PACS Project di Sonia Felotti

Generated by Doxygen 1.8.17

1 README 1

2 Namespace Index 3

2.1 Namespace List . . . 3

3 Hierarchical Index 5

3.1 Class Hierarchy . . . 5

4 Class Index 7

4.1 Class List . . . 7

5 File Index 9

5.1 File List . . . 9

6 Namespace Documentation 11

6.1 EquationData Namespace Reference . . . 11

6.2 Euler_DG Namespace Reference . . . 11

6.2.1 Typedef Documentation . . . 12

6.2.1.1 Number . . . 12

6.2.2 Function Documentation . . . 12

6.2.2.1 cell_number() . . . 12

6.2.2.2 minmod() . . . 12

6.2.2.3 vanAlbada() . . . 13

6.2.2.4 euler_velocity() . . . 13

6.2.2.5 euler_pressure() . . . 13

6.2.2.6 euler_flux() . . . 13

6.2.2.7 operator∗() . . . 13

6.2.2.8 sgn() . . . 14

6.2.2.9 euler_numerical_flux() . . . 14

6.2.2.10 evaluate_function() [1/2] . . . 14

6.2.2.11 evaluate_function() [2/2] . . . 14

6.2.3 Variable Documentation . . . 14

6.2.3.1 dimension . . . 14

6.3 Parameters Namespace Reference . . . 15

6.3.1 Detailed Description . . . 15

7 Class Documentation 17

7.1 EquationData::ExactSolution< dim > Class Template Reference . . . 17

7.1.1 Detailed Description . . . 17

7.1.2 Constructor & Destructor Documentation . . . 17

7.1.2.1 ExactSolution() . . . 18

7.1.3 Member Function Documentation . . . 18

7.1.3.1 value() . . . 18

7.1.4 Member Data Documentation . . . 18

7.1.4.1 parameters . . . 18
7.2 EquationData::InitialData< dim > Class Template Reference . . . 18
7.2.1 Detailed Description . . . 19
7.2.2 Constructor & Destructor Documentation . . . 19
7.2.2.1 InitialData() . . . 19
7.2.3 Member Function Documentation . . . 19
7.2.3.1 value() . . . 19
7.2.4 Member Data Documentation . . . 19
7.2.4.1 parameters . . . 19
7.3 EquationData::BoundaryData< dim > Class Template Reference . . . 20
7.3.1 Detailed Description . . . 20
7.3.2 Constructor & Destructor Documentation . . . 20
7.3.2.1 BoundaryData() . . . 20
7.3.3 Member Function Documentation . . . 21
7.3.3.1 value() . . . 21
7.3.4 Member Data Documentation . . . 21
7.3.4.1 a . . . 21
7.3.4.2 parameters . . . 21
7.4 Parameters::Data_Storage Struct Reference . . . 21
7.4.1 Detailed Description . . . 22
7.4.2 Constructor & Destructor Documentation . . . 22
7.4.2.1 Data_Storage() . . . 22
7.4.3 Member Function Documentation . . . 22
7.4.3.1 read_data() . . . 23
7.4.4 Member Data Documentation . . . 23
7.4.4.1 testcase . . . 23
7.4.4.2 n_stages . . . 23
7.4.4.3 fe_degree . . . 23
7.4.4.4 fe_degree_Q0 . . . 23
7.4.4.5 n_q_points_1d . . . 23
7.4.4.6 n_q_points_1d_Q0 . . . 24
7.4.4.7 max_loc_refinements . . . 24
7.4.4.8 min_loc_refinements . . . 24
7.4.4.9 gamma . . . 24
7.4.4.10 beta_density . . . 24
7.4.4.11 beta_momentum . . . 24
7.4.4.12 beta_energy . . . 24
7.4.4.13 beta . . . 24
7.4.4.14 M . . . 25
7.4.4.15 positivity . . . 25
7.4.4.16 function_limiter . . . 25
7.4.4.17 type . . . 25

7.4.4.18 refine . . . 25
7.4.4.19 refine_indicator . . . 25
7.4.4.20 final_time . . . 25
7.4.4.21 output_tick . . . 25
7.4.4.22 prm . . . 26
7.5 Parameters::Solver Struct Reference . . . 26
7.5.1 Detailed Description . . . 26
7.5.2 Member Enumeration Documentation . . . 26
7.5.2.1 EulerNumericalFlux . . . 26
7.5.3 Member Function Documentation . . . 27
7.5.3.1 declare_parameters() . . . 27
7.5.3.2 parse_parameters() . . . 27
7.5.4 Member Data Documentation . . . 27
7.5.4.1 numerical_flux_type . . . 27
7.6 Euler_DG::EulerOperator< dim, degree, n_points_1d > Class Template Reference . . . 27
7.6.1 Detailed Description . . . 29
7.6.2 Constructor & Destructor Documentation . . . 29
7.6.2.1 EulerOperator() . . . 29
7.6.3 Member Function Documentation . . . 29
7.6.3.1 reinit() . . . 29
7.6.3.2 set_inflow_boundary() . . . 30
7.6.3.3 set_subsonic_outflow_boundary() . . . 30
7.6.3.4 set_supersonic_outflow_boundary() . . . 30
7.6.3.5 set_wall_boundary() . . . 30
7.6.3.6 set_body_force() . . . 30
7.6.3.7 apply() . . . 31
7.6.3.8 project() . . . 31
7.6.3.9 compute_errors() . . . 32
7.6.3.10 compute_cell_transport_speed() . . . 32
7.6.3.11 initialize_vector() . . . 32
7.6.3.12 local_apply_inverse_mass_matrix() . . . 32
7.6.3.13 local_apply_cell() . . . 33
7.6.3.14 local_apply_face() . . . 34
7.6.3.15 local_apply_boundary_face() . . . 34
7.6.4 Member Data Documentation . . . 35
7.6.4.1 n_quadrature_points_1d . . . 35
7.6.4.2 parameters . . . 35
7.6.4.3 data . . . 35
7.6.4.4 timer . . . 35
7.6.4.5 inflow_boundaries . . . 36
7.6.4.6 subsonic_outflow_boundaries . . . 36
7.6.4.7 supersonic_outflow_boundaries . . . 36

7.6.4.8 wall_boundaries . . . 36
7.6.4.9 body_force . . . 36
7.7 Euler_DG::EulerProblem< dim > Class Template Reference . . . 36
7.7.1 Detailed Description . . . 38
7.7.2 Constructor & Destructor Documentation . . . 38
7.7.2.1 EulerProblem() . . . 38
7.7.3 Member Function Documentation . . . 39
7.7.3.1 run() . . . 39
7.7.3.2 make_grid() . . . 39
7.7.3.3 adapt_mesh() . . . 39
7.7.3.4 make_dofs() . . . 40
7.7.3.5 compute_cell_average() . . . 40
7.7.3.6 get_cell_average() . . . 40
7.7.3.7 compute_shock_indicator() . . . 40
7.7.3.8 apply_limiter_TVB() . . . 40
7.7.3.9 apply_filter() . . . 41
7.7.3.10 update() . . . 41
7.7.3.11 output_results() . . . 41
7.7.4 Member Data Documentation . . . 42
7.7.4.1 parameters . . . 42
7.7.4.2 solution . . . 42
7.7.4.3 tmp_solution . . . 42
7.7.4.4 sol_aux . . . 42
7.7.4.5 solution_Q0 . . . 42
7.7.4.6 tmp_solution_Q0 . . . 42
7.7.4.7 pcout . . . 43
7.7.4.8 triangulation . . . 43
7.7.4.9 fe . . . 43
7.7.4.10 fe_Q0 . . . 43
7.7.4.11 lcell . . . 43
7.7.4.12 rcell . . . 43
7.7.4.13 bcell . . . 44
7.7.4.14 tcell . . . 44
7.7.4.15 fe_cell . . . 44
7.7.4.16 dh_cell . . . 44
7.7.4.17 shock_indicator . . . 44
7.7.4.18 jump_indicator . . . 44
7.7.4.19 mapping . . . 44
7.7.4.20 mapping_Q0 . . . 45
7.7.4.21 dof_handler . . . 45
7.7.4.22 dof_handler_Q0 . . . 45
7.7.4.23 dof_handlers . . . 45

7.7.4.24 dof_handlers_Q0 . . . 45
7.7.4.25 estimated_indicator_per_cell . . . 45
7.7.4.26 quadrature . . . 45
7.7.4.27 quadratures . . . 46
7.7.4.28 quadratures_Q0 . . . 46
7.7.4.29 cell_degree . . . 46
7.7.4.30 re_update . . . 46
7.7.4.31 cell_average . . . 46
7.7.4.32 timer . . . 46
7.7.4.33 euler_operator . . . 46
7.7.4.34 euler_operator_Q0 . . . 47
7.7.4.35 time . . . 47
7.7.4.36 time_step . . . 47
7.8 Euler_DG::EulerProblem< dim >::Postprocessor Class Reference . . . 47
7.8.1 Constructor & Destructor Documentation . . . 48
7.8.1.1 Postprocessor() . . . 48
7.8.2 Member Function Documentation . . . 48
7.8.2.1 evaluate_vector_field() . . . 48
7.8.2.2 get_names() . . . 48
7.8.2.3 get_data_component_interpretation() . . . 48
7.8.2.4 get_needed_update_flags() . . . 49
7.8.3 Member Data Documentation . . . 49
7.8.3.1 parameters . . . 49
7.8.3.2 a . . . 49
7.8.3.3 do_schlieren_plot . . . 49

8 File Documentation **51**

8.1 main.cpp File Reference . . . 51
8.1.1 Function Documentation . . . 52
8.1.1.1 main() . . . 52
8.2 parameters.h File Reference . . . 53
8.3 parameters.cpp File Reference . . . 54
8.4 equationdata.h File Reference . . . 54
8.5 equationdata.cpp File Reference . . . 55
8.6 operations.h File Reference . . . 56
8.7 euleroperator.h File Reference . . . 57
8.8 euleroperator.cpp File Reference . . . 59
8.9 eulerproblem.h File Reference . . . 59
8.10 eulerproblem.cpp File Reference . . . 60

Index **63**

Chapter 1

README

An example of a program that solves the Euler equations of fluid dynamics using an explicit time integrator with the matrix-free framework applied to a high-order discontinuous Galerkin discretizatoin in space.

In the file `main.cpp` we can select different test cases and running the progrma with the corresponding inputfile.

- backward facing step test case -> `inputfile_bstep.prm`
- forward facing step test case"-> `inputfile_fstep.prm`
- sod-Shock problem -> `inputfile_sod.prm`
- supersonic flow past a circular cylinder test -> `inputfile_cylinder.prm`
- double-mach reflection problem; -> `inputfile_DMR.prm`
- 2D Riemann problem -> `inputfile_2Driemann.prm`

In each input file there are parameters that we can change in order to simulate the testcase with different setting. We can change the type of limiter (TVB or filtering procedures), the limiter TVB function (minmod or vanAlbada), the refine (do refine or not do refine), the refinement indicator (gradient of density or gradient of pressure), the numerical flux (Lax, HLL, HLLC, Roe, SLAU)

Other files in the directory:

- `parameters *.h *.cpp`
- `equationData *.h *.cpp`
- `eulerproblem *.h *.cpp`
- `euleroperator *.h *.cpp`
- `operation.h`

In the file `refman.pdf` there is all the description of the code Eulero. The output results are stored in `../Eulero/build_Eulero/results`

To generate a Makefile for this code using Cmake, type the following command into the terminal from the main directory Eulero

```
mkdir build
cmake -S ../Eulero -B ../Eulero/build_Eulero
cd build_Eulero
cmake --build_Eulero .
./main "../inputfile_***.prm"
```

For this last line, change the name of the `inputfile_***.prm`, choosing the name described before.

Chapter 2

Namespace Index

2.1 Namespace List

Here is a list of all namespaces with brief descriptions:

EquationData 11
Euler_DG 11
Parameters 15

Chapter 3

Hierarchical Index

3.1 Class Hierarchy

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

DataPostprocessor
  Euler_DG::EulerProblem< dim >::Postprocessor 47
Euler_DG::EulerOperator< dim, degree, n_points_1d > 27
Euler_DG::EulerOperator< 2, 0, 1 > 27
Euler_DG::EulerOperator< 2, 2, 5 > 27
Euler_DG::EulerProblem< dim > 36
Function
  EquationData::BoundaryData< dim > 20
  EquationData::ExactSolution< dim > 17
  EquationData::InitialData< dim > 18
Parameters::Solver 26
  Parameters::Data_Storage 21

Chapter 4

Class Index

4.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

EquationData::BoundaryData< dim > 20
Parameters::Data_Storage 21
Euler_DG::EulerOperator< dim, degree, n_points_1d > 27
Euler_DG::EulerProblem< dim > 36
EquationData::ExactSolution< dim > 17
EquationData::InitialData< dim > 18
Euler_DG::EulerProblem< dim >::Postprocessor 47
Parameters::Solver 26

Chapter 5

File Index

5.1 File List

Here is a list of all files with brief descriptions:

equationdata.cpp 55
equationdata.h 54
euleroperator.cpp 59
euleroperator.h 57
eulerproblem.cpp 60
eulerproblem.h 59
main.cpp 51
operations.h 56
parameters.cpp 54
parameters.h 53

Chapter 6

Namespace Documentation

6.1 EquationData Namespace Reference

Classes

- class BoundaryData
- class ExactSolution
- class InitialData

6.2 Euler_DG Namespace Reference

Classes

- class EulerOperator
- class EulerProblem

Typedefs

- using Number = double

Functions

- template<typename ITERATOR >
  unsigned int cell_number (const ITERATOR &cell)
- double minmod (const double &a, const double &b, const double &c, const double &Mdx2)
- double vanAlbada (const double &c, const double &a, const double &b, const double &Mdx2, const double &eps)
- template<int dim, typename Number >
  DEAL_II_ALWAYS_INLINE Tensor< 1, dim, Number > euler_velocity (const Tensor< 1, dim+2, Number > &conserved_variables)
- template<int dim, typename Number >
  DEAL_II_ALWAYS_INLINE Number euler_pressure (const Tensor< 1, dim+2, Number > &conserved_↩
  variables, Parameters::Data_Storage &par)

- template<int dim, typename Number >
  DEAL_II_ALWAYS_INLINE Tensor< 1, dim+2, Tensor< 1, dim, Number > > euler_flux (const Tensor< 1, dim+2, Number > &conserved_variables, Parameters::Data_Storage &par)
- template<int n_components, int dim, typename Number >
  DEAL_II_ALWAYS_INLINE Tensor< 1, n_components, Number > operator* (const Tensor< 1, n_components, Tensor< 1, dim, Number >> &matrix, const Tensor< 1, dim, Number > &vector)
- template<typename Number >
  DEAL_II_ALWAYS_INLINE Number sgn (Number &val)
- template<int dim, typename Number >
  DEAL_II_ALWAYS_INLINE Tensor< 1, dim+2, Number > euler_numerical_flux (const Tensor< 1, dim+2, Number > &u_m, const Tensor< 1, dim+2, Number > &u_p, const Tensor< 1, dim, Number > &normal, Parameters::Data_Storage &par)
- template<int dim, typename Number >
  VectorizedArray< Number > evaluate_function (const Function< dim > &function, const Point< dim, VectorizedArray< Number >> &p_vectorized, const unsigned int component)
- template<int dim, typename Number , int n_components = dim + 2>
  Tensor< 1, n_components, VectorizedArray< Number > > evaluate_function (const Function< dim > &function, const Point< dim, VectorizedArray< Number >> &p_vectorized)

Variables

- constexpr unsigned int dimension = 2

6.2.1 Typedef Documentation

6.2.1.1 Number

```
typedef double Euler_DG::Number
```

6.2.2 Function Documentation

6.2.2.1 cell_number()

```
template<typename ITERATOR >
unsigned int Euler_DG::cell_number (
            const ITERATOR & cell )
```

given a cell iterator, return the cell numbe of a given cell

6.2.2.2 minmod()

```
double Euler_DG::minmod (
            const double & a,
            const double & b,
            const double & c,
            const double & Mdx2 )
```

TVB version of minmod limiter. If Mdx2=0 then it is TVD limiter.

