Implementation of non-reflecting boundary conditions in OpenFOAM

Leandro Lucchese

Ph.D student,
Department of Mechanical and Aerospace engineering,
Sapienza university,
Rome, Italy.

January 15, 2023

Introduction

Whenever a CFD analysis is performed there will have to be, at some point, boundaries.

These boundaries can be physical, e.g walls, or numerical, e.g outlets from which the flow exits the domain).

When performing simulations of compressible flows, **reflection of waves** on these type of outlets can create problems.

There are many ways to solve this problem, but most of them involve solving the equations in an additional non-physical domain, which **highly increases the computational cost**.

One way of solving this issue is by using **non-reflecting boundary conditions (NRBCs)**, which are usually based on the **characteristic analysis** of the equations.

The goal of this type of boundary conditions is to have no reflection of waves of fluid quantities at the boundary.

Characteristic analysis of the N-S equations

N-S equations for compressible viscous flow:

$$\frac{\partial \rho}{\partial t} + \frac{\partial}{\partial x_i}(m_i) = 0,$$
$$\frac{\partial m_i}{\partial t} + \frac{\partial}{\partial x_j}(m_i u_j) + \frac{\partial p}{\partial x_i} = \frac{\partial \tau_{ij}}{\partial x_j},$$
$$\frac{\partial \rho E}{\partial t} + \frac{\partial}{\partial x_i}[(\rho E + p)u_i] = \frac{\partial}{\partial x_i}(u_j \tau_{ij}) - \frac{\partial q_i}{\partial x_i},$$

Where $m_i = \rho u_i$, and $\rho E = \frac{1}{2}\rho u_k u_k + \frac{p}{\gamma - 1}$.

In vector form, the system can be written as:

$$\frac{\partial \widetilde{\boldsymbol{U}}}{\partial t} + \frac{\partial \widetilde{\boldsymbol{F}}^i}{\partial x_i} + \frac{\partial \widetilde{\boldsymbol{D}}^i}{\partial x_i} = \boldsymbol{0},$$

Where $\widetilde{\boldsymbol{U}} = |\rho \quad \rho u_1 \quad \rho u_2 \quad \rho u_3 \quad \rho E|^{\mathrm{T}}$ is the vector of conservative variables.

Characteristic analysis of the N-S equations (2)

If a vector of primitive variables is defined as $\boldsymbol{U} = \left| \begin{array}{ccccc} \rho & u_1 & u_2 & u_3 & p \end{array} \right|^{\mathrm{T}}$, the equation can be rewritten in terms of primitive variables as

$$\frac{\partial \boldsymbol{U}}{\partial t} + \boldsymbol{F}^i \frac{\partial \boldsymbol{U}}{\partial x_i} + \boldsymbol{D} = \boldsymbol{0},$$

where

- $\boldsymbol{D} = \boldsymbol{P}^{-1} \partial \widetilde{\boldsymbol{D}}^i / \partial x_i$ includes all the viscous and diffusive terms.
- $\boldsymbol{F}^k$ is the non-conservative Jacobian matrix related to the k th direction.
- $\boldsymbol{P} = \partial \widetilde{\boldsymbol{U}} / \partial \boldsymbol{U}$ is the Jacobian matrix that allows to change coordinates between primitive and conservative variables.

Characteristic analysis of the N-S equations (3)

Each non-conservative Jacobian matrix $\boldsymbol{F}^k$ related to every direction k can be diagonalized through

$$\boldsymbol{S}_k^{-1}\boldsymbol{F}^k\boldsymbol{S}_k = \boldsymbol{\Lambda}^k,$$

and the eigenvalues are given by

$$\begin{aligned}\lambda_1^k &= u_k - c,\\ \lambda_{2,3,4}^k &= u_k,\\ \lambda_5^k &= u_k + c,\end{aligned}$$

where $c = \sqrt{\frac{\gamma p}{\rho}}$ is the speed of sound.

Once the equations have been written in this form, **a wide variety of different boundary conditions can be considered** depending on the type of boundary type.

Characteristic analysis of the N-S equations (4)

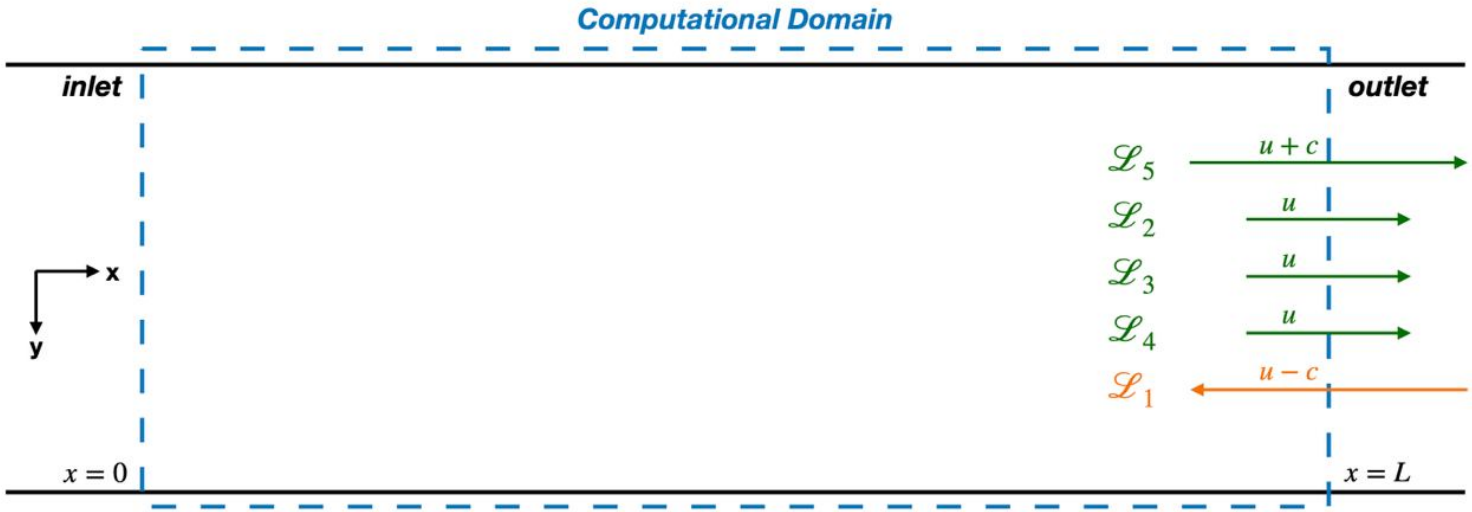

Figure: 2D domain with waves leaving and entering the domain

When considering a boundary with normal vector parallel to x_1, the characteristic waves considered will be those traveling along the x_1 direction and therefore **only $\boldsymbol{F}^1$ needs to be diagonalized**, the equation can be written as

$$\frac{\partial \boldsymbol{U}}{\partial t} + \boldsymbol{S}_1 \boldsymbol{\Lambda}^1 \boldsymbol{S}_1^{-1} \frac{\partial \boldsymbol{U}}{\partial x_1} + \boldsymbol{F}^2 \frac{\partial \boldsymbol{U}}{\partial x_2} + \boldsymbol{F}^3 \frac{\partial \boldsymbol{U}}{\partial x_3} + \boldsymbol{D} = \boldsymbol{0},$$

