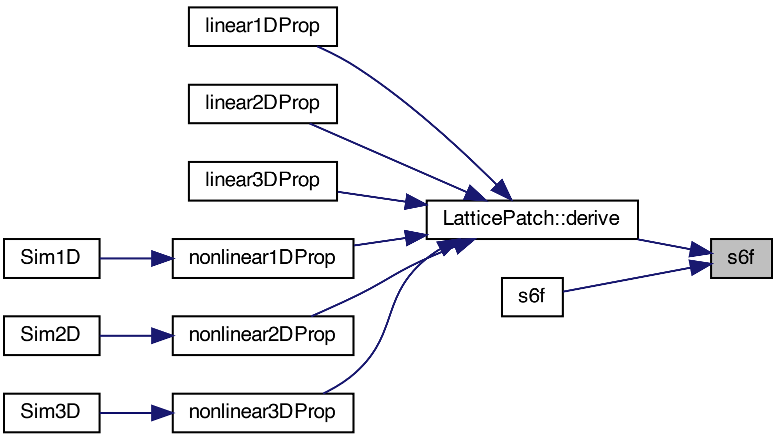
```
sunrealtype s6f (
            sunrealtype const * udata,
            const int dPD )  [inline]
```

Definition at line 79 of file DerivationStencils.h.

```
00079                                                        {
00080   return 1.0 / 60.0 * udata[-4 * dPD] - 2.0 / 15.0 * udata[-3 * dPD] +
00081          1.0 / 2.0 * udata[-2 * dPD] - 4.0 / 3.0 * udata[-1 * dPD] +
00082          7.0 / 12.0 * udata[0] + 2.0 / 5.0 * udata[1 * dPD] -
00083          1.0 / 30.0 * udata[2 * dPD];
00084 }
```

Referenced by LatticePatch::derive(), and s6f().

Here is the caller graph for this function:



6.4.2.51 s7b() [1/2]

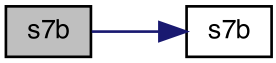
```
sunrealtype s7b (
              sunrealtype const * udata )  [inline]
```

Definition at line 265 of file DerivationStencils.h.

```
00265 { return s7b(udata, 6); }
```

References s7b().

Here is the call graph for this function:



6.4.2.52 s7b() [2/2]

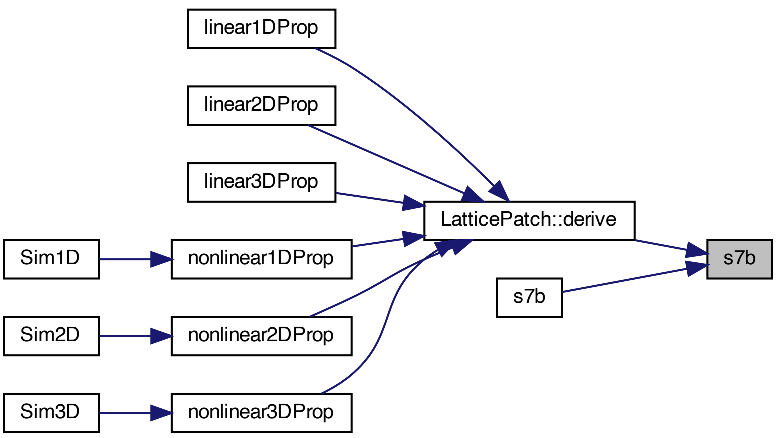
```
sunrealtype s7b (
              sunrealtype const * udata,
              const int dPD )  [inline]
```

Definition at line 106 of file DerivationStencils.h.

```
00106                                                                  {
00107   return -1.0 / 105.0 * udata[-3 * dPD] + 1.0 / 10.0 * udata[-2 * dPD] -
00108          3.0 / 5.0 * udata[-1 * dPD] - 1.0 / 4.0 * udata[0] + udata[1 * dPD] -
00109          3.0 / 10.0 * udata[2 * dPD] + 1.0 / 15.0 * udata[3 * dPD] -
00110          1.0 / 140.0 * udata[4 * dPD];
00111 }
```

Referenced by LatticePatch::derive(), and s7b().

Here is the caller graph for this function:



6.4.2.53 s7f() [1/2]

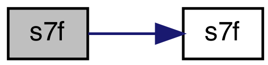
```
sunrealtype s7f (
        sunrealtype const * udata ) [inline]
```

Definition at line 264 of file DerivationStencils.h.

```
00264 { return s7f(udata, 6); }
```

References s7f().

Here is the call graph for this function:



6.4.2.54 s7f() [2/2]

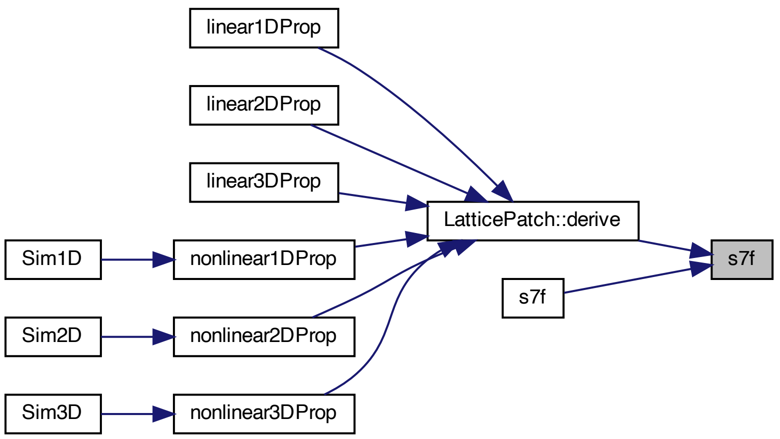
```
sunrealtype s7f (
        sunrealtype const * udata,
        const int dPD ) [inline]
```

Definition at line 99 of file DerivationStencils.h.

```
00099                                                                  {
00100   return 1.0 / 140.0 * udata[-4 * dPD] - 1.0 / 15.0 * udata[-3 * dPD] +
00101          3.0 / 10.0 * udata[-2 * dPD] - 1.0 / 1.0 * udata[-1 * dPD] +
00102          1.0 / 4.0 * udata[0] + 3.0 / 5.0 * udata[1 * dPD] -
00103          1.0 / 10.0 * udata[2 * dPD] + 1.0 / 105.0 * udata[3 * dPD];
00104 }
```

Referenced by LatticePatch::derive(), and s7f().

Here is the caller graph for this function:



6.4.2.55 s8b() [1/2]

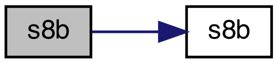
```
sunrealtype s8b (
            sunrealtype const * udata )  [inline]
```

Definition at line 268 of file DerivationStencils.h.

```
00268 { return s8b(udata, 6); }
```

References s8b().

Here is the call graph for this function:



6.4.2.56 s8b() [2/2]

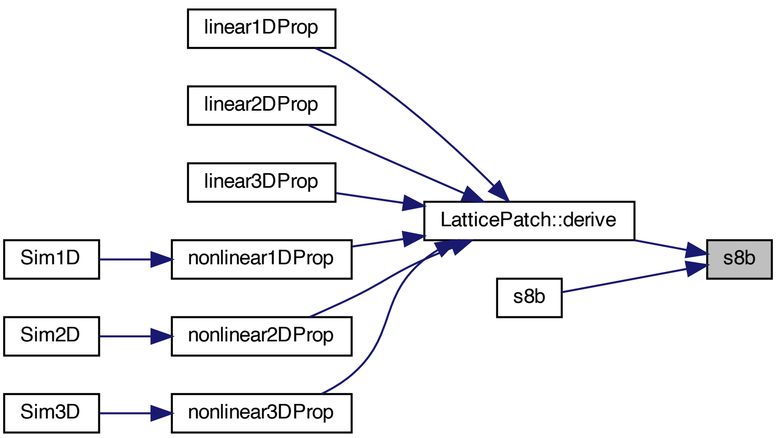
```
sunrealtype s8b (
            sunrealtype const * udata,
            const int dPD )  [inline]
```

Definition at line 128 of file DerivationStencils.h.

```
00128                                                          {
00129   return -1.0 / 168.0 * udata[-3 * dPD] + 1.0 / 14.0 * udata[-2 * dPD] -
00130          1.0 / 2.0 * udata[-1 * dPD] - 9.0 / 20.0 * udata[0] +
00131          5.0 / 4.0 * udata[1 * dPD] - 1.0 / 2.0 * udata[2 * dPD] +
00132          1.0 / 6.0 * udata[3 * dPD] - 1.0 / 28.0 * udata[4 * dPD] +
00133          1.0 / 280.0 * udata[5 * dPD];
00134 }
```

Referenced by LatticePatch::derive(), and s8b().

Here is the caller graph for this function:



6.4.2.57 s8c() [1/2]

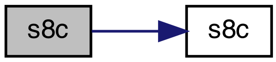
```
sunrealtype s8c (
            sunrealtype const * udata )  [inline]
```

Definition at line 267 of file DerivationStencils.h.

```
00267 { return s8c(udata, 6); }
```

References s8c().

Here is the call graph for this function:



6.4.2.58 s8c() [2/2]

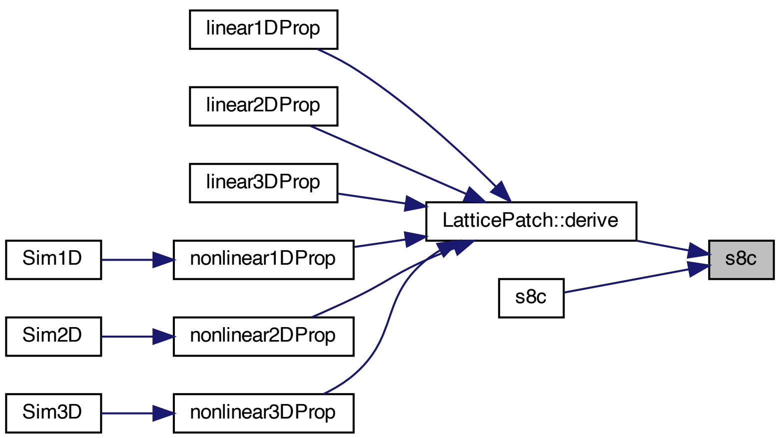
```
sunrealtype s8c (
            sunrealtype const * udata,
            const int dPD )  [inline]
```

Definition at line 121 of file DerivationStencils.h.

```
00121                                                                           {
00122   return 1.0 / 280.0 * udata[-4 * dPD] - 4.0 / 105.0 * udata[-3 * dPD] +
00123          1.0 / 5.0 * udata[-2 * dPD] - 4.0 / 5.0 * udata[-1 * dPD] + 0 +
00124          4.0 / 5.0 * udata[1 * dPD] - 1.0 / 5.0 * udata[2 * dPD] +
00125          4.0 / 105.0 * udata[3 * dPD] - 1.0 / 280.0 * udata[4 * dPD];
00126 }
```

Referenced by LatticePatch::derive(), and s8c().

Here is the caller graph for this function:



6.4.2.59 s8f() [1/2]

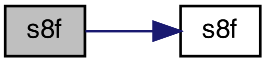
```
sunrealtype s8f (
            sunrealtype const * udata )  [inline]
```

Definition at line 266 of file DerivationStencils.h.

```
00266 { return s8f(udata, 6); }
```

References s8f().

Here is the call graph for this function:



6.4.2.60 s8f() [2/2]

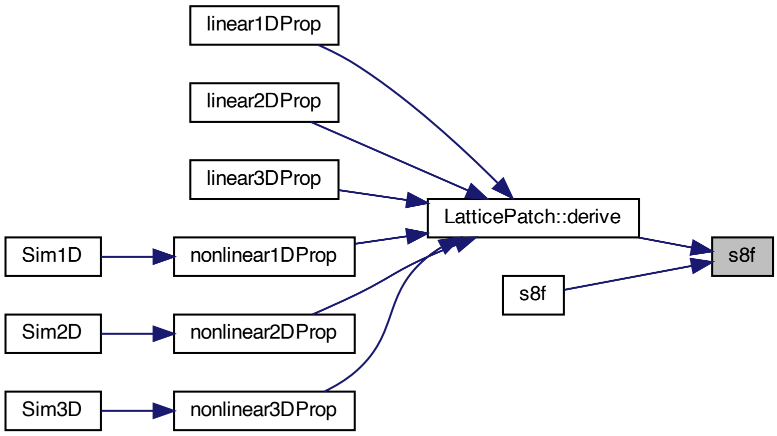
```
sunrealtype s8f (
            sunrealtype const * udata,
            const int dPD )  [inline]
```

Definition at line 113 of file DerivationStencils.h.

```
00113                                                                       {
00114   return -1.0 / 280.0 * udata[-5 * dPD] + 1.0 / 28.0 * udata[-4 * dPD] -
00115          1.0 / 6.0 * udata[-3 * dPD] + 1.0 / 2.0 * udata[-2 * dPD] -
00116          5.0 / 4.0 * udata[-1 * dPD] + 9.0 / 20.0 * udata[0] +
00117          1.0 / 2.0 * udata[1 * dPD] - 1.0 / 14.0 * udata[2 * dPD] +
00118          1.0 / 168.0 * udata[3 * dPD];
00119 }
```

Referenced by LatticePatch::derive(), and s8f().

Here is the caller graph for this function:



6.4.2.61 s9b() [1/2]

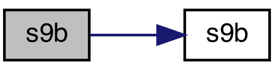
```
sunrealtype s9b (
              sunrealtype const * udata )  [inline]
```

Definition at line 270 of file DerivationStencils.h.

```
00270 { return s9b(udata, 6); }
```

References s9b().

Here is the call graph for this function:



6.4.2.62 s9b() [2/2]

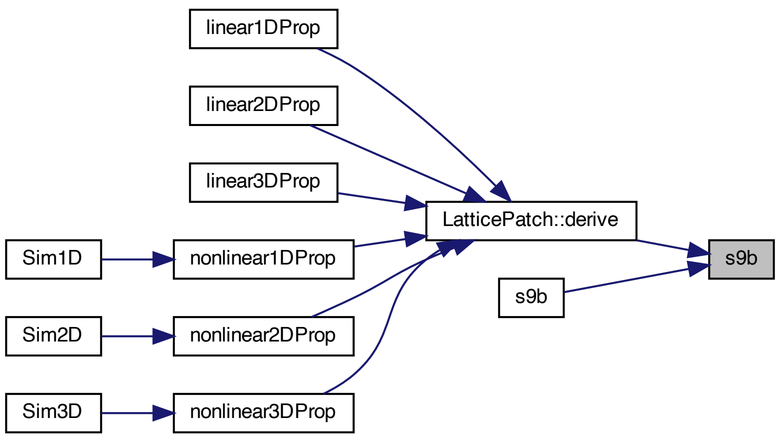
```
sunrealtype s9b (
            sunrealtype const * udata,
            const int dPD )  [inline]
```

Definition at line 144 of file DerivationStencils.h.

```
00144                                                                    {
00145   return 1.0 / 504.0 * udata[-4 * dPD] - 1.0 / 42.0 * udata[-3 * dPD] +
00146          1.0 / 7.0 * udata[-2 * dPD] - 2.0 / 3.0 * udata[-1 * dPD] -
00147          1.0 / 5.0 * udata[0] + udata[1 * dPD] - 1.0 / 3.0 * udata[2 * dPD] +
00148          2.0 / 21.0 * udata[3 * dPD] - 1.0 / 56.0 * udata[4 * dPD] +
00149          1.0 / 630.0 * udata[5 * dPD];
00150 }
```

Referenced by LatticePatch::derive(), and s9b().

Here is the caller graph for this function:



6.4.2.63 s9f() [1/2]

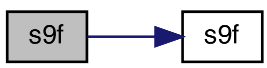
```
sunrealtype s9f (
            sunrealtype const * udata )  [inline]
```

Definition at line 269 of file DerivationStencils.h.

```
00269 { return s9f(udata, 6); }
```

References s9f().

Here is the call graph for this function:



6.4.2.64 s9f() [2/2]

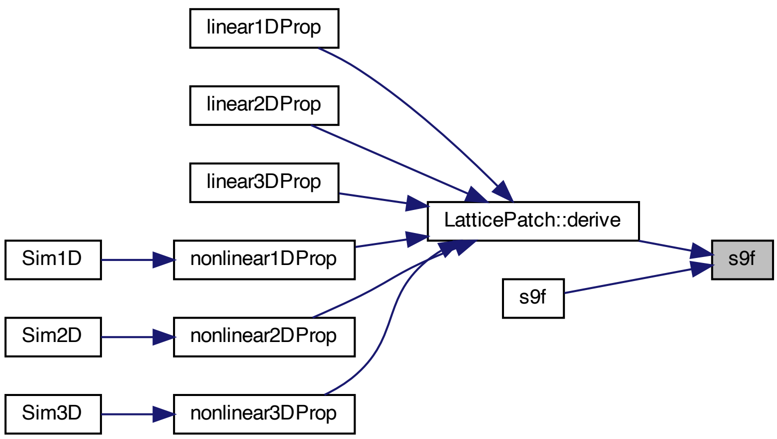
```
sunrealtype s9f (
            sunrealtype const * udata,
            const int dPD ) [inline]
```

Definition at line 136 of file DerivationStencils.h.

```
00136                                                         {
00137   return -1.0 / 630.0 * udata[-5 * dPD] + 1.0 / 56.0 * udata[-4 * dPD] -
00138          2.0 / 21.0 * udata[-3 * dPD] + 1.0 / 3.0 * udata[-2 * dPD] -
00139          1.0 / 1.0 * udata[-1 * dPD] + 1.0 / 5.0 * udata[0] +
00140          2.0 / 3.0 * udata[1 * dPD] - 1.0 / 7.0 * udata[2 * dPD] +
00141          1.0 / 42.0 * udata[3 * dPD] - 1.0 / 504.0 * udata[4 * dPD];
00142 }
```

Referenced by LatticePatch::derive(), and s9f().

Here is the caller graph for this function:



6.5 DerivationStencils.h

Go to the documentation of this file.

```
///////////////////////////////////////////////////////////////////
/// @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 dPD -- data point dimension //
/////////////////////////////////////////////////////////

// Downwind (forward) dfferentiating
#pragma omp declare simd uniform(dPD) notinbranch
inline sunrealtype s1f(sunrealtype const *udata, const int dPD) {
  return -1.0 / 1.0 * udata[-1 * dPD] + udata[0];
}
```

```
// Upwind (backward) differentiating
#pragma omp declare simd uniform(dPD) notinbranch
inline sunrealtype s1b(sunrealtype const *udata, const int dPD) {
  return -1.0 / 1.0 * udata[0] + udata[1 * dPD];
}

#pragma omp declare simd uniform(dPD) notinbranch
inline sunrealtype s2f(sunrealtype const *udata, const int dPD) {
  return 1.0 / 2.0 * udata[-2 * dPD] - 2.0 / 1.0 * udata[-1 * dPD] +
         3.0 / 2.0 * udata[0];
}
#pragma omp declare simd uniform(dPD) notinbranch
inline sunrealtype s2c(sunrealtype const *udata, const int dPD) {
  return -1.0 / 2.0 * udata[-1 * dPD] + 0 + 1.0 / 2.0 * udata[1 * dPD];
}
#pragma omp declare simd uniform(dPD) notinbranch
inline sunrealtype s2b(sunrealtype const *udata, const int dPD) {
  return -3.0 / 2.0 * udata[0] + 2.0 / 1.0 * udata[1 * dPD] -
         1.0 / 2.0 * udata[2 * dPD];
}
#pragma omp declare simd uniform(dPD) notinbranch
inline sunrealtype s3f(sunrealtype const *udata, const int dPD) {
  return 1.0 / 6.0 * udata[-2 * dPD] - 1.0 / 1.0 * udata[-1 * dPD] +
         1.0 / 2.0 * udata[0] + 1.0 / 3.0 * udata[1 * dPD];
}
#pragma omp declare simd uniform(dPD) notinbranch
inline sunrealtype s3b(sunrealtype const *udata, const int dPD) {
  return -1.0 / 3.0 * udata[-1 * dPD] - 1.0 / 2.0 * udata[0] + udata[1 * dPD] -
         1.0 / 6.0 * udata[2 * dPD];
}
#pragma omp declare simd uniform(dPD) notinbranch
inline sunrealtype s4f(sunrealtype const *udata, const int dPD) {
  return -1.0 / 12.0 * udata[-3 * dPD] + 1.0 / 2.0 * udata[-2 * dPD] -
         3.0 / 2.0 * udata[-1 * dPD] + 5.0 / 6.0 * udata[0] +
         1.0 / 4.0 * udata[1 * dPD];
}
#pragma omp declare simd uniform(dPD) notinbranch
inline sunrealtype s4c(sunrealtype const *udata, const int dPD) {
  return 1.0 / 12.0 * udata[-2 * dPD] - 2.0 / 3.0 * udata[-1 * dPD] + 0 +
         2.0 / 3.0 * udata[1 * dPD] - 1.0 / 12.0 * udata[2 * dPD];
}
#pragma omp declare simd uniform(dPD) notinbranch
inline sunrealtype s4b(sunrealtype const *udata, const int dPD) {
  return -1.0 / 4.0 * udata[-1 * dPD] - 5.0 / 6.0 * udata[0] +
         3.0 / 2.0 * udata[1 * dPD] - 1.0 / 2.0 * udata[2 * dPD] +
         1.0 / 12.0 * udata[3 * dPD];
}
#pragma omp declare simd uniform(dPD) notinbranch
inline sunrealtype s5f(sunrealtype const *udata, const int dPD) {
  return -1.0 / 30.0 * udata[-3 * dPD] + 1.0 / 4.0 * udata[-2 * dPD] -
         1.0 / 1.0 * udata[-1 * dPD] + 1.0 / 3.0 * udata[0] +
         1.0 / 2.0 * udata[1 * dPD] - 1.0 / 20.0 * udata[2 * dPD];
}
#pragma omp declare simd uniform(dPD) notinbranch
inline sunrealtype s5b(sunrealtype const *udata, const int dPD) {
  return 1.0 / 20.0 * udata[-2 * dPD] - 1.0 / 2.0 * udata[-1 * dPD] -
         1.0 / 3.0 * udata[0] + udata[1 * dPD] - 1.0 / 4.0 * udata[2 * dPD] +
         1.0 / 30.0 * udata[3 * dPD];
}
#pragma omp declare simd uniform(dPD) notinbranch
inline sunrealtype s6f(sunrealtype const *udata, const int dPD) {
  return 1.0 / 60.0 * udata[-4 * dPD] - 2.0 / 15.0 * udata[-3 * dPD] +
         1.0 / 2.0 * udata[-2 * dPD] - 4.0 / 3.0 * udata[-1 * dPD] +
         7.0 / 12.0 * udata[0] + 2.0 / 5.0 * udata[1 * dPD] -
         1.0 / 30.0 * udata[2 * dPD];
}
#pragma omp declare simd uniform(dPD) notinbranch
inline sunrealtype s6c(sunrealtype const *udata, const int dPD) {
  return -1.0 / 60.0 * udata[-3 * dPD] + 3.0 / 20.0 * udata[-2 * dPD] -
         3.0 / 4.0 * udata[-1 * dPD] + 0 + 3.0 / 4.0 * udata[1 * dPD] -
         3.0 / 20.0 * udata[2 * dPD] + 1.0 / 60.0 * udata[3 * dPD];
}
#pragma omp declare simd uniform(dPD) notinbranch
inline sunrealtype s6b(sunrealtype const *udata, const int dPD) {
  return 1.0 / 30.0 * udata[-2 * dPD] - 2.0 / 5.0 * udata[-1 * dPD] -
         7.0 / 12.0 * udata[0] + 4.0 / 3.0 * udata[1 * dPD] -
         1.0 / 2.0 * udata[2 * dPD] + 2.0 / 15.0 * udata[3 * dPD] -
         1.0 / 60.0 * udata[4 * dPD];
}
#pragma omp declare simd uniform(dPD) notinbranch
inline sunrealtype s7f(sunrealtype const *udata, const int dPD) {
  return 1.0 / 140.0 * udata[-4 * dPD] - 1.0 / 15.0 * udata[-3 * dPD] +
         3.0 / 10.0 * udata[-2 * dPD] - 1.0 / 1.0 * udata[-1 * dPD] +
         1.0 / 4.0 * udata[0] + 3.0 / 5.0 * udata[1 * dPD] -
         1.0 / 10.0 * udata[2 * dPD] + 1.0 / 105.0 * udata[3 * dPD];
}
#pragma omp declare simd uniform(dPD) notinbranch
```

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

```
        1.0 / 2772.0 * udata[6 * dPD];
}
#pragma omp declare simd uniform(dPD) notinbranch
inline sunrealtype s12f(sunrealtype const *udata, const int dPD) {
  return -1.0 / 5544.0 * udata[-7 * dPD] + 1.0 / 396.0 * udata[-6 * dPD] -
         1.0 / 60.0 * udata[-5 * dPD] + 5.0 / 72.0 * udata[-4 * dPD] -
         5.0 / 24.0 * udata[-3 * dPD] + 1.0 / 2.0 * udata[-2 * dPD] -
         7.0 / 6.0 * udata[-1 * dPD] + 13.0 / 42.0 * udata[0] +
         5.0 / 8.0 * udata[1 * dPD] - 5.0 / 36.0 * udata[2 * dPD] +
         1.0 / 36.0 * udata[3 * dPD] - 1.0 / 264.0 * udata[4 * dPD] +
         1.0 / 3960.0 * udata[5 * dPD];
}
#pragma omp declare simd uniform(dPD) notinbranch
inline sunrealtype s12c(sunrealtype const *udata, const int dPD) {
  return 1.0 / 5544.0 * udata[-6 * dPD] - 1.0 / 385.0 * udata[-5 * dPD] +
         1.0 / 56.0 * udata[-4 * dPD] - 5.0 / 63.0 * udata[-3 * dPD] +
         15.0 / 56.0 * udata[-2 * dPD] - 6.0 / 7.0 * udata[-1 * dPD] + 0 +
         6.0 / 7.0 * udata[1 * dPD] - 15.0 / 56.0 * udata[2 * dPD] +
         5.0 / 63.0 * udata[3 * dPD] - 1.0 / 56.0 * udata[4 * dPD] +
         1.0 / 385.0 * udata[5 * dPD] - 1.0 / 5544.0 * udata[6 * dPD];
}
#pragma omp declare simd uniform(dPD) notinbranch
inline sunrealtype s12b(sunrealtype const *udata, const int dPD) {
  return -1.0 / 3960.0 * udata[-5 * dPD] + 1.0 / 264.0 * udata[-4 * dPD] -
         1.0 / 36.0 * udata[-3 * dPD] + 5.0 / 36.0 * udata[-2 * dPD] -
         5.0 / 8.0 * udata[-1 * dPD] - 13.0 / 42.0 * udata[0] +
         7.0 / 6.0 * udata[1 * dPD] - 1.0 / 2.0 * udata[2 * dPD] +
         5.0 / 24.0 * udata[3 * dPD] - 5.0 / 72.0 * udata[4 * dPD] +
         1.0 / 60.0 * udata[5 * dPD] - 1.0 / 396.0 * udata[6 * dPD] +
         1.0 / 5544.0 * udata[7 * dPD];
}
#pragma omp declare simd uniform(dPD) notinbranch
inline sunrealtype s13f(sunrealtype const *udata, const int dPD) {
  return -1.0 / 12012.0 * udata[-7 * dPD] + 1.0 / 792.0 * udata[-6 * dPD] -
         1.0 / 110.0 * udata[-5 * dPD] + 1.0 / 24.0 * udata[-4 * dPD] -
         5.0 / 36.0 * udata[-3 * dPD] + 3.0 / 8.0 * udata[-2 * dPD] -
         1.0 / 1.0 * udata[-1 * dPD] + 1.0 / 7.0 * udata[0] +
         3.0 / 4.0 * udata[1 * dPD] - 5.0 / 24.0 * udata[2 * dPD] +
         1.0 / 18.0 * udata[3 * dPD] - 1.0 / 88.0 * udata[4 * dPD] +
         1.0 / 660.0 * udata[5 * dPD] - 1.0 / 10296.0 * udata[6 * dPD];
}
#pragma omp declare simd uniform(dPD) notinbranch
inline sunrealtype s13b(sunrealtype const *udata, const int dPD) {
  return 1.0 / 10296.0 * udata[-6 * dPD] - 1.0 / 660.0 * udata[-5 * dPD] +
         1.0 / 88.0 * udata[-4 * dPD] - 1.0 / 18.0 * udata[-3 * dPD] +
         5.0 / 24.0 * udata[-2 * dPD] - 3.0 / 4.0 * udata[-1 * dPD] -
         1.0 / 7.0 * udata[0] + udata[1 * dPD] - 3.0 / 8.0 * udata[2 * dPD] +
         5.0 / 36.0 * udata[3 * dPD] - 1.0 / 24.0 * udata[4 * dPD] +
         1.0 / 110.0 * udata[5 * dPD] - 1.0 / 792.0 * udata[6 * dPD] +
         1.0 / 12012.0 * udata[7 * dPD];
}

//////////////////////////////////
// Stencils with dPD fixed to 6 //
//////////////////////////////////

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

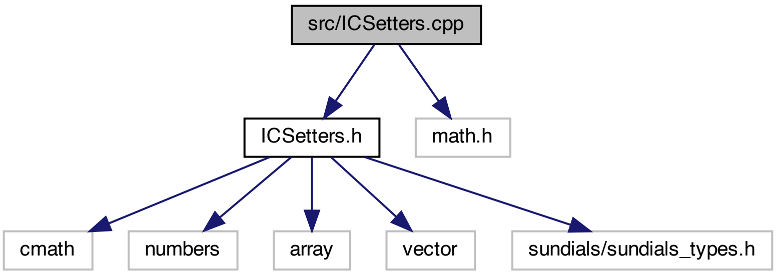
```
{ return s11b(udata, 6); }
inline sunrealtype s12f(sunrealtype const *udata)
{ return s12f(udata, 6); }
inline sunrealtype s12c(sunrealtype const *udata)
{ return s12c(udata, 6); }
inline sunrealtype s12b(sunrealtype const *udata)
{ return s12b(udata, 6); }
inline sunrealtype s13f(sunrealtype const *udata)
{ return s13f(udata, 6); }
inline sunrealtype s13b(sunrealtype const *udata)
{ return s13b(udata, 6); }

```

6.6 src/ICSetters.cpp File Reference

Implementation of the plane wave and Gaussian wave packets.

```
#include "ICSetters.h"
#include <math.h>
```

Include dependency graph for ICSetters.cpp:



6.6.1 Detailed Description

Implementation of the plane wave and Gaussian wave packets.

Definition in file ICSetters.cpp.

6.7 ICSetters.cpp

Go to the documentation of this file.

