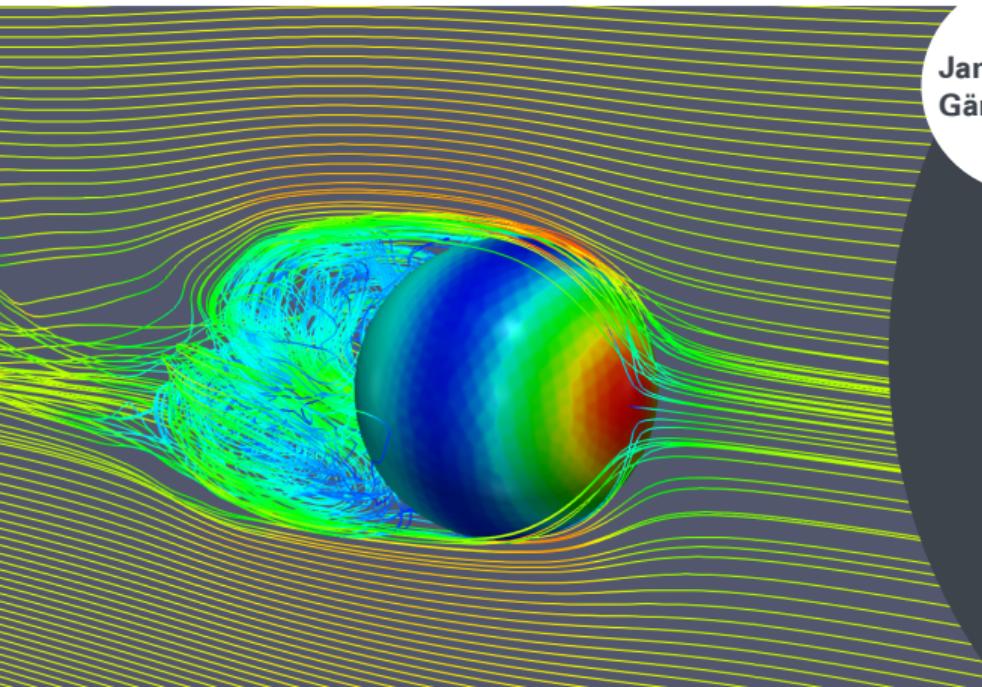


Universität Stuttgart



Jan Wilhelm  
Gärtner

**WENO Framework**

# Contributors



Universität Stuttgart

The **Weighted Essentially Non Oscillating (WENO)** scheme as implemented in this repository is originally written by,

**Tobias Martin**

**tobimartin2@googlemail.com**

Further main contributions are written by,

**Jan Wilhelm Gärtner**

**jan.gaertner@outlook.de**

The theory and implementation technique is based on several different authors mentioned within the attached Master Thesis. One main resource is the work of Pringuey [1].

# Motivation

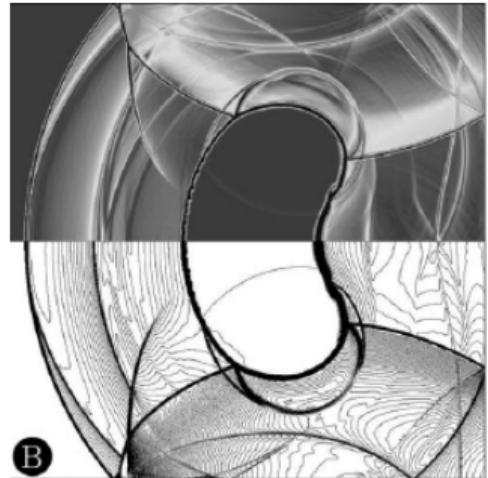
Simulating flows which contain discontinuities such as,

- shocks in supersonic flows,
- interfaces of two fluids

is a challenging task and requires numerical schemes that can give a high accuracy and yet, have enough dissipation around the discontinuity to avoid Gibbs oscillations.

To solve this conflicting requirements Weighted Essentially Non Oscillating (WENO) schemes have been developed that give a high accuracy within smooth regions and increase the dissipation around discontinuities.

Even though WENO schemes are widely used and have been established way over 20 years no WENO scheme implementation is available for OpenFOAM! Further, most implementations work on structured grids or only for finite difference methods. Here, we present a version that can handle any kind of mesh and can easily be included into any solver in OpenFOAM.

