Description and modification of the waveTransmissive boundary condition; a partially reflecting 3D wave transmissive boundary condition

Björn Jarfors

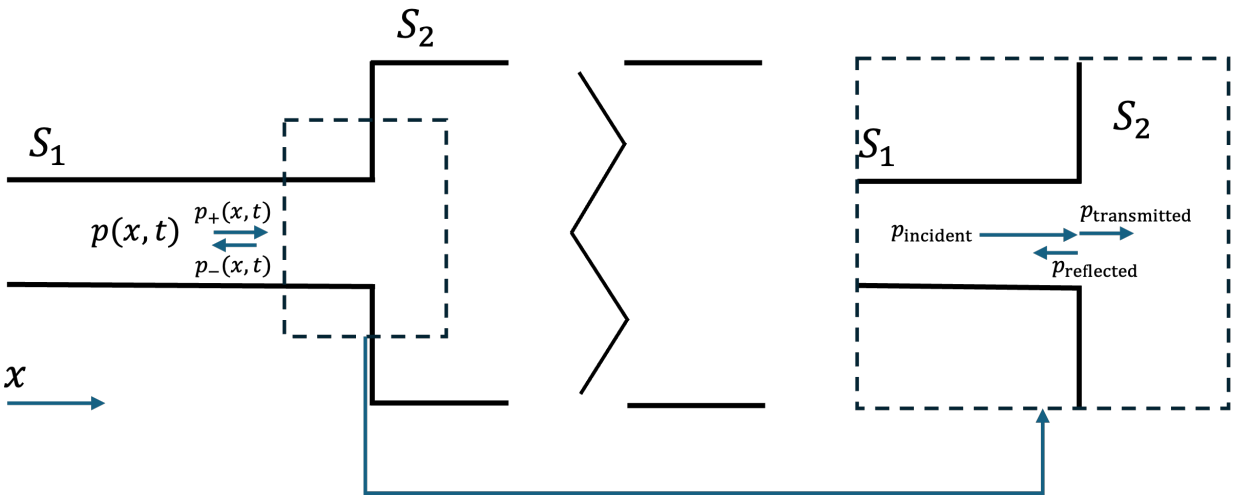
Energy Sciences/Heat Transfer,
Lunds Institute of Technology,
Lund, Sweden

January 19, 2025

Introduction

A simulation is generally performed on domains. These domains have boundaries. How do we define the boundaries?

Lets assume we have a confined channel with a cross-section jump. We can only afford to simulate the first cross-sections domain. What do we expect at the boundary between the two cross-sections?



Confined Channel with a Cross-Sectional Jump

Introduction

In Large Eddy Simulations (LES) or Direct Numerical Simulations (DNS), the numerical dissipation is greatly reduced, leading to propagation and survival of numerical waves

These numerical waves may originate at the boundaries. Therefore, well-posed boundary conditions are needed.

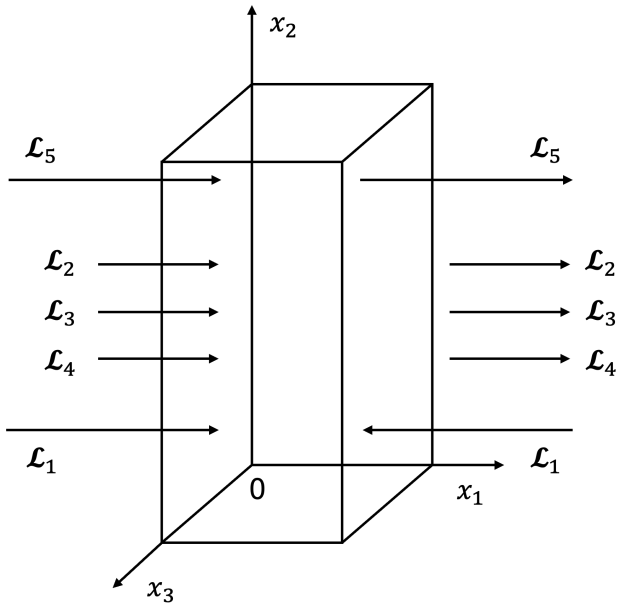
To make sure we have well-posed boundary conditions, a method to create boundary conditions was developed called Navier-Stokes Characteristic Boundary Conditions (NSCBC) which is often combined with Local One-Dimensional Inviscid (LODI) assumption.

This leads to obtain the so-called LODI-relations

LODI

The LODI-relations correspond to **Characteristic Waves** propagating in or out of the domain.

$\mathscr{L}_i$ corresponds to the information which propagates at a speed λ_i which corresponds to $\lambda_1 = u_1 - c$, $\lambda_2 = \lambda_3 = \lambda_4 = u_1$ and $\lambda_5 = u_1 + c$.



LODI

From the reactive compressible Navier-Stokes equations

$$\frac{\partial \rho}{\partial t} + \frac{\partial (\rho u_j)}{\partial x_j} = 0,$$

$$\frac{\partial (\rho u_i)}{\partial t} + \frac{\partial (\rho u_i u_j)}{\partial x_j} + \frac{\partial p}{\partial x_i} = \frac{\partial \tau_{ij}}{\partial x_j},$$

$$\frac{\partial \rho E}{\partial t} + \frac{\partial (\rho E + p) u_i}{\partial x_i} = \frac{\partial (u_i \tau_{ij})}{\partial x_i} - \frac{\partial q_i}{\partial x_i}, \quad (i = 1 \text{ to } 3),$$

Together with characteristic analysis to modify the hyperbolic terms we can obtain

$$\mathscr{L}_1 = \lambda_1 \left(\frac{\partial p}{\partial x_1} - \rho c \frac{\partial u_1}{\partial x_1} \right),$$

$$\mathscr{L}_2 = \lambda_2 \left(c^2 \frac{\partial \rho}{\partial x_1} - \frac{\partial p}{\partial x_1} \right),$$

$$\mathscr{L}_3 = \lambda_3 \frac{\partial u_2}{\partial x_1},$$

$$\mathscr{L}_4 = \lambda_4 \frac{\partial u_3}{\partial x_1},$$

$$\mathscr{L}_5 = \lambda_5 \left(\frac{\partial p}{\partial x_1} + \rho c \frac{\partial u_1}{\partial x_1} \right).$$

LODI

Recasting the LODI system in terms of primitive variables

$$\frac{\partial \rho}{\partial t} + \frac{1}{c^2}\left(\mathscr{L}_2 + \frac{1}{2}(\mathscr{L}_5 + \mathscr{L}_1)\right) = 0,$$

$$\frac{\partial p}{\partial t} + \frac{1}{2}(\mathscr{L}_5 + \mathscr{L}_1) = 0,$$

$$\frac{\partial u_1}{\partial t} + \frac{1}{2\rho c}(\mathscr{L}_5 - \mathscr{L}_1) = 0,$$

$$\frac{\partial u_2}{\partial t} + \mathscr{L}_3 = 0,$$

$$\frac{\partial u_3}{\partial t} + \mathscr{L}_4 = 0.$$

LODI

Or in terms of boundary normal gradients

$$\frac{\partial \rho}{\partial x_1} = \frac{1}{c^2}\left(\frac{\mathscr{L}_2}{u_1} + \frac{1}{2}\left(\frac{\mathscr{L}_5}{u_1 + c} + \frac{\mathscr{L}_1}{u_1 - c}\right)\right),$$

$$\frac{\partial p}{\partial x_1} = \frac{1}{2}\left(\frac{\mathscr{L}_5}{u_1 + c} + \frac{\mathscr{L}_1}{u_1 - c}\right),$$

$$\frac{\partial u_1}{\partial x_1} = \frac{1}{2\rho c}\left(\frac{\mathscr{L}_5}{u_1 + c} - \frac{\mathscr{L}_1}{u_1 - c}\right),$$

$$\frac{\partial T}{\partial x_1} = \frac{T}{\rho c^2}\left(-\frac{\mathscr{L}_2}{u_1} + \frac{1}{2}(\gamma - 1)\left(\frac{\mathscr{L}_5}{u_1 + c} + \frac{\mathscr{L}_1}{u_1 - c}\right)\right).$$

Deriving the current `waveTransmissive` Boundary Condition

Take these two equations

$$\frac{\partial p}{\partial t} + \frac{1}{2}(\mathscr{L}_5 + \mathscr{L}_1) = 0,$$

$$\frac{\partial p}{\partial x_1} = \frac{1}{2}\left(\frac{\mathscr{L}_5}{u_1 + c} + \frac{\mathscr{L}_1}{u_1 - c}\right),$$

By assuming $\mathscr{L}_1 = 0$ (Perfectly Non-Reflecting Condition) we can get

$$\frac{\partial p}{\partial t} + (u_1 + c)\frac{\partial p}{\partial x_1} = 0.$$

Now we should all ask, "is this a good boundary condition?". Usually we have mass-flow or velocity inlets with no pressure information. This condition, all information regarding pressure leaves the domain. Therefore, even if we say $\mathscr{L}_1 = 0$ in the derivation, we will add $\mathscr{L}_1 = K(p - p_\infty)$ to maintain mean pressure.

$$\frac{\partial p}{\partial t} + (u_1 + c)\frac{\partial p}{\partial x_1} + K(p - p_\infty) = 0.$$

Discretizing

Now we wish to discretize

$$\frac{\partial p}{\partial t} + (u_1 + c)\frac{\partial p}{\partial x_1} + K(p - p_\infty) = 0.$$

By setting $\alpha = u_1 \Delta t / \Delta x$ and $k = K \Delta t$ as well as defining a generic variable ϕ, we get

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

where $n + 1$ is the current time-step, n the previous time-step, and f and c correspond to the face- and cell-values, respectively.