6.2.2.3 vanAlbada()

```
double Euler_DG::vanAlbada (
            const double & c,
            const double & a,
            const double & b,
            const double & Mdx2,
            const double & eps )
```

TVB version of van Albada limiter.

6.2.2.4 euler_velocity()

```
template<int dim, typename Number >
DEAL_II_ALWAYS_INLINE Tensor<1, dim, Number> Euler_DG::euler_velocity (
            const Tensor< 1, dim+2, Number > & conserved_variables )  [inline]
```

In the following functions, we implement the various problem-specific operators pertaining to the Euler equations. Each function acts on the vector of conserved variables $[\rho, \rho\mathbf{u}, E]$ that we hold in the solution vectors, and computes various derived quantities.

First out is the computation of the velocity, that we derive from the momentum variable $\rho\mathbf{u}$ by division by ρ. One thing to note here is that we decorate all those functions with the keyword DEAL_II_ALWAYS_INLINE.

6.2.2.5 euler_pressure()

```
template<int dim, typename Number >
DEAL_II_ALWAYS_INLINE Number Euler_DG::euler_pressure (
            const Tensor< 1, dim+2, Number > & conserved_variables,
            Parameters::Data_Storage & par )  [inline]
```

The next function computes the pressure from the vector of conserved variables, using the formula $p = (\gamma - 1)\left(E - \frac{1}{2}\rho\mathbf{u}\cdot\mathbf{u}\right)$.

6.2.2.6 euler_flux()

```
template<int dim, typename Number >
DEAL_II_ALWAYS_INLINE Tensor<1, dim + 2, Tensor<1, dim, Number> > Euler_DG::euler_flux (
            const Tensor< 1, dim+2, Number > & conserved_variables,
            Parameters::Data_Storage & par )  [inline]
```

Here is the definition of the Euler flux function, i.e., the definition of the actual equation. Given the velocity and pressure (that the compiler optimization will make sure are done only once), this is straight-forward given the equation stated in the introduction.

6.2.2.7 operator∗()

```
template<int n_components, int dim, typename Number >
DEAL_II_ALWAYS_INLINE Tensor<1, n_components, Number> Euler_DG::operator* (
            const Tensor< 1, n_components, Tensor< 1, dim, Number >> & matrix,
            const Tensor< 1, dim, Number > & vector )  [inline]
```

This next function is a helper to simplify the implementation of the numerical flux, implementing the action of a tensor of tensors (with non-standard outer dimension of size dim + 2, so the standard overloads provided by deal.II's tensor classes do not apply here) with another tensor of the same inner dimension, i.e., a matrix-vector product.

6.2.2.8 sgn()

```
template<typename Number >
DEAL_II_ALWAYS_INLINE Number Euler_DG::sgn (
            Number & val )  [inline]
```

Function that returns the sign of a given value.

6.2.2.9 euler_numerical_flux()

```
template<int dim, typename Number >
DEAL_II_ALWAYS_INLINE Tensor<1, dim + 2, Number> Euler_DG::euler_numerical_flux (
            const Tensor< 1, dim+2, Number > & u_m,
            const Tensor< 1, dim+2, Number > & u_p,
            const Tensor< 1, dim, Number > & normal,
            Parameters::Data_Storage & par )  [inline]
```

This function implements the numerical flux (Riemann solver). It gets the state from the two sides of an interface and the normal vector, oriented from the side of the solution $\mathbf{w}^-$ towards the solution $\mathbf{w}^+$. In this and the following functions, we use variable suffixes _m and _p to indicate quantities derived from $\mathbf{w}^-$ and $\mathbf{w}^+$, i.e., values "here" and "there" relative to the current cell when looking at a neighbor cell.

6.2.2.10 evaluate_function() [1/2]

```
template<int dim, typename Number >
VectorizedArray<Number> Euler_DG::evaluate_function (
            const Function< dim > & function,
            const Point< dim, VectorizedArray< Number >> & p_vectorized,
            const unsigned int component )
```

This and the next function are helper functions to provide compact evaluation calls as multiple points get batched together via a VectorizedArray argument. This function is used for the subsonic outflow boundary conditions where we need to set the energy component to a prescribed value. The next one requests the solution on all components and is used for inflow boundaries where all components of the solution are set.

6.2.2.11 evaluate_function() [2/2]

```
template<int dim, typename Number , int n_components = dim + 2>
Tensor<1, n_components, VectorizedArray<Number> > Euler_DG::evaluate_function (
            const Function< dim > & function,
            const Point< dim, VectorizedArray< Number >> & p_vectorized )
```

6.2.3 Variable Documentation

6.2.3.1 dimension

```
constexpr unsigned int Euler_DG::dimension = 2  [constexpr]
```

6.3 Parameters Namespace Reference

Classes

- struct Data_Storage
- struct Solver

6.3.1 Detailed Description

We set up a class that holds all parameters that control the execution of the program.

Chapter 7

Class Documentation

7.1 EquationData::ExactSolution< dim > Class Template Reference

`#include <equationdata.h>`

Inheritance diagram for EquationData::ExactSolution< dim >:

Collaboration diagram for EquationData::ExactSolution< dim >:

Public Member Functions

- ExactSolution (const double time, Parameters::Data_Storage &parameters_in)
  constructor of ExactSolution
- virtual double value (const dealii::Point< dim > &p, const unsigned int component=0) const override

Private Attributes

- Parameters::Data_Storage & parameters

7.1.1 Detailed Description

template<int dim>
class EquationData::ExactSolution< dim >

We now define a class with the exact solution for the test case 4. We define an analytic solution of Sod-Shock tube problem that we try to reproduce with our discretization

7.1.2 Constructor & Destructor Documentation

7.1.2.1 ExactSolution()

```
template<int dim>
EquationData::ExactSolution< dim >::ExactSolution (
            const double time,
            Parameters::Data_Storage & parameters_in )
```

constructor of ExactSolution

7.1.3 Member Function Documentation

7.1.3.1 value()

```
template<int dim>
double EquationData::ExactSolution< dim >::value (
            const dealii::Point< dim > & p,
            const unsigned int component = 0 ) const  [override], [virtual]
```

7.1.4 Member Data Documentation

7.1.4.1 parameters

```
template<int dim>
Parameters::Data_Storage& EquationData::ExactSolution< dim >::parameters  [private]
```

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

- equationdata.h
- equationdata.cpp

7.2 EquationData::InitialData< dim > Class Template Reference

```
#include <equationdata.h>
```

Inheritance diagram for EquationData::InitialData< dim >:

Collaboration diagram for EquationData::InitialData< dim >:

Public Member Functions

- InitialData (Parameters::Data_Storage &parameters_in)

  constructor of InitialData
- virtual double value (const dealii::Point< dim > &p, const unsigned int component=0) const override

Private Attributes

- Parameters::Data_Storage & parameters

7.2.1 Detailed Description

template<int dim>
class EquationData::InitialData< dim >

We define a class for the initial condition. Given that the Euler equations are a problem with $d+2$ equations in d dimensions, we need to tell the Function base class about the correct number of components.

7.2.2 Constructor & Destructor Documentation

7.2.2.1 InitialData()

```
template<int dim>
EquationData::InitialData< dim >::InitialData (
            Parameters::Data_Storage & parameters_in )
```

constructor of InitialData

7.2.3 Member Function Documentation

7.2.3.1 value()

```
template<int dim>
double EquationData::InitialData< dim >::value (
            const dealii::Point< dim > & p,
            const unsigned int component = 0 ) const  [override], [virtual]
```

7.2.4 Member Data Documentation

7.2.4.1 parameters

```
template<int dim>
Parameters::Data_Storage& EquationData::InitialData< dim >::parameters  [private]
```

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

- equationdata.h
- equationdata.cpp

7.3 EquationData::BoundaryData< dim > Class Template Reference

`#include <equationdata.h>`

Inheritance diagram for EquationData::BoundaryData< dim >:

Collaboration diagram for EquationData::BoundaryData< dim >:

Public Member Functions

- BoundaryData (const double time, Parameters::Data_Storage &parameters_in, const unsigned int a)
  constructor of BoundaryData
- virtual double value (const dealii::Point< dim > &p, const unsigned int component=0) const override

Public Attributes

- unsigned int a

Private Attributes

- Parameters::Data_Storage & parameters

7.3.1 Detailed Description

template<int dim>
class EquationData::BoundaryData< dim >

We define a class for the boundary condition. Given that the Euler equations are a problem with $d+2$ equations in d dimensions, we need to tell the Function base class about the correct number of components.

7.3.2 Constructor & Destructor Documentation

7.3.2.1 BoundaryData()

```
template<int dim>
EquationData::BoundaryData< dim >::BoundaryData (
            const double time,
            Parameters::Data_Storage & parameters_in,
            const unsigned int a )
```

constructor of BoundaryData

7.3.3 Member Function Documentation

7.3.3.1 value()

```
template<int dim>
double EquationData::BoundaryData< dim >::value (
            const dealii::Point< dim > & p,
            const unsigned int component = 0 ) const  [override], [virtual]
```

7.3.4 Member Data Documentation

7.3.4.1 a

```
template<int dim>
unsigned int EquationData::BoundaryData< dim >::a
```

Integer used to select the corrisponding boundary condition

7.3.4.2 parameters

```
template<int dim>
Parameters::Data_Storage& EquationData::BoundaryData< dim >::parameters  [private]
```

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

- equationdata.h
- equationdata.cpp

7.4 Parameters::Data_Storage Struct Reference

```
#include <parameters.h>
```

Inheritance diagram for Parameters::Data_Storage:

Collaboration diagram for Parameters::Data_Storage:

Public Member Functions

- Data_Storage ()
- void read_data (const std::string &filename)

Public Attributes

- int testcase
- int n_stages
- int fe_degree
- int fe_degree_Q0
- int n_q_points_1d
- int n_q_points_1d_Q0
- int max_loc_refinements
- int min_loc_refinements
- double gamma
- double beta_density
- double beta_momentum
- double beta_energy
- double beta
- double M
- bool positivity
- std::string function_limiter
- std::string type
- bool refine
- std::string refine_indicator
- double final_time
- double output_tick

Protected Attributes

- ParameterHandler prm

Additional Inherited Members

7.4.1 Detailed Description

We collect all parameters that control the execution of the program. Besides the dimension and polynomial degree we want to run with, we also specify a number of points in the Gaussian quadrature formula we want to use for the nonlinear terms in the Euler equations. Furthermore, we specify parameters for the limiter problem (TVB and filter). We specify the parameters for the h-refinements. Depending on the test case, we also change the final time up to which we run the simulation, and a variable `output_tick` that specifies in which intervals we want to write output

7.4.2 Constructor & Destructor Documentation

7.4.2.1 Data_Storage()

```
Parameters::Data_Storage::Data_Storage ( )
```

7.4.3 Member Function Documentation

7.4.3.1 read_data()

```
void Parameters::Data_Storage::read_data (
            const std::string & filename )
```

Function to read all declared parameters

7.4.4 Member Data Documentation

7.4.4.1 testcase

```
int Parameters::Data_Storage::testcase
```

Number of specific testcase. Choices are : 1) Channel with hole; 2) backward facing step 3) forward facing step 4) sod-Shock problem; 5) //supersonic flow past a circular cylinder test; 6) double-mach reflection problem; 7) 2D Riemann problem

7.4.4.2 n_stages

```
int Parameters::Data_Storage::n_stages
```

number of stages in SSP runge kutta

7.4.4.3 fe_degree

```
int Parameters::Data_Storage::fe_degree
```

polynomial degree

7.4.4.4 fe_degree_Q0

```
int Parameters::Data_Storage::fe_degree_Q0
```

polynomial degree for Q0 solution

7.4.4.5 n_q_points_1d

```
int Parameters::Data_Storage::n_q_points_1d
```

number of points in the Gaussian quadrature formula

7.4.4.6 n_q_points_1d_Q0

```
int Parameters::Data_Storage::n_q_points_1d_Q0
```

number of points in the Gaussian quadrature formula when polynomial degree is zero

7.4.4.7 max_loc_refinements

```
int Parameters::Data_Storage::max_loc_refinements
```

number of maximum local refinements

7.4.4.8 min_loc_refinements

```
int Parameters::Data_Storage::min_loc_refinements
```

number of minimum local refinements

7.4.4.9 gamma

```
double Parameters::Data_Storage::gamma
```

adiabatic constant dependent on the type of the gas

7.4.4.10 beta_density

```
double Parameters::Data_Storage::beta_density
```

tolerance for density component used in filter technique

7.4.4.11 beta_momentum

```
double Parameters::Data_Storage::beta_momentum
```

tolerance for momentum component used in filter technique

7.4.4.12 beta_energy

```
double Parameters::Data_Storage::beta_energy
```

tolerance for energy component used in filter technique

7.4.4.13 beta

```
double Parameters::Data_Storage::beta
```

TVB limiter parameter

7.4.4.14 M

```
double Parameters::Data_Storage::M
```

TVB parameter

7.4.4.15 positivity

```
bool Parameters::Data_Storage::positivity
```

whether to use positivity limiter

7.4.4.16 function_limiter

```
std::string Parameters::Data_Storage::function_limiter
```

type of slope limiter function : minmod function or van Albada function

7.4.4.17 type

```
std::string Parameters::Data_Storage::type
```

type of limiter : filter technique or TVB limiter

7.4.4.18 refine

```
bool Parameters::Data_Storage::refine
```

true do refine, false not do refine

7.4.4.19 refine_indicator

```
std::string Parameters::Data_Storage::refine_indicator
```

do refine with density indicator or with pressure indicator

7.4.4.20 final_time

```
double Parameters::Data_Storage::final_time
```

The final time of the simulation

7.4.4.21 output_tick

```
double Parameters::Data_Storage::output_tick
```

This indicates between how many time steps we print the solution

7.4.4.22 prm

```
ParameterHandler Parameters::Data_Storage::prm  [protected]
```

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

- parameters.h
- parameters.cpp

7.5 Parameters::Solver Struct Reference

```
#include <parameters.h>
```

Inheritance diagram for Parameters::Solver:

Public Types

- enum EulerNumericalFlux {
  lax_friedrichs, harten_lax_vanleer, hllc_centered, HLLC,
  SLAU, roe }

Public Member Functions

- void parse_parameters (dealii::ParameterHandler &prm)

Static Public Member Functions

- static void declare_parameters (dealii::ParameterHandler &prm)

Public Attributes

- EulerNumericalFlux numerical_flux_type

7.5.1 Detailed Description

We select a detail of the spatial discretization, namely the numerical flux (Riemann solver) at the faces between cells. For this program, we have implemented a modified variant of the Lax–Friedrichs flux, the Harten–Lax–van Leer (HLL) flux, the HLLC flux, the Roe flux and the SLAU flux.