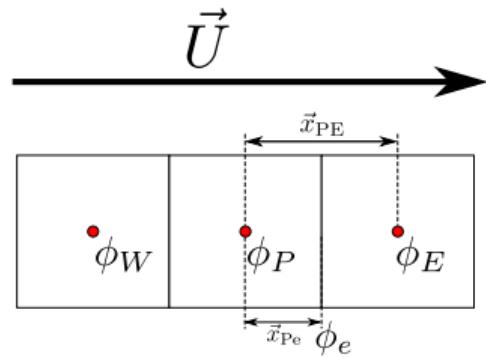


**Figure:** Shock wave interacting with helium bubble at  $\text{Ma}=1.22$  using a WENO scheme [2].

# Numerical Schemes Overview

In OpenFoam several numerical schemes are available for the surface interpolation of the cell centers onto the cell faces. Yet, schemes to resolve discontinuities, such as shocks, with a high order are not present. In the following important schemes of OpenFOAM are listed,

Scheme	Discretization	Accuracy
linear	$\phi_e = \frac{\vec{x}_{Pe}\phi_P + \vec{x}_{eE}\phi_E}{\vec{x}_{PE}}$	2nd order
upwind	$\phi_e = \phi_P$	1st order
linearUpwind	$\phi_e = \phi_P + \vec{x}_{Pe} \nabla \phi$	Up to 2nd order
WENO	$\phi_e = \frac{1}{ V } \int_V \phi(\vec{x}) dx dy dz$	higher order



The WENO scheme can reach potentially 3rd order of accuracy or higher depending on the polynomial representation of  $\phi(\vec{x})$ .

# **WENO**

## **Reconstruction**

### **Method**

**3**

## Polynomial and Degree of Freedom

The WENO schemes replaces the cell centered value  $\phi_i$  by a polynomial representation,

$$\phi_i = \frac{1}{|V_i|} \int_{V_i} \phi_i(\vec{x}) dx dy dz = \underbrace{\frac{1}{|V'_i|} \int_{V'_i} p_i(\vec{\xi}) d\xi d\eta d\zeta}_{\text{reference space}}, \quad (1)$$

where the cell is transformed into a reference space  $\vec{x} = \vec{x}(\xi, \eta, \zeta)$  to avoid scaling errors.  
The polynomial  $p_i(\xi)$  is represented by a set of basis functions,

$$p_i(\xi) = \phi_i + \sum_{k=1}^K a_k \Omega_k(\xi), \quad (2)$$

where  $K$  is the degree of freedom which depends on the order of the polynomial  $r$ ,

$$K = \frac{(r+1)(r+2)(r+3)}{6} - 1. \quad (3)$$

This means that for a order of 3 already 19 coefficients  $a_k$  are needed!

## Weighted Combination of Polynomials

To achieve a smooth solution a weighting of the polynomials is performed,

$$p_i(\xi) = \sum_{m=0}^N \omega_m p_m(\xi), \quad (4)$$

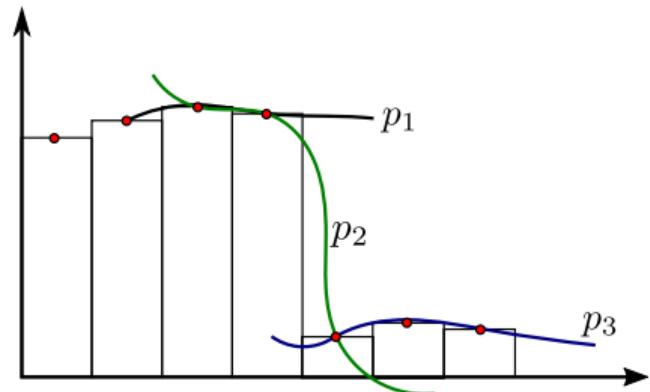
where  $N$  is the number of stencils and with the non linear weights,

$$\omega_m = \frac{\gamma_m}{\sum_{m=0}^N \gamma_m}. \quad (5)$$

Here  $\gamma_m$  is defined as,

$$\gamma_m = \frac{d_m}{(\epsilon + \mathcal{I}_{S,m})^p}. \quad (6)$$

$\epsilon$  is a small value to avoid division by zero and  $\mathcal{I}_{S,m}$  is the smoothness indicator function for the stencil  $m$ .  $d_m$  is used to increase the weight of the central stencil.