In OpenFOAM $k = (u_1 + c)\Delta t / l_\infty$, where the user specifies l_∞. The user also specifies ϕ_∞. These correspond to the distance to the far-field and its value.

Partially Reflecting Boundary Condition

Now lets say we do not assume $\mathscr{L}_1 = 0$ (but we still assume $\mathscr{L}_1 = K(p - p_\infty)$) in the derivation of the boundary condition. Then we obtain

$$\frac{\partial p}{\partial t} + (u_1 + c)\frac{\partial p}{\partial x_1} + \frac{1}{2}\left[\left(1 - \frac{u_1 + c}{u_1 - c}\right)\right] K(p - p_\infty) = 0.$$

which has an extra coefficient $\frac{1}{2}\left(1 - \frac{u_1+c}{u_1-c}\right)$ which may be reformulated in terms of the Mach number as $\frac{1}{2}\left(1 - \frac{Ma+1}{Ma-1}\right)$. If we discretize this equation similarly to

the Perfectly Non-Reflecting formulation, we get

$$\phi_f^{n+1} = (\phi_f^n + Tk\phi_\infty)\frac{1}{1 + \alpha_{uc} + Tk} + \frac{\alpha_{uc}}{1 + \alpha_{uc} + Tk}\phi_c^{n+1},$$

where $T = \frac{1}{2}\left(1 - \frac{Ma+1}{Ma-1}\right)$.

OpenFOAM Boundary Conditions

In OpenFOAM we have fixed value and fixed gradient boundary conditions as well as a mix between the two.

Furthermore, a boundary condition can either explicitly or implicitly affect your solution. This is determined whether the boundary conditions contribution affects the **source term** (explicit) or the diagonal of the **coefficient matrix** (implicit). Furthermore, it can affect these terms in the linear solution of the divergence (fvm::div) or laplacian ter (fvm::laplacian).

`waveTransmissive` is a *mixed* boundary condition based on the `advective` boundary condition.

Both of these utilize a `mixed` class which is a general-purpose boundary condition which requires input for `valueFraction`, `refValue`, and `refGrad`.

The mixed Class

The member functions in the mixed class include

- valueInternalCoeffs implicit contribution to the divergence term,
- valueBoundaryCoeffs explicit contribution to the divergence term,
- gradientInternalCoeffs implicit contribution to the laplacian term,
- gradientBoundaryCoeffs explicit contribution to the laplacian term,

Listing 1: Definition of valueBoundaryCoeffs in mixed

```
template<class Type>
Foam::tmp<Foam::Field<Type>>
Foam::mixedFvPatchField<Type>::valueBoundaryCoeffs
(
    const tmp<scalarField>&
) const
{
    return
        valueFraction_*refValue_
      + (1.0 - valueFraction_)*refGrad_/this->patch().deltaCoeffs();
}
```

The `mixed` class

Other member functions to the `mixed` class include

- `evaluate`, and
- `snGrad`,

where `evaluate` checks if the coefficients (`valueFraction`, `refValue`, and `refGrad`) are updated, else it calls the `updateCoeffs` member function in the associated class (such as the `advective` class) to update the coefficients. If they are updated, it calculates

$$f\phi_{\text{ref}} + (1.0 - f)(\phi_c^{n+1} + \nabla\phi_{\text{ref}}\mathbf{d})^1,$$

which corresponds to ϕ_f^{n+1} and can be directly compared to

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

by setting $\nabla\phi_{\text{ref}}^n = 0$.

[1] Note that `this->patch().deltaCoeffs();` corresponds to $1/\mathbf{d}$

The advective class

As mentioned in the previous slide, `evaluate` calls on `updateCoeffs` in the `advective` class. This looks like

Listing 2: Definition of `updateCoeffs` in `advective`

```
template<class Type>
void Foam::advectiveFvPatchField<Type>::updateCoeffs()
{
    if (this->updated())
    {
        return;
    }

    const fvMesh& mesh = this->internalField().mesh();

    word ddtScheme
    (
        mesh.ddtScheme(this->internalField().name())
    );
    scalar deltaT = this->db().time().deltaTValue();
```

The advective class

Listing 3: Definition of updateCoeffs in advective

```
1   const GeometricField<Type, fvPatchField, volMesh>& field =
2       this->db().objectRegistry::template
3       lookupObject<GeometricField<Type, fvPatchField, volMesh>>
4       (
5           this->internalField().name()
6       );
7
8   // Calculate the advection speed of the field wave
9   // If the wave is incoming set the speed to 0.
10  const scalarField w(Foam::max(advectionSpeed(), scalar(0)));
11
12  // Calculate the field wave coefficient alpha (See notes)
13  const scalarField alpha(w*deltaT*this->patch().deltaCoeffs());
14
15  label patchi = this->patch().index();
```

where line 10 corresponds to $\max(U \cdot S_f, 0)$ and line 13 corresponds to $\alpha = u_1 \Delta t / \Delta x$.

The advective class

Listing 4: Definition of updateCoeffs in advective

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

where we see $k = (U \cdot S_f)\Delta t / l_\infty$ on line 4, and the definitions for $\phi_{\text{ref}} = (\phi_f^n + k\phi_\infty)/(1.0 + k)$ and $f = (1.0 + k)/(1.0 + \alpha + k)$ on lines 10 and 14.

The advective class

Listing 5: Definition of advectionSpeed in advective

```
1  template<class Type>
2  Foam::tmp<Foam::scalarField>
3  Foam::advectiveFvPatchField<Type>::advectionSpeed() const
4  {
5      const surfaceScalarField& phi =
6          this->db().objectRegistry::template lookupObject<surfaceScalarField>
7          (phiName_);
8      // Look up the patch field of phiName_
9      // Check dimension of phi and return advectionSpeed based on mass flow or
       velocity. (omitted from presentation)
10     {
11         return phip/this->patch().magSf();
12     }
13 }
```

where we see $w = (U \cdot S_f)$ on line 11.

The waveTransmissive class

Listing 6: Definition of advectionSpeed in waveTransmissive

```
1  template<class Type>
2  Foam::tmp<Foam::scalarField>
3  Foam::waveTransmissiveFvPatchField<Type>::advectionSpeed() const
4  {
5      // Lookup the velocity and compressibility of the patch
6      const fvPatchField<scalar>& psip =
7          this->patch().template
8              lookupPatchField<volScalarField, scalar>(psiName_);
9
10     const surfaceScalarField& phi =
11         this->db().template lookupObject<surfaceScalarField>(this->phiName_);
12     // Look up the patch field of phiName_
13     // Check dimension of phi and divide psi by the density if it is based on
       mass flow, else leave be. (omitted from presentation)
14
15     return phip/this->patch().magSf() + sqrt(gamma_/psip);
16 }
```

where we see $w = (U \cdot S_f + \sqrt{\gamma/\psi})$ on line 15.

So what are we solving now?

To summarize, we obtain `valueFraction`, `refValue` and `refGrad` from `advective` and `waveTransmissive` class through

- $f = (1.0 + k)/(1.0 + \alpha + k)$
- $\phi_{\text{ref}} = (\phi_f^n + k\phi_\infty)/(1.0 + k)$
- $\nabla\phi_{\text{ref}} = 0$

The `evaluate` member function in `mixed` class determines if these are up to date, if they are, it returns ϕ_f^{n+1} based on

$$\phi_f^{n+1} = \phi_{\text{ref}}^n f + (1.0 - f)(\phi_c^{n+1} + \nabla\phi_{\text{ref}}^n \mathbf{d}).$$

The `mixedV2D` and LODI2D classes

Leandro Lucchese (a previous student in the OFCFD course) created a 2D-wave-transmissive boundary condition called LODI2D which includes the general-purpose mixedV2D class.

A few initial differences include

- merging of the `advective` and `waveTransmissive` classes into a `LODI2D` class,
- specifying that it is only for the purpose of a velocity vector, thus the templated property of the previous classes are removed, and
- specifying two different α such that the velocity components corresponding to $\mathscr{L}_1$ have $\lambda_1(\Delta t/\Delta x)$ and similarly, $\mathscr{L}_3$ and $\mathscr{L}_4$ have $\lambda_3(\Delta t/\Delta x)$ and $\lambda_4(\Delta t/\Delta x)$, respectively.

The latter components are thus advected out of the domain with an advection speed of $w = (U \cdot S_f)$. Since this is a 2D formulation, only one of $\mathscr{L}_3$ or $\mathscr{L}_4$ are considered.

