ORION

a C++ framework for numerical PDE solving

Jaroslav Fořt, Jan Karel, David Trdlička, **Matěj Klíma**, Lukáš Hájek

Department of Technical Mathematics
Faculty of Mechanical Engineering
Czech Technical University in Prague

15.3.2022

ORION is

- A C++ numerical library developed at Department of Tech. Math.
- Originially a solver for turbulent flow problems
- Later split into general **common** part and applications
- Aspiring to provide blocks from which the user can build 3D finite volume solvers for diverse PDE systems without having to code the problem-independent parts of the algorithm
- Also providing tools usable for other 2D/3D meshed numerical methods (mesh representation, data structures, IO etc.)
- Using MPI parallelism in its data structures

History

- Started in 2008 as a joint project with Université Sorbonne Paris Nord
- Originally J. Fořt and J. Karel
- The development soon shifts to Prague
- Formerly involved – V. Šíp and J. B. Montavon
- Team gradually grows to current size
- Around 2020 new applications are built on top of the common part

Current applications running on ORION

Implicit finite volume solver:

- **fluid *(J. Karel and L. Hájek)* – implicit solver for turbulent flow (RANS/DES)**
- streamer (D. Trdlička) – simulating electrical discharge in cold plasma – *developed at the same time as the RANS solver, now being adapted to current version of Orion libraries*

Other numerical methods:

- lag3d *(M. Klíma)* – lagrangian (ALE) gas dynamics – collaboration with R. Liska nad M. Kuchařík (KFE FJFI ČVUT)
- Possible collaboration with J. B. Clément, student projects?

Programming concepts used in ORION

Template programming

- Functions/classes with generic data types
- Instantiated at time of compilation
- No computational overhead
- Not possible to "hide" source code

Declaration:

```
template<typename T, int N>
class Vars
{
    ...
};
```

T ... data type

N ... dimension

Instantiation:

```
class CompressibleVars :  public Vars<double,5>
```

Object-oriented programming

- Classes – categories of objects – data structures + functions

Inheritance

- Prevents code repetition – features re-used in derived classes

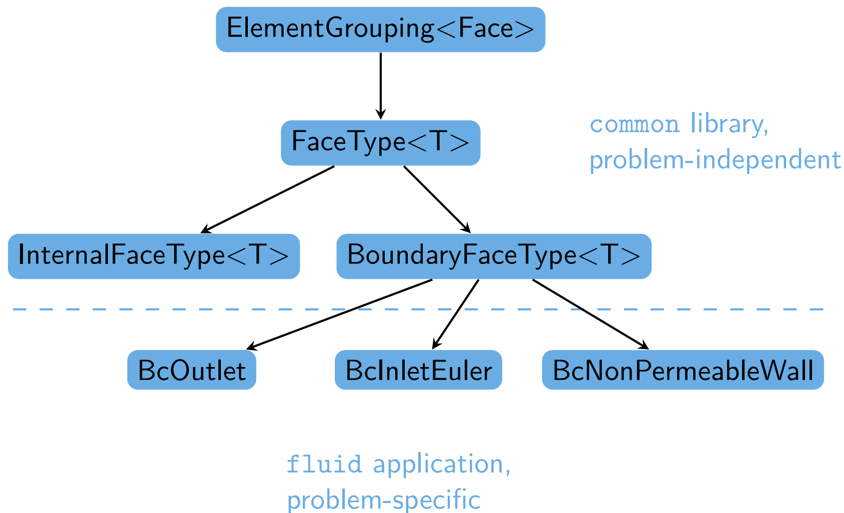
Encapsulation

- Access limits on variables/functions

Polymorphism

- Derived class objects can be treated as objects of a base class

Example class hierarchy: boundary conditions



Polymorphic classes

- General interface $\Longrightarrow$ many implementations
- Which implementation is used – resolved at runtime
- Resolving types similarly expensive as if statements

Polymorphic pointer:

```
std::shared_ptr<FaceType<T>> bc;
```

Can store any derived class:

```
class NsInternalFaceType : public FaceType<T>
class BcInletEuler : public FaceType<T>
class BcNonPermeableWall : public FaceType<T>
...
```

Usage example:

```
bc = new BcInletEuler(type,loc,g);

bc->setGhostCellsFromCenter(field);
```

