# centriMembSolver — Theory, Implementation and Usage Guide

**A combined user and developer manual for rotating membrane simulations**

Vitor Geraldes
Department of Chemical Engineering
Instituto Superior Técnico — Universidade de Lisboa
`vitor.geraldes@tecnico.ulisboa.pt`

January 16, 2026

## License and trademark notice

## Abstract

`centriMembSolver` is a finite–volume solver for coupled solvent and solute transport through rotating membranes. Developed within an academic research programme in rotating membrane filtration, it extends `bousinesqPimpleFOAM` to include solute transport and semi-permeable membrane boundary conditions, concentration-dependent viscosity, and a numerically robust semi-implicit treatment of the Centrifugal and Coriolis forces. This guide documents the underlying theory, the implementation of membrane flux models, and the overall solver architecture. A worked example, *rotatingSlit*, is included to illustrate case setup and post-processing. The document is intended both for end-users running membrane simulations and for developers wishing to extend the solver to new flux laws or multi-physics couplings.

# Contents

# 1 Introduction

Rotating membrane systems combine fluid flow, solute transport and selective separation in a centrifugal field. Applications range from ultra–scale–down (USD) process development, where small membrane coupons are rotated at high speed to emulate industrial centrifugation, to cryogenic membrane filtration in biotechnology. In these systems a pressure gradient drives solvent through a semi–permeable barrier while a concentration difference induces osmotic counterflow. Rotation adds centrifugal and Coriolis effects that modify the hydrodynamics and solute distribution.

`centriMembSolver` was developed to provide an open and customisable platform for simulating such coupled phenomena. Built on OpenFOAM® v2506, the solver solves the incompressible Navier–Stokes equations with source terms for rotation, a scalar transport equation for solute concentration and custom patch fields to model membrane fluxes. A key feature of the implementation is a *semi–implicit Coriolis integration* which retains numerical stability at high rotational speeds. In addition, the membrane boundary conditions implement well established flux laws—Spiegler–Kedem, constant flux and target average flux for the solvent, and intrinsic rejection, solute permeability or observed rejection for the solute. Diagnostic quantities such as observed rejection and concentration polarisation are computed automatically. As of the 2026 release, the solver has been extended to simulate a primary solute A and a passive tracer T with separate diffusivities and membrane flux models, enabling tracer-based validation experiments.

The following sections present the governing equations (including viscous and osmotic effects), derive the robust Coriolis algorithm, detail the boundary conditions, describe the solver structure and illustrate usage through a case study. Developer notes at the end provide guidance on extending the code and best practice for stable simulations.

# 2 Theory and Governing Equations

## 2.1 Hydrodynamics

In a non-inertial reference frame rotating with angular velocity $\mathbf{\Omega}$, the incompressible Navier–Stokes equations read

$$\nabla \cdot \mathbf{U} = 0, \qquad \rho \frac{\partial \mathbf{U}}{\partial t} + \rho \nabla \cdot (\mathbf{U}\mathbf{U}) = -\nabla p + \nabla \cdot (\mu_{\text{eff}} \nabla \mathbf{U}) + \rho\, \mathbf{f}_{\text{rot}}, \tag{1}$$

where $\mathbf{U}$ is the velocity, $p$ the kinematic pressure ($p = p_{\text{phys}}/\rho$), and the effective viscosity $\mu_{\text{eff}} = \rho(\nu + \nu_t)$ includes molecular and—if selected—turbulent viscosity.

The rotational body force $\mathbf{f}_{\text{rot}}$ contains centrifugal and Coriolis terms

$$\mathbf{f}_{\text{rot}} = \mathbf{\Omega} \times (\mathbf{\Omega} \times \mathbf{r}) - 2\,\mathbf{\Omega} \times \mathbf{U}, \tag{2}$$

with $\mathbf{r}$ the position vector in the rotating frame. Pressure–velocity coupling is handled by the PIMPLE algorithm, as implemented in `UEqn.H` and `pEqn.H`.

### 2.1.1 Modified pressure, hydrostatic potential and Boussinesq buoyancy

Rotating membrane devices exhibit very large hydrostatic pressure gradients, dominated by centrifugal effects. The physical pressure $p$ may increase by several bars over a few centimetres of radius, while the dynamic variations induced by fluid motion are much smaller. If $p$ is solved directly, the linear systems become ill–conditioned and the convergence of the PIMPLE algorithm deteriorates.

To avoid this, `centriMembSolver` follows the standard OpenFOAM strategy and works with three auxiliary fields:

- `p_rgh`: modified (dynamic) pressure
- `gh`: hydrostatic potential per unit mass
- `rhok`: dimensionless density ratio

The solver only solves for the dynamic component `p_rgh`; the hydrostatic contribution is treated analytically via `gh` and `rhok`.

**Boussinesq density field** Under the Boussinesq approximation, the density depends weakly on solute concentration but remains close to a reference value $\rho_0$:

$$\rho(\boldsymbol{x}) = \rho_0\, \rho_k(\boldsymbol{x}), \qquad |\rho_k - 1| \ll 1, \tag{3}$$

where the code field `rhok` corresponds to the scalar $\rho_k$. The flow is treated as incompressible,

$$\nabla \cdot \boldsymbol{U} = 0, \qquad \frac{\partial \rho}{\partial t} = 0, \tag{4}$$

but density variations ($\rho_k \neq 1$) still generate buoyancy forces.

**Effective acceleration in the rotating frame** Let $\boldsymbol{x}$ denote the position vector in the rotating frame and $\boldsymbol{x}_0$ the axis of rotation (from `SRFProperties`). A fluid particle at rest in this frame is subject to gravity and centrifugal acceleration:

$$\boldsymbol{a}_{\text{eff}}(\boldsymbol{x}) = \boldsymbol{g} - \boldsymbol{\Omega} \times \big(\boldsymbol{\Omega} \times (\boldsymbol{x} - \boldsymbol{x}_0)\big), \tag{5}$$

where $\boldsymbol{g}$ is the gravitational acceleration and $\boldsymbol{\Omega}$ the angular velocity vector.

The Coriolis term $-2\,\boldsymbol{\Omega} \times \boldsymbol{U}$ is non–conservative and is therefore *not* included in $\boldsymbol{a}_{\text{eff}}$.