The mixedV2D and LODI2D classes

This can be seen in

Listing 7: Some new additions to updateCoeffs in LODI2D

```
1     const scalarField wU(Foam::max(advectionSpeed(), scalar(0)));
2     const scalarField wUC(Foam::max(advectionSpeedWT(), scalar(0)));
3
4     const scalarField alphaU(wU*deltaT*this->patch().deltaCoeffs());
5     const scalarField alphaUC(wUC*deltaT*this->patch().deltaCoeffs());
```

The mixedV2D class

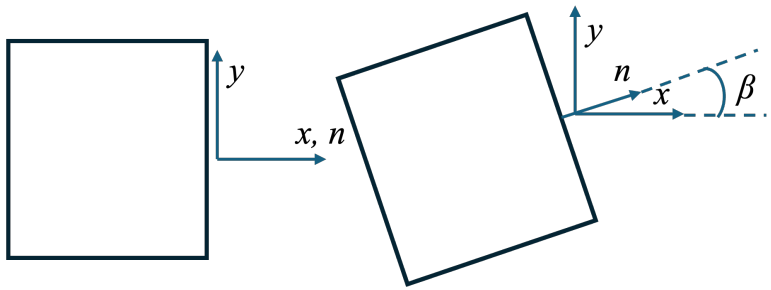
Listing 8: Constructor in mixedV2D

```
Foam::mixedV2DFvPatchVectorField::mixedV2DFvPatchVectorField
(
    const fvPatch& p,
    const DimensionedField<vector, volMesh>& iF
)
:
    fvPatchVectorField(p, iF),
    Un_(p.size()),
    Ut_(p.size()),
    n_(p.size()),
    refValueU_(p.size()),
    refValueUC_(p.size()),
    refGrad_(p.size()),
    valueFractionU_(p.size()),
    valueFractionUC_(p.size()),
    source_(p.size(), Zero),
    vector1_(p.size()),
    vector2_(p.size()),
    vector3_(p.size())
```

The mixedV2D class

Listing 9: Constructor in mixedV2D

```
1        n_ = this->patch().nf();
2        forAll(vector1_, i)
3        {
4            vector1_[i][0] = n_[i][0];
5            vector1_[i][1] = n_[i][1];
6            vector1_[i][2] = n_[i][2];
7        }
8        forAll(vector2_, i)
9        {
10           vector2_[i][0] = -n_[i][1];
11           vector2_[i][1] = n_[i][0];
12           vector2_[i][2] = n_[i][2];
13       }
14       forAll(vector3_, i)
15       {
16           vector3_[i][0] = 0.0;
17           vector3_[i][1] = 0.0;
18           vector3_[i][2] = 1.0;
19       }
20
```



The mixedV2D class

In the member functions; evaluate and snGrad, there exists a projection to the normal and transverse directions

Listing 10: Constructor in mixedV2D

```
1    forAll(U, i)
2    {
3        Un[i] = U[i][0]*n[i][0] + U[i][1]*n[i][1]; //ucos+vsin
4        Ut[i] = -U[i][0]*n[i][1] + U[i][1]*n[i][0]; //-usin+vcos
5    }
```

which correspond to the rotation transformation matrix.

The old and oold time-steps were populated in an identical manner in the LODI2D class.

To simplify the implementation of a 3D wave-transmissive boundary condition, the generalization the rotation brings is removed due to the inclusion of pitch, yaw and roll rotations needed, instead of simply only one of these (2D).

The modifiedMixedV2D class

Listing 11: Constructor in modifiedMixedV2D

```
Foam::modifiedLODI2DFvPatchVectorField::modifiedLODI2DFvPatchVectorField
(
    const fvPatch& p,
    const DimensionedField<vector, volMesh>& iF
)
:
    modifiedMixedV2DFvPatchVectorField(p, iF),
    phiName_("phi"),
    rhoName_("rho"),
    fieldInf_(Zero),
    lInf_(-GREAT),
    psiName_("thermo:psi"),
    gamma_(0.0)
{
    this->refValueU() = Zero;
    this->refValueUC() = Zero;
    this->refGrad() = Zero;
    this->valueFractionU() = 0.0;
    this->valueFractionUC() = 0.0;
}
```

The `modifiedMixedV2D` class

Notice that no unit vector definitions were needed now for the rotation and the velocity vector is simply obtained through

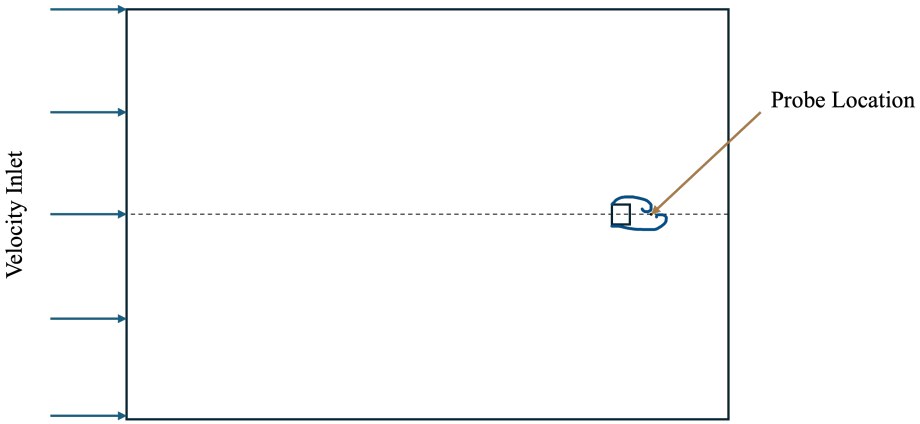
- `vectorField U = this->patchInternalField();`.

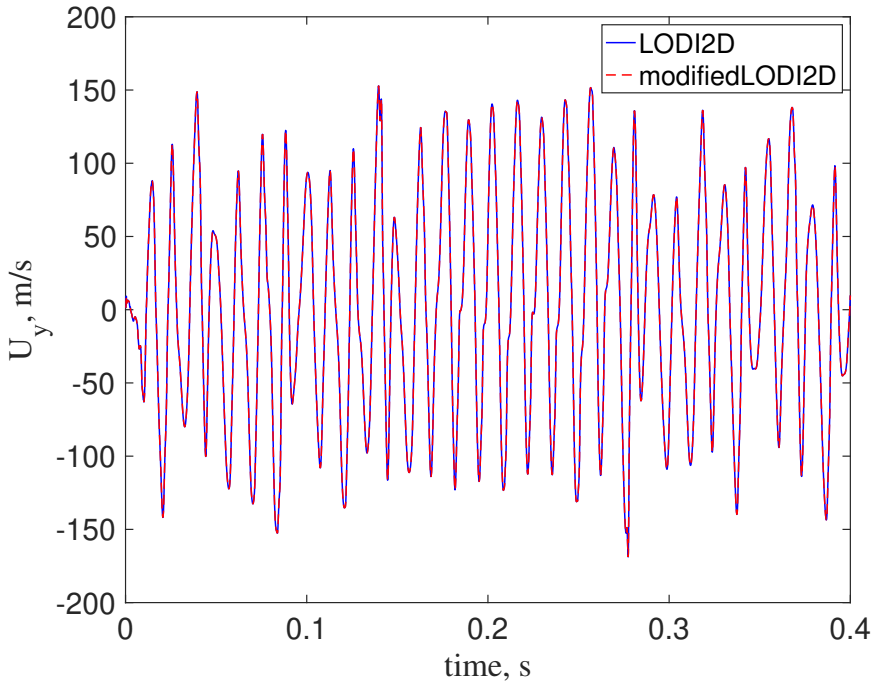
Similarly, the old and oold values of U is obtained through

- `vectorField Uold = field.oldTime().boundaryField()[patchi];`, and
- `vectorField Uoold = field.oldTime().oldTime().boundaryField()[patchi];`

Validation of the simplification

Take this unconfined bluff-body case, and we will track the vortex-shedding based on the probe data of the y-velocity component just behind the bluff body.





Which indicates a complete overlap in the time-series.

Expanding to 3D

Since we have an extra component to transport, we need to include an extra `valueFraction` and `refValue` corresponding to the third components conditions[2].
This results in the change in the `evaluate` member function in `modifiedMixedV3D` to include an extra component in its return function

Definition of `evaluate` in `modifiedMixedV3D`

```
1  void Foam::modifiedMixedV3DFvPatchVectorField::evaluate(const Pstream::
       commsTypes)
2  {
3      // Calculate valueU1
4      // Calculate valueU2
5      // Calculate valueUC
6      vectorField::operator=
7      (
8          valueUC*vector(1.0, 0.0, 0.0) + valueU1*vector(0.0, 1.0, 0.0) + valueU2*
       vector(0.0, 0.0, 1.0)
9      );
10     fvPatchVectorField::evaluate();
11 }
```

[2] `valueFraction` is identical for transverse components, see definition of α and k for this.