- Replaces messy if/select statements

The ClassRegistry paradigm

- Makes code easily extensible
- Connects polymorphic classes with input parameters
- Selects which class is created based on a pre-defined keyword (loaded eg. from input file)

```
typedef std::function<std::shared_ptr<FaceType<T>>(int,int,const Grid&)> FaceTypeCreator;

auto faceTypeMap = ClassRegistry<FaceTypeCreator>({
    {INTERNALFACETYPE,   NsInternalFaceType::Create},
    {HALOFACETYPE,       NsHaloFaceType::Create},
    {BCNONPERMEABLEWALL, BcCompressibleNonPermeableWall::Create},
    {BCDONOTHING,        NsInactiveFaceType::Create},
    {BCINLETEULER,       BcInletEuler::Create},
    ...
});

bc = faceTypeMap(icType)(type,loc,grid);
```

Adding a new boundary condition:

1. Define and implement a new BC class
2. Add the class creator function to ClassRegistry

ORION finite volume solver components

(review of tools in the `common` library)

Governing equations

- Flux (convective/dissipative) and source terms:

$$\frac{\partial W}{\partial t} + \nabla \cdot F(W, \nabla W) = Q(W)$$

- Explicit discretization:

$$\frac{\Delta W_i}{\Delta t} = -Res(W^n)_i = -\sum_{j \in N(i)} \frac{F_{ij}(W_i^n, W_j^n, (\nabla W^n)_{ij})}{V_i} \cdot \mathbf{S}_{ij} + \frac{Q(W_i^n)}{V_i}$$

- Implicit discretization requires term values and jacobians:

$$\left(\frac{I}{\Delta t} + \frac{\partial Res(W^n)_i}{\partial W_j} \right) \Delta W_j = -Res(W^n)_j$$

- Numerical fluxes, jacobians, BC, IC, mesh – problem-specific
- The rest of the method can be automated

ORION finite volume solver components

CompressibleVars : public Vars<double,5> $(\rho, \rho u_x, \rho u_y, \rho u_z, e)$

ReconField<CompressibleVars> W_i, $(\nabla W)_i$

Grid

FaceType<CompressibleVars>
(handles BCs and mesh interfaces)

ConvectiveFlux
$F_C(W_l, W_r)$

ViscousFlux
$F_D(W_l, W_r, (\nabla W)_{lr})$

SourceTerm
$Q(W_i)$

PetSc

SparseLinearSystem $A_{ij}\,\Delta W_j = -Res(W)_j$

Stepper Δt

Solver

Variable vectors

CompressibleVars : public Vars<double,5> $(\rho, \rho u_x, \rho u_y, \rho u_z, e)$

ReconField<CompressibleVars> W_i, $(\nabla W)_i$

Grid

FaceType<CompressibleVars>
(handles BCs and mesh interfaces)

ConvectiveFlux $F_C(W_l, W_r)$

ViscousFlux $F_D(W_l, W_r, (\nabla W)_{lr})$

SourceTerm $Q(W_i)$

PetSc

SparseLinearSystem $A_{ij}\,\Delta W_j = -Res(W)_j$

Stepper Δt

Solver

Vars class

- N independent variables
- Static array wrapper
- Overloaded arithmetic operators
- Multiplication/division by constant
- Derived class can define methods to calculate dependent variables (eg. pressure from EOS)

common library

```
template <typename T, int N>
class Vars<T,N> {
protected:
    T val[N];
    ...
};
```

fluid code

```
class CompressibleVars
    : public Vars<double,5> {
public:
    double & rho = val[0];
    Vector3_ref<double> rhoU
        = {val[1],val[2],val[3]};
    double & e = val[4];
    ...
};
```

Grid class

CompressibleVars : public Vars<double,5> $(\rho, \rho u_x, \rho u_y, \rho u_z, e)$

ReconField<CompressibleVars> W_i, $(\nabla W)_i$

Grid

FaceType<CompressibleVars>
(handles BCs and mesh interfaces)