**Hydrostatic potential `gh`** The key point is that the effective acceleration $\boldsymbol{a}_{\text{eff}}$ in (5) is conservative: it can be written as the gradient of a scalar potential per unit mass. OpenFOAM stores this potential in the field `gh`, defined so that

$$\nabla(\text{gh}) = \boldsymbol{a}_{\text{eff}}. \tag{6}$$

A convenient explicit expression is

$$\text{gh}(\boldsymbol{x}) = \boldsymbol{g} \cdot (\boldsymbol{x} - \boldsymbol{x}_{\text{ref}}) + \frac{1}{2}\,|\boldsymbol{\Omega} \times (\boldsymbol{x} - \boldsymbol{x}_0)|^2 - \text{gh}_{\text{ref}}, \tag{7}$$

where $\boldsymbol{x}_{\text{ref}}$ is a reference point and $\text{gh}_{\text{ref}}$ is a constant offset. This offset does not affect the physics, since only gradients of gh enter the equations.

**Physical meaning.** gh is the hydrostatic potential per unit mass, combining gravitational and centrifugal contributions. Surfaces of constant gh correspond to equilibrium surfaces of a fluid at rest under gravity and rotation.

**Modified pressure $p_{\mathrm{rgh}}$** In the solver, the physical pressure is reconstructed from the three fields `p_rgh`, `gh` and `rhok` as

$$p(\boldsymbol{x}) = \rho_0\big(p_{\mathrm{rgh}}(\boldsymbol{x}) + \rho_k(\boldsymbol{x})\,\mathrm{gh}(\boldsymbol{x})\big) + p_{\mathrm{ref}}, \tag{8}$$

where $p_{\mathrm{rgh}}$ corresponds to the OpenFOAM field `p_rgh` and $p_{\mathrm{ref}}$ is a constant reference pressure. Neglecting the constant $p_{\mathrm{ref}}/\rho_0$, this can be rearranged as

$$p_{\mathrm{rgh}} = \frac{p}{\rho_0} - \rho_k\,\mathrm{gh}. \tag{9}$$

Thus $p_{\mathrm{rgh}}$ represents the *dynamic* part of the pressure: the large hydrostatic background is removed and only the smaller fluctuations are handled by the linear solver.

**Momentum equation in $p_{\mathrm{rgh}}$ form** The incompressible rotating–frame Navier–Stokes equation under Boussinesq can be written as

$$\rho_0\rho_k\left(\frac{\partial \boldsymbol{U}}{\partial t} + \boldsymbol{U}\cdot\nabla\boldsymbol{U}\right) = -\nabla p + \nabla\cdot\big(\mu_{\mathrm{eff}}\nabla\boldsymbol{U}\big) + \rho_0\rho_k\,\boldsymbol{a}_{\mathrm{eff}} - 2\rho_0\rho_k\,\boldsymbol{\Omega}\times\boldsymbol{U}. \tag{10}$$

**Step-by-step transformation to the $p_{\mathrm{rgh}}$-based form** Using the definition (9),

$$p = \rho_0\left(p_{\mathrm{rgh}} + \rho_k\,\mathrm{gh}\right),$$

the pressure gradient becomes

$$\begin{aligned} -\nabla p &= -\rho_0\nabla(p_{\mathrm{rgh}} + \rho_k\,\mathrm{gh}) \\ &= -\rho_0\nabla p_{\mathrm{rgh}} - \rho_0\nabla(\rho_k\,\mathrm{gh}). \end{aligned} \tag{11}$$

Applying the product rule,

$$\nabla(\rho_k\,\mathrm{gh}) = \mathrm{gh}\,\nabla\rho_k + \rho_k\,\nabla(\mathrm{gh}) = \mathrm{gh}\,\nabla\rho_k + \rho_k\,\boldsymbol{a}_{\mathrm{eff}}, \tag{12}$$

where we used $\nabla(\mathrm{gh}) = \boldsymbol{a}_{\mathrm{eff}}$. Thus

$$-\nabla p = -\rho_0\nabla p_{\mathrm{rgh}} - \rho_0\,\mathrm{gh}\,\nabla\rho_k - \rho_0\rho_k\,\boldsymbol{a}_{\mathrm{eff}}. \tag{13}$$

Substituting (13) into (10) and cancelling the $\rho_0\rho_k\,\boldsymbol{a}_{\mathrm{eff}}$ terms on both sides yields

$$\rho_0\left(\frac{\partial \boldsymbol{U}}{\partial t} + \boldsymbol{U}\cdot\nabla\boldsymbol{U}\right) = -\rho_0\nabla p_{\mathrm{rgh}} - \rho_0\,\mathrm{gh}\,\nabla\rho_k + \nabla\cdot\big(\mu_{\mathrm{eff}}\nabla\boldsymbol{U}\big) - 2\rho_0\rho_k\,\boldsymbol{\Omega}\times\boldsymbol{U}. \tag{14}$$

Since $\rho_0$ is constant, we can divide the entire equation by $\rho_0$ to obtain the canonical form used in the solver:

$$\rho_k\left(\frac{\partial \boldsymbol{U}}{\partial t} + \boldsymbol{U}\cdot\nabla\boldsymbol{U}\right) = -\nabla p_{\mathrm{rgh}} - \mathrm{gh}\,\nabla\rho_k + \frac{1}{\rho_0}\nabla\cdot\big(\mu_{\mathrm{eff}}\nabla\boldsymbol{U}\big) - 2\rho_k\,\boldsymbol{\Omega}\times\boldsymbol{U}. \tag{15}$$

The hydrostatic load has disappeared from the unknown pressure ($p_{\mathrm{rgh}}$), and only the term proportional to $\nabla\rho_k$ remains as a Boussinesq buoyancy force.

**Discrete buoyancy term in `centriMembSolver`** The continuous term $-\,\mathrm{gh}\,\nabla\rho_k$ is discretised in OpenFOAM using face–based gradients:

$$-\,\mathrm{gh}\,\nabla\rho_k \quad\Longrightarrow\quad -\,\mathrm{gh}_f\,\mathrm{snGrad}(\rho_k), \tag{16}$$

where $\mathrm{gh}_f$ is the potential field interpolated to the faces, and $\mathrm{snGrad}(\rho_k)$ is the surface–normal gradient of $\rho_k$. In the actual code this appears as

```
- ghf * fvc::snGrad(rhok)
```

inside the reconstruction of the right–hand side of the momentum equation.

