ALE Implementation Headers Documentation

gsIncompressibleFlow Module

July 24, 2025

Contents

1 Overview

This document provides a detailed description of the core header files for the ALE (Arbitrary Lagrangian-Eulerian) implementation in the gsIncompressibleFlow module.

2 Mathematical Formulation

2.1 ALE Framework

The ALE method combines the advantages of Lagrangian and Eulerian descriptions. In the ALE framework, the computational mesh can move arbitrarily, independent of the fluid motion. The key concept is the introduction of a reference configuration $\hat{\Omega}$ that maps to the current configuration Ω_t through:

$$\mathcal{A}_t : \hat{\Omega} \to \Omega_t, \quad \hat{x} \mapsto x = \mathcal{A}_t(\hat{x}) \tag{1}$$

The mesh velocity is defined as:

$$w = \left.\frac{\partial \mathcal{A}_t}{\partial t}\right|_{\hat{x}} \tag{2}$$

2.2 Incompressible Navier-Stokes Equations in ALE Form

The incompressible Navier-Stokes equations in ALE form are:

$$\rho\left(\left.\frac{\partial u}{\partial t}\right|_{\hat{x}} + (u-w)\cdot\nabla u\right) - \nabla\cdot\sigma = f \quad \text{in } \Omega_t \tag{3}$$

$$\nabla\cdot u = 0 \quad \text{in } \Omega_t \tag{4}$$

where:

- u is the fluid velocity
- w is the mesh velocity
- $\sigma = -pI + 2\mu\varepsilon(u)$ is the stress tensor
- $\varepsilon(u) = \frac{1}{2}(\nabla u + \nabla u^T)$ is the strain rate tensor
- ρ is the fluid density
- μ is the dynamic viscosity
- p is the pressure
- f is the body force

2.3 Weak Formulation

Multiplying by test functions $(v, q) \in V \times Q$ and integrating by parts:

$$\int_{\Omega_t} \rho \left.\frac{\partial u}{\partial t}\right|_{\hat{x}} \cdot v\, d\Omega + \int_{\Omega_t} \rho(u-w)\cdot\nabla u\cdot v\, d\Omega \tag{5}$$

$$+ \int_{\Omega_t} 2\mu\varepsilon(u) : \varepsilon(v)\, d\Omega - \int_{\Omega_t} p\nabla\cdot v\, d\Omega = \int_{\Omega_t} f\cdot v\, d\Omega \tag{6}$$

$$\int_{\Omega_t} q\nabla\cdot u\, d\Omega = 0 \tag{7}$$

The key ALE contribution is the convective term:

$$\boxed{\int_{\Omega_t} \rho(u - w) \cdot \nabla u \cdot v \, d\Omega} \tag{8}$$

2.4 Time Discretization

Using implicit Euler (first-order backward difference) for time discretization:

$$\left.\frac{\partial u}{\partial t}\right|_{\hat{x}}^{n+1} \approx \frac{u^{n+1} - u^n}{\Delta t} \tag{9}$$

For the mesh velocity, we use:

$$w^{n+1} = \frac{x^{n+1} - x^n}{\Delta t} = \frac{d^{n+1} - d^n}{\Delta t} \tag{10}$$

where d represents the mesh displacement from the reference configuration.

2.5 Discretized Weak Form

The fully discretized weak form at time step $n + 1$ becomes:

$$\int_{\Omega_{n+1}} \rho \frac{u^{n+1} - u^n}{\Delta t} \cdot v \, d\Omega + \int_{\Omega_{n+1}} \rho(u^{n+1} - w^{n+1}) \cdot \nabla u^{n+1} \cdot v \, d\Omega \tag{11}$$

$$+ \int_{\Omega_{n+1}} 2\mu\varepsilon(u^{n+1}) : \varepsilon(v) \, d\Omega - \int_{\Omega_{n+1}} p^{n+1} \nabla \cdot v \, d\Omega = \int_{\Omega_{n+1}} f^{n+1} \cdot v \, d\Omega \tag{12}$$