7.5.2 Member Enumeration Documentation

7.5.2.1 EulerNumericalFlux

```
enum Parameters::Solver::EulerNumericalFlux
```

Enumerator

lax_friedrichs	
harten_lax_vanleer	
hllc_centered	
HLLC	
SLAU	
roe	

7.5.3 Member Function Documentation

7.5.3.1 declare_parameters()

```
void Parameters::Solver::declare_parameters (
            dealii::ParameterHandler & prm )  [static]
```

7.5.3.2 parse_parameters()

```
void Parameters::Solver::parse_parameters (
            dealii::ParameterHandler & prm )
```

7.5.4 Member Data Documentation

7.5.4.1 numerical_flux_type

```
EulerNumericalFlux Parameters::Solver::numerical_flux_type
```

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

- parameters.h
- parameters.cpp

7.6 Euler_DG::EulerOperator< dim, degree, n_points_1d > Class Template Reference

```
#include <euleroperator.h>
```

Collaboration diagram for Euler_DG::EulerOperator< dim, degree, n_points_1d >:

Public Member Functions

- EulerOperator (dealii::TimerOutput &timer_output, Parameters::Data_Storage &parameters_in)

  constructor
- void reinit (const dealii::MappingQ1< dim > &mapping, std::vector< const DoFHandler< dim > * > &dof↩ _handler, const std::vector< QGauss< 1 >> &quadratures)
- void set_inflow_boundary (const dealii::types::boundary_id boundary_id, std::unique_ptr< dealii::Function< dim >> inflow_function)
- void set_subsonic_outflow_boundary (const dealii::types::boundary_id boundary_id, std::unique_ptr< dealii::Function< dim >> outflow_energy)
- void set_supersonic_outflow_boundary (const dealii::types::boundary_id boundary_id)
- void set_wall_boundary (const dealii::types::boundary_id boundary_id)
- void set_body_force (std::unique_ptr< dealii::Function< dim >> body_force)
- void apply (const double current_time, const dealii::LinearAlgebra::distributed::Vector< Number > &src, dealii::LinearAlgebra::distributed::Vector< Number > &dst) const
- void project (const dealii::Function< dim > &function, dealii::LinearAlgebra::distributed::Vector< Number > &solution) const
- std::array< double, 3 > compute_errors (const dealii::Function< dim > &function, const dealii::Linear↩ Algebra::distributed::Vector< Number > &solution) const
- double compute_cell_transport_speed (const dealii::LinearAlgebra::distributed::Vector< Number > &solution) const
- void initialize_vector (dealii::LinearAlgebra::distributed::Vector< Number > &vector) const

Static Public Attributes

- static constexpr unsigned int n_quadrature_points_1d = n_points_1d

Private Member Functions

- void local_apply_inverse_mass_matrix (const dealii::MatrixFree< dim, Number > &data, dealii::Linear↩ Algebra::distributed::Vector< Number > &dst, const dealii::LinearAlgebra::distributed::Vector< Number > &src, const std::pair< unsigned int, unsigned int > &cell_range) const
- void local_apply_cell (const dealii::MatrixFree< dim, Number > &data, dealii::LinearAlgebra::distributed::↩ Vector< Number > &dst, const dealii::LinearAlgebra::distributed::Vector< Number > &src, const std::pair< unsigned int, unsigned int > &cell_range) const
- void local_apply_face (const dealii::MatrixFree< dim, Number > &data, dealii::LinearAlgebra::distributed::↩ Vector< Number > &dst, const dealii::LinearAlgebra::distributed::Vector< Number > &src, const std::pair< unsigned int, unsigned int > &face_range) const
- void local_apply_boundary_face (const dealii::MatrixFree< dim, Number > &data, dealii::LinearAlgebra↩ ::distributed::Vector< Number > &dst, const dealii::LinearAlgebra::distributed::Vector< Number > &src, const std::pair< unsigned int, unsigned int > &face_range) const

Private Attributes

- Parameters::Data_Storage & parameters
- dealii::MatrixFree< dim, Number > data
- dealii::TimerOutput & timer
- std::map< dealii::types::boundary_id, std::unique_ptr< dealii::Function< dim > > > inflow_boundaries
- std::map< dealii::types::boundary_id, std::unique_ptr< dealii::Function< dim > > > subsonic_outflow_boundaries
- std::set< dealii::types::boundary_id > supersonic_outflow_boundaries
- std::set< dealii::types::boundary_id > wall_boundaries
- std::unique_ptr< dealii::Function< dim > > body_force

7.6.1 Detailed Description

template<int dim, int degree, int n_points_1d>
class Euler_DG::EulerOperator< dim, degree, n_points_1d >

This class implements the evaluators for the Euler problem. Since the present operator is non-linear and does not require a matrix interface (to be handed over to preconditioners), we only implement an `apply` function as well as the combination of `apply` with the required vector updates for the Runge–Kutta time integrator. Furthermore, we have added 3 additional functions involving matrix-free routines, namely one to compute an estimate of the time step scaling (that is combined with the Courant number for the actual time step size) based on the velocity and speed of sound in the elements, one for the projection of solutions (specializing VectorTools::project() for the DG case), and one to compute the errors against a possible analytical solution or norms against some background state.

We provide a few functions to allow a user to pass in various forms of boundary conditions on different parts of the domain boundary marked by types::boundary_id variables, as well as possible body forces.

7.6.2 Constructor & Destructor Documentation

7.6.2.1 EulerOperator()

```
template<int dim, int degree, int n_points_1d>
Euler_DG::EulerOperator< dim, degree, n_points_1d >::EulerOperator (
             dealii::TimerOutput & timer_output,
             Parameters::Data_Storage & parameters_in )
```

constructor

7.6.3 Member Function Documentation

7.6.3.1 reinit()

```
template<int dim, int degree, int n_points_1d>
void Euler_DG::EulerOperator< dim, degree, n_points_1d >::reinit (
             const dealii::MappingQ1< dim > & mapping,
             std::vector< const DoFHandler< dim > * > & dof_handlers,
             const std::vector< QGauss< 1 >> & quadratures )
```

For the initialization of the Euler operator, we set up the MatrixFree variable contained in the class. This can be done given a mapping to describe possible curved boundaries as well as a DoFHandler object describing the degrees of freedom. Since we use a discontinuous Galerkin discretization in this tutorial program where no constraints are imposed strongly on the solution field, we do not need to pass in an AffineConstraints object and rather use a dummy for the construction.

7.6.3.2 set_inflow_boundary()

```
template<int dim, int degree, int n_points_1d>
void Euler_DG::EulerOperator< dim, degree, n_points_1d >::set_inflow_boundary (
             const dealii::types::boundary_id boundary_id,
             std::unique_ptr< dealii::Function< dim >> inflow_function )
```

The subsequent member functions are the ones that must be called from outside to specify the various types of boundaries. For an inflow boundary, we must specify all components in terms of density ρ, momentum $\rho\mathbf{u}$ and energy E. Given this information, we then store the function alongside the respective boundary id in a map member variable of this class. For the present DG code where boundary conditions are solely applied as part of the weak form (during time integration), the call to set the boundary conditions can appear both before or after the `reinit()` call to this class.

The checks added in each of the four function are used to ensure that boundary conditions are mutually exclusive on the various parts of the boundary, i.e., that a user does not accidentally designate a boundary as both an inflow and say a subsonic outflow boundary.

7.6.3.3 set_subsonic_outflow_boundary()

```
template<int dim, int degree, int n_points_1d>
void Euler_DG::EulerOperator< dim, degree, n_points_1d >::set_subsonic_outflow_boundary (
             const dealii::types::boundary_id boundary_id,
             std::unique_ptr< dealii::Function< dim >> outflow_energy )
```

Likewise, we proceed for the subsonic outflow boundaries (where we request a function as well, which we use to retrieve the energy)

7.6.3.4 set_supersonic_outflow_boundary()

```
template<int dim, int degree, int n_points_1d>
void Euler_DG::EulerOperator< dim, degree, n_points_1d >::set_supersonic_outflow_boundary (
             const dealii::types::boundary_id boundary_id )
```

for the supersonic outflow condition we impose neumann condition

7.6.3.5 set_wall_boundary()

```
template<int dim, int degree, int n_points_1d>
void Euler_DG::EulerOperator< dim, degree, n_points_1d >::set_wall_boundary (
             const dealii::types::boundary_id boundary_id )
```

for wall (no-penetration) boundaries $*$ where we impose zero normal velocity.

7.6.3.6 set_body_force()

```
template<int dim, int degree, int n_points_1d>
void Euler_DG::EulerOperator< dim, degree, n_points_1d >::set_body_force (
             std::unique_ptr< dealii::Function< dim >> body_force )
```

7.6.3.7 apply()

```
template<int dim, int degree, int n_points_1d>
void Euler_DG::EulerOperator< dim, degree, n_points_1d >::apply (
             const double current_time,
             const dealii::LinearAlgebra::distributed::Vector< Number > & src,
             dealii::LinearAlgebra::distributed::Vector< Number > & dst ) const
```

We now come to the function which implements the evaluation of the Euler operator as a whole, i.e., $\mathcal{M}^{-1}\mathcal{L}(t,\mathbf{w})$, calling into the local evaluators presented above. The steps should be clear from the previous code. One thing to note is that we need to adjust the time in the functions we have associated with the various parts of the boundary, in order to be consistent with the equation in case the boundary data is time-dependent. Then, we call MatrixFree::loop() to perform the cell and face integrals, including the necessary ghost data exchange in the `src` vector. The seventh argument to the function, `true`, specifies that we want to zero the `dst` vector as part of the loop, before we start accumulating integrals into it. This variant is preferred over explicitly calling `dst = 0.;` before the loop as the zeroing operation is done on a subrange of the vector in parts that are written by the integrals nearby. This enhances data locality and allows for caching, saving one roundtrip of vector data to main memory and enhancing performance. The last two arguments to the loop determine which data is exchanged: Since we only access the values of the shape functions one faces, typical of first-order hyperbolic problems, and since we have a nodal basis with nodes at the reference element surface, we only need to exchange those parts. This again saves precious memory bandwidth. Once the spatial operator $\mathcal{L}$ is applied, we need to make a second round and apply the inverse mass matrix. Here, we call `MatrixFree::cell_loop()` since only cell integrals appear. The cell loop is cheaper than the full loop as access only goes to the degrees of freedom associated with the locally owned cells, which is simply the locally owned degrees of freedom for DG discretizations. Thus, no ghost exchange is needed here. Around all these functions, we put timer scopes to record the computational time for statistics about the contributions of the various parts.

7.6.3.8 project()

```
template<int dim, int degree, int n_points_1d>
void Euler_DG::EulerOperator< dim, degree, n_points_1d >::project (
             const dealii::Function< dim > & function,
             dealii::LinearAlgebra::distributed::Vector< Number > & solution ) const
```

Having discussed the implementation of the functions that deal with advancing the solution by one time step, let us now move to functions that implement other, ancillary operations. Specifically, these are functions that compute projections, evaluate errors, and compute the speed of information transport on a cell. The first of these functions is essentially equivalent to `VectorTools::project()`, just much faster because it is specialized for DG elements where there is no need to set up and solve a linear system, as each element has independent basis functions. The reason why we show the code here, besides a small speedup of this non-critical operation, is that it shows additional functionality provided by `MatrixFreeOperators::CellwiseInverseMassMatrix`. The projection operation works as follows: If we denote the matrix of shape functions evaluated at quadrature points by S, the projection on cell K is an operation of the form $\underbrace{SJ^KS^{\mathrm{T}}}_{\mathcal{M}^K}\mathbf{w}^K = SJ^K\tilde{\mathbf{w}}(\mathbf{x}_q)_{q=1:n_q}$, where J^K is the diagonal matrix containing the determinant of the Jacobian times the quadrature weight (JxW), $\mathcal{M}^K$ is the cell-wise mass matrix, and $\tilde{\mathbf{w}}(\mathbf{x}_q)_{q=1:n_q}$ is the evaluation of the field to be projected onto quadrature points. (In reality the matrix S has additional structure through the tensor product, as explained in the introduction.) This system can now equivalently be written as $\mathbf{w}^K = \left(SJ^KS^{\mathrm{T}}\right)^{-1}SJ^K\tilde{\mathbf{w}}(\mathbf{x}_q)_{q=1:n_q} = S^{-\mathrm{T}}\left(J^K\right)^{-1}S^{-1}SJ^K\tilde{\mathbf{w}}(\mathbf{x}_q)_{q=1:n_q}$. Now, the term $S^{-1}S$ and then $\left(J^K\right)^{-1}J^K$ cancel, resulting in the final expression $\mathbf{w}^K = S^{-\mathrm{T}}\tilde{\mathbf{w}}(\mathbf{x}_q)_{q=1:n_q}$. This operation is implemented by `MatrixFreeOperators::CellwiseInverseMassMatrix::transform_from_q_points_to_basis()`. The name is derived from the fact that this projection is simply the multiplication by $S^{-\mathrm{T}}$, a basis change from the nodal basis in the points of the Gaussian quadrature to the given finite element basis. Note that we call `FEEvaluation::set_dof_values()` to write the result into the vector, overwriting previous content, rather than accumulating the results as typical in integration tasks – we can do this because every vector entry has contributions from only a single cell for discontinuous Galerkin discretizations.