Characteristic analysis of the N-S equations (5)

a vector $\mathcal{L}$ whose components $\mathcal{L}_i$ represent time variation of the amplitudes of the characteristic waves can be defined

$$\mathcal{L} = \boldsymbol{\Lambda}^1 \boldsymbol{S}_1^{-1} \frac{\partial \boldsymbol{U}}{\partial x_1} = \begin{pmatrix} \lambda_1 \left(\frac{\partial p}{\partial x_1} - \rho c \frac{\partial u_1}{\partial x_1} \right) \\ \lambda_2 \left(c^2 \frac{\partial \rho}{\partial x_1} - \frac{\partial p}{\partial x_1} \right) \\ \lambda_3 \frac{\partial u_2}{\partial x_1} \\ \lambda_4 \frac{\partial u_3}{\partial x_1} \\ \lambda_5 \left(\frac{\partial p}{\partial x_1} + \rho c \frac{\partial u_1}{\partial x_1} \right) \end{pmatrix}.$$

The conservation equation can finally be written as a function of the wave amplitude variations obtaining

$$\frac{\partial \boldsymbol{U}}{\partial t} + \boldsymbol{S}_1 \mathcal{L} + \boldsymbol{F}^2 \frac{\partial \boldsymbol{U}}{\partial x_2} + \boldsymbol{F}^3 \frac{\partial \boldsymbol{U}}{\partial x_3} + \boldsymbol{D} = \boldsymbol{0},$$

Characteristic analysis of the N-S equations (6)

For a 1-D inviscid flow (Euler flow), all terms regarding derivatives along the x_2 and x_3 directions, together with the $\boldsymbol{D}$ vector are set to zero.

The system arising from this description is called **local one dimensional inviscid (LODI)** system

$$
\begin{aligned}
&\frac{\partial \rho}{\partial t} + \frac{1}{c^2}\left[\mathcal{L}_2 + \frac{1}{2}\left(\mathcal{L}_5 + \mathcal{L}_1\right)\right] = 0,\\
&\frac{\partial p}{\partial t} + \frac{1}{2}\left(\mathcal{L}_5 + \mathcal{L}_1\right) = 0,\\
&\frac{\partial u_1}{\partial t} + \frac{1}{2\rho c}\left(\mathcal{L}_5 - \mathcal{L}_1\right) = 0,\\
&\frac{\partial u_2}{\partial t} + \mathcal{L}_3 = 0,\\
&\frac{\partial u_3}{\partial t} + \mathcal{L}_4 = 0,
\end{aligned}
$$

Characteristic based boundary conditions

By imposing different conditions to the amplitudes of characteristic waves $\mathcal{L}_i$ a series of physically meaningful boundary conditions can be imposed.

- A fixed pressure boundary condition for example can be obtained by setting $\mathcal{L}_5 = -\mathcal{L}_1$ to fix the amplitude variation of the wave entering the domain
- **For a perfectly non-reflecting boundary condition** the incoming wave amplitude has to be set to zero $\mathcal{L}_1 = 0$

This is an approximation, but in cell centered finite volume methods the contribution of every boundary face enters the system by summing a term to the diagonal part of the coefficient matrix or to the source term (or both).

The error made when using the LODI equations at the boundary is limited since the contribution of the boundary face is only one of many faces.

Perfectly non-reflecting boundary conditions

If the flow is in the positive x_1 direction, to impose a **perfectly non-reflecting boundary condition**, the upcoming wave amplitude has to be set to zero $\mathcal{L}_1 = 0$.

The LODI system of equations becomes:

$$\begin{aligned}
&\frac{\partial p}{\partial t} + (u_1 + c)\frac{\partial p}{\partial x} = 0,\\
&\frac{\partial u_1}{\partial t} + (u_1 + c)\frac{\partial u_1}{\partial x} = 0,\\
&\frac{\partial u_2}{\partial t} + u_1\frac{\partial u_2}{\partial x} = 0,\\
&\frac{\partial u_3}{\partial t} + u_1\frac{\partial u_3}{\partial x} = 0,
\end{aligned}$$

These equations have to be applied to the primitive variables in order to find their values at the boundary face.

Partially non-reflecting boundary conditions

Using perfectly non-reflecting conditions is often non recommendable since it may lead to an ill-posed problem.

For example, with a fixed mass flow rate inlet and a perfectly non-reflecting outlet, **there will be no constaint on the pressure** and it will drift form the initial value.

In order to add some physical information on the mean static pressure, **partially non-reflective** boundary conditions can be applied by imposing

$$\mathcal{L}_1 = K(p - p_\infty),$$

This is equivalent to imagining an outlet at a certain distance from the domain with pressure p_∞ that sends waves into the domain.

When this is applied the pressure equation becomes:

$$\frac{\partial p}{\partial t} + (u_1 + c)\frac{\partial p}{\partial x} + K(p - p_\infty) = 0,$$

Perfectly non-reflecting BCs in finite volume method

Every LODI equation is an **advection equation** with advection speed W

$$\frac{\partial \phi}{\partial t} + W \frac{\partial \phi}{\partial x} = 0.$$

This equations can be discretized (for example with an Euler time scheme), becoming

$$\frac{\phi_f^{n+1} - \phi_f^n}{\Delta t} + W \frac{\phi_f^{n+1} - \phi_c^{n+1}}{\Delta x} = 0,$$

that, when manipulated, becomes

$$\phi_f^{n+1} = \phi_f^n \frac{1}{1+\alpha} + \frac{\alpha}{1+\alpha} \phi_c^{n+1},$$

with $\alpha = \Delta t W / \mathbf{d}$.

Partially non-reflecting BCs in finite volume method

Applying the same procedure to the **partially non-reflecting** equation, after some manipulations the result becomes

$$\phi_f^{n+1} = (\phi_f^n + k\phi^\infty)\frac{1}{1+\alpha+k} + \frac{\alpha}{1+\alpha+k}\phi_c^{n+1},$$

These formulas are extremely important since **they correspond exactly to the way OpenFOAM defines its non-reflecting boundary conditions**.

General boundary conditions in OpenFOAM

The two main functions used for discretizing the PDE's in OpenFOAM are `fvm::div` and `fvm::laplacian`, and they can be seen in every top level solver.

These functions create the linear system corresponding to the discretized PDE in terms of coefficient matrix and source term.