$$\int_{\Omega_{n+1}} q \nabla \cdot u^{n+1} \, d\Omega = 0 \tag{13}$$

2.6 Matrix Form

After spatial discretization using finite elements, the system can be written in matrix form:

$$\begin{bmatrix} \frac{1}{\Delta t} M + A(u^{n+1}) + K & B^T \\ B & 0 \end{bmatrix} \begin{bmatrix} u^{n+1} \\ p^{n+1} \end{bmatrix} = \begin{bmatrix} F + \frac{1}{\Delta t} M u^n + A_{ALE}(w^{n+1}) \\ 0 \end{bmatrix} \tag{14}$$

where:

- M is the mass matrix
- K is the stiffness matrix (viscous term)
- $A(u)$ is the convection matrix
- $A_{ALE}(w)$ represents the ALE correction to convection
- B is the divergence matrix
- F includes body forces and boundary conditions

2.7 ALE Convection Term Implementation

The ALE convection term is implemented as:

$$[A_{ALE}]_{ij} = \int_\Omega \rho\varphi_i \cdot ((u - w) \cdot \nabla\varphi_j)\, d\Omega \tag{15}$$

where φ_i and φ_j are the finite element basis functions. The implementation computes:

1. Relative velocity: $u_{rel} = u - w$
2. Gradient in physical space: $\nabla\varphi_j = J^{-T}\nabla_\xi\varphi_j$
3. Integration: $\sum_q w_q\, \varphi_i(x_q) \cdot (u_{rel}(x_q) \cdot \nabla\varphi_j(x_q))\, |J|$

where J is the Jacobian of the geometric mapping, x_q are quadrature points, and w_q are quadrature weights.

2.8 Geometric Conservation Law (GCL)

For maintaining consistency, the discrete GCL should be satisfied:

$$\frac{\partial J}{\partial t} - \nabla \cdot (Jw) = 0 \tag{16}$$

This ensures that constant solutions remain constant on moving meshes. In our implementation, this is implicitly satisfied through consistent mesh velocity computation.

2.9 Mesh Motion Strategies

The implementation supports various mesh motion strategies:

2.9.1 Prescribed Motion

For known motion (e.g., rotating domains):

$$x(t) = R(t) \cdot (x_0 - c) + c + d(t) \tag{17}$$

where $R(t)$ is a rotation matrix, c is the rotation center, and $d(t)$ is translation.

2.9.2 FSI Coupling

For fluid-structure interaction, the mesh displacement at the interface follows:

$$d|_{\Gamma_{FSI}} = d_{struct}|_{\Gamma_{FSI}} \tag{18}$$

The displacement is then extended into the fluid domain using harmonic extension or elasticity equations.

2.9.3 Mesh Optimization

Using barrier functions to maintain mesh quality:

$$\min_x \sum_{K\in\mathcal{T}} \int_K \mu(J_K)\, dK \tag{19}$$

where $\mu(J) = \frac{1}{2}(J + \frac{1}{J}) - 1$ is the barrier function preventing element inversion.

3 Numerical Implementation Details

3.1 Taylor-Hood Elements

The implementation uses Taylor-Hood elements (P2-P1 or P3-P2) for velocity-pressure discretization:

$$u_h \in V_h = \{v \in [C^0(\Omega)]^d : v|_K \in [P_k(K)]^d, \forall K \in \mathcal{T}_h\} \tag{20}$$

$$p_h \in Q_h = \{q \in C^0(\Omega) : q|_K \in P_{k-1}(K), \forall K \in \mathcal{T}_h\} \tag{21}$$

This choice satisfies the inf-sup (LBB) condition for stability.

3.2 Solution Algorithm

The nonlinear system is solved using Newton's method with the following steps:

1. **Mesh Update**:

$$x^{n+1} = x^n + d^{n+1}, \quad w^{n+1} = \frac{d^{n+1} - d^n}{\Delta t} \tag{22}$$

2. **Mesh Optimization** (optional):

$$x_{opt}^{n+1} = \arg\min_x \sum_K \int_K \mu(J_K)\, dK \tag{23}$$

3. **Flow Solution**: Solve the nonlinear system using Newton iterations:

$$J^k \delta u^k = -R^k, \quad u^{k+1} = u^k + \delta u^k \tag{24}$$

where J^k is the Jacobian matrix and R^k is the residual.