7.6.3.9 compute_errors()

```
template<int dim, int degree, int n_points_1d>
std::array< double, 3 > Euler_DG::EulerOperator< dim, degree, n_points_1d >::compute_errors (
            const dealii::Function< dim > & function,
            const dealii::LinearAlgebra::distributed::Vector< Number > & solution ) const
```

The next function again repeats functionality also provided by the deal.II library, namely `VectorTools::integrate_difference()`. We here show the explicit code to highlight how the vectorization across several cells works and how to accumulate results via that interface: Recall that each *lane* of the vectorized array holds data from a different cell.

7.6.3.10 compute_cell_transport_speed()

```
template<int dim, int degree, int n_points_1d>
double Euler_DG::EulerOperator< dim, degree, n_points_1d >::compute_cell_transport_speed (
            const dealii::LinearAlgebra::distributed::Vector< Number > & solution ) const
```

This final function of the EulerOperator class is used to estimate the transport speed, scaled by the mesh size, that is relevant for setting the time step size in the explicit time integrator. In the Euler equations, there are two speeds of transport, namely the convective velocity $\mathbf{u}$ and the propagation of sound waves with sound speed $c = \sqrt{\gamma p/\rho}$ relative to the medium moving at velocity $\mathbf{u}$. In the formula for the time step size, we are interested not by these absolute speeds, but by the amount of time it takes for information to cross a single cell. For information transported along with the medium, $\mathbf{u}$ is scaled by the mesh size, so an estimate of the maximal velocity can be obtained by computing $\|J^{-\mathrm{T}}\mathbf{u}\|_\infty$, where J is the Jacobian of the transformation from real to the reference domain. Note that FEEvaluationBase::inverse_jacobian() returns the inverse and transpose Jacobian, representing the metric term from real to reference coordinates, so we do not need to transpose it again. We store this limit in the variable `convective_limit` in the code below. The sound propagation is isotropic, so we need to take mesh sizes in any direction into account. The appropriate mesh size scaling is then given by the minimal singular value of J or, equivalently, the maximal singular value of J^{-1}. Note that one could approximate this quantity by the minimal distance between vertices of a cell when ignoring curved cells. The speed of convergence of this method depends on the ratio of the largest to the next largest eigenvalue and the initial guess, which is the vector of all ones.

7.6.3.11 initialize_vector()

```
template<int dim, int degree, int n_points_1d>
void Euler_DG::EulerOperator< dim, degree, n_points_1d >::initialize_vector (
            dealii::LinearAlgebra::distributed::Vector< Number > & vector ) const
```

7.6.3.12 local_apply_inverse_mass_matrix()

```
template<int dim, int degree, int n_points_1d>
void Euler_DG::EulerOperator< dim, degree, n_points_1d >::local_apply_inverse_mass_matrix (
            const dealii::MatrixFree< dim, Number > & data,
            dealii::LinearAlgebra::distributed::Vector< Number > & dst,
            const dealii::LinearAlgebra::distributed::Vector< Number > & src,
            const std::pair< unsigned int, unsigned int > & cell_range ) const  [private]
```

The next function implements the inverse mass matrix operation. It does similar operations as the forward evaluation of the mass matrix, except with a different interpolation matrix, representing the inverse S^{-1} factors. These represent a change of basis from the specified basis (in this case, the Lagrange basis in the points of the Gauss–Lobatto

quadrature formula) to the Lagrange basis in the points of the Gauss quadrature formula. In the latter basis, we can apply the inverse of the point-wise JxW factor, i.e., the quadrature weight times the determinant of the Jacobian of the mapping from reference to real coordinates. Once this is done, the basis is changed back to the nodal Gauss-Lobatto basis again. All of these operations are done by the apply() function below. What we need to provide is the local fields to operate on (which we extract from the global vector by an FEEvaluation object) and write the results back to the destination vector of the mass matrix operation. One thing to note is that we added two integer arguments (that are optional) to the constructor of FEEvaluation, the first being 0 (selecting among the DoFHandler in multi-DoFHandler systems; here, we only have one) and the second being 1 to make the quadrature formula selection. As we use the quadrature formula 0 for the over-integration of nonlinear terms, we use the formula 1 with the default $p+1$ (or fe_degree+1 in terms of the variable name) points for the mass matrix. This leads to square contributions to the mass matrix and ensures exact integration, as explained in the introduction.

7.6.3.13 local_apply_cell()

```
template<int dim, int degree, int n_points_1d>
void Euler_DG::EulerOperator< dim, degree, n_points_1d >::local_apply_cell (
             const dealii::MatrixFree< dim, Number > & data,
             dealii::LinearAlgebra::distributed::Vector< Number > & dst,
             const dealii::LinearAlgebra::distributed::Vector< Number > & src,
             const std::pair< unsigned int, unsigned int > & cell_range ) const  [private]
```

Now we proceed to the local evaluators for the Euler problem. We use an FEEvaluation with a non-standard number of quadrature points. Whereas we previously always set the number of quadrature points to equal the polynomial degree plus one, we now set the number quadrature points as a separate variable (e.g. the polynomial degree plus two) to more accurately handle nonlinear terms. Since the evaluator is fed with the appropriate loop lengths via the template argument and keeps the number of quadrature points in the whole cell in the variable FEEvaluation::n_↩ q_points, we now automatically operate on the more accurate formula without further changes.

We are evaluating a multi-component system. The matrix-free framework provides several ways to handle the multi-component case. Here we use an FEEvaluation object with multiple components embedded into it, specified by the fourth template argument dim + 2 for the components in the Euler system. As a consequence, the return type of FEEvaluation::get_value() is not a scalar any more (that would return a VectorizedArray type, collecting data from several elements), but a Tensor of dim+2 components. The functionality is otherwise similar to the scalar case; it is handled by a template specialization of a base class, called FEEvaluationAccess. An alternative variant would have been to use several FEEvaluation objects, a scalar one for the density, a vector-valued one with dim components for the momentum, and another scalar evaluator for the energy. To ensure that those components point to the correct part of the solution, the constructor of FEEvaluation takes three optional integer arguments after the required MatrixFree field, namely the number of the DoFHandler for multi-DoFHandler systems (taking the first by default), the number of the quadrature point in case there are multiple Quadrature objects (see more below), and as a third argument the component within a vector system. As we have a single vector for all components, we would go with the third argument, and set it to 0 for the density, 1 for the vector-valued momentum, and dim+1 for the energy slot. FEEvaluation then picks the appropriate subrange of the solution vector during FEEvaluationBase::read↩ _dof_values() and FEEvaluation::distributed_local_to_global() or the more compact F↩ EEvaluation::gather_evaluate() and FEEvaluation::integrate_scatter() calls.

When it comes to the evaluation of the body force vector, we distinguish between two cases for efficiency reasons: In case we have a constant function (derived from Functions::ConstantFunction), we can precompute the value outside the loop over quadrature points and simply use the value everywhere. For a more general function, we instead need to call the evaluate_function() method we provided above; this path is more expensive because we need to access the memory associated with the quadrature point data.

Since we have implemented all physics for the Euler equations in the separate euler_flux() function, all we have to do here is to call this function given the current solution evaluated at quadrature points, returned by phi.↩ get_value(q), and tell the FEEvaluation object to queue the flux for testing it by the gradients of the shape functions (which is a Tensor of outer dim+2 components, each holding a tensor of dim components for the x,y,z component of the Euler flux). One final thing worth mentioning is the order in which we queue the data for testing by the value of the test function, phi.submit_value(), in case we are given an external function: We must do

this after calling `phi.get_value(q)`, because `get_value()` (reading the solution) and `submit_value()` (queuing the value for multiplication by the test function and summation over quadrature points) access the same underlying data field. Here it would be easy to achieve also without temporary variable `w_q` since there is no mixing between values and gradients. For more complicated setups, one has to first copy out e.g. both the value and gradient at a quadrature point and then queue results again by `FEEvaluationBase::submit_value()` and `FEEvaluationBase::submit_gradient()`.

As a final note, we mention that we do not use the first MatrixFree argument of this function, which is a call-back from MatrixFree::loop(). The interfaces imposes the present list of arguments, but since we are in a member function where the MatrixFree object is already available as the `data` variable, we stick with that to avoid confusion.

7.6.3.14 local_apply_face()

```
template<int dim, int degree, int n_points_1d>
void Euler_DG::EulerOperator< dim, degree, n_points_1d >::local_apply_face (
             const dealii::MatrixFree< dim, Number > & data,
             dealii::LinearAlgebra::distributed::Vector< Number > & dst,
             const dealii::LinearAlgebra::distributed::Vector< Number > & src,
             const std::pair< unsigned int, unsigned int > & face_range ) const  [private]
```

The next function concerns the computation of integrals on interior faces, where we need evaluators from both cells adjacent to the face. We associate the variable `phi_m` with the solution component $\mathbf{w}^-$ and the variable `phi_p` with the solution component $\mathbf{w}^+$. We distinguish the two sides in the constructor of FEFaceEvaluation by the second argument, with `true` for the interior side and `false` for the exterior side, with interior and exterior denoting the orientation with respect to the normal vector. Note that the calls `FEFaceEvaluation::gather_evaluate()` and `FEFaceEvaluation::integrate_scatter()` combine the access to the vectors and the sum factorization parts. This combined operation not only saves a line of code, but also contains an important optimization: Given that we use a nodal basis in terms of the Lagrange polynomials in the points of the Gauss-Lobatto quadrature formula, only $(p+1)^{d-1}$ out of the $(p+1)^d$ basis functions evaluate to non-zero on each face. Thus, the evaluator only accesses the necessary data in the vector and skips the parts which are multiplied by zero. If we had first read the vector, we would have needed to load all data from the vector, as the call in isolation would not know what data is required in subsequent operations. If the subsequent FEFaceEvaluation::evaluate() call requests values and derivatives, indeed all $(p+1)^d$ vector entries for each component are needed, as the normal derivative is nonzero for all basis functions. The arguments to the evaluators as well as the procedure is similar to the cell evaluation. We again use the more accurate (over-)integration scheme. At the quadrature points, we then go to our free-standing function for the numerical flux. It receives the solution evaluated at quadrature points from both sides (i.e., $\mathbf{w}^-$ and $\mathbf{w}^+$), as well as the normal vector onto the minus side. As explained above, the numerical flux is already multiplied by the normal vector from the minus side. We need to switch the sign because the boundary term comes with a minus sign in the weak form derived in the introduction. The flux is then queued for testing both on the minus sign and on the plus sign, with switched sign as the normal vector from the plus side is exactly opposed to the one from the minus side.

7.6.3.15 local_apply_boundary_face()

```
template<int dim, int degree, int n_points_1d>
void Euler_DG::EulerOperator< dim, degree, n_points_1d >::local_apply_boundary_face (
             const dealii::MatrixFree< dim, Number > & data,
             dealii::LinearAlgebra::distributed::Vector< Number > & dst,
             const dealii::LinearAlgebra::distributed::Vector< Number > & src,
             const std::pair< unsigned int, unsigned int > & face_range ) const  [private]
```

For faces located at the boundary, we need to impose the appropriate boundary conditions. In this tutorial program, we implement fifth cases as mentioned above. The discontinuous Galerkin method imposes boundary conditions not as constraints, but only weakly. Thus, the various conditions are imposed by finding an appropriate *exterior* quantity $\mathbf{w}^+$ that is then handed to the numerical flux function also used for the interior faces. In essence, we

"pretend" a state on the outside of the domain in such a way that if that were reality, the solution of the PDE would satisfy the boundary conditions we want. For wall boundaries, we need to impose a no-normal-flux condition on the momentum variable, whereas we use a Neumann condition for the density and energy with $\rho^+ = \rho^-$ and $E^+ = E^-$. To achieve the no-normal flux condition, we set the exterior values to the interior values and subtract two times the velocity in wall-normal direction, i.e., in the direction of the normal vector. For inflow boundaries, we simply set the given Dirichlet data $\mathbf{w}_\mathrm{D}$ as a boundary value. An alternative would have been to use $\mathbf{w}^+ = -\mathbf{w}^- + 2\mathbf{w}_\mathrm{D}$, the so-called mirror principle. The imposition of outflow is essentially a Neumann condition, i.e., setting $\mathbf{w}^+ = \mathbf{w}^-$. In the implementation below, we check for the various types of boundaries at the level of quadrature points. Of course, we could also have moved the decision out of the quadrature point loop and treat entire faces as of the same kind, which avoids some map/set lookups in the inner loop over quadrature points. However, the loss of efficiency is hardly noticeable, so we opt for the simpler code here. Also note that the final `else` clause will catch the case when some part of the boundary was not assigned any boundary condition via `EulerOperator::set_...↩ _boundary(...)`.

