DNDSR

A C++17 / Python CFD Research Code

Compact Finite Volume · Variational Reconstruction

MPI · OpenMP · CUDA · pybind11

`v0.2.0`

CHAPTER 1

Opening

Motivation · feature set · positioning

Why another CFD code?

The unstructured-CFD design space sits awkwardly:

- **CG/CAD** — complex polymorphic topology, small compute per element (Blender, FreeCAD, Gmsh).
- **Deep-learning frameworks** — massive homogeneous arrays, simple fixed-width tensors (PyTorch, JAX).
- **Unstructured CFD needs both** — heterogeneous topology + dense numeric kernels.

Two dominant existing paradigms leak abstractions:

- **OpenFOAM-style** — `primitiveMesh` owns topology + geometry inside a monolithic class hierarchy; communication lives at the class level.
- **SU2-style** — polymorphic `CDualGrid` / `CVertex` objects carry per-entity geometric and solution state.

Every new communicated field forces an edit to the object model.

DNDS: Distributed Numerical Data Structure

> *"DNDS is dedicated to providing c-like random-access arrays without the concern of MPI communication. Higher-level abstraction is left for the caller."*
> — `docs/architecture/Paradigm.md:161`

```
// NOT this:
struct Face {
    real area;
    vec  cent;
    // …more fields…
};
std::vector<Face> faces;

// THIS:
std::vector<real> faceArea;
std::vector<vec>  faceCent;
// each manages its
// own MPI pattern.
```

DNDSR at a glance

Capabilities

- **Solvers** — Euler / N-S (2D/3D), SA-IDDES, k-ω RANS (Wilcox & SST), reactive `NS_EX`, realizable k-ε.
- **Numerics** — CFV + variational reconstruction (orders 1–3), 13 Riemann variants, ESDIRK / HM3 / BDF2, p-Multigrid, WBAP / CWBAP limiters.
- **Parallelism** — persistent MPI + OpenMP throughout; CUDA via `DeviceTransferable` CRTP; EulerP purpose-built GPU evaluator.
- **Bindings** — pybind11 for DNDS · Geom · CFV · EulerP; PEP-561 typed (`.pyi` auto-generated).
- **Config** — typed JSON + auto-generated JSON Schema (`--emit-schema`); unknown-key detection built in.

Solver executables

Executable	Model
`euler` / `euler3D`	Compressible Navier–Stokes
`eulerSA` / `eulerSA3D`	Spalart–Allmaras RANS (IDDES)
`euler2EQ` / `euler2EQ3D`	k-ω two-equation RANS
`eulerEX` / `eulerEX3D`	Reactive / multi-species

Each `app/Euler/euler*.cpp` is a **one-line** `main` that instantiates `DNDS::Euler::RunSingleBlockConsoleApp<Model>` — a template dispatch on the `EulerModel` enum.

Shared code path, eight binaries.

Project shape in numbers

Code

- **~72k** LOC C++ (346 files)
- **~1.9k** LOC Python (17 files)
- **6** C++ modules + header-only Solver
- **4** pybind11 extension modules
- **8** solver executables

Tests

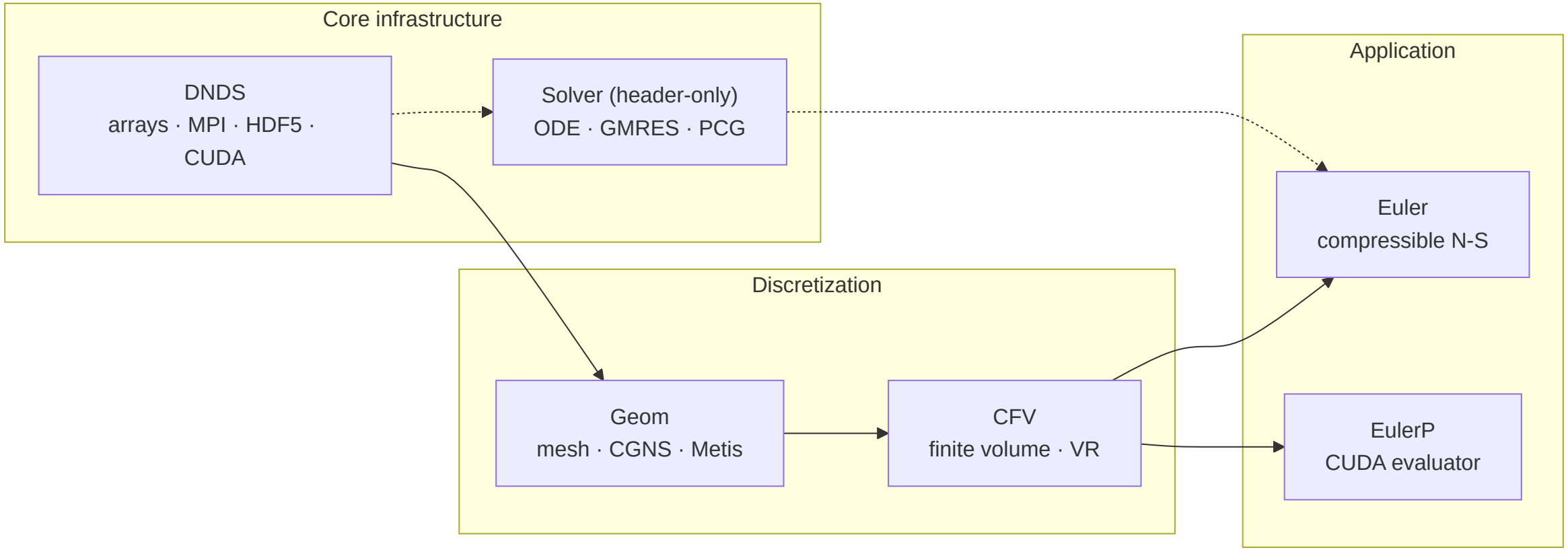
- **82** CTest registrations (29 executables)
- **np** ∈ **{1, 2, 4, 8}** for every MPI-aware test
- **~600** doctest test cases total
  - 249 DNDS · 193 Geom · 62 CFV
  - 62 Euler · 29 Solver
- **58** Python pytest functions (DNDS, CFV)
- **Metis seed = 42** → deterministic golden values

Docs

- **Sphinx + Breathe + Doxygen**
- Full class / call / include graphs (Graphviz)
- Incremental build < 1 s (no-op), ~2.5 min (full)
- Live at `cfdlab-thu.github.io/DNDSR`
- One-source Markdown for both engines via `doxygen_compat.py`

> 🧹 Clang-tidy milestone: the DNDS core module dropped from **24 597** → **1** diagnostic across 26 cleanup passes.

The one-slide map



Reading this graph. Every module depends only on those above it. `Solver` is header-only and depends only on `DNDS` data types — the Krylov and ODE code knows nothing about CFD. `EulerP` is a parallel-track CUDA evaluator alongside `Euler`, reusing `CFV` but replacing the flux/limiter pipeline with device-callable scalar loops.

From zero to a running solver in six commands

```
# 1. Fetch code and submodules
git clone --recursive https://github.com/CFDLAB-THU/DNDSR.git && cd DNDSR

# 2. Build binary external libraries (HDF5, CGNS, Metis, ParMetis, zlib, ...)
cd external/cfd_externals && CC=mpicc CXX=mpicxx python cfd_externals_build.py && cd ../..

# 3. Fetch header-only libraries (Eigen, Boost, CGAL, fmt, pybind11, nanoflann, ...)
curl -L -o external/external_headeronlys.tar.gz \
  https://github.com/harryzhou2000/cfd_externals_headeronlys/releases/latest/download/external_headeronlys.tar.gz
cd external && tar -xzf external_headeronlys.tar.gz && cd ..

# 4. Configure with a preset
cmake --preset release-test        # Release + DNDS_BUILD_TESTS=ON

# 5. Build a specific solver
cmake --build build -t euler -j32

# 6. Run
mpirun -np 4 ./build/app/euler.exe cases/euler_config_IV.json
```

Presets available: `release-test`, `debug`, `cuda`, `ci`. Python path: `pip install -e .` uses `scikit-build-core` under the hood.

CHAPTER 2

Architecture

Arrays, MPI, ghosts, state machines

Six modules — responsibilities

Module	Directory	Role	LoC
DNDS	`src/DNDS`	MPI arrays, serialization (JSON + HDF5), profiling, CUDA, config	large
Geom	`src/Geom`	Unstructured mesh, CGNS I/O, Metis/ParMetis partitioning	large
CFV	`src/CFV`	Compact Finite Volume, Variational Reconstruction, limiters	medium
Euler	`src/Euler`	Compressible N-S, SA, k-ω, dual-time orchestration	large
EulerP	`src/EulerP`	Alternative CUDA-optimized evaluator	medium
Solver	`src/Solver`	ODE integrators + Krylov — **header-only**	small

> **Why the split.** Each layer depends only on those above. `Solver` depends on DNDS data types only — the Krylov and ODE code knows nothing about CFD; this is how the same `GMRES_LeftPreconditioned` works across Euler, VR's `uRec` system, and the k-ω equations. `EulerP` sits alongside `Euler`, reusing all of `CFV` but replacing the flux kernel with device-callable scalar loops.

Delayed abstraction ⇒ independent comm patterns

Different field genres need different communication patterns. **Baking them into one class forces a monolithic serializer.**

Fragile: combined struct

```
class Solution {
    real rho, ru, rv, rw, E;   // comm phase A
    real u, v, w, p, T;        // derived, no comm
    real rho_1, ru_1, rv_1,
         rw_1, E_1;             // comm phase B
public:
    void WriteStream(ByteStream &);
    void ReadStream (ByteStream &);
};
std::vector<Solution> sols;
```

A single `WriteStream` can't express *which* fields participate in *which* MPI phase — any new field requires editing both methods.

DNDSR: split by genre

```
ArrayDof<5, 1>           u;       // conservative now
ArrayDof<5, 1>           u_prev;  // previous snapshot
ArrayDof<2, 5>           grad_u;  // gradients, 2D × 5 vars
ArrayDof<DynamicSize,5> uRec;    // variable-order reconstruction
```

Each array owns its own `ArrayTransformer`. Ghost footprints and communication phases are **independent and composable**:

- `u` and `u_prev` share the same ghost map → `BorrowGGIndexing`.
- `uRec` may have a different row size — new MPI types, same map.
- `grad_u` lives in a larger halo for gradient stencils.

`Array<T, rs, rm>` — five layouts in one template

```
template <class T,
          rowsize _row_size = 1,            // fixed | DynamicSize | NonUniformSize
          rowsize _row_max  = _row_size,    // controls padding vs CSR
          rowsize _align    = NoAlign>
class Array;

enum DataLayout {
    ErrorLayout,        // invalid template combination (compile error)
    TABLE_StaticFixed,  // fixed width, compile-time
    TABLE_Fixed,        // fixed width, runtime (uniform across rows)
    TABLE_Max,          // padded variable rows, runtime max
    TABLE_StaticMax,    // padded variable rows, compile-time max
    CSR,                // flat buffer + pRowStart[n+1]
};
```

`ComputeDataLayout()` maps `(rs, rm)` → layout tag:

`_row_size`	`_row_max`	Layout	Use case
`>= 0`	—	`TABLE_StaticFixed`	Cell volume (1 real), Euler state (5 reals)
`DynamicSize`	—	`TABLE_Fixed`	VR coefficients (order decided at runtime)
`NonUniformSize`	`>= 0`	`TABLE_StaticMax`	Per-face node counts for a single element type
`NonUniformSize`	`DynamicSize`	`TABLE_Max`	Padded variable rows, runtime max
`NonUniformSize`	`NonUniformSize`	`CSR`	`cell2node`, `cell2cell`, wide-stencil adjacency

`rowsize = int32_t`. Sentinels: `DynamicSize = -1`, `NonUniformSize = -2`. Alignment stub exists but only `NoAlign` is implemented today.

CSR has two internal modes

Decompressed mode

`std::vector<std::vector<T>>` — one inner vector per row.

```
ArrayAdjacency<NonUniformSize, NonUniformSize> c2n;
c2n.Decompress();           // → vector<vector<index>>
for (index iCell = 0; iCell < nCell; ++iCell) {
    c2n.ResizeRow(iCell, /*width*/ vertexCount[iCell]);
    for (int k = 0; k < vertexCount[iCell]; ++k)
        c2n(iCell, k) = globalNodeId[iCell][k];
}
c2n.Compress();             // required before MPI
```

Use during mesh construction — rows grow incrementally.

Compressed mode

Flat `std::vector<T>` + `pRowStart[n+1]` index.

- O(1) row access via `pRowStart[i]`.
- Zero overhead after construction.
- **Required** before any MPI call or serialization.
- Retains row-resizing only through `Decompress()` → edit → `Compress()`.

ArrayView

A device-callable non-owning view (`ArrayView<T, rs, rm>`) implements `operator[]` and `at()` for every layout — this is what ships to the GPU.

> ⚠ **Element-type constraint:** `array_comp_acceptable<T>()` requires `std::is_trivially_copyable_v<T>` **or** `is_fixed_data_real_eigen_matrix_v<T>`. No `std::string` rows, no `std::vector` rows — it would break MPI.

`ArrayTransformer` — anatomy

Members

- `MPIInfo mpi;`
- `t_pArray father, son;`
- `pLGlobalMapping` — local row → global index
- `pLGhostMapping` — global index → local father+son
- `pPushTypeVec / pPullTypeVec` — cached `(rank, MPI_Datatype)`
- `PushReqVec / PullReqVec` — persistent request handles
- `pushDevice / pullDevice` — Host or CUDA

Two strategies

- `HIndexed` — `MPI_Type_create_hindexed` scatter/gather (default).
- `InSituPack` — contiguous pack buffers, `MPI_Isend/Irecv` on packed memory.

Chosen per-process via `MPI::CommStrategy::Instance().GetArrayStrategy()`.

Lifecycle

```
// Setup — all collective
trans.setFatherSon(father, son);
trans.createFatherGlobalMapping();
trans.createGhostMapping(pullIdxGlobal);   // pull-based
// or:
trans.createGhostMapping(pushIdxLocal, pushStarts); // push-based
trans.createMPITypes();                    // hindexed datatypes

// Persistent init
trans.initPersistentPull();                // MPI_Recv_init + Send_init
trans.initPersistentPush();                // reverse direction

// Hot loop — any number of times
for (step = 0; step < N; ++step) {
    trans.startPersistentPull();           // MPI_Startall
    computeFluxes(/* reads ghosts */);
    trans.waitPersistentPull();            // MPI_Waitall
}

// Cleanup
trans.clearPersistentPull();
trans.clearMPITypes();
```

Father / son addressing

```
index:   0 .......... fatherSize-1  | fatherSize ...... fatherSize+sonSize-1
         └──────── owned (father) ──────┘ └─ ghost (son, copies from other ranks) ─┘
```

- father owns data — writes are legal
- son mirrors remote data — writes are ignored after the next pull
- operator[](i) routes to father or son by index range

Pull = father → son (read ghosts)

```
trans.initPersistentPull();
trans.startPersistentPull();      // non-blocking
// ... overlap computation ...
trans.waitPersistentPull();
```

Typical in flux-evaluation loops: read neighbor cell values.

Push = son → father (accumulate)

```
trans.initPersistentPush();
trans.startPersistentPush();
trans.waitPersistentPush();
```

Typical in node-based FEM-style assembly: accumulate partial sums from ghost copies back into the father.

> **Sharing ghost structure across arrays.** `BorrowGGIndexing(primary)` skips the expensive collective `createFatherGlobalMapping + createGhostMapping` phase; only `createMPITypes()` is rebuilt because the MPI datatypes depend on element size.

Typed wrappers: `ArrayDerived`

Each derived class inherits from `ParArray<T, rs, rm>` and overrides `operator[]` to return a **typed row view** instead of a raw pointer.

Type	`operator[](i)` returns	Use
`ArrayAdjacency<rs, rm>`	`AdjacencyRow` — lightweight span	mesh topology (`cell2node` , …)
`ArrayEigenVector<N>`	`Eigen::Map<Vector<real, N>>`	node coordinates (`coords`)
`ArrayEigenMatrix<M, N>`	`Eigen::Map<Matrix<real, M, N>>`	per-cell Jacobians, gradients
`ArrayEigenUniMatrixBatch<M, N>`	`j` -th matrix of a per-row batch	quadrature-point data

`ArrayPair<TArray>` — the convenience bundle

```
template <class TArray = ParArray<real, 1>>
struct ArrayPair {
    ssp<TArray>   father;
    ssp<TArray>   son;
    TTrans        trans;
    auto operator[](index i);       // → father or son by range
};
```

Common type aliases

Alias	Purpose
`ArrayAdjacencyPair<rs, rm>`	mesh connectivity
`ArrayEigenVectorPair<N>`	coords
`ArrayEigenMatrixPair<M, N>`	per-entity matrices
`ArrayEigenUniMatrixBatchPair<M,N>`	quadrature data

`ArrayDof` — the solver's vector space

```
template <int n_m, int n_n>
class ArrayDof : public ArrayEigenMatrixPair<n_m, n_n>;
```

Wraps `father + son + transformer` and adds **MPI-collective vector-space ops** — directly consumable by the Krylov solvers in `src/Solver`.

Operations (CPU + CUDA specializations)

```
void setConstant(real R);
void setConstant(const Eigen::Ref<...> &M);

void operator+=(const ArrayDof &R);
void operator-=(const ArrayDof &R);
void operator*=(real R);
void operator*=(const ArrayDof &R);    // Hadamard
void operator/=(const ArrayDof &R);

void addTo(const ArrayDof &R, real r); // AXPY

// MPI-collective reductions
real norm2();                  real norm2(const ArrayDof &R);
real dot(const ArrayDof &R);
real min();                    real max();    real sum();
```

CFV aliases

```
// src/CFV/VRDefines.hpp
template <int N>  using tUDof    = ArrayDof<N, 1>;
template <int N>  using tURec    = ArrayDof<DynamicSize, N>;
template <int N,
          int d>  using tUGrad   = ArrayDof<d, N>;
```

- `tUDof<N>` — cell-mean conservative variables (ρ, ρu, ρv, ρw, ρE).
- `tURec<N>` — reconstruction coefficients (nDOF chosen at runtime per order).
- `tUGrad<N, d>` — dim × N gradient matrix per cell.

Explicit instantiation covers `(n_m ∈ {1..8, Dynamic, NonUniform}, n_n ∈ {1..5})`.

Host / CUDA dispatch for DOF ops

```
template <DeviceBackend B, int n_m, int n_n>
class ArrayDofOp;

template <int n_m, int n_n>
class ArrayDofOp<DeviceBackend::Host, n_m, n_n> {  /* OpenMP-parallel impl  */ };

#ifdef DNDS_USE_CUDA
template <int n_m, int n_n>
class ArrayDofOp<DeviceBackend::CUDA, n_m, n_n> { /* thrust / raw kernels */ };
#endif
```

Runtime dispatch:

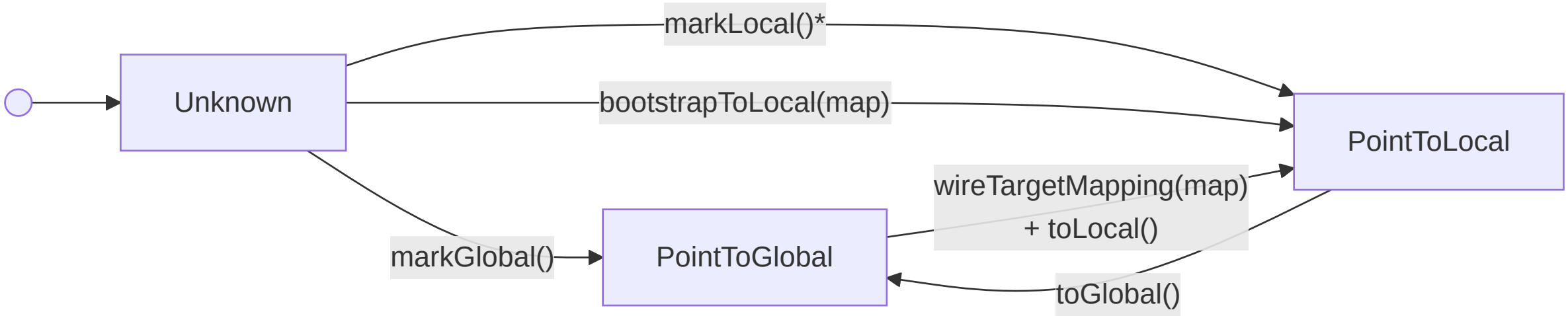
```
#define DNDS_ARRAY_OP_SWITCHER(backend, expr)  \
    switch (backend) {                          \
        case DeviceBackend::Host: { using Op = ArrayDofOp<DeviceBackend::Host, n_m, n_n>; expr; break; } \
        case DeviceBackend::CUDA: { using Op = ArrayDofOp<DeviceBackend::CUDA, n_m, n_n>; expr; break; } \
        default: DNDS_assert_info(false, "Unknown device");                \
    }
```

> **Consequence.** The solver's `norm2()` / `dot()` / `addTo()` calls are the same in C++ regardless of where the data lives — the host code just checks `father->device()` and routes. No `#ifdef CUDA` in Euler or Solver.

State-tracked mesh adjacency (1 / 2)

12+ adjacency arrays (`cell2node` , `face2cell` , `cell2cell` , `node2bnd` , …) must each be **globally or locally indexed** at any given moment — a classic bug surface.

```
enum MeshAdjState {
    Adj_Unknown      = 0,
    Adj_PointToLocal,
    Adj_PointToGlobal,
};
```



* `markLocal()` requires the target mapping to already be wired.
`markGlobal()` is an idempotent no-op when already in `PointToGlobal` .

State-tracked mesh adjacency (2 / 2)

`AdjIndexInfo` — private state + target map

```
struct AdjIndexInfo {
private:
    MeshAdjState     _state{Adj_Unknown};
    t_pLGhostMapping _targetMapping;     // map of the TARGET kind
public:
    // queries
    MeshAdjState state() const;
    bool isLocal(), isGlobal(), isBuilt(), isWired();
    // transitions
    void markGlobal();                   // Unknown|Global → Global
    void markLocal();                    // Unknown → Local (wired only)
    void wireTargetMapping(map);         // not when Local
    // conversions
    void toLocal (adj, nRows);           // & toLocalOMP
    void toGlobal(adj, nRows);           // & toGlobalOMP
    // bootstrap (one-shot)
    void bootstrapToLocal(map, adj, nRows);
};
```

Not-found entries after `toLocal` are encoded as `(-1 - globalIdx)` so they survive round-trips and remain distinguishable from valid local indices.