The modifiedMixedV3D class

modifiedMixedV3D

```
Foam::tmp<Foam::vectorField>
Foam::modifiedMixedV3DFvPatchVectorField
    ::valueBoundaryCoeffs
(
    const tmp<scalarField>&
) const
{
    scalarField valueU1 =
        valueFractionU1_*refValueU1_;
    scalarField valueU2 =
        valueFractionU2_*refValueU2_;
    scalarField valueUC =
        valueFractionUC_*refValueUC_;
    return valueUC*vector(1.0, 0.0, 0.0)
      + valueU1*vector(0.0, 1.0, 0.0) +
     valueU2*vector(0.0, 0.0, 1.0);
}
```

Listing 12: mixed

```
template<class Type>
Foam::tmp<Foam::Field<Type>>
Foam::mixedFvPatchField<Type>::
    valueBoundaryCoeffs
(
    const tmp<scalarField>&
) const
{
    return
        valueFraction_*refValue_
      + (1.0 - valueFraction_)*
     refGrad_/this->patch().
     deltaCoeffs();
}
```

The modifiedMixedV3D and modifiedLODI3D classes

Similarly, extra components are used for the rest of the member functions which handles calculations regarding the velocity components. This includes the updateCoeffs in modifiedLODI3D which is

Listing 13: modifiedLODI3D

```
    if (lInf_ > 0)
    {
    // Omitted code which remains unchanged
        {
            this->refValueU1() = Uold.component(1);
            this->refValueU2() = Uold.component(2);
            this->refValueUC() =
            (
                Uold.component(0) + k*fieldInf_.component(0)
            )/(1.0 + k);
            this->valueFractionU1() = 1.0/(1.0 + alphaU);
            this->valueFractionU2() = 1.0/(1.0 + alphaU);
            this->valueFractionUC() = (1.0 + k)/(1.0 + alphaUC + k);
        }
```

Partially reflecting boundary condition

To recap on what is needed to obtain the partially reflecting boundary condition, we recall the discretized equation we have gone through so far as

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

and when compared to the discretized partially reflecting equation for ϕ_f^{n+1}

$$\phi_f^{n+1} = (\phi_f^n + Tk\phi_\infty)\frac{1}{1+\alpha_{uc}+Tk} + \frac{\alpha_{uc}}{1+\alpha_{uc}+Tk}\phi_c^{n+1},$$

where $T = \frac{1}{2}\left(1 - \frac{Ma+1}{Ma-1}\right)$.
Observe that this was for pressure, and for velocity there is a phase change resulting in a term $\frac{1}{2}\left(\frac{Ma+1}{Ma-1} - 1\right)$ which is equal to $-T$ and lets refer to this as R. Thus we get

$$\phi_f^{n+1} = (\phi_f^n + Rk\phi_\infty)\frac{1}{1+\alpha_{uc}+Rk} + \frac{\alpha_{uc}}{1+\alpha_{uc}+Rk}\phi_c^{n+1},$$

Partially reflecting boundary condition

To implement this in our existing `modifiedLODI3D` code, we identify that $R = \frac{1}{2}\left(\frac{u_1+c}{u_1-c} - 1\right)$ where `wUC` corresponds to the numerator, and we need to define a function to obtain the denominator. This is obtained simply by including

Listing 14: `modifiedLODI3D, advectionSpeedWT2`

```
Foam::tmp<Foam::scalarField>
Foam::modifiedLODI3DPRFvPatchVectorField::advectionSpeedWT2() const
{
    // Lookup the velocity and compressibility of the patch
    // Check dimensions and adjust if needed
    return phip/this->patch().magSf() - sqrt(gamma_/psip); // U - C
}
```

and define

Listing 15: `modifiedLODI3D, updateCoeffs`

```
    const scalarField wUC2(Foam::min(advectionSpeedWT2(), scalar(0)));
    // partially reflecting factor
    const scalarField PR(0.5*(wUC1/wUC2 - 1.0));
```

Partially reflecting boundary condition

Observe that `wUC` is defined as the $\min(w_-, 0)$ instead of the maximum, since we expect a negative value for subsonic flows. Furthermore, `PR` corresponds to R. This is later implemented where `valueFraction` and `refValue` are updated

Listing 16: `modifiedLODI3D, updateCoeffs`

```
    if (lInf_ > 0)
    {
        const scalarField k(wUC1*deltaT/lInf_);
        if
        (
            ddtScheme == fv::EulerDdtScheme<scalar>::typeName
         || ddtScheme == fv::CrankNicolsonDdtScheme<scalar>::typeName
        )
        {
            this->refValueU1() = Uold.component(1);
            this->refValueU2() = Uold.component(2);
            this->refValueUC() =
            (
                Uold.component(0) + PR*k*fieldInf_.component(0)
            )/(1.0 + PR*k);
```

Compiling

The tree structure is

```
Make
        fields
            fvPatchFields
fields
    fvPatchFields
        basic
            modifiedMixedV2D
            modifiedMixedV3D
        derived
            modifiedLODI2D
            modifiedLODI3D
            modifiedLODI3DPR
lnInclude
```

So in summary we have

- modifiedMixedV2D,
- modifiedMixedV3D,
- modifiedLODI2D,
- modifiedLODI3D, and
- modifiedLODI3DPR.

The structure is kept the same as in OpenFOAM, following the `basic` and `derived` boundary conditions. Place these in `$WM_PROJECT_USER_DIR/src/finiteVolume`.

Compiling

The `files` file

Listing 17: The `files` file

```
fvPatchFields = fields/fvPatchFields
derivedFvPatchFields = $(fvPatchFields)/derived
basicFvPatchFields = $(fvPatchFields)/basic

$(basicFvPatchFields)/modifiedMixedV2D/modifiedMixedV2DFvPatchVectorField.C
$(derivedFvPatchFields)/modifiedLODI2D/modifiedLODI2DFvPatchVectorField.C
$(basicFvPatchFields)/modifiedMixedV3D/modifiedMixedV3DFvPatchVectorField.C
$(derivedFvPatchFields)/modifiedLODI3D/modifiedLODI3DFvPatchVectorField.C
$(derivedFvPatchFields)/modifiedLODI3DPR/modifiedLODI3DPRFvPatchVectorField.C
LIB = $(FOAM_USER_LIBBIN)/libmyFiniteVolume
```

Compiling

The `options` file

Listing 18: The `options` file

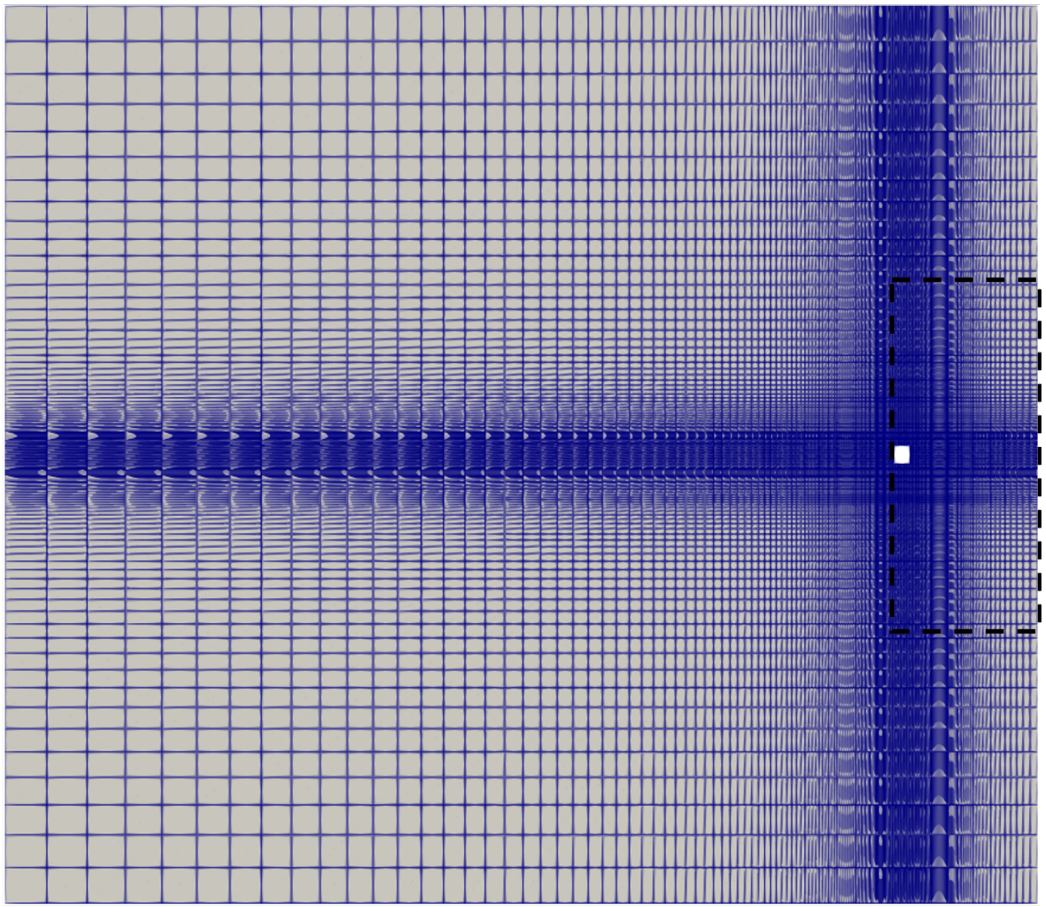
```
EXE_INC = \
    -I$(LIB_SRC)/fileFormats/lnInclude \
    -I$(LIB_SRC)/surfMesh/lnInclude \
    -I$(LIB_SRC)/meshTools/lnInclude \
    -I$(LIB_SRC)/dynamicMesh/lnInclude \
    -I$(LIB_SRC)/finiteVolume/lnInclude

LIB_LIBS = \
    -lOpenFOAM \
    -lfileFormats \
    -lmeshTools \
    -lfiniteVolume
```