## Polynomial Basis Functions

The WENO polynomial  $p_{\text{WENO}}(\xi)$  can now be expressed as,

$$p_{\text{WENO}}(\xi) = \sum_{m=0}^N \omega_m \left( \phi_i + \sum_{k=1}^K a_k^{(m)} \Omega_k(\xi) \right). \quad (7)$$

Rearranging the equation leads to,

$$p_{\text{WENO}}(\xi) = \phi_i + \sum_{k=1}^K \left( \sum_{m=0}^N \omega_m a_k^{(m)} \right) \Omega_k(\xi) = \phi_i + \sum_{k=1}^K \tilde{a}_k \Omega_k(\xi) \quad (8)$$

The basis functions  $\Omega_k$  are only dependent on the geometry and can be pre-calculated with,

$$\Omega_k(\xi) = \Psi_k(\xi) - \frac{1}{|V'_i|} \int_{V'_i} \Psi_k(\xi) d\xi d\eta d\zeta. \quad (9)$$

The basis function  $\Psi_k(\xi)$  is a result of a Taylor expansion around the center of  $V_i$  and is,

$$\Psi_k(\xi) = (\xi - \xi_i)^n (\eta - \eta_i)^m (\zeta - \zeta_i)^o. \quad (10)$$

## Calculate the Coefficients $a_k$

Evaluating the WENO polynomial of Eq. (8) for an adjacent cell  $j$  gives,

$$\phi_j = \frac{1}{|V'_j|} \int_{V'_j} p_i(\xi) d\xi d\eta d\zeta \quad (11)$$

$$\phi_j - \phi_i = \sum_{k=1}^K a_k^{(m)} \left( \frac{1}{|V'_j|} \int_{V'_j} \Omega_k(\xi) d\xi d\eta d\zeta \right) \quad (12)$$

leading to an equation system of,

$$\phi_j - \phi_i = \sum_{k=1}^K \mathcal{A}_{jk} a_k^{(m)} \quad (13)$$

The matrix  $\mathcal{A}$  only depends on known geometrical integrals and can be precomputed.

**Storing these matrices are the cause of the huge memory demand of this scheme. For a third order WENO scheme already 19x38 entries have to be stored for each stencil and cell. Leading to a memory demand of 40GB for 1 million cells. If no reduction is possible, see implementation section!**

## WENO Upwind Scheme

The presented equations can now be easily included into the corrected upwind scheme, similar as the linearUpwind scheme of OpenFOAM. Using the definition of the flux at one face  $F$ ,

$$\int_{F_l} \vec{n}_l \vec{u} \phi dF_l = \underbrace{\vec{u} \vec{n}_l F_l}_{\text{vol. flux } Q} \frac{1}{F'_l} \int_{F'_l} \phi dF'_l \quad (14)$$

$$= Q \left[ \phi_i + \underbrace{\frac{1}{F'_l} \left( \sum_{k=1}^K \tilde{a}_k \int_{F'_l} \Omega_k(\xi) dF'_l \right)}_{\text{explicit correction}} \right] \quad (15)$$

# **OpenFoam**

## **Implementation**

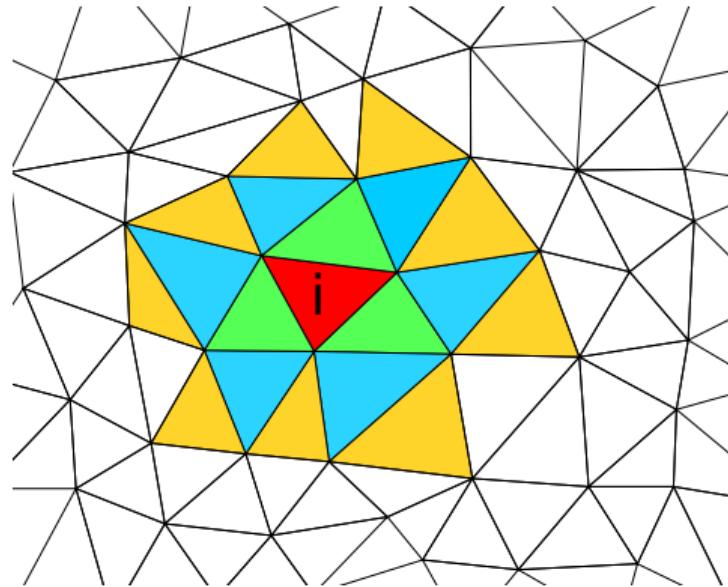
**4**

## Creating Stencil List

The coefficients  $\alpha_k$  are calculated on a compact set of stencils surrounding the current cell. The generation of this stencil list is done in three steps.