`AdjPairTracked<TPair>`

```
template <class TPair>
struct AdjPairTracked : public TPair {
    AdjIndexInfo idx;

    void toLocal();  void toGlobal();
    void toLocalOMP(); void toGlobalOMP();
    void bootstrapToLocal(map);
    MeshAdjState state() const;
    bool isLocal(), isGlobal(), isWired();

    template <DeviceBackend B>
    auto deviceView();
};
```

Three-layer DSL

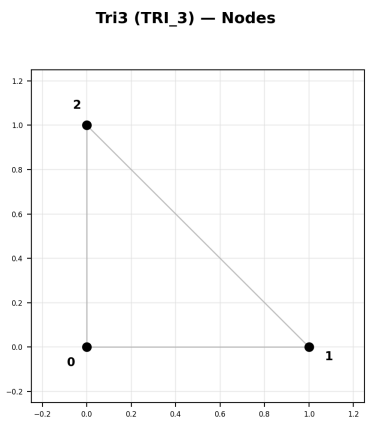
Layer	File	State-aware?
DSL	`MeshConnectivity.hpp`	❌
Checked wrappers	`MeshConnectivity_StateChecked.hpp`	✅ asserts `idx.state()`
`UnstructuredMesh`	`Mesh.cpp`	✅ owns `AdjPairTracked` members

CHAPTER 3

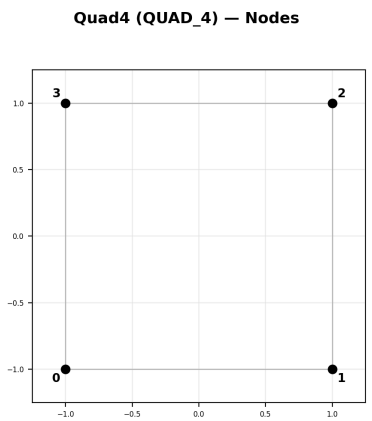
Geometry pipeline

Elements · mesh build · ghosts · DSL

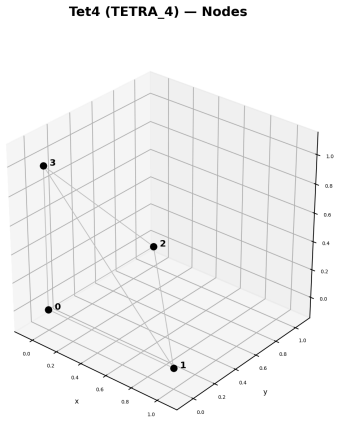
Supported elements — O1 / O2 pairs



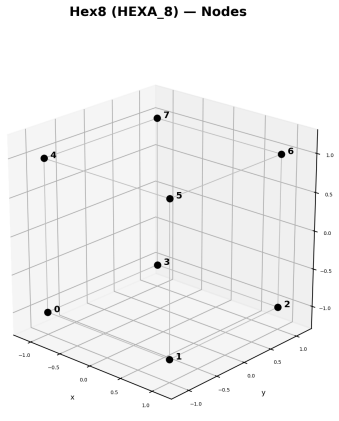
Tri3



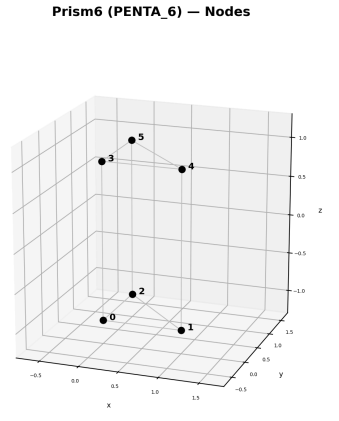
Quad4



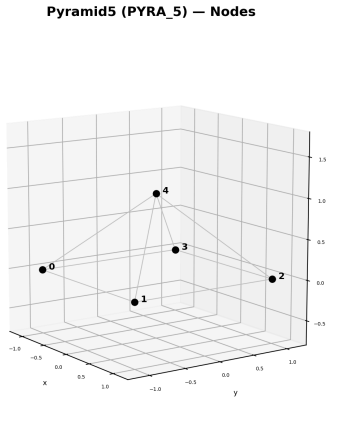
Tet4



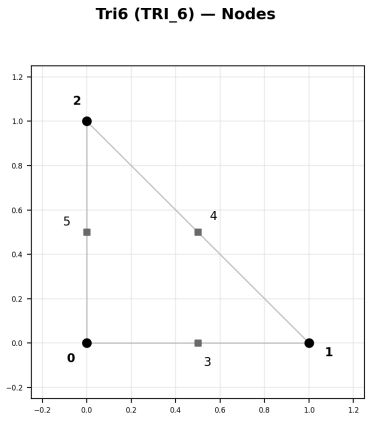
Hex8



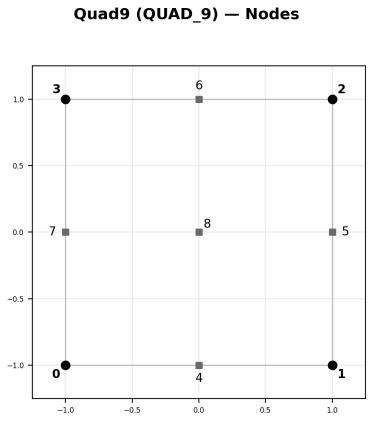
Prism6



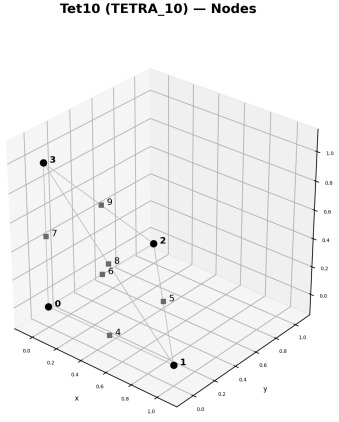
Pyramid5



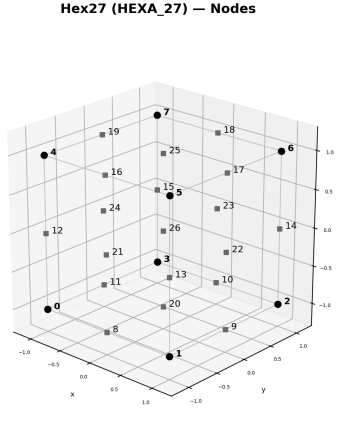
Tri6



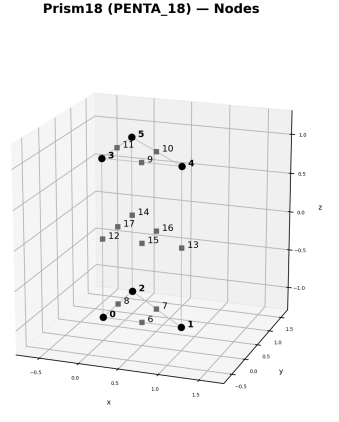
Quad9



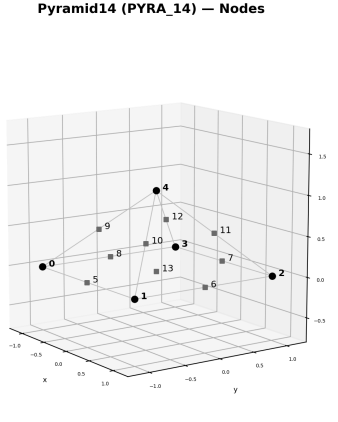
Tet10



Hex27



Prism18



Pyramid14

- **2D cells:** Tri3, Tri6, Quad4, Quad9.
- **3D cells:** Tet4, Tet10, Hex8, Hex27, Prism6, Prism18, Pyramid5, Pyramid14.
- **1D (BC / boundary meshes):** Line2, Line3.

- **Order elevation:** `BuildO2FromO1Elevation()` — Tri3 → Tri6, Quad4 → Quad9, Tet4 → Tet10, Hex8 → Hex27, Prism6 → Prism18, Pyramid5 → Pyramid14.
- **h-refinement:** `BuildBisectO1FormO2()` — one-step

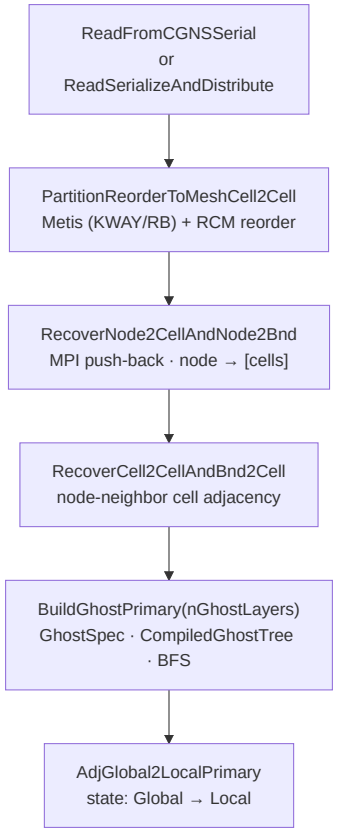
`UnstructuredMesh` — what it owns

```
class UnstructuredMesh : public DeviceTransferable<UnstructuredMesh> {
    // === State flags (five groups) ============================================
    MeshAdjState adjPrimaryState   {Adj_Unknown};  // cell2node, cell2cell, bnd2node, bnd2cell
    MeshAdjState adjFacialState    {Adj_Unknown};  // face2cell, face2node, face2bnd
    MeshAdjState adjC2FState       {Adj_Unknown};  // cell2face, bnd2face
    MeshAdjState adjN2CBState      {Adj_Unknown};  // node2cell, node2bnd
    MeshAdjState adjC2CFaceState   {Adj_Unknown};  // cell2cellFace

    // === Source-of-truth arrays (read / written to HDF5) ====================
    tCoordPair                coords;                    // node positions
    AdjPairTracked<tAdjPair>  cell2node, bnd2node;       // topology
    tElemInfoArrayPair        cellElemInfo, bndElemInfo;  // element type + zone
    tPbiPair                  cell2nodePbi, bnd2nodePbi;  // periodic bits

    // === Derived arrays (rebuilt each load) ==================================
    AdjPairTracked<tAdjPair>  cell2cell, node2cell, node2bnd;
    AdjPairTracked<tAdj2Pair> bnd2cell, face2cell;
    AdjPairTracked<tAdjPair>  cell2face, face2node;
    AdjPairTracked<tAdj1Pair> bnd2face, face2bnd;
    AdjPairTracked<tAdjPair>  cell2cellFace;
    tElemInfoArrayPair        faceElemInfo;

    // === Reorder tracking (restart lineage) ==================================
    tAdj1Pair cell2cellOrig, node2nodeOrig, bnd2bndOrig;
};
```

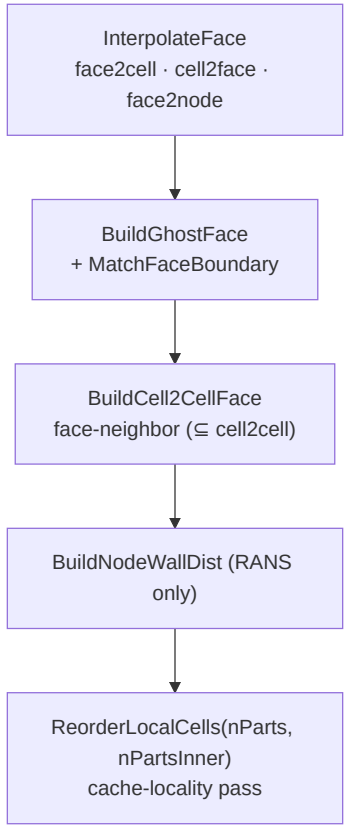
> Every `AdjPairTracked` member carries its own `AdjIndexInfo idx` — so each adjacency knows whether it is global or local, **independently of the 5 group flags**. Group flags remain as a coarse assertion tool.

Mesh build pipeline — end-to-end

Setup & adjacency



Faces & finalization



> ⚠️ **`cell2cell` audit** — every runtime call in CFV / Euler / EulerP was surveyed; `cell2cell` is queried **zero times** in hot loops. It exists exclusively to determine the ghost set, after which face-based traversal takes over. This motivates the DMPlex-style evolution on the roadmap.

Partitioning — `PartitionOptions`

```
struct PartitionOptions {
    std::string metisType        = "KWAY";  // or "RB" (recursive-bisection)
    int         metisUfactor     = 20;      // load imbalance factor
    int         metisSeed        = 0;       // 42 in tests → deterministic
    int         edgeWeightMethod = 0;       // 0: none, 1: face size
    int         metisNcuts       = 3;       // multiple cuts, keep best
};
```

Two partitioners

- **Metis (serial)** — initial cell partition after a serial CGNS read. `MeshPartitionCell2Cell(options)` drives it, then `PartitionReorderToMeshCell2Cell` reorders cells using the partition.
- **ParMetis (distributed)** — used inside `ReadSerializeAndDistribute` to **refine** an even-split load after an H5 restart or cross-np read.

Determinism

Tests fix `metisSeed = 42`. Combined with the `Jacobi` iteration in VR (instead of SOR) and the deterministic LU-SGS replacement, this yields **byte-stable golden values** across re-runs at any `np`.

Ordering

`ReorderLocalCells(nParts, nPartsInner)` runs a two-level cache-locality pass (inner + outer); `ObtainLocalFactFillOrdering` runs AMD / MMD for ILU.

The ghost specification DSL — types

```
enum class EntityKind : int8_t {
    Cell = 0, Face = 1, Edge = 2, Node = 3, Bnd = 4, NUM_KINDS = 5,
};
```

```
struct AdjKind {
    EntityKind from, to;
    EntityKind via;                 // for intra-level (from == to)
    constexpr AdjKind(EntityKind from, EntityKind to);               // direct cone/support
    constexpr AdjKind(EntityKind from, EntityKind to, EntityKind via); // intra-level
};

namespace Adj {
    // Direct cones (downward)
    constexpr AdjKind Cell2Node, Cell2Face, Cell2Edge, Face2Node, Face2Edge, Edge2Node, Bnd2Node;
    // Direct supports (upward)
    constexpr AdjKind Node2Cell, Node2Face, Node2Edge, Node2Bnd,
                      Face2Cell, Edge2Face, Edge2Cell,
                      Bnd2Cell,  Bnd2Face,  Face2Bnd;
    // Intra-level (via Node)
    constexpr AdjKind Cell2Cell, Bnd2Bnd, Face2Face;
    // Intra-level (via Face)
    constexpr AdjKind Cell2CellFace;
}

struct GhostChain  { EntityKind anchor; std::vector<AdjKind> hops; EntityKind target; };
struct GhostSpec   { std::vector<GhostChain> chains;
                     static GhostSpec defaultPrimary(int nLayers = 1); };
```

The ghost DSL — compile & evaluate

```
GhostSpec spec = GhostSpec::defaultPrimary(nLayers);
// Or customize:
spec.chains = {
    { EntityKind::Cell, {Adj::Cell2Cell, Adj::Cell2Cell},              EntityKind::Cell }, // 2 layers
    { EntityKind::Cell, {Adj::Cell2Cell, Adj::Cell2Cell, Adj::Cell2Node}, EntityKind::Node },
    { EntityKind::Bnd,  {Adj::Bnd2Node,  Adj::Node2Bnd},               EntityKind::Bnd  },
};

CompiledGhostTree tree   = CompiledGhostTree::compile(spec);   // merges prefixes → trie
GhostResult       result = dag.evaluateGhostTree(tree, mpi);
```

Evaluator pseudocode (BFS per level)

```
for level in 0..tree.maxLevel:
    for entry in tree.levels[level]:
        collect owned-side non-owned indices
    Allreduce: did any adjacency set grow?
    if yes → scratch-pull that adjacency
    traverse hop, populate next level
```

`GhostResult`

```
struct GhostResult {
    std::unordered_map<EntityKind,
        std::vector<index>> ghostIndices;  // sorted, deduped, global
    std::unordered_set<EntityKind>
        activeKinds;                       // collective (Allreduce)
    bool  hasGhosts(EntityKind) const;
    index totalGhosts() const;
};
```

DSL primitives on `MeshConnectivity`

Beyond the ghost evaluator, `MeshConnectivity` is a reusable DSL for distributed adjacency operations — used in many mesh-build steps.

Primitive	Signature	What it does
`Inverse<cone_rs>`	`(cone, nToLocal, mpi, fromL2G, toL2G, toGlobalMap) → tAdjPair`	A→B cone to B→A support, MPI push-back
`Compose<rs_AB, rs_BC, out_rs>`	`(AB, BC, ...) → tAdjPair`	A→B ∘ B→C → A→C
`ComposeFiltered`	`... pred, matchExtra=nullptr`	Compose with `SharedCountPredicate` filter
`Interpolate<p2n_rs>`	`(parent2node, SubEntityQuery, nParent, nNode, mpi)`	Local-only sub-entity extraction
`InterpolateGlobal<p2n_rs, e2p_rs>`		N-parent distributed interpolation with pbi-aware dedup
`evaluateGhostTree`	`(tree, mpi) → GhostResult`	BFS ghost evaluation

`SharedCountPredicate`

```
struct SharedCountPredicate {
    int  minShared  = 1;
    bool removeSelf = false;
};
```

Used to implement Jacobian-like stencils, e.g. "cells sharing ≥ 2 nodes" to filter face-neighbors from node-neighbors.

Adjacency registry

```
void registerAdj(AdjKind, ssp<AdjVariant>);
void registerGlobalMapping(EntityKind, ssp<GlobalOffsetsMapping>);

ssp<AdjVariant> resolveAdj(AdjKind) const;
bool            hasAdj(AdjKind) const;
```

Mesh build methods populate this registry so `evaluateGhostTree` can look up each hop's adjacency by kind at runtime.

Order elevation & bisection

O1 → O2 elevation

```
void BuildO2FromO1Elevation(UnstructuredMesh &meshO1);
void ElevatedNodesGetBoundarySmooth();
void ElevatedNodesSolveInternalSmooth();
void ElevatedNodesSolveInternalSmoothV1();
void ElevatedNodesSolveInternalSmoothV1Old();
void ElevatedNodesSolveInternalSmoothV2();
```

- **Boundary smooth** — RBF-based placement of added nodes on curved surfaces.
- **Internal smooth** — V1/V1Old/V2 variants of a Laplace-like solve to interpolate interior added-node positions.

O2 → O1 bisection

```
void BuildBisectO1FormO2(UnstructuredMesh &meshO2);
bool IsO1() const;
bool IsO2() const;
```

Per element type (see `docs/elements/*_nodes.png`):

- Tri3 → (elevate) → Tri6 → (bisect) → 4× Tri3.
- Quad4 → (elevate) → Quad9 → (bisect) → 4× Quad4.
- Hex8 → (elevate) → Hex27 → (bisect) → 8× Hex8.
- Prism6 / Pyramid5 — elevated + bisected analogously.

Used in practice for:

- **p-adaptivity studies** via order elevation, and
- **h-refinement benchmarks** via bisection, while keeping the same topology file.

Wall-distance computation

`BuildNodeWallDist(fBndIsWall, WallDistOptions = {})` (`Mesh.hpp:1011`). Used by SA / k-ω / DDES / IDDES models.

Options

```
struct WallDistOptions {
    int  subdivide_quad    = 1;   // refine quads for brute-force
    int  method            = 0;   // 0 = brute, 1 = tree (CGAL AABB)
    int  wallDistExecution = 0;   // 0 = all parallel,
                                  // 1 = serial rank 0,
                                  // N = N-batch ranks
    real minWallDist       = 1e-10;
    int  verbose           = 0;
};
```

Strategies

- **Brute force** — O(N · M) pair loop; trivially vectorizable; used for small cases.
- **CGAL AABB tree** — per-rank tree over wall faces; O(log M) per query.
- **Batched** — mitigate single-rank-memory ceiling by doing the tree build on subsets of ranks (`wallDistExecution > 1`).
- **Poisson** — `GetWallDist_Poisson()` in EulerEvaluator; p-Poisson solve on the mesh, gradient inverted → distance.

Distance is also computed *per face* for use in the SST blending functions.

Cross-`np` restart

Offset sentinels

```
static const index Offset_Parts     = -1;
static const index Offset_One       = -2;
static const index Offset_EvenSplit = -3;
static const index Offset_Unknown   = UnInitIndex;
```

Mode	Meaning
`Unknown`	Auto-detect from `rank_offsets`
`Parts`	`MPI_Scan` over local sizes
`One`	Rank 0 owns the whole dataset
`EvenSplit`	Read-time split into `~N/np`
(explicit)	`isDist()` → `true`; `{localSize, globalStart}`

`ReadSerializeRedistributed` — three cases

1. **No `origIndex`, same `np`** → falls back to `ReadSerialize`.
2. **`origIndex` present, same `np`** → normal read + local remap.
3. **`origIndex` present, different `np`** → `EvenSplit` read, then **3-round `MPI_Alltoallv` rendezvous** to build the directory `origIdx → globalReadIdx`, followed by one `ArrayTransformer` pull.

```
SerializerBase            (abstract)
├── SerializerH5          (collective HDF5)
└── SerializerJSON        (per-rank)
Array → ParArray → ArrayPair → ArrayRedistributor
```

> Write from 4 ranks, restart on 8 — `EulerSolver::ReadRestart` handles all three cases transparently.

CHAPTER 4

Numerics

CFV · VR · flux · limiters · ODE · Krylov

Compact Finite Volume — the reconstruction

Reconstruct a piecewise polynomial from cell means with a **zero-mean basis** per cell:

$$u_i(\mathbf{x}) = \overline{u}_i + \sum_{l=1}^{N_{\text{base}}} u_i^l \, \varphi_i^l(\mathbf{x})$$

- $\overline{u}_i$ — cell-mean, lives in `tUDof<N> = ArrayDof<N,1>`.
- u_i^l — reconstruction coefficients, in `tURec<N> = ArrayDof<Dyn,N>`.
- Basis φ_i^l is orthogonalized locally, normalized by cell scale; degree chosen at runtime per cell.

Supported polynomial orders: 1 – 3 (linear, quadratic, cubic).

```
// Static capacities (src/CFV/VariationalReconstruction.hpp:1051-1054)
maxRecDOFBatch = (dim == 2) ?  4 : 10;
maxRecDOF      = (dim == 2) ?  9 : 19;
maxNDiff       = (dim == 2) ? 10 : 20;
maxNeighbour   = 7;
```

The stencil is **one ring of node-neighbors** — which is exactly what `BuildGhostPrimary(1)` provides by default. Wider stencils (`nGhostLayers ≥ 2`) are available for higher-order variants.

Variational Reconstruction — the functional

Minimize the jumps of **all derivatives up to order k** across each face:

$$I_f = w_g(f) \int_f \sum_{p=0}^{k} w_d(p)^2 \left\| \mathcal{D}_p u_L - \mathcal{D}_p u_R \right\|^2_{\langle\,,\,\rangle_{f,p}} d\Gamma$$

Weights

- $w_g(f)$ — geometric weight; default $w_g = S_f^{-1}$ (area-inverse).
- $w_d(p)$ — dimensionless derivative weight; selects how aggressively each derivative order contributes.
- $\mathcal{D}_p u$ — the p-th derivative tensor (covariant only under linear coordinate changes).

Local system

$$A^i_{mn} u^n_i = \sum_{j \in S_i} \left(B^{i \leftarrow j}_{mn} u^n_j + b^{i \leftarrow j}_m (\overline{u}_j - \overline{u}_i) \right)$$

Solved iteratively — options below.

Three inner-product choices

- **Wang (normal):** $\langle \mathcal{D}_3 u, \mathcal{D}_3 v \rangle = d_f^6 \, \partial_{nnn} u \, \partial_{nnn} v$
- **Pan (X-Y aligned):** $\sum (\Delta_x^a \Delta_y^b \partial^{\cdot}_{xy} u) \, (\Delta_x^a \Delta_y^b \partial^{\cdot}_{xy} v)$
- **Huang (pre-isotropic):** d_f^{2p} weighting, directionally isotropic.

Reconstruction iteration schemes

(`VariationalReconstruction.hpp:938-1031`)

- `DoReconstructionIter` — Jacobi / SOR sweep (tests use Jacobi).
- `DoReconstructionIterDiff` — Jacobian-vector product (GMRES inner).
- `DoReconstructionIterSOR` — SOR with optional reverse pass.
- Fallbacks: `DoReconstruction2nd` , `DoReconstruction2ndGrad` .

VR setup — the three `Construct*` calls

```
template <int dim = 2>
class VariationalReconstruction : public FiniteVolume {
public:
    void ConstructMetrics();                                                    // via FiniteVolume
    void ConstructBaseAndWeight(tFGetBoundaryWeight id2faceDircWeight = …);     // basis + cached diff values
    void ConstructRecCoeff();                                                   // A, B, A^-1 B, secondary
    // …
};
```

What `ConstructMetrics` builds

- Cell volumes, face areas, unit normals, quadrature Jacobians.
- Inertia tensors, major-axis frames, bounding-box scales.
- Physical coords of every quadrature point.
- Smoothness scales for each cell.

What `ConstructBaseAndWeight` builds

- `cellBaseMoment` — basis moments per cell.
- `faceAlignedScales`, `faceMajorCoordScale`.
- `cellDiffBaseCache`, `faceDiffBaseCache` — cached derivative values at all quadrature points, for every neighbour in the stencil.
- `bndVRCaches` — boundary-face caches for BC-weighted VR.

What `ConstructRecCoeff` builds

- `matrixAB`, `vectorB` — per-neighbor RHS blocks.
- `matrixAAInvB`, `vectorAInvB` — precomputed $A^{-1}B$ to accelerate Jacobi / SOR iterations.
- `matrixSecondary`, `matrixAHalf_GG` — auxiliary reconstruction systems.
- `matrixA`, `matrixACholeskyL`, `volIntCholeskyL` — full system + Cholesky factor for dense local solves.

All arrays are `ArrayEigenMatrix*` or `ArrayEigenUniMatrixBatch*` — i.e., Eigen maps over an MPI-aware distributed memory block.

`FiniteVolume` — the metric cache

```
class FiniteVolume : public DeviceTransferable<FiniteVolume> {
    real sumVolume, minVolume{veryLargeReal}, maxVolume, volGlobal;

    tScalarPair  volumeLocal;         // per-cell volume
    tScalarPair  faceArea;            // per-face area
    tRecAtrPair  cellAtr,  faceAtr;   // (NDOF, NDIFF, Order, intOrder)
    tCoeffPair   cellIntJacobiDet, faceIntJacobiDet;
    t3VecsPair   faceUnitNorm;        // normal at each face quadrature pt
    t3VecPair    faceMeanNorm;
    t3VecPair    cellBary,  faceCent,  cellCent;
    t3VecsPair   cellIntPPhysics, faceIntPPhysics;
    t3VecPair    cellAlignedHBox, cellMajorHBox;
    t3MatPair    cellMajorCoord, cellInertia;
    tScalarPair  cellSmoothScale;

    int axisSymmetric = 0;            // wedge axisymmetry
    std::set<index> axisFaces;

    // CRTP: to_device(), to_host(), device(), deviceView<B>()
};
```

CUDA-transferable. `FiniteVolume` (and therefore `VariationalReconstruction`) inherits from `DeviceTransferable<FiniteVolume>`. One call to `fv.to_device()` migrates the entire metric cache to the GPU as a device-side view.