Case Description

The test-case consists of an unconfined bluff body as previously introduced. The mesh is

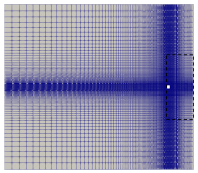
$U = (120\ 0\ 0)$ m/s
T – Zero Gradient
p – Zero Gradient



U – Case Specific
T – Advective
p – waveTransmissive

U – freeStreamVelocity
T – Zero Gradient
p – freeStreamPressure

Case Description

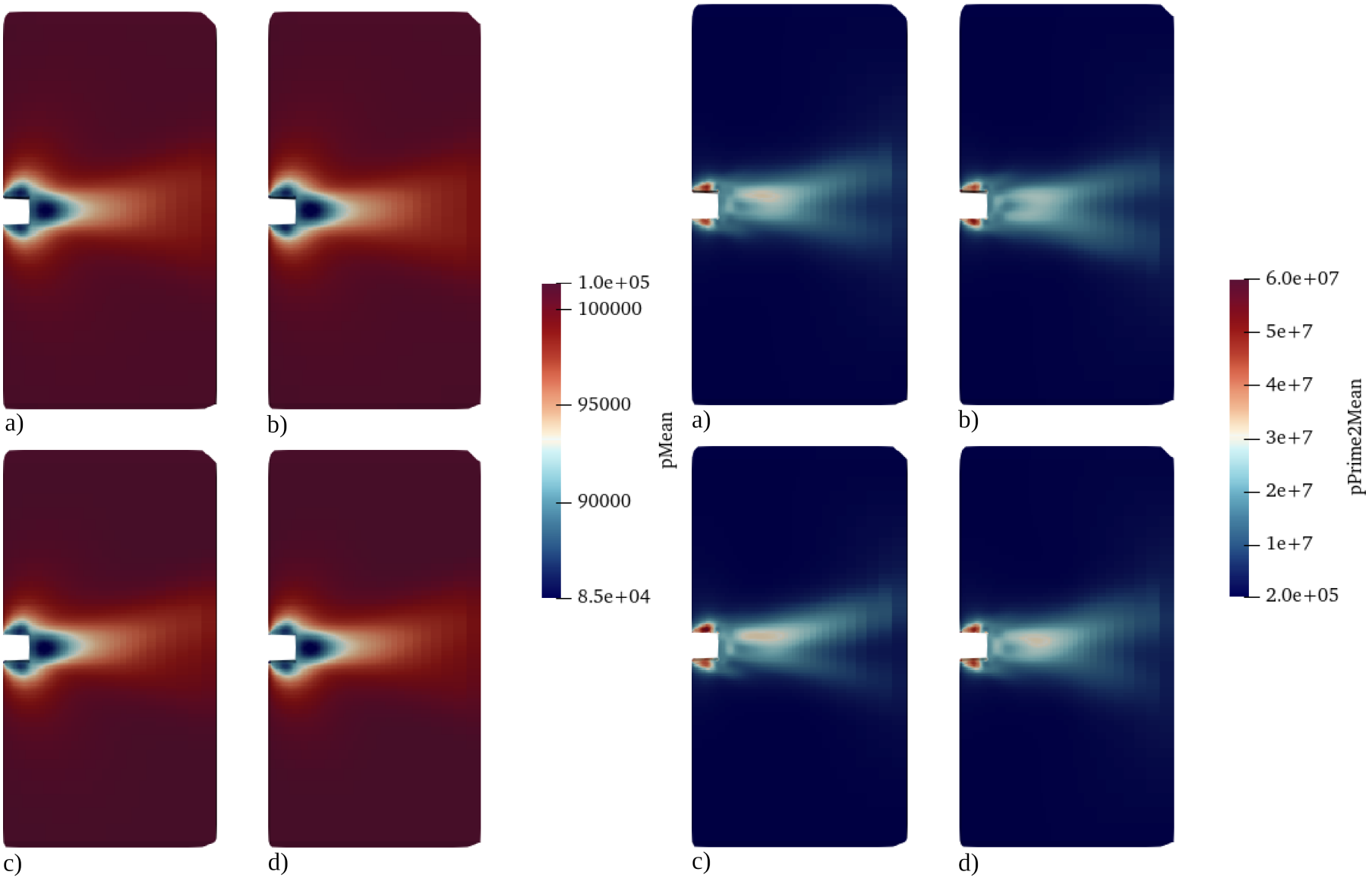


	Case 1	**Case 2**	**Case 3**	**Case 4**
p	wT	wT	wT	wT
U	wT	`LODI2D`	`LODI3D`	`LODI3DPR`
T	`advective`	`advective`	`advective`	`advective`
`lInf`	10	10	10	10