ConvectiveFlux
$F_C(W_l, W_r)$

ViscousFlux
$F_D(W_l, W_r, (\nabla W)_{lr})$

SourceTerm
$Q(W_i)$

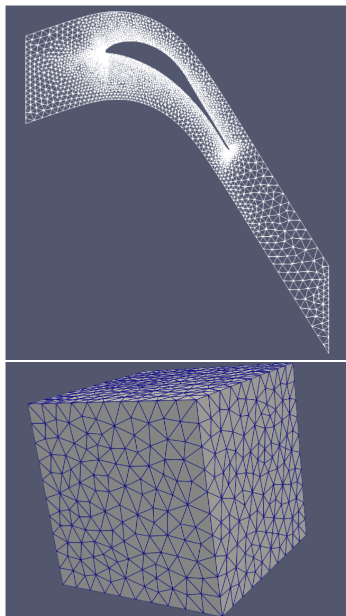
PetSc ↔ SparseLinearSystem $A_{ij} \, \Delta W_j = -Res(W)_j$

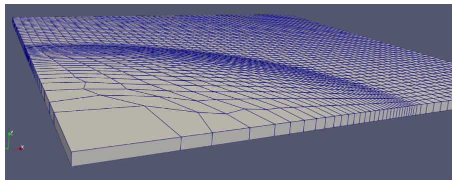
Stepper Δt

Solver

Grid class

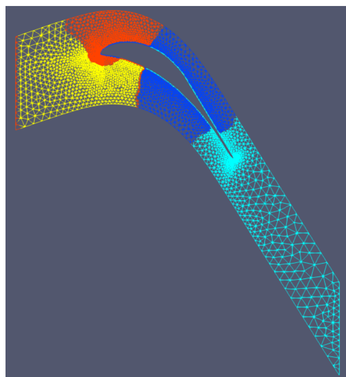
- 2D/3D Unstructured grid
- Full representation of neighbor indices
- Supported file formats: gmsh, .grd, OpenFOAM
- 2D $\implies$ 3D mesh extrusion
- Memory optimization for cartesian grids
- Separation of connectivity and geometrical data





Mesh parallelization

- METIS library for mesh partitioning
- Grid loading+partitioning done on master processor
- `GridTransfer` class – distributes local meshes to processors via MPI
- `Gridloc` class – mapping from global to local indices



- Halo cells – special ghost cells representing BC on subdomain boundaries, with data transfer after each time step
- Local/cross-subdomain periodic boundaries

Field data structures

CompressibleVars : public Vars<double,5> $(\rho, \rho u_x, \rho u_y, \rho u_z, e)$

ReconField<CompressibleVars> W_i, $(\nabla W)_i$ ← Grid

FaceType<CompressibleVars>
(handles BCs and mesh interfaces)

ConvectiveFlux
$F_C(W_l, W_r)$

ViscousFlux
$F_D(W_l, W_r, (\nabla W)_{lr})$

SourceTerm
$Q(W_i)$

PetSc ↔ SparseLinearSystem $A_{ij}\,\Delta W_j = -Res(W)_j$

Stepper Δt

Solver

Field data structures

- `CellField<T>`, `NodeField<T>` etc. for spatially discrete quantities defined on mesh elements
- Parallel gather/scatter operations for transferring quantities to/from master processor (for I/O, mesh adaptation etc.)
- Parallel data transfer on halo cells (for subdomain boundaries)
- `ReconField<T>` enables also gradient calculation (w/ slope limiters) and reconstruction in nodes
  (probably will be splitted into separate classes in the future)

Mesh element grouping, `FaceType`

CompressibleVars : public Vars<double,5> $(\rho, \rho u_x, \rho u_y, \rho u_z, e)$

ReconField<CompressibleVars> W_i, $(\nabla W)_i$

Grid

FaceType<CompressibleVars>
(handles BCs and mesh interfaces)

ConvectiveFlux $F_C(W_l, W_r)$

ViscousFlux $F_D(W_l, W_r, (\nabla W)_{lr})$

SourceTerm $Q(W_i)$

PetSc ↔ SparseLinearSystem $A_{ij}\,\Delta W_j = -Res(W)_j$

Stepper Δt

Solver

Mesh element grouping, `FaceType`

- There is often different behavior for internal/boundary mesh elements
- Iteration over all elements with subsequent determination of element type is expensive
- More effective - lists of element ids corresponding to each element type
- General class `ElementGrouping<T>`
- `FaceType` is a specialization for mesh faces, connects to `ReconField` to obtain reconstruction from L/R side of a face