3.3 ALE-Specific Assembly

The assembly process for ALE terms follows:

Algorithm 1 ALE Convection Term Assembly

```
1: for each element $K$ do
2:     for each quadrature point $q$ do
3:         Compute $u(x_q)$ from current solution
4:         Compute $w(x_q)$ from mesh velocity field
5:         $u_{rel} = u(x_q) - w(x_q)$
6:         Compute $\nabla\varphi_j(x_q)$ in physical space
7:         for each test function $i$ do
8:             for each trial function $j$ do
9:                 $A_{ij}^{elem} += w_q |J| \varphi_i(x_q) \cdot (u_{rel} \cdot \nabla\varphi_j(x_q))$
10:            end for
11:        end for
12:    end for
13: end for
```

3.4 Boundary Conditions in ALE

For moving boundaries, the boundary conditions must account for mesh motion:

3.4.1 No-slip on moving walls

$$u = u_{wall} = w \quad \text{on } \Gamma_{wall}(t) \tag{25}$$

3.4.2 Inflow/Outflow

$$u \cdot n = (u_{prescribed} - w) \cdot n \quad \text{on } \Gamma_{in/out}(t) \tag{26}$$

3.4.3 Free surface

$$\sigma \cdot n = p_{ext} n - \gamma \kappa n \quad \text{on } \Gamma_{free}(t) \tag{27}$$

where γ is surface tension and κ is curvature.

4 Stability and Convergence Properties

4.1 Stability Considerations

The ALE formulation maintains stability through:

4.1.1 Geometric Conservation Law

The discrete GCL ensures that:

$$\int_{\Omega(t)} 1 \, d\Omega = \text{constant} \tag{28}$$

4.1.2 CFL Condition

For explicit time stepping (if used), the CFL condition becomes:

$$\Delta t \leq C \min_K \left\{ \frac{h_K}{|u - w|_{\max,K}} \right\} \tag{29}$$

For implicit schemes (as in our implementation), this restriction is relaxed but mesh quality still affects convergence.

4.2 Error Estimates

For the ALE formulation with Taylor-Hood elements and implicit Euler time stepping:

$$\|u - u_h\|_{L^2(\Omega)} + \|p - p_h\|_{L^2(\Omega)} \leq C(h^{k+1} + \Delta t) \tag{30}$$

where k is the polynomial degree of velocity elements.

4.3 Mesh Quality Metrics

The implementation monitors mesh quality through:

4.3.1 Element Jacobian

$$0 < J_{\min} \leq J_K \leq J_{\max} < \infty \tag{31}$$

4.3.2 Aspect Ratio

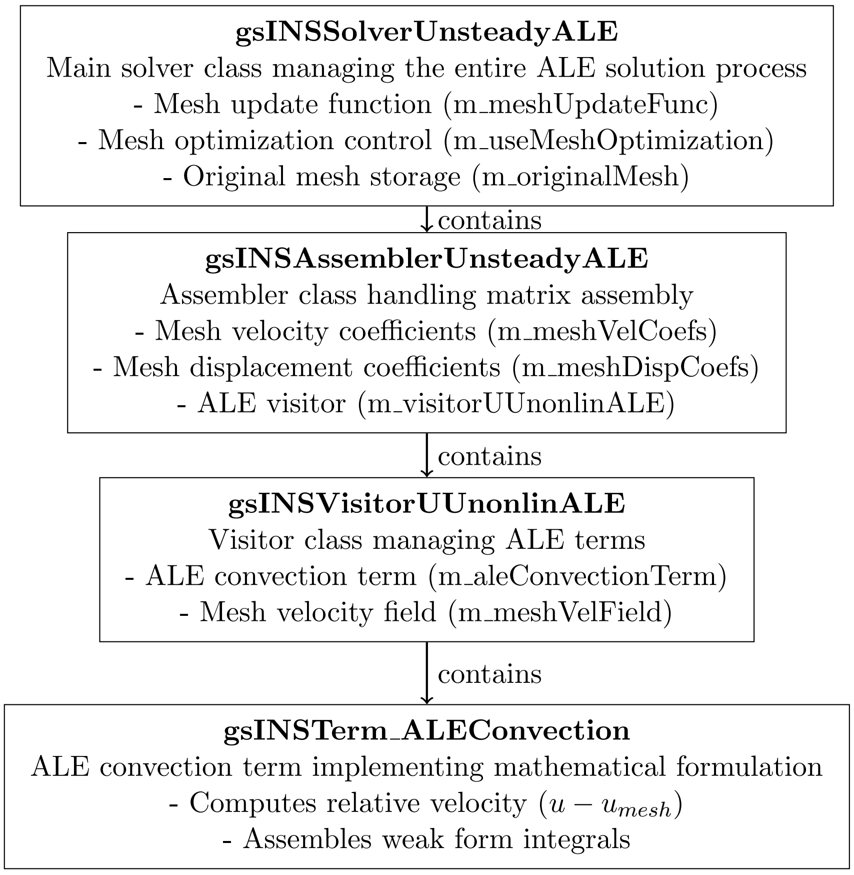
$$AR_K = \frac{h_{\max,K}}{h_{\min,K}} \tag{32}$$

4.3.3 Skewness

$$\text{Skew}_K = \max_{i,j} |\cos(\theta_{ij})| \tag{33}$$

where θ_{ij} are angles between element edges.

5 Class Structure Overview



gsINSSolverUnsteadyALE
Main solver class managing the entire ALE solution process
- Mesh update function (m_meshUpdateFunc)
- Mesh optimization control (m_useMeshOptimization)
- Original mesh storage (m_originalMesh)

contains

gsINSAssemblerUnsteadyALE
Assembler class handling matrix assembly
- Mesh velocity coefficients (m_meshVelCoefs)
- Mesh displacement coefficients (m_meshDispCoefs)
- ALE visitor (m_visitorUUnonlinALE)

contains

gsINSVisitorUUnonlinALE
Visitor class managing ALE terms
- ALE convection term (m_aleConvectionTerm)
- Mesh velocity field (m_meshVelField)

contains

gsINSTerm_ALEConvection
ALE convection term implementing mathematical formulation
- Computes relative velocity ($u - u_{mesh}$)
- Assembles weak form integrals

Figure 1: ALE implementation class hierarchy

6 gsINSTermsALE.h

6.1 Overview

Defines the weak form implementation of ALE convection terms, which is the core mathematical expression of the ALE method.

6.2 Main Class: gsINSTerm_ALEConvection

6.2.1 Class Definition (Lines 21-22)

```cpp
template <class T>
class gsINSTerm_ALEConvection : public gsFlowTermNonlin<T>
```

6.2.2 Core Functionality

- **ALE convection term**: Implements the weak form of $((u - u_{mesh}) \cdot \nabla\varphi_{trial}) * \varphi_{test}$ (Line 18)
- **Relative velocity computation**: $u_{rel} = u - u_{mesh}$ (Line 96)
- **Physical space gradient transformation**: Transforms parametric space gradients to physical space (Line 119)

6.2.3 Key Member Variables (Lines 30-36)

- `m_tarDim`: Target dimension (typically 2 or 3)
- `m_meshVelVals`: Mesh velocity values at quadrature points
- `m_meshVelField`: Mesh velocity field pointer

6.2.4 Core Methods

`computeMeshVelocity()` **(Lines 57-70)**

```
void computeMeshVelocity(const gsMapData<T>& mapData)
```

- Computes mesh velocity at quadrature points
- Defaults to zero if mesh velocity field is not set (Line 68)