```
//////////////////////////////////////////////////////////////////////////
/// @file ICSetters.cpp
/// @brief Implementation of the plane wave and Gaussian wave packets
//////////////////////////////////////////////////////////////////////////

#include "ICSetters.h"

#include <math.h>

/** PlaneWave1D construction with */
PlaneWave1D::PlaneWave1D(std::array<sunrealtype, 3> k,
        std::array<sunrealtype, 3> p,
        std::array<sunrealtype, 3> 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 */
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 *
      std::numbers::pi; /* \f$ 2\pi \ \vec{k} \cdot \vec{x} \f$ */
  // Plane wave definition
  const std::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(std::array<sunrealtype, 3> k,
        std::array<sunrealtype, 3> p,
        std::array<sunrealtype, 3> 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 */
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 *
      std::numbers::pi; /* \f$ 2\pi \ \vec{k} \cdot \vec{x} \f$ */
  // Plane wave definition
  const std::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(std::array<sunrealtype, 3> k,
        std::array<sunrealtype, 3> p,
        std::array<sunrealtype, 3> 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 */
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 *
      std::numbers::pi; /* \f$ 2\pi \ \vec{k} \cdot \vec{x} \f$ */
  // Plane wave definition
  const std::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(std::array<sunrealtype, 3> k, std::array<sunrealtype, 3> p,
                 std::array<sunrealtype, 3> xo, sunrealtype phig_,
                 std::array<sunrealtype, 3> 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 */
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 *
      std::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 std::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(std::array<sunrealtype, 3> dis_,
        std::array<sunrealtype, 3> 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 = std::numbers::pi * w0 * w0 / zr; // formula for wavelength
}

```

```
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("0x12345");  // unused parameter
  //  \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$
  // waist at position z
  const sunrealtype wz = w0 * sqrt(1 + (zg * zg / zr / zr));
  // \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("0x12345"); // beam curvature
  if (abs(zg) > 1e-15)
    Rz = zg * (1 + (zr * zr / zg / zg));
  else
    Rz = 1e308;
  // wavenumber \f$ k = 2\pi/\lambda \f$
  const sunrealtype k = 2 * std::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
  // no z-component for 2D propagation
  const sunrealtype sa = sqrt(1 - ca * ca);
  // 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(std::array<sunrealtype, 3> dis_,
        std::array<sunrealtype, 3> axis_,
        sunrealtype Amp_,
        // std::array<sunrealtype, 3> 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 = std::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("0x12345");
  if (abs(zg) > 1e-15)
    Rz = zg * (1 + (zr * zr / zg / zg));
  else
    Rz = 1e308;
  const sunrealtype k = 2 * std::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 then add initial field values */
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(std::array<sunrealtype, 3> k,
        std::array<sunrealtype, 3> p,
        std::array<sunrealtype, 3> phi) {
  planeWaves1D.emplace_back(PlaneWave1D(k, p, phi));
}

/** Add plane waves in 2D to their container vector */
void ICSetter::addPlaneWave2D(std::array<sunrealtype, 3> k,
        std::array<sunrealtype, 3> p,
        std::array<sunrealtype, 3> phi) {
  planeWaves2D.emplace_back(PlaneWave2D(k, p, phi));
}

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

/** Add Gaussian waves in 1D to their container vector */
void ICSetter::addGauss1D(std::array<sunrealtype, 3> k,
        std::array<sunrealtype, 3> p,
        std::array<sunrealtype, 3> xo, sunrealtype phig_,
        std::array<sunrealtype, 3> phi) {
  gauss1Ds.emplace_back(Gauss1D(k, p, xo, phig_, phi));
}

/** Add Gaussian waves in 2D to their container vector */
void ICSetter::addGauss2D(std::array<sunrealtype, 3> dis_,
        std::array<sunrealtype, 3> 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_));
}
```

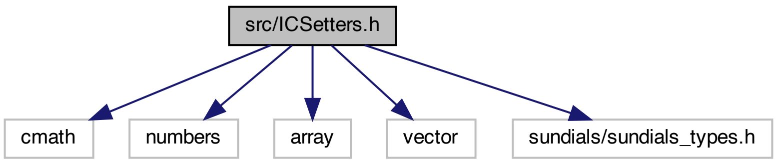
```
/** Add Gaussian waves in 3D to their container vector */
void ICSetter::addGauss3D(std::array<sunrealtype, 3> dis_,
        std::array<sunrealtype, 3> axis_,
        sunrealtype Amp_, sunrealtype phip_, sunrealtype w0_,
        sunrealtype zr_, sunrealtype Ph0_, sunrealtype PhA_)
{
  gauss3Ds.emplace_back(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

6.8 src/ICSetters.h File Reference

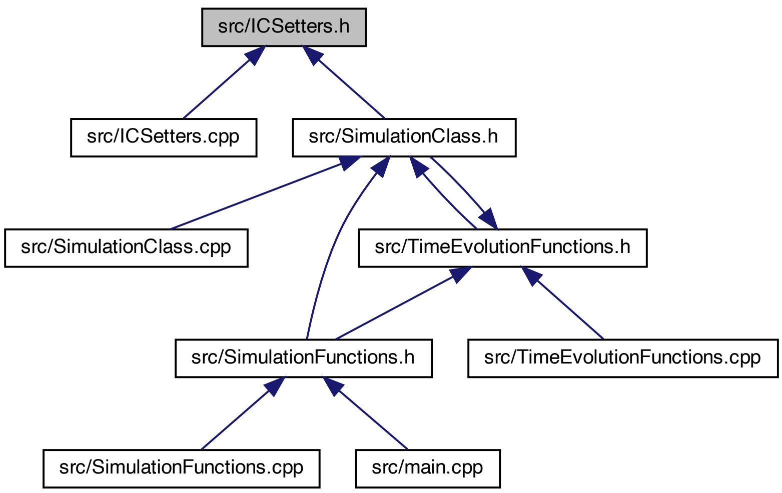
Declaration of the plane wave and Gaussian wave packets.

```
#include <cmath>
#include <numbers>
#include <array>
#include <vector>
#include <sundials/sundials_types.h>
```

Include dependency graph for ICSetters.h:



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



Data Structures

- class PlaneWave
  super-class for plane waves
- class PlaneWave1D
  class for plane waves in 1D
- class PlaneWave2D
  class for plane waves in 2D
- class PlaneWave3D
  class for plane waves in 3D
- class Gauss1D
  class for Gaussian pulses in 1D
- class Gauss2D
  class for Gaussian pulses in 2D
- class Gauss3D
  class for Gaussian pulses in 3D
- class ICSetter
  ICSetter class to initialize wave types with default parameters.

6.8.1 Detailed Description

Declaration of the plane wave and Gaussian wave packets.

Definition in file ICSetters.h.

6.9 ICSetters.h

Go to the documentation of this file.

```
//////////////////////////////////////////////////////////////////////
/// @file ICSetters.h
/// @brief Declaration of the plane wave and Gaussian wave packets
//////////////////////////////////////////////////////////////////////

#pragma once

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

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

/** @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(std::array<sunrealtype, 3> k = {1, 0, 0},
              std::array<sunrealtype, 3> p = {0, 0, 1},
              std::array<sunrealtype, 3> phi = {0, 0, 0});
  /// function 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(std::array<sunrealtype, 3> k = {1, 0, 0},
              std::array<sunrealtype, 3> p = {0, 0, 1},
              std::array<sunrealtype, 3> phi = {0, 0, 0});
  /// function 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(std::array<sunrealtype, 3> k = {1, 0, 0},
              std::array<sunrealtype, 3> p = {0, 0, 1},
              std::array<sunrealtype, 3> phi = {0, 0, 0});
  /// function for the actual implementation in space
  void addToSpace(sunrealtype x, sunrealtype y, sunrealtype z,
                  sunrealtype *pTo6Space) const;
};

/** @brief class for Gaussian pulses 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(std::array<sunrealtype, 3> k = {1, 0, 0},
          std::array<sunrealtype, 3> p = {0, 0, 1},
          std::array<sunrealtype, 3> xo = {0, 0, 0},
          sunrealtype phig_ = 1.0,
          std::array<sunrealtype, 3> phi = {0, 0, 0});
  /// function for the actual implementation in space
  void addToSpace(sunrealtype x, sunrealtype y, sunrealtype z,
                  sunrealtype *pTo6Space) const;

```

```
public:
};

/** @brief class for Gaussian pulses 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
  std::array<sunrealtype, 3> dis;
  /// normalized propagation axis
  std::array<sunrealtype, 3> 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(std::array<sunrealtype, 3> dis_ = {0, 0, 0},
          std::array<sunrealtype, 3> axis_ = {1, 0, 0},
          sunrealtype Amp_ = 1.0,
          sunrealtype phip_ = 0, sunrealtype w0_ = 1e-5,
          sunrealtype zr_ = 4e-5,
          sunrealtype Ph0_ = 2e-5, sunrealtype PhA_ = 0.45e-5);
  /// function for the actual implementation in space
  void addToSpace(sunrealtype x, sunrealtype y, sunrealtype z,
                  sunrealtype *pTo6Space) const;

public:
};

/** @brief class for Gaussian pulses 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
  std::array<sunrealtype, 3> dis;
  /// normalized propagation axis
  std::array<sunrealtype, 3> 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
  // std::array<sunrealtype, 3> pol;
  /// taille \f$ \omega_0 \f$
  sunrealtype w0;
  /// Rayleigh length \f$ z_R = \pi \omega_0^2 / \lambda \f$
  sunrealtype zr;
```

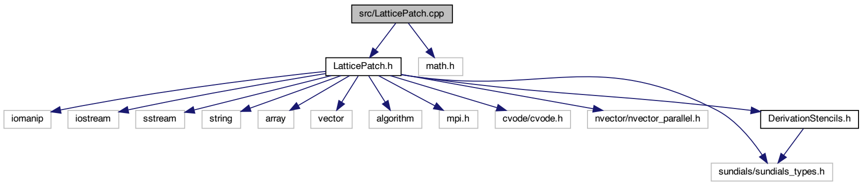
```
    /// center of beam \f$ \Phi_0 \f$
    sunrealtype Ph0;
    /// length of beam \f$ \Phi_A \f$
    sunrealtype PhA;
    /// amplitude projection on TE-mode (z-axis)
    sunrealtype A1;
    /// amplitude projection on xy-plane
    sunrealtype A2;
    /// wavelength \f$ \lambda \f$
    sunrealtype lambda;

public:
    /// construction with default parameters
    Gauss3D(std::array<sunrealtype, 3> dis_ = {0, 0, 0},
            std::array<sunrealtype, 3> axis_ = {1, 0, 0},
            sunrealtype Amp_ = 1.0,
            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
    std::vector<PlaneWave1D> planeWaves1D;
    /// container vector for plane waves in 2D
    std::vector<PlaneWave2D> planeWaves2D;
    /// container vector for plane waves in 3D
    std::vector<PlaneWave3D> planeWaves3D;
    /// container vector for Gaussian wave packets in 1D
    std::vector<Gauss1D> gauss1Ds;
    /// container vector for Gaussian wave packets in 2D
    std::vector<Gauss2D> gauss2Ds;
    /// container vector for Gaussian wave packets in 3D
    std::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(std::array<sunrealtype, 3> k = {1, 0, 0},
                        std::array<sunrealtype, 3> p = {0, 0, 1},
                        std::array<sunrealtype, 3> phi = {0, 0, 0});
    /// function to add plane waves in 2D to their container vector
    void addPlaneWave2D(std::array<sunrealtype, 3> k = {1, 0, 0},
                        std::array<sunrealtype, 3> p = {0, 0, 1},
                        std::array<sunrealtype, 3> phi = {0, 0, 0});
    /// function to add plane waves in 3D to their container vector
    void addPlaneWave3D(std::array<sunrealtype, 3> k = {1, 0, 0},
                        std::array<sunrealtype, 3> p = {0, 0, 1},
                        std::array<sunrealtype, 3> phi = {0, 0, 0});
    /// function to add Gaussian wave packets in 1D to their container vector
    void addGauss1D(std::array<sunrealtype, 3> k = {1, 0, 0},
                    std::array<sunrealtype, 3> p = {0, 0, 1},
                    std::array<sunrealtype, 3> xo = {0, 0, 0},
                    sunrealtype phig_ = 1.0,
                    std::array<sunrealtype, 3> phi = {0, 0, 0});
    /// function to add Gaussian wave packets in 2D to their container vector
    void addGauss2D(std::array<sunrealtype, 3> dis_ = {0, 0, 0},
                    std::array<sunrealtype, 3> axis_ = {1, 0, 0},
                    sunrealtype Amp_ = 1.0, sunrealtype phip_ = 0,
                    sunrealtype w0_ = 1e-5, sunrealtype zr_ = 4e-5,
                    sunrealtype Ph0_ = 2e-5, sunrealtype PhA_ = 0.45e-5);
    /// function to add Gaussian wave packets in 3D to their container vector
    void addGauss3D(std::array<sunrealtype, 3> dis_ = {0, 0, 0},
                    std::array<sunrealtype, 3> axis_ = {1, 0, 0},
                    sunrealtype Amp_ = 1.0, sunrealtype phip_ = 0,
                    sunrealtype w0_ = 1e-5, sunrealtype zr_ = 4e-5,
                    sunrealtype Ph0_ = 2e-5, sunrealtype PhA_ = 0.45e-5);
};
```

6.10 src/LatticePatch.cpp File Reference

Costruction of the overall envelope lattice and the lattice patches.

```
#include "LatticePatch.h"
#include <math.h>
```

Include dependency graph for LatticePatch.cpp:



Functions

- int generatePatchwork (const Lattice &envelopeLattice, LatticePatch &patchToMold, const int DLx, const int DLy, const int DLz)

  Set up the patchwork.
- void errorKill (const std::string &errorMessage)

  Print a specific error message to stderr.
- int check_error (int error, const char ∗funcname, int id)

  helper function to check MPI errors
- int check_retval (void ∗returnvalue, const char ∗funcname, int opt, int id)

  helper function to check CVode errors

6.10.1 Detailed Description

Costruction of the overall envelope lattice and the lattice patches.

Definition in file LatticePatch.cpp.

6.10.2 Function Documentation

6.10.2.1 check_error()

```
int check_error (
            int error,
            const char * funcname,
            int id )
```

helper function to check MPI errors

Check MPI errors. Error handler must be set.

Definition at line 912 of file LatticePatch.cpp.

```
00912                                                                  {
00913     int eclass, len;
00914     char errorstring[MPI_MAX_ERROR_STRING];
00915     if( error != MPI_SUCCESS ) {
00916         MPI_Error_class(error,&eclass);
00917         MPI_Error_string(error,errorstring,&len);
00918         std::cerr « "MPI Error(process " « id « ") in " « funcname « " : "
00919             « errorstring « ", from class " « eclass « std::endl;
00920         return 1;
00921     }
00922     return 0;
00923 }
```

6.10.2.2 check_retval()

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

helper function to check CVode errors

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

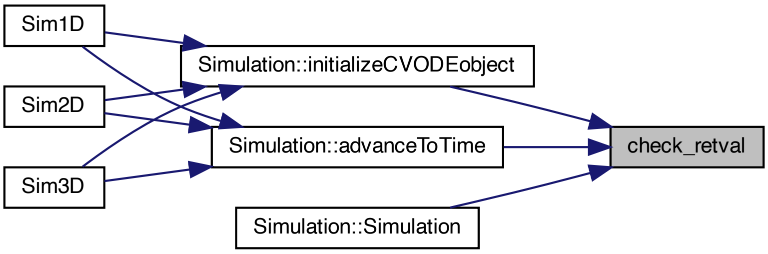
Definition at line 932 of file LatticePatch.cpp.

```
00932                                                                                  {
00933   int *retval = nullptr;
00934 
00935   /* Check if SUNDIALS function returned NULL pointer - no memory allocated */
00936   if (opt == 0 && returnvalue == nullptr) {
00937     fprintf(stderr,
00938             "\nSUNDIALS_ERROR(%d): %s() failed - returned NULL pointer\n\n", id,
00939             funcname);
00940     return (1);
00941   }
00942 
00943   /* Check if retval < 0 */
00944   else if (opt == 1) {
00945     retval = (int *)returnvalue;
00946     char *flagname = CVodeGetReturnFlagName(*retval);
00947     if (*retval < 0) {
00948       fprintf(stderr, "\nSUNDIALS_ERROR(%d): %s() failed with retval = %d: "
00949               "%s\n\n",
00950               id, funcname, *retval, flagname);
00951       return (1);
00952     }
00953   }
00954 
00955   /* Check if function returned NULL pointer - no memory allocated */
00956   else if (opt == 2 && returnvalue == nullptr) {
00957     fprintf(stderr,
00958             "\nMEMORY_ERROR(%d): %s() failed - returned NULL pointer\n\n", id,
00959             funcname);
```

```
00960     return (1);
00961   }
00962
00963   return (0);
00964 }
```

Referenced by Simulation::advanceToTime(), Simulation::initializeCVODEobject(), and Simulation::Simulation().

Here is the caller graph for this function:



6.10.2.3 errorKill()

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

Print a specific error message to stderr.

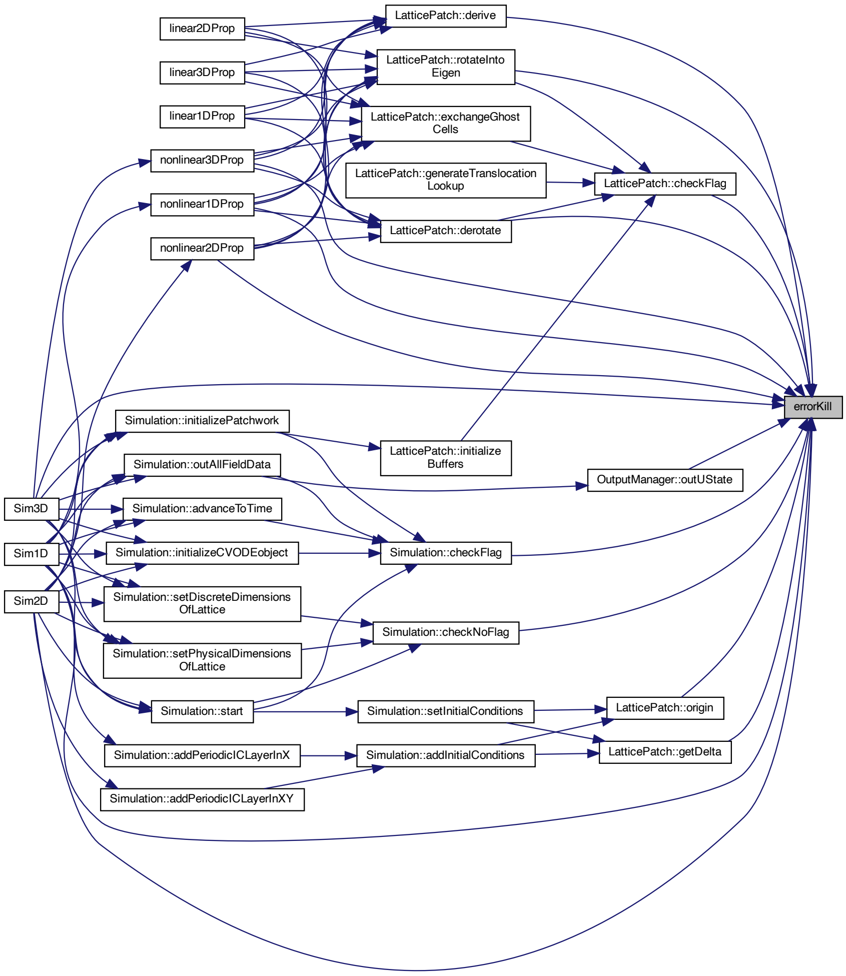
helper function for error messages

Definition at line 900 of file LatticePatch.cpp.

```
00900                                                 {
00901   int my_prc=0;
00902   MPI_Comm_rank(MPI_COMM_WORLD,&my_prc);
00903   if (my_prc==0) {
00904     std::cerr « std::endl « "Error: " « errorMessage
00905         « "\nAborting..." « std::endl;
00906     MPI_Abort(MPI_COMM_WORLD, 1);
00907     return;
00908   }
00909 }
```

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

Here is the caller graph for this function:



6.10.2.4 generatePatchwork()

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

Set up the patchwork.

friend function for creating the patchwork slicing of the overall lattice

Definition at line 109 of file LatticePatch.cpp.

```
00111                                                             {
00112   // Retrieve the ghost layer depth
00113   const int gLW = envelopeLattice.get_ghostLayerWidth();
00114   // Retrieve the data point dimension
```

```
    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();
    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
    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;
    /* 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 sunindextype s1 = patchToMold.nx, s2 = patchToMold.ny,
          s3 = patchToMold.nz;
    const sunindextype s_min = std::min(s1, std::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«std::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();
  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
  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;
  /* 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 sunindextype s1 = patchToMold.nx, s2 = patchToMold.ny,
        s3 = patchToMold.nz;
  const sunindextype s_min = std::min(s1, std::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
sunindextype 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;
  }
}
```

```
/** In order to avoid cache misses:
 * create vectors to translate u vector into space coordinates and vice versa
 * and same for left and right ghost layers to space */
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 sunindextype mx = nx + 2 * gLW;
  const sunindextype my = ny + 2 * gLW;
  const sunindextype mz = nz + 2 * gLW;
  // sizes for lookup vectors
  const sunindextype totalNP = nx * ny * nz;
  const sunindextype haloXSize = mx * ny * nz;
  const sunindextype haloYSize = nx * my * nz;
  const sunindextype haloZSize = nx * ny * mz;
  // generate u->uAux
  uTox.resize(totalNP);
  uToy.resize(totalNP);
  uToz.resize(totalNP);
  // generate uAux->u with length including halo
  xTou.resize(haloXSize);
  yTou.resize(haloYSize);
  zTou.resize(haloZSize);
  // same for ghost layer lookup tables
  const sunindextype ghostXSize = gLW * ny * nz;
  const sunindextype ghostYSize = gLW * nx * nz;
  const sunindextype ghostZSize = gLW * nx * ny;
  lgcTox.resize(ghostXSize);
  rgcTox.resize(ghostXSize);
  lgcToy.resize(ghostYSize);
  rgcToy.resize(ghostYSize);
  lgcToz.resize(ghostZSize);
  rgcToz.resize(ghostZSize);
  // variables for cartesian position in the 3D discrete lattice
  sunindextype px = 0, py = 0, pz = 0;
  // Fill the lookup tables
  #pragma omp parallel default(none) \
  private(px, py, pz) \
  shared(uTox, uToy, uToz, xTou, yTou, zTou, \
          nx, ny, mx, my, mz, gLW, totalNP, \
          lgcTox, rgcTox, lgcToy, rgcToy, lgcToz, rgcToz, \
          ghostXSize, ghostYSize, ghostZSize)
  {
  #pragma omp for simd schedule(static)
  for (sunindextype i = 0; i < totalNP; i++) { // loop over 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;
  }
  #pragma omp for simd schedule(static)
  for (sunindextype i = 0; i < ghostXSize; 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;
  }
  #pragma omp for simd schedule(static)
  for (sunindextype i = 0; i < ghostYSize; 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;
  }
  #pragma omp for simd schedule(static)
  for (sunindextype i = 0; i < ghostZSize; 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 the rotation
 * methods;
 * 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;
  }
}

/// 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 std::vector<sunindextype> &lookup) {
  sunindextype ii = 0, target = 0;
  const sunindextype size = lookup.size();
  const int dPD = envelopeLattice->get_dataPointDimension();
  #pragma omp parallel for simd \
  private(target, ii) \
  shared(lookup, outArray, inArray, size, dPD) \
  schedule(static)
  for (sunindextype i = 0; i < size; i++) {
    // get correct u-vector and spatial indices along previously defined lookup
    // tables
    target = dPD * lookup[i];
    ii = dPD * 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 std::vector<sunindextype> &lookup) {
  sunindextype ii = 0, target = 0;
  const int dPD = envelopeLattice->get_dataPointDimension();
  const sunindextype size = lookup.size();
  #pragma omp parallel for simd \
  private(target, ii) \
  shared(lookup, outArray, inArray, size, dPD) \
  schedule(static)
  for (sunindextype i = 0; i < size; i++) {
    target = dPD * lookup[i];
    ii = dPD * 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 std::vector<sunindextype> &lookup) {
  sunindextype ii = 0, target = 0;
  const sunindextype size = lookup.size();
  const int dPD = envelopeLattice->get_dataPointDimension();
  #pragma omp parallel for simd \
  private(target, ii) \
  shared(lookup, outArray, inArray, size, dPD) \
  schedule(static)
  for (sunindextype i = 0; i < size; i++) {
    target = dPD * lookup[i];
    ii = dPD * 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 sunindextype totalNP = discreteSize();
  sunindextype ii = 0, target = 0;
  switch (dir) {
  case 1:
    #pragma omp parallel for simd \
    private(ii, target) \
    shared(dPD, gLW, totalNP, uTox, uAux, buffOut) \
    schedule(static)
    for (sunindextype i = 0; i < totalNP; 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 parallel for simd \
    private(ii, target) \
    shared(dPD, gLW, totalNP, uTox, uAux, buffOut) \
    schedule(static)
    for (sunindextype i = 0; i < totalNP; 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 parallel for simd \
    private(ii, target) \
    shared(dPD, gLW, totalNP, uTox, uAux, buffOut) \
    schedule(static)
    for (sunindextype i = 0; i < totalNP; 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 sunindextype 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
  sunindextype li = 0, ui = 0;
  // Fill the halo buffers
  #pragma omp parallel for default(none) \
  private(ui) firstprivate(li) \
  shared(nx, ny, mx, my, mz, dPD, distToRight, uData, \
          ghostCellLeftToSend, ghostCellRightToSend)
  for (sunindextype iz = 0; iz < mz; iz++) {
    for (sunindextype 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)
      std::copy(&uData[ui], &uData[ui + mx * dPD], &ghostCellLeftToSend[li]);
      ui += distToRight * dPD;
      std::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 Irecv/Isend

  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);
  */
}

/// Check if all flags are set
void LatticePatch::checkFlag(unsigned int flag) const {
  if (!(statusFlags & flag)) {
    std::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 that the ghost cells are initialized ";
      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 adjusted to the chosen stencil order
  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 sunindextype dirWidth = discreteSize(dir) + 2 * gLW;
  // width of patch only in given direction
  const sunindextype dirWidthO = discreteSize(dir);
  // size of plane perpendicular to given dimension
  const sunindextype perpPlainSize = discreteSize() / discreteSize(dir);
  // physical distance between points in that direction
  sunrealtype dxi = nan("0x12345");
  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
const int order = envelopeLattice->get_stencilOrder();
switch (order) {
case 1:  // gLW=1
  #pragma omp parallel for default(none) \
  shared(perpPlainSize, dxi, dirWidth, dirWidthO, gLW, dPD, uAux)
  for (sunindextype i = 0; i < perpPlainSize; i++) {
    #pragma omp simd
    for (sunindextype 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:  // gLW=2
  #pragma omp parallel for default(none) \
  shared(perpPlainSize, dxi, dirWidth, dirWidthO, gLW, dPD, uAux)
  for (sunindextype i = 0; i < perpPlainSize; i++) {
    #pragma omp simd
    for (sunindextype 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:  // gLW=2
  #pragma omp parallel for default(none) \
  shared(perpPlainSize, dxi, dirWidth, dirWidthO, gLW, dPD, uAux)
  for (sunindextype i = 0; i < perpPlainSize; i++) {
    #pragma omp simd
    for (sunindextype 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:  // gLW=3
  #pragma omp parallel for default(none) \
  shared(perpPlainSize, dxi, dirWidth, dirWidthO, gLW, dPD, uAux)
  for (sunindextype i = 0; i < perpPlainSize; i++) {
    #pragma omp simd
    for (sunindextype 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:  // gLW=3
  #pragma omp parallel for default(none) \
  shared(perpPlainSize, dxi, dirWidth, dirWidthO, gLW, dPD, uAux)
  for (sunindextype i = 0; i < perpPlainSize; i++) {
    #pragma omp simd
    for (sunindextype 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:  // gLW=4
        #pragma omp parallel for default(none) \
        shared(perpPlainSize, dxi, dirWidth, dirWidthO, gLW, dPD, uAux)
        for (sunindextype i = 0; i < perpPlainSize; i++) {
          #pragma omp simd
          for (sunindextype 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:  // gLW=4
        #pragma omp parallel for default(none) \
        shared(perpPlainSize, dxi, dirWidth, dirWidthO, gLW, dPD, uAux)
        for (sunindextype i = 0; i < perpPlainSize; i++) {
          #pragma omp simd
          for (sunindextype 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:  // gLW=5
        #pragma omp parallel for default(none) \
        shared(perpPlainSize, dxi, dirWidth, dirWidthO, gLW, dPD, uAux)
        for (sunindextype i = 0; i < perpPlainSize; i++) {
          #pragma omp simd
          for (sunindextype 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:  // gLW=5
        #pragma omp parallel for default(none) \
        shared(perpPlainSize, dxi, dirWidth, dirWidthO, gLW, dPD, uAux)
        for (sunindextype i = 0; i < perpPlainSize; i++) {
          #pragma omp simd
          for (sunindextype 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:  // gLW=6
        #pragma omp parallel for default(none) \
        shared(perpPlainSize, dxi, dirWidth, dirWidthO, gLW, dPD, uAux)
        for (sunindextype i = 0; i < perpPlainSize; i++) {
          #pragma omp simd
          for (sunindextype 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:  // gLW=6
        #pragma omp parallel for default(none) \
        shared(perpPlainSize, dxi, dirWidth, dirWidthO, gLW, dPD, uAux)
```

```
    for (sunindextype i = 0; i < perpPlainSize; i++) {
      #pragma omp simd
      for (sunindextype 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:  // gLW=7
    #pragma omp parallel for default(none) \
    shared(perpPlainSize, dxi, dirWidth, dirWidthO, gLW, dPD, uAux)
    for (sunindextype i = 0; i < perpPlainSize; i++) {
      #pragma omp simd
      for (sunindextype 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:  // gLW=7
    // For all points in the plane perpendicular to the given direction
    #pragma omp parallel for default(none) \
    shared(perpPlainSize, dxi, dirWidth, dirWidthO, gLW, dPD, uAux)
    for (sunindextype i = 0; i < perpPlainSize; i++) {
      // iterate through the derivation direction
      #pragma omp simd
      for (sunindextype 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
        // and update position by ghost width shift
        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 stderr
void errorKill(const std::string & errorMessage) {
  int my_prc=0;
  MPI_Comm_rank(MPI_COMM_WORLD,&my_prc);
  if (my_prc==0) {
    std::cerr « std::endl « "Error: " « errorMessage
      « "\nAborting..." « std::endl;
    MPI_Abort(MPI_COMM_WORLD, 1);
    return;
  }
}

/** Check MPI errors. Error handler must be set. */
int check_error(int error, const char *funcname, int id) {
    int eclass, len;
    char errorstring[MPI_MAX_ERROR_STRING];
    if( error != MPI_SUCCESS ) {
        MPI_Error_class(error,&eclass);
        MPI_Error_string(error,errorstring,&len);
        std::cerr « "MPI Error(process " « id « ") in " « funcname « " : "
            « errorstring « ", from class " « eclass « std::endl;
        return 1;
    }
    return 0;
}

/** Check function return value. Adapted from CVode examples.
     opt == 0 means SUNDIALS function allocates memory so check if
```

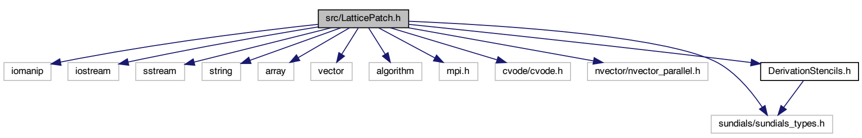
```
               returned NULL pointer
     opt == 1 means SUNDIALS function returns an integer value so check if
               retval < 0
     opt == 2 means function allocates memory so check if returned
               NULL pointer */
int check_retval(void *returnvalue, const char *funcname, int 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;
    char *flagname = CVodeGetReturnFlagName(*retval);
    if (*retval < 0) {
      fprintf(stderr, "\nSUNDIALS_ERROR(%d): %s() failed with retval = %d: "
              "%s\n\n",
              id, funcname, *retval, flagname);
      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

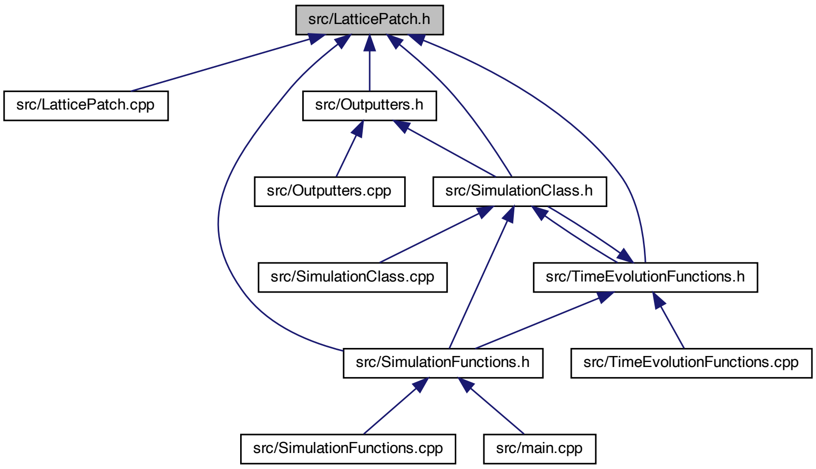
Declaration of the lattice and lattice patches.

```
#include <iomanip>
#include <iostream>
#include <sstream>
#include <string>
#include <array>
#include <vector>
#include <algorithm>
#include <mpi.h>
#include <cvode/cvode.h>
#include <nvector/nvector_parallel.h>
#include <sundials/sundials_types.h>
#include "DerivationStencils.h"
```

Include dependency graph for LatticePatch.h:



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



Data Structures

- class Lattice

  Lattice class for the construction of the enveloping discrete simulation space.
- class LatticePatch

  LatticePatch class for the construction of the patches in the enveloping lattice.

Functions

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

  helper function for error messages
- int check_error (int error, const char ∗funcname, int id)

  helper function to check MPI errors
- int check_retval (void ∗returnvalue, const char ∗funcname, int opt, int id)

  helper function to check CVode errors

Variables

- constexpr unsigned int FLatticeDimensionSet = 0x01

  lattice construction checking flag

- constexpr unsigned int FLatticePatchSetUp = 0x01
- constexpr unsigned int TranslocationLookupSetUp = 0x02
- constexpr unsigned int GhostLayersInitialized = 0x04
- constexpr unsigned int BuffersInitialized = 0x08

6.12.1 Detailed Description

Declaration of the lattice and lattice patches.

Definition in file LatticePatch.h.

6.12.2 Function Documentation

6.12.2.1 check_error()

```
int check_error (
        int error,
        const char * funcname,
        int id )
```

helper function to check MPI errors

Check MPI errors. Error handler must be set.

Definition at line 912 of file LatticePatch.cpp.

```
00912                                                         {
00913     int eclass, len;
00914     char errorstring[MPI_MAX_ERROR_STRING];
00915     if( error != MPI_SUCCESS ) {
00916         MPI_Error_class(error,&eclass);
00917         MPI_Error_string(error,errorstring,&len);
00918         std::cerr « "MPI Error(process " « id « ") in " « funcname « " : "
00919             « errorstring « ", from class " « eclass « std::endl;
00920         return 1;
00921     }
00922     return 0;
00923 }
```

6.12.2.2 check_retval()

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

helper function to check CVode errors

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

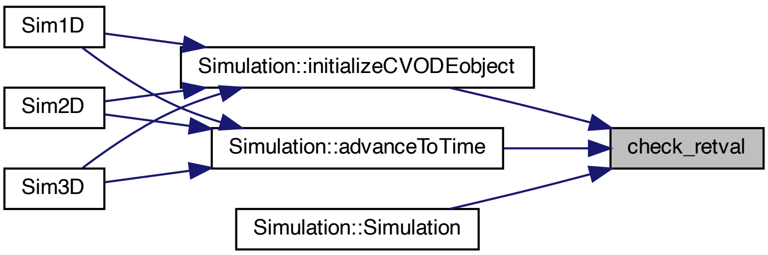
Definition at line 932 of file LatticePatch.cpp.