In this context, boundary conditions contribute either to the **diagonal coefficients** of the coefficient matrix, to the **source term**, or both. This can be clearly seen in the `fvmDiv` function of the `gaussConvectionScheme`

General boundary conditions in OpenFOAM (2)

a part of the `fvmDiv` function of `gaussConvectionScheme.C`

```
76      forAll(vf.boundaryField(), patchi)
77      {
78          const fvPatchField<Type>& psf = vf.boundaryField()[patchi];
79          const fvsPatchScalarField& patchFlux = faceFlux.boundaryField()[patchi];
80          const fvsPatchScalarField& pw = weights.boundaryField()[patchi];
81
82          fvm.internalCoeffs()[patchi] = patchFlux*psf.valueInternalCoeffs(pw);
83          fvm.boundaryCoeffs()[patchi] = -patchFlux*psf.valueBoundaryCoeffs(pw);
84      }
```

Two functions `valueInternalCoeffs()` and `valueBoundaryCoeffs()` of the class `fvPatchField<Type>` provide the **contribution of the boundary conditions** to the diagonal term of the coefficient matrix and the source term.

The Laplacian term works the same way but by using the functions `gradientInternalCoeffs()` and `gradientBoundaryCoeffs()`.

`mixed` boundary conditions in OpenFOAM

Boundary conditions in OpenFOAM can be either `fixedValue`, `fixedGradient`, or a mixture of the two, namely `mixed`.

The classes used for non reflecting boundaries are the `advective` and `waveTranmissive`, and are both sub-classes of the `mixed` boundary condition.

The way mixed boundary conditions work in OpenFOAM is by defining the value of the field at the boundary face as

$$\phi_f = w\phi_{\text{ref}} + (1 - w)(\phi_c + \mathbf{d}\nabla(\phi_{\text{ref}})),$$

where:

- ϕ_f is the boundary face value,
- ϕ_c is the boundary cell value,
- ϕ_{ref} is a reference value,
- $\mathbf{d}$ is the face-to-cell distance,
- w is the "value fraction".

`mixed` boundary conditions in OpenFOAM (2)

The main functions of the `mixedFvPatchField` class are:

- `evaluate`: Evaluates the patch Field.
- `snGrad`: Returns the patch normal gradient.
- `valueInternalCoeffs`: Returns the contribution of the divergence term to the coefficient matrix of the linear system at boundary patch.
- `valueBoundaryCoeffs`: Returns the contribution of the divergence term to the source term of the linear system at boundary patch.
- `gradientInternalCoeffs`: Returns the contribution of the laplacian term to the coefficient matrix of the linear system at boundary patch.
- `gradientBoundaryCoeffs`: Returns the contribution of the laplacian term to the source term of the linear system at boundary patch.

mixed boundary conditions in OpenFOAM (3)

valueInternalCoeffs function of mixedFvPatchField.C

```
92 template<class Type>
93 Foam::tmp<Foam::Field<Type>>
94 Foam::mixedFvPatchField<Type>::valueInternalCoeffs
95 (
96     const tmp<scalarField>&
97 ) const
98 {
99     return Type(pTraits<Type>::one)*(1.0 - valueFraction_);
00 }
```

Type(pTraits<Type>::one) substantially returns one, depending on the Type of the field given as input.

If we call the variable valueFraction_ = f, the function valieInternalCoeffs returns (1 - f).

The return of this function will be summed to the diagonal coefficient of the matrix, i.e will multiply ϕ_c.

mixed boundary conditions in OpenFOAM (4)

valueBoundaryCoeffs function of mixedFvPatchField.C

```
03 template<class Type>
04 Foam::tmp<Foam::Field<Type>>
05 Foam::mixedFvPatchField<Type>::valueBoundaryCoeffs
06 (
07     const tmp<scalarField>&
08 ) const
09 {
10     return
11         valueFraction_*refValue_
12       + (1.0 - valueFraction_)*refGrad_/this->patch().deltaCoeffs();
13 }
```

The function `valueBoundaryCoeffs` returns `f*refValue + (1-f)*refGrad*d`, this output will sum the RHS of the system.

If the two expressions are merged in order to obtain a single formula for the value that this boundary condition imposes to the field at the boundary patch, we obtain

$$\phi_f^{n+1} = f * \mathrm{refValue} + (1 - f)(\phi_c^{n+1} + \mathrm{refGrad} * \mathbf{d}).$$

advective boundary conditions in OpenFOAM

The `advectiveFvPatchField` class is **one of the two main classes used for non-reflecting boundary conditions** in OpenFOAM.

It is a **sub-class** of the `mixedFvPatchField` class.

The two main functions of the `advectiveFvPatchField` class are:

- `advectionSpeed()`: calculates and returns the value of the advection speed at the boundary.
- `updateCoeffs()`: updates the coefficients associated to the patch field, i.e the `refValue_`, `valueFraction_` and `refGrad_` used in the `mixedFvPatchField` boundary condition.

The `advectionSpeed` function returns `phip/this->patch().magSf()` where `phip` is the flux that in OpenFOAM is calculated as $\mathbf{U} \cdot \mathbf{S}_f$. The function hence returns u_n, the **velocity normal to the patch**.

advective boundary conditions in OpenFOAM (2)

updateCoeffs function of advectiveFvPatchField.C

```
1  const scalarField w(Foam::max(advectionSpeed(), scalar(0)));
2  const scalarField alpha(w*deltaT*this->patch().deltaCoeffs());
3
4  if (lInf_ > 0)
5  {
6  const scalarField k(w*deltaT/lInf_);
7
8  if
9  (
10     ddtScheme == fv::EulerDdtScheme<scalar>::typeName
11  || ddtScheme == fv::CrankNicolsonDdtScheme<scalar>::typeName
12 )
13 {
14     this->refValue() =
15     (
16         field.oldTime().boundaryField()[patchi] + k*fieldInf_
17     )/(1.0 + k);
18     this->valueFraction() = (1.0 + k)/(1.0 + alpha + k);
19 }
20
```

advective boundary conditions in OpenFOAM (3)

updateCoeffs function of advectiveFvPatchField.C

```
else
{
    if
    (
        ddtScheme == fv::EulerDdtScheme<scalar>::typeName
     || ddtScheme == fv::CrankNicolsonDdtScheme<scalar>::typeName
    )
    {
        this->refValue() = field.oldTime().boundaryField()[patchi];

        this->valueFraction() = 1.0/(1.0 + alpha);
    }

```

When the paremeter `lInf_ = 0` (fully non-reflecting conditions),

- `refValue` $= \phi_f$ at the previous time step,
- `valueFraction = 1/(1 + alpha)`, with `alpha`$= \frac{w\Delta t}{\mathbf{d}}$.

advective boundary conditions in OpenFOAM (4)

When `lInf_ > 0` (partially non-reflecting), the function defines a parameter `k`

$$k = \frac{w\Delta t}{\mathrm{lInf}},$$

and then changes the values of `refValue` and `valueFraction` as

$$\mathrm{refValue} = (\phi_f + k\phi^\infty)\frac{1}{1+k},$$
$$\mathrm{valueFraction} = \frac{1+k}{1+\alpha+k},$$

Where expressions are substituted in the return of `mixedFvPatchField`, we obtain:

$$\phi_f^{n+1} = (\phi_f^n + k\phi^\infty)\frac{1}{1+\alpha+k} + \frac{\alpha}{1+\alpha+k}\phi_c^{n+1},$$

I.e, the same formula obtained from the LODI system for partially non-reflecting boundaries **with an advection speed** $w = u_n$.

waveTranmissive boundary conditions in OpenFOAM

The waveTranmissiveFvPatchField class is a **sub-class** advectiveFvPatchField class.