`assemble()` **(Lines 78-134)**

```
virtual void assemble(const gsMapData<T>& mapData,
                      const gsVector<T>& quWeights,
                      const std::vector<gsMatrix<T>>& testFunData,
                      const std::vector<gsMatrix<T>>& trialFunData,
                      gsMatrix<T>& localMat)
```

- Assembles local matrix for ALE convection term
- Computes relative velocity `relativeVel = m_solUVals - m_meshVelVals` (Line 96)
- Computes gradients in physical space (Line 119)
- Integrates $\int(\varphi_{test} \cdot ((u - w) \cdot \nabla\varphi_{trial}))d\Omega$ (Lines 131-132)

6.2.5 Design Features

- **Flexibility**: Supports dynamic mesh velocity field setting
- **Efficiency**: Computes mesh velocity only when needed
- **Generality**: Supports both 2D and 3D problems

7 gsINSVisitorsALE.h

7.1 Overview

Provides ALE visitor classes that manage ALE convection terms and integrate them into the assembly process.

7.2 Main Class: gsINSVisitorUUnonlinALE

7.2.1 Class Definition (Lines 22-24)

```
template <class T, int MatOrder = RowMajor>
class gsINSVisitorUUnonlinALE : public gsINSVisitorUU<T, MatOrder>
```

7.2.2 Core Functionality

- **ALE term management**: Creates and manages `gsINSTerm_ALEConvection` instances
- **Interface adaptation**: Connects ALE terms with standard INS assembly framework
- **State management**: Updates mesh velocity fields and current solutions

7.2.3 Constructors

Two construction methods are provided:

Recommended Method (with paramsPtr) (Lines 43-58)

```
gsINSVisitorUUnonlinALE(typename gsFlowSolverParams<T>::Ptr paramsPtr,
                        const std::vector<gsDofMapper>& dofMappers,
                        index_t targetDim = 2,
                        const gsField<T>* meshVelField = nullptr)
```

Compatibility Method (without paramsPtr) (Lines 61-75)

```
gsINSVisitorUUnonlinALE(const std::vector<gsDofMapper>& dofMappers,
                        index_t targetDim = 2,
                        const gsField<T>* meshVelField = nullptr)
```

7.2.4 Key Methods

`setMeshVelocityField()` **(Lines 83-89)**

```
void setMeshVelocityField(const gsField<T>* meshVelField)
```

- Updates mesh velocity field
- Synchronously updates internal ALE convection term (Line 88)

`setCurrentSolution()` **(Lines 92-100)**

```
void setCurrentSolution(const gsField<T>& solution)
```

- Sets current velocity solution
- Passes to all nonlinear terms

8 gsINSAssemblerALE.h

8.1 Overview

Extends the unsteady INS assembler with ALE functionality, handling mesh motion and velocity computation.

8.2 Main Class: gsINSAssemblerUnsteadyALE

8.2.1 Class Definition (Lines 21-23)

```
template <class T, int MatOrder = RowMajor>
class gsINSAssemblerUnsteadyALE : public gsINSAssemblerUnsteady<T,
    MatOrder>
```

8.2.2 Key Member Variables (Lines 39-51)

```
gsMatrix<T> m_meshVelCoefs;      // Mesh velocity coefficients (Line
    40)
gsMatrix<T> m_meshDispCoefs;     // Current mesh displacement (Line 41)
gsMatrix<T> m_meshDispOld;       // Previous time step mesh
    displacement (Line 42)
gsINSVisitorUUnonlinALE<T, MatOrder>* m_visitorUUnonlinALE;  // ALE
    visitor (Line 45)
bool m_isALEActive;              // ALE activation flag (Line 48)
gsField<T>* m_tempMeshVelField;  // Temporary mesh velocity field (Line
     51)
```

8.2.3 Core Methods

`initialize()` **(Lines 70-86)**

```
virtual void initialize() override
```

- Initializes base class (Line 73)
- Allocates mesh coefficient storage space (Lines 76-79)
- Creates ALE visitor (Lines 82-83)