#### Summary and physical interpretation

- gh is the hydrostatic potential per unit mass: it combines gravitational and centrifugal contributions and is computed from geometry, $\boldsymbol{g}$ and $\boldsymbol{\Omega}$.
- $\rho_k$ (field `rhok`) encodes density variations relative to $\rho_0$ and is responsible for Boussinesq buoyancy.
- $p_{\mathrm{rgh}}$ (field `p_rgh`) is the pressure variable actually solved by PIMPLE; its gradients are much smaller than those of the physical pressure $p$.
- The buoyancy force appears as $-\,\mathrm{gh}\,\nabla\rho_k$, discretised in the code as `- ghf*fvc::snGrad(rhok)`.
- The physical pressure used for membrane transport and post–processing is reconstructed via (8).

### 2.1.2 Turbulence modelling

Although centrifugal membrane filtration can operate in laminar, transitional or weakly turbulent regimes, `centriMembSolver` allows the user to select any turbulence model available in the `RASModel` hierarchy of OpenFOAM. The model is instantiated via the usual run-time selection mechanism in `createFields.H`:

$$\nu_{\mathrm{eff}} = \nu + \nu_t(\mathbf{U}, k, \varepsilon, \dots). \tag{17}$$

Typical configurations rely on:

- direct laminar modelling (`simulationType laminar;`)
- one–equation eddy–viscosity models such as Spalart–Allmaras
- two–equation models such as standard or low-Re $k\!-\!\varepsilon$ and $k\!-\!\omega$

with parameters defined in `constant/turbulenceProperties`. Turbulent mass transport uses an eddy diffusivity $D_t = \nu_t/\mathrm{Sc}_t$, where $\mathrm{Sc}_t$ is the turbulent Schmidt number.

### 2.1.3 Concentration-dependent viscosity

Many cryoprotectant solutions exhibit a strong dependence of viscosity on the solute concentration $C_A$. `centriMembSolver` incorporates a polynomial parameterisation:

$$\nu(C_A) = \nu_0 + \nu_1 C_A + \nu_2 C_A^2 + \cdots, \tag{18}$$

with coefficients $\nu_0, \nu_1, \nu_2$ set in `transportProperties`. A clipping operator prevents unphysical values at high concentrations.

### 2.2 Solute and tracer transport

In the current release, the solver solves two advective–diffusive transport equations: one for a primary solute concentration $C_A$ and one for a passive tracer concentration $C_T$. Both species are advected by the same velocity field $\mathbf{U}$ but may have different diffusion coefficients. The governing equations read

$$\frac{\partial C_A}{\partial t} + \nabla\cdot(C_A\mathbf{U}) = \nabla\cdot(D_{A,\text{eff}}\,\nabla C_A), \tag{19}$$

$$\frac{\partial C_T}{\partial t} + \nabla\cdot(C_T\mathbf{U}) = \nabla\cdot(D_{T,\text{eff}}\,\nabla C_T), \tag{20}$$

where $D_{A,\text{eff}} = D_{AB}(C_A) + D_t$ is the effective diffusivity of the solute including the concentration–dependent molecular diffusivity $D_{AB}$ and a turbulent contribution $D_t$ (if enabled), and $D_{T,\text{eff}} = D_{T,0} + D_t$ is the effective diffusivity of the tracer with $D_{T,0}$ constant and independent of concentration. Membrane boundary conditions for the solute and tracer are described in §4.2 and §4.3, respectively.

### 2.3 Osmotic pressure and flux definitions

Transmembrane transport is driven by hydrostatic $\Delta p = p_f - p_p$ and osmotic $\Delta\pi = \pi(C_{A,f}) - \pi(C_{A,p})$ differences. Osmotic pressure uses a virial expansion:

$$\pi(C_A) = \sum_{k=1}^{N} a_k C_A^k, \tag{21}$$

reducing to the van 't Hoff law for $a_1 = RT$.

The solvent flux $J_v$ [m/s] is formulated using the Spiegler–Kedem relation:

$$J_v = A\big(\Delta p - \sigma\Delta\pi\big), \tag{22}$$

where $A$ is the hydraulic permeability and $\sigma$ the reflection coefficient. Solute flux $J_s$ [kg/(m$^2$s)] is

$$J_s = (1-\sigma)J_v C_w + B(C_w - C_p), \tag{23}$$

with intrinsic solute permeability $B$ and wall/permeate concentrations $C_w$ and $C_p$.

## 3 Coriolis Term and Its Numerical Treatment in the Momentum Equation

Rotating flows are formulated in a reference frame with angular velocity $\boldsymbol{\Omega}$. The Coriolis acceleration acting on a fluid parcel is

$$\mathbf{f}_C = -2\,\boldsymbol{\Omega}\times\mathbf{U}. \tag{24}$$

In `centriMembSolver`, the Coriolis operator is evaluated at the temporal midpoint in each time step,

$$\mathbf{U}^{n+1/2} \approx \frac{1}{2}\left(\mathbf{U}^n + \mathbf{U}^{n+1}\right), \tag{25}$$

using the updated velocity available in the current PIMPLE iteration. This ensures a centered-in-time discretization of the Coriolis acceleration,

$$\mathbf{f}_C^{n+1/2} = -2\,\boldsymbol{\Omega}\times\mathbf{U}^{n+1/2}, \tag{26}$$

preserving the skew–symmetry of the operator and therefore the discrete kinetic-energy balance of the rotational term.

### 3.1 Momentum equation

The momentum equation, solved within the PIMPLE loop, is

$$\mathcal{A}(\mathbf{U}) - \left(-2\,\boldsymbol{\Omega} \times \mathbf{U}^{n+1/2}\right) = \mathrm{RHS}, \tag{27}$$

implemented as:

```
Umid = 0.5*(U.oldTime() + U);
coriolisSrc = -2.0*(omega ^ Umid);

fvVectorMatrix UEqn
(
fvm::ddt(U)
+ fvm::div(phi,U)
+ turbulence->divDevReff(U)
- coriolisSrc
==
fvOptions(U)
);
```

### 3.2 Coupling strategy

Because the mid-point velocity uses the updated value of **U** from the current corrector, the Coriolis term evolves consistently with the pressure–velocity coupling. For DNS configurations, the `momentumPredictor` is disabled to avoid spurious Courant number excursions induced by the predictor in strongly rotating flows. A single outer corrector per time step is sufficient due to the small time steps imposed by the CFL restriction.