```
00932                                                                          {
00933   int *retval = nullptr;
00934 
00935   /* Check if SUNDIALS function returned NULL pointer - no memory allocated */
00936   if (opt == 0 && returnvalue == nullptr) {
00937     fprintf(stderr,
00938             "\nSUNDIALS_ERROR(%d): %s() failed - returned NULL pointer\n\n", id,
```

```
                 funcname);
      return (1);
    }

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

Referenced by Simulation::advanceToTime(), Simulation::initializeCVODEobject(), and Simulation::Simulation().

Here is the caller graph for this function:



6.12.2.3 errorKill()

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

helper function for error messages

helper function for error messages

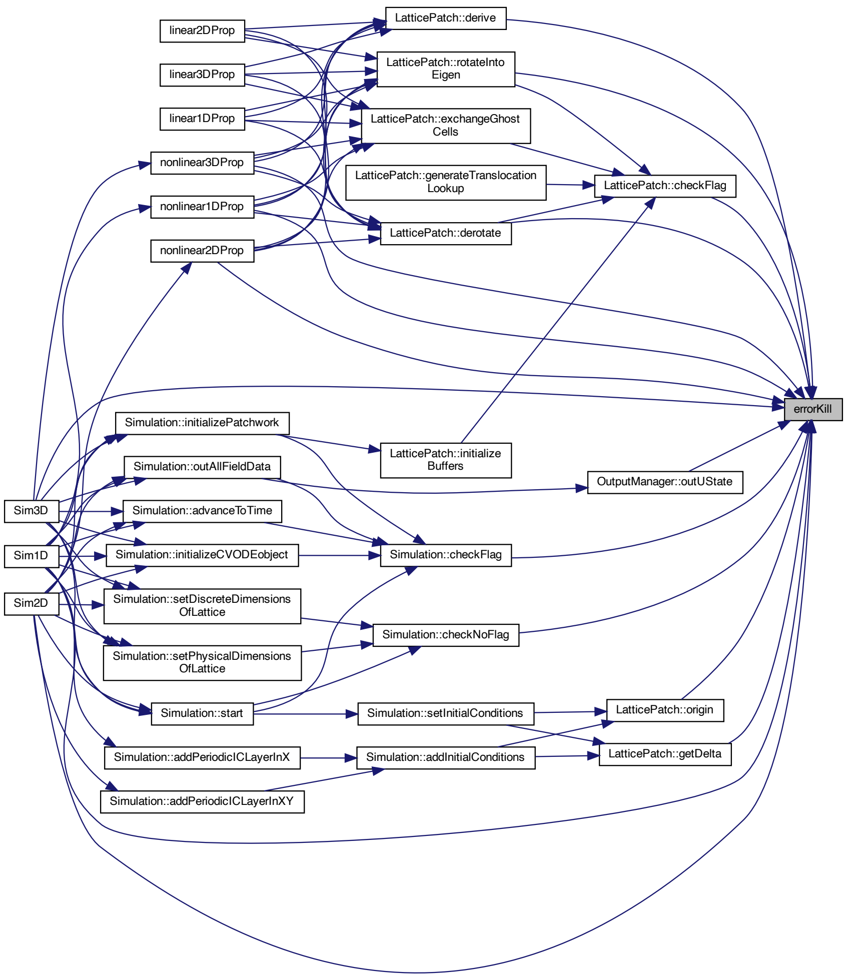
Definition at line 900 of file LatticePatch.cpp.

```
00900                                                    {
00901     int my_prc=0;
00902     MPI_Comm_rank(MPI_COMM_WORLD,&my_prc);
00903     if (my_prc==0) {
00904       std::cerr « std::endl « "Error: " « errorMessage
00905           « "\nAborting..." « std::endl;
00906       MPI_Abort(MPI_COMM_WORLD, 1);
00907       return;
00908     }
```

```
00909 }
```

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

Here is the caller graph for this function:



6.12.3 Variable Documentation

6.12.3.1 BuffersInitialized

```
constexpr unsigned int BuffersInitialized = 0x08  [constexpr]
```

lattice patch construction checking flag

Definition at line 42 of file LatticePatch.h.

Referenced by LatticePatch::checkFlag(), and LatticePatch::initializeBuffers().

6.12.3.2 FLatticeDimensionSet

```
constexpr unsigned int FLatticeDimensionSet = 0x01  [constexpr]
```

lattice construction checking flag

Definition at line 35 of file LatticePatch.h.

Referenced by LatticePatch::exchangeGhostCells(), LatticePatch::generateTranslocationLookup(), LatticePatch::initializeBuffers(), and Lattice::setPhysicalDimensions().

6.12.3.3 FLatticePatchSetUp

```
constexpr unsigned int FLatticePatchSetUp = 0x01  [constexpr]
```

lattice patch construction checking flag

Definition at line 39 of file LatticePatch.h.

Referenced by LatticePatch::checkFlag(), LatticePatch::derotate(), LatticePatch::exchangeGhostCells(), LatticePatch::rotateIntoEigen and LatticePatch::∼LatticePatch().

6.12.3.4 GhostLayersInitialized

```
constexpr unsigned int GhostLayersInitialized = 0x04  [constexpr]
```

lattice patch construction checking flag

Definition at line 41 of file LatticePatch.h.

Referenced by LatticePatch::checkFlag(), LatticePatch::exchangeGhostCells(), and LatticePatch::rotateIntoEigen().

6.12.3.5 TranslocationLookupSetUp

```
constexpr unsigned int TranslocationLookupSetUp = 0x02  [constexpr]
```

lattice patch construction checking flag

Definition at line 40 of file LatticePatch.h.

Referenced by LatticePatch::checkFlag(), LatticePatch::derotate(), LatticePatch::generateTranslocationLookup(), and LatticePatch::rotateIntoEigen().

6.13 LatticePatch.h

Go to the documentation of this file.

```
////////////////////////////////////////////////////////////
/// @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>
#if defined(_OPENMP)
#include <omp.h>
#endif

// 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"

/// lattice construction checking flag
constexpr unsigned int FLatticeDimensionSet = 0x01;

///@{
/** lattice patch construction checking flag */
constexpr unsigned int FLatticePatchSetUp = 0x01;
constexpr unsigned int TranslocationLookupSetUp = 0x02;
constexpr unsigned int GhostLayersInitialized = 0x04;
constexpr unsigned int BuffersInitialized = 0x08;
///@}

/** @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;
  /// lattice status flags
  unsigned int 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;
  /// 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;
  }
  ///@}
};

/** @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;
  /// lattice patch status flags
  unsigned int statusFlags;
  /// pointer to the enveloping lattice
  const Lattice *envelopeLattice;
  /// aid (auxilliarly) vector including ghost cells to compute the derivatives
  std::vector<sunrealtype> uAux;
  ///@{
  /** translocation lookup table */
  std::vector<sunindextype> uTox, uToy, uToz, xTou, yTou, zTou;
  ///@}
  ///@{
  /** buffer to save spatial derivative values */
  std::vector<sunrealtype> buffX, buffY, buffZ;
```

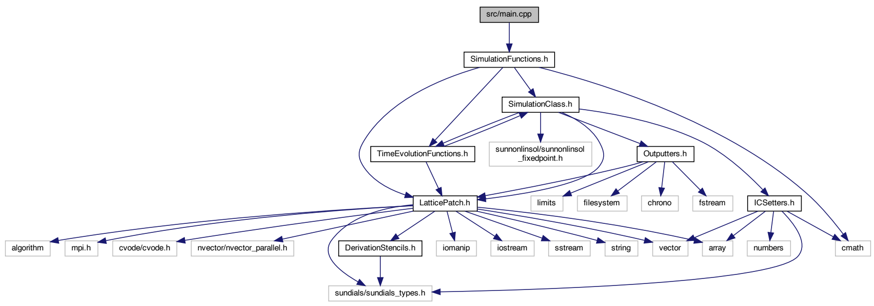
```
  ///@}
  ///@{
  /** buffer for passing ghost cell data */
  std::vector<sunrealtype> ghostCellLeft, ghostCellRight, ghostCellLeftToSend,
      ghostCellRightToSend, ghostCellsToSend, ghostCells;
  ///@}
  ///@{
  /** ghost cell translocation lookup table */
  std::vector<sunindextype> lgcTox, rgcTox, lgcToy, rgcToy, lgcToz, rgcToz;
  ///@}
  ///@{
  /** Rotate and translocate an input array according to a lookup into an output
   * array */
  inline void rotateToX(sunrealtype *outArray, const sunrealtype *inArray,
                        const std::vector<sunindextype> &lookup);
  inline void rotateToY(sunrealtype *outArray, const sunrealtype *inArray,
                        const std::vector<sunindextype> &lookup);
  inline void rotateToZ(sunrealtype *outArray, const sunrealtype *inArray,
                        const std::vector<sunindextype> &lookup);
  ///@}
public:
  /// ID of the LatticePatch, corresponds to process number (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;
  ///@}
  /// pointer to spatial derivative data buffers
  std::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
  sunindextype 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 cache efficiency
  void generateTranslocationLookup();
  /// function to rotate u into Z-matrix eigenraum
  void rotateIntoEigen(const int dir);
  /// function to derotate uAux into dudata lattice direction of x
  void derotate(int dir, sunrealtype *buffOut);
  /// initialize buffers to save derivatives
  void initializeBuffers();
  /// function to exchange ghost cells
  void exchangeGhostCells(const int dir);
  /// 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 std::string & errorMessage);

/// helper function to check MPI errors
int check_error(int error, const char *funcname, int id);

/// helper function to check CVode errors
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

A 2D simulation with specified

A 3D simulation with specified

Definition at line 9 of file main.cpp.

```
00010 {
00011     /** Determine the output directory.
00012      * A "SimResults" folder will be created if non-existent
00013      * with a subdirectory named in the identifier format
00014      * "yy-mm-dd_hh-MM-ss" that contains the csv files     */
```

```
    constexpr auto outputDirectory = "/path/to/directory/";

    if(!filesystem::exists(outputDirectory)) {
        cerr«"\nOutput directory nonexistent.\n";
        exit(1);
    }

    // Initialize MPI environment
    int provided;
    MPI_Init_thread(&argc, &argv, MPI_THREAD_SINGLE, &provided);


    //------------ 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=100;                                      /// -  output step multiples
    constexpr char outputStyle='c';                                    /// - output in csv (c) or binary (b) format

    /// 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=40e-6l;                       /// - physical total simulation time
    constexpr int numberOfSteps=100;                                   /// - discrete time steps
    constexpr int outputStep=100;                                      /// - output step multiples
    constexpr char outputStyle='c';                                    /// - output in csv (c) or binary (b) format

    /// 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=13;                                           /// - 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={800,800,200};     /// - number of lattice points in any dimension
    constexpr array<int,3> patches_per_dim= {8,8,2};                         /// - slicing of discrete dimensions into patches
```

```
    constexpr bool periodic=true;                                          /// - perodic or non-periodic boundaries
    int processOrder=3;                                                    /// - processes of the weak-field expansion, see README.md
    constexpr sunrealtype simulationTime=40e-6;                            /// - physical total simulation time
    constexpr int numberOfSteps=40;                                        /// - discrete time steps
    constexpr int outputStep=20;                                           /// - output step multiples
    constexpr char outputStyle='b';                                        /// - output in csv (c) or binary (b) format

    /// 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 = 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
    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 = 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<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.");
    static_assert(latticepoints_per_dim[0]/patches_per_dim[0] ==
            latticepoints_per_dim[1]/patches_per_dim[1] &&
            latticepoints_per_dim[0]/patches_per_dim[0] ==
            latticepoints_per_dim[2]/patches_per_dim[2],
            "At 3D simulations you are forced to make patches cubic in terms of "
            "lattice points as this is decisive for computational efficiency.");
    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 -------------//

    // Finalize MPI environment
    MPI_Finalize();

    return 0;
}
```

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 */

using namespace std;

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     */
    constexpr auto outputDirectory = "/path/to/directory/";

    if(!filesystem::exists(outputDirectory)) {
        cerr«"\nOutput directory nonexistent.\n";
        exit(1);
    }

    // Initialize MPI environment
    int provided;
    MPI_Init_thread(&argc, &argv, MPI_THREAD_SINGLE, &provided);


    //------------ 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=100;                                       /// -  output step multiples
    constexpr char outputStyle=’c’;                                     /// - output in csv (c) or binary (b) format

    /// 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=40e-6l;                       /// - physical total simulation time
    constexpr int numberOfSteps=100;                                   /// - discrete time steps
    constexpr int outputStep=100;                                      /// - output step multiples
    constexpr char outputStyle='c';                                    /// - output in csv (c) or binary (b) format

    /// 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,
```

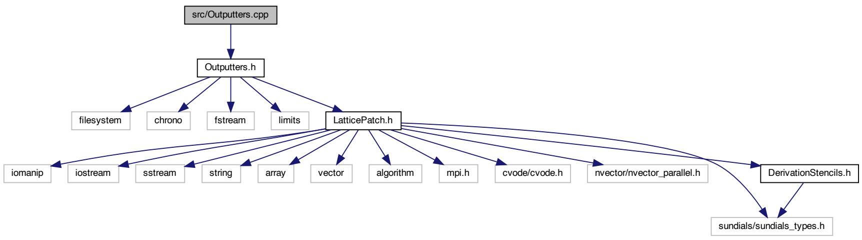
```
            outputStyle,planewaves,Gaussians2D);
    */
    ////////////////////////////////////////////////////


    //////////////////// -- 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=13;                                           /// - 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={800,800,200};     /// - number of lattice points in any dimension
    constexpr array<int,3> patches_per_dim= {8,8,2};                         /// - slicing of discrete dimensions into patches
    constexpr bool periodic=true;                                            /// - perodic or non-periodic boundaries
    int processOrder=3;                                                      /// - processes of the weak-field expansion, see README.md
    constexpr sunrealtype simulationTime=40e-6;                              /// - physical total simulation time
    constexpr int numberOfSteps=40;                                          /// - discrete time steps
    constexpr int outputStep=20;                                             /// - output step multiples
    constexpr char outputStyle='b';                                          /// - output in csv (c) or binary (b) format

    /// 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 = 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
    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 = 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<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.");
    static_assert(latticepoints_per_dim[0]/patches_per_dim[0] ==
            latticepoints_per_dim[1]/patches_per_dim[1] &&
            latticepoints_per_dim[0]/patches_per_dim[0] ==
            latticepoints_per_dim[2]/patches_per_dim[2],
            "At 3D simulations you are forced to make patches cubic in terms of "
            "lattice points as this is decisive for computational efficiency.");
    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 -------------//

    // 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"

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

/// 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
std::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};

  std::stringstream temp;
  temp « std::setfill('0') « std::setw(2)
       « static_cast<int>(ymd.year() - chrono::years(2000)) « "-"
       « std::setfill('0') « std::setw(2)
       « static_cast<unsigned>(ymd.month()) « "-"
       « std::setfill('0') « std::setw(2)
       « static_cast<unsigned>(ymd.day()) « "_"
       « std::setfill('0') « std::setw(2) « hms.hours().count()
       « "-" « std::setfill('0')
       « std::setw(2) « hms.minutes().count() « "-"
       « std::setfill('0') « std::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 std::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
  std::ifstream fin("main.cpp");
  std::ofstream fout(Path + "config.txt");
  std::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 « std::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
                    std::ofstream ofs;
  ofs.open(Path + std::to_string(state) + "_"
          + std::to_string(lattice.my_prc) + ".csv");
  // Precision of sunrealtype in significant decimal digits; 15 for IEEE double
  ofs « std::setprecision(std::numeric_limits<sunrealtype>::digits10);

  // Walk through each lattice point
  const sunindextype totalNP = latticePatch.discreteSize();
  for (sunindextype i = 0; i < totalNP * 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] « ","
```

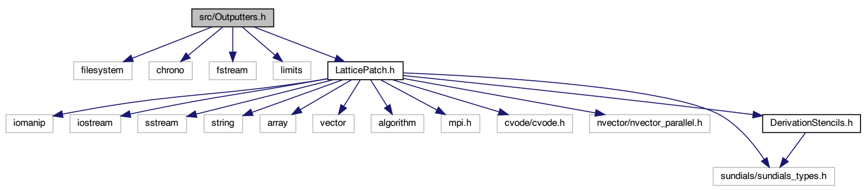
```
        « latticePatch.uData[i + 4] « "," « latticePatch.uData[i + 5]
        « std::endl;
    }
    ofs.close();
    break;
                }

      case 'b': { // a single binary file
    // Open the output file
    MPI_File fh;
    const std::string filename = Path+std::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 sunindextype 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_Request write_request;
    MPI_File_iwrite_all(fh,latticePatch.uData,count,MPI_SUNREALTYPE,
            &write_request);
    MPI_Wait(&write_request,MPI_STATUS_IGNORE);
    MPI_File_close(&fh);
    break;
                }
      default: {
    errorKill("No valid output style defined."
            " Choose between (c): one csv file per process,"
            " (b) one binary file");
    break;
               }
    }
}
```

6.18 src/Outputters.h File Reference

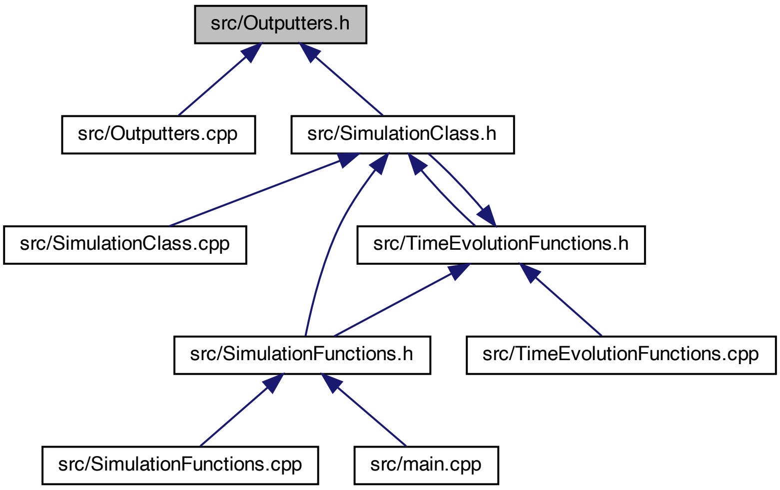
OutputManager class to outstream simulation data.

```
#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"

/** @brief Output Manager class to generate and coordinate output writing to
 * disk */
class OutputManager {
```

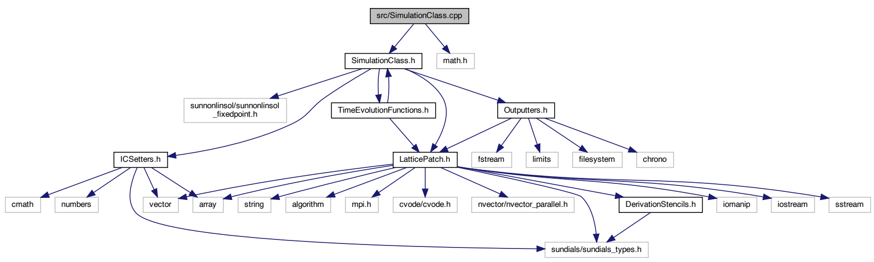
```
private:
  /// function to create the Code of the Simulations
  static std::string SimCodeGenerator();
  /// varible to safe the SimCode generated at execution
  std::string simCode;
  /// variable for the path to the output folder
  std::string Path;
  /// output style; csv or binary
  char outputStyle;
public:
  /// default constructor
  OutputManager();
  /// function that creates folder to save simulation data
  void generateOutputFolder(const std::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 std::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);
}

/// Free the CVode solver memory and Sundials context object with the finish of
/// the simulation
Simulation::~Simulation() {
  // Free solver memory
  if (statusFlags & CvodeObjectSetUp) {
    CVodeFree(&cvode_mem);
    SUNNonlinSolFree(NLS);
    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;

  // 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);

  // Initialize CVODE solver
  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);
```

```

  // Create fixed point nonlinear solver object (suitable for non-stiff ODE) and
  // attach it to CVode
  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);

  // Anderson damping factor
  retval = SUNNonlinSolSetDamping_FixedPoint(NLS,1);
  if (check_retval(&retval, "SUNNonlinSolSetDamping_FixedPoint", 1,
                   lattice.my_prc)) 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);

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

  // maximum number of warnings for too small h
  retval = CVodeSetMaxHnilWarns(cvode_mem,3);
  if (check_retval(&retval, "CVodeSetMaxHnilWarns", 1, lattice.my_prc))
    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 sunindextype nx = latticePatch.discreteSize(1);
  const sunindextype ny = latticePatch.discreteSize(2);
  const sunindextype totalNP = latticePatch.discreteSize();
  const sunrealtype x0 = latticePatch.origin(1);
  const sunrealtype y0 = latticePatch.origin(2);
  const sunrealtype z0 = latticePatch.origin(3);
  sunindextype px = 0, py = 0, pz = 0;
  #pragma omp parallel for default(none) \
  shared(nx, ny, totalNP, dx, dy, dz, x0, y0, z0) \
  firstprivate(px, py, pz) schedule(static)
  for (sunindextype i = 0; i < totalNP * 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 sunindextype xm,
        const sunindextype ym,
        const sunindextype zm /* zm=0 always */ ) {
  const sunrealtype dx = latticePatch.getDelta(1);
  const sunrealtype dy = latticePatch.getDelta(2);
  const sunrealtype dz = latticePatch.getDelta(3);
  const sunindextype nx = latticePatch.discreteSize(1);
  const sunindextype ny = latticePatch.discreteSize(2);
  const sunindextype totalNP = latticePatch.discreteSize();
  // 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();
  sunindextype px = 0, py = 0, pz = 0;
```

```
  for (sunindextype i = 0; i < totalNP * 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 retval = 0;
  retval = 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 (check_retval(&retval, "CVode", 1, lattice.my_prc))
    MPI_Abort(lattice.comm, 1);
}

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

/// Check presence of configuration flags
void Simulation::checkFlag(unsigned int flag) const {
  if (!(statusFlags & flag)) {
    std::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 absence of configuration flags
void Simulation::checkNoFlag(unsigned int flag) const {
  if ((statusFlags & flag)) {
    std::string errorMessage;
    switch (flag) {
    case LatticeDiscreteSetUp:
      errorMessage =
          "The discrete size of the Simulation has already been set up";
```

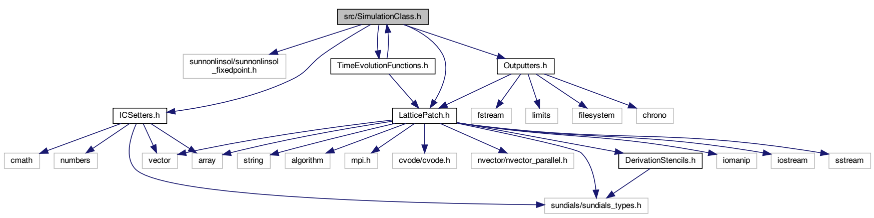
```
      break;
    case LatticePhysicalSetUp:
      errorMessage =
          "The physical size of the Simulation has already been set up";
      break;
    case LatticePatchworkSetUp:
      errorMessage = "The patchwork for the Simulation has already been set up";
      break;
    case CvodeObjectSetUp:
      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);
  }
  return;
}
```

6.22 src/SimulationClass.h File Reference

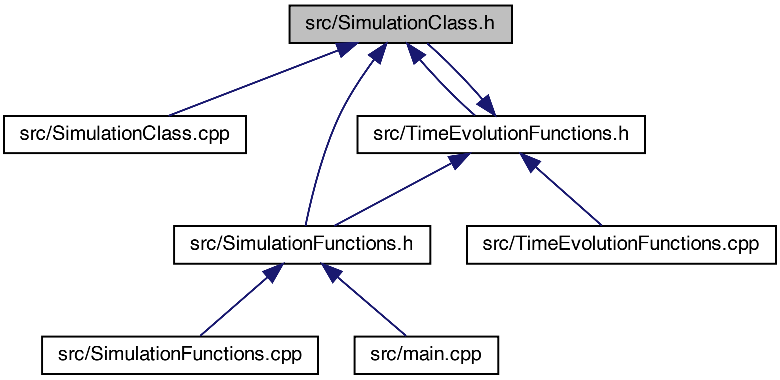
Class for the Simulation object calling all functionality: from wave settings over lattice construction, time evolution and outputs initialization of the CVode object.

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

Include dependency graph for SimulationClass.h:



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



Data Structures

- class Simulation

    Simulation class to instantiate the whole walkthrough of a Simulation.

Variables

- constexpr unsigned int LatticeDiscreteSetUp = 0x01
- constexpr unsigned int LatticePhysicalSetUp = 0x02
- constexpr unsigned int LatticePatchworkSetUp = 0x04
- constexpr unsigned int CvodeObjectSetUp = 0x08
- constexpr unsigned int SimulationStarted = 0x10

6.22.1 Detailed Description

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

Definition in file SimulationClass.h.

6.22.2 Variable Documentation

6.22.2.1 CvodeObjectSetUp

```
constexpr unsigned int CvodeObjectSetUp = 0x08 [constexpr]
```

simulation checking flag

Definition at line 24 of file SimulationClass.h.

Referenced by Simulation::checkFlag(), Simulation::checkNoFlag(), Simulation::initializeCVODEobject(), Simulation::start(), and Simulation::∼Simulation().

6.22.2.2 LatticeDiscreteSetUp

```
constexpr unsigned int LatticeDiscreteSetUp = 0x01 [constexpr]
```

simulation checking flag

Definition at line 21 of file SimulationClass.h.

Referenced by Simulation::checkFlag(), Simulation::checkNoFlag(), Simulation::initializePatchwork(), Simulation::setDiscreteDimensio
and Simulation::start().

6.22.2.3 LatticePatchworkSetUp

```
constexpr unsigned int LatticePatchworkSetUp = 0x04 [constexpr]
```

simulation checking flag

Definition at line 23 of file SimulationClass.h.

Referenced by Simulation::checkFlag(), Simulation::checkNoFlag(), Simulation::initializePatchwork(), Simulation::setDiscreteDimensio
Simulation::setPhysicalDimensionsOfLattice(), and Simulation::start().

6.22.2.4 LatticePhysicalSetUp

```
constexpr unsigned int LatticePhysicalSetUp = 0x02 [constexpr]
```

simulation checking flag

Definition at line 22 of file SimulationClass.h.

Referenced by Simulation::checkFlag(), Simulation::checkNoFlag(), Simulation::initializePatchwork(), Simulation::setPhysicalDimensio
and Simulation::start().

6.22.2.5 SimulationStarted

```
constexpr unsigned int SimulationStarted = 0x10  [constexpr]
```

simulation checking flag

Definition at line 25 of file SimulationClass.h.

Referenced by Simulation::advanceToTime(), Simulation::checkFlag(), Simulation::checkNoFlag(), Simulation::initializeCVODEobject(
Simulation::outAllFieldData(), and Simulation::start().

6.23 SimulationClass.h

Go to the documentation of this file.

```
00001 ////////////////////////////////////////////////////////////////////////////////
00002 /// @file SimulationClass.h
00003 /// @brief Class for the Simulation object calling all functionality:
00004 /// from wave settings over lattice construction, time evolution and outputs
00005 /// initialization of the CVode object
00006 ////////////////////////////////////////////////////////////////////////////////
00007
00008 #pragma once
00009
00010 /* access to the fixed point SUNNonlinearSolver */
00011 #include "sunnonlinsol/sunnonlinsol_fixedpoint.h"
00012
00013 // project subfile headers
00014 #include "ICSetters.h"
00015 #include "LatticePatch.h"
00016 #include "Outputters.h"
00017 #include "TimeEvolutionFunctions.h"
00018
00019 ///@{
00020 /** simulation checking flag */
00021 constexpr unsigned int LatticeDiscreteSetUp = 0x01;
00022 constexpr unsigned int LatticePhysicalSetUp = 0x02;
00023 constexpr unsigned int LatticePatchworkSetUp = 0x04; // not used anymore
00024 constexpr unsigned int CvodeObjectSetUp = 0x08;
00025 constexpr unsigned int SimulationStarted = 0x10;
00026 ///@}
00027
00028 /** @brief Simulation class to instantiate the whole walkthrough of a Simulation
00029  */
00030 class Simulation {
00031 private:
00032   /// Lattice object
00033   Lattice lattice;
00034   /// LatticePatch object
00035   LatticePatch latticePatch;
00036   /// current time of the simulation
00037   sunrealtype t;
00038   /// simulation status flags
00039   unsigned int statusFlags;
00040
00041 public:
00042   /// IC Setter object
00043   ICSetter icsettings;
00044   /// Output Manager object
00045   OutputManager outputManager;
00046   /// pointer to CVode memory object
00047   void *cvode_mem;
00048   /// nonlinear solver object
00049   SUNNonlinearSolver NLS;
00050   /// constructor function for the creation of the cartesian communicator
00051   Simulation(const int Nx, const int Ny, const int Nz, const int StencilOrder,
00052           const bool periodicity);
00053   /// destructor function freeing CVode memory and Sundials context
00054   ~Simulation();
00055   /// reference to the cartesian communicator of the lattice (for debugging)
00056   MPI_Comm *get_cart_comm() { return &lattice.comm; }
00057   /// function to set discrete dimensions of the lattice
00058   void setDiscreteDimensionsOfLattice(const sunindextype _tot_nx,
00059           const sunindextype _tot_ny, const sunindextype _tot_nz);
00060   /// function to set physical dimensions of the lattice
00061   void setPhysicalDimensionsOfLattice(const sunrealtype lx,
00062           const sunrealtype ly, const sunrealtype lz);
```

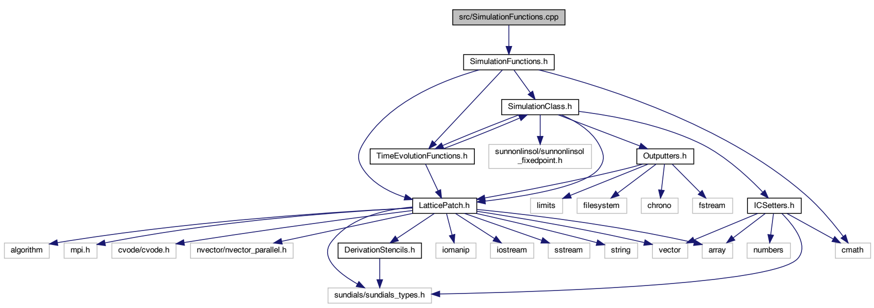
```
    /// function to initialize the Patchwork
    void initializePatchwork(const int nx, const int ny, const int nz);
    /// function to initialize the CVODE object with all requirements
    void initializeCVODEobject(const sunrealtype reltol,
                               const sunrealtype abstol);
    /// 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 sunindextype xm, const sunindextype ym,
            const sunindextype 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 write field data to disk
    void outAllFieldData(const int & state);
    /// function to check if flag has been set
    void checkFlag(unsigned int flag) const;
    /// function to check if flag has not been set
    // 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)

    MPI timer function.
- void Sim1D (const std::array< sunrealtype, 2 > CVodeTol, const int StencilOrder, const sunrealtype phys_↩ dim, const sunindextype disc_dim, const bool periodic, int ∗interactions, const sunrealtype endTime, const int numberOfSteps, const std::string outputDirectory, const int outputStep, const char outputStyle, const std↩ ::vector< planewave > &planes, const std::vector< gaussian1D > &gaussians)

    complete 1D Simulation function
- void Sim2D (const std::array< sunrealtype, 2 > CVodeTol, int const StencilOrder, const std::array< sunrealtype, 2 > phys_dims, const std::array< sunindextype, 2 > disc_dims, const std::array< int, 2 > patches, const bool periodic, int ∗interactions, const sunrealtype endTime, const int numberOfSteps, const std::string outputDirectory, const int outputStep, const char outputStyle, const std::vector< planewave > &planes, const std::vector< gaussian2D > &gaussians)

complete 2D Simulation function

- void Sim3D (const std::array< sunrealtype, 2 > CVodeTol, const int StencilOrder, const std::array< sunrealtype, 3 > phys_dims, const std::array< sunindextype, 3 > disc_dims, const std::array< int, 3 > patches, const bool periodic, int ∗interactions, const sunrealtype endTime, const int numberOfSteps, const std::string outputDirectory, const int outputStep, const char outputStyle, const std::vector< planewave > &planes, const std::vector< gaussian3D > &gaussians)

  complete 3D Simulation function

6.24.1 Detailed Description

Implementation of the complete simulation functions for 1D, 2D, and 3D, as called in the main function.

Definition in file SimulationFunctions.cpp.

6.24.2 Function Documentation

6.24.2.1 Sim1D()

```
void Sim1D (
            const std::array< sunrealtype, 2 > CVodeTol,
            const int StencilOrder,
            const sunrealtype phys_dim,
            const sunindextype disc_dim,
            const bool periodic,
            int * interactions,
            const sunrealtype endTime,
            const int numberOfSteps,
            const std::string outputDirectory,
            const int outputStep,
            const char outputStyle,
            const std::vector< planewave > & planes,
            const std::vector< gaussian1D > & gaussians )
```

complete 1D Simulation function

Conduct the complete 1D simulation process

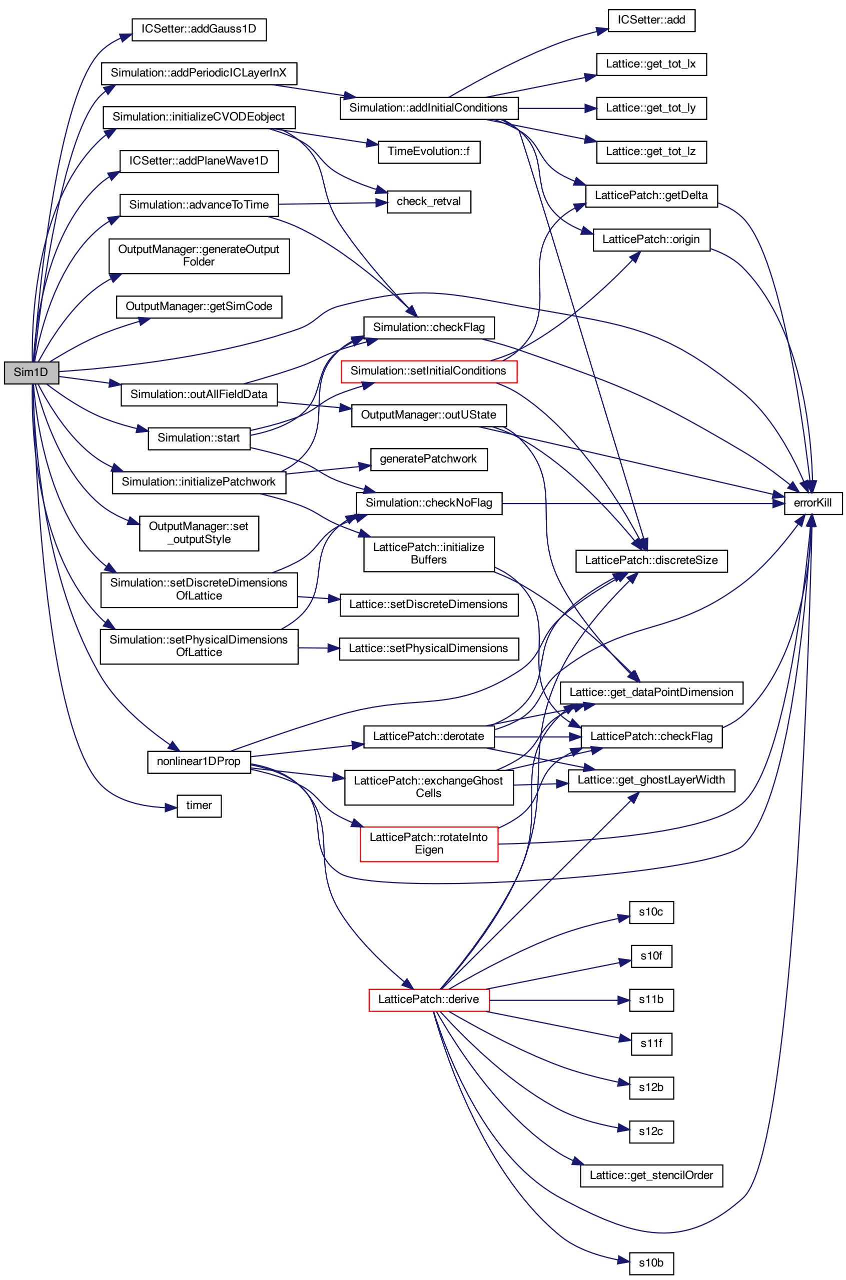
Definition at line 21 of file SimulationFunctions.cpp.