13 Riemann solvers

```
enum RiemannSolverType {
    UnknownRS = 0,
    Roe       = 1, HLLC     = 2, HLLEP    = 3, HLLEP_V1 = 21,
    Roe_M1    = 11, Roe_M2  = 12, Roe_M3  = 13, Roe_M4  = 14, Roe_M5 = 15,
    Roe_M6    = 16, Roe_M7  = 17, Roe_M8  = 18, Roe_M9  = 19,
};
```

Variant	Entropy-fix / eigenvalue scheme
`Roe`	standard Roe + Harten–Yee
`Roe_M1`	cLLF (central + Local Lax–Friedrichs)
`Roe_M2`	Lax–Friedrichs
`Roe_M3`	LD Roe (low-dissipation)
`Roe_M4`	ID Roe (intermediate dissipation)
`Roe_M5`	LD cLLF
`Roe_M6`	H-correction only
`Roe_M7`	Harten–Yee only, no H-correction
`Roe_M8`	H-correction + Harten–Yee
`Roe_M9`	Reserved (eigScheme 9, currently asserts false)
`HLLC`	Harten–Lax–van Leer–Contact
`HLLEP`	HLLE with pressure fix
`HLLEP_V1`	HLLEP variant 1

```
// Shared helper
template <int dim>
RoePreamble<dim> ComputeRoePreamble(ULm, URm, gamma, dumpInfo);
```

`RoePreamble` — the shared middle

```
template <int dim>
struct RoePreamble {
    TVec veloLm, veloRm;                    // primitive velocities
    real rhoLm, rhoRm, pLm, pRm, HLm, HRm;  // primitive state
    real veloLm0, veloRm0;                  // normal velocity components

    TVec veloRoe;                           // Roe-averaged velocity
    real sqrtRhoLm, sqrtRhoRm;
    real vsqrRoe, HRoe, asqrRoe, rhoRoe, aRoe;
};
```

Flux signature

```
template <int dim, int eigScheme>
void RoeFlux(UL, UR, ULm, URm, n, vgN,
             /*out*/ flux,
             /*out*/ dLambda,
             fixScale, gamma, dumpInfo);

template <int dim, int type>
void HLLEPFlux_IdealGas(UL, UR, ULm, URm, n, vgN,
                        flux, …, gamma, dumpInfo);

template <int dim>
void HLLCFlux(UL, UR, ULm, URm, n, vgN, …);
```

Why this factoring

All 13 variants share `ComputeRoePreamble` — the Roe average, H_{Roe}, a_{Roe}, etc. The `eigScheme` template parameter then selects the dissipation / entropy-fix strategy.

- **One template instantiation per (`dim`, `eigScheme`)** keeps code size bounded.
- **Compile-time dispatch** — no indirect calls in the flux kernel.
- **Same interface** for inviscid and full Navier-Stokes flux: `NSFluxInvis<dim>`, `NSFluxVis<dim>(U, gradU, T, mu, n, flux, adiabaticWall, useQCR)`.

Limiters — the FWBAP L2 family

Multi-way (≥ 2 directions)

- `FWBAP_L2_Multiway` — generic Eigen arrays.
- `FWBAP_L2_Multiway_Polynomial2D` — 2D polynomial-weighted norm.
- `FWBAP_L2_Multiway_PolynomialOrth` — orthogonal variant.
- `FMEMM_Multiway_Polynomial2D` — Modified Extremum-Monotone Mixer.

Power parameter: `p = 4` ; `verySmallReal_pDiP = std::pow(verySmallReal, 1.0/p)` stabilises near zero.

Biway (pair)

- `FWBAP_L2_Biway`
- `FWBAP_L2_Cut_Biway` — sign-cutoff
- `FMINMOD_Biway`
- `FVanLeer_Biway`
- `FWBAP_L2_Biway_PolynomialNorm<dim, nVarsFixed>`
- `FMEMM_Biway_PolynomialNorm<dim, nVarsFixed>`
- `FWBAP_L2_Biway_PolynomialOrth`

Configuration

```
"limiterProcedure":  0   // WBAP (V2)
"limiterProcedure":  1   // CWBAP (V3)  ← recommended
"usePPRecLimiter":   true
```

> **Positivity preservation** — `LimiterUGrad` (Euler side) clamps gradients; `EvaluateURecBeta` enforces cell-mean positivity on reconstructed values; `EvaluateCellRHSAlpha` enforces CFL-consistent per-cell RHS scaling.

VR's own limiter — WBAP with characteristic transform

```
template <int nVarsFixed>
void DoLimiterWBAP_C(tUDof<nVarsFixed>  &u,
                     tURec<nVarsFixed>  &uRec,
                     tURec<nVarsFixed>  &uRecNew,
                     tURec<nVarsFixed>  &uRecBuf,
                     tSmoothIndicator   &si,
                     bool                ifAll,
                     tFM   FM,                  // cons → char transform
                     tFMI  FMI,                 // char → cons transform
                     bool  putIntoNew = false);

template <int nVarsFixed>
void DoLimiterWBAP_3(...);                      // 3-mode variant
```

Flow

1. Compute per-face smoothness indicator `si`.
2. Transform reconstruction coefficients to characteristic variables (`FM`).
3. Apply WBAP limiter per characteristic, across the multi-way neighborhood.
4. Transform back (`FMI`).
5. Optionally write into `uRecNew` (double-buffer for iterative schemes).

Smoothness indicators

- `DoCalculateSmoothIndicator<nVarsFixed, nVarsSee=2>(si, uRec, u, varsSee)` — classical indicator over a subset of variables.
- `DoCalculateSmoothIndicatorV1<nVarsFixed>(si, uRec, u, varsSee, FPost)` — V1 with user-provided post-processing.

Time integration — the ODE zoo

All integrators descend from:

```
template <class TDATA, class TDTAU>
class ImplicitDualTimeStep {
    using Frhs     = std::function<void(TDATA&, TDATA&, TDTAU&, int, real, int)>;
    using Fdt      = std::function<void(TDATA&, TDTAU&, real, int)>;
    using Fsolve   = std::function<void(TDATA&, TDATA&, TDATA&, TDTAU&, real, real, TDATA&, int, real, int)>;
    using Fstop    = std::function<bool(int, TDATA&, int)>;
    using Fincrement = std::function<void(TDATA&, TDATA&, real, int)>;
    virtual void Step(TDATA &x, TDATA &xinc, const Frhs&, const Fdt&, const Fsolve&,
                      int maxIter, const Fstop&, const Fincrement&, real dt) = 0;
};
```

odeCode	Class	Scheme
`103`	`ImplicitEulerDualTimeStep`	Backward Euler
`0`	`ImplicitBDFDualTimeStep`	BDF2 / BDF-k
—	`ImplicitVBDFDualTimeStep`	Variable-step BDF-k
`1`	`ImplicitSDIRK4DualTimeStep` (`schemeCode` 0…4)	SDIRK-4 · ESDIRK2/3 · Trapezoidal
`101`	(alias for `1`)	(backward-compat `odeCode`)
`401`	`ImplicitHermite3SimpleJacobianDualStep`	**HM3 + p-Multigrid**
`2`	`ExplicitSSPRK3TimeStepAsImplicitDualTimeStep`	SSP-RK3

`SetExtraParams(json)` exposes scheme-specific knobs (e.g. `nMG`, `incFScale`).

HM3 + p-Multigrid

HM3 (Hermite-3) is a 3rd-order A-stable implicit scheme with three modes:

- **U2R2** — 2 solution states + 2 residual states.
- **U2R1** — 2 solution states + 1 residual state.
- **U3R1** — 3 solution states + 1 residual state.

p-MG inside the time step

Inside `ImplicitHermite3SimpleJacobianDualStep::Step()` a **nonzero** **`nMG`** triggers p-multigrid smoothing cycles:

```
// pseudocode inside the inner solve (lines 1250-1251)
fdt (xMG, dTau, 1.0, /*upos=*/2);      // lower-order pseudo-timestep
frhs(rhsbuf[1], xMG, dTau, iter, 1.0, /*upos=*/2);
```

The `upos=2` argument tells the evaluator to evaluate at a **lower polynomial order** (level-transition). VR provides `DownCastURecOrder(curOrder, iCell, uRec, downCastMethod)` to project reconstruction coefficients between orders.

Companions

- **`tpMG`** — toggle for multigrid in the outer dual-time loop.
- **`incFScale`** — incremental flux scaling on lower MG levels; integrated into the entropy fix path (`RELEASE_NOTES.md`).
- **Positivity-preserving limiters in `LimiterUGrad`** — prevent the lower-order coarse-grid correction from producing negative density / pressure.

Other `SDIRK4` codes

- `schemeCode = 0` — Nørsett 3-stage SDIRK-4
- `schemeCode = 1` — 6-stage ARK-family SDIRK
- `schemeCode = 2` — Kennedy–Carpenter ESDIRK3
- `schemeCode = 3` — Trapezoidal
- `schemeCode = 4` — ESDIRK2, `γ = 1 − √2/2`

Linear solvers — Krylov + LU-SGS preconditioner

Krylov methods

```
template <class TDATA>
class GMRES_LeftPreconditioned {
public:
    GMRES_LeftPreconditioned(index dofSize);
    void setSpace(int kSpace);
    bool solve(const TDATA &rhs, TDATA &x,
               FMatVec Ax, FPCApply PC,
               int maxIter, real tol);
};

template <class TDATA, class TScalar>
class PCG_PreconditionedRes { … };
```

Matrix-free: the caller supplies `Ax` and `PC` functors.

Matrix-free LU-SGS preconditioner

Provided by `EulerEvaluator`:

```
void LUSGSMatrixInit(JDiag, JSource, dTau, dt, alphaDiag, u, uRec, jacCode, t);
void LUSGSMatrixVec(alphaDiag, t, u, uInc, JDiag, AuInc);
void LUSGSMatrixToJacobianLU(alphaDiag, t, u, JDiag, jacLU);
void UpdateSGS(alphaDiag, t, rhs, u, uInc, uIncNew, JDiag,
               forward, gsUpdate, sumInc, uIncIsZero = false);
void LUSGSMatrixSolveJacobianLU(alphaDiag, t, rhs, u, uInc, uIncNew,
                                bBuf, JDiag, jacLU,
                                uIncIsZero, sumInc);
void UpdateSGSWithRec(alphaDiag, t, rhs, u, uRec, uInc, uRecInc,
                      JDiag, forward, sumInc);
```

Selector

```
"gmresCode": 0  // LUSGS only      (cheap, robust)
"gmresCode": 1  // GMRES            (matrix-free Krylov)
"gmresCode": 2  // LUSGS + GMRES    (LUSGS as PC for GMRES)
```

Direct path for small blocks: `src/Solver/Direct.hpp` (LU / LDLT). Optional **SuperLU_dist** via the `cfd_externals` submodule.

CHAPTER 5

Parallelism

MPI · OpenMP · CUDA

MPI — the "set up once" discipline

> *Setup is collective and expensive. Communication is local and cheap.*

Build-once phase — collective

```
trans.setFatherSon(father, son);

trans.createFatherGlobalMapping();
  // collective: MPI_Allgather over local sizes

trans.createGhostMapping(pullGlobal);
  // collective: sorts + dedups pullGlobal IN PLACE
  // — saves a copy if you need the original

trans.createMPITypes();
  // local: MPI_Type_create_hindexed describes
  // the scattered rows to send/recv
  // — ALSO resizes the son array to hold them

trans.initPersistentPull();
  // local: MPI_Recv_init + MPI_Send_init
```

The derived MPI datatypes persist with the transformer — teardown costs them nothing until destruction.

Hot-loop phase — local only

```
for (int step = 0; step < N; ++step) {
    trans.startPersistentPull();    // MPI_Startall
    computeFluxes(/* reads ghosts */);
    trans.waitPersistentPull();     // MPI_Waitall
}

trans.clearPersistentPull();
```

🪛 **v0.2.0 bug-fix:** `globalSize()` used to be collective and could deadlock when some ranks took short-cut paths. It's now cached at `createFatherGlobalMapping` time — fully local.

Two communication strategies

`MPI::CommStrategy::Instance().GetArrayStrategy()` selects:

`HIndexed` — default

```
MPI_Type_create_hindexed(count, blocklengths, displacements,
                         base_type, &new_type);
```

- Describes **scattered rows** directly in MPI's datatype system.
- MPI library + driver are free to pipeline and vectorize the pack.
- Zero-copy on the application side.
- Best on well-tuned MPI stacks over InfiniBand / Slingshot.

`InSituPack`

```
inSituBuffer[rank].clear();
for (index i : pushingIndexLocal[rank])
    inSituBuffer[rank].append(row(i));
MPI_Isend(inSituBuffer[rank].data(), ...);
```

- Explicit pack into contiguous buffers.
- Beats `HIndexed` on some **older MPI stacks** and on **CUDA-aware MPI** with GPU-Direct where the driver prefers flat buffers.
- One extra memory pass per phase — tradeoff.

> Both strategies live behind the same public API. The choice is a tuning knob — no application-level changes needed.

`BorrowGGIndexing` — avoid collective setup twice

```
// Primary array: does the full collective setup
ArrayTransformer<real, 5> cellUTrans;
cellUTrans.setFatherSon(uFather, uSon);
cellUTrans.createFatherGlobalMapping();
cellUTrans.createGhostMapping(pullGlobal);
cellUTrans.createMPITypes();

// Secondary array: reuses the *global + ghost* mapping.
// Only the MPI datatypes (which depend on the row size) are rebuilt.
ArrayTransformer<real, DynamicSize> recTrans;
recTrans.setFatherSon(uRecFather, uRecSon);
recTrans.BorrowGGIndexing(cellUTrans);   // <-- key line
recTrans.createMPITypes();
recTrans.initPersistentPull();
```

Consequence. In the Euler pipeline every DOF array (`u` , `uPrev` , `uInc` , `uRec` , `uRecInc` , `uRecB` , …) shares a single ghost map established from the `cell2cell` adjacency. Only the MPI datatypes differ, keyed on the row size of each array.

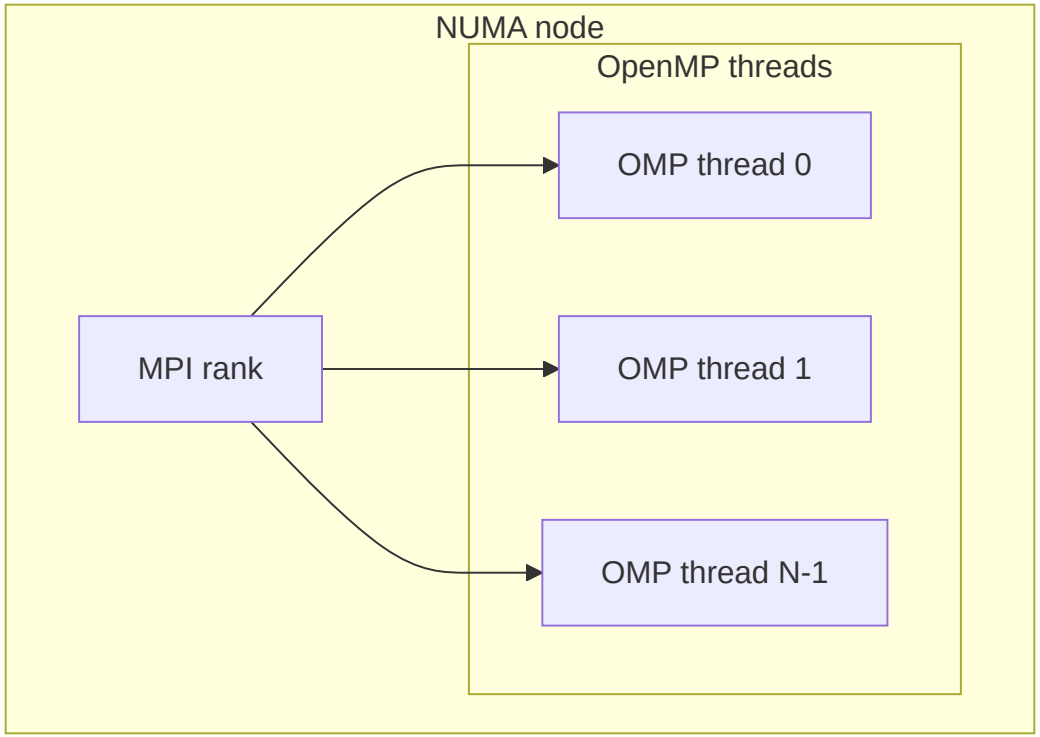
OpenMP in the stack

`-DDNDS_DIST_MT_USE_OMP=ON` activates threaded paths throughout:

Where OMP is already applied

- **ILU-OMP preconditioner** — parallel forward/backward sweeps (new in v0.2.0).
- **Eigen reductions** — `EigenVecMin`, `EigenVecSum` fold per thread, then combine.
- **State transitions** — `toLocalOMP` / `toGlobalOMP` / `bootstrapToLocalOMP` parallelize over the rows of adjacency arrays.
- **FV metric construction** — many `ConstructX()` methods in `FiniteVolume` loop over cells / faces with `#pragma omp parallel for`.
- **VR iteration** — `DoReconstructionIter` has an OMP variant.

Hybrid model



CI default `OMP_NUM_THREADS=2` (override at configure time via `DNDS_TEST_OMP_THREADS`). MPI-rank count per test configurable via `DNDS_TEST_NP_LIST`.

Typical production deployment: **1 MPI rank per NUMA node × OMP threads** within. MPI handles cross-socket / cross-node; OMP handles within.

CUDA path — `DeviceTransferable` CRTP

```
template <class TDerived>
class DeviceTransferable {
public:
    // Derived implements: device_array_list() returning a tuple of host-device arrays
    void to_device(DeviceBackend B = DeviceBackend::CUDA);
    void to_host();
    DeviceBackend device() const;
    template <DeviceBackend B> auto deviceView();
};

// Example user
class FiniteVolume : public DeviceTransferable<FiniteVolume> {
    auto device_array_list() {
        return std::tie(volumeLocal, faceArea, faceUnitNorm, cellBary,
                        cellInertia, cellIntJacobiDet, /* ... */);
    }
};
```

Usage

```
fv.to_device();
auto dv = fv.deviceView<CUDA>();
launchKernel<<<blocks, threads>>>(dv);
fv.to_host();
```

Already transferable

- `UnstructuredMesh` (connectivity)
- `FiniteVolume` (metrics)
- `VariationalReconstruction` (via base)
- `VRDefines` DOF arrays
- Per-element shape function tables

Build: `cmake --preset cuda` → `-DDNDS_USE_CUDA=ON` · Thrust fixes via `CMAKE_CUDA_ARCHITECTURE=native`.

EulerP — the purpose-built GPU evaluator

Problem: the stock `Euler` evaluator uses Eigen with compile-time `nVars`; Eigen matrix ops do not cleanly lower to device-callable scalar loops. CUDA kernel launches over tiny matrices cost more than the math.

Solution: a parallel-track evaluator in `src/EulerP/` that:

1. Drops the Eigen matrix abstraction inside kernels — scalar loops over `nVars`.
2. Splits into `EvaluatorDeviceView<B>` with `B ∈ {Host, CUDA}` — same interface, two implementations compiled in separate translation units (`.cpp` and `.cu`).
3. Bundles per-call arguments into `*_Arg` structs (e.g. `RecGradient_Arg`, `Flux2nd_Arg`) so the launching host code doesn't need to know argument order.

```
template <DeviceBackend B>
struct EvaluatorDeviceView {
    FiniteVolume::t_deviceView<B>   fv;
    BCHandlerDeviceView<B>          bc;
    PhysicsDeviceView<B>            physics;
};
```

Python driver: `python/DNDSR/EulerP/EulerP_Solver.py` orchestrates the full EulerP pipeline from Python with CUDA selected by runtime flag.

EulerP — the kernel pipeline

```
class Evaluator {
    ssp<CFV::FiniteVolume>  fv;
    ssp<BCHandler>           bcHandler;
    ssp<Physics>             physics;
    // face buffers (dense packed from ghost father+son)
    tUFaceBuffer u_face_bufferL, u_face_bufferR;
    tUScalarFaceBuffer uScalar_face_bufferL, uScalar_face_bufferR;

public:
    // Setup
    void BuildFaceBufferDof(TUDof &u);
    void BuildFaceBufferDofScalar(TUScalar &u);
    void PrepareFaceBuffer(int nVarsScalar);

    // Pipeline kernels (each host-or-device via Evaluator_impl<B>)
    void RecGradient   (RecGradient_Arg &arg);    // Green-Gauss + Barth-Jespersen
    void Cons2PrimMu   (Cons2PrimMu_Arg &arg);
    void Cons2Prim     (Cons2Prim_Arg   &arg);
    void RecFace2nd    (RecFace2nd_Arg  &arg);    // 2nd-order face reconstruction
    void Flux2nd       (Flux2nd_Arg     &arg);    // inviscid + viscous face flux
};
```

Why the arg-bundle struct

- All array references in one place → simple to serialize for a device kernel.
- Host/CUDA dispatch happens at a single call site (`Evaluator_impl<B>`).
- The launching host code sees the same identifier `EvaluateRHS` regardless of backend.

GPU engineering notes

Benchmarks already shipped

- **Block-sparse MatVec** — `src/Geom/Mesh/BenchmarkFiniteVolume.cu` exercises the metric arrays on-device with varied block sizes.
- **SoA vs AoS** — multiple layout variants benchmarked for the per-cell DOF blocks.

Memory model

- `host_device_vector<T>` — a vector that can shadow itself on device; used throughout `FiniteVolume` / `UnstructuredMesh`.
- Transfers are explicit (`to_device` / `to_host`) — no hidden synchronization.

Pitfalls avoided

- **Thrust + CMake:** `CMAKE_CUDA_ARCHITECTURE=native` fixes a class of compile errors in Thrust's internal machinery.
- **Accidental `to_device`:** a bug in the face-buffer creation path was copying host buffers to device needlessly; fixed in v0.2.0.
- **`py::classh` holders:** ensure safe Python↔C++ ownership when CUDA pointers survive across Python GC boundaries.

Work in progress

- Extend full `Euler` evaluator to CUDA (not just `EulerP`).
- GPU-aware MPI via `MPI_Type_create_hindexed` over pinned device memory.

CHAPTER 6

I/O & Interoperability

Serializer · JSON config · Python · VTK · CGNS

Serialization layer stack

Layer	Responsibility
`SerializerBase`	Abstract scalar / vector / byte-array interface
`SerializerH5`	MPI-parallel HDF5 (collective I/O)
`SerializerJSON`	Per-rank JSON (`IsPerRank() == true`), no MPI coordination
`Array`	Per-array metadata, structure tags, flat data buffer
`ParArray`	Global offsets, `EvenSplit` , CSR global row-starts
`ArrayPair`	Father-son bundle · `ReadSerializeRedistributed`
`ArrayRedistributor`	Rendezvous redistribution via `ArrayTransformer`

> **Key property.** Every method in `SerializerH5` is **MPI-collective** — every rank must call them in the same order, even when that rank has `size == 0` . Failing to participate causes a hang, not a crash.