It only has one function called advectionSpeed, which overrides the same function of its base class advectiveFvPatchField.

The function returns

```
phip/this->patch().magSf() + sqrt(gamma_/psip)
```

Therefore, the waveTransmissive class works exactly like the advective, i.e **it transports the field at the boundary through a simple advection equation**, but with advection speed

$$w = u_n + \sqrt{\frac{\gamma}{\psi}} = u_n + c,$$

How to use NR boundary conditions in OpenFOAM

According to the theory, the way these boundary conditions should be used is by **applying an `advective` boundary condition to the variables that are advected with u_n and a `waveTransmissive` boundary condition to those that travel with $u + c$.**

In the OpenFOAM-v2112 tutorials however, the `waveTransmissive` boundary condition is used in 10 tutorials and **is always only applied only to pressure** while for velocity, temperature and other variables the `inletOutlet` boundary condition is used.

In order to understand how to properly use this boundary conditions, a simple test case has been studied:

- A 2-D square domain with a fixed temperature constraint at the center (so-called "spark") causing a pressure wave to travel towards the output.

Test cases setup

The simulation has been runned twice with two sets of boundary conditions

	Case 1	**Case 2**
p	waveTransmissive	waveTransmissive
U	pressureInletOutletVelocity	waveTransmissive
T	inletOutlet	advective
lInf	10	10

- **Case 1** corresponds to the OpenFOAM tutorials' way to apply non-reflecting boundary conditions.
- **Case 2** corresponds to the way these boundary conditions should be used according to the LODI theory.

Results

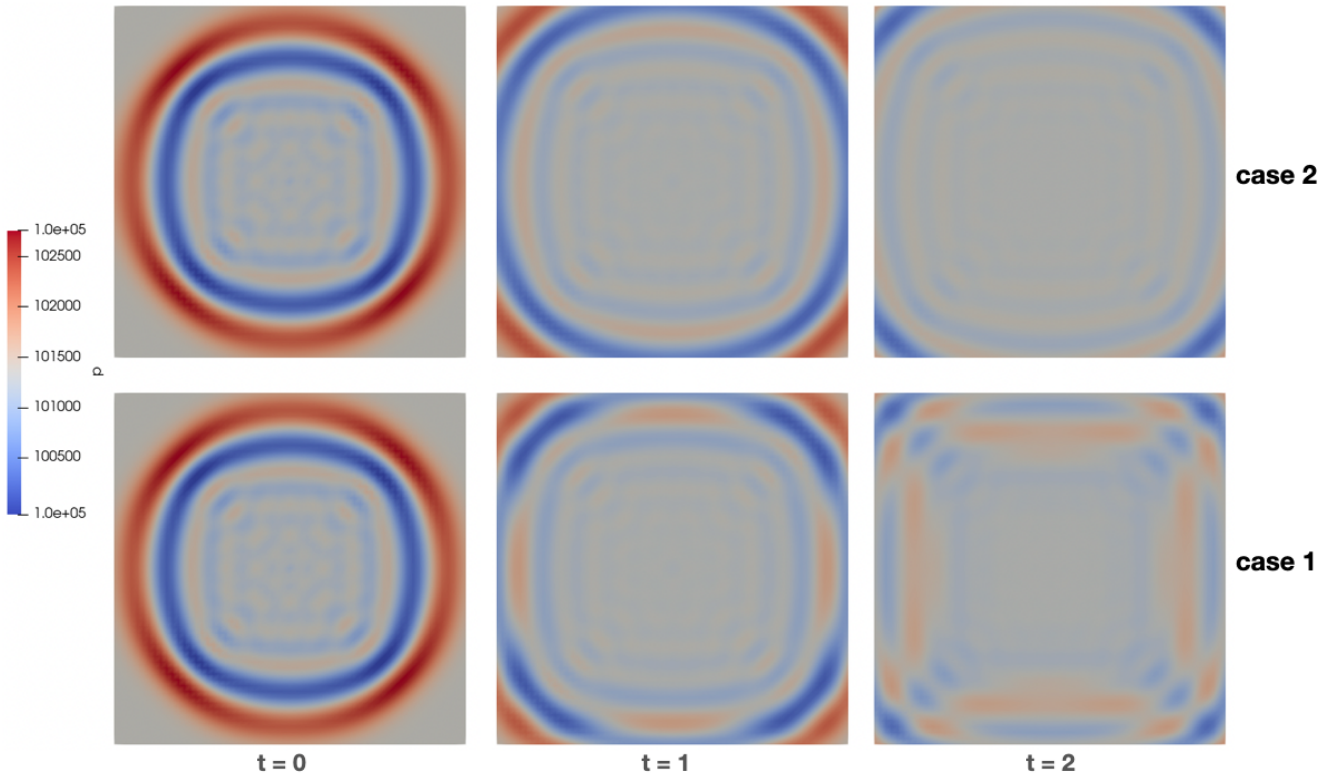

Figure: Pressure field for three different time steps in the 2-D square simulation

Results show that **the correct way to apply OpenFOAM's native set of boundary condition is that described by Case 2**.

The problem with OpenFOAM's approach

The problem OpenFOAM's approach is that, according to the LODI theory, the only velocity component that should be advected with an advection speed $w = u_n + c$ is **the one normal to the outlet patch**.

Instead, the components of the velocity **orthogonal to the outlet patch** should travel with $w = u_n$.

To fix this problem, the boundary conditions have to be modified in order to:

1. Project the velocity on a reference frame normal to the outlet patch.
2. Separately solve the transport equations for the **normal component** u_n with an advection speed $w = u_n + c$, and the **tangential component** u_t with $w = u_n$.
3. Project the velocity back to the cartesian reference frame.