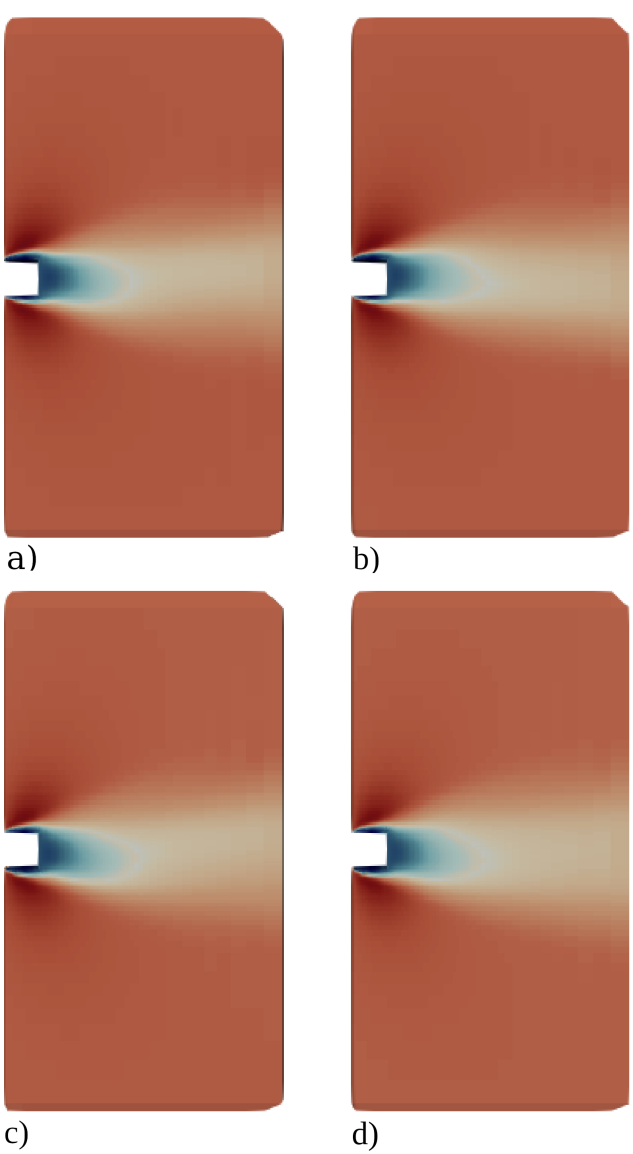
wT - `waveTransmissive`

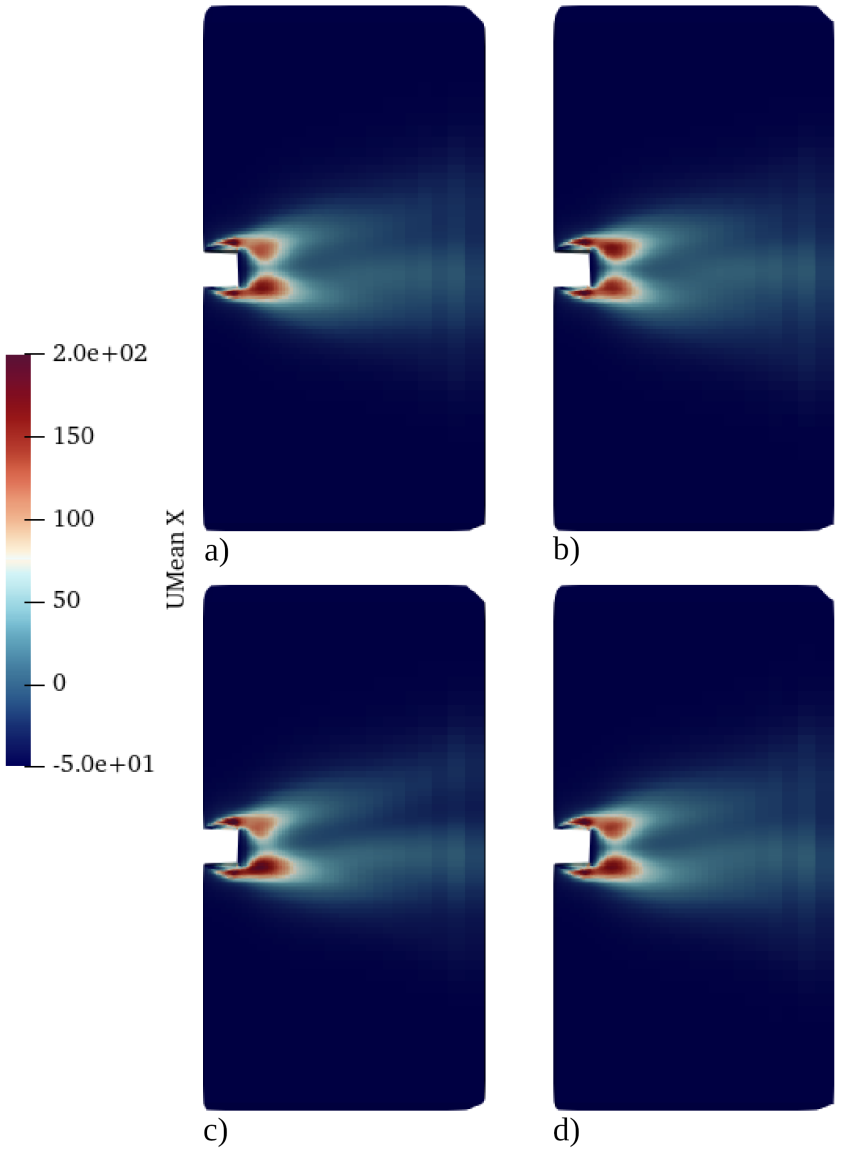
- `rhoPimpleFoam`
- `maxCo` 0.3
- 200 000 cells (10 cells in the spanwise direction)
- $Re \approx 40000$ (based on bluff-body size)