- 1) A big central stencil with n-layers of surrounding cells is generated.
- 2) Split the central stencil into several sectorial stencils. A sector typically corresponds to a face of the current cell. Thus, for a tetrahedral cell 4 faces equal 4 sectorial stencil plus the central stencil.
- 3) Cut the sectorial stencil list to the necessary size

The stencils are hereby stored as a list of lists of cell IDs.



# Matrix Condition

The matrix system,

$$\mathcal{A}x = b,$$

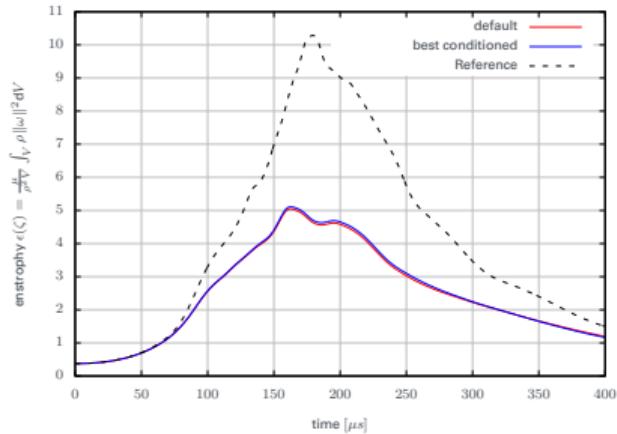
accuracy depends on the condition of  $\mathcal{A}$ .

## Find Best Conditioned Matrix

Add iteratively new cells to the matrix until the condition of  $\mathcal{A}$  goes up again. Return the best conditioned matrix system with the upper limit of  $2K$  cells, where  $K$  is the degree of freedom.

## Performance increase:

There is only a small performance increase. However, memory concerns are here the main driving point, see next slide.



**Figure:** Enstrophy of a Taylor Green vortex test case with once the default settings and the best conditioned matrix option.

# Storing Inverse of $\mathcal{A}$

## Problem:

Storing the inverse of the matrix system  $\mathcal{A}$  for each stencil in each cell requires a lot of memory. In many cases meshes are not completely irregular and thus as  $\mathcal{A}$  only depends on geometry information the memory demand can be reduced.

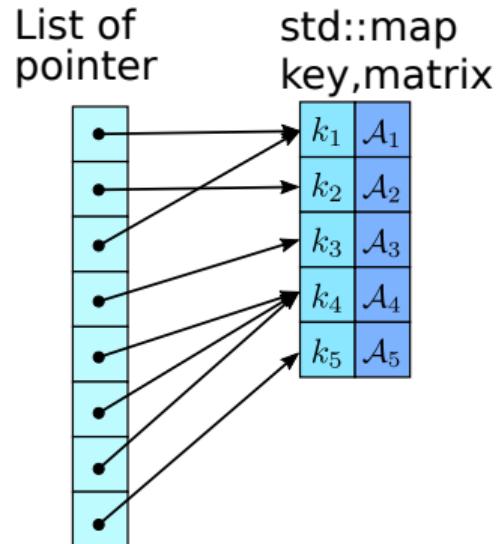
## Matrix Data Bank System

To efficiently store the inverse matrices for all cells a new matrix data bank system is implemented. For each cell and each stencil only a pointer is stored, which references a matrix within the data bank implemented as a `std::multimap`.

## Memory Reduction

For a 2D (400 cells) and 3D (64 000 cells) case using a regular grid of tetrahedrons, the memory demand can be substantially reduced:

Case	Memory Total [MB]	Matrices $\mathcal{A}^+$ [MB]	Relative Reduction
2D no matrix DB	93.8	65.9	-
2D with matrix DB	35.5	4.8	62.2%
3D no matrix DB	689.6	500	-
3D with matrix DB	306.9	93	55.5%
3D best conditioned	236.1	57.3	65.8%