Example: inlet BC in finite volume code

```
class BcInletEuler : public BoundaryFaceType<CompressibleVars>
{
public:
    BcInletEuler(int typeID, int locID, const Grid& g)
                         : BoundaryFaceType<CompressibleVars>(typeID,locID,g) {};
    ~BcInletEuler(){};
    static std::shared_ptr<FaceType<CompressibleVars>> Create(int typeID, int locID, const Grid& g)
                         { return std::make_shared<BcInletEuler>(typeID,locID,g); };
```

The following methods need to be implemented in .cpp file to define the BC behavior:

```
    void setGhostCell(const CompressibleVars & wl, CompressibleVars & wr, const Face& face);

    virtual Jacobian<CompressibleVars> boundaryJacobian(const Face& face, const CompressibleVars& wl) cons

    virtual map<string, double> getInternalParams();

    void read(map<int, vector<string> > & dataFile);
    void print(std::ostream & os);
private:
    double p0;
    double rho0;
    double entryAngle;
};
```

PDE terms calculation

CompressibleVars : public Vars<double,5> $(\rho, \rho u_x, \rho u_y, \rho u_z, e)$

ReconField<CompressibleVars> $W_i, (\nabla W)_i$

Grid

FaceType<CompressibleVars>
(handles BCs and mesh interfaces)

ConvectiveFlux
$F_C(W_l, W_r)$

ViscousFlux
$F_D(W_l, W_r, (\nabla W)_{lr})$

SourceTerm
$Q(W_i)$

PetSc ↔ SparseLinearSystem $A_{ij}\,\Delta W_j = -Res(W)_j$

Stepper Δt

Solver

Contrib class

```
// T .. Variable data type (eg. CompressibleVars)
// F .. Flux type     (eg. CompressibleVars for explicit, CompressibleVars + Jacobian for implicit...)
// R .. Residuum type (eg. Cell_field<..> for explicit method, SparseLinearSystem<..> for implicit...)
// U .. FaceType Creator (type of the constructor wrapper function for the specified FaceType)

template <typename T, typename F, typename R, typename U>
class Contrib {
public:
```

The computation of flux value on a single mesh face (user must implement)

```
    virtual F computeOnFace(int face_id, const FaceType<T>& ft, const ReconField<T>& fld) = 0;
```

The addition of the flux terms to the residuum (user must implement)

```
    virtual void distributeToCells(F& val, int face_id, const FaceType<T>& ft, const ReconField<T>& fld,
                                   R& res, const FaceTypeList<T,U>& bcs) = 0;
```

Compute fluxes on all faces and distribute to the residuum

```
    void computeOnAllFaces(const FaceTypeList<T, U>& bcs, const Grid& grid,
                           const ReconField<T>& fld, R& r);
}
```

ImplicitFlux class

Specialization of Contrib class for implicit solvers

```
enum BoundaryJacobianLocation { FACE_BOUNDARY_JACOBIAN, CELL_BOUNDARY_JACOBIAN };
template <typename T, typename U, BoundaryJacobianLocation bjl>
class ImplicitFlux : public Contrib<T,VarWithJacobian<T>,SparseLinearSystem<T>,U> {
```

The computation of flux value + jacobian on a face (user must implement)

```
virtual VarWithJacobian<T> computeOnFace(int face_id, const FaceType<T>& ft, const ReconField<T>& fld) = (
```

The addition of flux terms to the linear system is implemented in this class

```
void distributeToCells(VarWithJacobian<T>& val,
                       int face_id,
                       const FaceType<T>& ft,
                       const ReconField<T>& fld,
                       SparseLinearSystem<T>& res,
                       const FaceTypeList<T, U>& bcs);

...
```

SourceTerm class

```
template <typename T>
class SourceTerm {
public:
```

The computation of source term value + jacobian in a mesh cell (user must implement)

```
virtual T value(int cell_id, const Grid& grid, const ReconField<T>& fld) const = 0;
virtual Jacobian<T> jacobian(int cell_id, const Grid& grid, const ReconField<T>& fld) const = 0;
```

Compute source terms in all cells and distribute to the linear system

```
virtual void computeOnAllCells(const Grid& grid, const ReconField<T>& fld, SparseLinearSystem<T>& res) co
};
```

Remaining solver components

CompressibleVars : public Vars<double,5> $(\rho, \rho u_x, \rho u_y, \rho u_z, e)$

ReconField<CompressibleVars> W_i, $(\nabla W)_i$

Grid

FaceType<CompressibleVars>
(handles BCs and mesh interfaces)