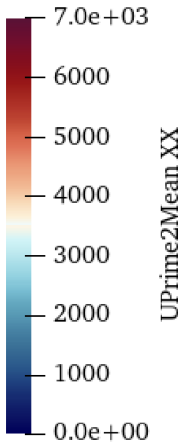
Simulation Results - Pressure



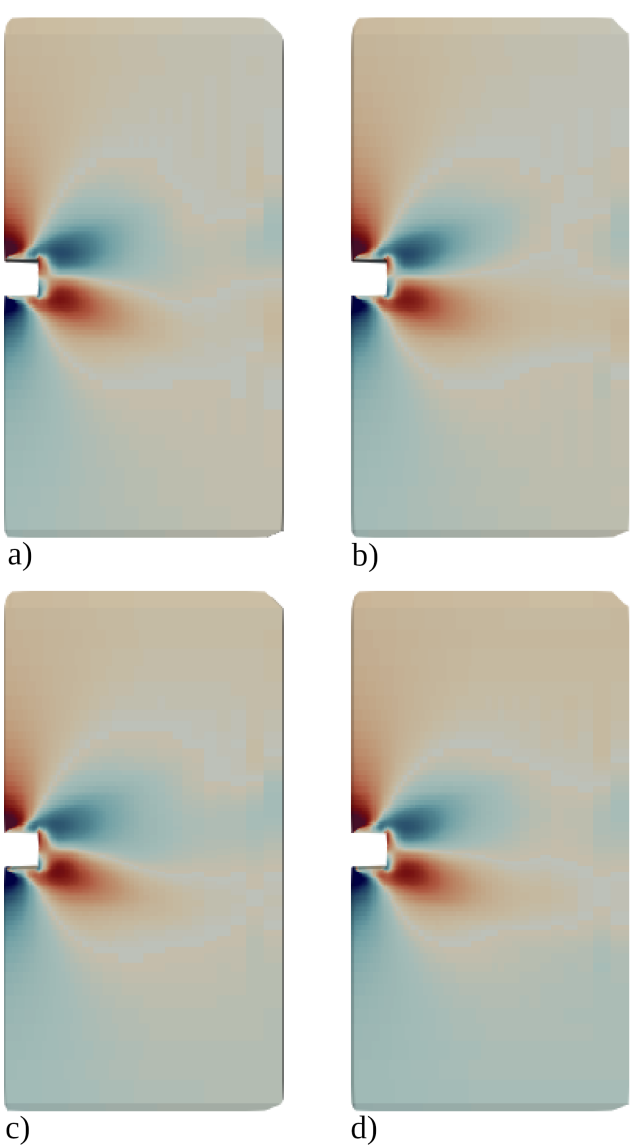
Simulation Results - x Velocity

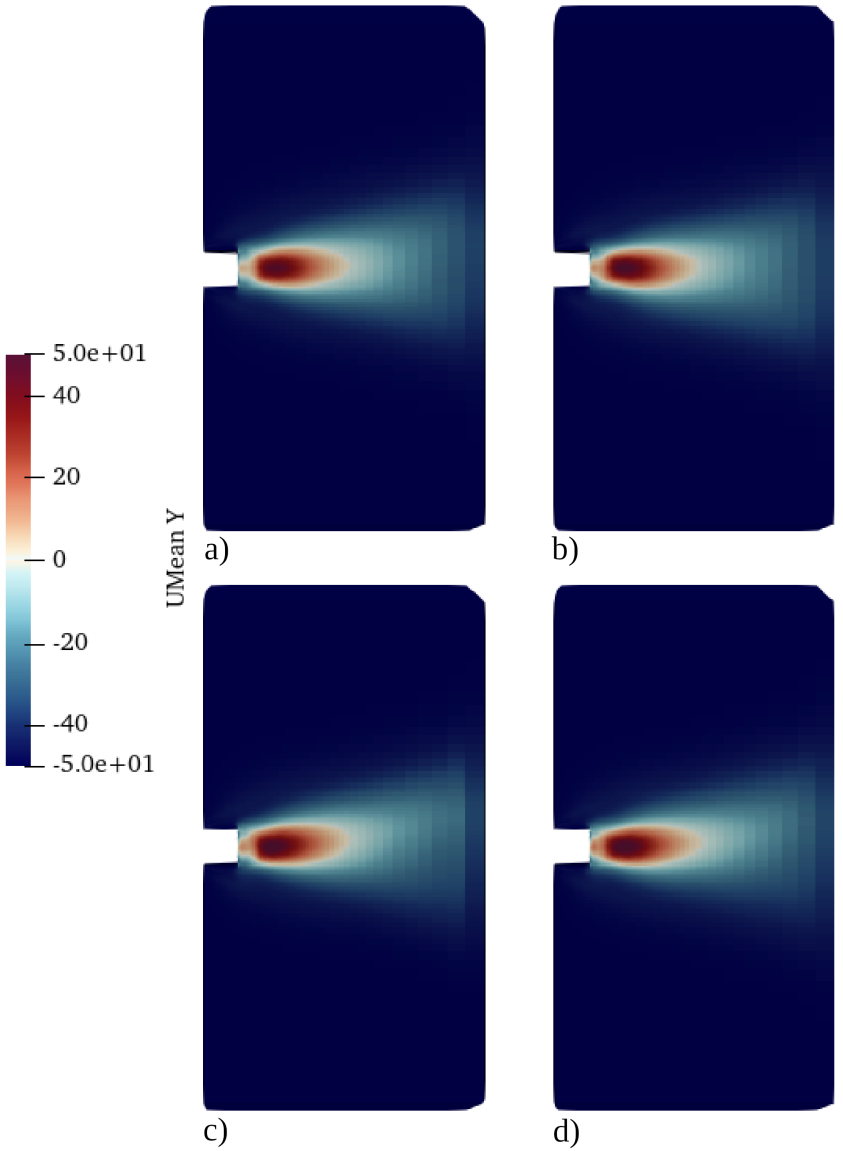




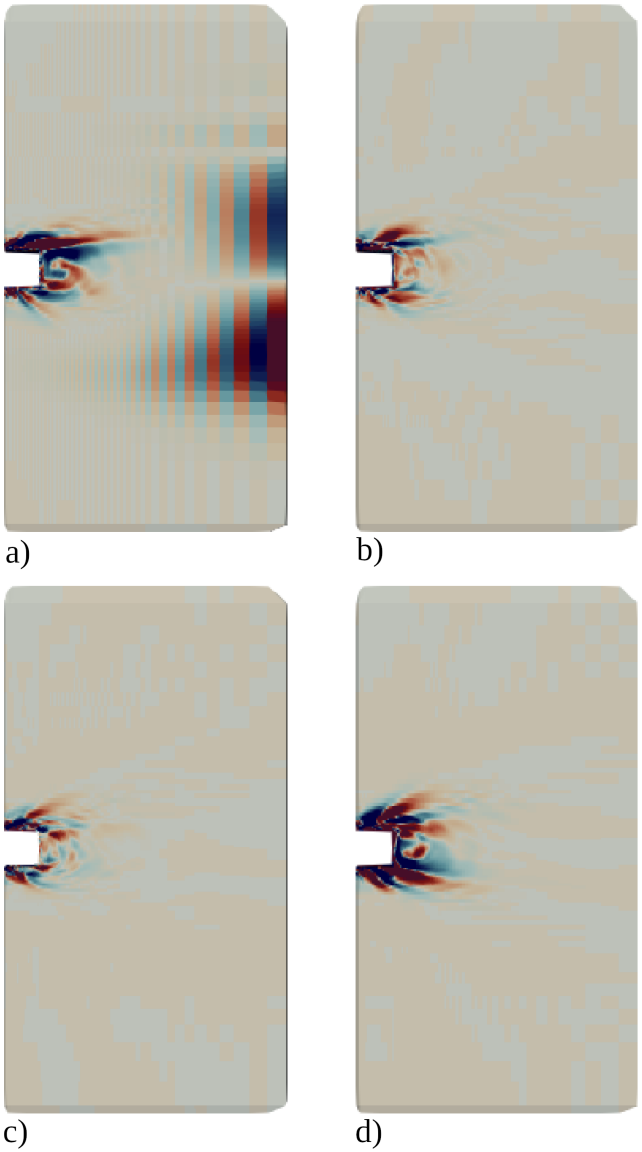


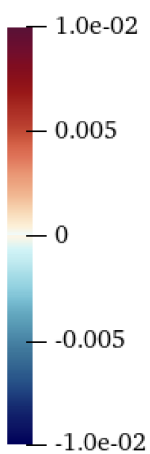
Simulation Results - y Velocity

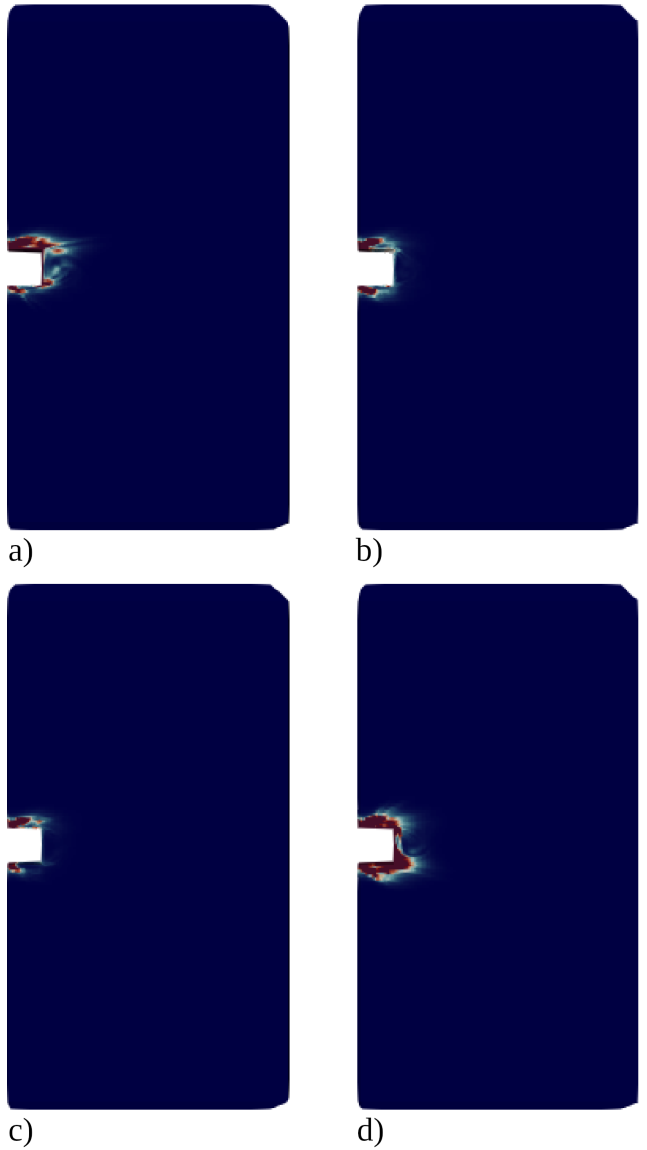


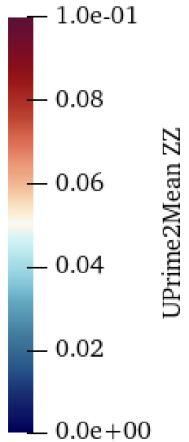


Simulation Results - z Velocity

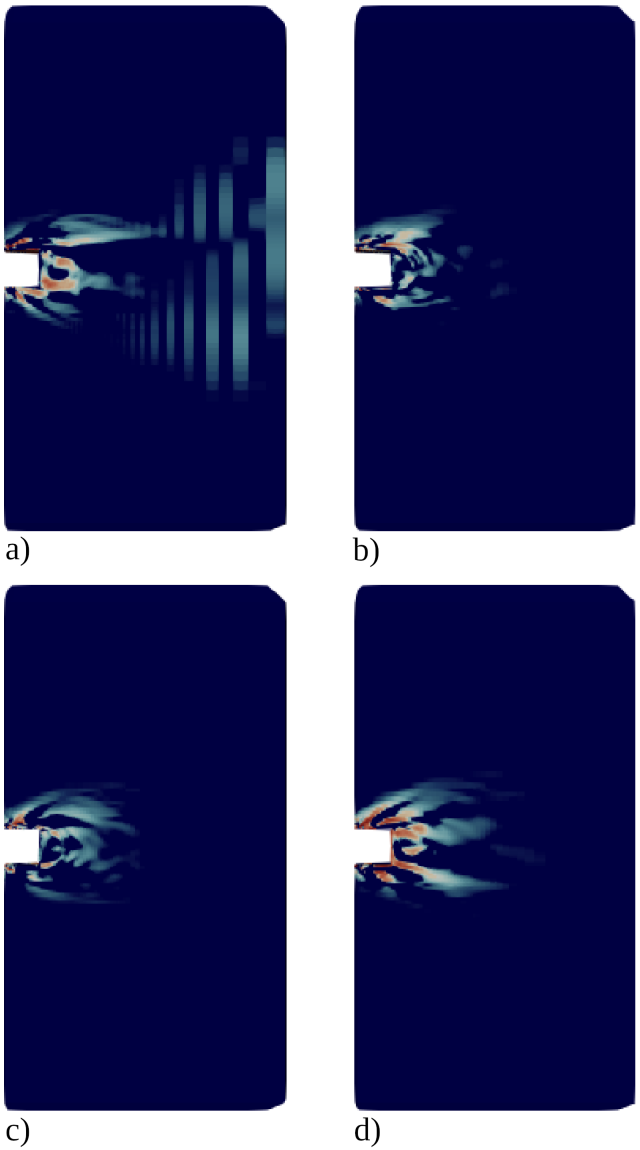


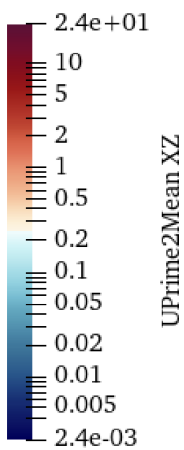


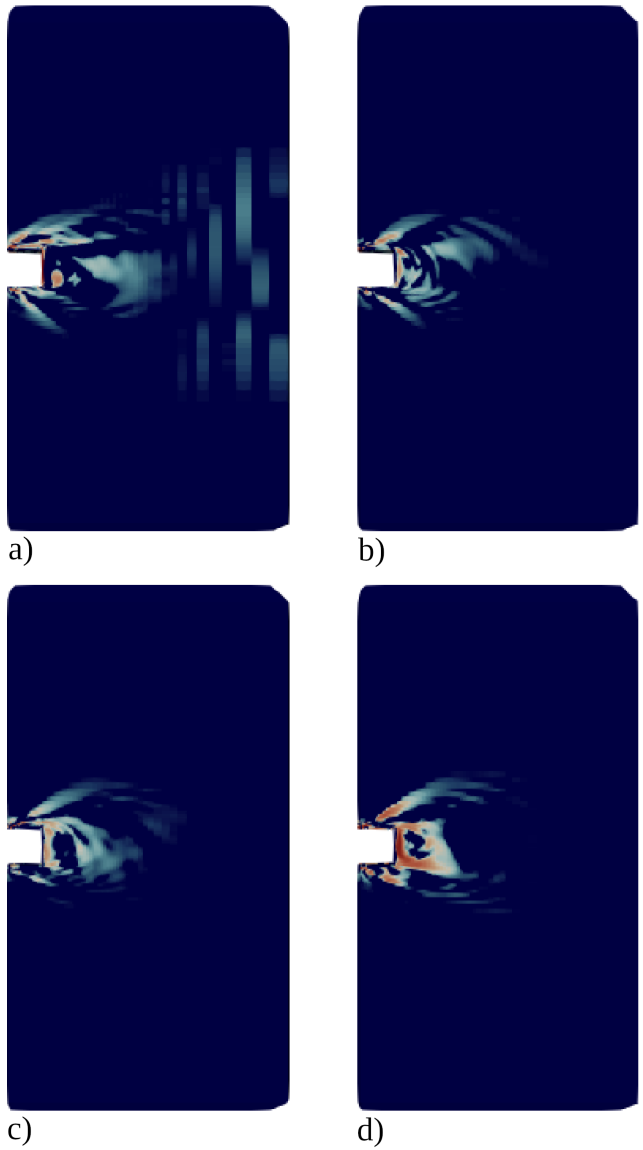


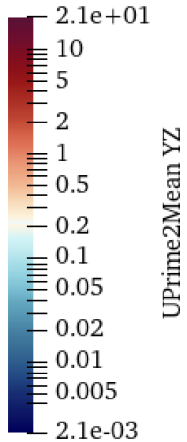


Simulation Results - xz and yz UPrime2Mean



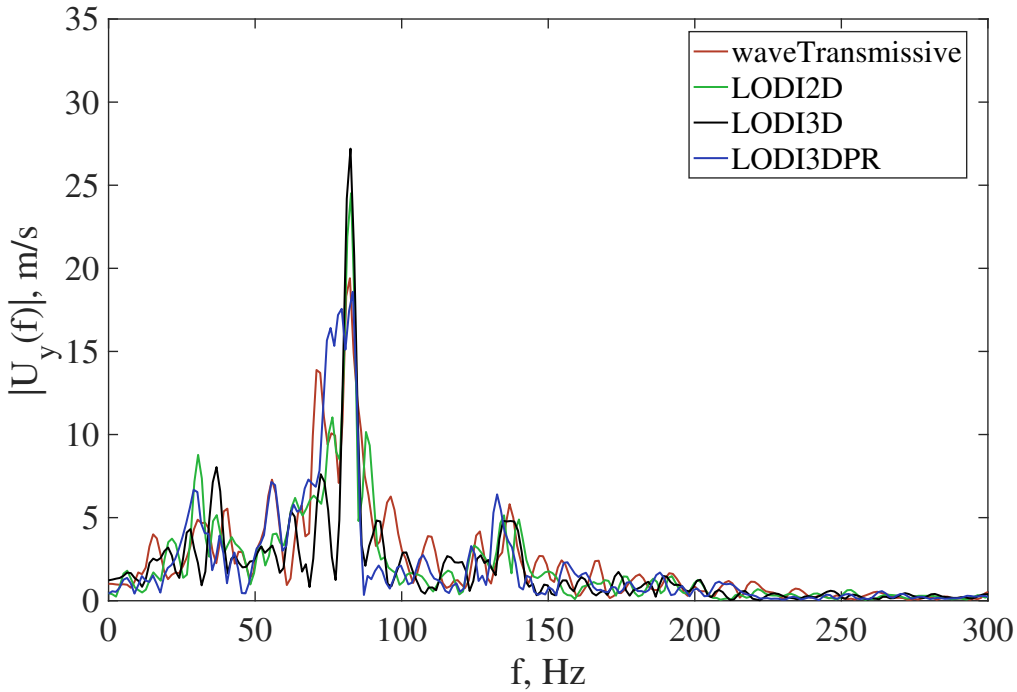






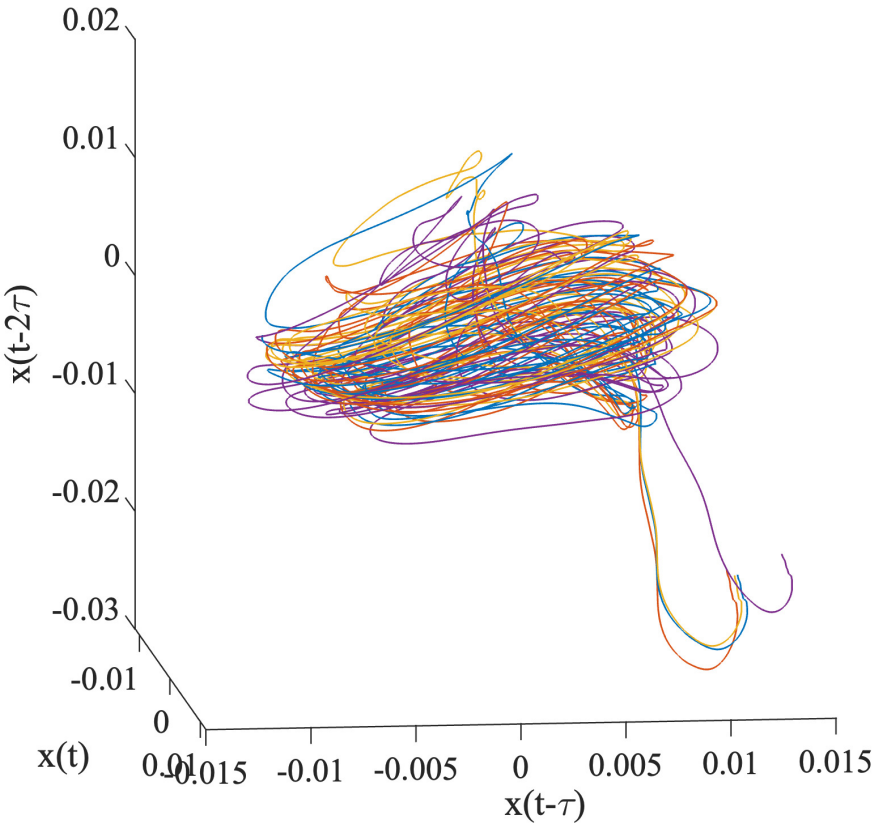
Simulation Results - FFT

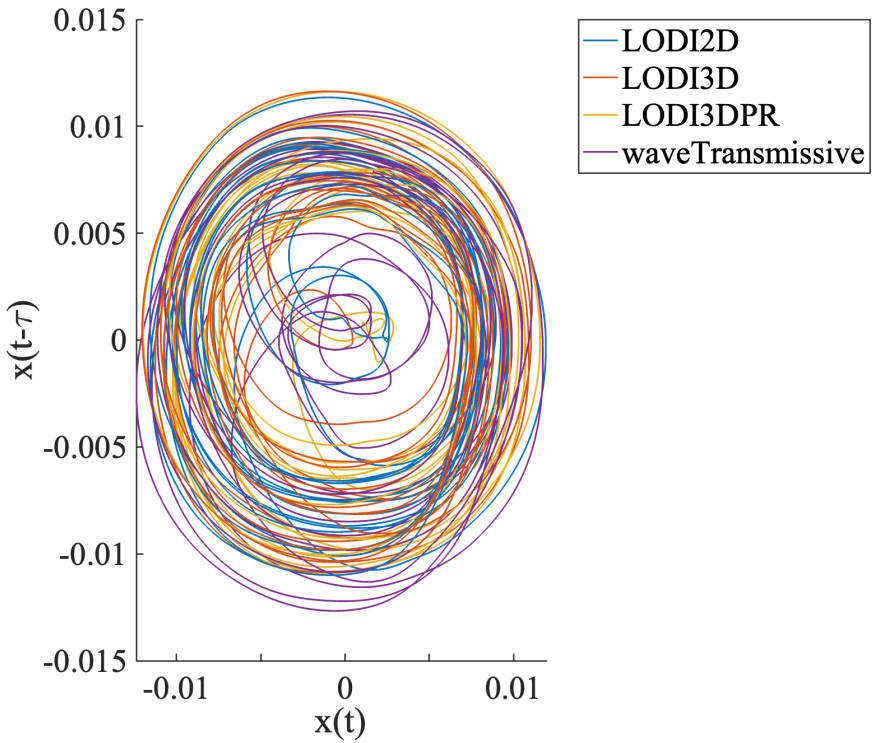
The main frequency peak remains the same, however the amplitude is slightly different from case to case. The sharpest peak corresponds to `modifiedLODI3D`, whilst the largest spread, `modifiedLODI3DPR`. Stronger reflections naturally affect the vortex-shedding more.



Simulation Results - TDE

The Time-Delay Embedding shows similar trajectories for all cases. The limit-cycle is found in all cases with different amounts of intermittent behaviors where `waveTransmissive` corresponds to the case which has the most intermittent events.





Summary and Conclusion