`updateMesh()` **(Lines 108-131)**

```
void updateMesh(const gsMatrix<T>& meshDispNew)
```

- Computes mesh velocity: $v_{mesh} = (disp_{new} - disp_{old})/dt$ (Line 123)
- Updates displacement storage (Line 130)
- Extends vectors to full size (includes zero pressure part) (Lines 121-127)

`assembleNonlinearPart()` **(Lines 154-195)**

```
virtual void assembleNonlinearPart() override
```

- Uses ALE visitor if ALE is active (Line 156)
- Creates temporary mesh velocity field (Lines 159-160)
- Assembles ALE convection terms (Lines 181, 187)

9 gsINSSolverALE.h

9.1 Overview

Top-level ALE solver class providing complete ALE flow solution functionality, including mesh optimization and FSI coupling support.

9.2 Main Class: gsINSSolverUnsteadyALE

9.2.1 Class Definition (Lines 24-26)

```
template <class T = real_t, int MatOrder = RowMajor>
class gsINSSolverUnsteadyALE : public gsINSSolverUnsteady<T, MatOrder>
```

9.2.2 Key Member Variables (Lines 38-61)

```
bool m_isALEActive;                          // ALE activation flag (
    Line 38)
std::function<gsMatrix<T>(T)> m_meshUpdateFunc;  // Mesh update
    function (Line 41)
bool m_useMeshOptimization;                  // Mesh optimization flag
    (Line 44)
bool m_useDynamicBoundaryMapping;            // Dynamic boundary
    mapping flag (Line 47)
gsMultiPatch<T> m_originalMesh;              // Original mesh (Line 53)
gsMatrix<T> m_previousDisp;                  // Previous time step
    displacement (Line 56)
T m_rotationPeriod;                          // Rotation period (Line
    59)
gsVector<T> m_rotationCenter;                // Rotation center (Line
    60)
```

9.2.3 Core Methods

`setALEActive()` (Lines 87-99)

```
void setALEActive(bool active)
```

- Activates/deactivates ALE (Line 91)
- Stores original mesh on first activation (Line 96)
- Synchronously updates assembler state (Line 91)

`nextIteration()` (Lines 183-200)

```
virtual void nextIteration() override
```

Main ALE time stepping function with the following sequence:

1. Apply mesh displacement (Line 189)
2. Mesh optimization (optional) (Line 194)
3. Solve flow problem (Line 199)

`optimizeMesh()` (Lines 247-290)

```
void optimizeMesh()
```

- Uses gsBarrierPatch to optimize mesh quality (Line 275)
- Supports standard and dynamic boundary mapping modes (Line 254)
- Exception handling: continues computation if optimization fails (Lines 286-289)

9.3 Helper Class: gsFSIHelper (Lines 296-387)

Provides utility functions for fluid-structure interaction (framework reserved for future implementation).

9.3.1 Static Utility Methods (Lines 365-370)

```
static gsMatrix<T> computeMeshVelocity(const gsMatrix<T>& dispNew,
                                       const gsMatrix<T>& dispOld,
                                       T dt)
```

- Computes mesh velocity from displacement history
- Uses backward difference scheme (Line 369)

10 Class Hierarchy

10.1 Inheritance Relationships

```
gsFlowTermNonlin<T>
    |-- gsINSTerm_ALEConvection<T>

gsINSVisitorUU<T, MatOrder>
    |-- gsINSVisitorUUnonlinALE<T, MatOrder>

gsINSAssemblerUnsteady<T, MatOrder>
    |-- gsINSAssemblerUnsteadyALE<T, MatOrder>

gsINSSolverUnsteady<T, MatOrder>
    |-- gsINSSolverUnsteadyALE<T, MatOrder>

Independent helper class:
    gsFSIHelper<T>
```

10.2 Detailed Class Structure

The following sections provide detailed UML-style descriptions of each class:

10.2.1 `gsINSTerm_ALEConvection<T>`

- **Private Members:**
  - `m_tarDim: index_t` – Target dimension
  - `m_meshVelVals: gsMatrix<T>` – Mesh velocity values
  - `m_meshVelField: const gsField<T>*` – Mesh velocity field pointer
- **Public Methods:**
  - `setMeshVelocityField(field): void`
  - `computeMeshVelocity(mapData): void`
  - `assemble(...): void override`

10.2.2 `gsINSVisitorUUnonlinALE<T, MatOrder>`

- **Private Members:**
  - `m_tarDim: index_t` – Target dimension
  - `m_meshVelField: const gsField<T>*` – Mesh velocity field
  - `m_aleConvectionTerm: gsINSTerm_ALEConvection<T>*` – ALE term
- **Public Methods:**
  - `setMeshVelocityField(field): void`
  - `setCurrentSolution(solution): void`

10.2.3 `gsINSAssemblerUnsteadyALE<T, MatOrder>`

- **Private Members:**
  - `m_meshVelCoefs: gsMatrix<T>` – Mesh velocity coefficients
  - `m_meshDispCoefs: gsMatrix<T>` – Mesh displacement coefficients
  - `m_meshDispOld: gsMatrix<T>` – Previous mesh displacement
  - `m_visitorUUnonlinALE: gsINSVisitorUUnonlinALE<T>*` – ALE visitor
  - `m_isALEActive: bool` – ALE activation flag
  - `m_tempMeshVelField: gsField<T>*` – Temporary mesh velocity field
- **Public Methods:**
  - `initialize(): void override`
  - `updateMesh(meshDispNew): void`
  - `assembleNonlinearPart(): void override`
  - `getMeshVelocityField(): gsField<T>`
  - `getMeshDisplacementField(): gsField<T>`

10.2.4 `gsINSSolverUnsteadyALE<T, MatOrder>`

- **Private Members:**
  - `m_isALEActive: bool` – ALE activation flag
  - `m_meshUpdateFunc: std::function<gsMatrix<T>(T)>` – Mesh update function
  - `m_useMeshOptimization: bool` – Mesh optimization flag
  - `m_originalMesh: gsMultiPatch<T>` – Original mesh
  - `m_previousDisp: gsMatrix<T>` – Previous displacement
- **Public Methods:**
  - `setALEActive(active): void`
  - `setMeshUpdateFunction(func): void`
  - `nextIteration(): void override`
  - `optimizeMesh(): void`
  - `setMeshOptimization(enable): void`

11 Usage Flow

11.1 Basic Usage Steps

1. **Create solver**

```
gsINSSolverUnsteadyALE<> solver(paramsPtr);
solver.initialize();
```

2. **Activate ALE**

```
solver.setALEActive(true);
```

3. **Set mesh motion**

```
solver.setMeshUpdateFunction([](real_t t) {
    return computeDisplacement(t);
});
```

4. **Optional: Enable mesh optimization**

```
solver.setMeshOptimization(true);
solver.getMeshOptOptions().setInt("Verbose", 1);
```

5. **Time stepping**

```
for (int step = 0; step < nSteps; ++step) {
    solver.nextIteration();
}
```

11.2 Data Flow

```
User-defined mesh motion
    |
    v
gsINSSolverUnsteadyALE::nextIteration()
    |
    v
applyMeshDisplacement()
    |
    v
gsINSAssemblerUnsteadyALE::updateMesh()
    |
    v
gsINSAssemblerUnsteadyALE::assembleNonlinearPart()
    |
    v
gsINSVisitorUUnonlinALE (with mesh velocity)
    |
    v
gsINSTerm_ALEConvection::assemble()
    |
    v
Assemble ALE convection term matrix
```

11.3 Key Design Decisions

1. **Minimal invasiveness**: Extension through inheritance rather than base class modification
2. **Flexibility**: Support for dynamic switching between ALE/standard modes
3. **Efficiency**: Additional data structures created only when needed
4. **Extensibility**: Reserved interfaces for FSI and other coupling methods
5. **Robustness**: Fault tolerance when mesh optimization fails

12 Mesh Optimization with Barrier Functions

12.1 Overview

The ALE implementation integrates barrier function-based mesh optimization to maintain mesh quality during large deformations. This is particularly important for FSI simulations where the mesh can undergo significant distortion.