Projecting the velocity vector

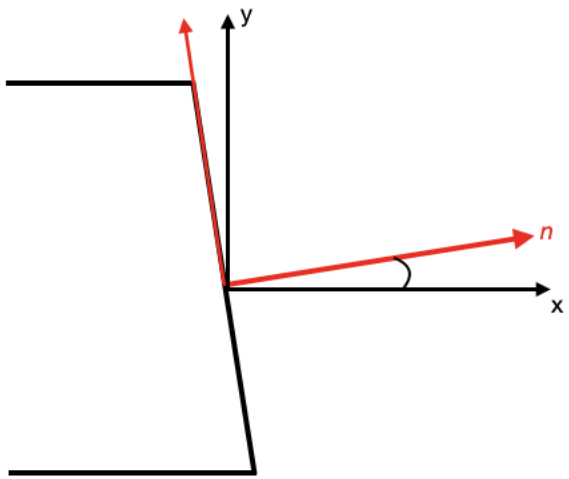

Figure: Patch normal vector in the cartesian reference frame

the components of the patch normal and tangential vectors **n** and **t** in the Cartesian reference frame are

$$\mathbf{n} = \begin{bmatrix} \cos(\theta) \\ \sin(\theta) \end{bmatrix}, \quad \mathbf{t} = \begin{bmatrix} -\sin(\theta) \\ \cos(\theta) \end{bmatrix},$$

Projecting the velocity vector (2)

The velocity components in the patch-normal reference frame u_n and u_t are calculated through:

$$\begin{bmatrix} u_n \\ u_t \end{bmatrix} = \begin{bmatrix} \cos(\theta) & \sin(\theta) \\ -\sin(\theta) & \cos(\theta) \end{bmatrix} \begin{bmatrix} u \\ v \end{bmatrix},$$

They have to be transported at the boundary face according to the LODI relations, which for a fully non-reflecting case are:

$$(u_n)_f^{n+1} = (u_n)_f^n \frac{1}{1+\alpha_{uc}} + \frac{\alpha_{uc}}{1+\alpha_{uc}} (u_n)_c^{n+1},$$
$$(u_t)_f^{n+1} = (u_t)_f^n \frac{1}{1+\alpha_{u}} + \frac{\alpha_{u}}{1+\alpha_{u}} (u_t)_c^{n+1},$$

Where the **normal component** is advected with $u_n + c$, hence $\alpha_{uc} = \frac{\Delta t(u_n+c)}{\mathbf{d}}$.

and the **tangential component** with u_n, hence $\alpha_u = \frac{\Delta t u_n}{\mathbf{d}}$.

Projecting the velocity vector (3)

Once the velocity components are transported, it's necessary to **transform them back the Cartesian reference frame** through

$$\begin{bmatrix} u \\ v \end{bmatrix}_f^{n+1} = \begin{bmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{bmatrix} \begin{bmatrix} u_n \\ u_t \end{bmatrix}_f^{n+1} =$$
$$(u_n)_f^{n+1} \begin{bmatrix} \cos(\theta) \\ \sin(\theta) \end{bmatrix} + (u_t)_f^{n+1} \begin{bmatrix} -\sin(\theta) \\ \cos(\theta) \end{bmatrix},$$

This procedure has been implemented in the custom non-reflecting boundary conditions

For this purpose, two new classes have been created: "`basic/mixedV2D`" and "`derived/LODI2D`" by modifying, adding functionalities and merging the `mixed`, `advective` and `waveTransmissive` boundary conditions.

Implementation of the `mixedV2D` boundary condition

A new version of the `mixed` boundary condition named `mixedV2D` has been created, containing a series of new member data:

- `Un_` and `Ut_` are the velocities normal and tangential to the patch.
- `n_` is the patch normal vector.
- `refValueU_` and `refValueUC_` are the value fields necessary to build the transport equations of the normal and tangential velocity components with different advection speeds.
- `valueFractionU_` and `valueFractionUC_` are the weights necessary to build the transport equations of the normal and tangential velocity components.
- `vector1_`, `vector2_` and `vector3_` are the three vectors that define the rotation matrix to go back to Cartesian coordinates.

Differently from the `mixed` class, which is a **templated class**, the new `mixedV2D` boundary condition has to be defined **only for vectorField type of inputs**, hence all the functions have to be re-defined accordingly.

Implementation of the `mixedV2D` boundary condition (2)

`vector1_`, `vector2_` and `vector3_` data are initialized in the constructors.

Part of a constructor of the `mixedV2DFvPatchField` class

```
57      n_ = this->patch().nf();
58      forAll(vector1_, i)
59      {
60          vector1_[i][0] = n_[i][0];
61          vector1_[i][1] = n_[i][1];
62          vector1_[i][2] = n_[i][2];
63      }
64      forAll(vector2_, i)
65      {
66          vector2_[i][0] = -n_[i][1];
67          vector2_[i][1] = n_[i][0];
68          vector2_[i][2] = n_[i][2];
69      }
70      forAll(vector3_, i)
71      {
72          vector3_[i][0] = 0.0;
73          vector3_[i][1] = 0.0;
74          vector3_[i][2] = 1.0;
```

Implementation of the `mixedV2D` boundary condition (3)

The components of the velocity in the patch-normal reference frame are calculated by using the patch normal vector `n`

Part of the `evaluate` function of the `mixedV2DFvPatchField` class

```
23 forAll(U, i)
24     {
25         Un[i] = U[i][0]*n[i][0] + U[i][1]*n[i][1]; //ucos+vsin
26         Ut[i] = -U[i][0]*n[i][1] + U[i][1]*n[i][0]; //-usin+vcos
27     }
```

and are used in the various functions of the class to define the rest of the member data.

Implementation of the `mixedV2D` boundary condition (4)

The functions of the new class use two different `refValue_` and `valueFraction_` data, one referred to the u_n component and the other to u_t. Before returning the output, these are multiplied by the vectors to go from patch-normal to Cartesian reference frame.

`valueInternalCoeffs` function of the `mixedV2DFvPatchField` class

```
84 Foam::tmp<Foam::vectorField>
85 Foam::mixedV2DFvPatchVectorField::valueInternalCoeffs
86 (
87     const tmp<scalarField>&
88 ) const
89 {
90     scalarField valueU =
91     (1.0 - valueFractionU_);
92
93     scalarField valueUC =
94     (1.0 - valueFractionUC_);
95
96     return vector1_ * valueUC + vector2_ * valueU;
97 }
```

Implementation of the LODI2D boundary condition

The member variables `valueFractionU_`, `valueFractionUC_`, `refValueU_` and `refValueUC_` used inside the functions of `mixedV2D` are defined inside a second class that inherits from `mixedV2D`, called `LODI2D`.

This class' structure is identical to the `advective` class, but with some differences:

- It is not templated, it has to take into account only vector type of inputs.
- It has **two different advectionSpeed functions** in order to transport the normal and orthogonal components of the velocity differently.
- It has much more member data, required to assign the values to the data of the `mixedV2D` class.

Implementation of the LODI2D boundary condition (2)

The member data added to the LODI2D class are:

- Unold_, Utold_, Unoold_ and Utoold_ are the normal and tangential velocity vectors at the previous and even previous times.
- fieldInf_ is the field (the velocity in this case) at infinity.
- UnInf_ is the component of the velocity at infinity normal to the boundary patch.
- lInf_ is the relaxation length used to calculate the strength of the reflecting wave when considering partially non-reflecting boundary conditions.