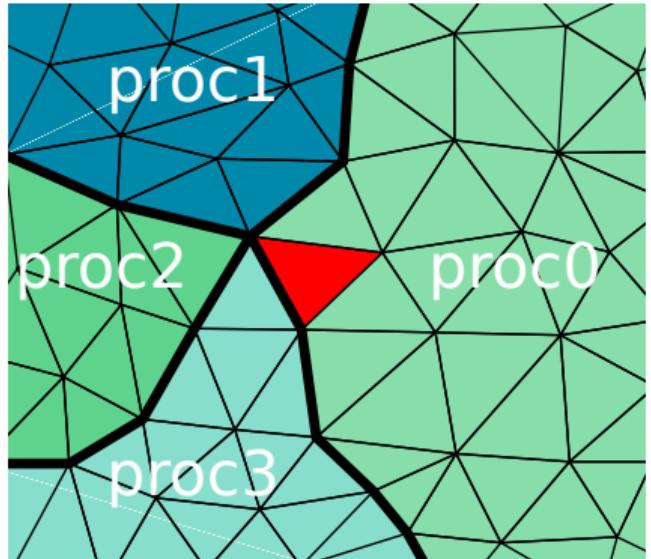
# Parallelization in OpenFOAM

Parallelization of the WENO scheme is not as straight forward as for any other OpenFOAM scheme. In difference to other schemes it

- requires cell node values of more than the patch neighbor cell,
- requires information of processors that do not have a common boundary.

**Example:** The red marked cell in processor 4 requires the cell node values of processor 1, 2 and 3. Getting the cell nodal values raises following challenges:

- The light blue marked cells are not accessible through the patches as they extend further into the domain.
- Processor 1 does not share a boundary with processor 4 and thus cannot be found through the patch neighbor information of OpenFOAM



# Parallelization in OpenFOAM

## Pre-Processing:

The WENO scheme is solved for a decomposed domain with the following steps:

- Each processor reconstructs a regional mesh of itself and neighboring processors.  
Implemented in class `globalFvMesh`
- Collecting the stencils for the processor local cells on the global mesh and storing the global cell ID
- Correcting the global cell ID with the local cell ID for cells within the own processor. Other cells are marked as non-local halo cells.
- Map the correct processor ID to the halo cells
- Store a list of cells required by other processors as well as the cells required by your own stencil list

## Run-Time:

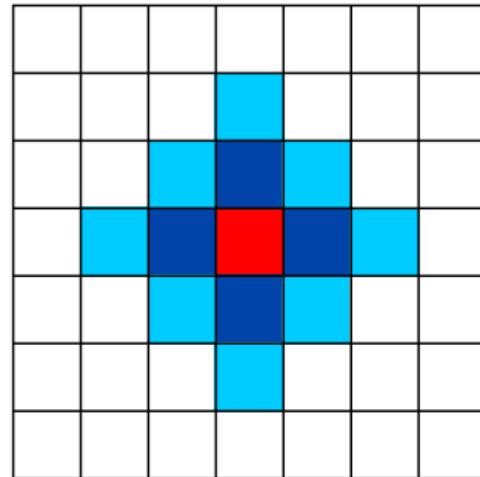
- At run time the cell nodal value of cells required by other processors are send and the halo cell data is received.
- Solve the matrix system of Eq. (13) to calculate the coefficients  $a_k$
- Calculate the weighting factors  $\omega$  and determine the weighted coefficients  $\tilde{a}_k$ .
- Calculate the explicit correction for the WENO upwind fit.

# Reconstruct Regional Mesh

## Find Neighbor processor

Every processor reconstructs the mesh of its own and neighboring processors. This is done in four steps:

- 1) Get the direct processor neighbors through the patch neighbor function, marked in dark blue.
- 2) Collect the neighbors of these processors, called second neighbors, marked in light blue.
- 3) Potentially repeat step 2 to get enough processors that each processor reconstructs the same number of meshes. This is important to avoid MPI blocking.
- 4) The processor meshes are now read from their constant folder and combined to one regional mesh for each processor.



## Memory Demand

The memory demand for reconstructing the regional mesh depend on the mesh size and decomposition. The memory demand differs also for 3D or 2D mesh.

---

**2D** Maximum 12 processor meshes are reconstructed for each processor.

**3D** Maximum 32 processor meshes are reconstructed for each processor.

---



Universität Stuttgart



Jan Wilhelm Gärtner

Institut für Technische Verbrennung, Universität Stuttgart  
Herdweg 51, 70174 Stuttgart, Germany

E-Mail: [jan.gaertner@outlook.de](mailto:jan.gaertner@outlook.de)

Phone: +49-711-685 674 69

## References

- [1] Thibault Roland Christophe Maurice Pringuey. *Large eddy simulation of primary liquid-sheet breakup*. PhD thesis, University of Cambridge, 2012.
- [2] Eric Johnsen and Tim Colonius. Implementation of weno schemes in compressible multicomponent flow problems. *Journal of Computational Physics*, 219(2):715–732, 2006.