### 3.3 Summary

The Coriolis acceleration in `centriMembSolver` is discretized at the temporal midpoint and updated at every PIMPLE iteration. This yields a robust, energy-consistent and publication-grade formulation suitable for the extreme rotational conditions characteristic of centrifugal membrane systems.

## 4 Membrane Boundary Conditions

Membrane patches couple the feed region to the permeate side by imposing fluxes of solvent and solute according to physical models. `centriMembSolver` implements two specialised patch field classes: `membraneSolventFluxFvPatchVectorField` for the velocity and `membraneSoluteFluxFvPatchScalarFi` for the concentration. The feed side of the membrane is the patch corresponding to the field values; permeate side conditions are either fixed (for constant pressure/concentration) or inferred from the flux law.

### 4.1 Solvent flux boundary (`U`)

The solvent flux condition determines the normal component of **U** at the membrane. Three models are provided:

**Spiegler–Kedem model.** Implements Eq. (22). Dictionary entries include the hydraulic permeability `Ah`, the reflection coefficient `sigma`, virial coefficients `virialCoeffs` for the osmotic pressure and an optional permeate pressure `pPermConst` and concentration `CAPermConst`. The example below sets `Ah = 0.5e-11`, `sigma = 1.0`, `virialCoeffs (0 0)` for a van 't Hoff law and assumes permeate pressure zero:

```
membraneA
{
    type    membraneSolventFlux;
    model   SpieglerKedem;
    Ah      0.5e-11;        // [m/(Pa s)]
    sigma   1.0;            // [-]
    virialCoeffs (0);       // a1 = RT, a2 = 0
    pPermConst 0;           // [Pa]
    CAPermConst 0;          // [kg/m3]
    value  uniform (0 0 0);
}
```

**Constant flux model.** Prescribes a uniform solvent flux $J_v$ independent of driving forces. Use the key `Jv` to set the flux magnitude (in m/s). This model is useful for benchmarking or when the membrane permeability is unknown.

**Target average flux model.** Tunes an effective permeability $A_{\text{eff}}$ such that the area– averaged flux matches a target `JvAverage`. Driving forces are still proportional to $\Delta p - \sigma\Delta\pi$ as in the Spiegler–Kedem model, but $A_{\text{eff}}$ is updated each time step within user–specified bounds `AhMin` and `AhMax`. This is particularly useful for calibrating models against measured permeabilities.

**Diagnostics.** For each membrane patch the solver writes the instantaneous and time–windowed area–average of $J_v$. If window averaging is enabled (`windowAverage true`), the time scale `Twindow` sets the averaging duration. The internal field `Jv` is available for further post–processing.

## 4.2 Solute flux boundary (`CA`)

The solute boundary condition couples the solute transport to the solvent flux. Three models are available:

**Intrinsic rejection model.** Assumes that a fraction $R_{\text{int}}$ of the solute is rejected at the membrane such that the permeate concentration is $C_p = (1 - R_{\text{int}})\, C_w$. The wall concentration $C_w$ follows a film balance

$$k\,(C_w - C_i) = R_{\text{int}}\, J_v\, C_w, \tag{28}$$

where $k$ is the mass transfer coefficient and $C_i$ the bulk concentration adjacent to the membrane. Solute flux is simply $J_s = J_v C_p$. Dictionary entries are `model intrinsicRejection`, `Rint` and the reference concentration `CAb` used for diagnostics.

**Solute permeability model.** Implements Eq. (23) with a solute permeability $B$. If `sigma` is specified it overrides the reflection coefficient from the solvent BC; otherwise $\sigma$ is inherited. This model reduces to a solution–diffusion model when $\sigma = 0$.

**Observed rejection model.** Targets a desired observed rejection $R_{\mathrm{obs}}$ defined by

$$R_{\mathrm{obs}} = 1 - \frac{\langle J_v C_p \rangle}{\langle J_v \rangle \, C_{A,b}}, \tag{29}$$

where $\langle \cdot \rangle$ denotes an area average and $C_{A,b}$ is the bulk reference concentration. The solver iteratively determines an intrinsic rejection $R_{\mathrm{int}}$ such that the observed rejection matches the target within a specified tolerance. Controls include `Robs` (target), `RintMin`, `RintMax`, `tol` and `maxIter`. This is particularly useful when matching experimental data for rejection.

**Diagnostics.** Per–face fluxes $J_s$, $J_v$ and permeate concentration $C_p$ are exposed to the solver via the patch fields `Js`, `Jv` and `CAp`, and are written to the file `postProcessing/membraneSoluteFlux/<patch>/R`
The solver also outputs the observed rejection and concentration polarisation factor $\Gamma = (C_w - C_{A,b})/C_{A,b}$ for post– processing.

## 4.3 Tracer flux boundary (`CT`)

The tracer boundary condition is analogous to the solute flux condition but uses distinct parameters. Three models are provided:

**Intrinsic rejection model.** Assumes that a fraction $R_{T,\mathrm{int}}$ of the tracer is rejected at the membrane such that the permeate concentration is $C_{T,p} = (1 - R_{T,\mathrm{int}})\, C_{T,w}$. A film balance for the tracer yields

$$k_T\,(C_{T,w} - C_{T,i}) = R_{T,\mathrm{int}}\, J_v\, C_{T,w},$$

where $k_T$ is the tracer mass–transfer coefficient and $C_{T,i}$ the bulk tracer concentration adjacent to the membrane. The tracer flux is $J_{s,T} = J_v\, C_{T,p}$. Dictionary entries are `model intrinsicRejection`, `RTint` and the reference tracer concentration `CTb` used for diagnostics.

**Tracer permeability model.** This option uses a Kedem–Katchalsky closure analogous to Eq. (23), with a tracer permeability $B_T$ and an optional tracer reflection coefficient $\sigma_T$. When $\sigma_T$ is omitted, the reflection coefficient is inherited from the solvent boundary condition on `U`. The model enforces

$$J_{s,T} = (1 - \sigma_T)\, J_v C_{T,w} + B_T (C_{T,w} - C_{T,p}),$$

and computes an implied permeate concentration $C_{T,p}$ such that $J_{s,T} = J_v\, C_{T,p}$. Dictionary keywords are `model solutePermeability`, `BT` and optionally `sigmaT`.