7.6.4 Member Data Documentation

7.6.4.1 n_quadrature_points_1d

```
template<int dim, int degree, int n_points_1d>
constexpr unsigned int Euler_DG::EulerOperator< dim, degree, n_points_1d >::n_quadrature_↩
points_1d = n_points_1d  [static], [constexpr]
```

7.6.4.2 parameters

```
template<int dim, int degree, int n_points_1d>
Parameters::Data_Storage& Euler_DG::EulerOperator< dim, degree, n_points_1d >::parameters
[private]
```

7.6.4.3 data

```
template<int dim, int degree, int n_points_1d>
dealii::MatrixFree<dim, Number> Euler_DG::EulerOperator< dim, degree, n_points_1d >::data
[private]
```

7.6.4.4 timer

```
template<int dim, int degree, int n_points_1d>
dealii::TimerOutput& Euler_DG::EulerOperator< dim, degree, n_points_1d >::timer  [private]
```

7.6.4.5 inflow_boundaries

```
template<int dim, int degree, int n_points_1d>
std::map<dealii::types::boundary_id, std::unique_ptr<dealii::Function<dim> > > Euler_DG::EulerOperator<
dim, degree, n_points_1d >::inflow_boundaries  [private]
```

7.6.4.6 subsonic_outflow_boundaries

```
template<int dim, int degree, int n_points_1d>
std::map<dealii::types::boundary_id, std::unique_ptr<dealii::Function<dim> > > Euler_DG::EulerOperator<
dim, degree, n_points_1d >::subsonic_outflow_boundaries  [private]
```

7.6.4.7 supersonic_outflow_boundaries

```
template<int dim, int degree, int n_points_1d>
std::set<dealii::types::boundary_id> Euler_DG::EulerOperator< dim, degree, n_points_1d >↩
::supersonic_outflow_boundaries  [private]
```

7.6.4.8 wall_boundaries

```
template<int dim, int degree, int n_points_1d>
std::set<dealii::types::boundary_id> Euler_DG::EulerOperator< dim, degree, n_points_1d >↩
::wall_boundaries  [private]
```

7.6.4.9 body_force

```
template<int dim, int degree, int n_points_1d>
std::unique_ptr<dealii::Function<dim> > Euler_DG::EulerOperator< dim, degree, n_points_1d
>::body_force  [private]
```

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

- euleroperator.h
- euleroperator.cpp

7.7 Euler_DG::EulerProblem< dim > Class Template Reference

```
#include <eulerproblem.h>
```

Collaboration diagram for Euler_DG::EulerProblem< dim >:

Classes

- class Postprocessor

Public Member Functions

- EulerProblem (Parameters::Data_Storage &parameters_in)
- void run ()

Private Member Functions

- void make_grid ()
- void adapt_mesh ()
- void make_dofs ()
- void compute_cell_average (LinearAlgebra::distributed::Vector< double > &current_solution)
- void get_cell_average (const typename dealii::DoFHandler< dim >::cell_iterator &cell, dealii::Vector< double > &avg)
- void compute_shock_indicator (LinearAlgebra::distributed::Vector< double > &current_solution)
- LinearAlgebra::distributed::Vector< Number > apply_limiter_TVB (LinearAlgebra::distributed::Vector< Number > &solution)
- LinearAlgebra::distributed::Vector< Number > apply_filter (LinearAlgebra::distributed::Vector< Number > &sol_H, LinearAlgebra::distributed::Vector< Number > &sol_M)
- void update (const double current_time, const double time_step, LinearAlgebra::distributed::Vector< Number > &solution_np, LinearAlgebra::distributed::Vector< Number > &tmp_sol_n, LinearAlgebra::distributed::↩ Vector< Number > &solution_np_Q0, LinearAlgebra::distributed::Vector< Number > &tmp_sol_n_Q0)
- void output_results (const unsigned int result_number)

Private Attributes

- Parameters::Data_Storage & parameters
- LinearAlgebra::distributed::Vector< Number > solution
- LinearAlgebra::distributed::Vector< Number > tmp_solution
- LinearAlgebra::distributed::Vector< Number > sol_aux
- LinearAlgebra::distributed::Vector< Number > solution_Q0
- LinearAlgebra::distributed::Vector< Number > tmp_solution_Q0
- ConditionalOStream pcout
- Triangulation< dim > triangulation
- FESystem< dim > fe
- FESystem< dim > fe_Q0
- std::vector< typename dealii::DoFHandler< dim >::cell_iterator > lcell
- std::vector< typename dealii::DoFHandler< dim >::cell_iterator > rcell
- std::vector< typename dealii::DoFHandler< dim >::cell_iterator > bcell
- std::vector< typename dealii::DoFHandler< dim >::cell_iterator > tcell
- const dealii::FE_DGQ< dim > fe_cell
- dealii::DoFHandler< dim > dh_cell
- Vector< double > shock_indicator
- Vector< double > jump_indicator
- const MappingQ1< dim > mapping
- const MappingQ1< dim > mapping_Q0
- DoFHandler< dim > dof_handler
- DoFHandler< dim > dof_handler_Q0
- std::vector< const DoFHandler< dim > * > dof_handlers

- std::vector< const DoFHandler< dim > * > dof_handlers_Q0
- Vector< double > estimated_indicator_per_cell
- const QGauss< dim > quadrature
- std::vector< QGauss< 1 > > quadratures
- std::vector< QGauss< 1 > > quadratures_Q0
- std::vector< unsigned int > cell_degree
- std::vector< bool > re_update
- std::vector< dealii::Vector< double > > cell_average
- TimerOutput timer
- EulerOperator< 2, 2, 5 > euler_operator
- EulerOperator< 2, 0, 1 > euler_operator_Q0
- double time
- double time_step

7.7.1 Detailed Description

template<int dim>
class Euler_DG::EulerProblem< dim >

This class combines the EulerOperator class with the time integrator and the usual global data structures such as FiniteElement and DoFHandler, to actually run the simulations of the Euler problem.

The member variables are a triangulation, a finite element, a mapping, and a DoFHandler to describe the degrees of freedom. In this class, we implemente the member functions usefull for refinement and compute limiter as TVB or filter limiter. In the update funcion we implement all the stages for the SSP Runge-Kutta in which at each stage we do the limiter of the solution. In addition, we keep an instance of the EulerOperator described above around, which will do all heavy lifting in terms of integrals, and some parameters for time integration like the current time or the time step size. Furthermore, we use a PostProcessor instance to write some additional information to the output file. The interface of the DataPostprocessor class is intuitive, requiring us to provide information about what needs to be evaluated (typically only the values of the solution, except for the Schlieren plot that we only enable in 2D where it makes sense), and the names of what gets evaluated. Note that it would also be possible to extract most information by calculator tools within visualization programs such as ParaView, but it is so much more convenient to do it already when writing the output.

7.7.2 Constructor & Destructor Documentation

7.7.2.1 EulerProblem()

```
template<int dim>
Euler_DG::EulerProblem< dim >::EulerProblem (
            Parameters::Data_Storage & parameters_in )
```

The constructor for this class is unsurprising: We set up a parallel triangulation based on the `MPI_COMM_WORLD` communicator, a vector finite element with `dim+2` components for density, momentum, and energy, a high-order mapping of the same degree as the underlying finite element, initialize the time and time step to zero, and finite element usefull for the TVB limiter.

7.7.3 Member Function Documentation

7.7.3.1 run()

```
template<int dim>
void Euler_DG::EulerProblem< dim >::run
```

The EulerProblem::run() function puts all pieces together. It starts off by calling the function that creates the mesh and sets up data structures, Before we start the time loop, we compute the time step size by the `EulerOperator::compute_cell_transport_speed()` function. For reasons of comparison, we compare the result obtained there with the minimal mesh size and print them to screen. For velocities and speeds of sound close to unity as in this tutorial program, the predicted effective mesh size will be close, but they could vary if scaling were different.

Now we are ready to start the time loop, which we run until the time has reached the desired end time. Every 5 time steps, we compute a new estimate for the time step – since the solution is nonlinear, it is most effective to adapt the value during the course of the simulation. In case the Courant number was chosen too aggressively, the simulation will typically blow up with time step NaN, so that is easy to detect here. One thing to note is that roundoff errors might propagate to the leading digits due to an interaction of slightly different time step selections that in turn lead to slightly different solutions. To decrease this sensitivity, it is common practice to round or truncate the time step size to a few digits, e.g. 3 in this case. In case the current time is near the prescribed 'tick' value for output (e.g. 0.02), we also write the output. After the end of the time loop, we summarize the computation by printing some statistics, which is mostly done by the TimerOutput::print_wall_time_statistics() function.

7.7.3.2 make_grid()

```
template<int dim>
void Euler_DG::EulerProblem< dim >::make_grid  [private]
```

As a mesh, this program implements different options, depending on the global variable `testcase`. if testcase == 1, then grid is channel with hole if testcase == 2, then grid is backward step if testcase == 3, then grid is forward step if testcase == 4, then grid is a rectangle if testcase == 5, then grid is a half cylinder (inner shell is sphere while outer shell is ellipse) if testcase == 6, then grid is a rectangle if testcase == 7, then grid is a cube

For each testcase we impose also the boundary condition on the ghost cells.

7.7.3.3 adapt_mesh()

```
template<int dim>
void Euler_DG::EulerProblem< dim >::adapt_mesh  [private]
```

This function take care of the adaptive mesh refinement. the three tasks this function performs is to first find out wich cells to refine, then to actually do the refinement and eventually transfer the solution vectors between the two different grids. The first task is simply achieved by computing the refinements indicator, as the gradient of the density $\eta_K = \log\left(1 + |\nabla \rho(x_K)|\right)$ where x_K is the center of the cell or as the gradient of the pressure $\eta = |\nabla p|$. The second task is to loop over all cells and mark those that we think should be refined. Then we need to transfer the various solution vectors from the old to the new grid while we do refinement. The SolutionTransfer class is our friend here.

7.7.3.4 make_dofs()

```
template<int dim>
void Euler_DG::EulerProblem< dim >::make_dofs  [private]
```

We call `make_dofs` every time we compute a refine of the mesh. With respect to quadrature, we want to select two different ways of computing the underlying integrals: The first is a flexible one, based on a template parameter `n_points_1d`. More accurate integration is necessary to avoid the aliasing problem due to the variable coefficients in the Euler operator. The second less accurate quadrature formula is a tight one based on `fe_degree+1` and needed for the inverse mass matrix. While that formula provides an exact inverse only on affine element shapes and not on deformed elements, it enables the fast inversion of the mass matrix by tensor product techniques, necessary to ensure optimal computational efficiency overall. For each cell, find neighbourig cell. This is needed for limiter

7.7.3.5 compute_cell_average()

```
template<int dim>
void Euler_DG::EulerProblem< dim >::compute_cell_average (
            LinearAlgebra::distributed::Vector< double > & current_solution )  [private]
```

Compute cell average solution

7.7.3.6 get_cell_average()

```
template<int dim>
void Euler_DG::EulerProblem< dim >::get_cell_average (
            const typename dealii::DoFHandler< dim >::cell_iterator & cell,
            dealii::Vector< double > & avg )  [private]
```

if cell is activem return cell average. if cell is not active, return area average of child cells.

7.7.3.7 compute_shock_indicator()

```
template<int dim>
void Euler_DG::EulerProblem< dim >::compute_shock_indicator (
            LinearAlgebra::distributed::Vector< double > & current_solution )  [private]
```

Compute shock indicator - KXRCF indicator.

7.7.3.8 apply_limiter_TVB()

```
template<int dim>
LinearAlgebra::distributed::Vector< Number > Euler_DG::EulerProblem< dim >::apply_limiter_TVB
(
            LinearAlgebra::distributed::Vector< Number > & current_solution )  [private]
```

Apply the TVB limiter using or minmod function or van Albada function

7.7.3.9 apply_filter()

```
template<int dim>
LinearAlgebra::distributed::Vector< Number > Euler_DG::EulerProblem< dim >::apply_filter (
            LinearAlgebra::distributed::Vector< Number > & sol_H,
            LinearAlgebra::distributed::Vector< Number > & sol_M )  [private]
```

Apply filtering monotonization approach

7.7.3.10 update()

```
template<int dim>
void Euler_DG::EulerProblem< dim >::update (
            const double current_time,
            const double time_step,
            LinearAlgebra::distributed::Vector< Number > & solution_np,
            LinearAlgebra::distributed::Vector< Number > & tmp_sol_n,
            LinearAlgebra::distributed::Vector< Number > & solution_np_Q0,
            LinearAlgebra::distributed::Vector< Number > & tmp_sol_n_Q0 )  [private]
```

Let us move to the function that does an entire stage of a Runge–Kutta update. It calls EulerOperator::apply() followed by some updates to the vectors. Rather than performing these steps through the vector interfaces, we here present an alternative strategy that is faster on cache-based architectures. At each step, we apply the limiter in order to control the solution

7.7.3.11 output_results()

```
template<int dim>
void Euler_DG::EulerProblem< dim >::output_results (
            const unsigned int result_number )  [private]
```

We let the postprocessor defined above control most of the output, except for the primal field that we write directly. For the analytical solution test case, we also perform another projection of the analytical solution and print the difference between that field and the numerical solution. Once we have defined all quantities to be written, we build the patches for output. We create a high-order VTK output by setting the appropriate flag, which enables us to visualize fields of high polynomial degrees. Finally, we call the `DataOutInterface::write_vtu_in↩_parallel()` function to write the result to the given file name. This function uses special MPI parallel write facilities, which are typically more optimized for parallel file systems than the standard library's `std::ofstream` variants used in most other tutorial programs. A particularly nice feature of the `write_vtu_in_parallel()` function is the fact that it can combine output from all MPI ranks into a single file, making it unnecessary to have a central record of all such files (namely, the "pvtu" file).

For parallel programs, it is often instructive to look at the partitioning of cells among processors. To this end, one can pass a vector of numbers to DataOut::add_data_vector() that contains as many entries as the current processor has active cells; these numbers should then be the rank of the processor that owns each of these cells. Such a vector could, for example, be obtained from GridTools::get_subdomain_association(). On the other hand, on each MPI process, DataOut will only read those entries that correspond to locally owned cells, and these of course all have the same value: namely, the rank of the current process. What is in the remaining entries of the vector doesn't actually matter, and so we can just get away with a cheap trick: We just fill *all* values of the vector we give to DataOut::add_data_vector() with the rank of the current MPI process. The key is that on each process, only the entries corresponding to the locally owned cells will be read, ignoring the (wrong) values in other entries. The fact that every process submits a vector in which the correct subset of entries is correct is all that is necessary.