12.2 Key Components

12.2.1 gsBarrierPatch Integration

The `gsINSSolverUnsteadyALE` class integrates with G+Smo's `gsBarrierPatch` for mesh optimization:

```
// In gsINSSolverUnsteadyALE.h
#include <gsModeling/gsBarrierPatch.h>
#include <gsIncompressibleFlow/src/gsBarrierPatchDynamic.h>
```

12.2.2 Mesh Optimization Method

The `optimizeMesh()` method (Lines 247-290) implements the barrier function optimization:

```
void optimizeMesh()
{
    // Get current mesh from assembler
    gsMultiPatch<T>& patches = const_cast<gsMultiPatch<T>&>(
        m_assemblerPtr->getPatches());

    if (m_useDynamicBoundaryMapping && m_meshUpdateFunc)
    {
        // Apply gsBarrierPatchDynamic for rotating domains
        gsBarrierPatchDynamic<2, T> opt;
        opt.setDynamicBoundaryMapping(true);
        opt.setRotationAngle(getCurrentRotationAngle());
        opt.setRotationCenter(m_rotationCenter);
        patches = opt.compute(patches, m_meshOptOptions);
    }
    else
    {
        // Use standard gsBarrierPatch
        gsBarrierPatch<2, T> opt(patches, false);
        opt.options() = m_meshOptOptions;
        opt.compute();
        patches = opt.result();
    }
}
```

12.3 Barrier Function Formulation

The barrier function approach prevents mesh inversion by minimizing an objective function:

$$\min_{\mathbf{x}} \sum_{e} \int_{\Omega_e} \mu(J_e)\, d\Omega \tag{34}$$

where $\mu(J)$ is a barrier function that approaches infinity as the Jacobian $J \to 0$:

$$\mu(J) = \frac{1}{2}\left(J + \frac{1}{J}\right) - 1 \tag{35}$$

12.4 Configuration Options

The mesh optimization can be configured through `m_meshOptOptions`:

```
// Initialize mesh optimization options
m_meshOptOptions.addInt("Verbose", "Verbosity␣level", 0);
m_meshOptOptions.addInt("ParamMethod", "Parametrization␣method", 1);
m_meshOptOptions.addInt("AAPreconditionType", "AA␣precondition␣type",
    0);
```

12.5 Usage Example

```
// Enable mesh optimization
solver.setMeshOptimization(true);

// Configure optimization parameters
solver.getMeshOptOptions().setInt("Verbose", 1);
solver.getMeshOptOptions().setInt("ParamMethod", 1);

// For rotating domains
solver.setDynamicBoundaryMapping(true);
solver.setRotationParameters(period, center);
```

12.6 Dynamic Boundary Mapping

For rotating machinery applications, the `gsBarrierPatchDynamic` extension provides:

- Automatic detection of approaching boundaries
- Dynamic remapping of boundary nodes
- Preservation of mesh topology during rotation
- Prevention of mesh tangling at sliding interfaces

12.7 Error Handling

The implementation includes robust error handling:

```cpp
try {
    // Mesh optimization code
    patches = opt.compute(patches, m_meshOptOptions);
}
catch (const std::exception& e) {
    gsWarn << "Mesh␣optimization␣failed:␣" << e.what() << "\n";
    gsWarn << "Continuing␣with␣unoptimized␣mesh.\n";
}
```

This ensures that the simulation continues even if mesh optimization fails, improving robustness for production use.

13 Summary

This ALE implementation provides a complete, efficient, and flexible moving mesh flow solution framework. Through a layered design from low-level weak form terms to high-level solvers, each layer has clear responsibilities and interfaces. The integration of barrier function-based mesh optimization ensures mesh quality is maintained throughout the simulation, making it particularly suitable for handling complex flow problems involving moving boundaries such as fluid-structure interaction and rotating machinery.