**Observed rejection model.** Analogous to the solute case, this option specifies a target observed tracer rejection $R_{T,\mathrm{obs}}$ defined by

$$R_{T,\mathrm{obs}} = 1 - \frac{\langle J_v C_{T,p} \rangle}{\langle J_v \rangle \, C_{T,b}},$$

where $C_{T,b}$ is the reference bulk tracer concentration. The solver iteratively adjusts $R_{T,\mathrm{int}}$ until the observed rejection matches the target within tolerance. Parameters are `RTobs`, `RTintMin`, `RTintMax`, `tol` and `maxIter`.

**Diagnostics.** Per–face tracer flux $J_{s,T}$, solvent flux $J_v$ and permeate tracer concentration $C_{T,p}$ are exposed via the patch fields `JsT`, `Jv` and `CTp`. The solver writes the observed tracer rejection and concentration polarisation factor $\Gamma_T = (C_{T,w} - C_{T,b})/C_{T,b}$ to the summary files for post–processing.

### 4.4 Membrane patch dictionaries

Tables 1 and 2 summarise the principal dictionary entries for solvent and solute boundary conditions. Most values default to physically reasonable numbers; only those shown as required must be provided.

Table 1: Summary of membrane solvent flux boundary entries.

| Key | Units | Description |
|---|---|---|
| `model` | – | `SpieglerKedem`, `constantFlux` or `targetAverageFlux` |
| `Ah` | m/(Pa s) | Hydraulic permeability for Spiegler–Kedem |
| `sigma` | – | Reflection coefficient $\sigma$; defaults to 1 |
| `virialCoeffs` | – | List $(a_1, a_2, \dots)$ for osmotic pressure |
| `Jv` | m/s | Constant flux magnitude (constantFlux model) |
| `JvAverage` | m/s | Target area–average flux (targetAverageFlux) |
| `AhMin`, `AhMax` | m/(Pa s) | Limits on tuned permeability |
| `pPermConst`, `CAPermConst` | – | Fixed permeate side pressure/concentration |
| `windowAverage`, `Twindow` | – | Enable and set window for time–averaged output |

Table 2: Summary of membrane solute flux boundary entries.

| Key | Units | Description |
|---|---|---|
| `model` | – | `intrinsicRejection`, `solutePermeability` or `observedRejection` |
| `Rint` | – | Intrinsic rejection coefficient (intrinsicRejection) |
| `B` | m/s | Solute permeability (solutePermeability) |
| `sigma` | – | Optional reflection coefficient (overrides solvent BC) |
| `Robs` | – | Target observed rejection (observedRejection) |
| `CAb` | kg/m$^3$ | Reference bulk concentration for diagnostics |
| `RintMin`, `RintMax` | – | Bounds for inverse rejection search |
| `tol`, `maxIter` | – | Tolerance and maximum iterations for secant/bisection |
| `UName` | – | Name of velocity field (defaults to `U`) |

## 5 Solver structure and implementation

The `centriMembSolver` follows a PIMPLE–type pressure–velocity coupling, extended with (i) concentration–dependent transport properties, (ii) a rotating reference frame, and (iii) custom membrane boundary conditions and diagnostics. The overall time loop is driven by `runTime.run()`, and at each time step a nested PIMPLE loop solves momentum, pressure and solute transport. Additional headers perform pressure reconstruction and conservation monitoring before the fields are written.

### 5.1 Overall time loop and custom PIMPLE algorithm

For readers familiar with C++, the main structure of `centriMembSolver.C` can be summarised schematically as

Table 3: Summary of membrane tracer flux boundary entries.

| Key | Units | Description |
|---|---|---|
| `model` | – | `intrinsicRejection`, `solutePermeability` or `observedRejection` |
| `RTint` | – | Intrinsic tracer rejection coefficient (intrinsicRejection) |
| `BT` | m/s | Tracer permeability (solutePermeability) |
| `sigmaT` | – | Optional tracer reflection coefficient (overrides solvent BC) |
| `RTobs` | – | Target observed tracer rejection (observedRejection) |
| `CTb` | kg/m$^3$ | Reference bulk tracer concentration for diagnostics |
| `RTintMin`, `RTintMax` | – | Bounds for inverse rejection search |
| `tol`, `maxIter` | – | Tolerance and maximum iterations for secant/bisection |
| `UName` | – | Name of velocity field (defaults to `U`) |

```
createTime;
createMesh;
createControl;              // pimpleControl
createFields;               // U, p_rgh, CA, etc.
membraneFields;             // Js, JsT, Jv, CAp, CTp, Gama, GamaT
updateTransportProperties;  // nu(CA), DAB(CA), rhok

while (runTime.run())
{
 // (normally: compute Courant number and time-step)

 ++runTime;
 Info<< "Time = " << runTime.timeName() << endl;

 while (pimple.loop())           // outer PIMPLE loop
 {
  #include "UEqn.H"           // assemble & solve momentum
  #include "CAEqn.H"          // assemble & solve solute
  #include "updateTransportProperties.H"

  while (pimple.correct())    // inner pressure loop
  {
   #include "pEqn.H"
  }

  if (pimple.turbCorr())
  {
   turbulence->correct();
  }
 }
```

```
  #include "CTEqn.H"
  #include "pReconstruct.H"
  #include "monitorConservation.H"
  #include "Gama.H"
  #include "GamaT.H"
  #include "forces.H"

  runTime.write();
  runTime.printExecutionTime(Info);
 }
```

Compared with a "standard" OpenFOAM transient solver:

- momentum (`UEqn.H`) and solute (`CAEqn.H`) are solved together at the beginning of each PIMPLE outer iteration;
- transport properties (viscosity, diffusivity, density) are updated *inside* the PIMPLE loop via `updateTransportProperties.H`, so they respond iteratively to the current concentration field;
- several post–processing steps (absolute pressure, conservation, membrane diagnostic $\Gamma$, body forces) are executed at the end of every time step.

This design makes the coupling between flow, solute transport and membrane behaviour explicit while still using the familiar PIMPLE machinery.

## 5.2 Field creation, membrane diagnostics and transport properties

Field creation is handled in two main headers:

- `createFields.H` reads the primary fields (`U`, `p_rgh`, `CA`, etc.) from the `0/` directory and the physical parameters from `constant/transportProperties` and `constant/SRFProperties`. It also constructs the turbulence model if required.
- `membraneFields.H` declares seven additional volume fields used for membrane diagnostics. These include `Js` and `JsT` (solute and tracer fluxes), `Jv` (solvent flux), `CAp` and `CTp` (permeate solute and tracer concentrations) and `Gama` and `GamaT` (dimensionless concentration polarisation factors for solute and tracer). All diagnostic fields are created with `AUTO_WRITE` so they are automatically written at output times.