`SerializerBase` — the public interface

```
// File lifecycle
virtual void OpenFile(const std::string &fName, bool read) = 0;
virtual void CloseFile() = 0;

// Path navigation (think HDF5 group structure)
virtual void CreatePath(const std::string &p) = 0;
virtual void GoToPath(const std::string &p)  = 0;
virtual std::string              GetCurrentPath() = 0;
virtual std::set<std::string>    ListCurrentPath() = 0;
virtual bool                     IsPerRank() = 0;   // true for JSON
virtual int   GetMPIRank() = 0;   int GetMPISize() = 0;
virtual const MPIInfo &getMPI() = 0;

// Scalars (per-rank)
virtual void WriteInt(const std::string &name, int64_t v) = 0;
virtual void WriteIndex/WriteReal/WriteString(...) = 0;
virtual void ReadInt /ReadIndex / ReadReal / ReadString(...) = 0;

// Vectors (COLLECTIVE under H5)
virtual void WriteIndexVector(const std::string &name, const std::vector<index> &v,
                              ArrayGlobalOffset offset) = 0;
virtual void ReadIndexVector (const std::string &name,       std::vector<index> &v,
                              ArrayGlobalOffset &offset) = 0;   // offset is in/out
// ... Rowsize, Real, SharedIndex, SharedRowsize
virtual void WriteUint8Array(const std::string &name, const uint8_t *data,
                             index size, ArrayGlobalOffset offset) = 0;
virtual void ReadUint8Array (const std::string &name, uint8_t *data,
                             index &size, ArrayGlobalOffset &offset) = 0;
```

`ArrayGlobalOffset` — five offset modes

```
static const index Offset_Parts     = -1;
static const index Offset_One       = -2;
static const index Offset_EvenSplit = -3;
static const index Offset_Unknown   = UnInitIndex;

class ArrayGlobalOffset {
    index _size{0};
    index _offset{0};
public:
    ArrayGlobalOffset(index sz, index ofs);
    index size()   const;
    index offset() const;
    ArrayGlobalOffset operator*(index R) const;   // scales size (and offset if real)
    ArrayGlobalOffset operator/(index R) const;
    void CheckMultipleOf(index R) const;
    bool operator==(const ArrayGlobalOffset &other) const;
    bool isDist() const;                           // _offset >= 0
};

extern ArrayGlobalOffset ArrayGlobalOffset_Unknown, _One, _Parts, _EvenSplit;
```

Sentinel	Meaning
`Unknown`	Auto-detect from companion `rank_offsets` dataset
`Parts`	Compute offset via `MPI_Scan` over local sizes
`One`	Rank 0 writes / reads the whole dataset
`EvenSplit`	Read: each rank gets `~nGlobal / nRanks` rows
`isDist()`	Explicit `{localSize, globalStart}`

Zero-size partition safety

The trap

When `nGlobal < nRanks` (5 entries across 8 ranks), `EvenSplitRange` assigns 0 rows to some ranks. Collective HDF5 calls still demand every rank participates — and `std::vector<>::data()` on an empty vector may return `nullptr`.

```
std::vector<index> v(size);        // size may be 0
ReadDataVector<index>(name, v.data(), ...);  // may pass nullptr → hang
```

Caller-side helpers like `__ReadSerializerData` and `ReadUint8Array` would skip the `H5Dread` when `buf == nullptr`, and the collective hangs.

The fix

Every caller in `SerializerBase.cpp` passes a **stack-allocated dummy pointer** when `size == 0`:

```
index dummy;
ReadDataVector<index>(name,
    size == 0 ? &dummy : v.data(),
    ...);
```

`ReadUint8Array` exposes the two-pass pattern:

1. First call: `data = nullptr`, returns the size.
2. Second call: allocate + call again with a real (or dummy) pointer.

All collectives proceed with 0-count hyperslabs on the empty ranks — no application-level branching.

Write-N-read-M — the rendezvous pattern

Consequence. `EulerSolver::ReadRestart` is a single call. The user writes from 4 ranks on a login node, restarts on 1024 ranks on a compute partition, and the same JSON config runs. Ranks with `localRows == 0` participate in every collective with empty buffers.

Typed JSON configs — `DNDS_DECLARE_CONFIG`

```
struct ImplicitCFLControl {
    real CFL                   = 10.0;
    int  nForceLocalStartStep  = INT_MAX;
    bool useLocalDt            = true;
    real RANSRelax             = 1.0;

    DNDS_DECLARE_CONFIG(ImplicitCFLControl) {
        DNDS_FIELD(CFL,                  "CFL for implicit local dt");
        DNDS_FIELD(nForceLocalStartStep, "Step to force local dt",
                   DNDS::Config::range(0));
        DNDS_FIELD(useLocalDt,           "Use local (vs uniform) dTau");
        DNDS_FIELD(RANSRelax,            "RANS under-relaxation factor",
                   DNDS::Config::range(0.0, 1.0));

        config.check([](const T &s) -> DNDS::CheckResult {
            if (s.RANSRelax <= 0) return {false, "RANSRelax must be positive"};
            return {true, ""};
        });
    }
};
```

What the macro gives you. No base class, no virtual members, no per-instance data — the struct stays a POD safe for CUDA. Underneath, a static `_dnds_do_register()` method is generated that fills a `ConfigRegistry<T>` singleton with `FieldMeta` records.

Configs — field kinds & cross-field checks

```
// Simple scalars & bounded scalars
DNDS_FIELD(CFL,         "CFL number");
DNDS_FIELD(nInternalIt, "Inner iterations",  DNDS::Config::range(0));
DNDS_FIELD(relax,       "Relaxation factor", DNDS::Config::range(0.0, 1.0));

// Enum with value names (appears in schema as enum constraint)
DNDS_FIELD(rsType,      "Riemann solver type",
           DNDS::Config::enum_values({"Roe","HLLC","HLLEP","HLLEP_V1",
                                      "Roe_M1","Roe_M2","Roe_M3","Roe_M4",
                                      "Roe_M5","Roe_M6","Roe_M7","Roe_M8","Roe_M9"}));

// Documentation kwargs — emitted as "x-..." extensions in schema
DNDS_FIELD(CFL, "CFL number", DNDS::Config::info("units", "nondim"),
                              DNDS::Config::info("ref",   "Jameson 1985"));

// Nested sub-section
config.field_section(&T::frameRotation, "frameConstRotation", "Rotating frame");

// Arrays / maps of sub-objects
config.field_array_of<BoxInit>   (&T::boxInits,   "boxInitializers", "Box initializers");
config.field_map_of<CoarseCtrl>  (&T::coarseList, "coarseGridList",  "Per-level controls");

// Opaque JSON (for scheme-specific extras)
config.field_json  (&T::extra,    "odeSettingsExtra", "Opaque ODE scheme settings");

// Renaming / aliases (backward compatibility)
config.field_alias (&T::rsType,   "riemannSolverType", "Riemann solver type");
```

Configs — auto-generated JSON Schema & validation

```
// Emit the schema (run-time or ahead-of-time)
nlohmann::ordered_json schema = ConfigRegistry<EulerConfig>::Instance().emitSchema("Euler solver config");
```

```
{
  "$schema": "http://json-schema.org/draft-07/schema#",
  "type": "object",
  "description": "Euler solver config",
  "properties": {
    "CFL":     { "type": "number", "default": 10.0, "description": "..." },
    "rsType":  { "type": "string", "default": "Roe",
                 "enum": ["Roe","HLLC",...,"Roe_M9"] }
  }
}
```

CLI + tooling

```
./build/app/euler.exe --emit-schema > euler_schema.json
# drops ~107 KB per-solver schema
```

VS Code + any JSON-schema-aware editor give autocompletion and in-line validation. Pre-computed schemas ship in `cases/euler_schema.json`, `eulerSA3D_schema.json`, etc.

Runtime validation

```
auto &reg = ConfigRegistry<EulerConfig>::Instance();
reg.readFromJson(j, cfg);            // deserialize + range checks
reg.validate(cfg);                   // cross-field
reg.validateWithContext(cfg, ctx);   // uses nVars, dim, modelCode
reg.validateKeys(userJson);          // throws on unknown keys
```

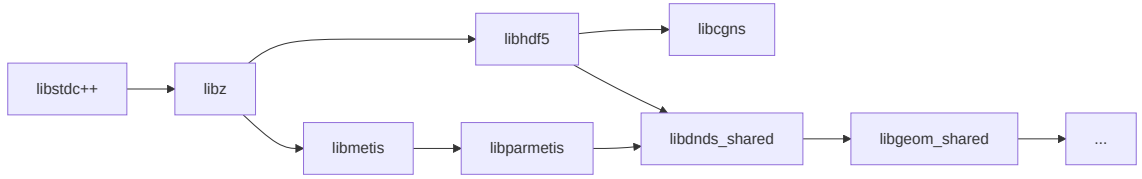
`validateKeys` is automatic — no hand-maintained list of allowed fields.

Python bindings — the import chain

```
from DNDSR import DNDS
  ↓
python/DNDSR/__init__.py
  ↓
python/DNDSR/DNDS/__init__.py
  ├── _loader.preload("dnds")                    # ctypes.CDLL · RTLD_GLOBAL
  ├── from ._ext.dnds_pybind11 import *          # pybind11 extension
  └── _init_mpi()                                # MPI_Init_thread AT IMPORT TIME
```

The preload order matters

`_loader.py` loads external dependencies with `RTLD_GLOBAL` **before** the pybind11 extension opens. If they were loaded later with the default `RTLD_LOCAL`, the extension would not find the symbols it depends on.



Four modules, one package

- `DNDSR.DNDS` — arrays, MPI, serializer
- `DNDSR.Geom` — mesh reading & manipulation
- `DNDSR.CFV` — finite volume / VR / Fourier analysis
- `DNDSR.EulerP` — GPU-friendly Euler evaluator

Top-level `__init__.py` imports all four so a single `from DNDSR import *` works.

Python mesh-read — the demo case

```python
from DNDSR import DNDS
from DNDSR.Geom.utils import read_mesh, prepare_mesh, build_bnd_mesh

# 1. MPI bootstrap (implicit MPI_Init_thread already ran at import)
mpi = DNDS.MPIInfo(); mpi.setWorld()

# 2. Read a CGNS mesh with elevation and bisection
result = read_mesh(
    "data/mesh/UniformSquare_10.cgns",
    mpi       = mpi,
    dim       = 2,
    elevation = "O2",      # Quad4 → Quad9
    bisect    = 1,         # one round of h-refinement
)

# 3. Finish the mesh (build ghosts, interpolate faces, reorder cells)
prepare_mesh(result.mesh, result.reader)

# 4. Extract surface mesh and dump VTK
bnd = build_bnd_mesh(result.mesh)
result.mesh.BuildVTKConnectivity()
```

PEP 561 compliant. A `py.typed` marker ships in the package; `.pyi` stubs are auto-generated by `pybind11-stubgen` during `cmake --install`. Pyright, mypy, and Pylance see full C++ type signatures.

CFV Python — Fourier dissipation-dispersion analysis

```python
from DNDSR.CFV import ModelEvaluator  # pure-Python wrapper over pybind11 class

me = ModelEvaluator(mesh, fv, vr)
me.set_order(3)

# Fourier analysis: plug in a plane wave, read back the complex amplification
kx_range = np.linspace(-pi, pi, 200)
for kx in kx_range:
    lam = me.fourier_amplification_factor(kx)
    print(kx, lam.real, lam.imag)
```

> **Why this matters for a research code.** VR's dispersion/dissipation properties depend on order, limiter, and inner-product choice. Having a Python harness to sweep them over a discrete Fourier spectrum means parameter studies (limiter combinations, inner-product choices, derivative weights) are done in hours, not weeks.

Other Python-exposed bits:

- `ArrayPair`, `ArrayEigenMatrix/Vector/Batch`
- `BuildUDof / BuildURec / BuildUGrad` (typed constructors)
- VTK output, wall-distance, `to_device / to_host`
- The full `MeshAdjState` enum and `AdjPairTracked::idx` queries (query-only, no mutation from Python — intentional)

CHAPTER 7

Solvers & Validation

Euler family · EulerEvaluator · cases

The Euler / N-S family — one binary per model

```
// app/Euler/euler.cpp — the entire file
#include "EulerSolver.hpp"
int main(int argc, char *argv[]) {
    return DNDS::Euler::RunSingleBlockConsoleApp<
                DNDS::Euler::NS>(argc, argv);
}
```

Model enum (`EulerModel`)

```
enum EulerModel {
    NS,         // 2D Navier-Stokes
    NS_3D,
    NS_SA,      // 2D Spalart-Allmaras
    NS_SA_3D,
    NS_2D,      // alias for NS
    NS_2EQ,     // k-omega two-equation
    NS_2EQ_3D,
    NS_EX,      // reactive / multi-species
    NS_EX_3D,
};
```

Template dispatch on `EulerModel` produces one binary per solver — shared source, separated object files.

RANS model enum

```
enum RANSModel {
    RANS_None,
    RANS_SA,        // Spalart-Allmaras (IDDES capable)
    RANS_KOWilcox, // Wilcox k-omega
    RANS_KOSST,     // Menter k-omega SST
    RANS_RKE,       // Realizable k-epsilon
};
```

Each has a `RANSModelTraits<>` specialization with its own wall BC, source terms, and spectral radius.

`EulerSolver` — the top-level conductor

The Euler module extends CFV's generic `tUDof` / `tURec` aliases with solver-specific array types that add higher-level operators (initialization, boundary anchors, positivity-preserving limiters):

- `ArrayDOFV<N>` inherits from `CFV::tUDof<N>` (= `ArrayDof<N,1>`).
- `ArrayRECV<N>` inherits from `CFV::tURec<N>` (= `ArrayDof<DynamicSize,N>`).

```
template <EulerModel model>
class EulerSolver {
    typedef EulerEvaluator<model> TEval;
    static const int nVarsFixed = TEval::nVarsFixed;

    MPIInfo                                mpi;
    ssp<Geom::UnstructuredMesh>            mesh, meshBnd;
    TpVFV                                   vfv;              // VariationalReconstruction
    ssp<Geom::UnstructuredMeshSerialRW>    reader, readerBnd;
    ssp<EulerEvaluator<model>>             pEval;
    ssp<BoundaryHandler<model>>            pBCHandler;

    // Solver state (DOF arrays)
    ArrayDOFV<nVarsFixed>                  u, uIncBufODE, wAveraged, uAveraged;
    ObjectPool<ArrayDOFV<nVarsFixed>>      uPool;            // rent/return buffers
    ArrayRECV<nVarsFixed>                  uRec, uRecLimited, uRecNew, uRecNew1,
                                           uRecOld, uRec1, uRecInc, uRecInc1,
                                           uRecB, uRecB1;
    JacobianDiagBlock<nVarsFixed>          JD, JD1, JDTmp, JSource, JSource1, JSourceTmp;
    ssp<JacobianLocalLU<nVarsFixed>>       JLocalLU;
    ArrayDOFV<1>                           alphaPP, alphaPP1, betaPP, betaPP1,
                                           alphaPP_tmp, dTauTmp;

    // Config + output
    Configuration                          config;           // nested sub-configs
    nlohmann::ordered_json                 gSetting;
    std::string                            output_stamp;
    // ... outDist* / outSerial* / outDist2SerialTrans* for VTK
};
```

`Configuration` — everything that tunes a run

Every sub-section uses `DNDS_DECLARE_CONFIG` so the full JSON schema is auto-generated.

- **`TimeMarchControl`** — `dtImplicit`, `nTimeStep`, `steadyQuit`, `useRestart`, `useImplicitPP`, `odeCode`, `odeSetting1..4`, `odeSettingsExtra` (opaque JSON), `dtCFLLimitScale`, …
- **`ImplicitReconstructionControl`** — `useExplicit`, `nInternalRecStep`, `recLinearScheme` (0 = SOR, 1 = GMRES), `nGmresSpace/Iter`, `fpcgReset*`, `recThreshold`.
- **`OutputControl`** — `outputIntervalStep`, `outputFormat` (VTK, PLT, VTKHDF, series), parallel vs serial write.
- **`CFLControl`** — initial / max CFL, ramping schedule.
- **`ConvergenceControl`** — residual thresholds, monitor variables.
- **`DataIOControl`** — read/write paths, restart checkpointing.
- **`BoundaryDefinition`** — per-face-zone BC types, free-stream state.
- **`LimiterControl`** — `limiterProcedure`, `usePPRecLimiter`, WBAP order.
- **`LinearSolverControl`** — `gmresCode`, Krylov sub-space, iterations.
- **`TimeAverageControl`** — long-time averaging for statistics.
- **`EvaluatorSettings`** wraps `EulerEvaluatorSettings<model>`.
- **`VFVSettings`** wraps `VRSettings`.

> `--emit-schema` dumps the entire tree as a single JSON Schema document — `euler_schema.json` / `eulerSA3D_schema.json` / etc., each ~107 KB.

`EulerEvaluator<model>` — the spatial operator

```
void EvaluateRHS(ArrayDOFV<nVarsFixed>            &rhs,
                 JacobianDiagBlock<nVarsFixed>    &JSource,
                 ArrayDOFV<nVarsFixed>            &u,
                 ArrayRECV<nVarsFixed>            &uRecUnlim,
                 ArrayRECV<nVarsFixed>            &uRec,
                 ArrayDOFV<1>                     &uRecBeta,
                 ArrayDOFV<1>                     &cellRHSAlpha,
                 bool  onlyOnHalfAlpha,
                 real  t,
                 uint64_t flags = RHS_No_Flags);
```

Flags

- `RHS_Ignore_Viscosity`
- `RHS_Dont_Update_Integration`
- `RHS_Dont_Record_Bud_Flux`
- `RHS_Direct_2nd_Rec` — bypass VR, use GG-based 2nd-order
- `RHS_Direct_2nd_Rec_1st_Conv` — 2nd-order rec but 1st-order convective
- `RHS_Direct_2nd_Rec_use_limiter`
- `RHS_Direct_2nd_Rec_already_have_uGradBufNoLim`
- `RHS_Recover_IncFScale`

Flags compose bitwise — they cover fallback / diagnostic modes used by p-MG and PP sub-steps.

Other top-level calls

- `EvaluateDt(...)` — CFL-based local dt, spectral-radius based.
- `EvaluateURecBeta` — PP limiter β per cell.
- `EvaluateCellRHSAlpha` — per-cell RHS scaling for PP.
- `LimiterUGrad` — gradient limiter, optional shock detection.
- `LUSGSMatrixInit/Vec/ToJacobianLU` and `UpdateSGS(WithRec)`.
- Wall distance: `GetWallDist_AABB`, `GetWallDist_BatchedAABB`, `GetWallDist_Poisson`.
- Viscosity: `muEff(U, T)` with Sutherland or constant models.

Boundary conditions — strategy pattern

Each BC is a class implementing a common interface; `BoundaryHandler<model>` routes face-zone IDs to BC instances at runtime.

BC	Use
`BCWall`	No-slip wall (adiabatic)
`BCWallIsothermal`	No-slip wall at fixed temperature
`BCWallInvis`	Slip / symmetry
`BCSym`	Explicit symmetry plane
`BCFarField`	Riemann-invariant farfield
`BCIn`	Specified inflow
`BCOut` / `BCOutP`	Specified outflow / pressure-outflow
`BCPeriodic`	Standard periodic
`BCPeriodicRot`	Rotating periodic (turbomachinery)
`BCProfileIn`	Tabulated profile (boundary layer, RANS)
`BCActuator`	Actuator disk source term

Specialized turbomachinery BCs: `BCTotalInlet`, `BCRadialEqOutlet`, `BCMixingPlane`, and the **CL driver** for AoA-adaptive lift matching (`pCLDriver` in the evaluator).

Canonical benchmarks — Riemann, shocks, smooth

Riemann / blast

- **Sod** `euler_config_1DRiemann.json`
- **LeBlanc** `euler_config_1DRiemann_LeBlanc.json`
- **Sedov 1D** `euler_config_1DSedov.json`
- **Sedov 2D** `euler_config_2DSedov.json`
- **Noh (3D)** `euler3D_config_Noh.json`
- **Cylindrical blast** `euler_config_blast.json`
- **M2000 astrophysical jet** `euler_config_M2000Jet.json`

Hypersonic / shock interaction

- **Double Mach Reflection** 2D + 3D
- **M5 shock diffraction** `euler_config_M5Diffraction.json`
- **Hypersonic cylinder** `euler_config_cylinderHS.json`
- **Double ellipse / double cone** 3D
- **Sphere shock** `euler3D_config_SphereShock.json`

Smooth / steady / unsteady

- **Isentropic Vortex** `euler_config_IV.json` — convergence study
- **Taylor-Green Vortex 3D** `euler3D_config_TGV.json`, `euler3D_config_BenchTGV.json`
- **Lid-driven cavity** (incl. hypersonic variant)
- **Von Kármán vortex street** 2D + 3D
- **Laminar flat-plate BL**
- **Inviscid cylinder (MG bench)** `config_cylinderInvis_mg_bench.json`

Rotating / periodic frames

- Rotating-frame simple convergence test
- Rotating-periodic Isentropic Vortex

Aerospace & industrial benchmarks

External aerodynamics

- **NACA 0012** — SA (`eulerSA_config_0012_AOA15.json`) and k-ω (`euler2EQ/...`) variants, with O2 elevation (`..._Elev.json`) and MG benchmarks (`config_0012_mg_bench.json`).
- **30p30n** high-lift `eulerSA_config_30p30n.json` .
- **NASA CRM** — regular + CRM-HL high-lift.
- **DLR-F6** transport wing-body.
- **DPW-W1** drag-prediction wing.
- **Periodic hill** — LES vs RANS comparison.

Turbomachinery

- **Rotor 37** transonic compressor `eulerSA3D_config_Rotor37.json` .
- **Axial fan A1** `eulerSA3D_config_FanA1.json` .

New solver features in v0.2.0

ODE & preconditioning

- **HM3 revamp** — U2R2 / U2R1 / U3R1 modes, `tpMG`, `incFScale`, positivity-preserving coupling with `LimiterUGrad`.
- **ESDIRK2 / ESDIRK3 / Trapezoidal** added.
- **ILU-OMP** preconditioner.

Turbulence

- **DES → DDES → IDDES** progression on SA.
- **ψ-term fixes**, rotation correction variants, ft2 toggle.
- **k-ω two-equation** model with dedicated `euler2EQ / euler2EQ3D` executables.
- **BCProfileIn** for RANS inlet profiles.

Flux / limiter / BC

- **Roe_M8** flux; **HLLE+** (experimental).
- **incFScale** (incremental flux scaling) integrated into entropy fix.
- **Isothermal wall BC** (`BCWallIsothermal`).
- **Axisymmetric wedge** metric in reconstruction.
- **Positivity-preserving** reconstruction limiter in `LimiterUGrad`.

Physics

- **Rotating frames** (periodic + simple convergence).
- **Overset grid exploration** — hole cutting, distance map, cell-cell connectivity (2D demo).

Workflow

- `source2nd`, `mergeMultiResidual`, `normOrd`, `restartOutAtInit`, `resBaseType` options.

The main loop — `RunImplicitEuler`

```
void RunImplicitEuler() {
    InitializeRunningEnvironment(env);
    // optional restart
    if (config.restartState.useRestart)
        ReadRestart(config.dataIO.readRestart);

    for (int step = 1; step <= config.timeMarch.nTimeStep; ++step) {
        EvaluateDt(dt, u, uRec, CFL, dtMinAll, config.timeMarch.dtImplicit,
                   config.cflControl.useLocalDt, t);

        // Inner pseudo-time loop (driven by the chosen ODE integrator)
        odeIntegrator.Step(
            u, uInc,
            /*frhs*/     [&](rhs, u, dTau, iter, alpha, upos) { pEval->EvaluateRHS(...); },
            /*fdt */     [&](u,   dTau, alpha, upos)          { pEval->EvaluateDt(...); },
            /*fsolve*/   [&](x, rhs, uInc, dTau, alpha, ...)  { Krylov + LUSGS; },
            maxInnerIter, fStop, fIncrement, config.timeMarch.dtImplicit);

        UpdateCFL();
        if (step % config.outputControl.outputIntervalStep == 0)
            PrintData(fname, series, …);
        if (step % config.outputControl.restartInterval == 0)
            PrintRestart(fname);
        if (Converged() && config.timeMarch.steadyQuit) break;
    }
}
```

The lambdas above are where `EulerEvaluator`, `GMRES_LeftPreconditioned`, and `LUSGSMatrix*` plug in — the ODE integrator never knows which solver is instantiating it.