- To create well-posed boundary conditions, we can use NSCBC to obtain LODI relations.
- `mixed` is a general-purpose boundary condition class which requires input regarding f, ϕ_{ref}, and $\nabla\phi_{\text{ref}}$. These are obtained through the `updateCoeffs` member function in the `advective` class.
- The `waveTransmissive` class alters the definition of the member function `advectionSpeed` in the `advective` class.
- The `evaluate` member function in `mixed` calls the `updateCoeffs` member function in `advective` class to obtain f, ϕ_{ref}, and $\nabla\phi_{\text{ref}}$ in order to calculate ϕ_f^{n+1}.
- A simplified version of `LODI2D` and `mixedV2D` was created in order to make the transition to 3D simpler. This included neglecting the rotation transformation.
- A 3D non-reflecting boundary condition was developed corresponding to `modifiedMixedV3D` and `modifiedLODI3D`.
- This 3D non-reflecting boundary condition was slightly altered to make it a 3D partially-reflecting boundary condition through a constant `PR`. This corresponds to `modifiedLODI3DPR`.

Thank you for listening!

People take the longest possible paths, digress to numerous dead ends, and make all kinds of mistakes. Then historians come along and write summaries of this messy, nonlinear process and make it appear like a simple, straight line.

- Dean Kaman