ConvectiveFlux $F_C(W_l, W_r)$

ViscousFlux $F_D(W_l, W_r, (\nabla W)_{lr})$

SourceTerm $Q(W_i)$

PetSc

SparseLinearSystem $A_{ij}\,\Delta W_j = -Res(W)_j$

Stepper Δt

Solver

Remaining solver components

`Stepper` class

- Time stepping for parallel computation
- Includes tools for local time stepping
- Time step calculation itself must be defined by the user

`SparseLinearSystem` class

- Wrapper for a numerical linear algebra solver
- Currently using the PetSc library
- GMRES solver, Jacobi preconditioner

The solver itself is constructed by the user using the building blocks provided

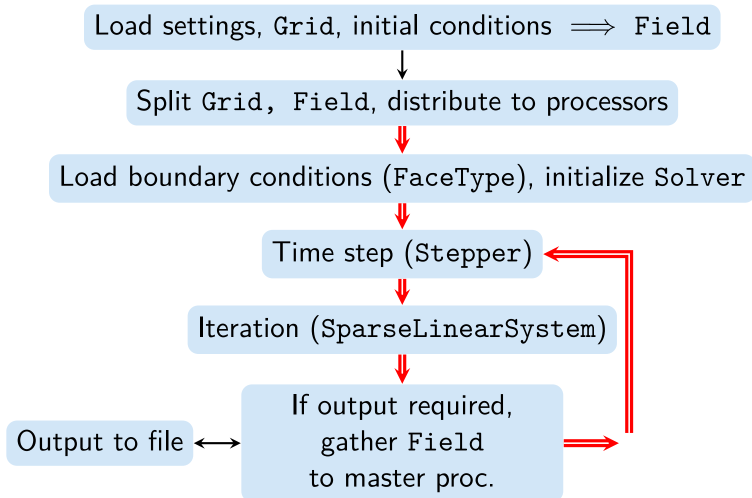
A solver for fluid dynamics based on ORION

(examples shown from the **fluid** code)

Turbulent flow modelling

- Stationary flow problems
- Reynolds-averaged Navier-Stokes (RANS) + turbulence model
- SST, TNT models ($k - \omega$), DES in progress
- Independent solvers for turbulent/compressible variables
- Exchange terms
- Separation of variables and boundary conditions

Algorithm flowchart

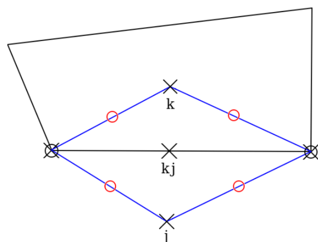


Input/output

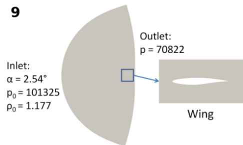
- Input – mesh, config, BC, IC files
- BCs specific for compressible/turbulence variables
- IC – problem specific, loaded using ClassRegistry
- Output data: .vtk – TecPlot, Paraview readable
- Output fields in volumes or interpolated to nodes
- Possible to save full snapshots for restarting computation

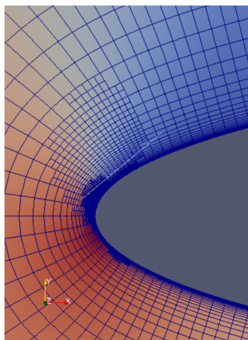
RANS equations

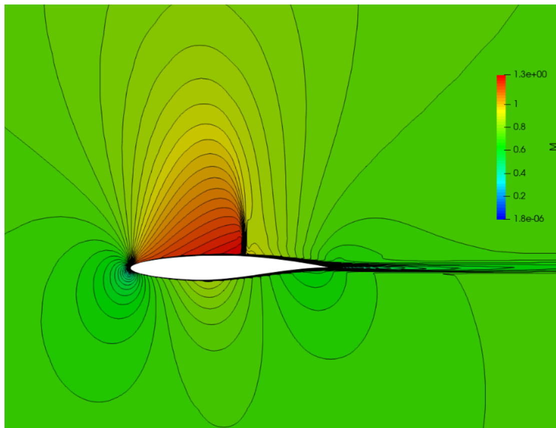
- Explicit solver (mainly for testing)
- Implicit solver with BDF, local time stepping
- Convective fluxes AUSM+up, HLLC with analytically derived Jacobians
- Least squares reconstruction + Barth-Jespersen limiter (in `common` library)
- Dissipative fluxes via diamond cell at faces