CHAPTER 8

Engineering Quality

Testing · build · docs · code hygiene

Test suite at a glance

Module	C++ executables	test cases	Python tests	np values
DNDS	8	249	9	1, 2, 4, 8
Geom	9	193	2	1, 2, 4, 8
CFV	4	67	43	1, 2, 4, 8
Euler	4	62	4	1, 2, 4, 8
Solver	4	29	—	1

Totals. 29 C++ executables, 600 test cases, 58 Python tests across 82 CTest registrations. All MPI-aware tests are CTest-registered at each np value. Serial tests have a 60–120 s timeout; parallel tests 120–600 s depending on module.

```
# Build + run everything
cmake -B build -DDNDS_BUILD_TESTS=ON
cmake --build build -t all_unit_tests -j8
ctest --test-dir build --output-on-failure
```

C++ test catalogue (1 / 2)

DNDS

- `test_array` — layouts, row views, iterators
- `test_mpi` — MPI wrapper, collective ops
- `test_array_transformer` — father/son ghost exchange
- `test_array_derived` — AdjacencyRow, EigenMap rows
- `test_array_dof` — vector-space ops, norms, AXPY
- `test_index_mapping` — global ↔ local, EvenSplit
- `test_serializer` — H5 + JSON, redistribute
- `test_permutation_transfer` — MPL renumber compression / decompression

Geom

- `test_elements` — shape functions, jacobians
- `test_quadrature` — orders, weights
- `test_mesh_index_conversion` — state transitions
- `test_mesh_pipeline` — full build chain
- `test_mesh_distributed_read` — ParMetis repartition
- `test_mesh_connectivity` — Inverse / Compose DSL
- `test_mesh_connectivity_ghost` — GhostSpec BFS
- `test_mesh_connectivity_interpolate` — face interp
- `test_mesh_reorder` — reverse Cuthill-McKee / Hilbert ordering

Typical test structure

```
TEST_CASE("ArrayTransformer: round-trip ghost pull" *
          doctest::description("np=1,2,4") *
          doctest::timeout(120.0)) {
    MPIInfo mpi; mpi.setWorld();
    auto father = make_ssp<ParArray<real, 5>>();
    auto son    = make_ssp<ParArray<real, 5>>();
    father->Resize(localN);  father->createGlobalMapping();
    // ... populate father ...

    ArrayTransformer<real, 5> trans;
    trans.setFatherSon(father, son);
    trans.createFatherGlobalMapping();
    trans.createGhostMapping(pullGlobal);
    trans.createMPITypes();
    trans.initPersistentPull();
    trans.pullOnce();

    CHECK(son->operator[](0).isApprox(expected, 1e-14));
}
```

C++ test catalogue (2 / 2)

CFV

- `test_reconstruction` · tests of VR convergence on analytic fields.
- `test_reconstruction3d` · 3D variants; Jacobi/SOR comparison.
- `test_limiters` · WBAP / CWBAP on contrived data; exercises the full limiter menu.
- `test_device_transferable` *(CUDA only)* · round-trip of `FiniteVolume` to GPU and back.

Euler

- `test_gas_thermo` · ideal gas Cv/Cp, T/p relations, Mach → state.
- `test_riemann_solvers` · 13 variants, exact-solution agreement on 1D Riemann problems.
- `test_rans` · SA + k-ω source terms, wall distance integration, trip location.
- `test_evaluator_pipeline` · full `EvaluateRHS` on a fixed mesh — **golden values**.

Solver

- `test_ode` · BDF / SDIRK / HM3 on ODE benchmarks (Van der Pol, stiff scalar).
- `test_linear` · GMRES + PCG convergence on canonical matrices.
- `test_direct` · small-block LU / LDLT correctness.
- `test_scalar` · scalar transport advection-diffusion regression.

Determinism — how golden values stay stable

Many tests compare computed results against pre-captured **golden values** with relative tolerance `1e-6` to `1e-8`. For this to be meaningful, runs must be byte-stable across re-executions.

Sources of non-determinism — eliminated

- **Partitioning order** → `metisSeed = 42` (fixed).
- **SOR update order** (depends on partition) → Jacobi iteration used instead in VR tests.
- **LU-SGS sweep direction** (partition-ordered) → Jacobi-style updates in Euler pipeline tests.
- **OMP reduction order** (thread count) → scalar reductions are deterministic at fixed thread count.

Sentinel value pattern

When a golden value **has not yet been captured**, the test stores the sentinel `1e300`:

```
const real gold_kinetic = 1e300;           // TODO: capture
const real computed     = evaluate();
if (gold_kinetic < 1e299)
    CHECK(computed == doctest::Approx(gold_kinetic).epsilon(1e-8));
else
    CHECK(std::isfinite(computed) && computed >= 0);
```

So the first run of a new test is a finite/non-negative sanity check, and the developer updates the golden in a follow-up commit.

Python tests — pytest + pytest-mpi

What's covered

- `test/DNDS/test_basic.py` (9 tests) — import chain, MPIInfo, small array round-trip.
- `test/Geom/test_basic_geom.py` (2 tests) — CGNS read, elevation, bisection.
- `test/CFV/test_fv_correctness.py` (16 tests) — cell volume / face area / jacobian correctness on wall meshes.
- `test/CFV/test_vr_correctness.py` (16 tests) — VR order convergence on sin(x)sin(y).
- `test/CFV/test_basic_fv.py` + `test_basic_cfv.py` + `test_cfv_dissdisp.py` (11 tests) — FV/CFV smoke tests and dissipation-dispersion analysis.
- `test/EulerP/test_basic_eulerP.py` (1 test) — host + CUDA round-trip.
- `test/Euler/test_restart_redistribute.py` (3 tests) — solver restart with MPI repartition.

Running

```
# Serial
pytest test/DNDS/test_basic.py -v

# MPI
mpirun -np 4 python -m pytest test/DNDS/test_basic.py

# Some tests support standalone
python           test/DNDS/test_basic.py
mpirun -np 2 python test/DNDS/test_basic.py
```

Critical rebuild dance

```
# 1. Rebuild pybind11 shared libs
cmake --build build -t dnds_pybind11 geom_pybind11 \
                       cfv_pybind11 eulerP_pybind11 -j32

# 2. Reinstall into python/DNDSR/ (MANDATORY)
cmake --install build --component py

# 3. Only now, run tests
source venv/bin/activate
PYTHONPATH=<root>/python pytest test/ -v
```

> ⚠️ Skipping the install step after changing C++ source leaves stale `.so` files and produces misleading segfaults that look like code bugs. `git checkout` changes source but does **not** rebuild binaries.

docs/tests/overview.md:104-124

Build system — presets

```
{
  "configurePresets": [
    {
      "name": "release-test",
      "generator": "Ninja",
      "binaryDir": "${sourceDir}/build",
      "cacheVariables": {
        "CMAKE_BUILD_TYPE":    "Release",
        "DNDS_BUILD_TESTS":    "ON",
        "DNDS_USE_OMP":        "ON"
      }
    },
    { "name": "debug",  "inherits": "release-test",
      "cacheVariables": { "CMAKE_BUILD_TYPE": "Debug" } },
    { "name": "cuda",   "inherits": "release-test",
      "cacheVariables": { "DNDS_USE_CUDA": "ON",
                          "CMAKE_CUDA_ARCHITECTURES": "native" } },
    { "name": "ci",     "inherits": "release-test",
      "cacheVariables": { "DNDS_TEST_NP_LIST":     "1;2;4",
                          "DNDS_TEST_OMP_THREADS": "2" } }
  ]
}
```

Aggregate targets: `dnds_unit_tests`, `geom_unit_tests`, `cfv_unit_tests`, `euler_unit_tests`, `solver_unit_tests`, `all_unit_tests` — all `EXCLUDE_FROM_ALL` so plain `cmake --build` stays fast.

Python packaging — `scikit-build-core`

```toml
# pyproject.toml
[build-system]
requires = ["scikit-build-core>=0.8", "pybind11", "pybind11-stubgen"]
build-backend = "scikit_build_core.build"

[project]
name = "DNDSR"
version = "0.2.0"              # synchronized with VERSION file + git describe

[tool.scikit-build]
cmake.args = ["-DDNDS_BUILD_PYTHON=ON", "-DDNDS_PYBIND11_NO_LTO=ON"]
install.components = ["py"]     # only install the py component
```

Build & install

```
CC=mpicc CXX=mpicxx \
    CMAKE_BUILD_PARALLEL_LEVEL=32 \
    pip install -e .
```

- Builds all `*_pybind11` targets.
- Copies them into `python/DNDSR/*/_ext/`.
- Runs `pybind11-stubgen` to produce `.pyi` files.
- Copies external shared libs into `python/DNDSR/_lib/`.

Why system Python (not conda)

> Conda/Anaconda Python embeds an `RPATH` to conda's bundled libstdc++, which may be older than what the MPI compiler produces. System Python uses the system libstdc++ and avoids this conflict.
> — `README.md`

macOS has a dedicated fmtlib workaround, also shipped.

Clang-tidy sanitation

DNDS core — the cleanup

- **24 597 diagnostics** at start of the sweep.
- **26 passes** applied in careful slice order.
- **1 remaining** — an unrelated Eigen PCH `omp.h` include issue.
- Full per-pass record and `.clang-tidy` rationale preserved in `docs/dev/clang_tidy_plan.md`.

The `.clang-tidy` disables (representative)

- `cppcoreguidelines-pro-bounds-pointer-arithmetic` — unavoidable in CSR / row-flat arrays.
- `fuchsia-default-arguments-declarations` — MPI defaults.
- `llvm-header-guard` — we use `#pragma once`.
- `modernize-use-trailing-return-type` — style preference.

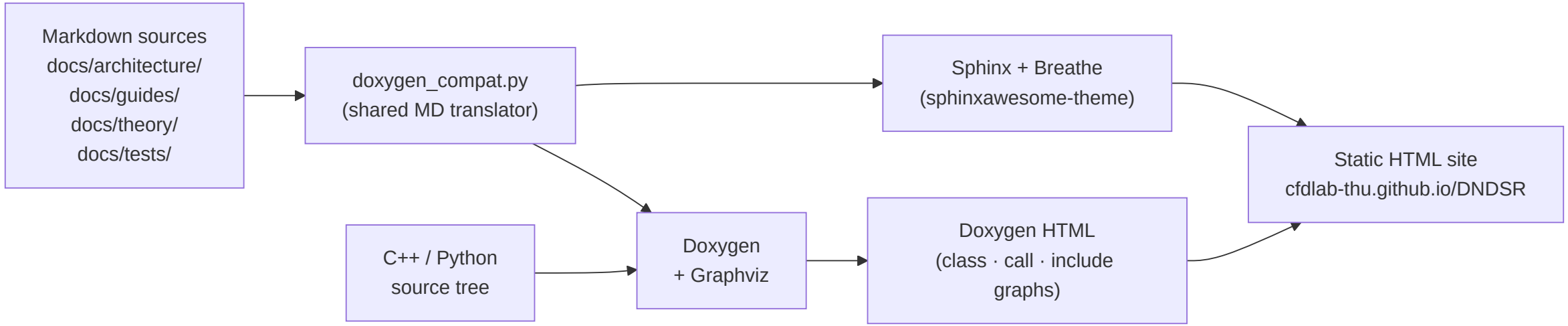
Running it yourself

```
# Per-module histogram
python scripts/run_clang_tidy.py DNDS
python scripts/run_clang_tidy.py Geom
python scripts/run_clang_tidy.py CFV
python scripts/run_clang_tidy.py Euler
python scripts/run_clang_tidy.py Solver
```

What's next

Solver / Geom / CFV / Euler / EulerP are **not yet sanitised** — same recipe to apply. The `.clang-tidy` disables carry forward.

Documentation system — architecture



Key features

- **One Markdown source** renders in both Sphinx and Doxygen via `doxygen_compat.py`.
- **Doxygen HTML** is embedded at `/doxygen/` on the Sphinx site.
- **Graphviz** class inheritance, call graphs, include graphs.
- **sphinxawesome-theme** with rich code highlighting.

Build speeds

Trigger	Time
No-op rebuild	**< 1 s**
Markdown-only edit	~10 s
Full (Doxygen + Sphinx)	~2.5 min

```
cmake --build build -t serve-docs
# → http://localhost:8000 with hot reload
```

CI & release automation

GitHub Actions — Pages deployment

- **Manual dispatch** workflow (avoid spending minutes on every push).
- **3-layer caching:**
  i. Ubuntu apt packages (doxygen, graphviz, libmpich-dev).
  ii. External `cfd_externals` binary libraries (HDF5, CGNS, Metis, ParMetis).
  iii. Python venv + Sphinx build cache.
- Cache hit → full docs build in ~3 minutes; cache miss → ~20 minutes.

Style + hygiene

- `.clang-format` ships at repo root; CI checks a diff in a separate job.
- `POSIX index()` ambiguity guard — code style requires `DNDS::index` whenever `using namespace DNDS;` is active (documented in `docs/tests/overview.md`).

Version string

- `VERSION` file at repo root (`0.2.0`).
- CMake combines it with `git describe --tags --long`.
- Exposed as:
  - C++ macro `DNDS_VERSION_STRING`.
  - Python `DNDSR.__version__` (PEP 440 compliant).
  - JSON schema `x-version` field.

Release workflow

```
# Bump VERSION file
echo 0.2.0 > VERSION
git tag v0.2.0
git push --tags
# Pages workflow + release notes kick off.
```

CHAPTER 9

Results

Selected 2D / 3D Euler & NS cases

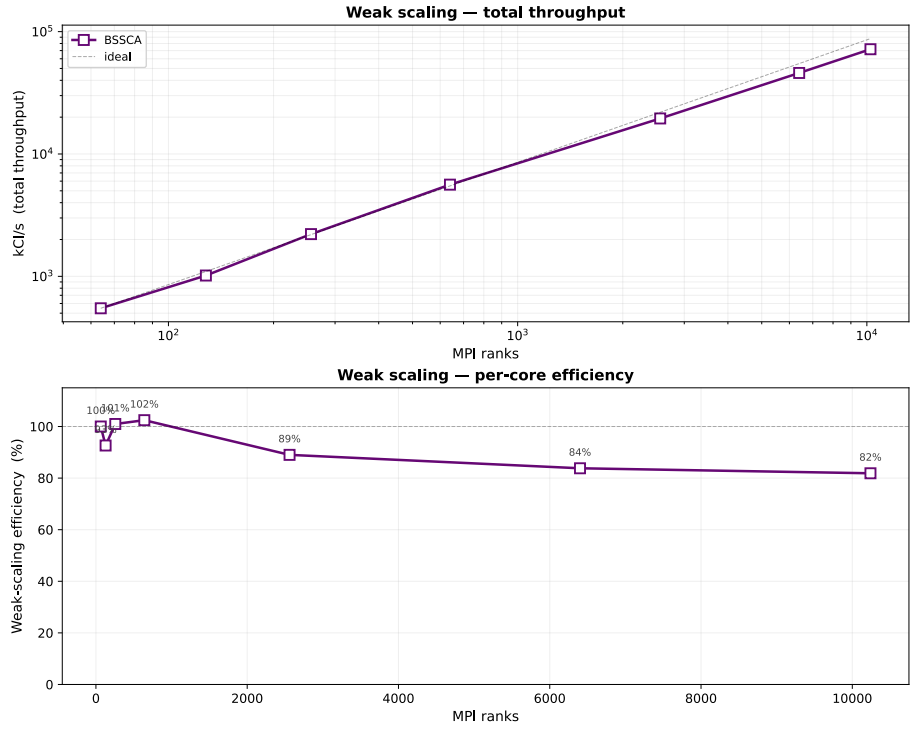
MPI weak scaling — TGV benchmark

Compressible Taylor–Green vortex at Re = 1600, 100 iterations, fixed **~4k cells per rank** on a single HPC node.

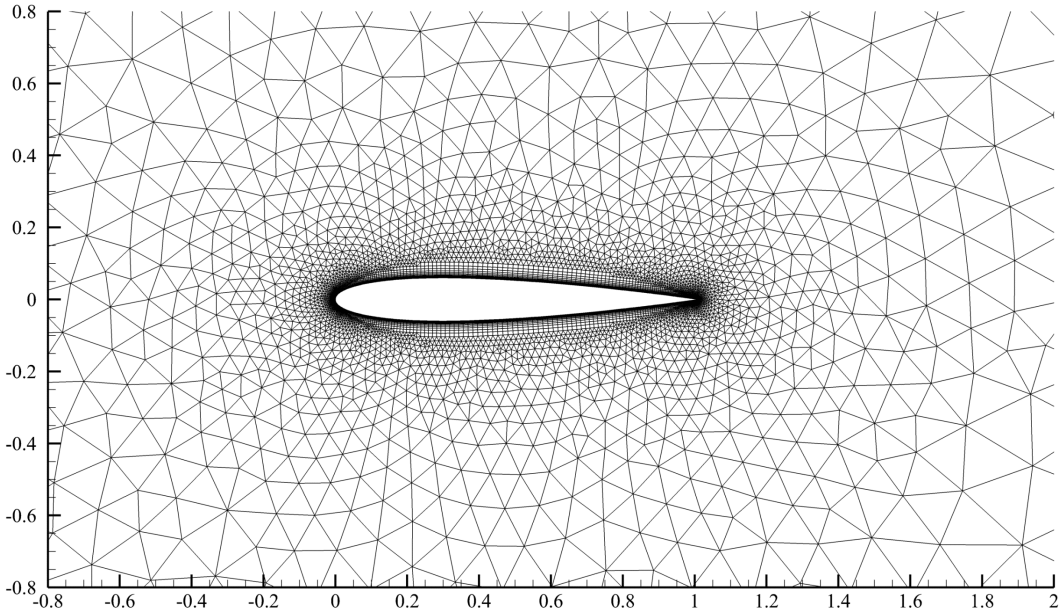
Series	Solver	
BSSCA	DNDSR /BSSCA	64 → 10240 ranks
BSSCT	DNDSR /BSSCA	96 → 1920 ranks
CS	DNDSR /JS	32 → 256 ranks

- **Throughput** scales from 548 kCI/s (64 ranks) to **71.8 MCI/s** (10240 ranks).
- Per-core efficiency holds at **82–102%** across two orders of magnitude of parallelism.

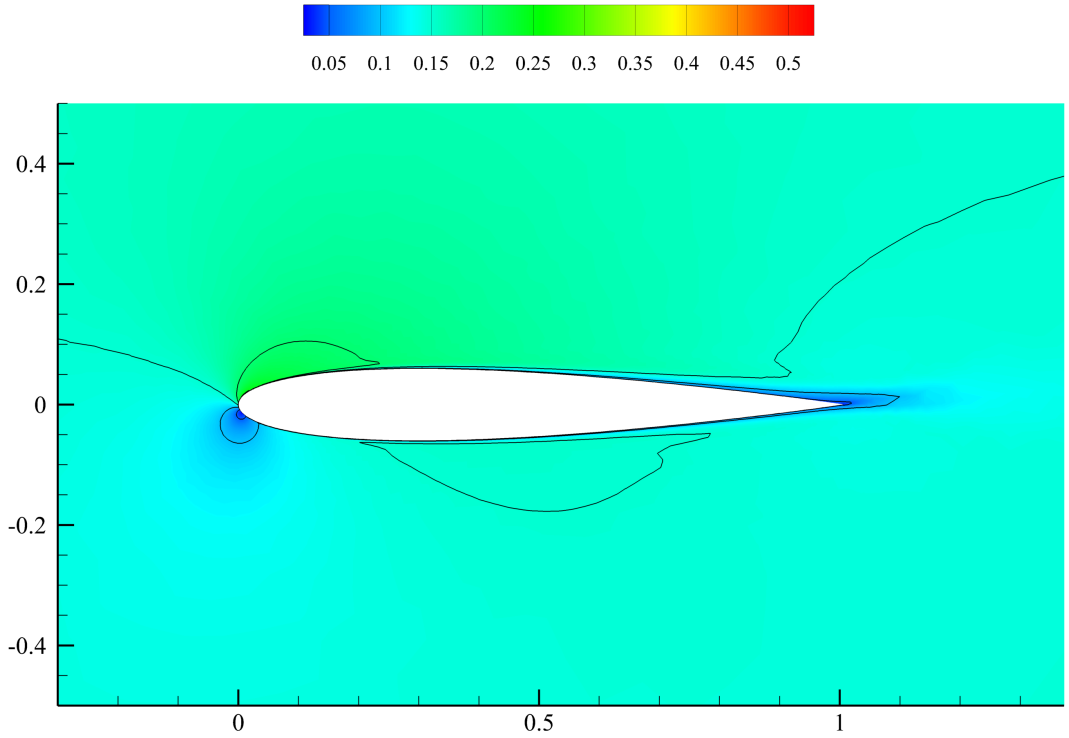
kCI/s = kilo cell-iterations per second; one cell-iteration is one RHS evaluation on one cell.



NACA 0012 (SA RANS)