The Unold_, Utold_, Unoold_ and Utoold_ data are calculated through the field.oldTime() and the field.oldTime.oldTime() functions, and projected into the patch-normal reference frame.

Implementation of the LODI2D boundary condition (2)

Part of the updateCoeffs function of the LODI2DFvPatchField class

```
84      const scalarField wU(Foam::max(advectionSpeed(), scalar(0)));
85      // advection speed U +- C
86      const scalarField wUC(Foam::max(advectionSpeedWT(), scalar(0)));
87
88      // Calculate the field wave coefficient alpha with U and U+-C(See notes)
89      const scalarField alphaU(wU*deltaT*this->patch().deltaCoeffs());
90      const scalarField alphaUC(wUC*deltaT*this->patch().deltaCoeffs());
```

The updateCoeffs function is the core of the class.

The two advectionSpeed functions (identical to those from the advective and waveTransmissive classes) are used to calculate the advection speeds.

The two parameters alpha corresponding to the two advection speeds are calculated.

Like for the advective class, the core of the function corresponds to a series of if statements that fill the valueFraction_ and refValue_ scalar fields.

implementation of the LODI2D boundary condition (3)

Part of the updateCoeffs function of the LODI2DFvPatchField class

```
20 if (lInf_ > 0)
21     {
22         const scalarField k(wUC*deltaT/lInf_);
23         if
24         (
25             ddtScheme == fv::EulerDdtScheme<scalar>::typeName
26          || ddtScheme == fv::CrankNicolsonDdtScheme<scalar>::typeName
27         )
28         {
29             this->refValueU() = Utold_;
30
31             this->refValueUC() =
32             (
33                 Unold_ + k*UnInf_
34             )/(1.0 + k);
35
36             this->valueFractionU() = 1.0/(1.0 + alphaU);
37             this->valueFractionUC() = (1.0 + k)/(1.0 + alphaUC + k);
38         }
```

implementation of the LODI2D boundary condition (4)

When inserted in the expression of the functions of the `mixedV2D` class, this `refValue` and `valueFraction` data yield the following expressions for the normal and tangential velocity components

$$(u_n)_f^{n+1} = ((u_n)_f^n + k(u_n)^\infty)\frac{1}{1+\alpha_{uc}+k} + \frac{\alpha_{uc}}{1+\alpha_{uc}+k}(u_n)_c^{n+1},$$
$$(u_t)_f^{n+1} = (u_t)_f^n\frac{1}{1+\alpha_u} + \frac{\alpha_u}{1+\alpha_u}(u_t)_c^{n+1},$$

Meaning that **the normal component of the velocity is being transported with** $u_n + c$, and sees a partially non-reflecting boundary, while **the tangential component is transported with** u and sees a perfectly non-reflecting boundary.

Compiling the custom boundary conditions

For compiling the custom BCs place the `mixedV2D` and `LODI2D` folders inside the user's `src/finiteVolume` folder with the following structure:

```
finiteVolume
├── Make
│   ├── files
│   └── options
└── fields
    └── fvPatchFields
        ├── basic
        │   └── mixedV2D
        │       ├── mixedV2DFvPatchVectorField.H
        │       └── mixedV2DFvPatchVectorField.C
        └── derived
            └── LODI2D
                ├── LODI2D2DFvPatchVectorField.H
                └── LODI2D2DFvPatchVectorField.C
```

Introduction

Simulations are performed with **OpenFOAM native non-reflecting boundary conditions** and with the **custom boundary conditions**.

The difference between the custom `LODI2D` and the native `waveTransmissive` boundary conditions is substantial, however, many problems become numerically very similar since **in most practical cases the velocity is already normal to the outlet patches**, and the **tangential components tend to be very small**.

Moreover, the custom boundary conditions up to this point have only been implemented for a 2-D case on the $x - y$ plane, whilst turbulence and other flow phenomena are **intrinsically three-dimensional**.

For these reasons, defining a test case to assess properly the performance of the new boundary conditions requires more time and focus, and will be part of future work, together with an implementation of the full three-dimensional implementation of the LODI relations.

2-D circle simulation

The first test case is a simple 2-D circle with a diameter of 2 m and a temperature spark in the center that causes a series of pressure waves to travel towards the boundaries.

The simulation has been performed with **two different sets of BCs**, the OpenFOAM native boundary conditions and the custom `LODI2D` boundary conditions for the velocity

	Case 1	**Case 2**
p	waveTransmissive	waveTransmissive
U	waveTransmissive	LODI2D
T	advective	advective
lInf	10	10

The solver is `rhoPimpleFoam` for a time interval of $t = 0.01s$ and an initial $\Delta t = 2e + 6$.

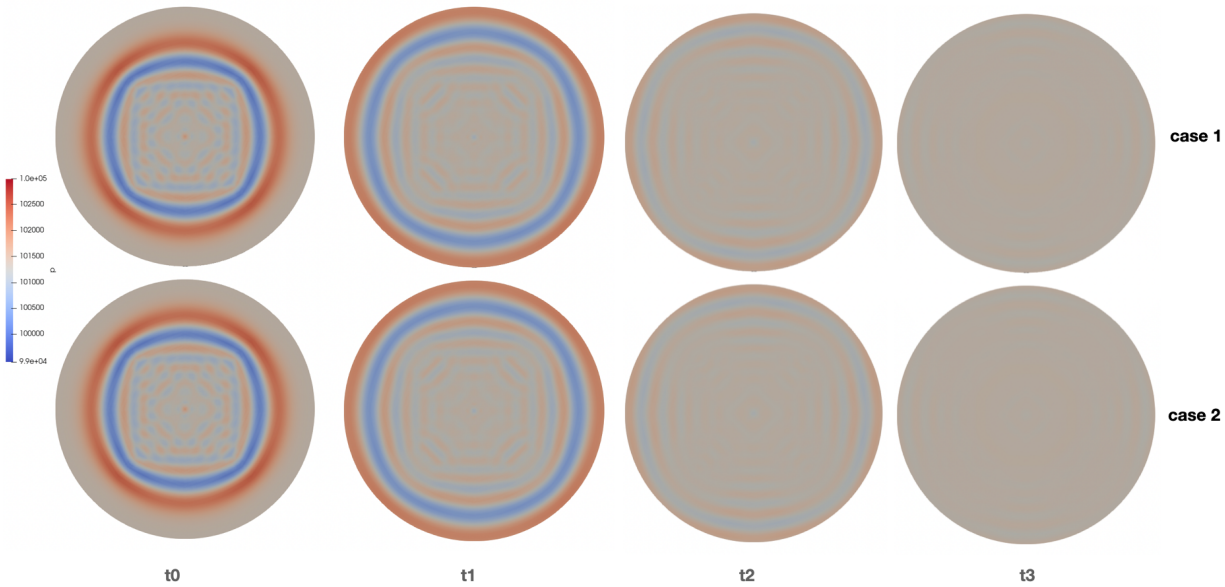

Figure: Propagation of pressure waves in the domain.