Results - RAE 2822



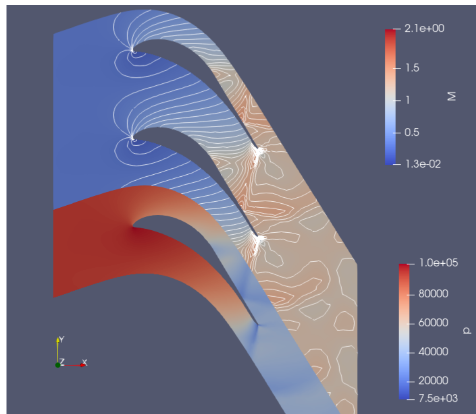


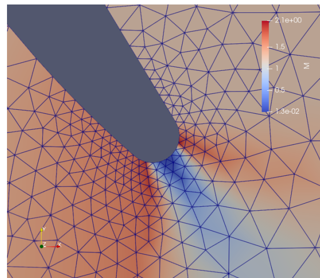


Transonic flow over RAE 2822 airfoil (Mach number plot)

AMR on leading edge in development

Results - SE1050





Transonic flow in SE1050 turbine cascade (Mach number/pressure)

More general usage of ORION

(examples shown from the **lag3d** code)

Non-finite volume solver

- Lagrangian method with generalised Lax-Wendroff type explicit scheme
- Reduced set of tools from the common library
- Utilising mainly mesh representation and parallelism
- Discrete variables in both cells and nodes

SpecificVars : public Vars<double,5> (ν, u_x, u_y, u_z, e)

CellField<SpecificVars> W_i

NodeField<SpecificVars> W_p

Grid

ElementGrouping<Node>
BCs treated at mesh nodes

Stepper Δt

Solver

Euler equations in Lagrangian form

Material derivative (transformation in Lag. coords):

$$\frac{d}{dt} = \frac{\partial}{\partial t} + \mathbf{u} \cdot \nabla,$$

Conservation laws:

$$\frac{d\nu}{dt} - \nu \, \nabla \cdot \mathbf{u} = 0,$$

$$\frac{d\mathbf{u}}{dt} + \nu \, \nabla p = 0,$$

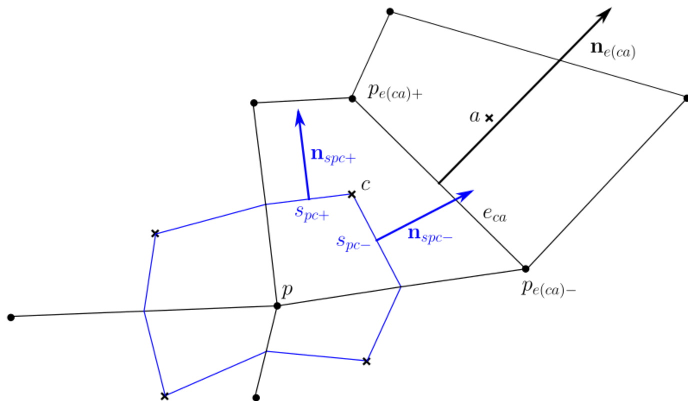
$$\frac{de}{dt} + \nu \, \nabla \cdot (p \, \mathbf{u}) = 0,$$

Mesh moves with the fluid $\implies$ mass conserved by design:

$$\frac{dx_p}{dt} = \mathbf{u}_p,$$

Reduced dual mesh representation

- Predictor – integration over dual cell
- Flux function $\mathbf{F}(\mathbf{w}) = (\mathbf{u}, -p\,\mathbf{I}, -p\,\mathbf{u})$



$$m_p \mathbf{W}_p^{n+1/2} = \sum_{c \in C(p)} m_{p,c} \mathbf{w}_c^n + \frac{\Delta t}{2} \mathbf{F}(\mathbf{W}_c) \cdot (\mathbf{n}_{spc+} + \mathbf{n}_{spc-})$$

Corrector + artificial dissipation

- Flux calculated from nodal values
- HLL-like dissipative term to prevent oscillations