The header `updateTransportProperties.H` is called once before the time loop and then at each outer PIMPLE iteration. It performs three updates:

1. *Viscosity*: the laminar transport model `laminarTransport` is corrected using the current concentration field `CA`, so that the kinematic viscosity $\nu(\text{CA})$ follows the chosen `CAViscosity` model.
2. *Molecular diffusivity*: the scalar field `DAB` is updated as a function of `CA`, using a polynomial relation. Boundary conditions on `DAB` are corrected afterwards.
3. *Density*: a non–dimensional density field `rhok` is recomputed according to a Boussinesq–type law, $\rho_k = 1 + \beta\,\text{CA}$, and its boundary conditions are corrected.

Updating these properties inside the PIMPLE loop strengthens the coupling between momentum and solute transport: any change in `CA` is immediately reflected in $\nu$, $D_{AB}$ and $\rho_k$.

## 5.3 Momentum and pressure equations

Inside the outer `while (pimple.loop())`:

- `UEqn.H` assembles the transient momentum equation using implicit time discretisation (`fvm::ddt`), a convection term `fvm::div(phi,U)`, viscous/turbulent stresses via `turbulence->divDevReff` and a Coriolis source term $2\rho\,\mathbf{\Omega} \times \mathbf{U}$ for the rotating frame. The equation is then solved for the velocity field `U`.
- `CAEqn.H` assembles and solves the unsteady advection–diffusion equation for the solute concentration `CA`, using a Crank–Nicolson scheme in time and a bounded convection scheme in space.

After these solves, `updateTransportProperties.H` is called again to refresh $\nu(\mathrm{CA})$, $D_{AB}(\mathrm{CA})$ and $\rho_k$ based on the newly updated `CA` field.

The pressure equation is then handled in the inner loop `while (pimple.correct())` through `pEqn.H`. This step enforces continuity, updates `p_rgh` (and the associated face flux `phi`) and accounts for the rotation–induced hydrostatic term and the specified pressure reference. When `pimple.turbCorr()` is true, the turbulence model is also corrected in the same outer iteration.

## 5.4 Membrane boundary conditions and the Γ diagnostic

The actual membrane physics is implemented in custom boundary condition classes applied to `U` and `CA`. These boundary conditions:

- compute, for each membrane face, the solvent flux $J_v$ and solute flux $J_s$ according to the selected transport model (e.g. pressure–driven with osmotic correction);
- update the permeate concentration $C_p$ and a dimensionless diagnostic $\Gamma$ that characterises local solute rejection or concentration polarisation.

At volume level, these per–face quantities are exported into the diagnostic fields declared in `membraneFields.H`. The header `Gama.H` is responsible for copying the per–face $\Gamma$ values from the membrane solute boundary condition into the boundary of the volume field `Gama`, so that `Gama` can be visualised and sampled like any other `volScalarField`. For efficiency, `Gama.H` caches the membrane patch indices and a pointer to the `CA` field the first time it is called.

## 5.5 Post-processing: pressure reconstruction, conservation and forces

After the PIMPLE loop converges at a given time, four post–processing headers are executed before writing:

`pReconstruct.H` The solver works internally with the modified pressure `p_rgh` to simplify the treatment of gravity in a rotating frame. For analysis, it is convenient to have the absolute pressure field. The header `pReconstruct.H` reconstructs

$$p = (p\_\mathrm{rgh} + \rho_k\, g\, h)\, \rho_0 + p_{\mathrm{ref}},$$

where $\rho_0$ is a reference density and $p_{\mathrm{ref}}$ the chosen pressure reference, and then calls `p.correctBoundaryConditio` to keep the boundaries consistent.

`monitorConservation.H` This header checks global solute conservation in a parallel–safe way. It requires the fields `CA`, `U`, `Js`, `Jv`, `CAp` and the face flux `phi`; an optional `Deff` field can be supplied to account for diffusive fluxes at walls. It builds (or appends to) the file

`postProcessing/conservation/cons.dat`,

writing at each output time the total solute mass in the domain, the time derivative of that mass, the integrated inlet, outlet, membrane and diffusive contributions, and a residual and relative error that measure how well the discrete balance is satisfied. This file is the main tool to verify that the coupled flow–membrane scheme is globally conservative.

**`Gama.H` and `GamaT.H`** As described above, `Gama.H` maps the per–face diagnostic $\Gamma$ from the membrane solute boundary condition into the volume field `Gama`. In the 2026 release, a companion header `GamaT.H` performs the analogous mapping for the tracer polarisation factor `GamaT`. This is done patch by patch on the membrane boundaries, with simple consistency checks (same number of faces) to avoid copying errors.

`forces.H` Finally, `forces.H` computes two auxiliary vector fields:

$$\begin{aligned}\mathbf{a}_{\text{Coriolis}} &= 2\,\boldsymbol{\Omega}\times\mathbf{U},\\ \mathbf{a}_{\text{buoy}} &= g\,h\,\nabla\rho_k,\end{aligned}$$

stored in the fields `aCoriolis` and `aBuoy`. These fields do not influence the solution; they are provided for post–processing and interpretation, allowing the user to visualise the spatial structure and relative magnitude of Coriolis and buoyancy effects in the rotating device.

At the end of this post–processing stage, `runTime.write()` writes all `AUTO_WRITE` fields (`U`, `p`, `p_rgh`, `CA`, `Js`, `Jv`, `CAp`, `Gama`, `aCoriolis`, `aBuoy`, etc.), and the solver advances to the next time level.

## 6 Case studies with `centriMembSolver`

This section presents two example cases that illustrate how to use `centriMembSolver`. They are designed to be readable even for readers who are not familiar with OpenFOAM.

In OpenFOAM, each simulation ("case") is stored in a directory with a standard structure:

- `0/` contains the *initial and boundary conditions* for all fields (for example velocity, pressure and concentration);
- `constant/` contains the *physical properties* (for example fluid properties, solute properties, membrane parameters, rotation settings);
- `system/` contains the *numerical setup* (mesh generation with `blockMesh`, time step controls, linear solvers, etc.).