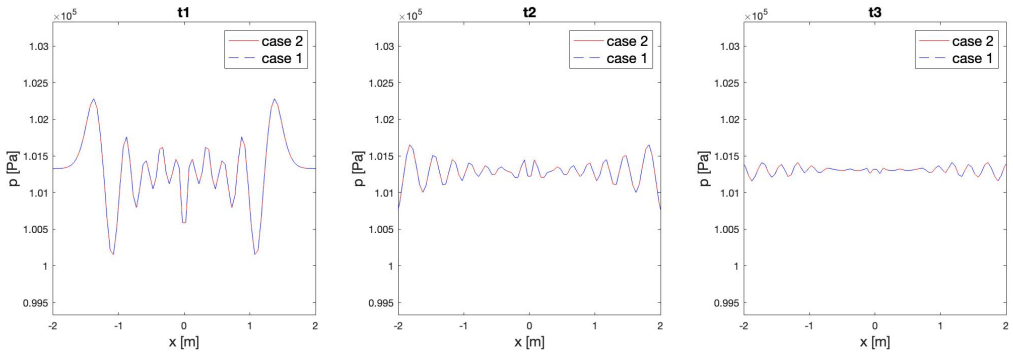

Figure: Propagation of pressure waves along the horizontal axis.

2-D circle simulation results

The pressure waves created by the spark travels towards the boundary and, for both cases, **exits the domain with no visible reflection.**

The two solutions are identical, since because of the geometry of the domain, the **velocity is always parallel to the outlet patches** therefore transporting both components of the velocity with advection speed $w = u + c$ becomes equivalent to rotating the velocity, transporting the normal component with $u + c$ and going back to the Cartesian reference frame, since the velocity component tangential to the patch is always null.

In order to further investigate the correct usage of OpenFOAM's native set of non-reflecting boundary conditions, the same simulation has been also performed with the setup implemented in all the tutorials,

2-D flow around bluff body

This test case consists of a **bluff body inside a laminar free-stream flow**.

The characteristic length of the body is $L = 0.1\ m$ while the length of the domain is $L_D = 115L$.

The overall computational cost of the simulation is kept small by applying a strong mesh refinement in correspondence of the body and its wake, which limits the overall cell count to 38200 cells.

The laminar flow past the square body is simulated at $M_\infty = 0.1$ to establish the effectiveness of numerical outflow boundary treatments for flows past bluff bodies.

Under these flow conditions, **vortices are periodically shed in the bluff body wake** and preventing spurious reflection of pressure waves from the outflow is challenging for numerical boundary conditions.

2-D flow around bluff body - results

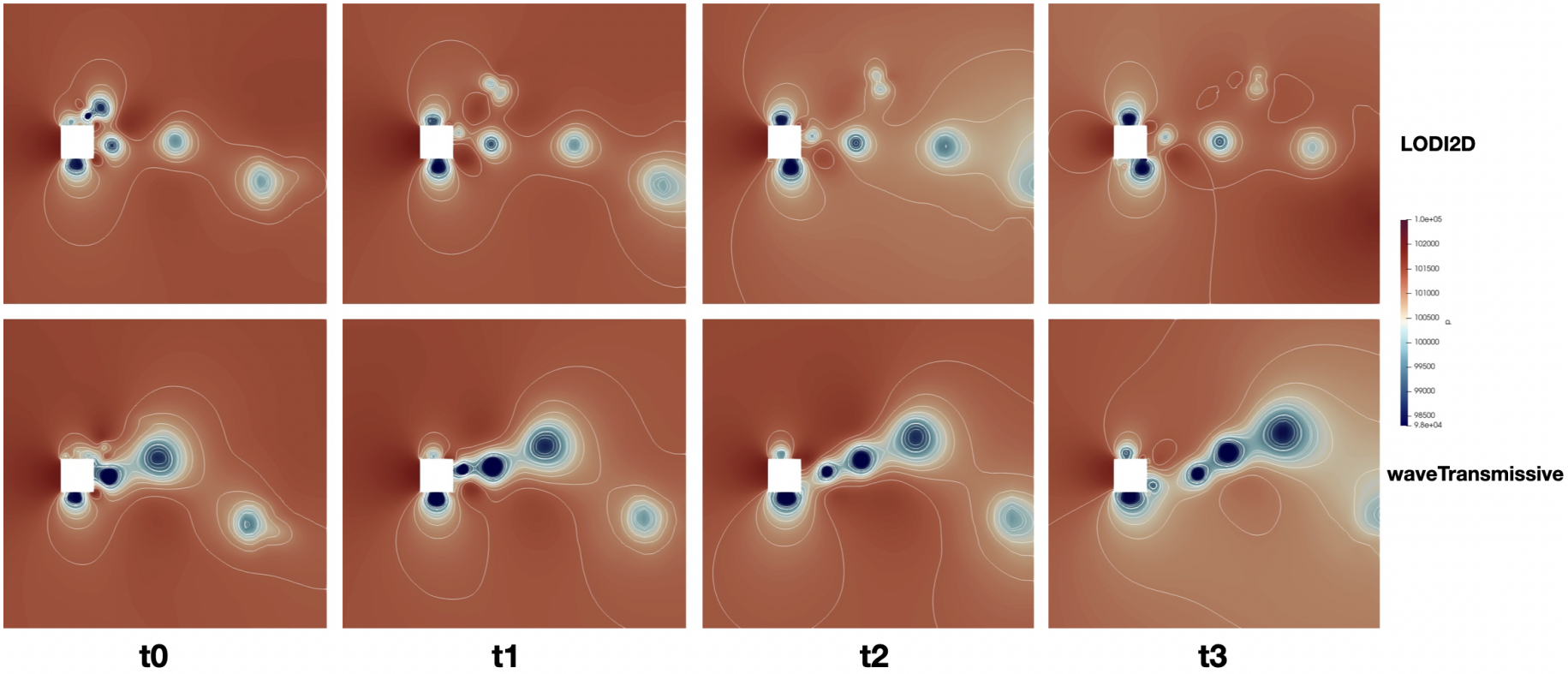

Figure: Pressure field of the 2-D bluff body simulation for four successive time steps for the `LODI2D` case (top) and the `waveTranmissive` case (bottom)

2-D flow around bluff body - results (2)