7.7.4 Member Data Documentation

7.7.4.1 parameters

```
template<int dim>
Parameters::Data_Storage& Euler_DG::EulerProblem< dim >::parameters  [private]
```

7.7.4.2 solution

```
template<int dim>
LinearAlgebra::distributed::Vector<Number> Euler_DG::EulerProblem< dim >::solution  [private]
```

7.7.4.3 tmp_solution

```
template<int dim>
LinearAlgebra::distributed::Vector<Number> Euler_DG::EulerProblem< dim >::tmp_solution  [private]
```

7.7.4.4 sol_aux

```
template<int dim>
LinearAlgebra::distributed::Vector<Number> Euler_DG::EulerProblem< dim >::sol_aux  [private]
```

7.7.4.5 solution_Q0

```
template<int dim>
LinearAlgebra::distributed::Vector<Number> Euler_DG::EulerProblem< dim >::solution_Q0  [private]
```

7.7.4.6 tmp_solution_Q0

```
template<int dim>
LinearAlgebra::distributed::Vector<Number> Euler_DG::EulerProblem< dim >::tmp_solution_Q0
[private]
```

7.7.4.7 pcout

```
template<int dim>
ConditionalOStream Euler_DG::EulerProblem< dim >::pcout  [private]
```

7.7.4.8 triangulation

```
template<int dim>
Triangulation<dim> Euler_DG::EulerProblem< dim >::triangulation  [private]
```

7.7.4.9 fe

```
template<int dim>
FESystem<dim> Euler_DG::EulerProblem< dim >::fe  [private]
```

7.7.4.10 fe_Q0

```
template<int dim>
FESystem<dim> Euler_DG::EulerProblem< dim >::fe_Q0  [private]
```

7.7.4.11 lcell

```
template<int dim>
std::vector<typename dealii::DoFHandler<dim>::cell_iterator> Euler_DG::EulerProblem< dim >↩
::lcell  [private]
```

7.7.4.12 rcell

```
template<int dim>
std::vector<typename dealii::DoFHandler<dim>::cell_iterator> Euler_DG::EulerProblem< dim >↩
::rcell  [private]
```

7.7.4.13 bcell

```
template<int dim>
std::vector<typename dealii::DoFHandler<dim>::cell_iterator> Euler_DG::EulerProblem< dim >↩
::bcell  [private]
```

7.7.4.14 tcell

```
template<int dim>
std::vector<typename dealii::DoFHandler<dim>::cell_iterator> Euler_DG::EulerProblem< dim >↩
::tcell  [private]
```

7.7.4.15 fe_cell

```
template<int dim>
const dealii::FE_DGQ<dim> Euler_DG::EulerProblem< dim >::fe_cell  [private]
```

7.7.4.16 dh_cell

```
template<int dim>
dealii::DoFHandler<dim> Euler_DG::EulerProblem< dim >::dh_cell  [private]
```

7.7.4.17 shock_indicator

```
template<int dim>
Vector<double> Euler_DG::EulerProblem< dim >::shock_indicator  [private]
```

7.7.4.18 jump_indicator

```
template<int dim>
Vector<double> Euler_DG::EulerProblem< dim >::jump_indicator  [private]
```

7.7.4.19 mapping

```
template<int dim>
const MappingQ1<dim> Euler_DG::EulerProblem< dim >::mapping  [private]
```

7.7.4.20 mapping_Q0

```
template<int dim>
const MappingQ1<dim> Euler_DG::EulerProblem< dim >::mapping_Q0  [private]
```

7.7.4.21 dof_handler

```
template<int dim>
DoFHandler<dim> Euler_DG::EulerProblem< dim >::dof_handler  [private]
```

7.7.4.22 dof_handler_Q0

```
template<int dim>
DoFHandler<dim> Euler_DG::EulerProblem< dim >::dof_handler_Q0  [private]
```

7.7.4.23 dof_handlers

```
template<int dim>
std::vector<const DoFHandler<dim>*> Euler_DG::EulerProblem< dim >::dof_handlers  [private]
```

7.7.4.24 dof_handlers_Q0

```
template<int dim>
std::vector<const DoFHandler<dim>*> Euler_DG::EulerProblem< dim >::dof_handlers_Q0  [private]
```

7.7.4.25 estimated_indicator_per_cell

```
template<int dim>
Vector<double> Euler_DG::EulerProblem< dim >::estimated_indicator_per_cell  [private]
```

7.7.4.26 quadrature

```
template<int dim>
const QGauss<dim> Euler_DG::EulerProblem< dim >::quadrature  [private]
```

7.7.4.27 quadratures

```
template<int dim>
std::vector<QGauss<1> > Euler_DG::EulerProblem< dim >::quadratures  [private]
```

7.7.4.28 quadratures_Q0

```
template<int dim>
std::vector<QGauss<1> > Euler_DG::EulerProblem< dim >::quadratures_Q0  [private]
```

7.7.4.29 cell_degree

```
template<int dim>
std::vector<unsigned int> Euler_DG::EulerProblem< dim >::cell_degree  [private]
```

7.7.4.30 re_update

```
template<int dim>
std::vector<bool> Euler_DG::EulerProblem< dim >::re_update  [private]
```

7.7.4.31 cell_average

```
template<int dim>
std::vector< dealii::Vector<double> > Euler_DG::EulerProblem< dim >::cell_average  [private]
```

7.7.4.32 timer

```
template<int dim>
TimerOutput Euler_DG::EulerProblem< dim >::timer  [private]
```

7.7.4.33 euler_operator

```
template<int dim>
EulerOperator<2, 2,5> Euler_DG::EulerProblem< dim >::euler_operator  [private]
```

7.7.4.34 euler_operator_Q0

```
template<int dim>
EulerOperator<2, 0,1> Euler_DG::EulerProblem< dim >::euler_operator_Q0  [private]
```

7.7.4.35 time

```
template<int dim>
double Euler_DG::EulerProblem< dim >::time  [private]
```

7.7.4.36 time_step

```
template<int dim>
double Euler_DG::EulerProblem< dim >::time_step  [private]
```

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

- eulerproblem.h
- eulerproblem.cpp

7.8 Euler_DG::EulerProblem< dim >::Postprocessor Class Reference

Inheritance diagram for Euler_DG::EulerProblem< dim >::Postprocessor:

Collaboration diagram for Euler_DG::EulerProblem< dim >::Postprocessor:

Public Member Functions

- Postprocessor (Parameters::Data_Storage &parameters, int a)
- virtual void evaluate_vector_field (const DataPostprocessorInputs::Vector< dim > &inputs, std::vector< Vector< double >> &computed_quantities) const override
- virtual std::vector< std::string > get_names () const override
- virtual std::vector< DataComponentInterpretation::DataComponentInterpretation > get_data_component_interpretation () const override
- virtual UpdateFlags get_needed_update_flags () const override

Public Attributes

- Parameters::Data_Storage & parameters
- int a

Private Attributes

- const bool do_schlieren_plot

7.8.1 Constructor & Destructor Documentation

7.8.1.1 Postprocessor()

```
template<int dim>
Euler_DG::EulerProblem< dim >::Postprocessor::Postprocessor (
            Parameters::Data_Storage & parameters_in,
            int a_ )
```

constructor of Postprocessor class

7.8.2 Member Function Documentation

7.8.2.1 evaluate_vector_field()

```
template<int dim>
void Euler_DG::EulerProblem< dim >::Postprocessor::evaluate_vector_field (
            const DataPostprocessorInputs::Vector< dim > & inputs,
            std::vector< Vector< double >> & computed_quantities ) const  [override], [virtual]
```

For the main evaluation of the field variables, we first check that the lengths of the arrays equal the expected values. Then we loop over all evaluation points and fill the respective information: First we fill the primal solution variables of density ρ, momentum $\rho\mathbf{u}$ and energy E, then we compute the derived velocity $\mathbf{u}$, the pressure p, the speed of sound $c = \sqrt{\gamma p/\rho}$, as well as the Schlieren plot showing $s = |\nabla\rho|^2$ in case it is enabled.

7.8.2.2 get_names()

```
template<int dim>
std::vector< std::string > Euler_DG::EulerProblem< dim >::Postprocessor::get_names  [override],
[virtual]
```

7.8.2.3 get_data_component_interpretation()

```
template<int dim>
std::vector< DataComponentInterpretation::DataComponentInterpretation > Euler_DG::EulerProblem<
dim >::Postprocessor::get_data_component_interpretation  [override], [virtual]
```

For the interpretation of quantities, we have scalar density, energy, pressure, speed of sound, and the Schlieren plot, and vectors for the momentum and the velocity.

7.8.2.4 get_needed_update_flags()

```
template<int dim>
UpdateFlags Euler_DG::EulerProblem< dim >::Postprocessor::get_needed_update_flags  [override],
[virtual]
```

With respect to the necessary update flags, we only need the values for all quantities but the Schlieren plot, which is based on the density gradient.

7.8.3 Member Data Documentation

7.8.3.1 parameters

```
template<int dim>
Parameters::Data_Storage& Euler_DG::EulerProblem< dim >::Postprocessor::parameters
```

7.8.3.2 a

```
template<int dim>
int Euler_DG::EulerProblem< dim >::Postprocessor::a
```

int value for plot different solution

7.8.3.3 do_schlieren_plot

```
template<int dim>
const bool Euler_DG::EulerProblem< dim >::Postprocessor::do_schlieren_plot  [private]
```

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

- eulerproblem.h
- eulerproblem.cpp

Chapter 8

File Documentation

8.1 main.cpp File Reference

```
#include <deal.II/base/conditional_ostream.h>
#include <deal.II/base/function.h>
#include <deal.II/base/logstream.h>
#include <deal.II/base/timer.h>
#include <deal.II/base/parameter_handler.h>
#include <deal.II/base/time_stepping.h>
#include <deal.II/base/utilities.h>
#include <deal.II/base/vectorization.h>
#include <deal.II/base/quadrature_lib.h>
#include <deal.II/base/thread_local_storage.h>
#include <deal.II/base/revision.h>
#include <deal.II/grid/manifold_lib.h>
#include <deal.II/grid/tria_iterator.h>
#include <deal.II/grid/tria_accessor.h>
#include <deal.II/base/aligned_vector.h>
#include <deal.II/distributed/tria.h>
#include <deal.II/meshworker/mesh_loop.h>
#include <deal.II/dofs/dof_handler.h>
#include <deal.II/dofs/dof_tools.h>
#include <deal.II/dofs/dof_renumbering.h>
#include <deal.II/fe/fe_q.h>
#include <deal.II/fe/fe_dgq.h>
#include <deal.II/fe/fe_tools.h>
#include <deal.II/fe/fe_system.h>
#include <deal.II/fe/fe_values.h>
#include <deal.II/fe/mapping_q1.h>
#include <deal.II/grid/grid_generator.h>
#include <deal.II/grid/tria.h>
#include <deal.II/grid/grid_refinement.h>
#include <deal.II/grid/grid_out.h>
#include <deal.II/distributed/grid_refinement.h>
#include <deal.II/lac/affine_constraints.h>
#include <deal.II/lac/la_parallel_vector.h>
#include <deal.II/lac/vector.h>
#include <deal.II/lac/sparse_matrix.h>
#include <deal.II/distributed/solution_transfer.h>
#include <deal.II/matrix_free/fe_evaluation.h>
```

```
#include <deal.II/matrix_free/matrix_free.h>
#include <deal.II/matrix_free/operators.h>
#include <deal.II/numerics/data_out.h>
#include <deal.II/numerics/derivative_approximation.h>
#include <deal.II/numerics/solution_transfer.h>
#include <deal.II/numerics/matrix_tools.h>
#include <deal.II/numerics/error_estimator.h>
#include <deal.II/numerics/vector_tools.h>
#include <fstream>
#include <iomanip>
#include <iostream>
#include <array>
#include "parameters.h"
#include "eulerproblem.h"
```

Namespaces

- Euler_DG

Functions

- int main (int argc, char **argv)

Variables

- constexpr unsigned int Euler_DG::dimension = 2

8.1.1 Function Documentation

8.1.1.1 main()

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

The main() function is not surprising and follows what was done in all previous MPI programs: As we run an MPI program, we need to call `MPI_Init()` and `MPI_Finalize()`, which we do through the Utilities::MPI::MPI_InitFinalize data structure. Note that we run the program only with MPI, and set the thread count to 1. argv[1] can be inputfile_sod.prm, inputfile_DMR.prm, inputfile_fstep.prm, inputfile_bstep.prm, inputfile_cylinder.prm, inputfile_2Driemann.prm

8.2 parameters.h File Reference

```
#include <deal.II/base/conditional_ostream.h>
#include <deal.II/base/function.h>
#include <deal.II/base/logstream.h>
#include <deal.II/base/timer.h>
#include <deal.II/base/time_stepping.h>
#include <deal.II/base/utilities.h>
#include <deal.II/base/vectorization.h>
#include <deal.II/base/quadrature_lib.h>
#include <deal.II/base/thread_local_storage.h>
#include <deal.II/base/revision.h>
#include <deal.II/grid/manifold_lib.h>
#include <deal.II/grid/tria_iterator.h>
#include <deal.II/grid/tria_accessor.h>
#include <deal.II/base/aligned_vector.h>
#include <deal.II/distributed/tria.h>
#include <deal.II/meshworker/mesh_loop.h>
#include <deal.II/dofs/dof_handler.h>
#include <deal.II/dofs/dof_tools.h>
#include <deal.II/dofs/dof_renumbering.h>
#include <deal.II/fe/fe_dgq.h>
#include <deal.II/fe/fe_system.h>
#include <deal.II/fe/fe_values.h>
#include <deal.II/fe/mapping_q1.h>
#include <deal.II/grid/grid_generator.h>
#include <deal.II/grid/tria.h>
#include <deal.II/grid/grid_refinement.h>
#include <deal.II/grid/grid_out.h>
#include <deal.II/distributed/grid_refinement.h>
#include <deal.II/lac/affine_constraints.h>
#include <deal.II/lac/la_parallel_vector.h>
#include <deal.II/lac/vector.h>
#include <deal.II/lac/sparse_matrix.h>
#include <deal.II/distributed/solution_transfer.h>
#include <deal.II/matrix_free/fe_evaluation.h>
#include <deal.II/matrix_free/matrix_free.h>
#include <deal.II/matrix_free/operators.h>
#include <deal.II/numerics/data_out.h>
#include <deal.II/numerics/derivative_approximation.h>
#include <deal.II/numerics/solution_transfer.h>
#include <deal.II/numerics/matrix_tools.h>
#include <deal.II/numerics/error_estimator.h>
#include <deal.II/numerics/vector_tools.h>
#include <deal.II/base/parameter_handler.h>
#include <fstream>
#include <iomanip>
#include <iostream>
#include <array>
```

Include dependency graph for parameters.h: This graph shows which files directly or indirectly include this file:

Classes

- struct Parameters::Solver
- struct Parameters::Data_Storage

Namespaces

- Parameters

8.3 parameters.cpp File Reference

```
#include "parameters.h"
```
Include dependency graph for parameters.cpp:

Namespaces

- Parameters

8.4 equationdata.h File Reference

```
#include <deal.II/base/conditional_ostream.h>
#include <deal.II/base/function.h>
#include <deal.II/base/logstream.h>
#include <deal.II/base/timer.h>
#include <deal.II/base/time_stepping.h>
#include <deal.II/base/utilities.h>
#include <deal.II/base/vectorization.h>
#include <deal.II/base/quadrature_lib.h>
#include <deal.II/base/thread_local_storage.h>
#include <deal.II/base/revision.h>
#include <deal.II/grid/manifold_lib.h>
#include <deal.II/grid/tria_iterator.h>
#include <deal.II/grid/tria_accessor.h>
#include <deal.II/base/aligned_vector.h>
#include <deal.II/distributed/tria.h>
#include <deal.II/meshworker/mesh_loop.h>
#include <deal.II/dofs/dof_handler.h>
#include <deal.II/dofs/dof_tools.h>
#include <deal.II/dofs/dof_renumbering.h>
#include <deal.II/fe/fe_dgq.h>
#include <deal.II/fe/fe_system.h>
#include <deal.II/fe/fe_values.h>
#include <deal.II/fe/mapping_q1.h>
#include <deal.II/grid/grid_generator.h>
#include <deal.II/grid/tria.h>
#include <deal.II/grid/grid_refinement.h>
#include <deal.II/grid/grid_out.h>
#include <deal.II/distributed/grid_refinement.h>
#include <deal.II/lac/affine_constraints.h>
#include <deal.II/lac/la_parallel_vector.h>
#include <deal.II/lac/vector.h>
#include <deal.II/lac/sparse_matrix.h>
#include <deal.II/distributed/solution_transfer.h>
#include <deal.II/matrix_free/fe_evaluation.h>
#include <deal.II/matrix_free/matrix_free.h>
#include <deal.II/numerics/data_out.h>
#include <deal.II/numerics/derivative_approximation.h>
```

```
#include <deal.II/numerics/solution_transfer.h>
#include <deal.II/numerics/matrix_tools.h>
#include <deal.II/numerics/error_estimator.h>
#include <deal.II/numerics/vector_tools.h>
#include <deal.II/matrix_free/operators.h>
#include <fstream>
#include <iomanip>
#include <iostream>
#include <array>
```

Include dependency graph for equationdata.h: This graph shows which files directly or indirectly include this file:

Classes

- class EquationData::ExactSolution< dim >
- class EquationData::BoundaryData< dim >
- class EquationData::InitialData< dim >

Namespaces

- EquationData

8.5 equationdata.cpp File Reference

```
#include <deal.II/base/conditional_ostream.h>
#include <deal.II/base/function.h>
#include <deal.II/base/logstream.h>
#include <deal.II/base/timer.h>
#include <deal.II/base/time_stepping.h>
#include <deal.II/base/utilities.h>
#include <deal.II/base/vectorization.h>
#include <deal.II/base/quadrature_lib.h>
#include <deal.II/base/thread_local_storage.h>
#include <deal.II/base/revision.h>
#include <deal.II/grid/manifold_lib.h>
#include <deal.II/grid/tria_iterator.h>
#include <deal.II/grid/tria_accessor.h>
#include <deal.II/base/aligned_vector.h>
#include <deal.II/distributed/tria.h>
#include <deal.II/meshworker/mesh_loop.h>
#include <deal.II/dofs/dof_handler.h>
#include <deal.II/dofs/dof_tools.h>
#include <deal.II/dofs/dof_renumbering.h>
#include <deal.II/fe/fe_dgq.h>
#include <deal.II/fe/fe_system.h>
#include <deal.II/fe/fe_values.h>
#include <deal.II/fe/mapping_q1.h>
#include <deal.II/grid/grid_generator.h>
#include <deal.II/grid/tria.h>
#include <deal.II/grid/grid_refinement.h>
#include <deal.II/grid/grid_out.h>
#include <deal.II/distributed/grid_refinement.h>
#include <deal.II/lac/affine_constraints.h>
#include <deal.II/lac/la_parallel_vector.h>
```

```
#include <deal.II/lac/vector.h>
#include <deal.II/lac/sparse_matrix.h>
#include <deal.II/distributed/solution_transfer.h>
#include <deal.II/matrix_free/fe_evaluation.h>
#include <deal.II/matrix_free/matrix_free.h>
#include <deal.II/numerics/data_out.h>
#include <deal.II/numerics/derivative_approximation.h>
#include <deal.II/numerics/solution_transfer.h>
#include <deal.II/numerics/matrix_tools.h>
#include <deal.II/numerics/error_estimator.h>
#include <deal.II/numerics/vector_tools.h>
#include <deal.II/matrix_free/operators.h>
#include <fstream>
#include <iomanip>
#include <iostream>
#include <array>
#include "parameters.h"
#include "equationdata.h"
```

Include dependency graph for equationdata.cpp:

8.6 operations.h File Reference

```
#include <deal.II/base/conditional_ostream.h>
#include <deal.II/base/function.h>
#include <deal.II/base/logstream.h>
#include <deal.II/base/timer.h>
#include <deal.II/base/time_stepping.h>
#include <deal.II/base/utilities.h>
#include <deal.II/base/vectorization.h>
#include <deal.II/base/quadrature_lib.h>
#include <deal.II/base/thread_local_storage.h>
#include <deal.II/base/revision.h>
#include <deal.II/grid/manifold_lib.h>
#include <deal.II/grid/tria_iterator.h>
#include <deal.II/grid/tria_accessor.h>
#include <deal.II/base/aligned_vector.h>
#include <deal.II/distributed/tria.h>
#include <deal.II/meshworker/mesh_loop.h>
#include <deal.II/dofs/dof_handler.h>
#include <deal.II/dofs/dof_tools.h>
#include <deal.II/dofs/dof_renumbering.h>
#include <deal.II/fe/fe_dgq.h>
#include <deal.II/fe/fe_system.h>
#include <deal.II/fe/fe_values.h>
#include <deal.II/fe/mapping_q1.h>
#include <deal.II/grid/grid_generator.h>
#include <deal.II/grid/tria.h>
#include <deal.II/grid/grid_refinement.h>
#include <deal.II/grid/grid_out.h>
#include <deal.II/distributed/grid_refinement.h>
#include <deal.II/lac/affine_constraints.h>
#include <deal.II/lac/la_parallel_vector.h>
#include <deal.II/lac/vector.h>
#include <deal.II/lac/sparse_matrix.h>
#include <deal.II/distributed/solution_transfer.h>
#include <deal.II/matrix_free/fe_evaluation.h>
```

```
#include <deal.II/matrix_free/matrix_free.h>
#include <deal.II/matrix_free/operators.h>
#include <deal.II/numerics/data_out.h>
#include <deal.II/numerics/derivative_approximation.h>
#include <deal.II/numerics/solution_transfer.h>
#include <deal.II/numerics/matrix_tools.h>
#include <deal.II/numerics/error_estimator.h>
#include <deal.II/numerics/vector_tools.h>
#include <fstream>
#include <iomanip>
#include <iostream>
#include <array>
#include <deal.II/base/parameter_handler.h>
#include "equationdata.h"
#include "parameters.h"
```

Include dependency graph for operations.h: This graph shows which files directly or indirectly include this file:

Namespaces

- Euler_DG

Functions

- template<int dim, typename Number >
  DEAL_II_ALWAYS_INLINE Tensor< 1, dim, Number > Euler_DG::euler_velocity (const Tensor< 1, dim+2, Number > &conserved_variables)
- template<int dim, typename Number >
  DEAL_II_ALWAYS_INLINE Number Euler_DG::euler_pressure (const Tensor< 1, dim+2, Number > &conserved_variables, Parameters::Data_Storage &par)
- template<int dim, typename Number >
  DEAL_II_ALWAYS_INLINE Tensor< 1, dim+2, Tensor< 1, dim, Number > > Euler_DG::euler_flux (const Tensor< 1, dim+2, Number > &conserved_variables, Parameters::Data_Storage &par)
- template<int n_components, int dim, typename Number >
  DEAL_II_ALWAYS_INLINE Tensor< 1, n_components, Number > Euler_DG::operator* (const Tensor< 1, n_components, Tensor< 1, dim, Number >> &matrix, const Tensor< 1, dim, Number > &vector)
- template<typename Number >
  DEAL_II_ALWAYS_INLINE Number Euler_DG::sgn (Number &val)
- template<int dim, typename Number >
  DEAL_II_ALWAYS_INLINE Tensor< 1, dim+2, Number > Euler_DG::euler_numerical_flux (const Tensor< 1, dim+2, Number > &u_m, const Tensor< 1, dim+2, Number > &u_p, const Tensor< 1, dim, Number > &normal, Parameters::Data_Storage &par)
- template<int dim, typename Number >
  VectorizedArray< Number > Euler_DG::evaluate_function (const Function< dim > &function, const Point< dim, VectorizedArray< Number >> &p_vectorized, const unsigned int component)
- template<int dim, typename Number , int n_components = dim + 2>
  Tensor< 1, n_components, VectorizedArray< Number > > Euler_DG::evaluate_function (const Function< dim > &function, const Point< dim, VectorizedArray< Number >> &p_vectorized)

8.7 euleroperator.h File Reference

```
#include <deal.II/base/conditional_ostream.h>
#include <deal.II/base/function.h>
```

```
#include <deal.II/base/logstream.h>
#include <deal.II/base/timer.h>
#include <deal.II/base/time_stepping.h>
#include <deal.II/base/utilities.h>
#include <deal.II/base/vectorization.h>
#include <deal.II/base/quadrature_lib.h>
#include <deal.II/base/thread_local_storage.h>
#include <deal.II/base/revision.h>
#include <deal.II/grid/manifold_lib.h>
#include <deal.II/grid/tria_iterator.h>
#include <deal.II/grid/tria_accessor.h>
#include <deal.II/base/aligned_vector.h>
#include <deal.II/distributed/tria.h>
#include <deal.II/meshworker/mesh_loop.h>
#include <deal.II/dofs/dof_handler.h>
#include <deal.II/dofs/dof_tools.h>
#include <deal.II/dofs/dof_renumbering.h>
#include <deal.II/fe/fe_dgq.h>
#include <deal.II/fe/fe_system.h>
#include <deal.II/fe/fe_values.h>
#include <deal.II/fe/mapping_q1.h>
#include <deal.II/grid/grid_generator.h>
#include <deal.II/grid/tria.h>
#include <deal.II/grid/grid_refinement.h>
#include <deal.II/grid/grid_out.h>
#include <deal.II/distributed/grid_refinement.h>
#include <deal.II/lac/affine_constraints.h>
#include <deal.II/lac/la_parallel_vector.h>
#include <deal.II/lac/vector.h>
#include <deal.II/lac/sparse_matrix.h>
#include <deal.II/distributed/solution_transfer.h>
#include <deal.II/matrix_free/fe_evaluation.h>
#include <deal.II/matrix_free/matrix_free.h>
#include <deal.II/numerics/data_out.h>
#include <deal.II/numerics/derivative_approximation.h>
#include <deal.II/numerics/solution_transfer.h>
#include <deal.II/numerics/matrix_tools.h>
#include <deal.II/numerics/error_estimator.h>
#include <deal.II/numerics/vector_tools.h>
#include <deal.II/matrix_free/operators.h>
#include <fstream>
#include <iomanip>
#include <iostream>
#include <array>
#include "parameters.h"
#include "equationdata.h"
#include "operations.h"
```

Include dependency graph for euleroperator.h: This graph shows which files directly or indirectly include this file:

Classes

- class Euler_DG::EulerOperator< dim, degree, n_points_1d >

Namespaces

- Euler_DG

Typedefs

- using Euler_DG::Number = double

8.8 euleroperator.cpp File Reference

```
#include "euleroperator.h"
```
Include dependency graph for euleroperator.cpp:

Namespaces

- Euler_DG

8.9 eulerproblem.h File Reference

```
#include <deal.II/base/conditional_ostream.h>
#include <deal.II/base/function.h>
#include <deal.II/base/logstream.h>
#include <deal.II/base/timer.h>
#include <deal.II/base/time_stepping.h>
#include <deal.II/base/utilities.h>
#include <deal.II/base/vectorization.h>
#include <deal.II/base/quadrature_lib.h>
#include <deal.II/base/parameter_handler.h>
#include <deal.II/base/thread_local_storage.h>
#include <deal.II/base/revision.h>
#include <deal.II/grid/manifold_lib.h>
#include <deal.II/grid/tria_iterator.h>
#include <deal.II/grid/tria_accessor.h>
#include <deal.II/base/aligned_vector.h>
#include <deal.II/distributed/tria.h>
#include <deal.II/meshworker/mesh_loop.h>
#include <deal.II/dofs/dof_handler.h>
#include <deal.II/dofs/dof_tools.h>
#include <deal.II/dofs/dof_renumbering.h>
#include <deal.II/fe/fe_dgq.h>
#include <deal.II/fe/fe_system.h>
#include <deal.II/fe/fe_values.h>
#include <deal.II/fe/mapping_q1.h>
#include <deal.II/grid/grid_generator.h>
#include <deal.II/grid/tria.h>
#include <deal.II/grid/grid_refinement.h>
#include <deal.II/grid/grid_out.h>
#include <deal.II/distributed/grid_refinement.h>
#include <deal.II/lac/affine_constraints.h>
#include <deal.II/lac/la_parallel_vector.h>
#include <deal.II/lac/vector.h>
#include <deal.II/lac/sparse_matrix.h>
#include <deal.II/distributed/solution_transfer.h>
#include <deal.II/matrix_free/fe_evaluation.h>
#include <deal.II/matrix_free/matrix_free.h>
#include <deal.II/matrix_free/operators.h>
```

```
#include <deal.II/numerics/data_out.h>
#include <deal.II/numerics/derivative_approximation.h>
#include <deal.II/numerics/solution_transfer.h>
#include <deal.II/numerics/matrix_tools.h>
#include <deal.II/numerics/error_estimator.h>
#include <deal.II/numerics/vector_tools.h>
#include <fstream>
#include <iomanip>
#include <iostream>
#include <array>
#include <vector>
#include "parameters.h"
#include "equationdata.h"
#include "euleroperator.h"
#include "operations.h"
```

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

Classes

- class Euler_DG::EulerProblem< dim >
- class Euler_DG::EulerProblem< dim >::Postprocessor