```
                                              {

  // MPI data
  double ts = MPI_Wtime();
  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
  std::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) {
    std::cout « "Simulation code: " « sim.outputManager.getSimCode()
        « std::endl;
  }
  sim.outputManager.set_outputStyle(outputStyle);

  //MPI_Barrier(MPI_COMM_WORLD);  // insure correct output
  //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) {
      MPI_Barrier(MPI_COMM_WORLD);  // insure correct output
      sim.outAllFieldData(step);
    }
    double tn = MPI_Wtime();
    if (!myPrc) {
      std::cout « "\rStep " « step « "\t\t" « std::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().

Here is the call graph for this function:



6.24.2.2 Sim2D()

```
void Sim2D (
            const std::array< sunrealtype, 2 > CVodeTol,
            int const StencilOrder,
            const std::array< sunrealtype, 2 > phys_dims,
            const std::array< sunindextype, 2 > disc_dims,
```

```
const std::array< int, 2 > patches,
const bool periodic,
int * interactions,
const sunrealtype endTime,
const int numberOfSteps,
const std::string outputDirectory,
const int outputStep,
const char outputStyle,
const std::vector< planewave > & planes,
const std::vector< gaussian2D > & gaussians )
```

complete 2D Simulation function

Conduct the complete 2D simulation process

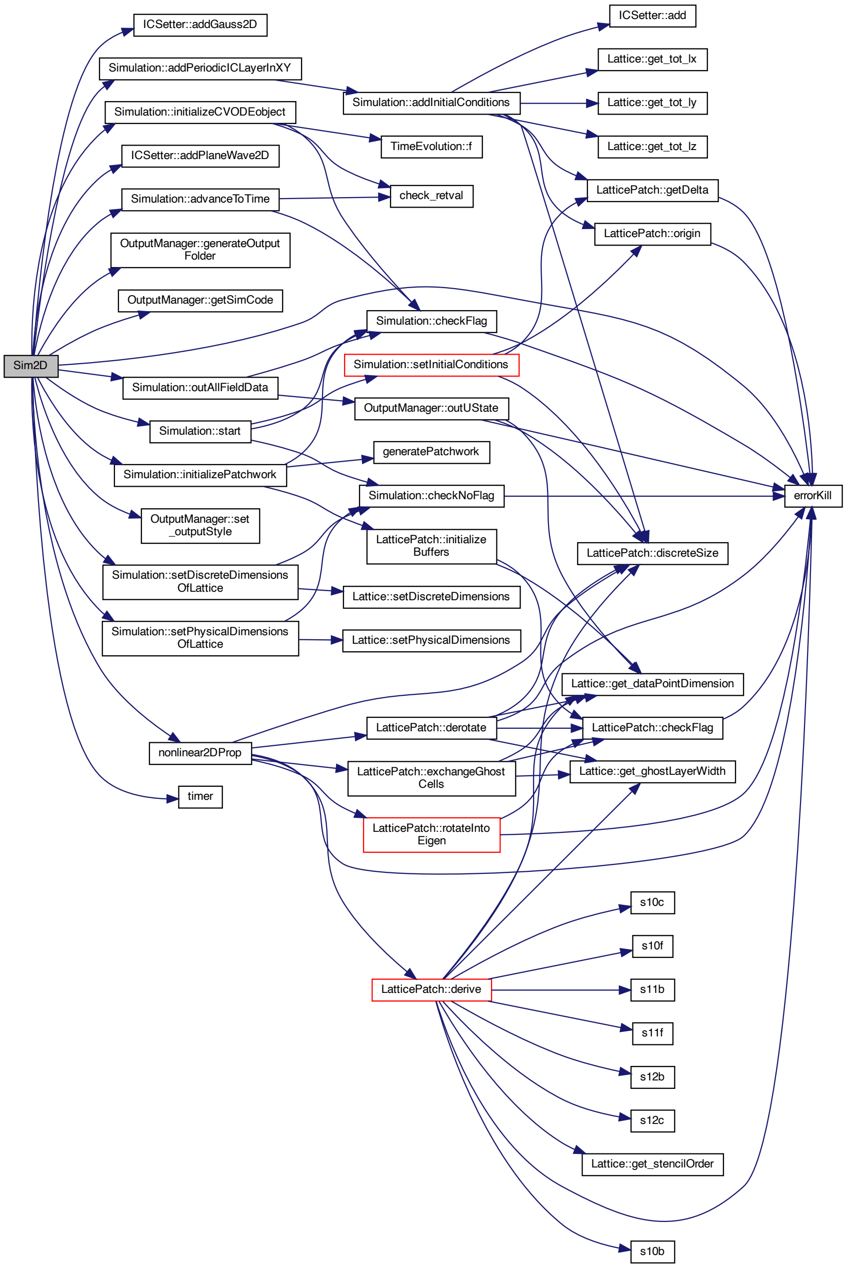
Definition at line 99 of file SimulationFunctions.cpp.

```
                                                {

  // MPI data
  double ts = MPI_Wtime();
  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);

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

  //MPI_Barrier(MPI_COMM_WORLD);  // insure correct output
  //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) {
      MPI_Barrier(MPI_COMM_WORLD);  // insure correct output
      sim.outAllFieldData(step);
```

```
    }
    double tn = MPI_Wtime();
    if (!myPrc) {
      std::cout « "\rStep " « step « "\t\t" « std::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().

Here is the call graph for this function:



6.24.2.3 Sim3D()

```
void Sim3D (
        const std::array< sunrealtype, 2 > CVodeTol,
        const int StencilOrder,
        const std::array< sunrealtype, 3 > phys_dims,
        const std::array< sunindextype, 3 > disc_dims,
        const std::array< int, 3 > patches,
        const bool periodic,
        int * interactions,
        const sunrealtype endTime,
        const int numberOfSteps,
        const std::string outputDirectory,
        const int outputStep,
        const char outputStyle,
        const std::vector< planewave > & planes,
        const std::vector< gaussian3D > & gaussians )
```

complete 3D Simulation function

Conduct the complete 3D simulation process

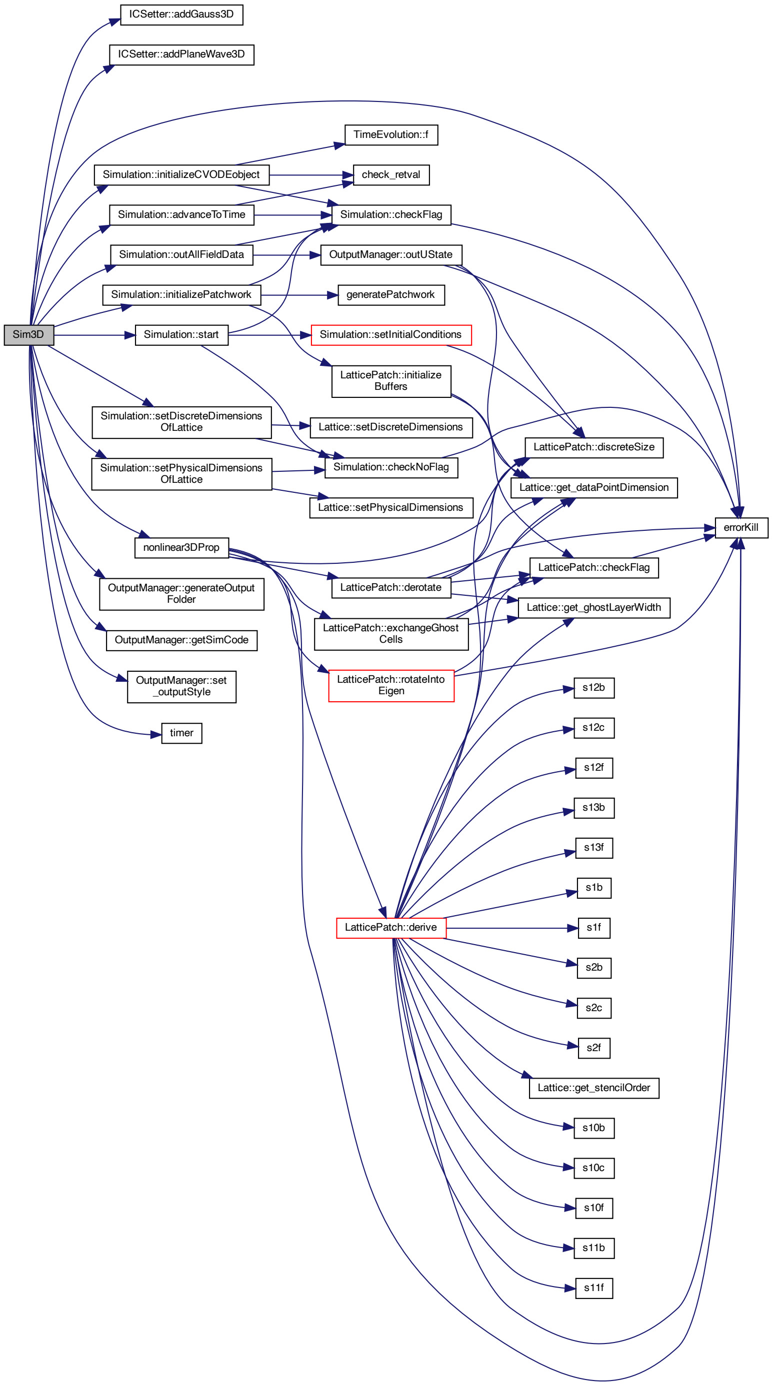
Definition at line 183 of file SimulationFunctions.cpp.

```
                                                 {

  // MPI data
  double ts = MPI_Wtime();
  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] ) ) {
        std::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) {
    std::cout « "Simulation code: " « sim.outputManager.getSimCode()
        « std::endl;
  }
  sim.outputManager.set_outputStyle(outputStyle);

  //MPI_Barrier(MPI_COMM_WORLD);  // insure correct output
  //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) {
      MPI_Barrier(MPI_COMM_WORLD);  // insure correct output
      sim.outAllFieldData(step);
    }
    double tn = MPI_Wtime();
    if (!myPrc) {
      std::cout « "\rStep " « step « "\t\t" « std::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().

Here is the call graph for this function:



6.24.2.4 timer()

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

MPI timer function.

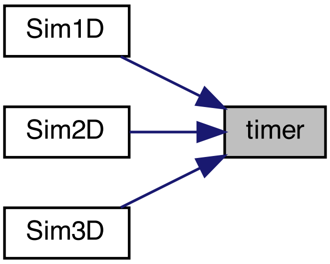
Calculate and print the total simulation time

Definition at line 10 of file SimulationFunctions.cpp.

```
00010                                             {
00011   printf("Elapsed time:  %fs\n", (t2 - t1));
00012 }
```

Referenced by Sim1D(), Sim2D(), and Sim3D().

Here is the caller graph for this function:



6.25 SimulationFunctions.cpp

Go to the documentation of this file.

```
00001 ///////////////////////////////////////////////////////////////////////
00002 /// @file SimulationFunctions.cpp
00003 /// @brief Implementation of the complete simulation functions for
00004 /// 1D, 2D, and 3D, as called in the main function
00005 ///////////////////////////////////////////////////////////////////////
00006
00007 #include "SimulationFunctions.h"
00008
00009 /** Calculate and print the total simulation time */
00010 inline void timer(double &t1, double &t2) {
00011   printf("Elapsed time:  %fs\n", (t2 - t1));
00012 }
00013
00014 // Instantiate and preliminarily initialize the time evolver
00015 // non-const statics to be defined in actual simulation process
00016 int *TimeEvolution::c = nullptr;
00017 void (*TimeEvolution::TimeEvolver)(LatticePatch *, N_Vector, N_Vector,
00018                                    int *) = nonlinear1DProp;
00019
00020 /** Conduct the complete 1D simulation process */
00021 void Sim1D(const std::array<sunrealtype,2> CVodeTol, const int StencilOrder,
00022         const sunrealtype phys_dim, const sunindextype disc_dim,
00023         const bool periodic, int *interactions,
00024         const sunrealtype endTime, const int numberOfSteps,
00025         const std::string outputDirectory, const int outputStep,
00026         const char outputStyle,
00027         const std::vector<planewave> &planes,
00028         const std::vector<gaussian1D> &gaussians) {
00029
00030   // MPI data
00031   double ts = MPI_Wtime();
00032   int myPrc = 0, nPrc = 0;
00033   MPI_Comm_size(MPI_COMM_WORLD, &nPrc);
00034   MPI_Comm_rank(MPI_COMM_WORLD, &myPrc);
00035
00036   // Check feasibility of the patchwork decomposition
```

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

  //MPI_Barrier(MPI_COMM_WORLD);  // insure correct output
  //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) {
      MPI_Barrier(MPI_COMM_WORLD);  // insure correct output
      sim.outAllFieldData(step);
    }
    double tn = MPI_Wtime();
    if (!myPrc) {
      std::cout « "\rStep " « step « "\t\t" « std::flush;
      timer(ts, tn);
    }
  }

  return;
}

/** Conduct the complete 2D simulation process */
void Sim2D(const std::array<sunrealtype,2> CVodeTol, int const StencilOrder,
        const std::array<sunrealtype,2> phys_dims,
        const std::array<sunindextype,2> disc_dims,
        const std::array<int,2> patches, const bool periodic, int *interactions,
        const sunrealtype endTime, const int numberOfSteps,
        const std::string outputDirectory, const int outputStep,
        const char outputStyle,
        const std::vector<planewave> &planes,
        const std::vector<gaussian2D> &gaussians) {

  // MPI data
  double ts = MPI_Wtime();
  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);

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

  //MPI_Barrier(MPI_COMM_WORLD);  // insure correct output
  //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) {
      MPI_Barrier(MPI_COMM_WORLD);  // insure correct output
      sim.outAllFieldData(step);
    }
    double tn = MPI_Wtime();
    if (!myPrc) {
      std::cout « "\rStep " « step « "\t\t" « std::flush;
      timer(ts, tn);
    }
  }

  return;
}

/** Conduct the complete 3D simulation process */
void Sim3D(const std::array<sunrealtype,2> CVodeTol, const int StencilOrder,
        const std::array<sunrealtype,3> phys_dims,
        const std::array<sunindextype,3> disc_dims,
        const std::array<int,3> patches,
        const bool periodic, int *interactions, const sunrealtype endTime,
        const int numberOfSteps, const std::string outputDirectory,
        const int outputStep, const char outputStyle,
        const std::vector<planewave> &planes,
        const std::vector<gaussian3D> &gaussians) {

  // MPI data
  double ts = MPI_Wtime();
  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] ) ) {
        std::clog
```

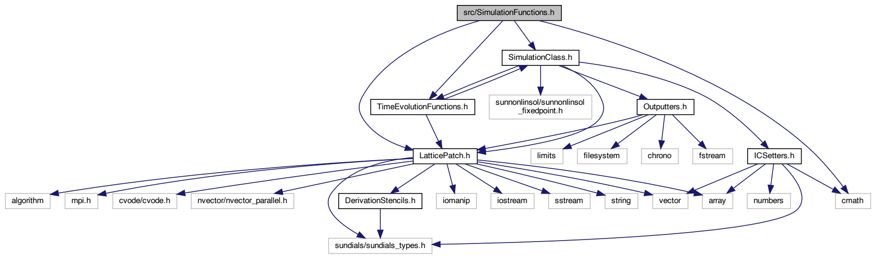
```
          « "\nWarning: Patches should be cubic in terms of the lattice "
             "points for the computational efficiency of larger simulations.\n";
    }
    */
  }

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

  //MPI_Barrier(MPI_COMM_WORLD);  // insure correct output
  //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) {
      MPI_Barrier(MPI_COMM_WORLD);  // insure correct output
      sim.outAllFieldData(step);
    }
    double tn = MPI_Wtime();
    if (!myPrc) {
      std::cout « "\rStep " « step « "\t\t" « std::flush;
      timer(ts, tn);
    }
  }
  return;
}
```

6.26 src/SimulationFunctions.h File Reference

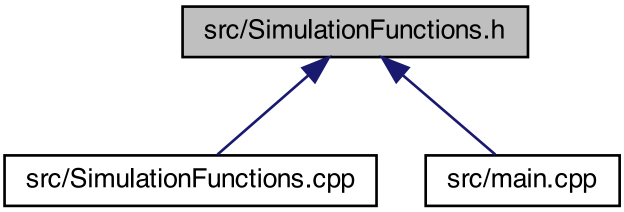
Full simulation functions for 1D, 2D, and 3D used in main.cpp.

```
#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 std::array< sunrealtype, 2 >, const int, const sunrealtype, const sunindextype, const bool, int ∗, const sunrealtype, const int, const std::string, const int, const char, const std::vector< planewave > &, const std::vector< gaussian1D > &)

  complete 1D Simulation function
- void Sim2D (const std::array< sunrealtype, 2 >, const int, const std::array< sunrealtype, 2 >, const std↩ ::array< sunindextype, 2 >, const std::array< int, 2 >, const bool, int ∗, const sunrealtype, const int, const std::string, const int, const char, const std::vector< planewave > &, const std::vector< gaussian2D > &)

  complete 2D Simulation function

- void Sim3D (const std::array< sunrealtype, 2 >, const int, const std::array< sunrealtype, 3 >, const std::array< sunindextype, 3 >, const std::array< int, 3 >, const bool, int ∗, const sunrealtype, const int, const std::string, const int, const char, const std::vector< planewave > &, const std::vector< gaussian3D > &)

    complete 3D Simulation function
- void timer (double &, double &)

    MPI timer function.

6.26.1 Detailed Description

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

Definition in file SimulationFunctions.h.

6.26.2 Function Documentation

6.26.2.1 Sim1D()

```
void Sim1D (
            const std::array< sunrealtype, 2 > CVodeTol,
            const int StencilOrder,
            const sunrealtype phys_dim,
            const sunindextype disc_dim,
            const bool periodic,
            int * interactions,
            const sunrealtype endTime,
            const int numberOfSteps,
            const std::string outputDirectory,
            const int outputStep,
            const char outputStyle,
            const std::vector< planewave > & planes,
            const std::vector< gaussian1D > & gaussians )
```

complete 1D Simulation function

Conduct the complete 1D simulation process

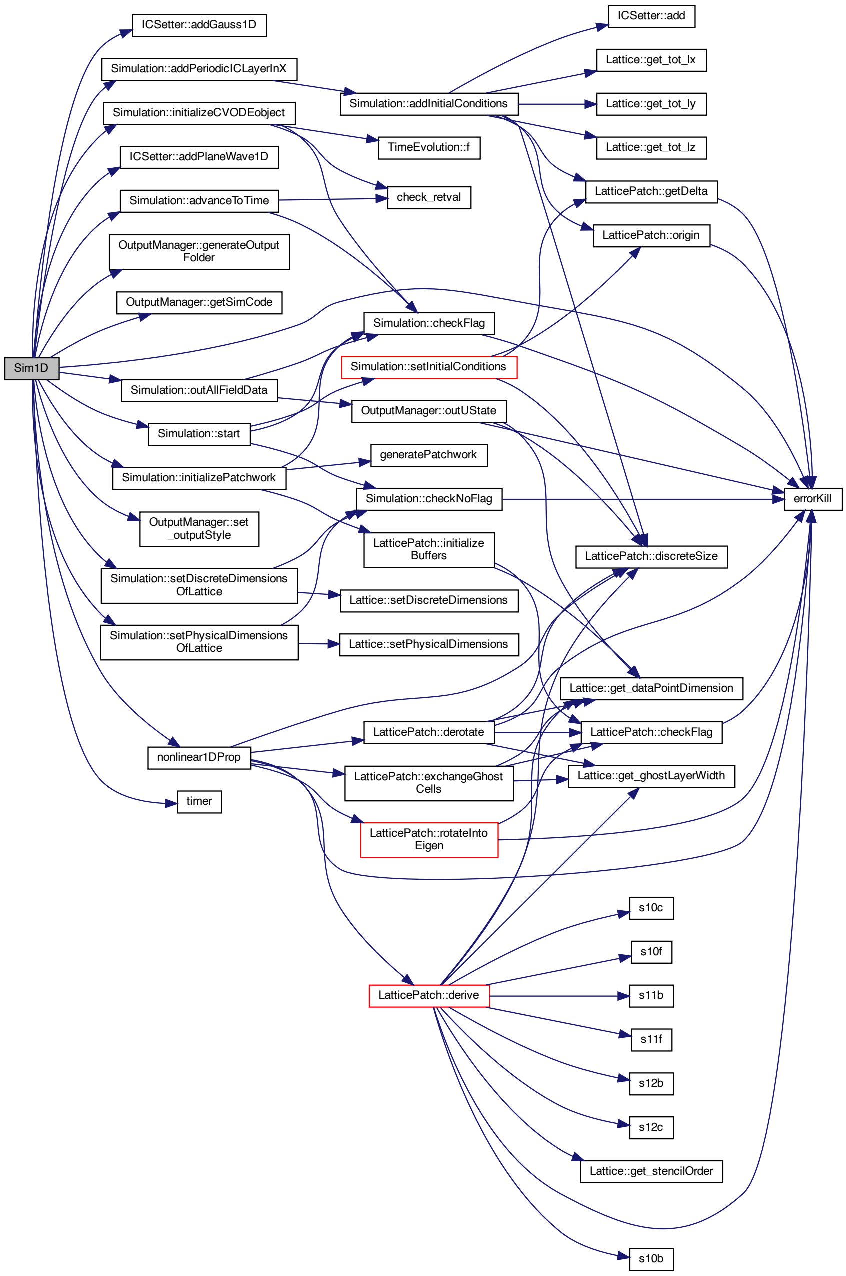
Definition at line 21 of file SimulationFunctions.cpp.

```
                                                     {

    // MPI data
    double ts = MPI_Wtime();
    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
    std::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) {
    std::cout « "Simulation code: " « sim.outputManager.getSimCode()
        « std::endl;
  }
  sim.outputManager.set_outputStyle(outputStyle);

  //MPI_Barrier(MPI_COMM_WORLD);  // insure correct output
  //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) {
      MPI_Barrier(MPI_COMM_WORLD);  // insure correct output
      sim.outAllFieldData(step);
    }
    double tn = MPI_Wtime();
    if (!myPrc) {
      std::cout « "\rStep " « step « "\t\t" « std::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().

Here is the call graph for this function:



6.26.2.2 Sim2D()

```
void Sim2D (
            const std::array< sunrealtype, 2 > CVodeTol,
            int const StencilOrder,
            const std::array< sunrealtype, 2 > phys_dims,
            const std::array< sunindextype, 2 > disc_dims,
```

```
const std::array< int, 2 > patches,
const bool periodic,
int * interactions,
const sunrealtype endTime,
const int numberOfSteps,
const std::string outputDirectory,
const int outputStep,
const char outputStyle,
const std::vector< planewave > & planes,
const std::vector< gaussian2D > & gaussians )
```

complete 2D Simulation function

Conduct the complete 2D simulation process

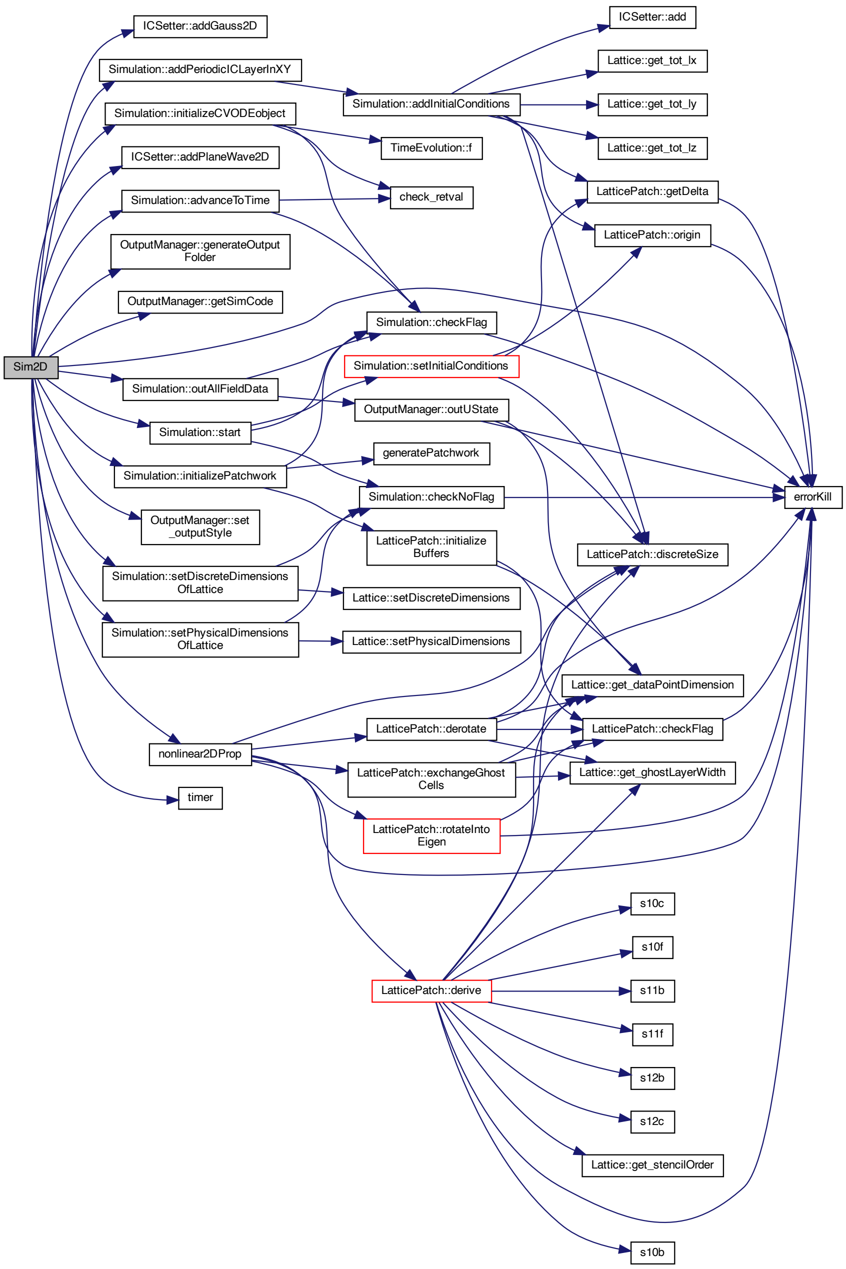
Definition at line 99 of file SimulationFunctions.cpp.

```
                                                  {

  // MPI data
  double ts = MPI_Wtime();
  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);

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

  //MPI_Barrier(MPI_COMM_WORLD);  // insure correct output
  //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) {
      MPI_Barrier(MPI_COMM_WORLD);  // insure correct output
      sim.outAllFieldData(step);
```

```
    }
    double tn = MPI_Wtime();
    if (!myPrc) {
      std::cout « "\rStep " « step « "\t\t" « std::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().

Here is the call graph for this function:



6.26.2.3 Sim3D()

```
void Sim3D (
        const std::array< sunrealtype, 2 > CVodeTol,
        const int StencilOrder,
        const std::array< sunrealtype, 3 > phys_dims,
        const std::array< sunindextype, 3 > disc_dims,
        const std::array< int, 3 > patches,
        const bool periodic,
        int * interactions,
        const sunrealtype endTime,
        const int numberOfSteps,
        const std::string outputDirectory,
        const int outputStep,
        const char outputStyle,
        const std::vector< planewave > & planes,
        const std::vector< gaussian3D > & gaussians )
```

complete 3D Simulation function

Conduct the complete 3D simulation process

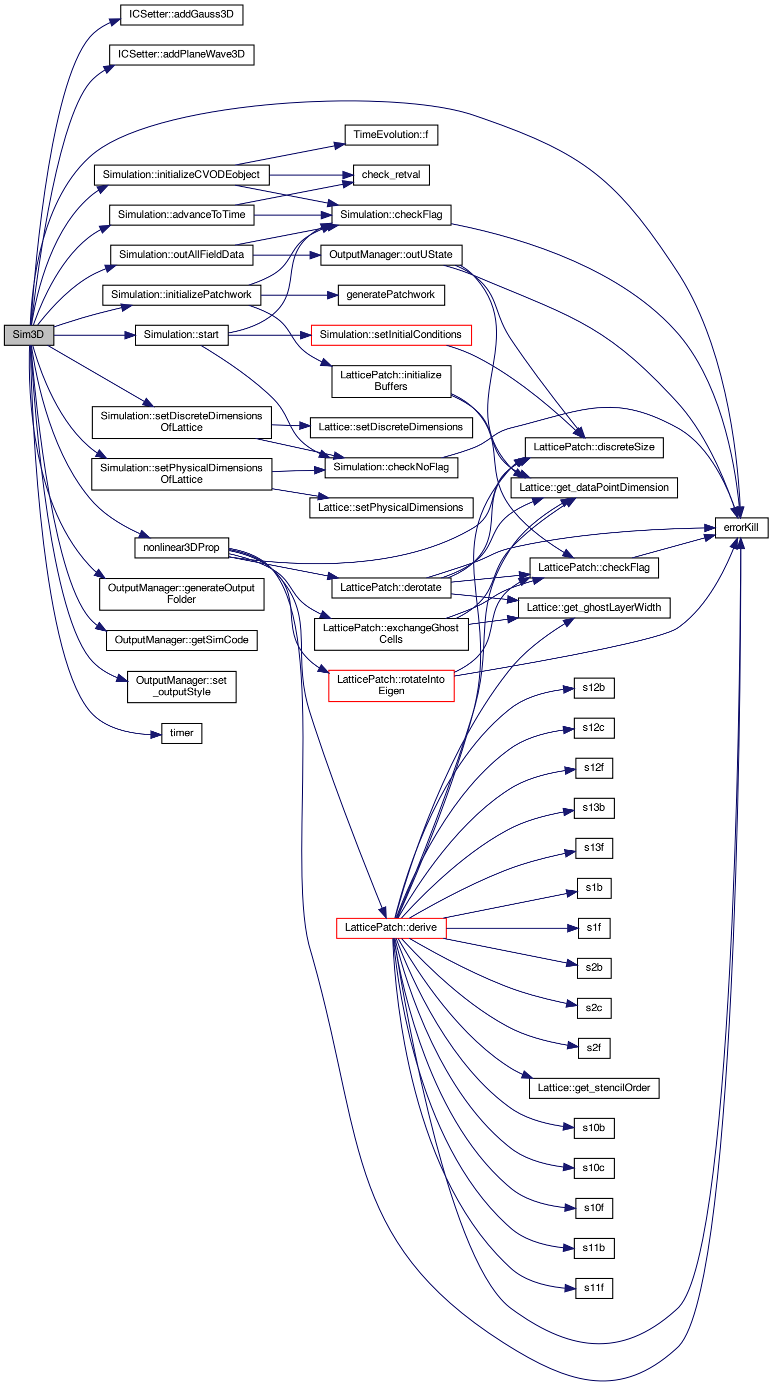
Definition at line 183 of file SimulationFunctions.cpp.

```
                                                {

  // MPI data
  double ts = MPI_Wtime();
  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] ) ) {
        std::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) {
    std::cout « "Simulation code: " « sim.outputManager.getSimCode()
        « std::endl;
  }
  sim.outputManager.set_outputStyle(outputStyle);

  //MPI_Barrier(MPI_COMM_WORLD);  // insure correct output
  //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) {
      MPI_Barrier(MPI_COMM_WORLD);  // insure correct output
      sim.outAllFieldData(step);
    }
    double tn = MPI_Wtime();
    if (!myPrc) {
      std::cout « "\rStep " « step « "\t\t" « std::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().

Here is the call graph for this function:



6.26.2.4 timer()

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

MPI timer function.

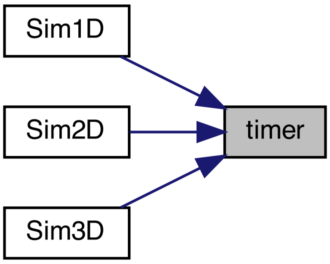
Calculate and print the total simulation time

Definition at line 10 of file SimulationFunctions.cpp.