To run a case, one typically:

1. generates the mesh (for example with `blockMesh` or a helper script);
2. runs the solver (here `centriMembSolver`);

3. inspects the results in ParaView or in text files written in `postProcessing/`.

Below we describe two specific cases: a simple 2D slit (`rotatingSlit`) and a centrifugal configuration (`O-CFM`).

## 6.1 `rotatingSlit`

The example case `rotatingSlit` distributed with the solver provides a simple yet instructive demonstration. It consists of a two-dimensional slit domain rotated about its centre. The membrane occupies a section of the side wall; the remaining boundaries are inlet, outlet, symmetry and empty patches.

### Geometry and mesh

The geometry is defined by `system/blockMeshDict`. A narrow slit of length 4 cm and width 1 mm is discretised into a structured mesh. One boundary, named `membraneB`, is assigned the membrane boundary conditions. A second boundary `membraneA` is present but disabled in this case. Front and back patches are declared as `empty` to produce a two-dimensional simulation.

### Initial and boundary conditions

The inlet velocity **U** is a fixed value, typically 0.01 m/s, driving flow along the slit. The initial solute concentration is uniform ($5\,\mathrm{kg/m^3}$) and the inlet condition sets `CAin` to a higher or lower value to induce concentration polarisation. The membrane on patch `membraneB` is configured with

- a target average solvent flux of $3 \times 10^{-5}$ m/s,
- limits `AhMin = 1e-13` and `AhMax = 1e-9`,
- reflection coefficient $\sigma = 1$, and
- virial coefficients $(7314.5\,\mathrm{Pa\,m^3\,kg^{-1}}, 8.253\,\mathrm{Pa\,m^6\,kg^{-2}})$.

On the solute side the membrane uses the intrinsic rejection model with `Rint = 0.99` and reference concentration `CAb = 1.0`. A fixed–flux pressure condition ensures continuity of volumetric flow.

### Running the case

To run the example, follow these steps in the case directory:

1. Compile the solver (only needed once) by executing `wmake` in the `solver` directory.
2. Generate the mesh using

```
blockMesh
```

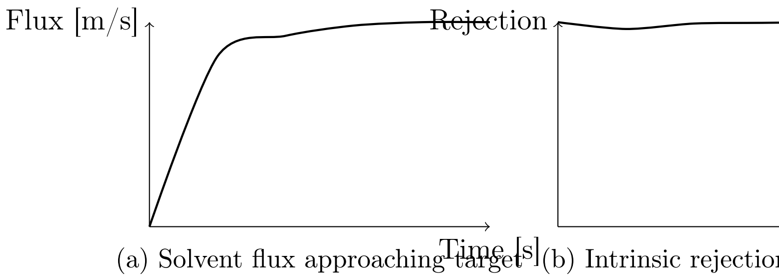
3. Adjust `constant/SRFProperties` to set the rotation speed $\boldsymbol{\Omega}$ if desired. The default example uses `rpm = 0`, but any non–zero value can be specified.

4. Run the simulation with

```
centriMembSolver -case rotatingSlit > log &
```

5. Monitor progress via the log or the files in `postProcessing`. Key outputs include `Rint_vs_time.dat` (intrinsic rejection over time), `calibratedProperties.dat` (effective permeability for target average flux) and `cons.dat` (mass conservation).

Figure 1 sketches typical results: the area– average solvent flux approaches the target $3\times10^{-5}$ m/s after a short transient; the intrinsic rejection remains close to the specified value; and the concentration polarisation builds up at the membrane.



(a) Solvent flux approaching target (b) Intrinsic rejectio

Figures (a) and (b) illustrate schematic behaviour of flux and rejection. Actual outputs should be generated from simul

Figure 1: Qualitative results from the `rotatingSlit` example: (a) the area–averaged solvent flux converges to the target value specified in the dictionary; (b) the intrinsic rejection computed by the solver stabilises to the prescribed parameter.

## 6.2 `O-CFM`

The `O-CFM` case represents a centrifugal membrane configuration, in which the domain rotates at a prescribed angular velocity. Rotation creates a radial pressure gradient that drives solvent through a membrane located at the outer radius. This case also demonstrates the use of helper scripts for mesh generation and parallel execution.

### Geometry and mesh

The geometry and mesh are defined indirectly through the script `makeMesh`, which in turn uses `system/blockMeshDict`. The domain can be thought of as an annular sector:

- an inner radius $r_{\text{in}}$ that acts as the feed region,
- an outer radius $r_{\text{out}}$ where the membrane is located,
- a small angular extent (sector) in the circumferential direction,
- front and back patches set to `empty` to obtain a 2D approximation.

The membrane is assigned to the outer radial wall (for example a patch named `outerWall` in the mesh). The mesh resolution and sector angle are chosen to provide a compromise between accuracy and runtime.

To generate the mesh, one simply runs

```
./makeMesh
```

in the case directory.

### Rotation, physics and solute properties

Rotation is defined in `constant/SRFProperties` by specifying the angular velocity (for example via `rpm`). The solver then includes the corresponding centrifugal and Coriolis terms in the momentum equation, so that the apparent pressure increases with radius.

The solute in this case is magnesium sulphate ($MgSO_4$). Its properties are stored in files such as `constant/MgSO4` and `constant/transportProperties`, which define, for example:

- diffusion coefficient of $MgSO_4$ in water,
- solution density and viscosity as functions of concentration,
- parameters for the osmotic pressure model (for example virial coefficients).

Boundary conditions follow the typical centrifugal filtration concept:

- feed is introduced near $r_{\text{in}}$ with a specified velocity and solute concentration,
- the membrane at $r_{\text{out}}$ imposes a controlled solvent flux and a given intrinsic rejection for $MgSO_4$,
- the pressure field adjusts to balance centrifugal forcing, membrane permeation and any imposed outlet or recycle conditions.

### Running the case

The `O-CFM` case is prepared to run in parallel. A convenience script `runCaseParallel` handles decomposition, solver execution and reconstruction. From the case directory one can execute:

```
./makeMesh
./runCaseParallel
```

Internally, this typically performs the following steps:

1. decomposes the domain into subdomains (for example using `decomposePar`),
2. calls `mpirun` or an equivalent launcher with `centriMembSolver`,
3. reconstructs the final fields for visualisation.

If desired, the case can also be run in serial by skipping the decomposition and calling `centriMembSolver` directly.