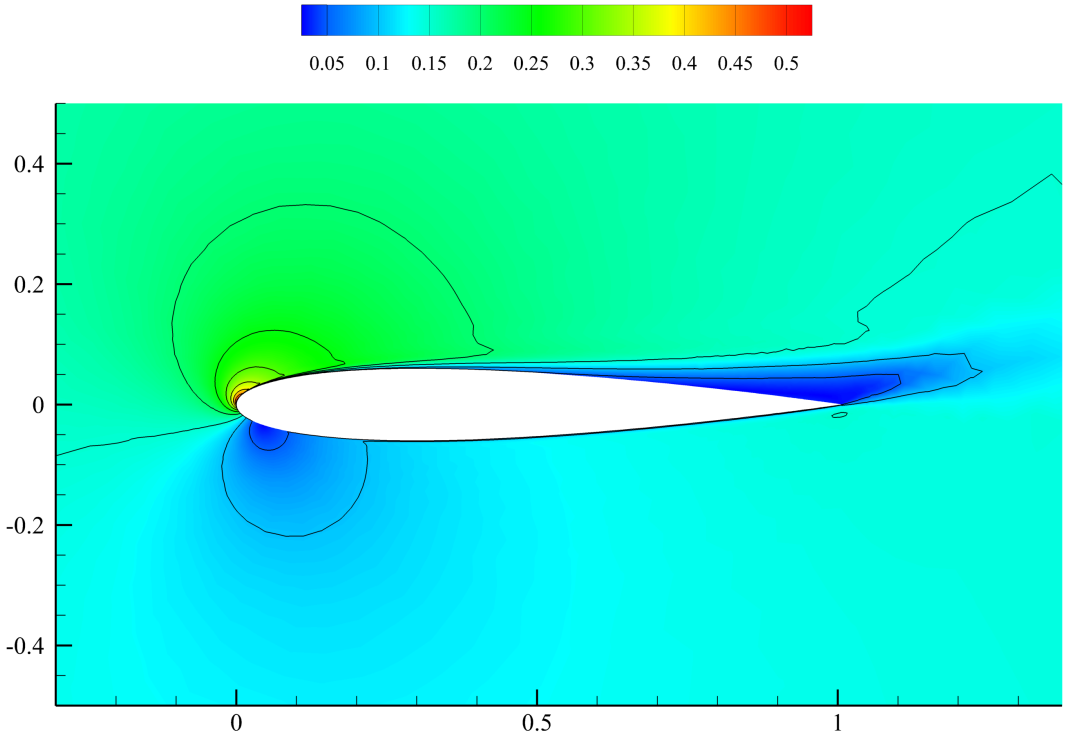


mesh

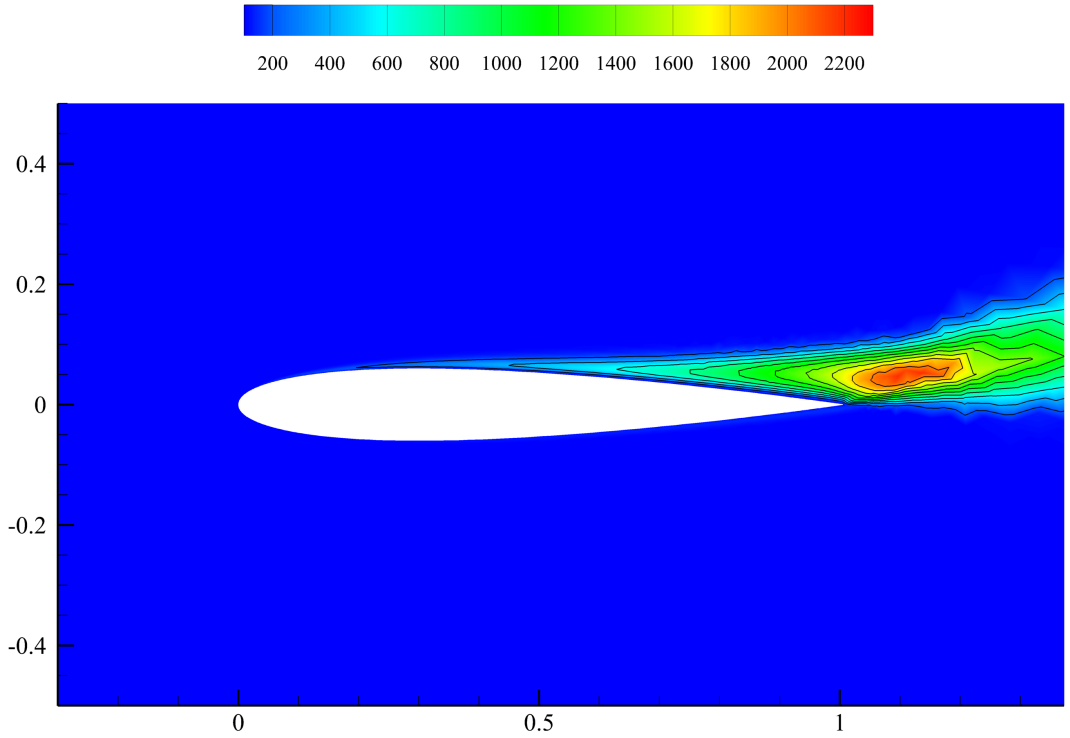


AOA = 5°, Mach number

NACA 0012 — AOA 15°



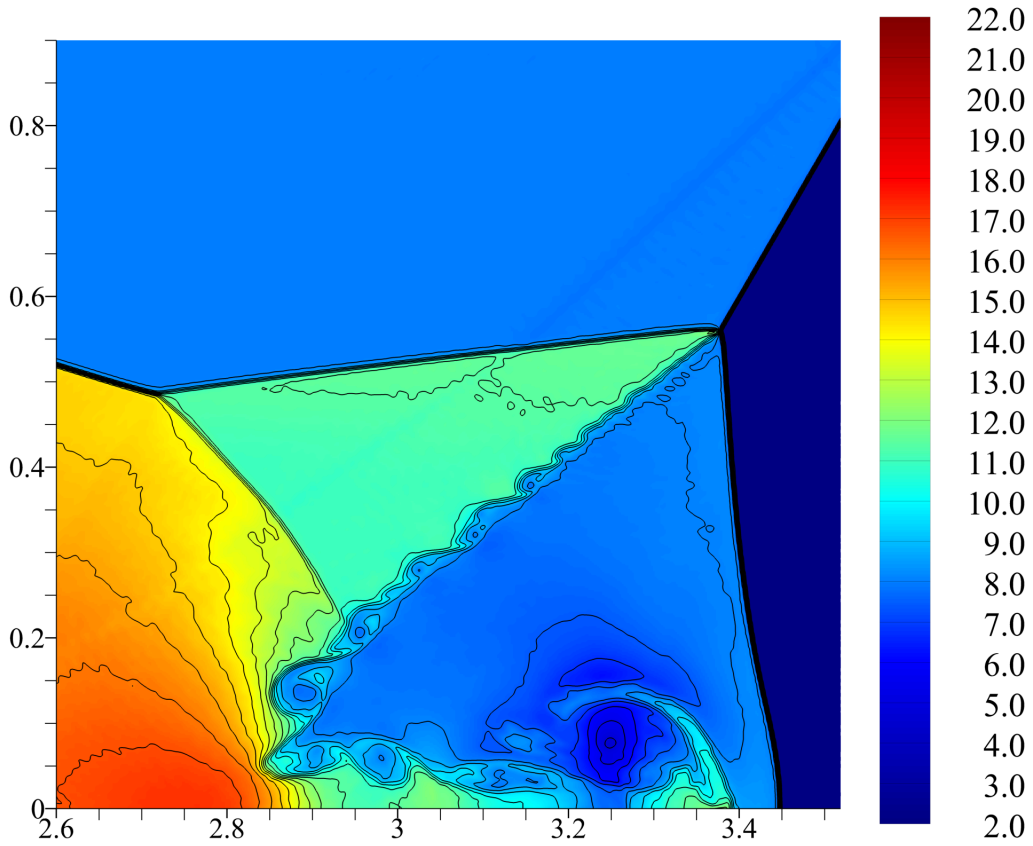
AOA = 15°, Mach number



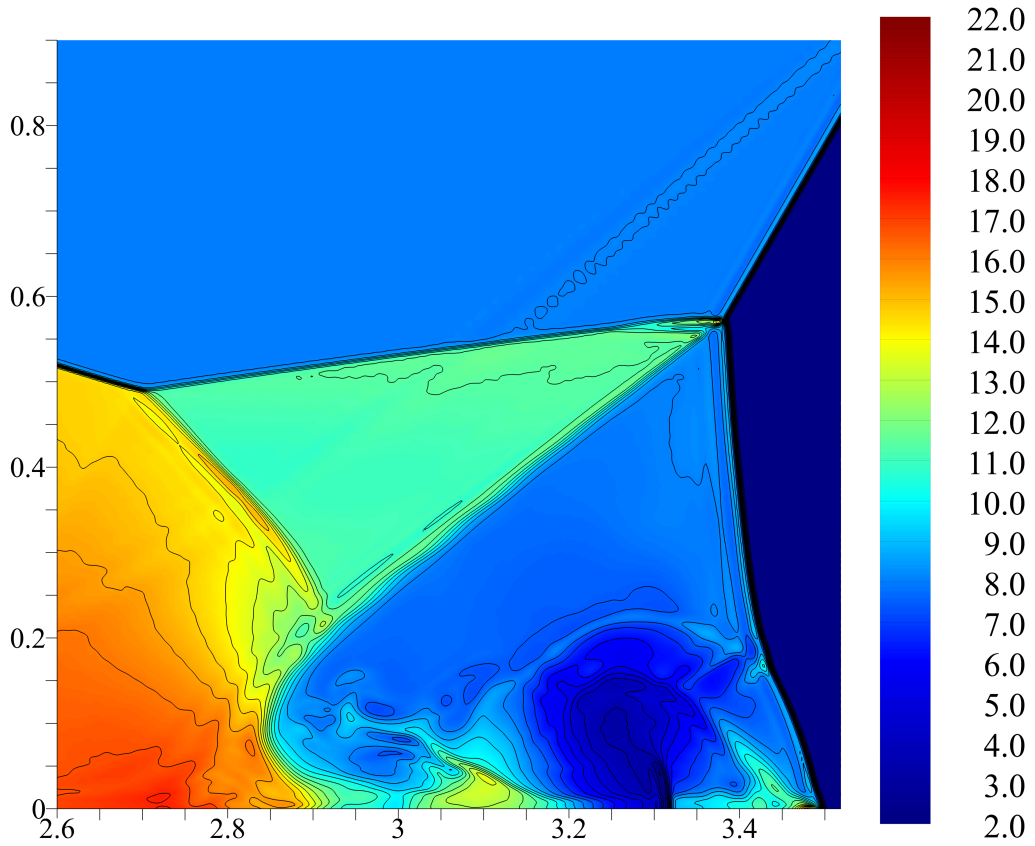
AOA = 15°, $\rho\tilde{\nu}$ field

Double Mach Reflection (2D Euler)

Density at t = 0.2, Mach 10 shock on 30° wedge.



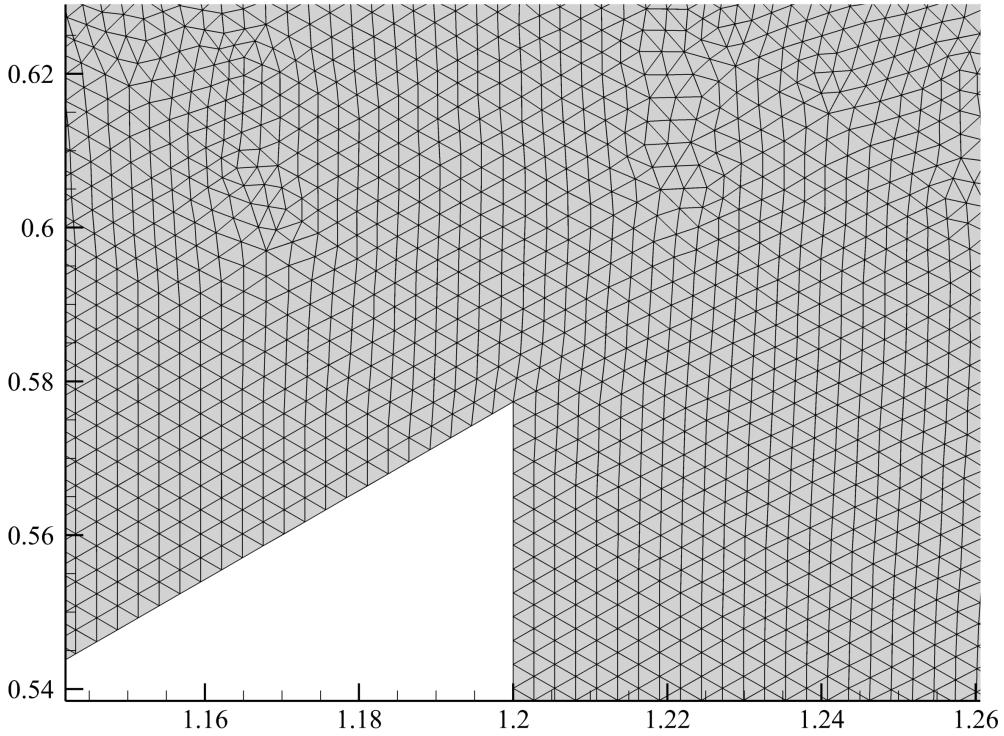
DITR U2R2, $c_2 = 0.55$



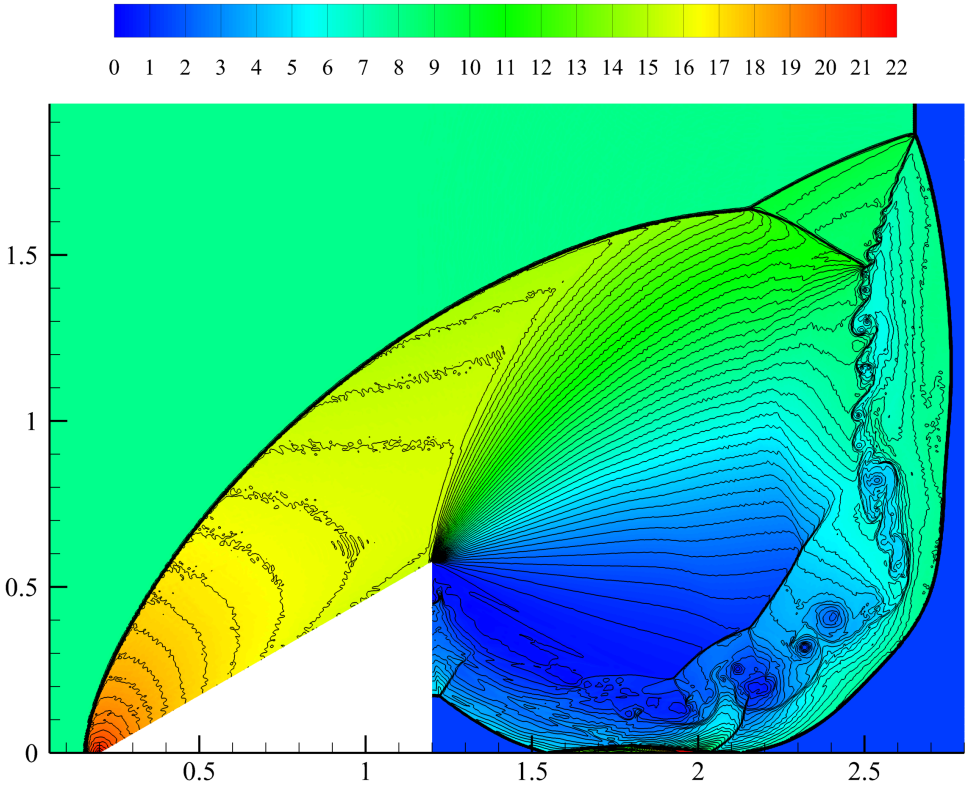
BDF2

Supersonic Wedge (2D Euler)

Mach 3 inviscid flow over a 15° compression corner.

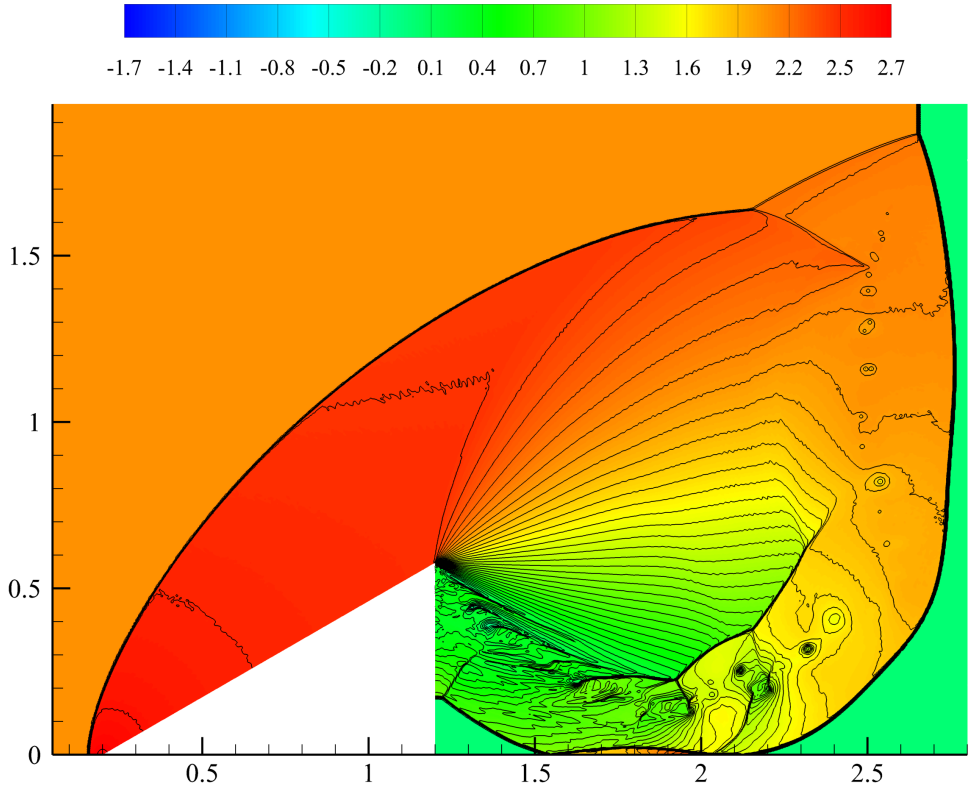


mesh

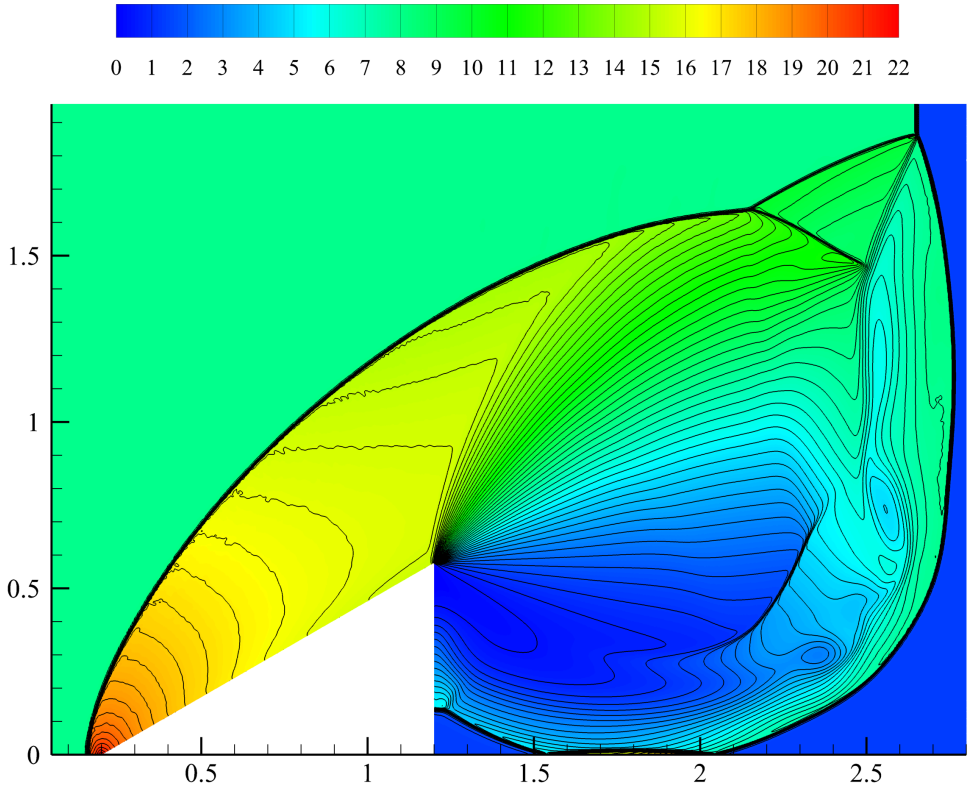


Density, Re = ∞

Wedge — viscous (Re = 100) + pressure



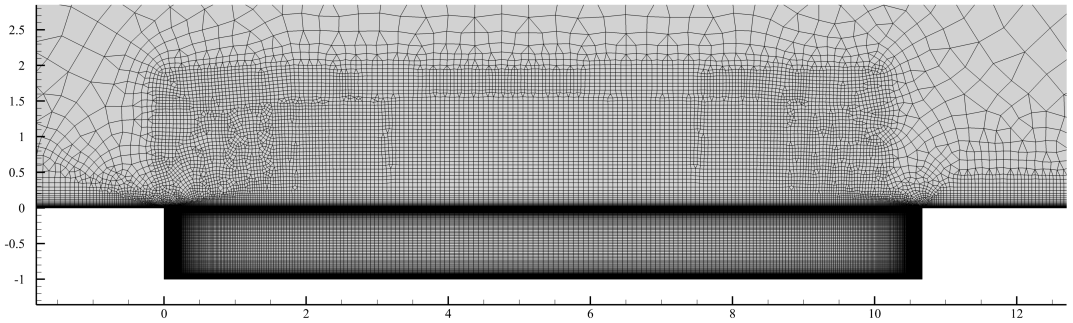
Pressure, $Re = \infty$



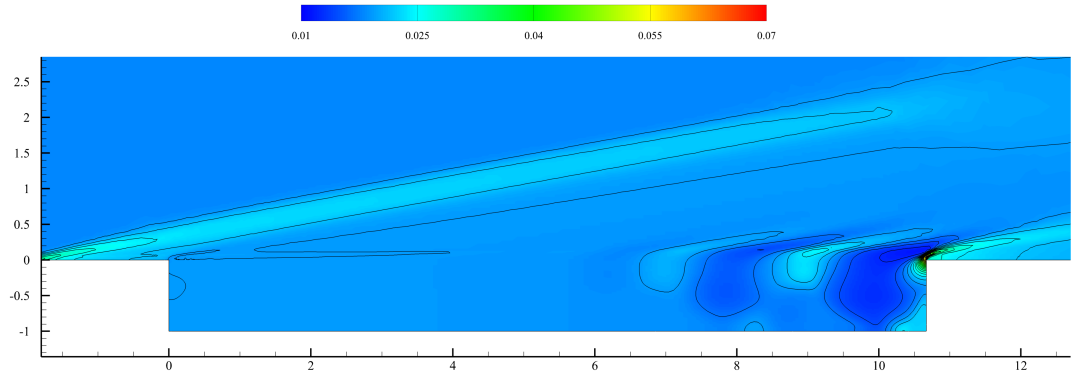
Density, Re = 100

Hypersonic Cavity (2D Navier–Stokes)

Time-averaged results.