```
00010                                             {
00011   printf("Elapsed time:  %fs\n", (t2 - t1));
00012 }
```

Referenced by Sim1D(), Sim2D(), and Sim3D().

Here is the caller graph for this function:



6.27 SimulationFunctions.h

Go to the documentation of this file.

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

#pragma once

// math
#include <cmath>

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

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

/// plane wave structure
struct planewave {
  std::array<sunrealtype, 3> k;   /**< wavevector (normalized to \f$ 1/\lambda \f$) */
  std::array<sunrealtype, 3> p;   /**< amplitde & polarization vector */
  std::array<sunrealtype, 3> phi; /**< phase shift */
};

/// 1D Gaussian wave structure
struct gaussian1D {
  std::array<sunrealtype, 3> k;   /**< wavevector (normalized to \f$ 1/\lambda \f$) */
  std::array<sunrealtype, 3> p;   /**< amplitude & polarization vector */
  std::array<sunrealtype, 3> x0;  /**< shift from origin */
  sunrealtype phig;        /**< width */
  std::array<sunrealtype, 3> phi; /**< phase shift */
};

/// 2D Gaussian wave structure
struct gaussian2D {
  std::array<sunrealtype, 3> x0;   /**< center */
```

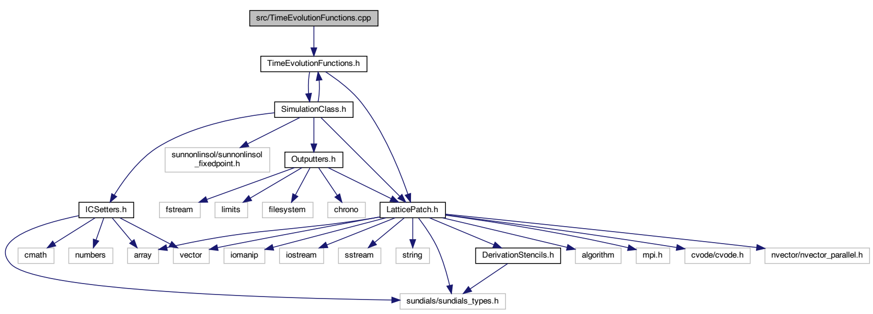
```
  std::array<sunrealtype, 3> axis; /**< direction from where it comes */
  sunrealtype amp;           /**< amplitude */
  sunrealtype phip;          /**< polarization rotation */
  sunrealtype w0;            /**< taille */
  sunrealtype zr;            /**< Rayleigh length */
  sunrealtype ph0;           /**< beam center */
  sunrealtype phA;           /**< beam length */
};

/// 3D Gaussian wave structure
struct gaussian3D {
  std::array<sunrealtype, 3> x0;   /**< center */
  std::array<sunrealtype, 3> axis; /**< direction from where it comes */
  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 std::array<sunrealtype,2>, const int, const sunrealtype,
        const sunindextype, const bool, int *, const sunrealtype, const int,
        const std::string, const int, const char,
        const std::vector<planewave> &,
        const std::vector<gaussian1D> &);
/// complete 2D Simulation function
void Sim2D(const std::array<sunrealtype,2>, const int,
        const std::array<sunrealtype,2>,
        const std::array<sunindextype,2>, const std::array<int,2>,
        const bool, int *,
        const sunrealtype, const int, const std::string,
        const int, const char,
        const std::vector<planewave> &, const std::vector<gaussian2D> &);
/// complete 3D Simulation function
void Sim3D(const std::array<sunrealtype,2>, const int,
        const std::array<sunrealtype,3>,
        const std::array<sunindextype,3>, const std::array<int,3>,
        const bool, int *,
        const sunrealtype, const int, const std::string,
        const int, const char,
        const std::vector<planewave> &, const std::vector<gaussian3D> &);

/// 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 dependency graph for TimeEvolutionFunctions.cpp:



Functions

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

  only under-the-hood-callable Maxwell propagation in 1D;
- void nonlinear1DProp (LatticePatch ∗data, N_Vector u, N_Vector udot, int ∗c)

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

  only under-the-hood-callable Maxwell propagation in 2D
- void nonlinear2DProp (LatticePatch ∗data, N_Vector u, N_Vector udot, int ∗c)

  nonlinear 2D HE propagation
- void linear3DProp (LatticePatch ∗data, N_Vector u, N_Vector udot, int ∗c)

  only under-the-hood-callable Maxwell propagation in 3D
- void nonlinear3DProp (LatticePatch ∗data, N_Vector u, N_Vector udot, int ∗c)

  nonlinear 3D HE propagation

6.28.1 Detailed Description

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

Definition in file TimeEvolutionFunctions.cpp.

6.28.2 Function Documentation

6.28.2.1 linear1DProp()

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

only under-the-hood-callable Maxwell propagation in 1D;

Maxwell propagation function for 1D – only for reference.

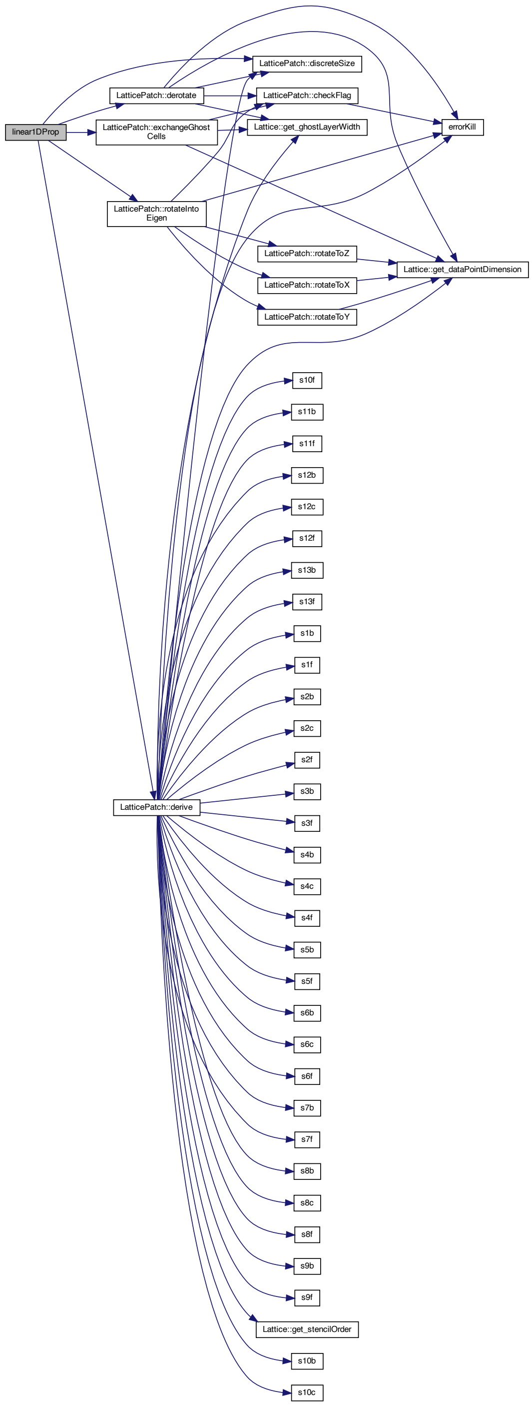
Definition at line 43 of file TimeEvolutionFunctions.cpp.

```
00043                                                                          {
00044
00045   // pointers to temporal and spatial derivative data
00046   sunrealtype *duData = data->duData;
00047   sunrealtype *dxData = data->buffData[1 - 1];
00048
00049   // sequence along any dimension according to the scheme:
00050   data->exchangeGhostCells(1); // -> exchange halos
00051   data->rotateIntoEigen(
00052       1);           // -> rotate all data to prepare derivative operation
00053   data->derive(1); // -> perform derivative approximation operation on it
00054   data->derotate(
00055       1, dxData); // -> derotate derived data for ensuing time-evolution
00056
00057   const sunindextype totalNP = data->discreteSize();
00058   sunindextype pp = 0;
00059   for (sunindextype i = 0; i < totalNP; i++) {
00060     pp = i * 6;
```

```
    /*
     simple vacuum Maxwell equations for the temporal derivatives using the
     spatial deriative only in x-direction without 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];
  }
}
```

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

Here is the call graph for this function:



6.28.2.2 linear2DProp()

```
void linear2DProp (
            LatticePatch * data,
```

```
N_Vector u,
N_Vector udot,
int * c )
```

only under-the-hood-callable Maxwell propagation in 2D

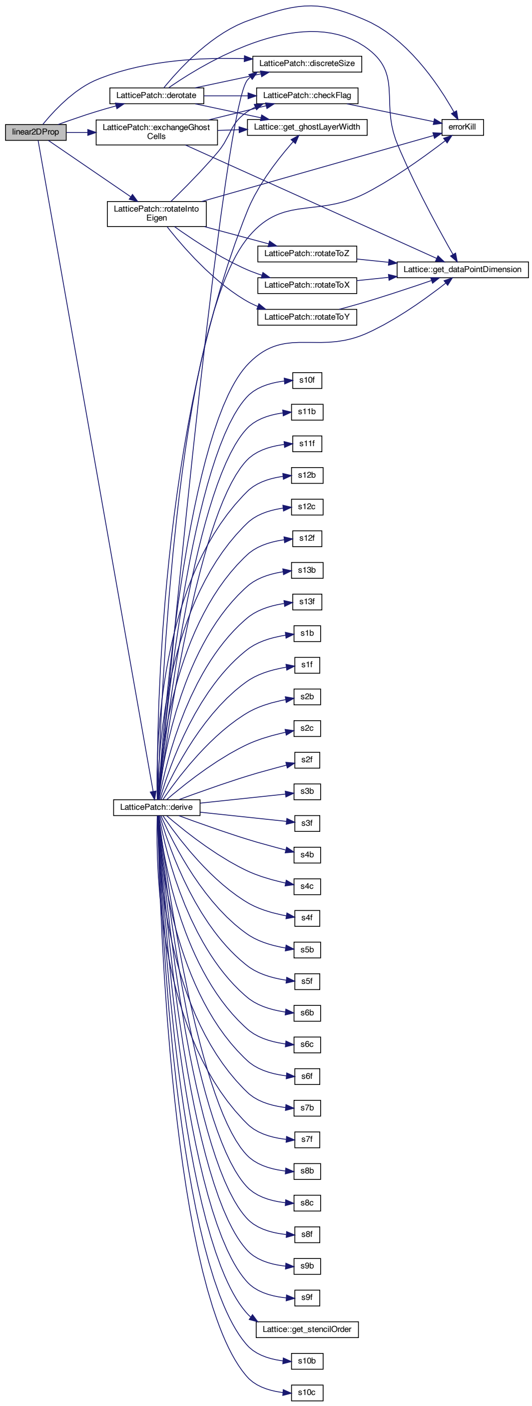
Maxwell propagation function for 2D – only for reference.

Definition at line 286 of file TimeEvolutionFunctions.cpp.

```
00286                                                                        {
00287
00288   sunrealtype *duData = data->duData;
00289   sunrealtype *dxData = data->buffData[1 - 1];
00290   sunrealtype *dyData = data->buffData[2 - 1];
00291
00292   data->exchangeGhostCells(1);
00293   data->rotateIntoEigen(1);
00294   data->derive(1);
00295   data->derotate(1, dxData);
00296   data->exchangeGhostCells(2);
00297   data->rotateIntoEigen(2);
00298   data->derive(2);
00299   data->derotate(2, dyData);
00300
00301   const sunindextype totalNP = data->discreteSize();
00302   sunindextype pp = 0;
00303   for (sunindextype i = 0; i < totalNP; i++) {
00304     pp = i * 6;
00305     duData[pp + 0] = dyData[pp + 5];
00306     duData[pp + 1] = -dxData[pp + 5];
00307     duData[pp + 2] = -dyData[pp + 3] + dxData[pp + 4];
00308     duData[pp + 3] = -dyData[pp + 2];
00309     duData[pp + 4] = dxData[pp + 2];
00310     duData[pp + 5] = dyData[pp + 0] - dxData[pp + 1];
00311   }
00312 }
```

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

Here is the call graph for this function:



6.28.2.3 linear3DProp()

```
void linear3DProp (
            LatticePatch * data,
```

```
N_Vector u,
N_Vector udot,
int * c )
```

only under-the-hood-callable Maxwell propagation in 3D

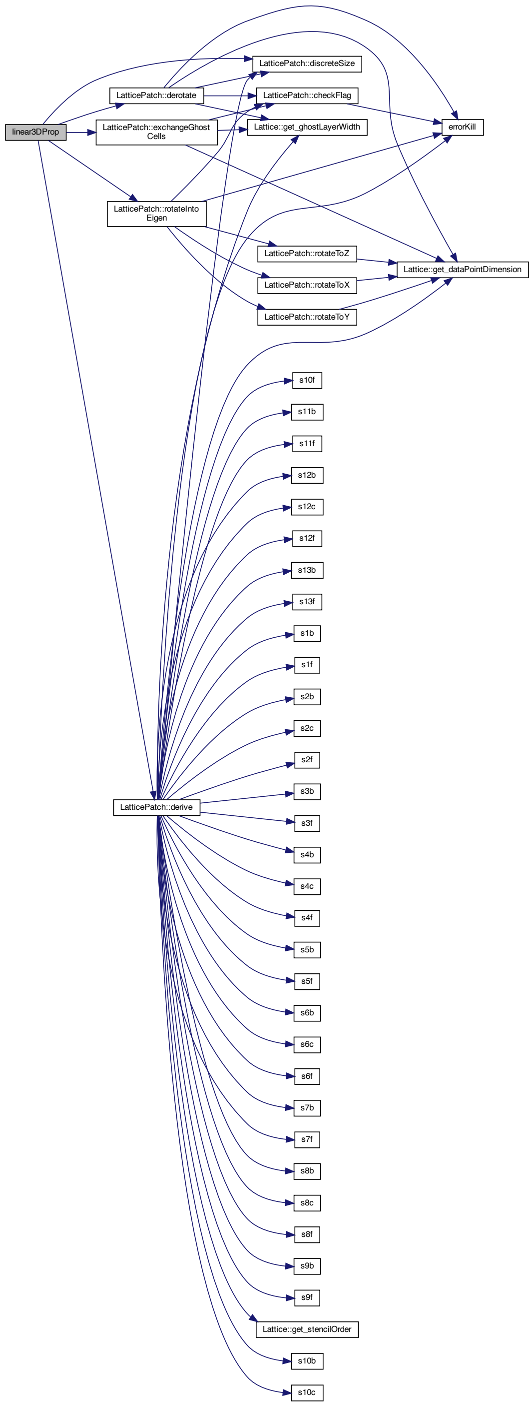
Maxwell propagation function for 3D – only for reference.

Definition at line 494 of file TimeEvolutionFunctions.cpp.

```
00494                                                                  {
00495
00496   sunrealtype *duData = data->duData;
00497   sunrealtype *dxData = data->buffData[1 - 1];
00498   sunrealtype *dyData = data->buffData[2 - 1];
00499   sunrealtype *dzData = data->buffData[3 - 1];
00500
00501   data->exchangeGhostCells(1);
00502   data->rotateIntoEigen(1);
00503   data->derive(1);
00504   data->derotate(1, dxData);
00505   data->exchangeGhostCells(2);
00506   data->rotateIntoEigen(2);
00507   data->derive(2);
00508   data->derotate(2, dyData);
00509   data->exchangeGhostCells(3);
00510   data->rotateIntoEigen(3);
00511   data->derive(3);
00512   data->derotate(3, dzData);
00513
00514   const sunindextype totalNP = data->discreteSize();
00515   sunindextype pp = 0;
00516   for (sunindextype i = 0; i < totalNP; i++) {
00517     pp = i * 6;
00518     duData[pp + 0] = dyData[pp + 5] - dzData[pp + 4];
00519     duData[pp + 1] = dzData[pp + 3] - dxData[pp + 5];
00520     duData[pp + 2] = dxData[pp + 4] - dyData[pp + 3];
00521     duData[pp + 3] = -dyData[pp + 2] + dzData[pp + 1];
00522     duData[pp + 4] = -dzData[pp + 0] + dxData[pp + 2];
00523     duData[pp + 5] = -dxData[pp + 1] + dyData[pp + 0];
00524   }
00525 }
```

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

Here is the call graph for this function:



6.28.2.4 nonlinear1DProp()

```
void nonlinear1DProp (
            LatticePatch * data,
```

```
N_Vector u,
N_Vector udot,
int * c )
```

nonlinear 1D HE propagation

HE propagation function for 1D.

Definition at line 76 of file TimeEvolutionFunctions.cpp.

```
00076                                                                                  {
00077
00078   // NVector pointers to provided field values and their temp. derivatives
00079   sunrealtype *udata = NV_DATA_P(u),
00080               *dudata = NV_DATA_P(udot);
00081
00082   // pointer to spatial derivatives via pach data
00083   sunrealtype *dxData = data->buffData[1 - 1];
00084
00085   // same sequence as in the linear case
00086   data->exchangeGhostCells(1);
00087   data->rotateIntoEigen(1);
00088   data->derive(1);
00089   data->derotate(1, dxData);
00090
00091   /*
00092   F and G are nonzero in the nonlinear case,
00093   polarization and magnetization derivatives
00094   w.r.t. E- and B-field go into the e.o.m.
00095   */
00096   static sunrealtype f, g; // em field invariants F, G
00097   // derivatives of HE Lagrangian w.r.t. field invariants
00098   static sunrealtype lf, lff, lfg, lg, lgg;
00099   // matrix to hold derivatives of polarization and magnetization
00100   static std::array<sunrealtype, 21> JMM;
00101   // array to hold E^2 and B^2 components
00102   static std::array<sunrealtype, 6> Quad;
00103   // array to hold intermediate temp. derivatives of E and B
00104   static std::array<sunrealtype, 6> h;
00105   // determinant needed for explicit matrix inversion
00106   static sunrealtype detC = nan("0x12345");
00107
00108   // number of points in the patch
00109   const sunindextype totalNP = data->discreteSize();
00110   #pragma omp parallel for default(none) \
00111   private(JMM, Quad, h, detC) \
00112   shared(totalNP, c, f, g, lf, lff, lfg, lg, lgg, udata, dudata, dxData) \
00113   schedule(static)
00114   for (sunindextype pp = 0; pp < totalNP * 6;
00115        pp += 6) { // loop over all 6dim points in the patch
00116     // em field Lorentz invariants F and G
00117     f = 0.5 * ((Quad[0] = udata[pp] * udata[pp]) +
00118                (Quad[1] = udata[pp + 1] * udata[pp + 1]) +
00119                (Quad[2] = udata[pp + 2] * udata[pp + 2]) -
00120                (Quad[3] = udata[pp + 3] * udata[pp + 3]) -
00121                (Quad[4] = udata[pp + 4] * udata[pp + 4]) -
00122                (Quad[5] = udata[pp + 5] * udata[pp + 5]));
00123     g = udata[pp] * udata[pp + 3] + udata[pp + 1] * udata[pp + 4] +
00124         udata[pp + 2] * udata[pp + 5];
00125     // process/expansion order and corresponding derivative values of L
00126     // w.r.t. F, G
00127     switch (*c) {
00128     case 0:  // linear Maxwell vacuum
00129       lf = 0;
00130       lff = 0;
00131       lfg = 0;
00132       lg = 0;
00133       lgg = 0;
00134       break;
00135     case 1:  // only 4-photon processes
00136       lf = 0.000206527095658582755255648 * f;
00137       lff = 0.000206527095658582755255648;
00138       lfg = 0;
00139       lg = 0.0003614224174025198216973841 * g;
00140       lgg = 0.0003614224174025198216973841;
00141       break;
00142     case 2:  // only 6-photon processes
00143       lf = 0.000354046449700427580438254 * f * f +
00144            0.000191775160254398272737387 * g * g;
00145       lff = 0.0007080928994008551608765075 * f;
00146       lfg = 0.0003835503205087965454747749 * g;
00147       lg = 0.0003835503205087965454747749 * f * g;
00148       lgg = 0.0003835503205087965454747749 * f;
00149       break;
```

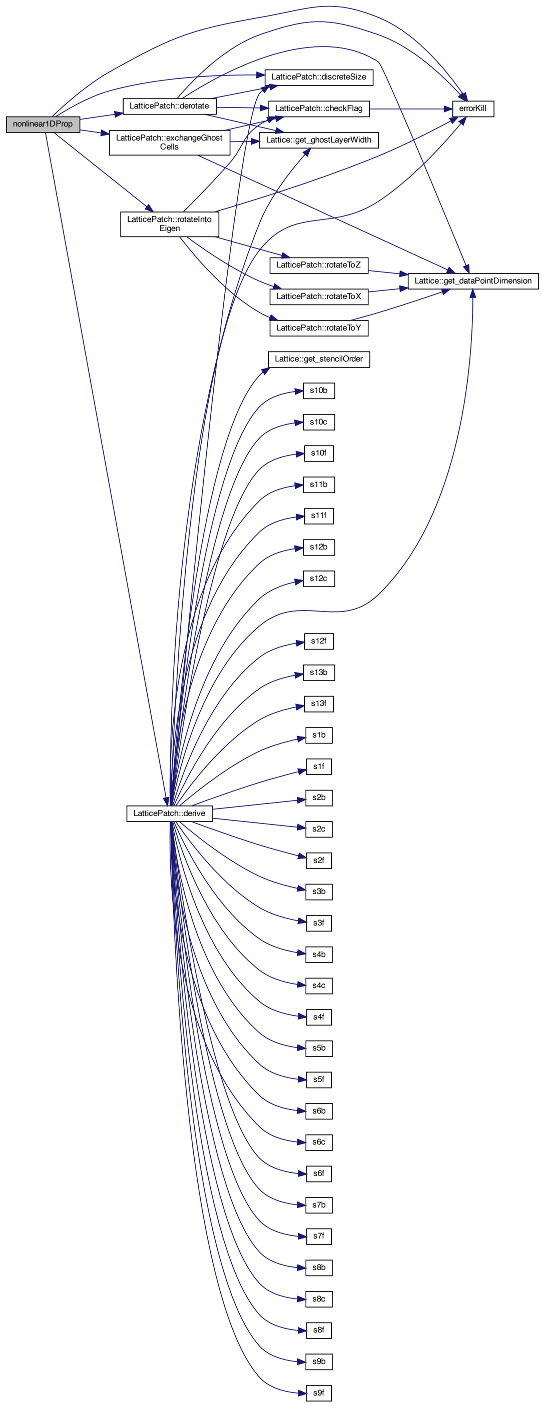
```
    case 3:  // 4- and 6-photon processes
      lf = (0.000206527095658582755255648 + 0.000354046449700427580438254 * f) *
                f +
           0.000191775160254398272737387 * g * g;
      lff = 0.000206527095658582755255648 + 0.000708092899400855160876508 * f;
      lfg = 0.000383550320508796545474749 * g;
      lg = (0.000361422417402519821697384 +
            0.000383550320508796545474749 * f) *
           g;
      lgg = 0.000361422417402519821697384 + 0.000383550320508796545474749 * f;
      break;
    default:
      errorKill(
          "You need to specify a correct order in the weak-field expansion.");
    }

    // derivatives of polarization and magnetization w.r.t. E and B
    // Jpx(Ex)
    JMM[0] = lf + lff * Quad[0] +
             udata[3 + pp] * (2 * lfg * udata[pp] + lgg * udata[3 + pp]);
    // Jpx(Ey)
    JMM[1] =
        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];
    // Jpy(Ey)
    JMM[2] = lf + lff * Quad[1] +
             udata[4 + pp] * (2 * lfg * udata[1 + pp] + lgg * udata[4 + pp]);
    // Jpx(Ez) = Jpz(Ex)
    JMM[3] =
        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];
    // Jpy(Ez) = Jpz(Ey)
    JMM[4] = 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];
    // Jpz(Ez)
    JMM[5] = lf + lff * Quad[2] +
              udata[5 + pp] * (2 * lfg * udata[2 + pp] + lgg * udata[5 + pp]);
    // Jpx(Bx) = Jmx(Ex)
    JMM[6] = lg + lfg * (Quad[0] - Quad[3 + 0]) +
              (-lff + lgg) * udata[pp] * udata[3 + pp];
    // Jpy(Bx) = Jmx(Ey)
    JMM[7] = -(udata[3 + pp] * (lff * udata[1 + pp] + lfg * udata[4 + pp])) +
              udata[pp] * (lfg * udata[1 + pp] + lgg * udata[4 + pp]);
    // Jpz(Bx) = Jmx(Ez)
    JMM[8] = -(udata[3 + pp] * (lff * udata[2 + pp] + lfg * udata[5 + pp])) +
              udata[pp] * (lfg * udata[2 + pp] + lgg * udata[5 + pp]);
    // Jmx(Bx)
    JMM[9] = -lf + lgg * Quad[0] +
              udata[3 + pp] * (-2 * lfg * udata[pp] + lff * udata[3 + pp]);
    // Jpx(By) = Jmy(Ex)
    JMM[10] = udata[1 + pp] * (lfg * udata[pp] + lgg * udata[3 + pp]) -
               (lff * udata[pp] + lfg * udata[3 + pp]) * udata[4 + pp];
    // Jpy(By) = Jmy(Ey)
    JMM[11] = lg + lfg * (Quad[1] - Quad[4 + 0]) +
               (-lff + lgg) * udata[1 + pp] * udata[4 + pp];
    // Jpz(By) = Jmy(Ez)
    JMM[12] = -(udata[4 + pp] * (lff * udata[2 + pp] + lfg * udata[5 + pp])) +
              udata[1 + pp] * (lfg * udata[2 + pp] + lgg * udata[5 + pp]);
    // Jmx(By) = Jmy(Bx)
    JMM[13] = 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]);
    // Jmy(By)
    JMM[14] = -lf + lgg * Quad[1] +
              udata[4 + pp] * (-2 * lfg * udata[1 + pp] + lff * udata[4 + pp]);
    // Jmz(Ex) = Jpx(Bz)
    JMM[15] = udata[2 + pp] * (lfg * udata[pp] + lgg * udata[3 + pp]) -
               (lff * udata[pp] + lfg * udata[3 + pp]) * udata[5 + pp];
    // Jmz(Ey) = Jpy(Bz)
    JMM[16] = udata[2 + pp] * (lfg * udata[1 + pp] + lgg * udata[4 + pp]) -
               (lff * udata[1 + pp] + lfg * udata[4 + pp]) * udata[5 + pp];
    // Jpz(Bz) = Jmz(Ez)
    JMM[17] = lg + lfg * (Quad[2] - Quad[5 + 0]) +
               (-lff + lgg) * udata[2 + pp] * udata[5 + pp];
    // Jmz(Bx) = Jmx(Bz)
    JMM[18] = 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]);
    // Jmy(Bz) = Jmz(By)
    JMM[19] =
        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]);
    // Jmz(Bz)
    JMM[20] = -lf + lgg * Quad[2] +
```

```
               udata[5 + pp] * (-2 * lfg * udata[2 + pp] + lff * udata[5 + pp]);

    // apply Z
    // top bock: -QJm(E)*E, Q-QJm(B)*B
    h[0] = 0;
    h[1] = dxData[pp] * JMM[15] + dxData[1 + pp] * JMM[16] +
           dxData[2 + pp] * JMM[17] + dxData[3 + pp] * JMM[18] +
           dxData[4 + pp] * JMM[19] + dxData[5 + pp] * (-1 + JMM[20]);
    h[2] = -(dxData[pp] * JMM[10]) - dxData[1 + pp] * JMM[11] -
           dxData[2 + pp] * JMM[12] - dxData[3 + pp] * JMM[13] +
           dxData[4 + pp] * (1 - JMM[14]) - dxData[5 + pp] * JMM[19];
    // bottom blocks: -Q*E
    h[3] = 0;
    h[4] = dxData[2 + pp];
    h[5] = -dxData[1 + pp];
    // (1+A)^-1 applies only to E components
    // -Jp(B)*B
    h[0] -= h[3] * JMM[6] + h[4] * JMM[10] + h[5] * JMM[15];
    h[1] -= h[3] * JMM[7] + h[4] * JMM[11] + h[5] * JMM[16];
    h[2] -= h[3] * JMM[8] + h[4] * JMM[12] + h[5] * JMM[17];
    // apply C^-1 explicitly, with C=1+Jp(E)
    dudata[pp + 0] =
        h[2] * (-(JMM[3] * (1 + JMM[2])) + JMM[1] * JMM[4]) +
        h[1] * (JMM[3] * JMM[4] - JMM[1] * (1 + JMM[5])) +
        h[0] * (1 - JMM[4] * JMM[4] + JMM[5] + JMM[2] * (1 + JMM[5]));
    dudata[pp + 1] =
        h[2] * (JMM[3] * JMM[1] - (1 + JMM[0]) * JMM[4]) +
        h[1] * (1 - JMM[3] * JMM[3] + JMM[5] + JMM[0] * (1 + JMM[5])) +
        h[0] * (JMM[4] * JMM[3] - JMM[1] * (1 + JMM[5]));
    dudata[pp + 2] =
        h[2] * (1 - JMM[1] * JMM[1] + JMM[2] + JMM[0] * (1 + JMM[2])) +
        h[1] * (JMM[1] * JMM[3] - (1 + JMM[0]) * JMM[4]) +
        h[0] * (-((1 + JMM[2]) * JMM[3]) + JMM[1] * JMM[4]);
    detC =  // determinant of C
        -((1 + JMM[2]) * (-1 + JMM[3] * JMM[3])) +
        (JMM[3] * JMM[1] - JMM[4]) * JMM[4] + JMM[5] + JMM[2] * JMM[5] +
        JMM[0] * (1 + JMM[2] - JMM[4] * JMM[4] + (1 + JMM[2]) * JMM[5]) -
        JMM[1] * (-(JMM[4] * JMM[3]) + JMM[1] * (1 + JMM[5]));
    dudata[pp + 0] /= detC;
    dudata[pp + 1] /= detC;
    dudata[pp + 2] /= detC;
    dudata[pp + 3] = h[3];
    dudata[pp + 4] = h[4];
    dudata[pp + 5] = h[5];
  }
  return;
}
```

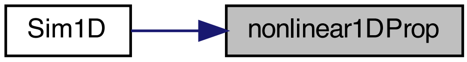
References LatticePatch::buffData, LatticePatch::derive(), LatticePatch::derotate(), LatticePatch::discreteSize(), errorKill(), LatticePatch::exchangeGhostCells(), and LatticePatch::rotateIntoEigen().

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 315 of file TimeEvolutionFunctions.cpp.