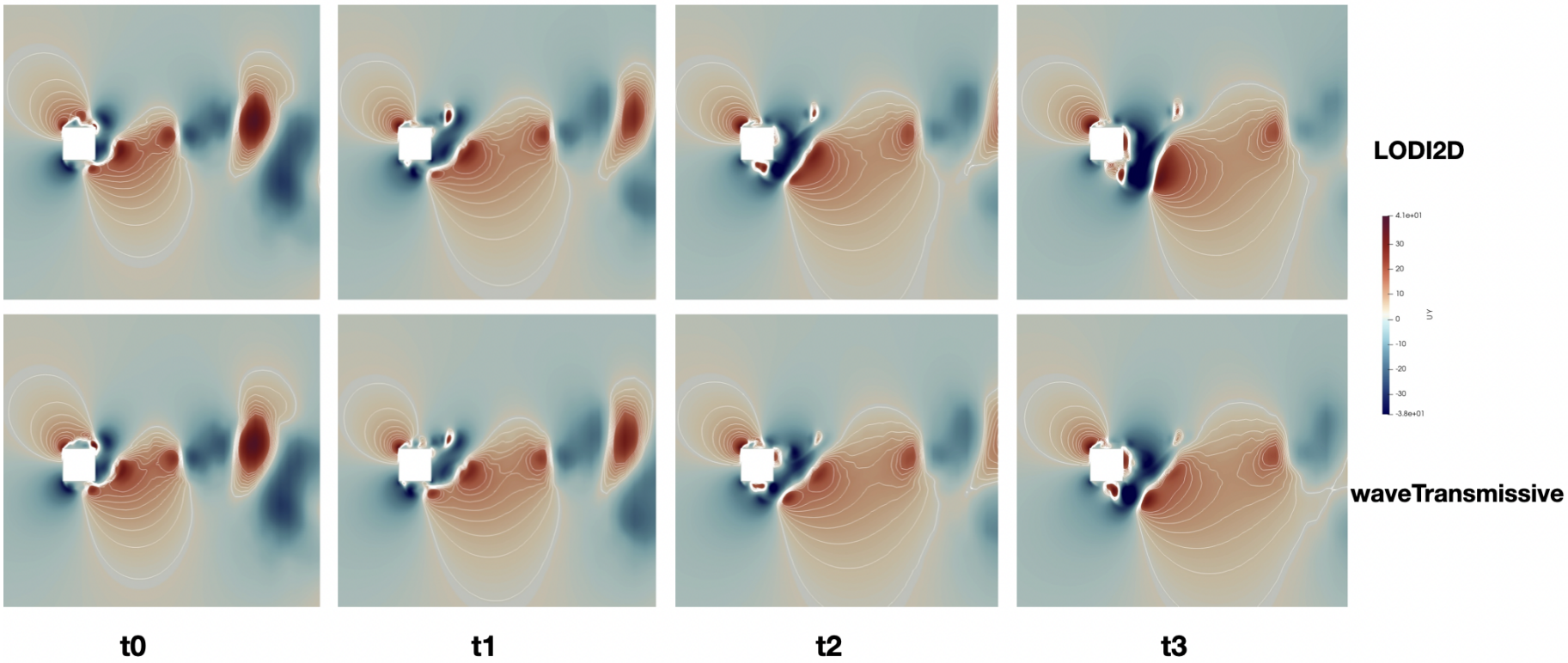

Figure: Uy field of the 2-D bluff body simulation for four successive time intervals, for the two cases. The iso contours are shown for Uy ranging from 1 to 15 m/s.

2-D flow around bluff body - results (3)

- $\bar{p'^2}$ represents the average of the product of the pressure fluctuations.
- It is a parameter that measures **the magnitude of the pressure fluctuations** in the field.
- In the `waveTransmissive` case, the pressure fluctuations seem to cover a larger area downstream of the bluff body.

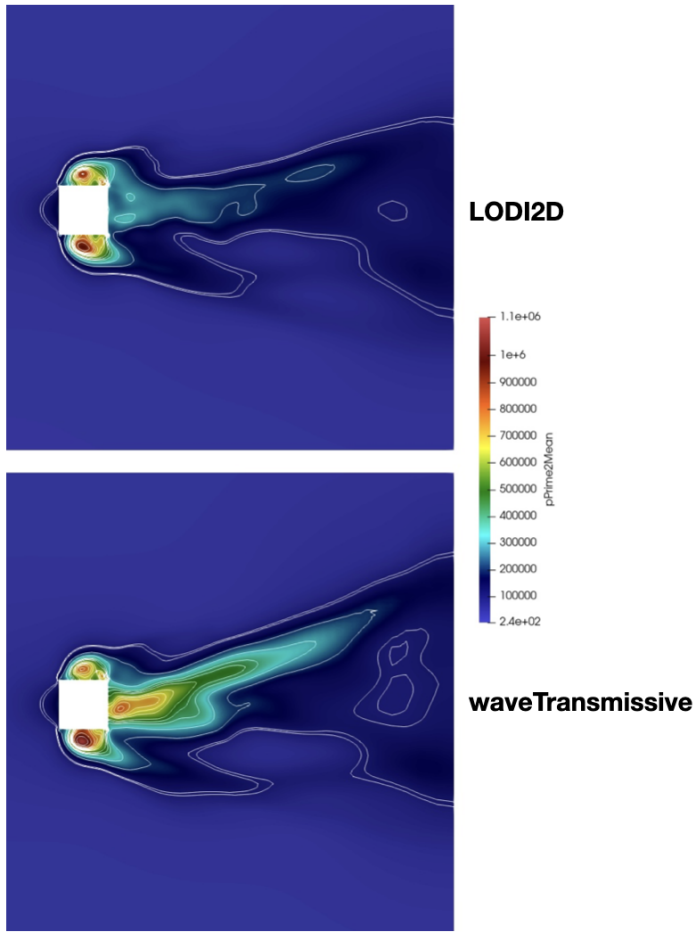

Figure: $\bar{p'^2}$ field of the 2-D bluff body simulation.

2-D flow around bluff body - results (3)

For both cases, **some minor pressure waves can be observed at the outlet**.

The $\overline{p'^2}$ field shows a **slightly larger pressure fluctuation zone downstream of the body** in the `waveTransmissive` case, this could indicate a better performance of the `LODI2D` boundary conditions.

It is worth noting that **for this particular geometry the outlet patch is perfectly parallel to the Cartesian reference frame** and the only difference between the two simulations is that in the `waveTransmissive` case, U_y at the boundary is transported with an advection speed $u + c$, while in the `LODI2D` case, it is transported with u.

Regardless of this, **the overall behavior of the two simulations is similar**, therefore further investigation is clearly required.

Conclusions

In conclusion:

- OpenFOAM's approach to non-reflecting boundary conditions consists of two classes: `advective` and `waveTransmissive`.
- These classes allow to transport the velocity vector with advection speed $u_n + c$, which theory shows that **should be applied only to the patch-normal component**.
- Two new classes: `mixedV2D` and `LODI2D` have been implemented in order to transport the velocity components with the correct advection speeds.
- Regardless of the difference between OpenFOAM's native boundary conditions and the theory, **their performance is already satisfactory for most applications**.
- However, **the way OpenFOAM's native boundary conditions have to be applied is different to the way they are applied in the tutorials**.
- Preliminary simulations have shown some slight improvement in terms of performance of the `LODI2D` boundary conditions, but firther investigation is required.

Goodbye

Thank you for your attention!