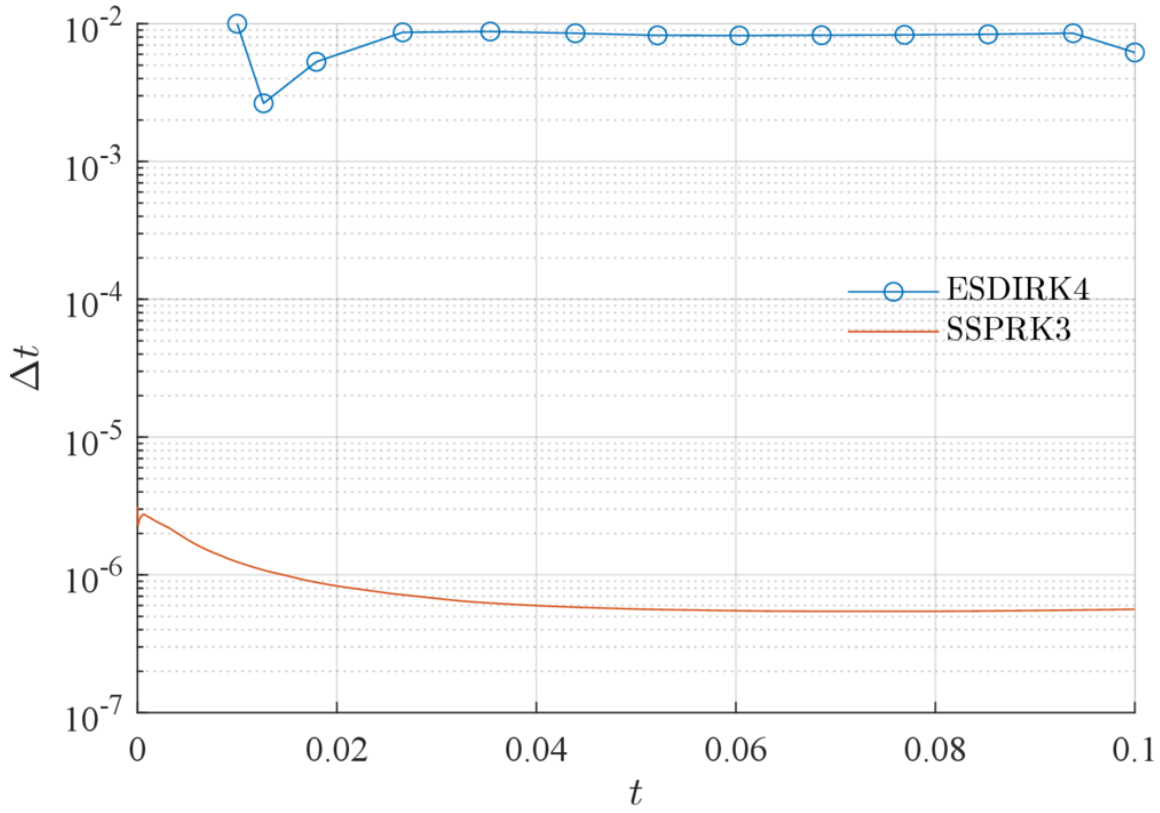


mesh

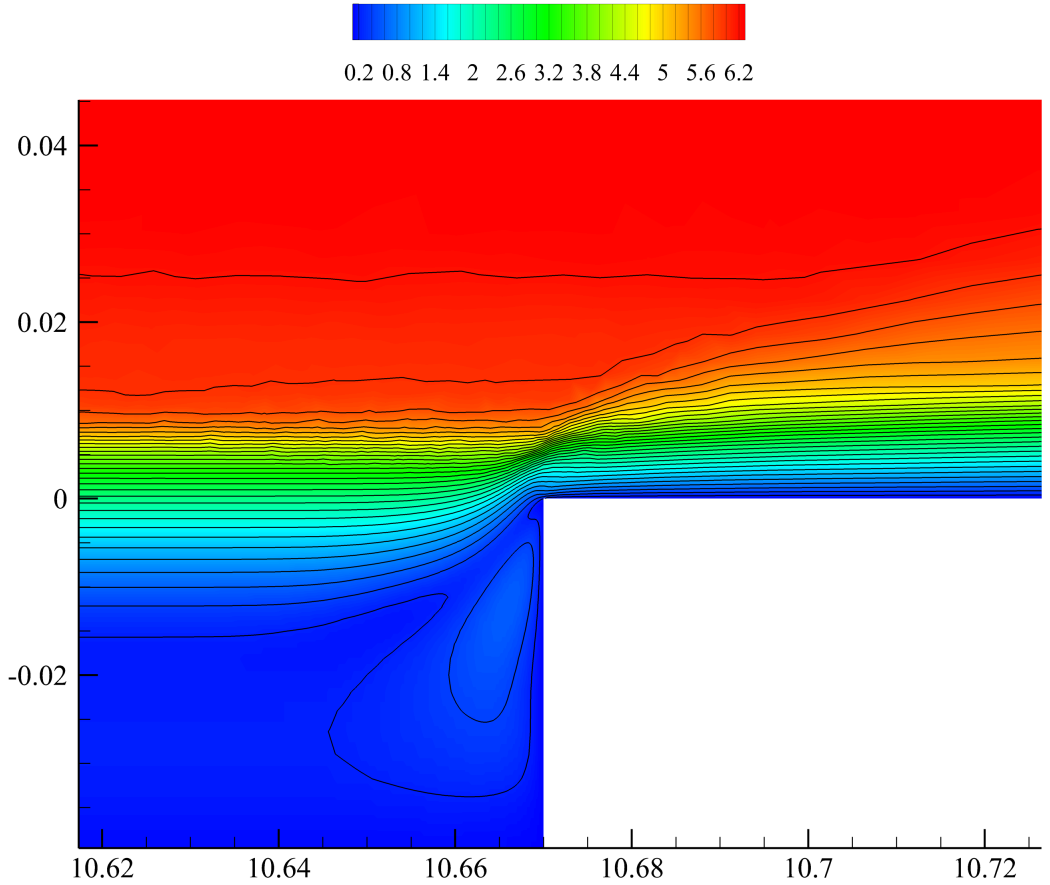


Time-averaged $p/(\rho_\infty U_\infty^2)$, 27 isolines

Hypersonic Cavity — time step + Ma = 0.1

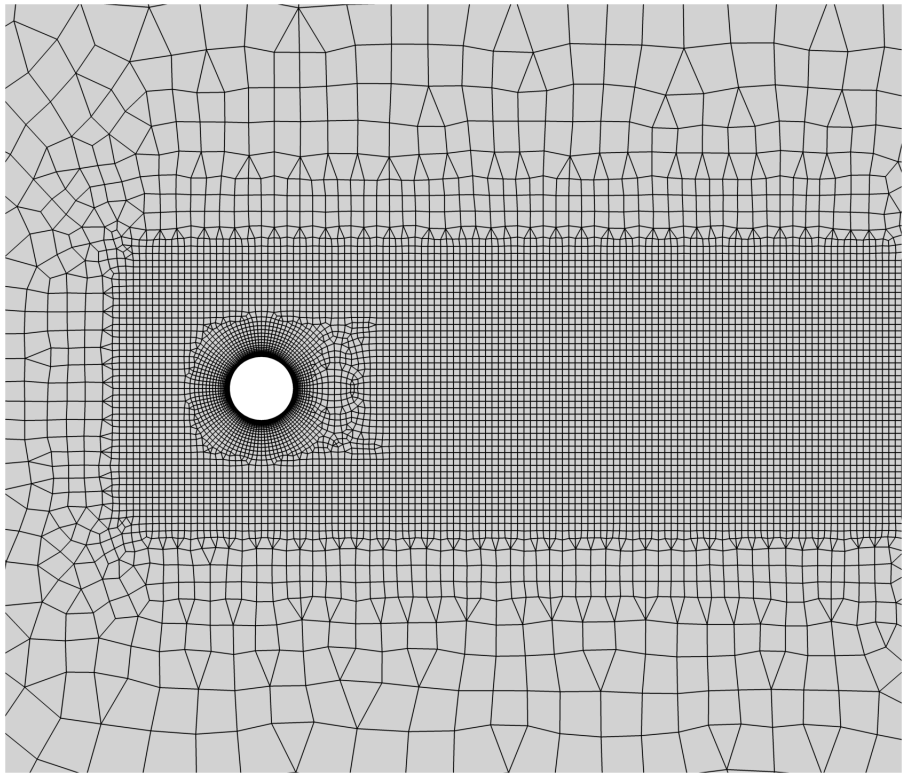


Explicit time step comparison

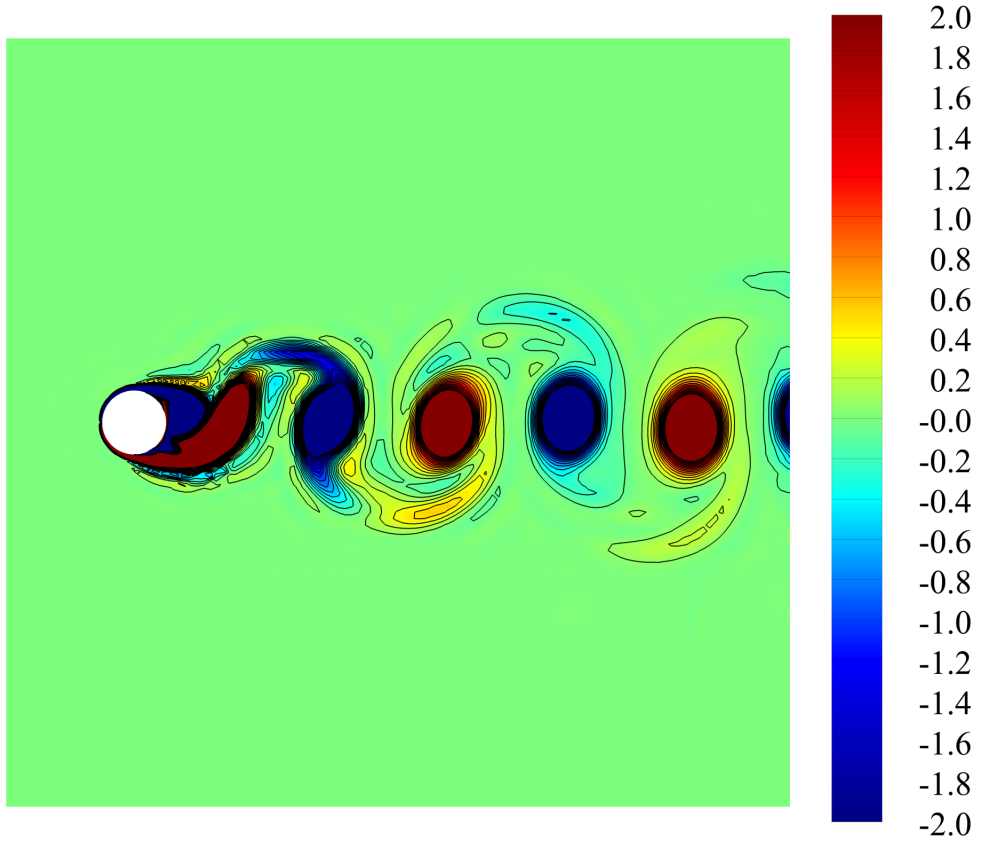


t = 0.1 Mach, 32 isolines (explicit 2nd-order FV)

Cylinder Re = 1200 (2D NS, unsteady)



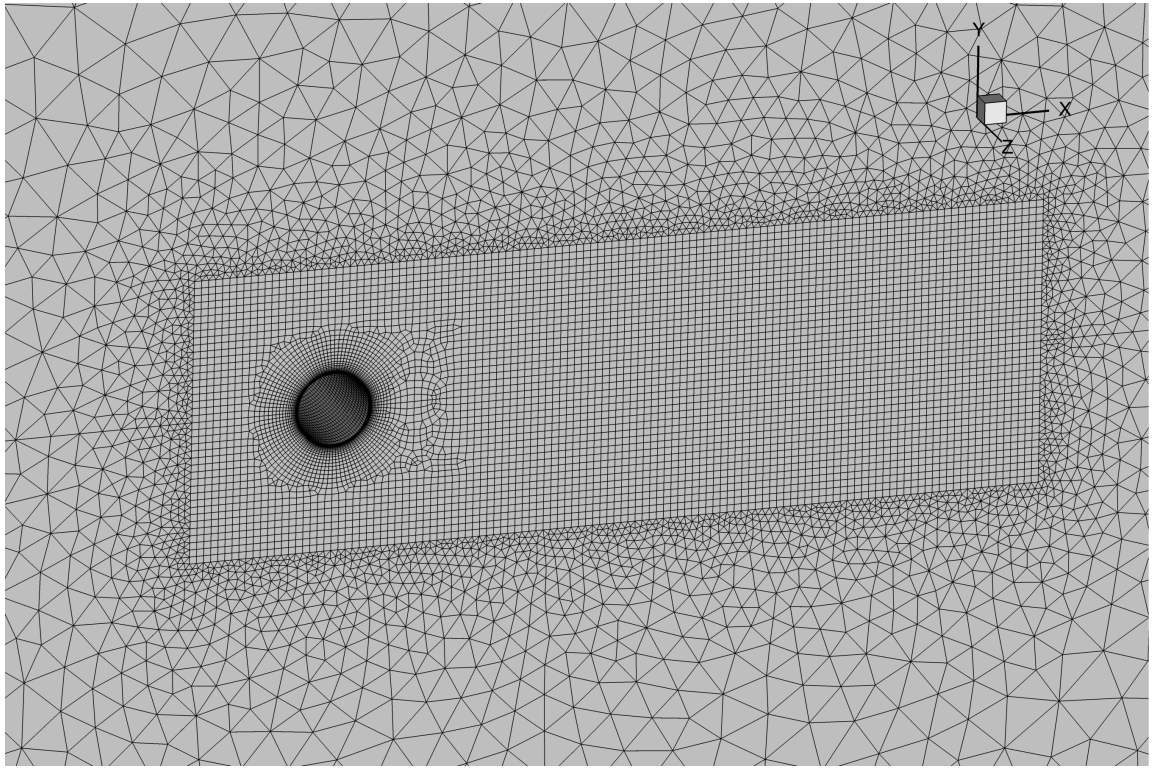
Mesh



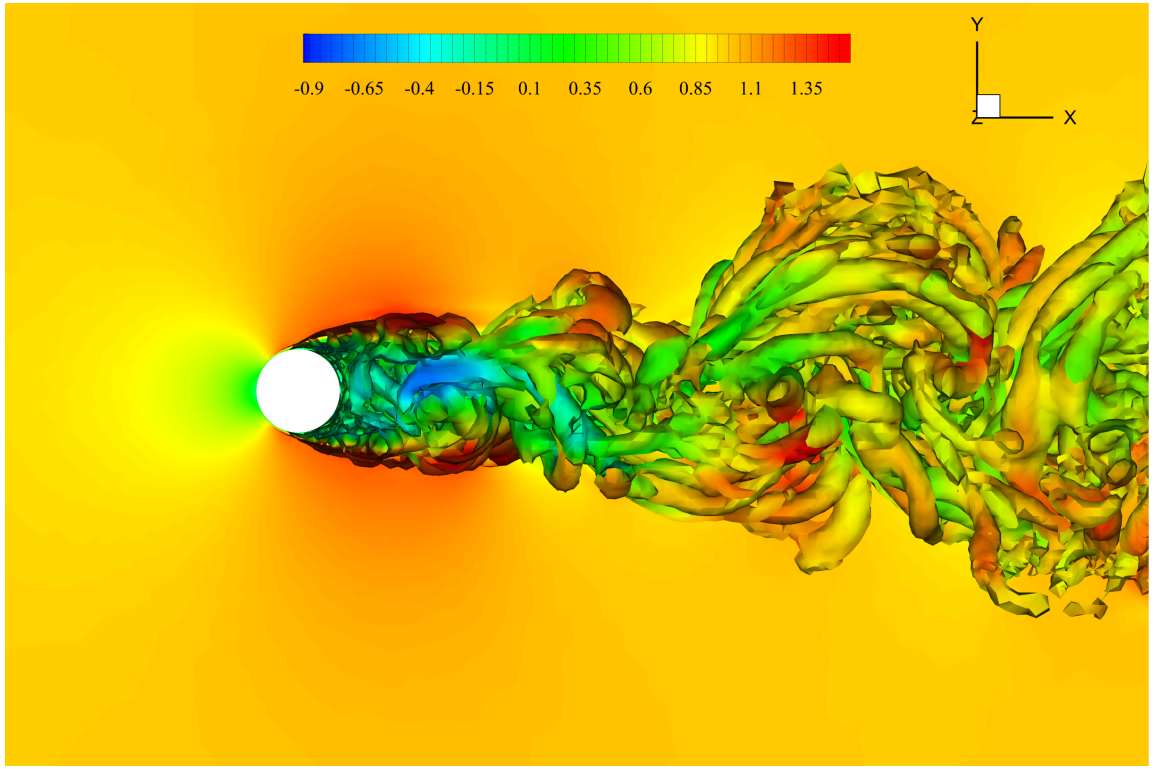
Vorticity

3D Turbulent Cylinder (ILES, Re = 1.2 × 10³)

Q-criterion iso-surfaces coloured by Mach number.

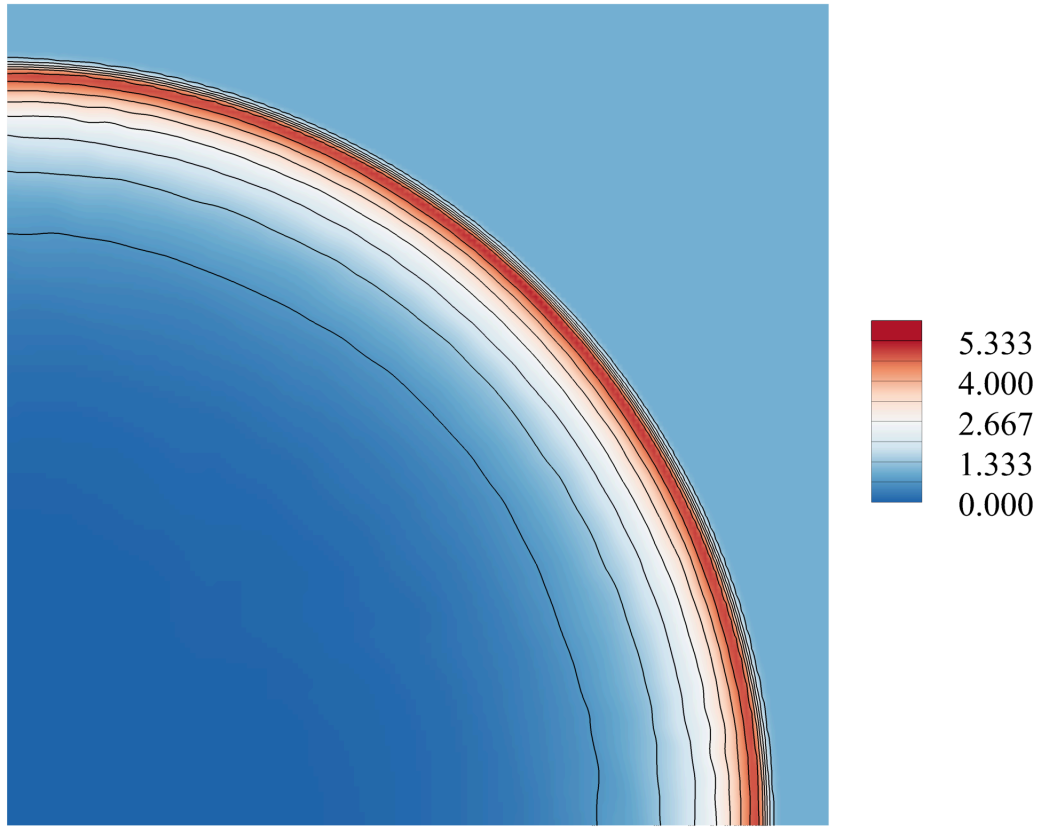


mesh

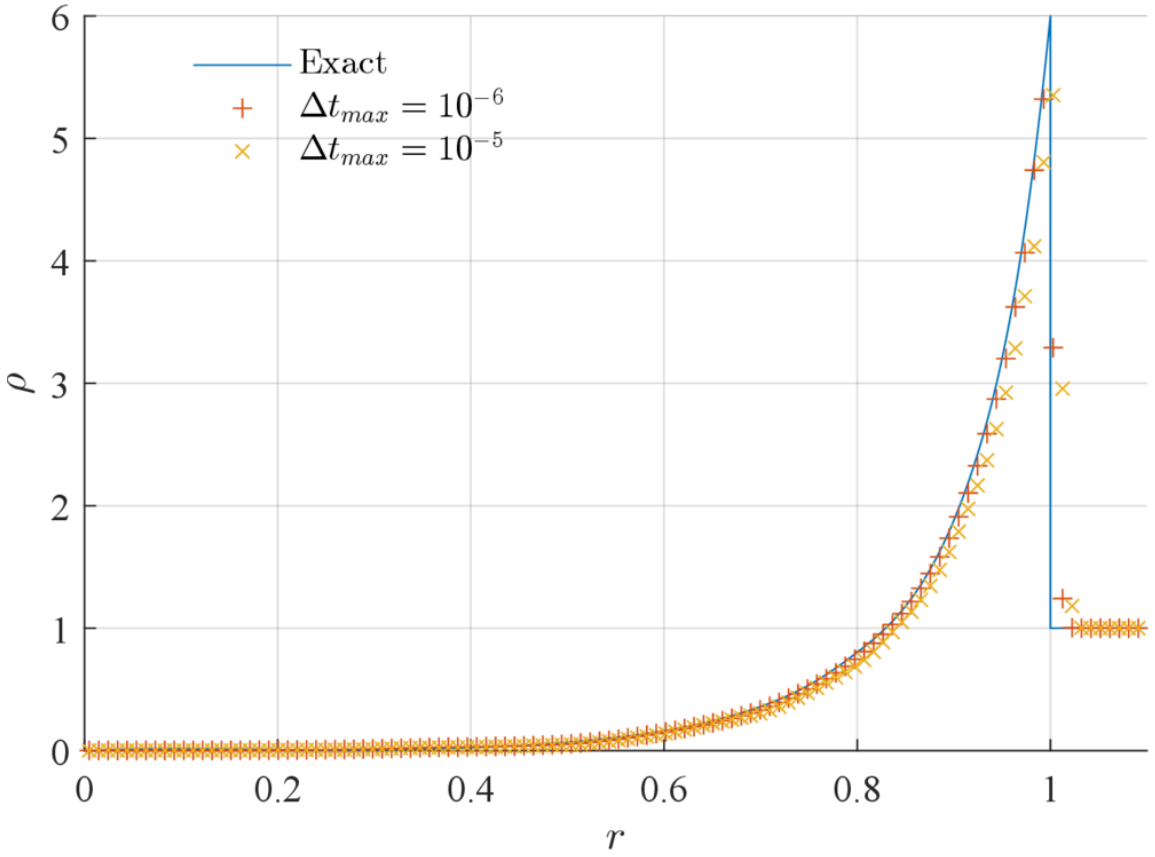


Q-criterion iso-surfaces

Sedov Blast Wave (2D Euler)

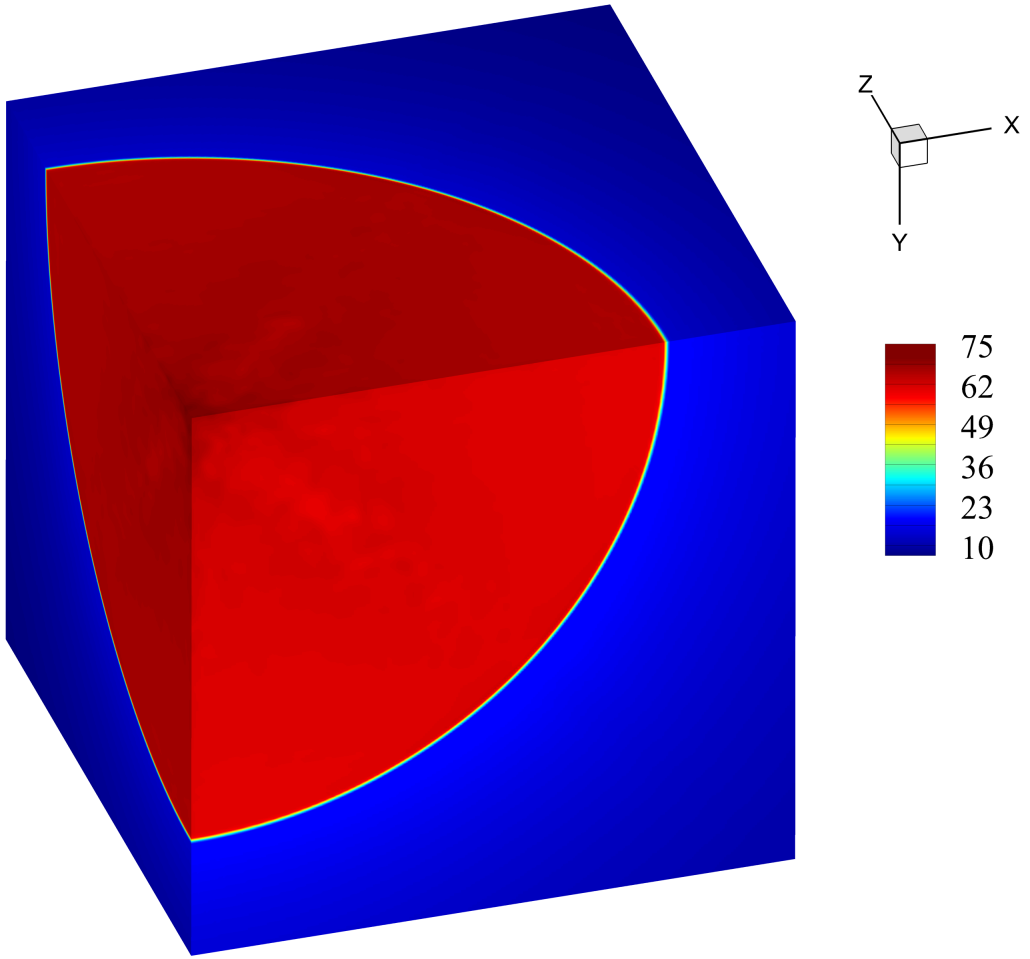


Density

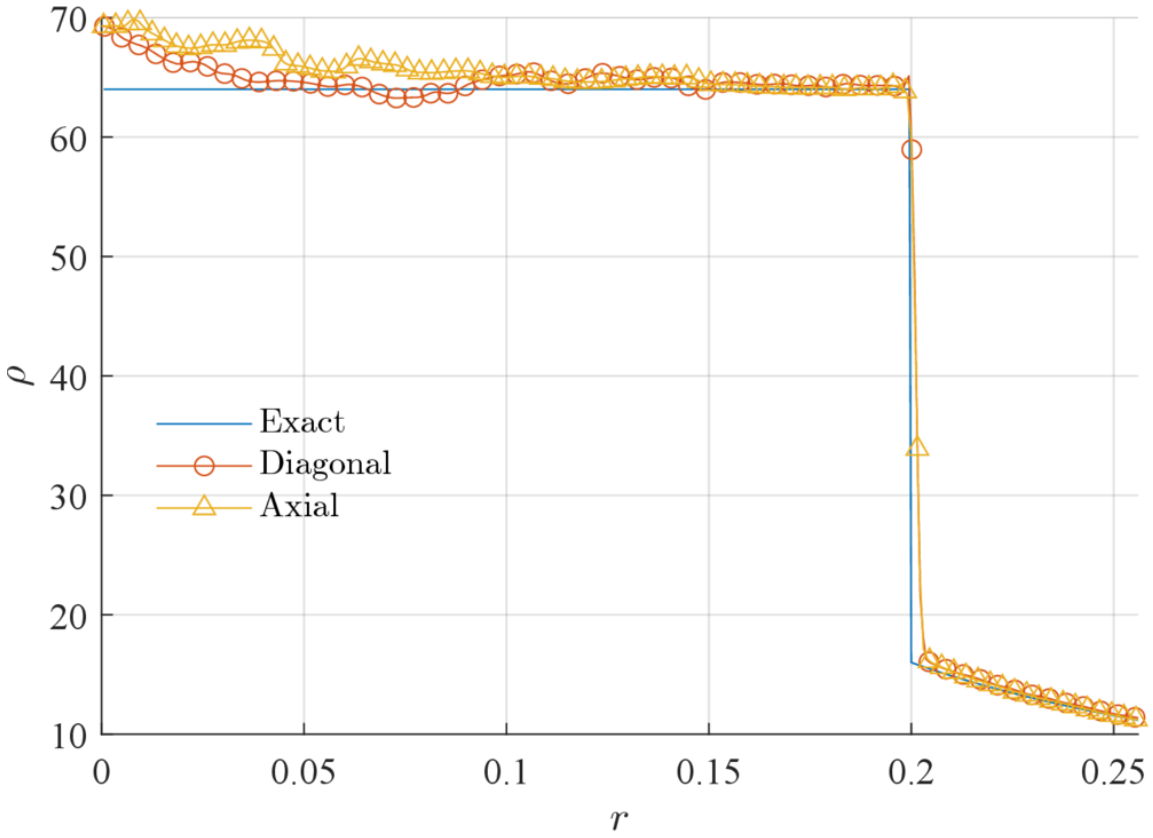


Density along diagonal

Noh Problem



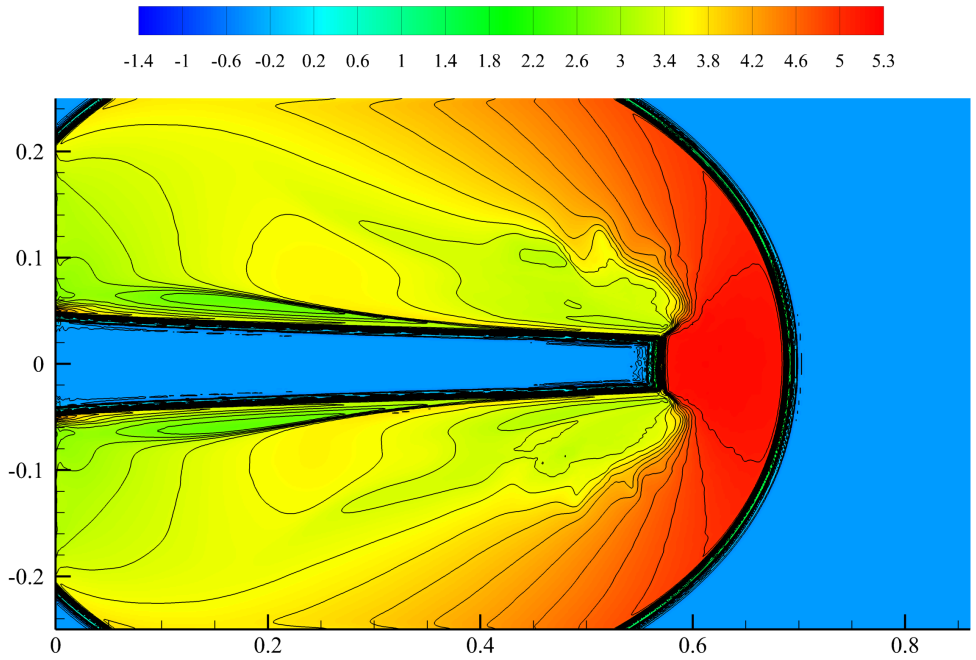
Density at t = 0.6



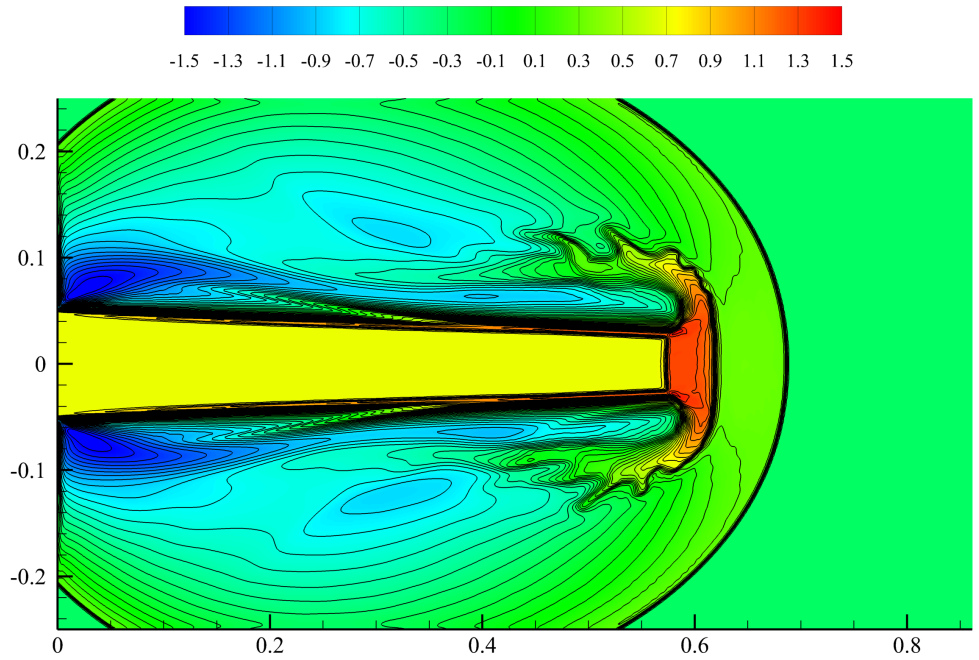
Density along diagonal

M2000 Jet (2D Euler)