```
00315                                                                              {
00316
00317   sunrealtype *udata = NV_DATA_P(u),
00318               *dudata = NV_DATA_P(udot);
00319
00320   sunrealtype *dxData = data->buffData[1 - 1];
00321   sunrealtype *dyData = data->buffData[2 - 1];
00322
00323   data->exchangeGhostCells(1);
00324   data->rotateIntoEigen(1);
00325   data->derive(1);
00326   data->derotate(1, dxData);
00327   data->exchangeGhostCells(2);
00328   data->rotateIntoEigen(2);
00329   data->derive(2);
00330   data->derotate(2, dyData);
00331
00332   static sunrealtype f, g;
00333   static sunrealtype lf, lff, lfg, lg, lgg;
00334   static std::array<sunrealtype, 21> JMM;
00335   static std::array<sunrealtype, 6> Quad;
00336   static std::array<sunrealtype, 6> h;
00337   static sunrealtype detC;
00338
00339   const sunindextype totalNP = data->discreteSize();
00340   #pragma omp parallel for default(none) \
00341   private(JMM, Quad, h, detC) \
00342   shared(totalNP, c, f, g, lf, lff, lfg, lg, lgg, udata, dudata,\
00343           dxData, dyData) \
00344   schedule(static)
00345   for (sunindextype pp = 0; pp < totalNP * 6; pp += 6) {
00346     f = 0.5 * ((Quad[0] = udata[pp] * udata[pp]) +
00347                (Quad[1] = udata[pp + 1] * udata[pp + 1]) +
00348                (Quad[2] = udata[pp + 2] * udata[pp + 2]) -
00349                (Quad[3] = udata[pp + 3] * udata[pp + 3]) -
00350                (Quad[4] = udata[pp + 4] * udata[pp + 4]) -
00351                (Quad[5] = udata[pp + 5] * udata[pp + 5]));
00352     g = udata[pp] * udata[pp + 3] + udata[pp + 1] * udata[pp + 4] +
00353         udata[pp + 2] * udata[pp + 5];
00354     switch (*c) {
00355     case 0:
00356       lf = 0;
00357       lff = 0;
```

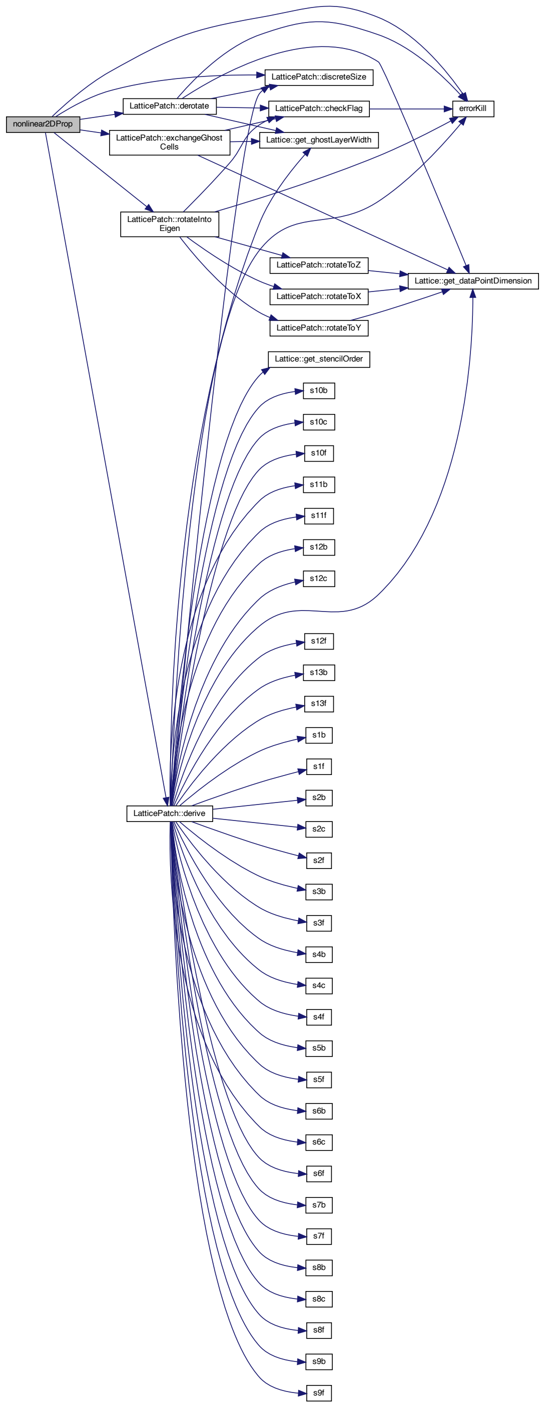
```
      lfg = 0;
      lg = 0;
      lgg = 0;
      break;
    case 1:
      lf = 0.00020652709565858275525648 * f;
      lff = 0.00020652709565858275525648;
      lfg = 0;
      lg = 0.00036142241740251982169738 41 * g;
      lgg = 0.0003614224174025198216973841;
      break;
    case 2:
      lf = 0.00035404644970042758043825 4 * f * f +
           0.00019177516025439827273738 7 * g * g;
      lff = 0.00070809289940085516087650 75 * f;
      lfg = 0.00038355032050879654547477 49 * g;
      lg = 0.00038355032050879654547477 49 * f * g;
      lgg = 0.00038355032050879654547477 49 * f;
      break;
    case 3:
      lf = (0.00020652709565858275525648 + 0.00035404644970042758043825 4 * f) *
               f +
           0.00019177516025439827273738 7 * g * g;
      lff = 0.00020652709565858275525648 + 0.00070809289940085516087650 8 * f;
      lfg = 0.00038355032050879654547477 49 * g;
      lg = (0.00036142241740251982169738 4 + 0.00038355032050879654547477 5 * f) *
           g;
      lgg = 0.00036142241740251982169738 4 + 0.00038355032050879654547477 5 * f;
      break;
    default:
      errorKill(
          "You need to specify a correct order in the weak-field expansion.");
    }

    JMM[0] = lf + lff * Quad[0] +
             udata[3 + pp] * (2 * lfg * udata[pp] + lgg * udata[3 + pp]);
    JMM[1] =
        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[2] = lf + lff * Quad[1] +
             udata[4 + pp] * (2 * lfg * udata[1 + pp] + lgg * udata[4 + pp]);
    JMM[3] =
        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[4] = 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[5] = lf + lff * Quad[2] +
              udata[5 + pp] * (2 * lfg * udata[2 + pp] + lgg * udata[5 + pp]);
    JMM[6] = lg + lfg * (Quad[0] - Quad[3 + 0]) +
              (-lff + lgg) * udata[pp] * udata[3 + pp];
    JMM[7] = -(udata[3 + pp] * (lff * udata[1 + pp] + lfg * udata[4 + pp])) +
              udata[pp] * (lfg * udata[1 + pp] + lgg * udata[4 + pp]);
    JMM[8] = -(udata[3 + pp] * (lff * udata[2 + pp] + lfg * udata[5 + pp])) +
              udata[pp] * (lfg * udata[2 + pp] + lgg * udata[5 + pp]);
    JMM[9] = -lf + lgg * Quad[0] +
              udata[3 + pp] * (-2 * lfg * udata[pp] + lff * udata[3 + pp]);
    JMM[10] = udata[1 + pp] * (lfg * udata[pp] + lgg * udata[3 + pp]) -
              (lff * udata[pp] + lfg * udata[3 + pp]) * udata[4 + pp];
    JMM[11] = lg + lfg * (Quad[1] - Quad[4 + 0]) +
              (-lff + lgg) * udata[1 + pp] * udata[4 + pp];
    JMM[12] = -(udata[4 + pp] * (lff * udata[2 + pp] + lfg * udata[5 + pp])) +
              udata[1 + pp] * (lfg * udata[2 + pp] + lgg * udata[5 + pp]);
    JMM[13] = 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[14] = -lf + lgg * Quad[1] +
              udata[4 + pp] * (-2 * lfg * udata[1 + pp] + lff * udata[4 + pp]);
    JMM[15] = udata[2 + pp] * (lfg * udata[pp] + lgg * udata[3 + pp]) -
              (lff * udata[pp] + lfg * udata[3 + pp]) * udata[5 + pp];
    JMM[16] = udata[2 + pp] * (lfg * udata[1 + pp] + lgg * udata[4 + pp]) -
              (lff * udata[1 + pp] + lfg * udata[4 + pp]) * udata[5 + pp];
    JMM[17] = lg + lfg * (Quad[2] - Quad[5 + 0]) +
              (-lff + lgg) * udata[2 + pp] * udata[5 + pp];
    JMM[18] = 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[19] =
        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[20] = -lf + lgg * Quad[2] +
              udata[5 + pp] * (-2 * lfg * udata[2 + pp] + lff * udata[5 + pp]);

    h[0] = 0;
    h[1] = dxData[pp] * JMM[15] + dxData[1 + pp] * JMM[16] +
```

```
            dxData[2 + pp] * JMM[17] + dxData[3 + pp] * JMM[18] +
            dxData[4 + pp] * JMM[19] + dxData[5 + pp] * (-1 + JMM[20]);
    h[2] = -(dxData[pp] * JMM[10]) - dxData[1 + pp] * JMM[11] -
            dxData[2 + pp] * JMM[12] - dxData[3 + pp] * JMM[13] +
            dxData[4 + pp] * (1 - JMM[14]) - dxData[5 + pp] * JMM[19];
    h[3] = 0;
    h[4] = dxData[2 + pp];
    h[5] = -dxData[1 + pp];
    h[0] += -(dyData[pp] * JMM[15]) - dyData[1 + pp] * JMM[16] -
             dyData[2 + pp] * JMM[17] - dyData[3 + pp] * JMM[18] -
             dyData[4 + pp] * JMM[19] + dyData[5 + pp] * (1 - JMM[20]);
    h[1] += 0;
    h[2] += dyData[pp] * JMM[6] + dyData[1 + pp] * JMM[7] +
             dyData[2 + pp] * JMM[8] + dyData[3 + pp] * (-1 + JMM[9]) +
             dyData[4 + pp] * JMM[13] + dyData[5 + pp] * JMM[18];
    h[3] += -dyData[2 + pp];
    h[4] += 0;
    h[5] += dyData[pp];
    h[0] -= h[3] * JMM[6] + h[4] * JMM[10] + h[5] * JMM[15];
    h[1] -= h[3] * JMM[7] + h[4] * JMM[11] + h[5] * JMM[16];
    h[2] -= h[3] * JMM[8] + h[4] * JMM[12] + h[5] * JMM[17];
    dudata[pp + 0] =
        h[2] * (-(JMM[3] * (1 + JMM[2])) + JMM[1] * JMM[4]) +
        h[1] * (JMM[3] * JMM[4] - JMM[1] * (1 + JMM[5])) +
        h[0] * (1 - JMM[4] * JMM[4] + JMM[5] + JMM[2] * (1 + JMM[5]));
    dudata[pp + 1] =
        h[2] * (JMM[3] * JMM[1] - (1 + JMM[0]) * JMM[4]) +
        h[1] * (1 - JMM[3] * JMM[3] + JMM[5] + JMM[0] * (1 + JMM[5])) +
        h[0] * (JMM[4] * JMM[3] - JMM[1] * (1 + JMM[5]));
    dudata[pp + 2] =
        h[2] * (1 - JMM[1] * JMM[1] + JMM[2] + JMM[0] * (1 + JMM[2])) +
        h[1] * (JMM[1] * JMM[3] - (1 + JMM[0]) * JMM[4]) +
        h[0] * (-((1 + JMM[2]) * JMM[3]) + JMM[1] * JMM[4]);
    detC =
        -((1 + JMM[2]) * (-1 + JMM[3] * JMM[3])) +
        (JMM[3] * JMM[1] - JMM[4]) * JMM[4] + JMM[5] + JMM[2] * JMM[5] +
        JMM[0] * (1 + JMM[2] - JMM[4] * JMM[4] + (1 + JMM[2]) * JMM[5]) -
        JMM[1] * (-(JMM[4] * JMM[3]) + JMM[1] * (1 + JMM[5]));
    dudata[pp + 0] /= detC;
    dudata[pp + 1] /= detC;
    dudata[pp + 2] /= detC;
    dudata[pp + 3] = h[3];
    dudata[pp + 4] = h[4];
    dudata[pp + 5] = h[5];
  }
  return;
}
```

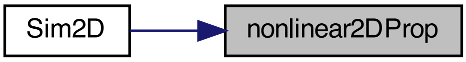
References LatticePatch::buffData, LatticePatch::derive(), LatticePatch::derotate(), LatticePatch::discreteSize(), errorKill(), LatticePatch::exchangeGhostCells(), and LatticePatch::rotateIntoEigen().

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 528 of file TimeEvolutionFunctions.cpp.

```
00528                                                                       {
00529
00530   sunrealtype *udata = NV_DATA_P(u),
00531               *dudata = NV_DATA_P(udot);
00532
00533   sunrealtype *dxData = data->buffData[1 - 1];
00534   sunrealtype *dyData = data->buffData[2 - 1];
00535   sunrealtype *dzData = data->buffData[3 - 1];
00536
00537   data->exchangeGhostCells(1);
00538   data->rotateIntoEigen(1);
00539   data->derive(1);
00540   data->derotate(1,dxData);
00541   data->exchangeGhostCells(2);
00542   data->rotateIntoEigen(2);
00543   data->derive(2);
00544   data->derotate(2,dyData);
00545   data->exchangeGhostCells(3);
00546   data->rotateIntoEigen(3);
00547   data->derive(3);
00548   data->derotate(3,dzData);
00549
00550   static sunrealtype f, g;
00551   static sunrealtype lf, lff, lfg, lg, lgg;
00552   static std::array<sunrealtype, 21> JMM;
00553   static std::array<sunrealtype, 6> Quad;
00554   static std::array<sunrealtype, 6> h;
00555   static sunrealtype detC = nan("0x12345");
00556
00557   const sunindextype totalNP = data->discreteSize();
00558   #pragma omp parallel for default(none) \
00559   private(JMM, Quad, h, detC) \
00560   shared(totalNP, c, f, g, lf, lff, lfg, lg, lgg, udata, dudata,\
00561           dxData, dyData, dzData) \
00562   schedule(static)
00563   for (sunindextype pp = 0; pp < totalNP * 6; pp += 6) {
00564     f = 0.5 * ((Quad[0] = udata[pp] * udata[pp]) +
00565                (Quad[1] = udata[pp + 1] * udata[pp + 1]) +
00566                (Quad[2] = udata[pp + 2] * udata[pp + 2]) -
00567                (Quad[3] = udata[pp + 3] * udata[pp + 3]) -
00568                (Quad[4] = udata[pp + 4] * udata[pp + 4]) -
00569                (Quad[5] = udata[pp + 5] * udata[pp + 5]));
00570     g = udata[pp] * udata[pp + 3] + udata[pp + 1] * udata[pp + 4] +
```

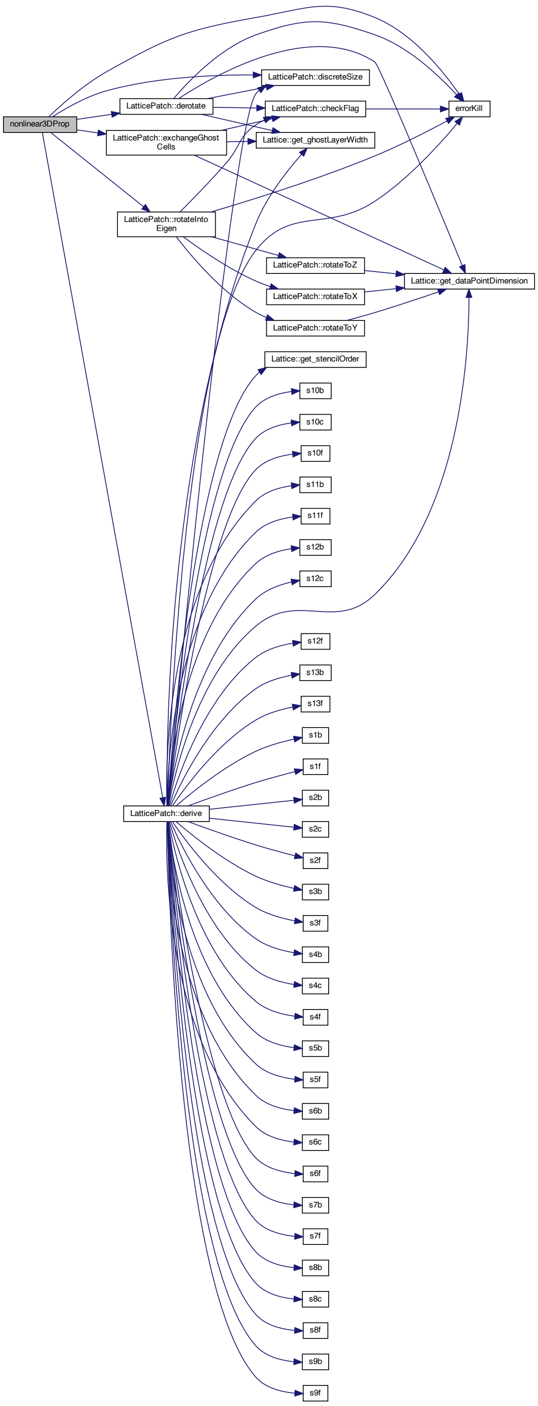
```
            udata[pp + 2] * udata[pp + 5];
      switch (*c) {
      case 0:
        lf = 0;
        lff = 0;
        lfg = 0;
        lg = 0;
        lgg = 0;
        break;
      case 1:
        lf = 0.00020652709565858275525564 8 * f;
        lff = 0.000206527095658582755255648;
        lfg = 0;
        lg = 0.000361422417402519821697384 1 * g;
        lgg = 0.0003614224174025198216973841;
        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 3:
        lf = (0.000206527095658582755255648 + 0.000354046449700427580438254 * f) *
                 f +
             0.000191775160254398272737387 * g * g;
        lff = 0.000206527095658582755255648 + 0.00070809289940085516087650 8 * 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.");
      }

      JMM[0] = lf + lff * Quad[0] +
               udata[3 + pp] * (2 * lfg * udata[pp] + lgg * udata[3 + pp]);
      JMM[1] =
          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[2] = lf + lff * Quad[1] +
               udata[4 + pp] * (2 * lfg * udata[1 + pp] + lgg * udata[4 + pp]);
      JMM[3] =
          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[4] = 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[5] = lf + lff * Quad[2] +
                udata[5 + pp] * (2 * lfg * udata[2 + pp] + lgg * udata[5 + pp]);
      JMM[6] = lg + lfg * (Quad[0] - Quad[3 + 0]) +
                (-lff + lgg) * udata[pp] * udata[3 + pp];
      JMM[7] = -(udata[3 + pp] * (lff * udata[1 + pp] + lfg * udata[4 + pp])) +
                udata[pp] * (lfg * udata[1 + pp] + lgg * udata[4 + pp]);
      JMM[8] = -(udata[3 + pp] * (lff * udata[2 + pp] + lfg * udata[5 + pp])) +
                udata[pp] * (lfg * udata[2 + pp] + lgg * udata[5 + pp]);
      JMM[9] = -lf + lgg * Quad[0] +
                udata[3 + pp] * (-2 * lfg * udata[pp] + lff * udata[3 + pp]);
      JMM[10] = udata[1 + pp] * (lfg * udata[pp] + lgg * udata[3 + pp]) -
                (lff * udata[pp] + lfg * udata[3 + pp]) * udata[4 + pp];
      JMM[11] = lg + lfg * (Quad[1] - Quad[4 + 0]) +
                (-lff + lgg) * udata[1 + pp] * udata[4 + pp];
      JMM[12] = -(udata[4 + pp] * (lff * udata[2 + pp] + lfg * udata[5 + pp])) +
                udata[1 + pp] * (lfg * udata[2 + pp] + lgg * udata[5 + pp]);
      JMM[13] = 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[14] = -lf + lgg * Quad[1] +
                udata[4 + pp] * (-2 * lfg * udata[1 + pp] + lff * udata[4 + pp]);
      JMM[15] = udata[2 + pp] * (lfg * udata[pp] + lgg * udata[3 + pp]) -
                (lff * udata[pp] + lfg * udata[3 + pp]) * udata[5 + pp];
      JMM[16] = udata[2 + pp] * (lfg * udata[1 + pp] + lgg * udata[4 + pp]) -
                (lff * udata[1 + pp] + lfg * udata[4 + pp]) * udata[5 + pp];
      JMM[17] = lg + lfg * (Quad[2] - Quad[5 + 0]) +
                (-lff + lgg) * udata[2 + pp] * udata[5 + pp];
      JMM[18] = 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[19] =
          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[20] = -lf + lgg * Quad[2] +
              udata[5 + pp] * (-2 * lfg * udata[2 + pp] + lff * udata[5 + pp]);

    h[0] = 0;
    h[1] = dxData[pp] * JMM[15] + dxData[1 + pp] * JMM[16] +
           dxData[2 + pp] * JMM[17] + dxData[3 + pp] * JMM[18] +
           dxData[4 + pp] * JMM[19] + dxData[5 + pp] * (-1 + JMM[20]);
    h[2] = -(dxData[pp] * JMM[10]) - dxData[1 + pp] * JMM[11] -
           dxData[2 + pp] * JMM[12] - dxData[3 + pp] * JMM[13] +
           dxData[4 + pp] * (1 - JMM[14]) - dxData[5 + pp] * JMM[19];
    h[3] = 0;
    h[4] = dxData[2 + pp];
    h[5] = -dxData[1 + pp];
    h[0] += -(dyData[pp] * JMM[15]) - dyData[1 + pp] * JMM[16] -
            dyData[2 + pp] * JMM[17] - dyData[3 + pp] * JMM[18] -
            dyData[4 + pp] * JMM[19] + dyData[5 + pp] * (1 - JMM[20]);
    h[1] += 0;
    h[2] += dyData[pp] * JMM[6] + dyData[1 + pp] * JMM[7] +
            dyData[2 + pp] * JMM[8] + dyData[3 + pp] * (-1 + JMM[9]) +
            dyData[4 + pp] * JMM[13] + dyData[5 + pp] * JMM[18];
    h[3] += -dyData[2 + pp];
    h[4] += 0;
    h[5] += dyData[pp];
    h[0] += dzData[pp] * JMM[10] + dzData[1 + pp] * JMM[11] +
            dzData[2 + pp] * JMM[12] + dzData[3 + pp] * JMM[13] +
            dzData[4 + pp] * (-1 + JMM[14]) + dzData[5 + pp] * JMM[19];
    h[1] += -(dzData[pp] * JMM[6]) - dzData[1 + pp] * JMM[7] -
            dzData[2 + pp] * JMM[8] + dzData[3 + pp] * (1 - JMM[9]) -
            dzData[4 + pp] * JMM[13] - dzData[5 + pp] * JMM[18];
    h[2] += 0;
    h[3] += dzData[1 + pp];
    h[4] += -dzData[pp];
    h[5] += 0;
    h[0] -= h[3] * JMM[6] + h[4] * JMM[10] + h[5] * JMM[15];
    h[1] -= h[3] * JMM[7] + h[4] * JMM[11] + h[5] * JMM[16];
    h[2] -= h[3] * JMM[8] + h[4] * JMM[12] + h[5] * JMM[17];
    dudata[pp + 0] =
        h[2] * (-(JMM[3] * (1 + JMM[2])) + JMM[1] * JMM[4]) +
        h[1] * (JMM[3] * JMM[4] - JMM[1] * (1 + JMM[5])) +
        h[0] * (1 - JMM[4] * JMM[4] + JMM[5] + JMM[2] * (1 + JMM[5]));
    dudata[pp + 1] =
        h[2] * (JMM[3] * JMM[1] - (1 + JMM[0]) * JMM[4]) +
        h[1] * (1 - JMM[3] * JMM[3] + JMM[5] + JMM[0] * (1 + JMM[5])) +
        h[0] * (JMM[4] * JMM[3] - JMM[1] * (1 + JMM[5]));
    dudata[pp + 2] =
        h[2] * (1 - JMM[1] * JMM[1] + JMM[2] + JMM[0] * (1 + JMM[2])) +
        h[1] * (JMM[1] * JMM[3] - (1 + JMM[0]) * JMM[4]) +
        h[0] * (-((1 + JMM[2]) * JMM[3]) + JMM[1] * JMM[4]);
    detC =
        -((1 + JMM[2]) * (-1 + JMM[3] * JMM[3])) +
        (JMM[3] * JMM[1] - JMM[4]) * JMM[4] + JMM[5] + JMM[2] * JMM[5] +
        JMM[0] * (1 + JMM[2] - JMM[4] * JMM[4] + (1 + JMM[2]) * JMM[5]) -
        JMM[1] * (-(JMM[4] * JMM[3]) + JMM[1] * (1 + JMM[5]));
    dudata[pp + 0] /= detC;
    dudata[pp + 1] /= detC;
    dudata[pp + 2] /= detC;
    dudata[pp + 3] = h[3];
    dudata[pp + 4] = h[4];
    dudata[pp + 5] = h[5];
  }
  return;
}
```

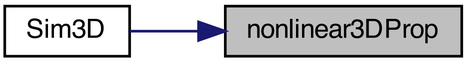
References LatticePatch::buffData, LatticePatch::derive(), LatticePatch::derotate(), LatticePatch::discreteSize(), errorKill(), LatticePatch::exchangeGhostCells(), and LatticePatch::rotateIntoEigen().

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"

/// 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 field data resides
  LatticePatch *data = static_cast<LatticePatch *>(data_loc);

  // update circle
  // Access provided field values and temp. derivatieves with NVector pointers
  sunrealtype *udata = NV_DATA_P(u),
              *dudata = NV_DATA_P(udot);

  // Store original data location of the patch
  sunrealtype *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 - field data is here
// accessed implicitly via user data (lattice patch);
// same effect 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 according to the scheme:
  data->exchangeGhostCells(1); // -> exchange halos
  data->rotateIntoEigen(
      1);         // -> rotate all data to prepare derivative operation
  data->derive(1); // -> perform derivative approximation operation on it
  data->derotate(
      1, dxData); // -> derotate derived data for ensuing time-evolution

  const sunindextype totalNP = data->discreteSize();
  sunindextype pp = 0;
  for (sunindextype i = 0; i < totalNP; i++) {
    pp = i * 6;
    /*
```

```
        simple vacuum Maxwell equations for the temporal derivatives using the
        spatial deriative only in x-direction without 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) {

  // NVector pointers to provided field values and their temp. derivatives
  sunrealtype *udata = NV_DATA_P(u),
              *dudata = NV_DATA_P(udot);

  // pointer to spatial derivatives via pach data
  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 derivatives
  w.r.t. E- and B-field go into the e.o.m.
  */
  static sunrealtype f, g; // em field invariants F, G
  // derivatives of HE Lagrangian w.r.t. field invariants
  static sunrealtype lf, lff, lfg, lg, lgg;
  // matrix to hold derivatives of polarization and magnetization
  static std::array<sunrealtype, 21> JMM;
  // array to hold E^2 and B^2 components
  static std::array<sunrealtype, 6> Quad;
  // array to hold intermediate temp. derivatives of E and B
  static std::array<sunrealtype, 6> h;
  // determinant needed for explicit matrix inversion
  static sunrealtype detC = nan("0x12345");

  // number of points in the patch
  const sunindextype totalNP = data->discreteSize();
  #pragma omp parallel for default(none) \
  private(JMM, Quad, h, detC) \
  shared(totalNP, c, f, g, lf, lff, lfg, lg, lgg, udata, dudata, dxData) \
  schedule(static)
  for (sunindextype pp = 0; pp < totalNP * 6;
       pp += 6) { // loop over all 6dim points in the patch
    // em field Lorentz invariants 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];
    // process/expansion order and corresponding derivative values of L
    // w.r.t. F, G
    switch (*c) {
    case 0:  // linear Maxwell vacuum
      lf = 0;
      lff = 0;
      lfg = 0;
      lg = 0;
      lgg = 0;
      break;
    case 1:  // only 4-photon processes
      lf = 0.000206527095658582755255648 * f;
      lff = 0.000206527095658582755255648;
      lfg = 0;
      lg = 0.000361422417402519821697384 1 * g;
      lgg = 0.0003614224174025198216973841;
      break;
    case 2:  // only 6-photon processes
      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 3:  // 4- and 6-photon processes
      lf = (0.00020652709565858275525564 8 + 0.00035404644970042758043825 4 * f) *
             f +
           0.000191775160254398272737387 * 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.");
    }

    // derivatives of polarization and magnetization w.r.t. E and B
    // Jpx(Ex)
    JMM[0] = lf + lff * Quad[0] +
             udata[3 + pp] * (2 * lfg * udata[pp] + lgg * udata[3 + pp]);
    // Jpx(Ey)
    JMM[1] =
        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];
    // Jpy(Ey)
    JMM[2] = lf + lff * Quad[1] +
             udata[4 + pp] * (2 * lfg * udata[1 + pp] + lgg * udata[4 + pp]);
    // Jpx(Ez) = Jpz(Ex)
    JMM[3] =
        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];
    // Jpy(Ez) = Jpz(Ey)
    JMM[4] = 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];
    // Jpz(Ez)
    JMM[5] = lf + lff * Quad[2] +
              udata[5 + pp] * (2 * lfg * udata[2 + pp] + lgg * udata[5 + pp]);
    // Jpx(Bx) = Jmx(Ex)
    JMM[6] = lg + lfg * (Quad[0] - Quad[3 + 0]) +
              (-lff + lgg) * udata[pp] * udata[3 + pp];
    // Jpy(Bx) = Jmx(Ey)
    JMM[7] = -(udata[3 + pp] * (lff * udata[1 + pp] + lfg * udata[4 + pp])) +
              udata[pp] * (lfg * udata[1 + pp] + lgg * udata[4 + pp]);
    // Jpz(Bx) = Jmx(Ez)
    JMM[8] = -(udata[3 + pp] * (lff * udata[2 + pp] + lfg * udata[5 + pp])) +
              udata[pp] * (lfg * udata[2 + pp] + lgg * udata[5 + pp]);
    // Jmx(Bx)
    JMM[9] = -lf + lgg * Quad[0] +
              udata[3 + pp] * (-2 * lfg * udata[pp] + lff * udata[3 + pp]);
    // Jpx(By) = Jmy(Ex)
    JMM[10] = udata[1 + pp] * (lfg * udata[pp] + lgg * udata[3 + pp]) -
               (lff * udata[pp] + lfg * udata[3 + pp]) * udata[4 + pp];
    // Jpy(By) = Jmy(Ey)
    JMM[11] = lg + lfg * (Quad[1] - Quad[4 + 0]) +
               (-lff + lgg) * udata[1 + pp] * udata[4 + pp];
    // Jpz(By) = Jmy(Ez)
    JMM[12] = -(udata[4 + pp] * (lff * udata[2 + pp] + lfg * udata[5 + pp])) +
              udata[1 + pp] * (lfg * udata[2 + pp] + lgg * udata[5 + pp]);
    // Jmx(By) = Jmy(Bx)
    JMM[13] = 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]);
    // Jmy(By)
    JMM[14] = -lf + lgg * Quad[1] +
              udata[4 + pp] * (-2 * lfg * udata[1 + pp] + lff * udata[4 + pp]);
    // Jmz(Ex) = Jpx(Bz)
    JMM[15] = udata[2 + pp] * (lfg * udata[pp] + lgg * udata[3 + pp]) -
               (lff * udata[pp] + lfg * udata[3 + pp]) * udata[5 + pp];
    // Jmz(Ey) = Jpy(Bz)
    JMM[16] = udata[2 + pp] * (lfg * udata[1 + pp] + lgg * udata[4 + pp]) -
               (lff * udata[1 + pp] + lfg * udata[4 + pp]) * udata[5 + pp];
    // Jpz(Bz) = Jmz(Ez)
    JMM[17] = lg + lfg * (Quad[2] - Quad[5 + 0]) +
               (-lff + lgg) * udata[2 + pp] * udata[5 + pp];
    // Jmz(Bx) = Jmx(Bz)
    JMM[18] = 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]);
    // Jmy(Bz) = Jmz(By)
    JMM[19] =
        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]);
    // Jmz(Bz)
```

```
    JMM[20] = -lf + lgg * Quad[2] +
              udata[5 + pp] * (-2 * lfg * udata[2 + pp] + lff * udata[5 + pp]);

    // apply Z
    // top bock: -QJm(E)*E, Q-QJm(B)*B
    h[0] = 0;
    h[1] = dxData[pp] * JMM[15] + dxData[1 + pp] * JMM[16] +
           dxData[2 + pp] * JMM[17] + dxData[3 + pp] * JMM[18] +
           dxData[4 + pp] * JMM[19] + dxData[5 + pp] * (-1 + JMM[20]);
    h[2] = -(dxData[pp] * JMM[10]) - dxData[1 + pp] * JMM[11] -
           dxData[2 + pp] * JMM[12] - dxData[3 + pp] * JMM[13] +
           dxData[4 + pp] * (1 - JMM[14]) - dxData[5 + pp] * JMM[19];
    // bottom blocks: -Q*E
    h[3] = 0;
    h[4] = dxData[2 + pp];
    h[5] = -dxData[1 + pp];
    // (1+A)^-1 applies only to E components
    // -Jp(B)*B
    h[0] -= h[3] * JMM[6] + h[4] * JMM[10] + h[5] * JMM[15];
    h[1] -= h[3] * JMM[7] + h[4] * JMM[11] + h[5] * JMM[16];
    h[2] -= h[3] * JMM[8] + h[4] * JMM[12] + h[5] * JMM[17];
    // apply C^-1 explicitly, with C=1+Jp(E)
    dudata[pp + 0] =
        h[2] * (-(JMM[3] * (1 + JMM[2])) + JMM[1] * JMM[4]) +
        h[1] * (JMM[3] * JMM[4] - JMM[1] * (1 + JMM[5])) +
        h[0] * (1 - JMM[4] * JMM[4] + JMM[5] + JMM[2] * (1 + JMM[5]));
    dudata[pp + 1] =
        h[2] * (JMM[3] * JMM[1] - (1 + JMM[0]) * JMM[4]) +
        h[1] * (1 - JMM[3] * JMM[3] + JMM[5] + JMM[0] * (1 + JMM[5])) +
        h[0] * (JMM[4] * JMM[3] - JMM[1] * (1 + JMM[5]));
    dudata[pp + 2] =
        h[2] * (1 - JMM[1] * JMM[1] + JMM[2] + JMM[0] * (1 + JMM[2])) +
        h[1] * (JMM[1] * JMM[3] - (1 + JMM[0]) * JMM[4]) +
        h[0] * (-((1 + JMM[2]) * JMM[3]) + JMM[1] * JMM[4]);
    detC =  // determinant of C
        -((1 + JMM[2]) * (-1 + JMM[3] * JMM[3])) +
        (JMM[3] * JMM[1] - JMM[4]) * JMM[4] + JMM[5] + JMM[2] * JMM[5] +
        JMM[0] * (1 + JMM[2] - JMM[4] * JMM[4] + (1 + JMM[2]) * JMM[5]) -
        JMM[1] * (-(JMM[4] * JMM[3]) + JMM[1] * (1 + JMM[5]));
    dudata[pp + 0] /= detC;
    dudata[pp + 1] /= detC;
    dudata[pp + 2] /= detC;
    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
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);

  const sunindextype totalNP = data->discreteSize();
  sunindextype pp = 0;
  for (sunindextype 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 *udata = NV_DATA_P(u),
              *dudata = NV_DATA_P(udot);

  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);

static sunrealtype f, g;
static sunrealtype lf, lff, lfg, lg, lgg;
static std::array<sunrealtype, 21> JMM;
static std::array<sunrealtype, 6> Quad;
static std::array<sunrealtype, 6> h;
static sunrealtype detC;

const sunindextype totalNP = data->discreteSize();
#pragma omp parallel for default(none) \
private(JMM, Quad, h, detC) \
shared(totalNP, c, f, g, lf, lff, lfg, lg, lgg, udata, dudata,\
        dxData, dyData) \
schedule(static)
for (sunindextype pp = 0; pp < totalNP * 6; pp += 6) {
  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];
  switch (*c) {
  case 0:
    lf = 0;
    lff = 0;
    lfg = 0;
    lg = 0;
    lgg = 0;
    break;
  case 1:
    lf = 0.000206527095658582755255648 * f;
    lff = 0.000206527095658582755255648;
    lfg = 0;
    lg = 0.000361422417402519821697384 1 * g;
    lgg = 0.0003614224174025198216973841;
    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 3:
    lf = (0.000206527095658582755255648 + 0.000354046449700427580438254 * f) *
             f +
         0.000191775160254398272737387 * g * g;
    lff = 0.000206527095658582755255648 + 0.000708092899400855160876508 * f;
    lfg = 0.000383550320508796545474 7749 * 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.");
  }

  JMM[0] = lf + lff * Quad[0] +
           udata[3 + pp] * (2 * lfg * udata[pp] + lgg * udata[3 + pp]);
  JMM[1] =
      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[2] = lf + lff * Quad[1] +
           udata[4 + pp] * (2 * lfg * udata[1 + pp] + lgg * udata[4 + pp]);
  JMM[3] =
      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[4] = 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[5] = lf + lff * Quad[2] +
            udata[5 + pp] * (2 * lfg * udata[2 + pp] + lgg * udata[5 + pp]);
  JMM[6] = lg + lfg * (Quad[0] - Quad[3 + 0]) +
            (-lff + lgg) * udata[pp] * udata[3 + pp];
```

```
    JMM[7] = -(udata[3 + pp] * (lff * udata[1 + pp] + lfg * udata[4 + pp])) +
             udata[pp] * (lfg * udata[1 + pp] + lgg * udata[4 + pp]);
    JMM[8] = -(udata[3 + pp] * (lff * udata[2 + pp] + lfg * udata[5 + pp])) +
             udata[pp] * (lfg * udata[2 + pp] + lgg * udata[5 + pp]);
    JMM[9] = -lf + lgg * Quad[0] +
             udata[3 + pp] * (-2 * lfg * udata[pp] + lff * udata[3 + pp]);
    JMM[10] = udata[1 + pp] * (lfg * udata[pp] + lgg * udata[3 + pp]) -
              (lff * udata[pp] + lfg * udata[3 + pp]) * udata[4 + pp];
    JMM[11] = lg + lfg * (Quad[1] - Quad[4 + 0]) +
              (-lff + lgg) * udata[1 + pp] * udata[4 + pp];
    JMM[12] = -(udata[4 + pp] * (lff * udata[2 + pp] + lfg * udata[5 + pp])) +
             udata[1 + pp] * (lfg * udata[2 + pp] + lgg * udata[5 + pp]);
    JMM[13] = 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[14] = -lf + lgg * Quad[1] +
             udata[4 + pp] * (-2 * lfg * udata[1 + pp] + lff * udata[4 + pp]);
    JMM[15] = udata[2 + pp] * (lfg * udata[pp] + lgg * udata[3 + pp]) -
              (lff * udata[pp] + lfg * udata[3 + pp]) * udata[5 + pp];
    JMM[16] = udata[2 + pp] * (lfg * udata[1 + pp] + lgg * udata[4 + pp]) -
              (lff * udata[1 + pp] + lfg * udata[4 + pp]) * udata[5 + pp];
    JMM[17] = lg + lfg * (Quad[2] - Quad[5 + 0]) +
              (-lff + lgg) * udata[2 + pp] * udata[5 + pp];
    JMM[18] = 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[19] =
        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[20] = -lf + lgg * Quad[2] +
             udata[5 + pp] * (-2 * lfg * udata[2 + pp] + lff * udata[5 + pp]);

    h[0] = 0;
    h[1] = dxData[pp] * JMM[15] + dxData[1 + pp] * JMM[16] +
           dxData[2 + pp] * JMM[17] + dxData[3 + pp] * JMM[18] +
           dxData[4 + pp] * JMM[19] + dxData[5 + pp] * (-1 + JMM[20]);
    h[2] = -(dxData[pp] * JMM[10]) - dxData[1 + pp] * JMM[11] -
           dxData[2 + pp] * JMM[12] - dxData[3 + pp] * JMM[13] +
           dxData[4 + pp] * (1 - JMM[14]) - dxData[5 + pp] * JMM[19];
    h[3] = 0;
    h[4] = dxData[2 + pp];
    h[5] = -dxData[1 + pp];
    h[0] += -(dyData[pp] * JMM[15]) - dyData[1 + pp] * JMM[16] -
            dyData[2 + pp] * JMM[17] - dyData[3 + pp] * JMM[18] -
            dyData[4 + pp] * JMM[19] + dyData[5 + pp] * (1 - JMM[20]);
    h[1] += 0;
    h[2] += dyData[pp] * JMM[6] + dyData[1 + pp] * JMM[7] +
            dyData[2 + pp] * JMM[8] + dyData[3 + pp] * (-1 + JMM[9]) +
            dyData[4 + pp] * JMM[13] + dyData[5 + pp] * JMM[18];
    h[3] += -dyData[2 + pp];
    h[4] += 0;
    h[5] += dyData[pp];
    h[0] -= h[3] * JMM[6] + h[4] * JMM[10] + h[5] * JMM[15];
    h[1] -= h[3] * JMM[7] + h[4] * JMM[11] + h[5] * JMM[16];
    h[2] -= h[3] * JMM[8] + h[4] * JMM[12] + h[5] * JMM[17];
    dudata[pp + 0] =
        h[2] * (-(JMM[3] * (1 + JMM[2])) + JMM[1] * JMM[4]) +
        h[1] * (JMM[3] * JMM[4] - JMM[1] * (1 + JMM[5])) +
        h[0] * (1 - JMM[4] * JMM[4] + JMM[5] + JMM[2] * (1 + JMM[5]));
    dudata[pp + 1] =
        h[2] * (JMM[3] * JMM[1] - (1 + JMM[0]) * JMM[4]) +
        h[1] * (1 - JMM[3] * JMM[3] + JMM[5] + JMM[0] * (1 + JMM[5])) +
        h[0] * (JMM[4] * JMM[3] - JMM[1] * (1 + JMM[5]));
    dudata[pp + 2] =
        h[2] * (1 - JMM[1] * JMM[1] + JMM[2] + JMM[0] * (1 + JMM[2])) +
        h[1] * (JMM[1] * JMM[3] - (1 + JMM[0]) * JMM[4]) +
        h[0] * (-((1 + JMM[2]) * JMM[3]) + JMM[1] * JMM[4]);
    detC =
        -((1 + JMM[2]) * (-1 + JMM[3] * JMM[3])) +
        (JMM[3] * JMM[1] - JMM[4]) * JMM[4] + JMM[5] + JMM[2] * JMM[5] +
        JMM[0] * (1 + JMM[2] - JMM[4] * JMM[4] + (1 + JMM[2]) * JMM[5]) -
        JMM[1] * (-(JMM[4] * JMM[3]) + JMM[1] * (1 + JMM[5]));
    dudata[pp + 0] /= detC;
    dudata[pp + 1] /= detC;
    dudata[pp + 2] /= detC;
    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
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];

  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);

  const sunindextype totalNP = data->discreteSize();
  sunindextype pp = 0;
  for (sunindextype 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 *udata = NV_DATA_P(u),
              *dudata = NV_DATA_P(udot);

  sunrealtype *dxData = data->buffData[1 - 1];
  sunrealtype *dyData = data->buffData[2 - 1];
  sunrealtype *dzData = data->buffData[3 - 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);
  data->exchangeGhostCells(3);
  data->rotateIntoEigen(3);
  data->derive(3);
  data->derotate(3,dzData);

  static sunrealtype f, g;
  static sunrealtype lf, lff, lfg, lg, lgg;
  static std::array<sunrealtype, 21> JMM;
  static std::array<sunrealtype, 6> Quad;
  static std::array<sunrealtype, 6> h;
  static sunrealtype detC = nan("0x12345");

  const sunindextype totalNP = data->discreteSize();
  #pragma omp parallel for default(none) \
  private(JMM, Quad, h, detC) \
  shared(totalNP, c, f, g, lf, lff, lfg, lg, lgg, udata, dudata,\
          dxData, dyData, dzData) \
  schedule(static)
  for (sunindextype pp = 0; pp < totalNP * 6; pp += 6) {
    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];
    switch (*c) {
    case 0:
      lf = 0;
      lff = 0;
      lfg = 0;
      lg = 0;
      lgg = 0;
      break;
    case 1:
      lf = 0.000206527095658582755255648 * f;
      lff = 0.000206527095658582755255648;
      lfg = 0;
```

```
      lg = 0.000361422417402519821697384 1 * g;
      lgg = 0.000361422417402519821697384 1;
      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 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.");
    }

    JMM[0] = lf + lff * Quad[0] +
             udata[3 + pp] * (2 * lfg * udata[pp] + lgg * udata[3 + pp]);
    JMM[1] =
        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[2] = lf + lff * Quad[1] +
             udata[4 + pp] * (2 * lfg * udata[1 + pp] + lgg * udata[4 + pp]);
    JMM[3] =
        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[4] = 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[5] = lf + lff * Quad[2] +
              udata[5 + pp] * (2 * lfg * udata[2 + pp] + lgg * udata[5 + pp]);
    JMM[6] = lg + lfg * (Quad[0] - Quad[3 + 0]) +
              (-lff + lgg) * udata[pp] * udata[3 + pp];
    JMM[7] = -(udata[3 + pp] * (lff * udata[1 + pp] + lfg * udata[4 + pp])) +
              udata[pp] * (lfg * udata[1 + pp] + lgg * udata[4 + pp]);
    JMM[8] = -(udata[3 + pp] * (lff * udata[2 + pp] + lfg * udata[5 + pp])) +
              udata[pp] * (lfg * udata[2 + pp] + lgg * udata[5 + pp]);
    JMM[9] = -lf + lgg * Quad[0] +
              udata[3 + pp] * (-2 * lfg * udata[pp] + lff * udata[3 + pp]);
    JMM[10] = udata[1 + pp] * (lfg * udata[pp] + lgg * udata[3 + pp]) -
              (lff * udata[pp] + lfg * udata[3 + pp]) * udata[4 + pp];
    JMM[11] = lg + lfg * (Quad[1] - Quad[4 + 0]) +
              (-lff + lgg) * udata[1 + pp] * udata[4 + pp];
    JMM[12] = -(udata[4 + pp] * (lff * udata[2 + pp] + lfg * udata[5 + pp])) +
              udata[1 + pp] * (lfg * udata[2 + pp] + lgg * udata[5 + pp]);
    JMM[13] = 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[14] = -lf + lgg * Quad[1] +
              udata[4 + pp] * (-2 * lfg * udata[1 + pp] + lff * udata[4 + pp]);
    JMM[15] = udata[2 + pp] * (lfg * udata[pp] + lgg * udata[3 + pp]) -
              (lff * udata[pp] + lfg * udata[3 + pp]) * udata[5 + pp];
    JMM[16] = udata[2 + pp] * (lfg * udata[1 + pp] + lgg * udata[4 + pp]) -
              (lff * udata[1 + pp] + lfg * udata[4 + pp]) * udata[5 + pp];
    JMM[17] = lg + lfg * (Quad[2] - Quad[5 + 0]) +
              (-lff + lgg) * udata[2 + pp] * udata[5 + pp];
    JMM[18] = 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[19] =
        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[20] = -lf + lgg * Quad[2] +
              udata[5 + pp] * (-2 * lfg * udata[2 + pp] + lff * udata[5 + pp]);

    h[0] = 0;
    h[1] = dxData[pp] * JMM[15] + dxData[1 + pp] * JMM[16] +
           dxData[2 + pp] * JMM[17] + dxData[3 + pp] * JMM[18] +
           dxData[4 + pp] * JMM[19] + dxData[5 + pp] * (-1 + JMM[20]);
    h[2] = -(dxData[pp] * JMM[10]) - dxData[1 + pp] * JMM[11] -
           dxData[2 + pp] * JMM[12] - dxData[3 + pp] * JMM[13] +
           dxData[4 + pp] * (1 - JMM[14]) - dxData[5 + pp] * JMM[19];
    h[3] = 0;
    h[4] = dxData[2 + pp];
    h[5] = -dxData[1 + pp];
```