Namespaces

- Euler_DG

8.10 eulerproblem.cpp File Reference

```
#include <deal.II/base/conditional_ostream.h>
#include <deal.II/base/function.h>
#include <deal.II/base/logstream.h>
#include <deal.II/base/timer.h>
#include <deal.II/base/time_stepping.h>
#include <deal.II/base/utilities.h>
#include <deal.II/base/vectorization.h>
#include <deal.II/base/quadrature_lib.h>
#include <deal.II/base/parameter_handler.h>
#include <deal.II/base/thread_local_storage.h>
#include <deal.II/base/revision.h>
#include <deal.II/grid/manifold_lib.h>
#include <deal.II/grid/tria_iterator.h>
#include <deal.II/grid/tria_accessor.h>
#include <deal.II/base/aligned_vector.h>
#include <deal.II/distributed/tria.h>
#include <deal.II/meshworker/mesh_loop.h>
#include <deal.II/dofs/dof_handler.h>
#include <deal.II/dofs/dof_tools.h>
#include <deal.II/dofs/dof_renumbering.h>
#include <deal.II/fe/fe_dgq.h>
#include <deal.II/fe/fe_system.h>
#include <deal.II/fe/fe_values.h>
#include <deal.II/fe/mapping_q1.h>
#include <deal.II/grid/grid_generator.h>
```

```
#include <deal.II/grid/tria.h>
#include <deal.II/grid/grid_refinement.h>
#include <deal.II/grid/grid_out.h>
#include <deal.II/distributed/grid_refinement.h>
#include <deal.II/lac/affine_constraints.h>
#include <deal.II/lac/la_parallel_vector.h>
#include <deal.II/lac/vector.h>
#include <deal.II/lac/sparse_matrix.h>
#include <deal.II/distributed/solution_transfer.h>
#include <deal.II/matrix_free/fe_evaluation.h>
#include <deal.II/matrix_free/matrix_free.h>
#include <deal.II/matrix_free/operators.h>
#include <deal.II/numerics/data_out.h>
#include <deal.II/numerics/derivative_approximation.h>
#include <deal.II/numerics/solution_transfer.h>
#include <deal.II/numerics/matrix_tools.h>
#include <deal.II/numerics/error_estimator.h>
#include <deal.II/numerics/vector_tools.h>
#include <fstream>
#include <iomanip>
#include <iostream>
#include <array>
#include "eulerproblem.h"
```

Include dependency graph for eulerproblem.cpp:

Namespaces

- Euler_DG

Functions

- template<typename ITERATOR >
  unsigned int Euler_DG::cell_number (const ITERATOR &cell)
- double Euler_DG::minmod (const double &a, const double &b, const double &c, const double &Mdx2)
- double Euler_DG::vanAlbada (const double &c, const double &a, const double &b, const double &Mdx2, const double &eps)

Index

a
EquationData::BoundaryData< dim >, 21
Euler_DG::EulerProblem< dim >::Postprocessor, 49
adapt_mesh
Euler_DG::EulerProblem< dim >, 39
apply
Euler_DG::EulerOperator< dim, degree, n_points_1d >, 30
apply_filter
Euler_DG::EulerProblem< dim >, 40
apply_limiter_TVB
Euler_DG::EulerProblem< dim >, 40

bcell
Euler_DG::EulerProblem< dim >, 43
beta
Parameters::Data_Storage, 24
beta_density
Parameters::Data_Storage, 24
beta_energy
Parameters::Data_Storage, 24
beta_momentum
Parameters::Data_Storage, 24
body_force
Euler_DG::EulerOperator< dim, degree, n_points_1d >, 36
BoundaryData
EquationData::BoundaryData< dim >, 20

cell_average
Euler_DG::EulerProblem< dim >, 46
cell_degree
Euler_DG::EulerProblem< dim >, 46
cell_number
Euler_DG, 12
compute_cell_average
Euler_DG::EulerProblem< dim >, 40
compute_cell_transport_speed
Euler_DG::EulerOperator< dim, degree, n_points_1d >, 32
compute_errors
Euler_DG::EulerOperator< dim, degree, n_points_1d >, 31
compute_shock_indicator
Euler_DG::EulerProblem< dim >, 40

data
Euler_DG::EulerOperator< dim, degree, n_points_1d >, 35
Data_Storage
Parameters::Data_Storage, 22
declare_parameters
Parameters::Solver, 27
dh_cell
Euler_DG::EulerProblem< dim >, 44
dimension
Euler_DG, 14
do_schlieren_plot
Euler_DG::EulerProblem< dim >::Postprocessor, 49
dof_handler
Euler_DG::EulerProblem< dim >, 45
dof_handler_Q0
Euler_DG::EulerProblem< dim >, 45
dof_handlers
Euler_DG::EulerProblem< dim >, 45
dof_handlers_Q0
Euler_DG::EulerProblem< dim >, 45

EquationData, 11
equationdata.cpp, 55
equationdata.h, 54
EquationData::BoundaryData< dim >, 20
a, 21
BoundaryData, 20
parameters, 21
value, 21
EquationData::ExactSolution< dim >, 17
ExactSolution, 17
parameters, 18
value, 18
EquationData::InitialData< dim >, 18
InitialData, 19
parameters, 19
value, 19
estimated_indicator_per_cell
Euler_DG::EulerProblem< dim >, 45
Euler_DG, 11
cell_number, 12
dimension, 14
euler_flux, 13
euler_numerical_flux, 14
euler_pressure, 13
euler_velocity, 13
evaluate_function, 14
minmod, 12
Number, 12
operator*, 13
sgn, 13

vanAlbada, 12
Euler_DG::EulerOperator< dim, degree, n_points_1d >, 27
apply, 30
body_force, 36
compute_cell_transport_speed, 32
compute_errors, 31
data, 35
EulerOperator, 29
inflow_boundaries, 35
initialize_vector, 32
local_apply_boundary_face, 34
local_apply_cell, 33
local_apply_face, 34
local_apply_inverse_mass_matrix, 32
n_quadrature_points_1d, 35
parameters, 35
project, 31
reinit, 29
set_body_force, 30
set_inflow_boundary, 29
set_subsonic_outflow_boundary, 30
set_supersonic_outflow_boundary, 30
set_wall_boundary, 30
subsonic_outflow_boundaries, 36
supersonic_outflow_boundaries, 36
timer, 35
wall_boundaries, 36
Euler_DG::EulerProblem< dim >, 36
adapt_mesh, 39
apply_filter, 40
apply_limiter_TVB, 40
bcell, 43
cell_average, 46
cell_degree, 46
compute_cell_average, 40
compute_shock_indicator, 40
dh_cell, 44
dof_handler, 45
dof_handler_Q0, 45
dof_handlers, 45
dof_handlers_Q0, 45
estimated_indicator_per_cell, 45
euler_operator, 46
euler_operator_Q0, 46
EulerProblem, 38
fe, 43
fe_cell, 44
fe_Q0, 43
get_cell_average, 40
jump_indicator, 44
lcell, 43
make_dofs, 39
make_grid, 39
mapping, 44
mapping_Q0, 44
output_results, 41
parameters, 42
pcout, 42
quadrature, 45
quadratures, 45
quadratures_Q0, 46
rcell, 43
re_update, 46
run, 39
shock_indicator, 44
sol_aux, 42
solution, 42
solution_Q0, 42
tcell, 44
time, 47
time_step, 47
timer, 46
tmp_solution, 42
tmp_solution_Q0, 42
triangulation, 43
update, 41
Euler_DG::EulerProblem< dim >::Postprocessor, 47
a, 49
do_schlieren_plot, 49
evaluate_vector_field, 48
get_data_component_interpretation, 48
get_names, 48
get_needed_update_flags, 48
parameters, 49
Postprocessor, 48
euler_flux
Euler_DG, 13
euler_numerical_flux
Euler_DG, 14
euler_operator
Euler_DG::EulerProblem< dim >, 46
euler_operator_Q0
Euler_DG::EulerProblem< dim >, 46
euler_pressure
Euler_DG, 13
euler_velocity
Euler_DG, 13
EulerNumericalFlux
Parameters::Solver, 26
EulerOperator
Euler_DG::EulerOperator< dim, degree, n_points_1d >, 29
euleroperator.cpp, 59
euleroperator.h, 57
EulerProblem
Euler_DG::EulerProblem< dim >, 38
eulerproblem.cpp, 60
eulerproblem.h, 59
evaluate_function
Euler_DG, 14
evaluate_vector_field
Euler_DG::EulerProblem< dim >::Postprocessor, 48
ExactSolution
EquationData::ExactSolution< dim >, 17

fe
Euler_DG::EulerProblem< dim >, 43
fe_cell
Euler_DG::EulerProblem< dim >, 44
fe_degree
Parameters::Data_Storage, 23
fe_degree_Q0
Parameters::Data_Storage, 23
fe_Q0
Euler_DG::EulerProblem< dim >, 43
final_time
Parameters::Data_Storage, 25
function_limiter
Parameters::Data_Storage, 25

gamma
Parameters::Data_Storage, 24
get_cell_average
Euler_DG::EulerProblem< dim >, 40
get_data_component_interpretation
Euler_DG::EulerProblem< dim >::Postprocessor, 48
get_names
Euler_DG::EulerProblem< dim >::Postprocessor, 48
get_needed_update_flags
Euler_DG::EulerProblem< dim >::Postprocessor, 48

harten_lax_vanleer
Parameters::Solver, 27
HLLC
Parameters::Solver, 27
hllc_centered
Parameters::Solver, 27

inflow_boundaries
Euler_DG::EulerOperator< dim, degree, n_points_1d >, 35
InitialData
EquationData::InitialData< dim >, 19
initialize_vector
Euler_DG::EulerOperator< dim, degree, n_points_1d >, 32

jump_indicator
Euler_DG::EulerProblem< dim >, 44

lax_friedrichs
Parameters::Solver, 27
lcell
Euler_DG::EulerProblem< dim >, 43
local_apply_boundary_face
Euler_DG::EulerOperator< dim, degree, n_points_1d >, 34
local_apply_cell
Euler_DG::EulerOperator< dim, degree, n_points_1d >, 33
local_apply_face
Euler_DG::EulerOperator< dim, degree, n_points_1d >, 34
local_apply_inverse_mass_matrix
Euler_DG::EulerOperator< dim, degree, n_points_1d >, 32

M
Parameters::Data_Storage, 24
main
main.cpp, 52
main.cpp, 51
main, 52
make_dofs
Euler_DG::EulerProblem< dim >, 39
make_grid
Euler_DG::EulerProblem< dim >, 39
mapping
Euler_DG::EulerProblem< dim >, 44
mapping_Q0
Euler_DG::EulerProblem< dim >, 44
max_loc_refinements
Parameters::Data_Storage, 24
min_loc_refinements
Parameters::Data_Storage, 24
minmod
Euler_DG, 12

n_q_points_1d
Parameters::Data_Storage, 23
n_q_points_1d_Q0
Parameters::Data_Storage, 23
n_quadrature_points_1d
Euler_DG::EulerOperator< dim, degree, n_points_1d >, 35
n_stages
Parameters::Data_Storage, 23
Number
Euler_DG, 12
numerical_flux_type
Parameters::Solver, 27

operations.h, 56
operator∗
Euler_DG, 13
output_results
Euler_DG::EulerProblem< dim >, 41
output_tick
Parameters::Data_Storage, 25

Parameters, 15
parameters
EquationData::BoundaryData< dim >, 21
EquationData::ExactSolution< dim >, 18
EquationData::InitialData< dim >, 19
Euler_DG::EulerOperator< dim, degree, n_points_1d >, 35
Euler_DG::EulerProblem< dim >, 42
Euler_DG::EulerProblem< dim >::Postprocessor, 49

parameters.cpp, 54
parameters.h, 53
Parameters::Data_Storage, 21
beta, 24
beta_density, 24
beta_energy, 24
beta_momentum, 24
Data_Storage, 22
fe_degree, 23
fe_degree_Q0, 23
final_time, 25
function_limiter, 25
gamma, 24
M, 24
max_loc_refinements, 24
min_loc_refinements, 24
n_q_points_1d, 23
n_q_points_1d_Q0, 23
n_stages, 23
output_tick, 25
positivity, 25
prm, 25
read_data, 22
refine, 25
refine_indicator, 25
testcase, 23
type, 25
Parameters::Solver, 26
declare_parameters, 27
EulerNumericalFlux, 26
harten_lax_vanleer, 27
HLLC, 27
hllc_centered, 27
lax_friedrichs, 27
numerical_flux_type, 27
parse_parameters, 27
roe, 27
SLAU, 27
parse_parameters
Parameters::Solver, 27
pcout
Euler_DG::EulerProblem< dim >, 42
positivity
Parameters::Data_Storage, 25
Postprocessor
Euler_DG::EulerProblem< dim >::Postprocessor, 48
prm
Parameters::Data_Storage, 25
project
Euler_DG::EulerOperator< dim, degree, n_points_1d >, 31

quadrature
Euler_DG::EulerProblem< dim >, 45
quadratures
Euler_DG::EulerProblem< dim >, 45
quadratures_Q0
Euler_DG::EulerProblem< dim >, 46

rcell
Euler_DG::EulerProblem< dim >, 43
re_update
Euler_DG::EulerProblem< dim >, 46
read_data
Parameters::Data_Storage, 22
refine
Parameters::Data_Storage, 25
refine_indicator
Parameters::Data_Storage, 25
reinit
Euler_DG::EulerOperator< dim, degree, n_points_1d >, 29
roe
Parameters::Solver, 27
run
Euler_DG::EulerProblem< dim >, 39

set_body_force
Euler_DG::EulerOperator< dim, degree, n_points_1d >, 30
set_inflow_boundary
Euler_DG::EulerOperator< dim, degree, n_points_1d >, 29
set_subsonic_outflow_boundary
Euler_DG::EulerOperator< dim, degree, n_points_1d >, 30
set_supersonic_outflow_boundary
Euler_DG::EulerOperator< dim, degree, n_points_1d >, 30
set_wall_boundary
Euler_DG::EulerOperator< dim, degree, n_points_1d >, 30
sgn
Euler_DG, 13
shock_indicator
Euler_DG::EulerProblem< dim >, 44
SLAU
Parameters::Solver, 27
sol_aux
Euler_DG::EulerProblem< dim >, 42
solution
Euler_DG::EulerProblem< dim >, 42
solution_Q0
Euler_DG::EulerProblem< dim >, 42
subsonic_outflow_boundaries
Euler_DG::EulerOperator< dim, degree, n_points_1d >, 36
supersonic_outflow_boundaries
Euler_DG::EulerOperator< dim, degree, n_points_1d >, 36

tcell
Euler_DG::EulerProblem< dim >, 44
testcase
Parameters::Data_Storage, 23
time
Euler_DG::EulerProblem< dim >, 47
time_step

Euler_DG::EulerProblem< dim >, 47
timer
Euler_DG::EulerOperator< dim, degree, n_points_1d >, 35
Euler_DG::EulerProblem< dim >, 46
tmp_solution
Euler_DG::EulerProblem< dim >, 42
tmp_solution_Q0
Euler_DG::EulerProblem< dim >, 42
triangulation
Euler_DG::EulerProblem< dim >, 43
type
Parameters::Data_Storage, 25

update
Euler_DG::EulerProblem< dim >, 41

value
EquationData::BoundaryData< dim >, 21
EquationData::ExactSolution< dim >, 18
EquationData::InitialData< dim >, 19
vanAlbada
Euler_DG, 12

wall_boundaries
Euler_DG::EulerOperator< dim, degree, n_points_1d >, 36