**Typical results and diagnostics**

In this centrifugal configuration the rotation produces:

- an increasing pressure profile from $r_{\text{in}}$ to $r_{\text{out}}$,
- enhanced solvent flux through the membrane at the outer radius,
- strong radial concentration gradients and concentration polarisation at the membrane,
- significant shear near the membrane, which tends to limit the thickness of the concentration boundary layer.

As with the `rotatingSlit` case, the solver can write diagnostic files in `postProcessing/` to monitor, for example:

- global mass conservation (similar to `cons.dat`),
- time evolution of average membrane flux,
- average intrinsic rejection of $MgSO_4$.

Together, the `rotatingSlit` and `O-CFM` cases provide a pair of simple but representative examples: a planar slit for basic membrane calibration and a rotating annulus for centrifugal membrane operation.

# 7 Developer Notes and Best Practices

## 7.1 Extending boundary conditions

New flux models can be added by subclassing the existing patch field classes. To create a new solvent flux law, derive from `membraneSolventFluxFvPatchVectorField`, add a new enumerated identifier in the `Model` enum and implement the logic in `updateCoeffs()`. Register the class with the runtime selection table using `addToRunTimeSelectionTable`. Follow a similar procedure for solute flux models.

Ensure that any new model reads user–specified parameters from the dictionary and writes meaningful diagnostics. When coupling new physics (e.g. temperature or multiple solutes), extend the field lists in `createFields.H` and update the transport properties accordingly.

## 7.2 Stability and time stepping

Although the semi–implicit Coriolis integration relaxes the time–step constraint on rotation, the convective CFL condition still applies. Maintain the Courant number below 1 near the membrane. When using target average flux, choose `AhMin` and `AhMax` such that the calibration remains stable; too wide a range may lead to oscillatory behaviour. For observed rejection, choose tight tolerances `tol` and ensure that the target lies within the physically realistic range defined by `RintMin` and `RintMax`.

### 7.3 Parallel consistency

The boundary conditions and diagnostics use parallel reductions to ensure consistent results across processors. Nevertheless, for complex geometries or very fine meshes it is advisable to use `redistributePar` when decomposing the domain so that membrane patches are not split across processors. The conservation monitor provides a quick check on mass balances across processor boundaries.

### 7.4 Further development

Possible extensions include: coupling multiple solutes with different permeabilities; including heat transfer and temperature–dependent permeability; implementing steric hindrance models; coupling with chemical reactions or variable membrane area. The modular design of `centriMembSolver` facilitates such developments. To implement temperature effects, for instance, one would introduce a temperature field, modify the viscosity and osmotic pressure models to depend on temperature and augment the boundary condition dictionaries with temperature–dependent coefficients.

## 8 References

1. Spiegler, K. S. & Kedem, O. (1966). Thermodynamics of hyperfiltration (reverse osmosis): criteria for efficient membranes. *Desalination*, 1, 311–326.

2. Wu, J. J. (2019). On the application of the Spiegler–Kedem model to forward osmosis. *BMC Chemical Engineering*, 1, 15. (Discusses how the S–K model allows deviation from ideal semi–permeability.)

3. Coriolis force formula. The acceleration of a mass in a rotating frame includes a term $-2\,\boldsymbol{\Omega} \times \mathbf{v}$ acting orthogonal to the velocity.

## A Appendix A: Implementing Boundary Conditions in OpenFOAM

This appendix is intended for students who want to understand how the membrane boundary conditions are implemented inside OpenFOAM, and for those who may wish to extend or modify them.

### A.1 B.1 Boundary conditions as C++ objects

Every boundary condition in OpenFOAM is a small C++ class derived from:

- `fvPatchField` (generic base),
- `fvPatchScalarField` (scalar fields),
- `fvPatchVectorField` (vector fields).

Each patch entry in `0/U`, `0/p` or `0/CA` creates one of these C++ objects at run time.

A custom BC must define:

- a **TypeName** so OpenFOAM can select it,
- constructors to read user parameters,
- an **updateCoeffs()** method to apply the physics.

The file structure is typically:

```
MyBC/
MyBCFvPatchField.H
MyBCFvPatchField.C
```

Compiled into a dynamic library with `wmake libso`.

## A.2 B.2 What constructors do (and must NOT do)

Constructors:

- read parameters from the dictionary,
- initialise private member data,
- check dimensions and required fields.

They must *not* impose the boundary condition. This is the role of `updateCoeffs()`.

## A.3 B.3 The updateCoeffs() method

`updateCoeffs()` is the core of every BC. It is called whenever OpenFOAM assembles the matrix for the equation being solved.

Its responsibilities:

1. Guard against double updates with `if (updated()) return;`
2. Read internal values (e.g. $C_i$) from the adjacent cells.
3. Compute the required boundary physics (e.g. $C_w$, $J_s$, $C_p$).
4. Set the mixed condition via:

   ```
   valueFraction()[f] = alpha;
   refValue()[f] = Cref;
   refGrad()[f]  = gradient;
   ```

5. Call the base-class update with:

   ```
   mixedFvPatchScalarField::updateCoeffs();
   ```

For the solute membrane BC, this method contains the entire membrane transport model; the solver itself does not handle membrane physics.

## A.4 B.4 Accessing geometry and fields

Useful geometry functions inside a BC:

- `patch().Sf()` : face area vectors,
- `patch().nf()` : face normals,
- `patch().magSf()` : magnitudes of areas,
- `deltaCoeffs()` : $1/\Delta x$ across the boundary.

Useful to read fields:

- `this->patchInternalField()` : internal cell values,
- `db().lookupObject` : access other fields (e.g. `Jv`).

## A.5 B.5 Using custom boundary conditions

After compiling your BC:

1. Add the library name to `system/controlDict`:

   ```
   libs ("libmembraneBCs.so");
   ```

2. Specify the BC in the field file:

   ```
   membrane
   {
    type membraneSoluteFlux;
    CAp CAp;
    Jv  Jv;
    Js  Js;
   }
   ```

3. Run the solver — the BC is applied automatically.

## A.6 B.6 Debugging and good practice

- Insert `Info` « messages inside `updateCoeffs()`.
- Always ensure consistent units (OpenFOAM checks dimensions).
- To diagnose conservation, print $J_s$, $J_v$, $C_p$ each iteration.
- For parallel runs, use `Pstream::reduce()` to sum fluxes.

This appendix should provide enough structure for students to navigate and extend the membrane boundary conditions within OpenFOAM.