Mach 2000 jet

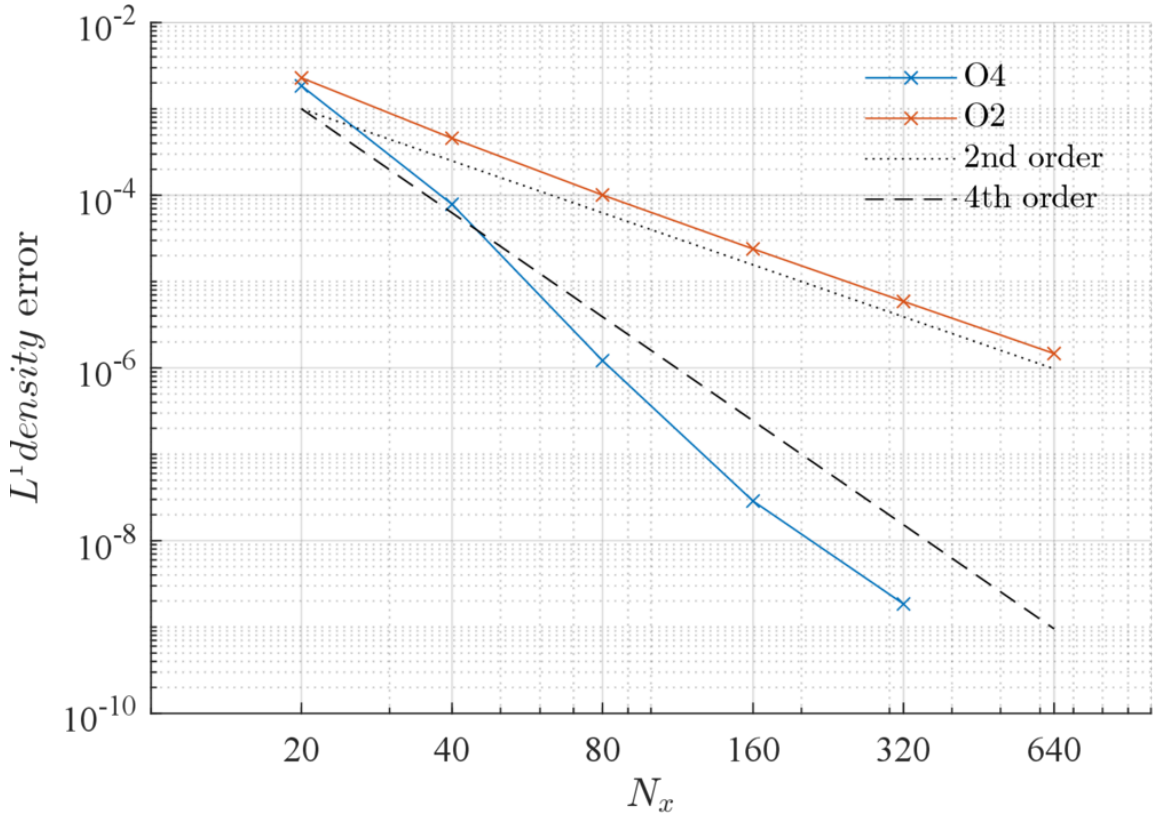


$\log_{10}(p)$, *Re = 100*

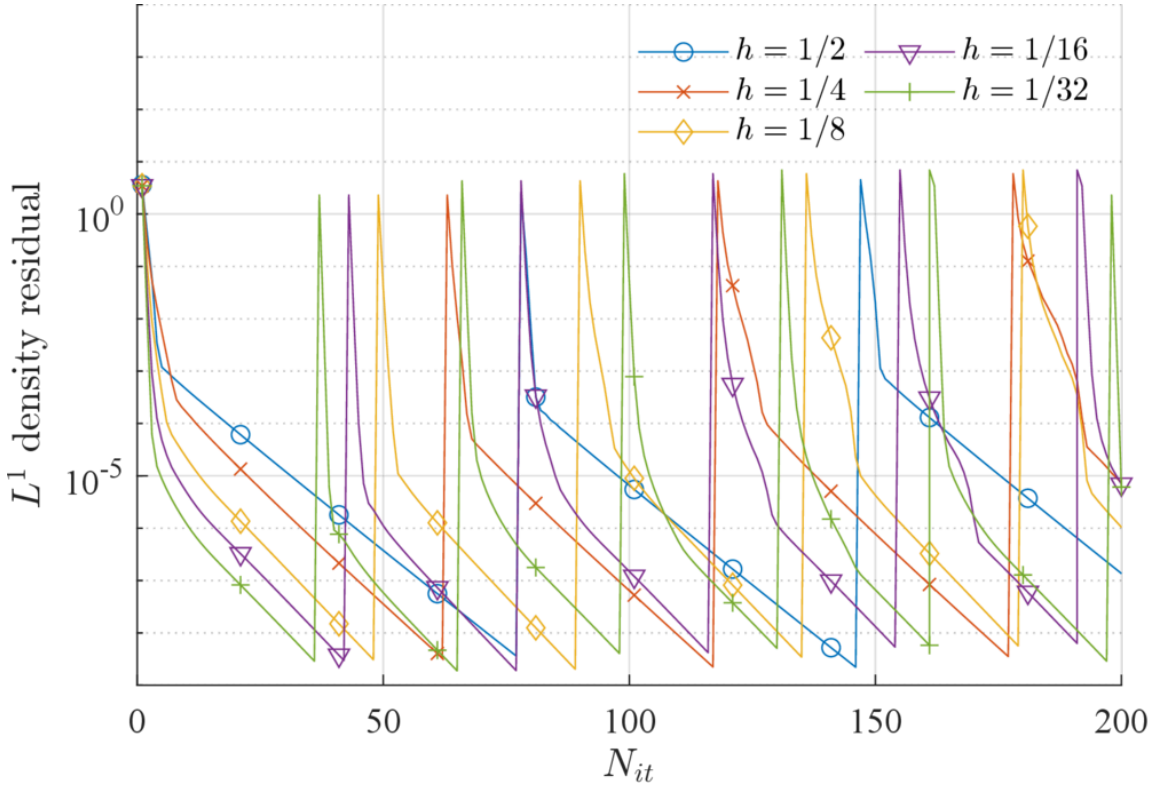


$\log_{10}(\rho)$, *Re = 100*

Isentropic Vortex (2D Euler) — error convergence



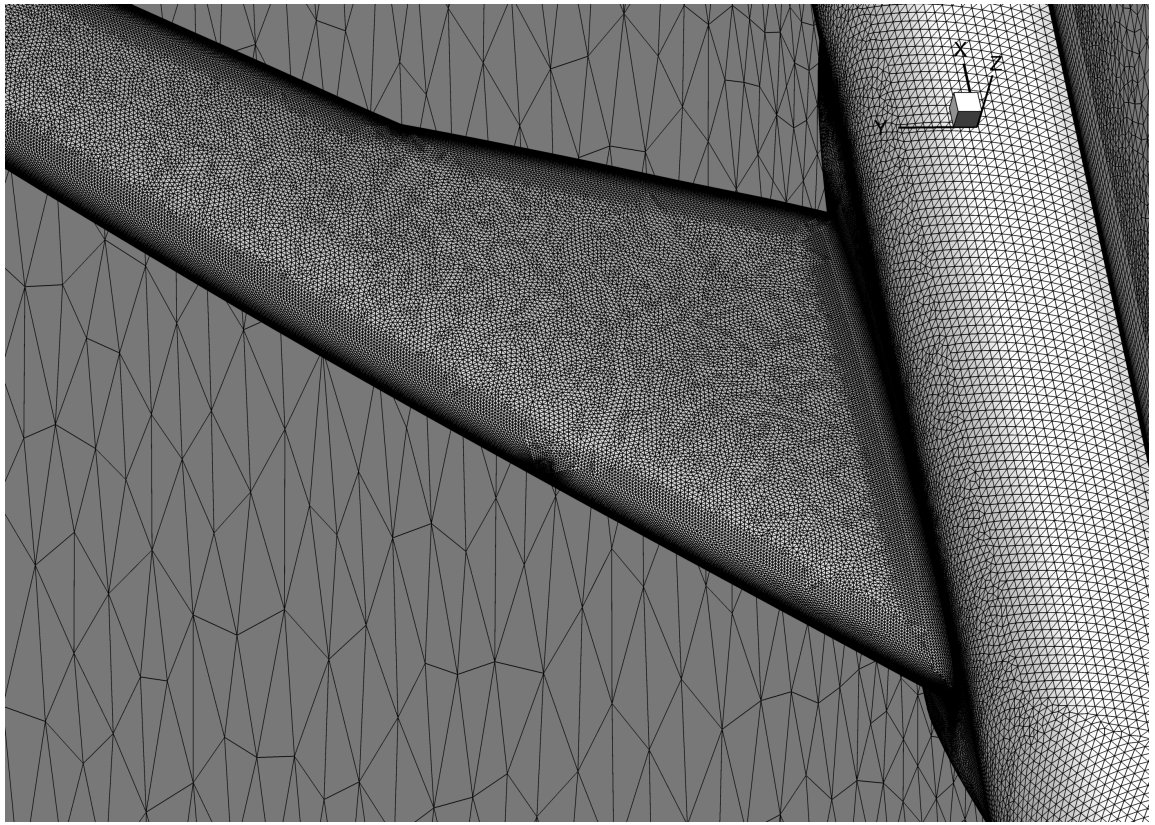
L^1 *density error vs. grid size*



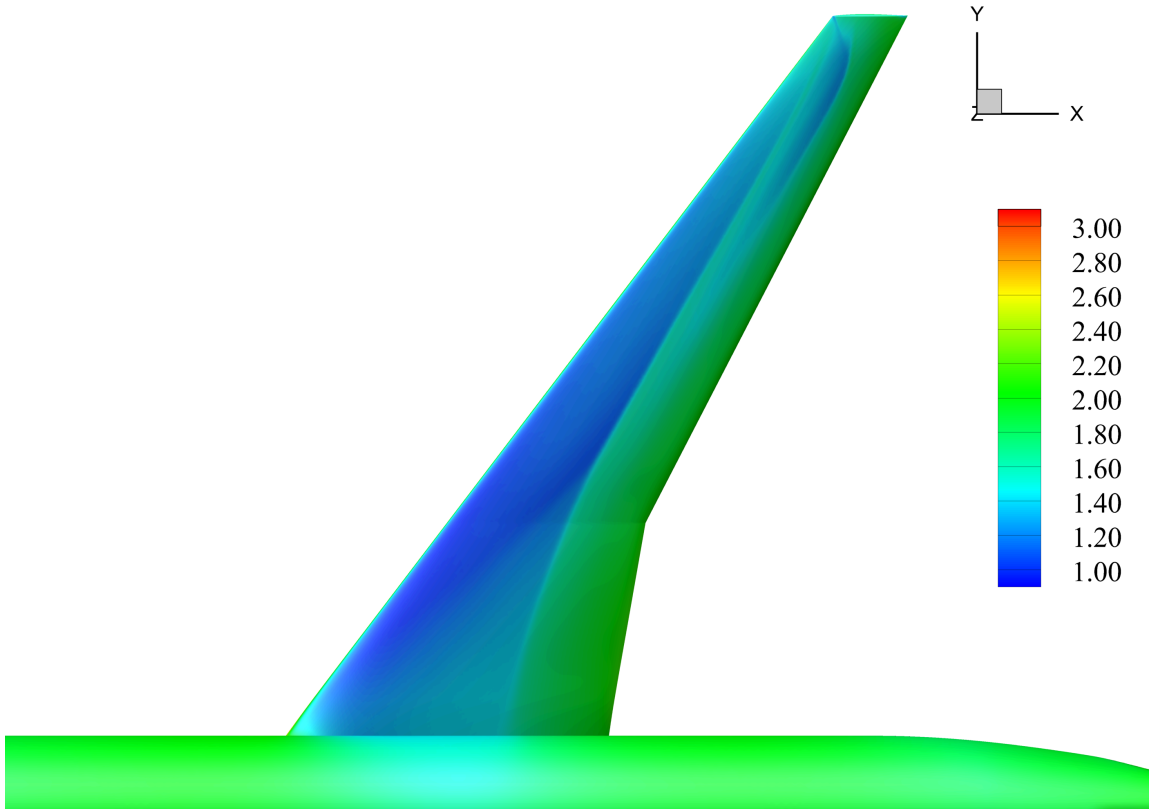
Residual convergence

NASA CRM (3D SA RANS)

Wing-body

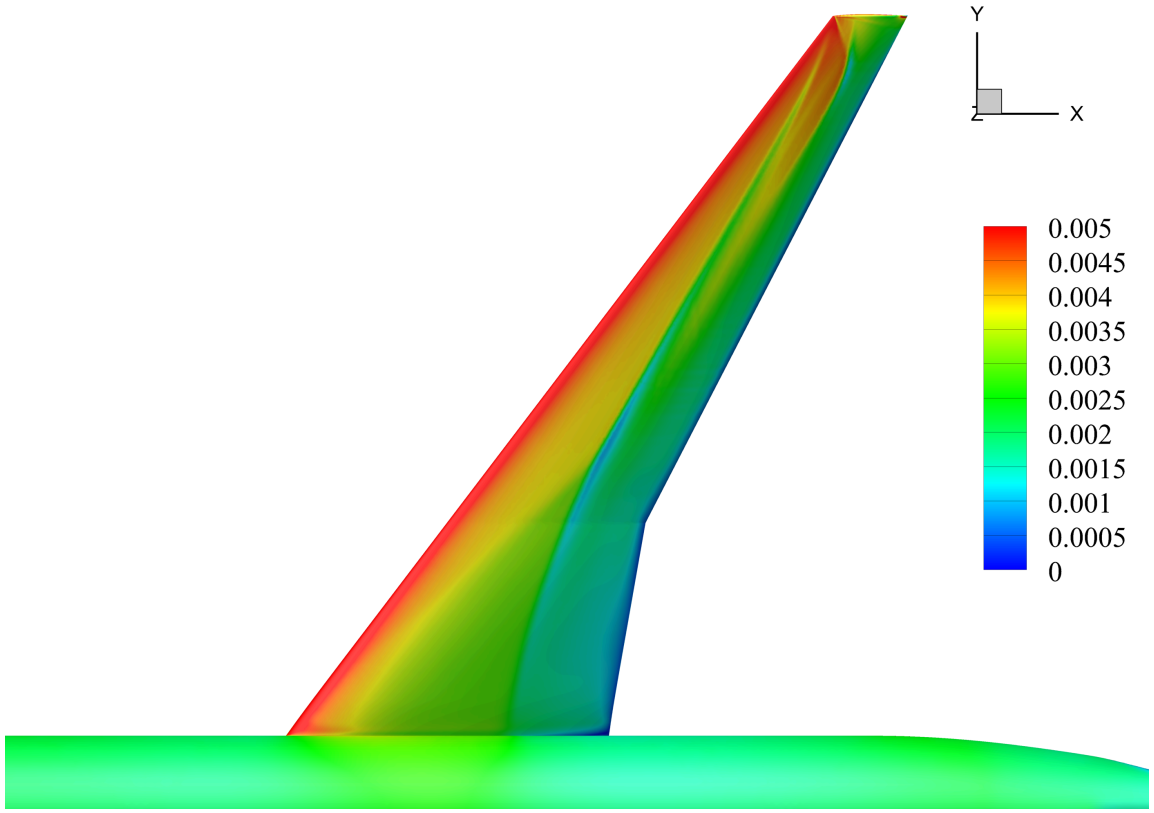


Surface mesh

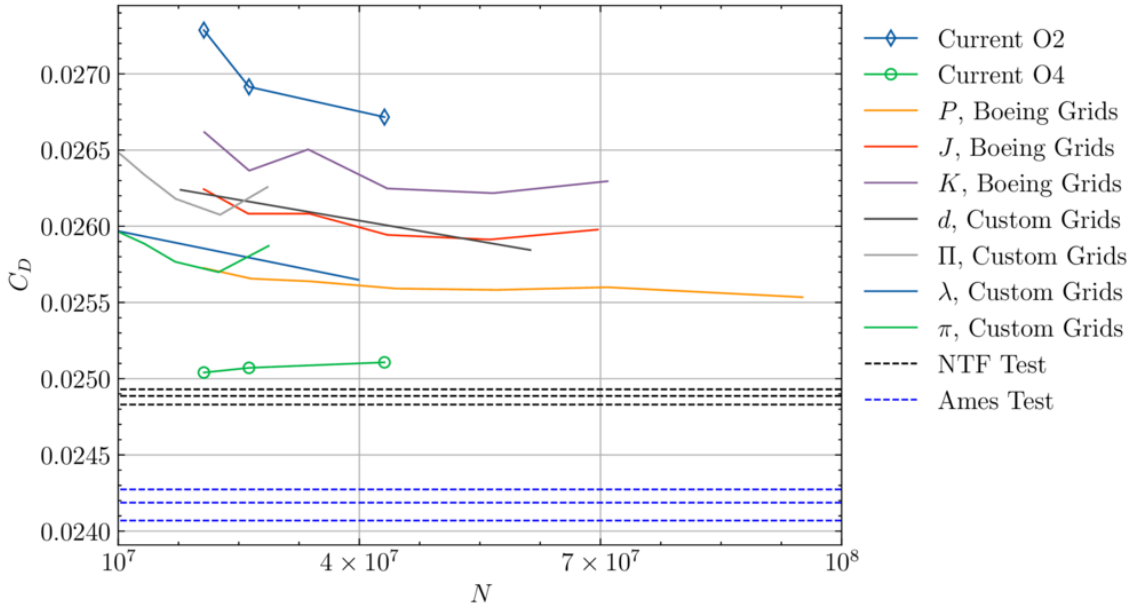


Surface C_p

NASA CRM — C_f + C_L/C_D



Surface C_f



Force-coefficient convergence

CHAPTER 10

Roadmap & Close

What's next · pointers · acknowledgements

Roadmap — mesh & topology

Near-term: configurable ghost

```
struct GhostRequirement {
    int  cellRings       = 1;     // # of cell2cell rings
    bool nodeNeighbor    = true;  // cell2cell by vertex share vs face share
    bool complementNodes = true;  // ghost cells keep all their nodes
    bool complementBnds  = true;  // owned nodes keep all their bnds
};
```

- FEM needs a **smaller** ghost set than compact FV.
- Wide-stencil FV (2+ layers) needs a **larger** set.
- Current default ghost is hard-coded — `nGhostLayers` only adjusts depth, not the *kind* of neighbor.

Medium-term: edge entities

- Add `edge2node`, `cell2edge`, `node2edge` with the same `AdjPairTracked` discipline.
- Extract the face-interpolation algorithm into a **generic** codim-K template.
- Needed for **node-based FV** and **FEM** workflows.

Long-term: DMPlex-style DAG

- Unified point numbering across nodes / edges / faces / cells.
- Parameterized adjacency — `useCone` × `useClosure` Boolean matrix.
- PetscSF-style communication structure.
- PetscSection-style decoupling of topology from discretization.
- Proposal in `docs/architecture/MeshDAGDesign.md` (765 lines).

Roadmap — solvers, parallelism, V&V

Solvers

- **Reactive / multi-species** (`NS_EX`) — maturity pass, published validation cases.
- **Overset grids** — 2D demo exists (hole creation, distance map, cell-cell connectivity); extend to 3D and add full transfer operators.
- **Full (assembled) Jacobian** — currently matrix-free + LU-SGS only. Assembling a block CSR Jacobian opens up AMG + SuperLU_dist + PETSc as preconditioners.

Parallelism

- Extend CUDA coverage from `EulerP` to the full `Euler` evaluator.
- More SoA kernels; GPU-aware MPI over pinned device memory.
- Broader device coverage for VR reconstruction.
- Task-based execution experiments.

V&V workflow

- Standardized **verification & validation** harness running a fixed case set on every release, publishing convergence plots to the doc site.
- More turbomachinery cases: NASA Rotor 67, low-speed fan stage.
- Noise-source diagnostics for aeroacoustics.

Documentation & dev-UX

- **Expand Python examples** to match current C++ coverage.
- **Clang-tidy sanitation** for remaining modules (Solver, Geom, CFV, Euler, EulerP) — same recipe as DNDS.
- **Contributor on-boarding** guide, already outlined in `docs/dev/`.

Where to read next

Architecture

- **`array_infrastructure.md`** — bottom-up tour of `Array` → `ArrayTransformer` → `ArrayPair` → `ArrayDof`.
- **`MeshConnectivity.md`** — the AdjPairTracked state machine, the ghost-spec DSL, the DAG roadmap.
- **`Serialization.md`** — layer-cake I/O, cross-np restart, offset modes.
- **`Paradigm.md`** — the delayed-abstraction philosophy contrasted with OpenFOAM / SU2.

Theory

- **`Variational_Reconstruction.md`** / **`.pdf`** — full derivation of the facial functional, inner-product choices, and local system.
- **`Shape_Functions.md`** — per-element shape functions and quadrature.

Guides

- **`building.md`** — externals, headers, CMake presets.
- **`array_usage.md`** — how to write code with `Array` / `ArrayDof`.
- **`geom_usage.md`** — mesh construction and VR pipeline.
- **`python_geom_guide.md`** — full Python Geom API reference.
- **`serialization_usage.md`** — HDF5 checkpoints, redistribution.
- **`style_guide.md`** — C++ and Python conventions.
- **`examples.md`** — runnable `examples/ex_*.cpp` programs.

Tests

- **`docs/tests/overview.md`** — golden values, determinism, suite totals.
- Per-module test pages under `docs/tests/{dnds,geom,cfv,euler,solver}_unit_tests.md`.

Thank you

Try it in three commands

```
cmake --preset release-test
cmake --build build -t euler -j32
mpirun -np 4 ./build/app/euler.exe cases/euler_config_IV.json
```

Code · github.com/CFDLAB-THU/DNDSR **Docs** · cfdlab-thu.github.io/DNDSR **Release notes** · `RELEASE_NOTES.md` (v0.2.0)

CFD Lab, Tsinghua University