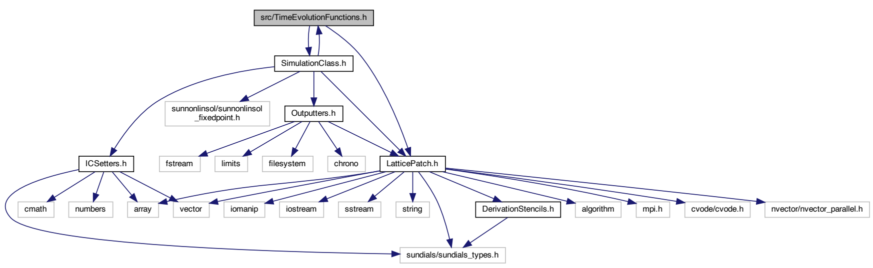
```
    h[0] += -(dyData[pp] * JMM[15]) - dyData[1 + pp] * JMM[16] -
            dyData[2 + pp] * JMM[17] - dyData[3 + pp] * JMM[18] -
            dyData[4 + pp] * JMM[19] + dyData[5 + pp] * (1 - JMM[20]);
    h[1] += 0;
    h[2] += dyData[pp] * JMM[6] + dyData[1 + pp] * JMM[7] +
            dyData[2 + pp] * JMM[8] + dyData[3 + pp] * (-1 + JMM[9]) +
            dyData[4 + pp] * JMM[13] + dyData[5 + pp] * JMM[18];
    h[3] += -dyData[2 + pp];
    h[4] += 0;
    h[5] += dyData[pp];
    h[0] += dzData[pp] * JMM[10] + dzData[1 + pp] * JMM[11] +
            dzData[2 + pp] * JMM[12] + dzData[3 + pp] * JMM[13] +
            dzData[4 + pp] * (-1 + JMM[14]) + dzData[5 + pp] * JMM[19];
    h[1] += -(dzData[pp] * JMM[6]) - dzData[1 + pp] * JMM[7] -
            dzData[2 + pp] * JMM[8] + dzData[3 + pp] * (1 - JMM[9]) -
            dzData[4 + pp] * JMM[13] - dzData[5 + pp] * JMM[18];
    h[2] += 0;
    h[3] += dzData[1 + pp];
    h[4] += -dzData[pp];
    h[5] += 0;
    h[0] -= h[3] * JMM[6] + h[4] * JMM[10] + h[5] * JMM[15];
    h[1] -= h[3] * JMM[7] + h[4] * JMM[11] + h[5] * JMM[16];
    h[2] -= h[3] * JMM[8] + h[4] * JMM[12] + h[5] * JMM[17];
    dudata[pp + 0] =
        h[2] * (-(JMM[3] * (1 + JMM[2])) + JMM[1] * JMM[4]) +
        h[1] * (JMM[3] * JMM[4] - JMM[1] * (1 + JMM[5])) +
        h[0] * (1 - JMM[4] * JMM[4] + JMM[5] + JMM[2] * (1 + JMM[5]));
    dudata[pp + 1] =
        h[2] * (JMM[3] * JMM[1] - (1 + JMM[0]) * JMM[4]) +
        h[1] * (1 - JMM[3] * JMM[3] + JMM[5] + JMM[0] * (1 + JMM[5])) +
        h[0] * (JMM[4] * JMM[3] - JMM[1] * (1 + JMM[5]));
    dudata[pp + 2] =
        h[2] * (1 - JMM[1] * JMM[1] + JMM[2] + JMM[0] * (1 + JMM[2])) +
        h[1] * (JMM[1] * JMM[3] - (1 + JMM[0]) * JMM[4]) +
        h[0] * (-((1 + JMM[2]) * JMM[3]) + JMM[1] * JMM[4]);
    detC =
        -((1 + JMM[2]) * (-1 + JMM[3] * JMM[3])) +
        (JMM[3] * JMM[1] - JMM[4]) * JMM[4] + JMM[5] + JMM[2] * JMM[5] +
        JMM[0] * (1 + JMM[2] - JMM[4] * JMM[4] + (1 + JMM[2]) * JMM[5]) -
        JMM[1] * (-(JMM[4] * JMM[3]) + JMM[1] * (1 + JMM[5]));
    dudata[pp + 0] /= detC;
    dudata[pp + 1] /= detC;
    dudata[pp + 2] /= detC;
    dudata[pp + 3] = h[3];
    dudata[pp + 4] = h[4];
    dudata[pp + 5] = h[5];
  }
  return;
}
```

6.30 src/TimeEvolutionFunctions.h File Reference

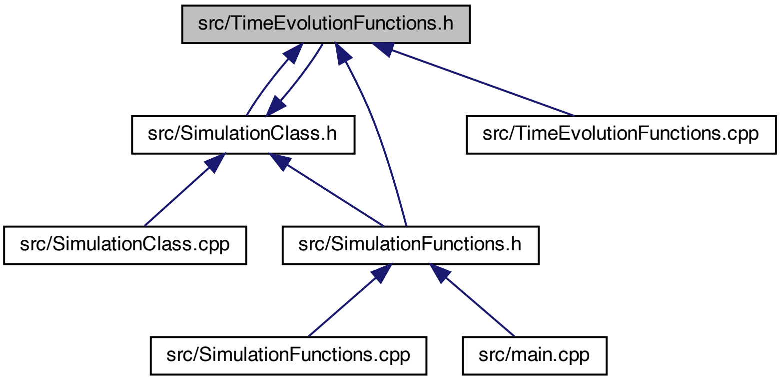
Functions to propagate data vectors in time according to Maxwell's equations, and various orders in the HE weak-field expansion.

```
#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.

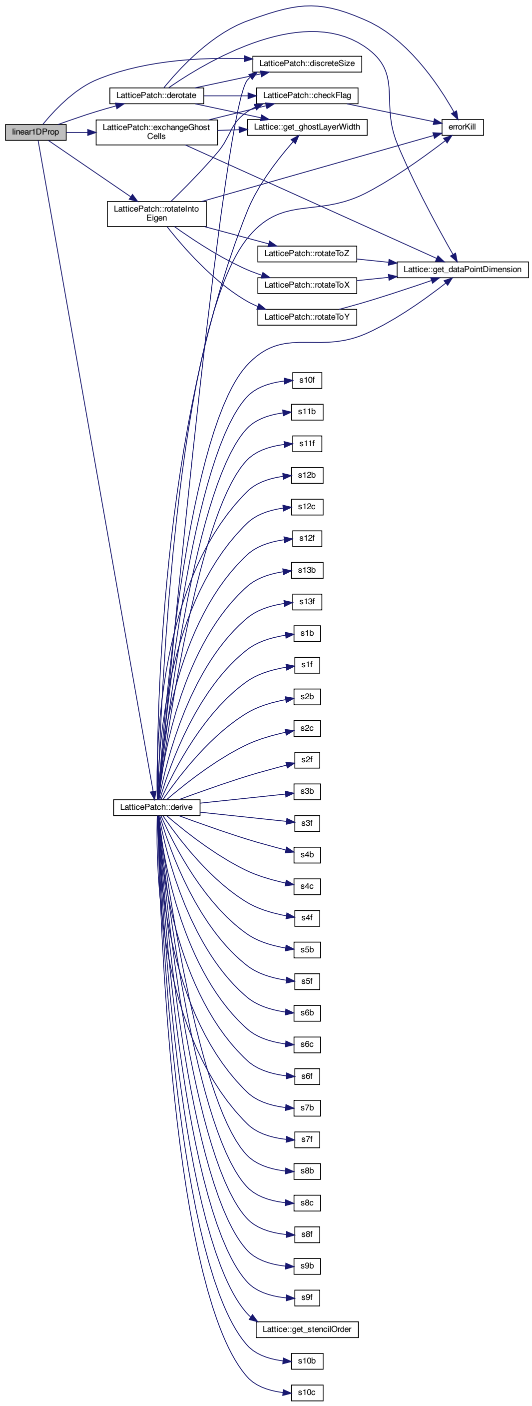
Maxwell propagation function for 1D – only for reference.

Definition at line 43 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 according to the scheme:
  data->exchangeGhostCells(1); // -> exchange halos
  data->rotateIntoEigen(
      1);           // -> rotate all data to prepare derivative operation
  data->derive(1); // -> perform derivative approximation operation on it
  data->derotate(
      1, dxData); // -> derotate derived data for ensuing time-evolution

  const sunindextype totalNP = data->discreteSize();
  sunindextype pp = 0;
  for (sunindextype i = 0; i < totalNP; i++) {
    pp = i * 6;
    /*
     simple vacuum Maxwell equations for the temporal derivatives using the
     spatial deriative only in x-direction without 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];
  }
}
```

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

Here is the call graph for this function:



6.30.2.2 linear2DProp()

```
void linear2DProp (
            LatticePatch * data,
```

```
N_Vector u,
N_Vector udot,
int * c )
```

Maxwell propagation function for 2D – only for reference.

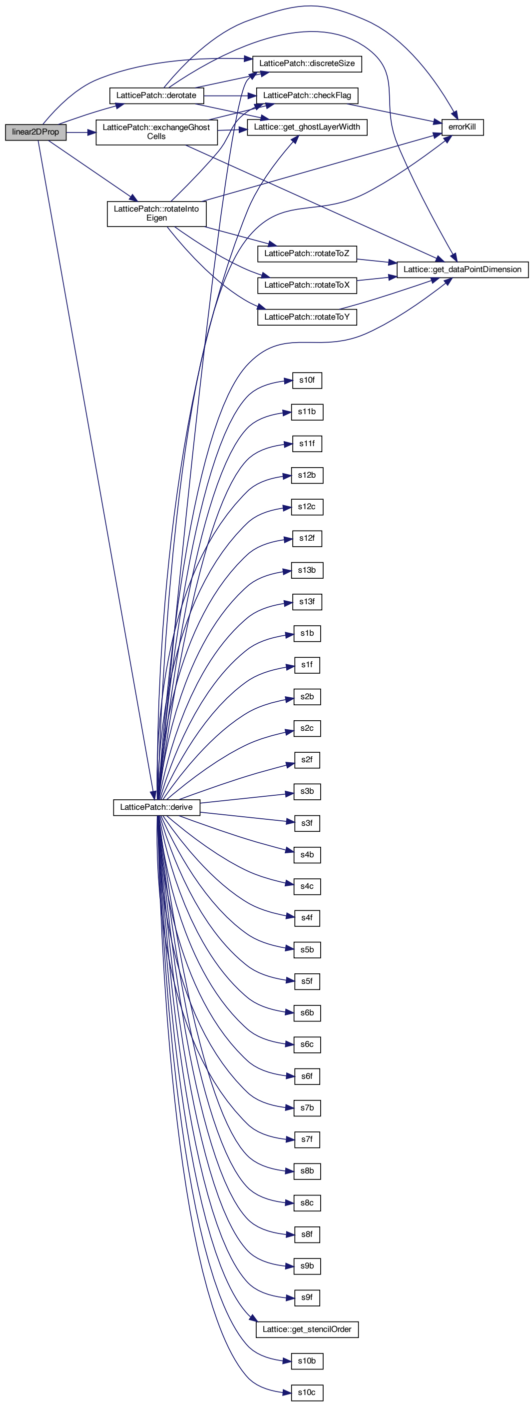
Maxwell propagation function for 2D – only for reference.

Definition at line 286 of file TimeEvolutionFunctions.cpp.

```
00286                                                                          {
00287
00288   sunrealtype *duData = data->duData;
00289   sunrealtype *dxData = data->buffData[1 - 1];
00290   sunrealtype *dyData = data->buffData[2 - 1];
00291
00292   data->exchangeGhostCells(1);
00293   data->rotateIntoEigen(1);
00294   data->derive(1);
00295   data->derotate(1, dxData);
00296   data->exchangeGhostCells(2);
00297   data->rotateIntoEigen(2);
00298   data->derive(2);
00299   data->derotate(2, dyData);
00300
00301   const sunindextype totalNP = data->discreteSize();
00302   sunindextype pp = 0;
00303   for (sunindextype i = 0; i < totalNP; i++) {
00304     pp = i * 6;
00305     duData[pp + 0] = dyData[pp + 5];
00306     duData[pp + 1] = -dxData[pp + 5];
00307     duData[pp + 2] = -dyData[pp + 3] + dxData[pp + 4];
00308     duData[pp + 3] = -dyData[pp + 2];
00309     duData[pp + 4] = dxData[pp + 2];
00310     duData[pp + 5] = dyData[pp + 0] - dxData[pp + 1];
00311   }
00312 }
```

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

Here is the call graph for this function:



6.30.2.3 linear3DProp()

```
void linear3DProp (
            LatticePatch * data,
```

```
N_Vector u,
N_Vector udot,
int * c )
```

Maxwell propagation function for 3D – only for reference.

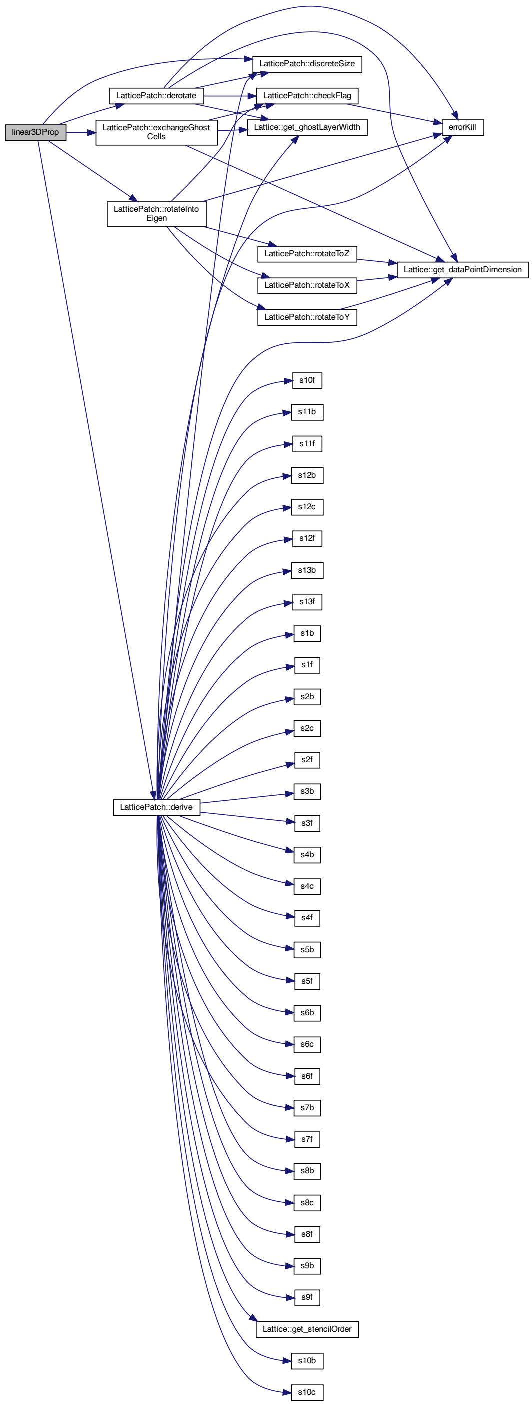
Maxwell propagation function for 3D – only for reference.

Definition at line 494 of file TimeEvolutionFunctions.cpp.

```
                                                                    {

  sunrealtype *duData = data->duData;
  sunrealtype *dxData = data->buffData[1 - 1];
  sunrealtype *dyData = data->buffData[2 - 1];
  sunrealtype *dzData = data->buffData[3 - 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);
  data->exchangeGhostCells(3);
  data->rotateIntoEigen(3);
  data->derive(3);
  data->derotate(3, dzData);

  const sunindextype totalNP = data->discreteSize();
  sunindextype pp = 0;
  for (sunindextype 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];
  }
}
```

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

Here is the call graph for this function:



6.30.2.4 nonlinear1DProp()

```
void nonlinear1DProp (
            LatticePatch * data,
```

N_Vector *u*,
N_Vector *udot*,
int * *c*)

HE propagation function for 1D.

HE propagation function for 1D.

Definition at line 76 of file TimeEvolutionFunctions.cpp.

```
00076                                                                              {
00077
00078   // NVector pointers to provided field values and their temp. derivatives
00079   sunrealtype *udata = NV_DATA_P(u),
00080               *dudata = NV_DATA_P(udot);
00081
00082   // pointer to spatial derivatives via pach data
00083   sunrealtype *dxData = data->buffData[1 - 1];
00084
00085   // same sequence as in the linear case
00086   data->exchangeGhostCells(1);
00087   data->rotateIntoEigen(1);
00088   data->derive(1);
00089   data->derotate(1, dxData);
00090
00091   /*
00092   F and G are nonzero in the nonlinear case,
00093   polarization and magnetization derivatives
00094   w.r.t. E- and B-field go into the e.o.m.
00095   */
00096   static sunrealtype f, g; // em field invariants F, G
00097   // derivatives of HE Lagrangian w.r.t. field invariants
00098   static sunrealtype lf, lff, lfg, lg, lgg;
00099   // matrix to hold derivatives of polarization and magnetization
00100   static std::array<sunrealtype, 21> JMM;
00101   // array to hold E^2 and B^2 components
00102   static std::array<sunrealtype, 6> Quad;
00103   // array to hold intermediate temp. derivatives of E and B
00104   static std::array<sunrealtype, 6> h;
00105   // determinant needed for explicit matrix inversion
00106   static sunrealtype detC = nan("0x12345");
00107
00108   // number of points in the patch
00109   const sunindextype totalNP = data->discreteSize();
00110   #pragma omp parallel for default(none) \
00111   private(JMM, Quad, h, detC) \
00112   shared(totalNP, c, f, g, lf, lff, lfg, lg, lgg, udata, dudata, dxData) \
00113   schedule(static)
00114   for (sunindextype pp = 0; pp < totalNP * 6;
00115        pp += 6) { // loop over all 6dim points in the patch
00116     // em field Lorentz invariants F and G
00117     f = 0.5 * ((Quad[0] = udata[pp] * udata[pp]) +
00118                (Quad[1] = udata[pp + 1] * udata[pp + 1]) +
00119                (Quad[2] = udata[pp + 2] * udata[pp + 2]) -
00120                (Quad[3] = udata[pp + 3] * udata[pp + 3]) -
00121                (Quad[4] = udata[pp + 4] * udata[pp + 4]) -
00122                (Quad[5] = udata[pp + 5] * udata[pp + 5]));
00123     g = udata[pp] * udata[pp + 3] + udata[pp + 1] * udata[pp + 4] +
00124         udata[pp + 2] * udata[pp + 5];
00125     // process/expansion order and corresponding derivative values of L
00126     // w.r.t. F, G
00127     switch (*c) {
00128     case 0:  // linear Maxwell vacuum
00129       lf = 0;
00130       lff = 0;
00131       lfg = 0;
00132       lg = 0;
00133       lgg = 0;
00134       break;
00135     case 1:  // only 4-photon processes
00136       lf = 0.000206527095658582755255648 * f;
00137       lff = 0.000206527095658582755255648;
00138       lfg = 0;
00139       lg = 0.000361422417402519821697384l * g;
00140       lgg = 0.000361422417402519821697384l;
00141       break;
00142     case 2:  // only 6-photon processes
00143       lf = 0.000354046449700427580438254 * f * f +
00144            0.000191775160254398272737387 * g * g;
00145       lff = 0.000708092899400855160876507 * f;
00146       lfg = 0.000383550320508796545474774 * g;
00147       lg = 0.000383550320508796545474774 * f * g;
00148       lgg = 0.000383550320508796545474774 * f;
00149       break;
```