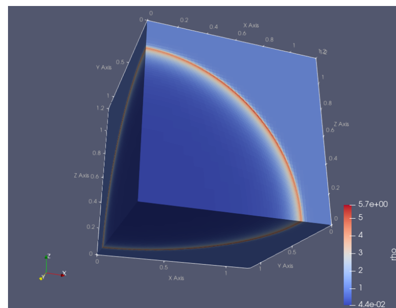
$$
\begin{aligned}
m_c \mathbf{W}_c^{n+1} = m_c \mathbf{W}_c^{n} \quad & + \Delta t \sum_{p \in P(c)} \mathbf{F}\left(\mathbf{W}_p^{n+1/2}\right) \cdot \frac{\mathbf{n}_e^{n+1/2} + \mathbf{n}_{e+1}^{n+1/2}}{2} \\
& + \Delta t \sum_{e \in E(c)} D_\tau \, \mathbf{D}\left(\mathbf{W}_c^n, \mathbf{W}_a^n\right),
\end{aligned}
$$

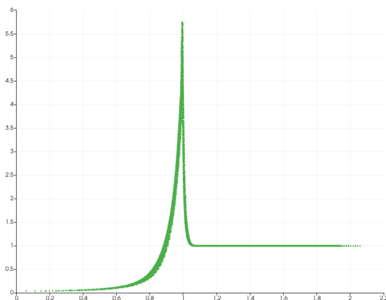
$$
D_\tau = \begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & \tau & 0 & 0 \\ 0 & 0 & \tau & 0 \\ 0 & 0 & 0 & \tau \end{pmatrix}, \quad \tau \in (1, 2),
$$

$$
\mathbf{D}\left(\mathbf{W}_c^n, \mathbf{W}_a^n\right) = \frac{z_{ca} z_{ac} \left| \mathbf{n}_{e(ca)} \right|}{z_{ca} + z_{ac}} \left(\mathbf{W}_a^n - \mathbf{W}_c^n\right)
$$

Results - "Sedov" test case

- Non-stationary shock propagation
- Conversion internal $\Longrightarrow$ kinetic energy
- Starting with equidistant mesh, $\epsilon_0 = 0.106384$ in 1 cell, $\rho_0 = 1.0$, $p_0 = 1.0e-6$

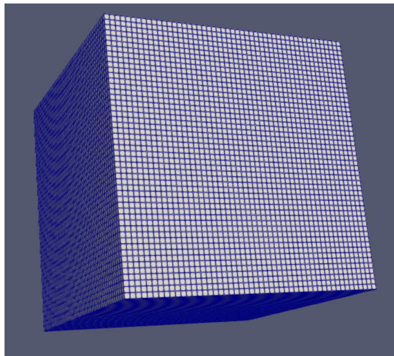


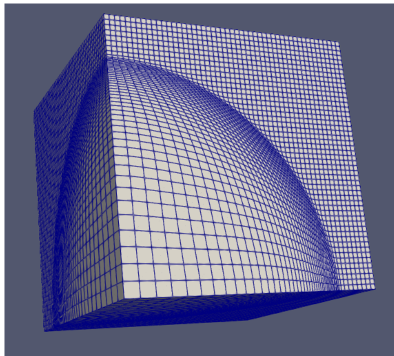


Density at $t = 1.0$, $N = 50^3$

Results - "Sedov" test case

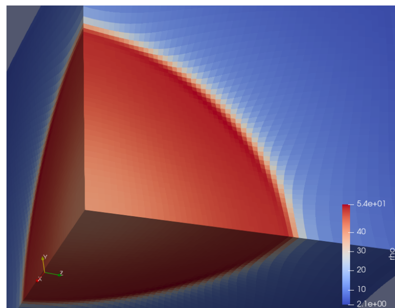
- Mesh movement:

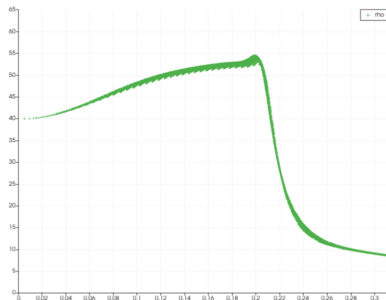




Results - "Noh" test case

- Implosion-driven shock
- Starting with equidistant mesh, $|\mathbf{u}_0| = 1.0$ directed radially to mesh center, $\rho_0 = 1.0$, $p_0 = 1.0e-6$





Density at $t = 0.6$, $N = 50^3$

Thank you for your attention