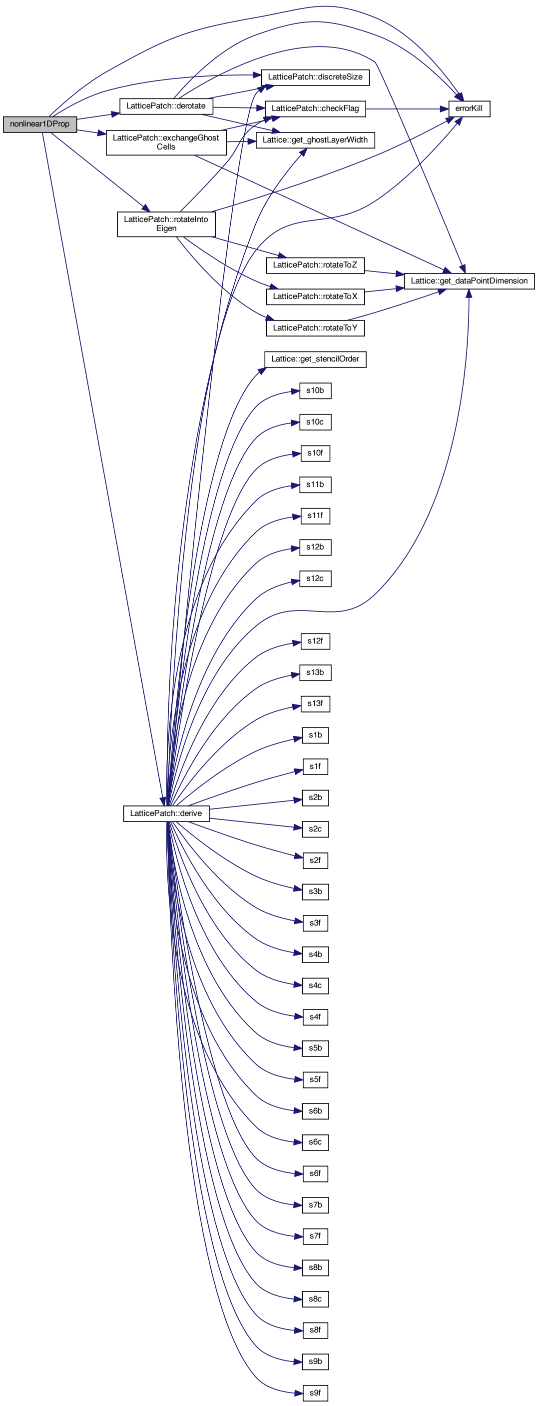
```
    case 3:  // 4- and 6-photon processes
      lf = (0.000206527095658582755255648 + 0.000354046449700427580438254 * f) *
               f +
           0.000191775160254398272737387 * g * g;
      lff = 0.000206527095658582755255648 + 0.000708092899400855160876507 5 * f;
      lfg = 0.000383550320508796545474774 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.");
    }

    // derivatives of polarization and magnetization w.r.t. E and B
    // Jpx(Ex)
    JMM[0] = lf + lff * Quad[0] +
             udata[3 + pp] * (2 * lfg * udata[pp] + lgg * udata[3 + pp]);
    // Jpx(Ey)
    JMM[1] =
        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];
    // Jpy(Ey)
    JMM[2] = lf + lff * Quad[1] +
             udata[4 + pp] * (2 * lfg * udata[1 + pp] + lgg * udata[4 + pp]);
    // Jpx(Ez) = Jpz(Ex)
    JMM[3] =
        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];
    // Jpy(Ez) = Jpz(Ey)
    JMM[4] = 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];
    // Jpz(Ez)
    JMM[5] = lf + lff * Quad[2] +
              udata[5 + pp] * (2 * lfg * udata[2 + pp] + lgg * udata[5 + pp]);
    // Jpx(Bx) = Jmx(Ex)
    JMM[6] = lg + lfg * (Quad[0] - Quad[3 + 0]) +
              (-lff + lgg) * udata[pp] * udata[3 + pp];
    // Jpy(Bx) = Jmx(Ey)
    JMM[7] = -(udata[3 + pp] * (lff * udata[1 + pp] + lfg * udata[4 + pp])) +
              udata[pp] * (lfg * udata[1 + pp] + lgg * udata[4 + pp]);
    // Jpz(Bx) = Jmx(Ez)
    JMM[8] = -(udata[3 + pp] * (lff * udata[2 + pp] + lfg * udata[5 + pp])) +
              udata[pp] * (lfg * udata[2 + pp] + lgg * udata[5 + pp]);
    // Jmx(Bx)
    JMM[9] = -lf + lgg * Quad[0] +
              udata[3 + pp] * (-2 * lfg * udata[pp] + lff * udata[3 + pp]);
    // Jpx(By) = Jmy(Ex)
    JMM[10] = udata[1 + pp] * (lfg * udata[pp] + lgg * udata[3 + pp]) -
               (lff * udata[pp] + lfg * udata[3 + pp]) * udata[4 + pp];
    // Jpy(By) = Jmy(Ey)
    JMM[11] = lg + lfg * (Quad[1] - Quad[4 + 0]) +
               (-lff + lgg) * udata[1 + pp] * udata[4 + pp];
    // Jpz(By) = Jmy(Ez)
    JMM[12] = -(udata[4 + pp] * (lff * udata[2 + pp] + lfg * udata[5 + pp])) +
              udata[1 + pp] * (lfg * udata[2 + pp] + lgg * udata[5 + pp]);
    // Jmx(By) = Jmy(Bx)
    JMM[13] = 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]);
    // Jmy(By)
    JMM[14] = -lf + lgg * Quad[1] +
              udata[4 + pp] * (-2 * lfg * udata[1 + pp] + lff * udata[4 + pp]);
    // Jmz(Ex) = Jpx(Bz)
    JMM[15] = udata[2 + pp] * (lfg * udata[pp] + lgg * udata[3 + pp]) -
               (lff * udata[pp] + lfg * udata[3 + pp]) * udata[5 + pp];
    // Jmz(Ey) = Jpy(Bz)
    JMM[16] = udata[2 + pp] * (lfg * udata[1 + pp] + lgg * udata[4 + pp]) -
               (lff * udata[1 + pp] + lfg * udata[4 + pp]) * udata[5 + pp];
    // Jpz(Bz) = Jmz(Ez)
    JMM[17] = lg + lfg * (Quad[2] - Quad[5 + 0]) +
               (-lff + lgg) * udata[2 + pp] * udata[5 + pp];
    // Jmz(Bx) = Jmx(Bz)
    JMM[18] = 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]);
    // Jmy(Bz) = Jmz(By)
    JMM[19] =
        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]);
    // Jmz(Bz)
    JMM[20] = -lf + lgg * Quad[2] +
```

```
              udata[5 + pp] * (-2 * lfg * udata[2 + pp] + lff * udata[5 + pp]);

    // apply Z
    // top bock: -QJm(E)*E, Q-QJm(B)*B
    h[0] = 0;
    h[1] = dxData[pp] * JMM[15] + dxData[1 + pp] * JMM[16] +
           dxData[2 + pp] * JMM[17] + dxData[3 + pp] * JMM[18] +
           dxData[4 + pp] * JMM[19] + dxData[5 + pp] * (-1 + JMM[20]);
    h[2] = -(dxData[pp] * JMM[10]) - dxData[1 + pp] * JMM[11] -
           dxData[2 + pp] * JMM[12] - dxData[3 + pp] * JMM[13] +
           dxData[4 + pp] * (1 - JMM[14]) - dxData[5 + pp] * JMM[19];
    // bottom blocks: -Q*E
    h[3] = 0;
    h[4] = dxData[2 + pp];
    h[5] = -dxData[1 + pp];
    // (1+A)^-1 applies only to E components
    // -Jp(B)*B
    h[0] -= h[3] * JMM[6] + h[4] * JMM[10] + h[5] * JMM[15];
    h[1] -= h[3] * JMM[7] + h[4] * JMM[11] + h[5] * JMM[16];
    h[2] -= h[3] * JMM[8] + h[4] * JMM[12] + h[5] * JMM[17];
    // apply C^-1 explicitly, with C=1+Jp(E)
    dudata[pp + 0] =
        h[2] * (-(JMM[3] * (1 + JMM[2])) + JMM[1] * JMM[4]) +
        h[1] * (JMM[3] * JMM[4] - JMM[1] * (1 + JMM[5])) +
        h[0] * (1 - JMM[4] * JMM[4] + JMM[5] + JMM[2] * (1 + JMM[5]));
    dudata[pp + 1] =
        h[2] * (JMM[3] * JMM[1] - (1 + JMM[0]) * JMM[4]) +
        h[1] * (1 - JMM[3] * JMM[3] + JMM[5] + JMM[0] * (1 + JMM[5])) +
        h[0] * (JMM[4] * JMM[3] - JMM[1] * (1 + JMM[5]));
    dudata[pp + 2] =
        h[2] * (1 - JMM[1] * JMM[1] + JMM[2] + JMM[0] * (1 + JMM[2])) +
        h[1] * (JMM[1] * JMM[3] - (1 + JMM[0]) * JMM[4]) +
        h[0] * (-((1 + JMM[2]) * JMM[3]) + JMM[1] * JMM[4]);
    detC =  // determinant of C
        -((1 + JMM[2]) * (-1 + JMM[3] * JMM[3])) +
        (JMM[3] * JMM[1] - JMM[4]) * JMM[4] + JMM[5] + JMM[2] * JMM[5] +
        JMM[0] * (1 + JMM[2] - JMM[4] * JMM[4] + (1 + JMM[2]) * JMM[5]) -
        JMM[1] * (-(JMM[4] * JMM[3]) + JMM[1] * (1 + JMM[5]));
    dudata[pp + 0] /= detC;
    dudata[pp + 1] /= detC;
    dudata[pp + 2] /= detC;
    dudata[pp + 3] = h[3];
    dudata[pp + 4] = h[4];
    dudata[pp + 5] = h[5];
  }
  return;
}
```

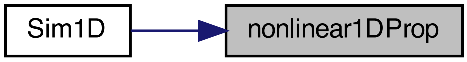
References LatticePatch::buffData, LatticePatch::derive(), LatticePatch::derotate(), LatticePatch::discreteSize(), errorKill(), LatticePatch::exchangeGhostCells(), and LatticePatch::rotateIntoEigen().

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 315 of file TimeEvolutionFunctions.cpp.

```
00315                                                                                 {
00316
00317   sunrealtype *udata = NV_DATA_P(u),
00318               *dudata = NV_DATA_P(udot);
00319
00320   sunrealtype *dxData = data->buffData[1 - 1];
00321   sunrealtype *dyData = data->buffData[2 - 1];
00322
00323   data->exchangeGhostCells(1);
00324   data->rotateIntoEigen(1);
00325   data->derive(1);
00326   data->derotate(1, dxData);
00327   data->exchangeGhostCells(2);
00328   data->rotateIntoEigen(2);
00329   data->derive(2);
00330   data->derotate(2, dyData);
00331
00332   static sunrealtype f, g;
00333   static sunrealtype lf, lff, lfg, lg, lgg;
00334   static std::array<sunrealtype, 21> JMM;
00335   static std::array<sunrealtype, 6> Quad;
00336   static std::array<sunrealtype, 6> h;
00337   static sunrealtype detC;
00338
00339   const sunindextype totalNP = data->discreteSize();
00340   #pragma omp parallel for default(none) \
00341   private(JMM, Quad, h, detC) \
00342   shared(totalNP, c, f, g, lf, lff, lfg, lg, lgg, udata, dudata,\
00343           dxData, dyData) \
00344   schedule(static)
00345   for (sunindextype pp = 0; pp < totalNP * 6; pp += 6) {
00346     f = 0.5 * ((Quad[0] = udata[pp] * udata[pp]) +
00347                (Quad[1] = udata[pp + 1] * udata[pp + 1]) +
00348                (Quad[2] = udata[pp + 2] * udata[pp + 2]) -
00349                (Quad[3] = udata[pp + 3] * udata[pp + 3]) -
00350                (Quad[4] = udata[pp + 4] * udata[pp + 4]) -
00351                (Quad[5] = udata[pp + 5] * udata[pp + 5]));
00352     g = udata[pp] * udata[pp + 3] + udata[pp + 1] * udata[pp + 4] +
00353         udata[pp + 2] * udata[pp + 5];
00354     switch (*c) {
00355     case 0:
00356       lf = 0;
00357       lff = 0;
```

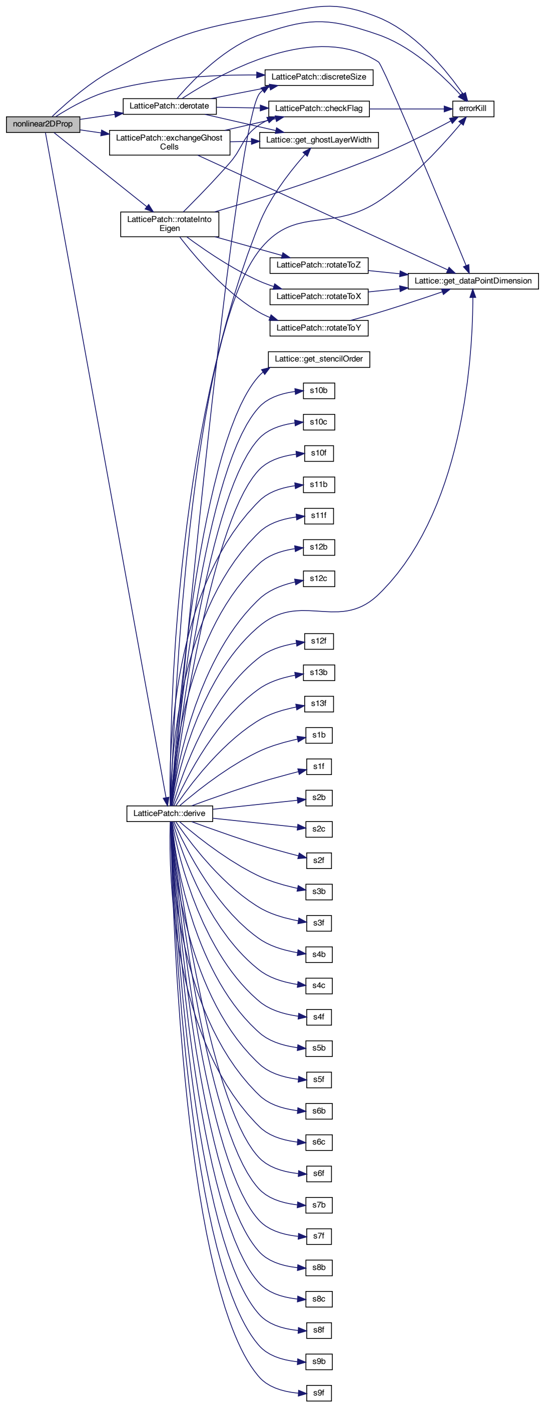
```
      lfg = 0;
      lg = 0;
      lgg = 0;
      break;
    case 1:
      lf = 0.000206527095658582755255648 * f;
      lff = 0.000206527095658582755255648;
      lfg = 0;
      lg = 0.0003614224174025198216973841 * g;
      lgg = 0.0003614224174025198216973841;
      break;
    case 2:
      lf = 0.000354046449700427580438254 * f * f +
           0.000191775160254398272737387 * g * g;
      lff = 0.0007080928994008551608765075 * f;
      lfg = 0.0003835503205087965454747749 * g;
      lg = 0.0003835503205087965454747749 * f * g;
      lgg = 0.0003835503205087965454747749 * f;
      break;
    case 3:
      lf = (0.000206527095658582755255648 + 0.000354046449700427580438254 * f) *
               f +
           0.000191775160254398272737387 * g * g;
      lff = 0.000206527095658582755255648 + 0.000708092899400855160876508 * f;
      lfg = 0.0003835503205087965454747749 * 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.");
    }

    JMM[0] = lf + lff * Quad[0] +
             udata[3 + pp] * (2 * lfg * udata[pp] + lgg * udata[3 + pp]);
    JMM[1] =
        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[2] = lf + lff * Quad[1] +
             udata[4 + pp] * (2 * lfg * udata[1 + pp] + lgg * udata[4 + pp]);
    JMM[3] =
        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[4] = 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[5] = lf + lff * Quad[2] +
              udata[5 + pp] * (2 * lfg * udata[2 + pp] + lgg * udata[5 + pp]);
    JMM[6] = lg + lfg * (Quad[0] - Quad[3 + 0]) +
              (-lff + lgg) * udata[pp] * udata[3 + pp];
    JMM[7] = -(udata[3 + pp] * (lff * udata[1 + pp] + lfg * udata[4 + pp])) +
              udata[pp] * (lfg * udata[1 + pp] + lgg * udata[4 + pp]);
    JMM[8] = -(udata[3 + pp] * (lff * udata[2 + pp] + lfg * udata[5 + pp])) +
              udata[pp] * (lfg * udata[2 + pp] + lgg * udata[5 + pp]);
    JMM[9] = -lf + lgg * Quad[0] +
              udata[3 + pp] * (-2 * lfg * udata[pp] + lff * udata[3 + pp]);
    JMM[10] = udata[1 + pp] * (lfg * udata[pp] + lgg * udata[3 + pp]) -
              (lff * udata[pp] + lfg * udata[3 + pp]) * udata[4 + pp];
    JMM[11] = lg + lfg * (Quad[1] - Quad[4 + 0]) +
              (-lff + lgg) * udata[1 + pp] * udata[4 + pp];
    JMM[12] = -(udata[4 + pp] * (lff * udata[2 + pp] + lfg * udata[5 + pp])) +
              udata[1 + pp] * (lfg * udata[2 + pp] + lgg * udata[5 + pp]);
    JMM[13] = 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[14] = -lf + lgg * Quad[1] +
              udata[4 + pp] * (-2 * lfg * udata[1 + pp] + lff * udata[4 + pp]);
    JMM[15] = udata[2 + pp] * (lfg * udata[pp] + lgg * udata[3 + pp]) -
              (lff * udata[pp] + lfg * udata[3 + pp]) * udata[5 + pp];
    JMM[16] = udata[2 + pp] * (lfg * udata[1 + pp] + lgg * udata[4 + pp]) -
              (lff * udata[1 + pp] + lfg * udata[4 + pp]) * udata[5 + pp];
    JMM[17] = lg + lfg * (Quad[2] - Quad[5 + 0]) +
              (-lff + lgg) * udata[2 + pp] * udata[5 + pp];
    JMM[18] = 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[19] =
        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[20] = -lf + lgg * Quad[2] +
              udata[5 + pp] * (-2 * lfg * udata[2 + pp] + lff * udata[5 + pp]);

    h[0] = 0;
    h[1] = dxData[pp] * JMM[15] + dxData[1 + pp] * JMM[16] +
```

```
            dxData[2 + pp] * JMM[17] + dxData[3 + pp] * JMM[18] +
            dxData[4 + pp] * JMM[19] + dxData[5 + pp] * (-1 + JMM[20]);
    h[2] = -(dxData[pp] * JMM[10]) - dxData[1 + pp] * JMM[11] -
            dxData[2 + pp] * JMM[12] - dxData[3 + pp] * JMM[13] +
            dxData[4 + pp] * (1 - JMM[14]) - dxData[5 + pp] * JMM[19];
    h[3] = 0;
    h[4] = dxData[2 + pp];
    h[5] = -dxData[1 + pp];
    h[0] += -(dyData[pp] * JMM[15]) - dyData[1 + pp] * JMM[16] -
             dyData[2 + pp] * JMM[17] - dyData[3 + pp] * JMM[18] -
             dyData[4 + pp] * JMM[19] + dyData[5 + pp] * (1 - JMM[20]);
    h[1] += 0;
    h[2] += dyData[pp] * JMM[6] + dyData[1 + pp] * JMM[7] +
             dyData[2 + pp] * JMM[8] + dyData[3 + pp] * (-1 + JMM[9]) +
             dyData[4 + pp] * JMM[13] + dyData[5 + pp] * JMM[18];
    h[3] += -dyData[2 + pp];
    h[4] += 0;
    h[5] += dyData[pp];
    h[0] -= h[3] * JMM[6] + h[4] * JMM[10] + h[5] * JMM[15];
    h[1] -= h[3] * JMM[7] + h[4] * JMM[11] + h[5] * JMM[16];
    h[2] -= h[3] * JMM[8] + h[4] * JMM[12] + h[5] * JMM[17];
    dudata[pp + 0] =
        h[2] * (-(JMM[3] * (1 + JMM[2])) + JMM[1] * JMM[4]) +
        h[1] * (JMM[3] * JMM[4] - JMM[1] * (1 + JMM[5])) +
        h[0] * (1 - JMM[4] * JMM[4] + JMM[5] + JMM[2] * (1 + JMM[5]));
    dudata[pp + 1] =
        h[2] * (JMM[3] * JMM[1] - (1 + JMM[0]) * JMM[4]) +
        h[1] * (1 - JMM[3] * JMM[3] + JMM[5] + JMM[0] * (1 + JMM[5])) +
        h[0] * (JMM[4] * JMM[3] - JMM[1] * (1 + JMM[5]));
    dudata[pp + 2] =
        h[2] * (1 - JMM[1] * JMM[1] + JMM[2] + JMM[0] * (1 + JMM[2])) +
        h[1] * (JMM[1] * JMM[3] - (1 + JMM[0]) * JMM[4]) +
        h[0] * (-((1 + JMM[2]) * JMM[3]) + JMM[1] * JMM[4]);
    detC =
        -((1 + JMM[2]) * (-1 + JMM[3] * JMM[3])) +
        (JMM[3] * JMM[1] - JMM[4]) * JMM[4] + JMM[5] + JMM[2] * JMM[5] +
        JMM[0] * (1 + JMM[2] - JMM[4] * JMM[4] + (1 + JMM[2]) * JMM[5]) -
        JMM[1] * (-(JMM[4] * JMM[3]) + JMM[1] * (1 + JMM[5]));
    dudata[pp + 0] /= detC;
    dudata[pp + 1] /= detC;
    dudata[pp + 2] /= detC;
    dudata[pp + 3] = h[3];
    dudata[pp + 4] = h[4];
    dudata[pp + 5] = h[5];
  }
  return;
}
```

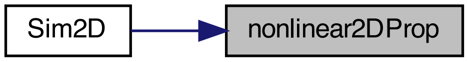
References LatticePatch::buffData, LatticePatch::derive(), LatticePatch::derotate(), LatticePatch::discreteSize(), errorKill(), LatticePatch::exchangeGhostCells(), and LatticePatch::rotateIntoEigen().

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 528 of file TimeEvolutionFunctions.cpp.

```
00528 {
00529
00530   sunrealtype *udata = NV_DATA_P(u),
00531               *dudata = NV_DATA_P(udot);
00532
00533   sunrealtype *dxData = data->buffData[1 - 1];
00534   sunrealtype *dyData = data->buffData[2 - 1];
00535   sunrealtype *dzData = data->buffData[3 - 1];
00536
00537   data->exchangeGhostCells(1);
00538   data->rotateIntoEigen(1);
00539   data->derive(1);
00540   data->derotate(1,dxData);
00541   data->exchangeGhostCells(2);
00542   data->rotateIntoEigen(2);
00543   data->derive(2);
00544   data->derotate(2,dyData);
00545   data->exchangeGhostCells(3);
00546   data->rotateIntoEigen(3);
00547   data->derive(3);
00548   data->derotate(3,dzData);
00549
00550   static sunrealtype f, g;
00551   static sunrealtype lf, lff, lfg, lg, lgg;
00552   static std::array<sunrealtype, 21> JMM;
00553   static std::array<sunrealtype, 6> Quad;
00554   static std::array<sunrealtype, 6> h;
00555   static sunrealtype detC = nan("0x12345");
00556
00557   const sunindextype totalNP = data->discreteSize();
00558   #pragma omp parallel for default(none) \
00559   private(JMM, Quad, h, detC) \
00560   shared(totalNP, c, f, g, lf, lff, lfg, lg, lgg, udata, dudata,\
00561          dxData, dyData, dzData) \
00562   schedule(static)
00563   for (sunindextype pp = 0; pp < totalNP * 6; pp += 6) {
00564     f = 0.5 * ((Quad[0] = udata[pp] * udata[pp]) +
00565                (Quad[1] = udata[pp + 1] * udata[pp + 1]) +
00566                (Quad[2] = udata[pp + 2] * udata[pp + 2]) -
00567                (Quad[3] = udata[pp + 3] * udata[pp + 3]) -
00568                (Quad[4] = udata[pp + 4] * udata[pp + 4]) -
00569                (Quad[5] = udata[pp + 5] * udata[pp + 5]));
00570     g = udata[pp] * udata[pp + 3] + udata[pp + 1] * udata[pp + 4] +
```

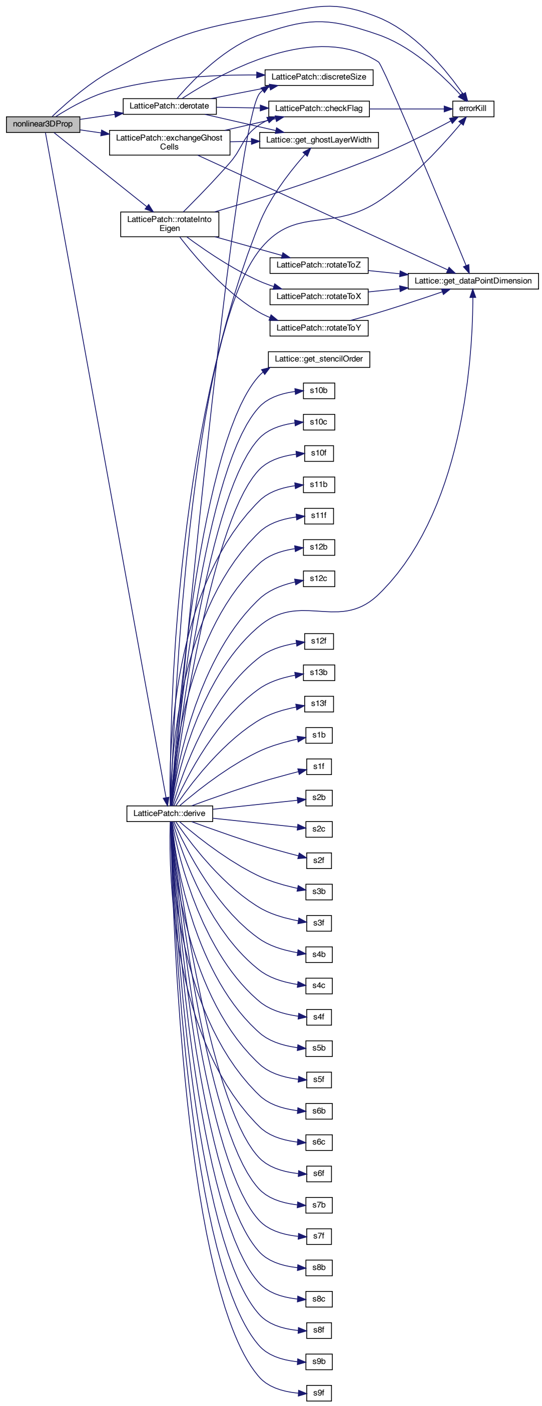
```
      udata[pp + 2] * udata[pp + 5];
  switch (*c) {
  case 0:
    lf = 0;
    lff = 0;
    lfg = 0;
    lg = 0;
    lgg = 0;
    break;
  case 1:
    lf = 0.00020652709565858275525564 8 * f;
    lff = 0.000206527095658582755255648;
    lfg = 0;
    lg = 0.000361422417402519821697384 1 * g;
    lgg = 0.0003614224174025198216973841;
    break;
  case 2:
    lf = 0.00035404644970042758043825 4 * f * f +
         0.00019177516025439827273738 7 * 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 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.");
  }

  JMM[0] = lf + lff * Quad[0] +
           udata[3 + pp] * (2 * lfg * udata[pp] + lgg * udata[3 + pp]);
  JMM[1] =
      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[2] = lf + lff * Quad[1] +
           udata[4 + pp] * (2 * lfg * udata[1 + pp] + lgg * udata[4 + pp]);
  JMM[3] =
      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[4] = 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[5] = lf + lff * Quad[2] +
            udata[5 + pp] * (2 * lfg * udata[2 + pp] + lgg * udata[5 + pp]);
  JMM[6] = lg + lfg * (Quad[0] - Quad[3 + 0]) +
            (-lff + lgg) * udata[pp] * udata[3 + pp];
  JMM[7] = -(udata[3 + pp] * (lff * udata[1 + pp] + lfg * udata[4 + pp])) +
            udata[pp] * (lfg * udata[1 + pp] + lgg * udata[4 + pp]);
  JMM[8] = -(udata[3 + pp] * (lff * udata[2 + pp] + lfg * udata[5 + pp])) +
            udata[pp] * (lfg * udata[2 + pp] + lgg * udata[5 + pp]);
  JMM[9] = -lf + lgg * Quad[0] +
            udata[3 + pp] * (-2 * lfg * udata[pp] + lff * udata[3 + pp]);
  JMM[10] = udata[1 + pp] * (lfg * udata[pp] + lgg * udata[3 + pp]) -
            (lff * udata[pp] + lfg * udata[3 + pp]) * udata[4 + pp];
  JMM[11] = lg + lfg * (Quad[1] - Quad[4 + 0]) +
            (-lff + lgg) * udata[1 + pp] * udata[4 + pp];
  JMM[12] = -(udata[4 + pp] * (lff * udata[2 + pp] + lfg * udata[5 + pp])) +
            udata[1 + pp] * (lfg * udata[2 + pp] + lgg * udata[5 + pp]);
  JMM[13] = 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[14] = -lf + lgg * Quad[1] +
            udata[4 + pp] * (-2 * lfg * udata[1 + pp] + lff * udata[4 + pp]);
  JMM[15] = udata[2 + pp] * (lfg * udata[pp] + lgg * udata[3 + pp]) -
            (lff * udata[pp] + lfg * udata[3 + pp]) * udata[5 + pp];
  JMM[16] = udata[2 + pp] * (lfg * udata[1 + pp] + lgg * udata[4 + pp]) -
            (lff * udata[1 + pp] + lfg * udata[4 + pp]) * udata[5 + pp];
  JMM[17] = lg + lfg * (Quad[2] - Quad[5 + 0]) +
            (-lff + lgg) * udata[2 + pp] * udata[5 + pp];
  JMM[18] = 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[19] =
      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[20] = -lf + lgg * Quad[2] +
              udata[5 + pp] * (-2 * lfg * udata[2 + pp] + lff * udata[5 + pp]);

    h[0] = 0;
    h[1] = dxData[pp] * JMM[15] + dxData[1 + pp] * JMM[16] +
           dxData[2 + pp] * JMM[17] + dxData[3 + pp] * JMM[18] +
           dxData[4 + pp] * JMM[19] + dxData[5 + pp] * (-1 + JMM[20]);
    h[2] = -(dxData[pp] * JMM[10]) - dxData[1 + pp] * JMM[11] -
           dxData[2 + pp] * JMM[12] - dxData[3 + pp] * JMM[13] +
           dxData[4 + pp] * (1 - JMM[14]) - dxData[5 + pp] * JMM[19];
    h[3] = 0;
    h[4] = dxData[2 + pp];
    h[5] = -dxData[1 + pp];
    h[0] += -(dyData[pp] * JMM[15]) - dyData[1 + pp] * JMM[16] -
            dyData[2 + pp] * JMM[17] - dyData[3 + pp] * JMM[18] -
            dyData[4 + pp] * JMM[19] + dyData[5 + pp] * (1 - JMM[20]);
    h[1] += 0;
    h[2] += dyData[pp] * JMM[6] + dyData[1 + pp] * JMM[7] +
            dyData[2 + pp] * JMM[8] + dyData[3 + pp] * (-1 + JMM[9]) +
            dyData[4 + pp] * JMM[13] + dyData[5 + pp] * JMM[18];
    h[3] += -dyData[2 + pp];
    h[4] += 0;
    h[5] += dyData[pp];
    h[0] += dzData[pp] * JMM[10] + dzData[1 + pp] * JMM[11] +
            dzData[2 + pp] * JMM[12] + dzData[3 + pp] * JMM[13] +
            dzData[4 + pp] * (-1 + JMM[14]) + dzData[5 + pp] * JMM[19];
    h[1] += -(dzData[pp] * JMM[6]) - dzData[1 + pp] * JMM[7] -
            dzData[2 + pp] * JMM[8] + dzData[3 + pp] * (1 - JMM[9]) -
            dzData[4 + pp] * JMM[13] - dzData[5 + pp] * JMM[18];
    h[2] += 0;
    h[3] += dzData[1 + pp];
    h[4] += -dzData[pp];
    h[5] += 0;
    h[0] -= h[3] * JMM[6] + h[4] * JMM[10] + h[5] * JMM[15];
    h[1] -= h[3] * JMM[7] + h[4] * JMM[11] + h[5] * JMM[16];
    h[2] -= h[3] * JMM[8] + h[4] * JMM[12] + h[5] * JMM[17];
    dudata[pp + 0] =
        h[2] * (-(JMM[3] * (1 + JMM[2])) + JMM[1] * JMM[4]) +
        h[1] * (JMM[3] * JMM[4] - JMM[1] * (1 + JMM[5])) +
        h[0] * (1 - JMM[4] * JMM[4] + JMM[5] + JMM[2] * (1 + JMM[5]));
    dudata[pp + 1] =
        h[2] * (JMM[3] * JMM[1] - (1 + JMM[0]) * JMM[4]) +
        h[1] * (1 - JMM[3] * JMM[3] + JMM[5] + JMM[0] * (1 + JMM[5])) +
        h[0] * (JMM[4] * JMM[3] - JMM[1] * (1 + JMM[5]));
    dudata[pp + 2] =
        h[2] * (1 - JMM[1] * JMM[1] + JMM[2] + JMM[0] * (1 + JMM[2])) +
        h[1] * (JMM[1] * JMM[3] - (1 + JMM[0]) * JMM[4]) +
        h[0] * (-((1 + JMM[2]) * JMM[3]) + JMM[1] * JMM[4]);
    detC =
        -((1 + JMM[2]) * (-1 + JMM[3] * JMM[3])) +
        (JMM[3] * JMM[1] - JMM[4]) * JMM[4] + JMM[5] + JMM[2] * JMM[5] +
        JMM[0] * (1 + JMM[2] - JMM[4] * JMM[4] + (1 + JMM[2]) * JMM[5]) -
        JMM[1] * (-(JMM[4] * JMM[3]) + JMM[1] * (1 + JMM[5]));
    dudata[pp + 0] /= detC;
    dudata[pp + 1] /= detC;
    dudata[pp + 2] /= detC;
    dudata[pp + 3] = h[3];
    dudata[pp + 4] = h[4];
    dudata[pp + 5] = h[5];
  }
  return;
}
```

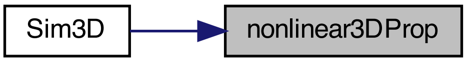
References LatticePatch::buffData, LatticePatch::derive(), LatticePatch::derotate(), LatticePatch::discreteSize(), errorKill(), LatticePatch::exchangeGhostCells(), and LatticePatch::rotateIntoEigen().

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