Goal-oriented mesh adaptation for Firedrake

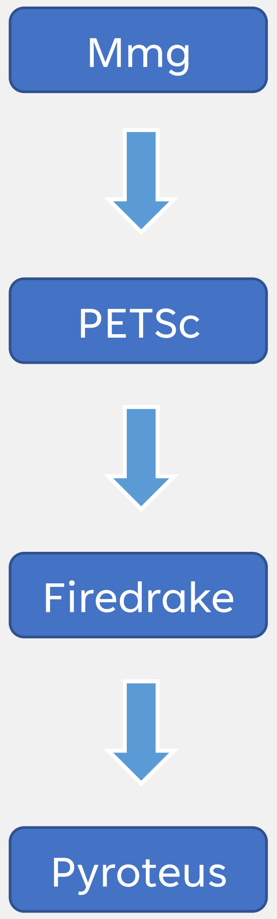
PI: Matt Piggott

Technical staff: Joe Wallwork

+

Main objectives

1. **Extend PETSc's metric-based mesh adaptation functionality.**
2. Pipe PETSc's mesh adaptation functionality through to Firedrake.
3. Develop a new goal-oriented error estimation and mesh adaptation module, Pyroteus.
4. Applications and documentation.



Metric-based mesh adaptation package. Written in C.

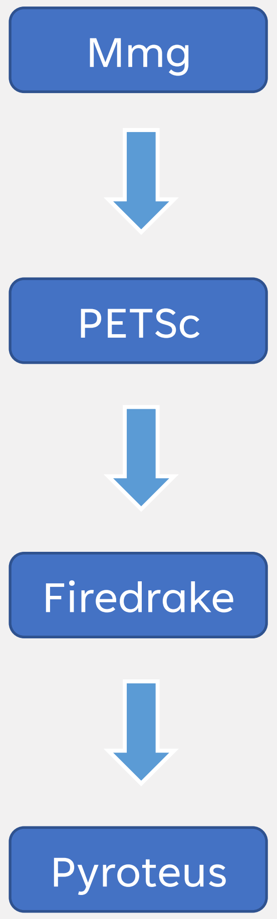
Solver library (amongst much more!). Written in C.

Finite element library. Written in Python, with automatic C code generation.

Goal-oriented error estimation module. Written in Python.

Main objectives

1. Extend PETSc's metric-based mesh adaptation functionality.
2. **Pipe PETSc's mesh adaptation functionality through to Firedrake.**
3. Develop a new goal-oriented error estimation and mesh adaptation module, Pyroteus.
4. Applications and documentation.



Metric-based mesh adaptation package. Written in C.

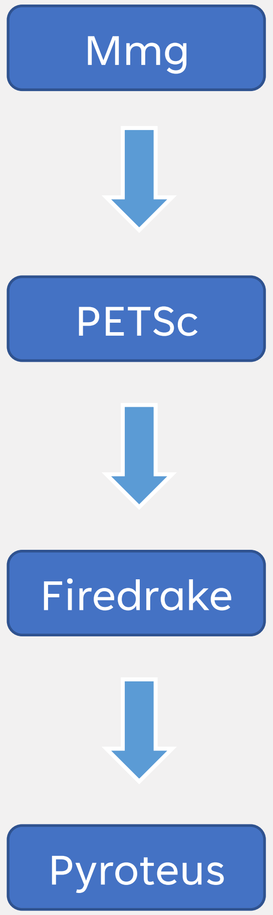
Solver library (amongst much more!). Written in C.

Finite element library. Written in Python, with automatic C code generation.

Goal-oriented error estimation module. Written in Python.

Main objectives

1. Extend PETSc’s metric-based mesh adaptation functionality.
2. Pipe PETSc’s mesh adaptation functionality through to Firedrake.
3. **Develop a new goal-oriented error estimation and mesh adaptation module, Pyroteus.**
4. Applications and documentation.



Metric-based mesh adaptation package. Written in C.

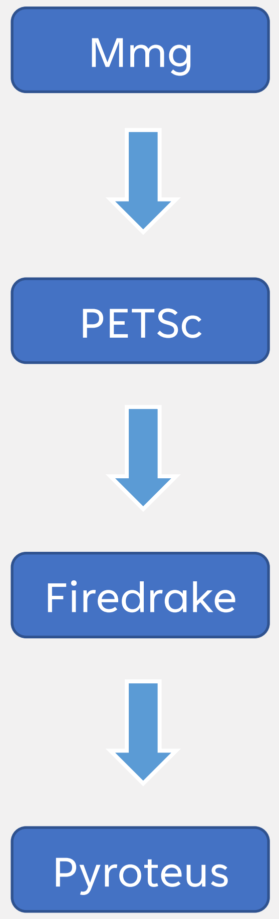
Solver library (amongst much more!). Written in C.

Finite element library. Written in Python, with automatic C code generation.

Goal-oriented error estimation module. Written in Python.

Main objectives

1. Extend PETSc’s metric-based mesh adaptation functionality.
2. Pipe PETSc’s mesh adaptation functionality through to Firedrake.
3. Develop a new goal-oriented error estimation and mesh adaptation module, Pyroteus.
4. **Applications and documentation.**



Metric-based mesh adaptation package. Written in C.

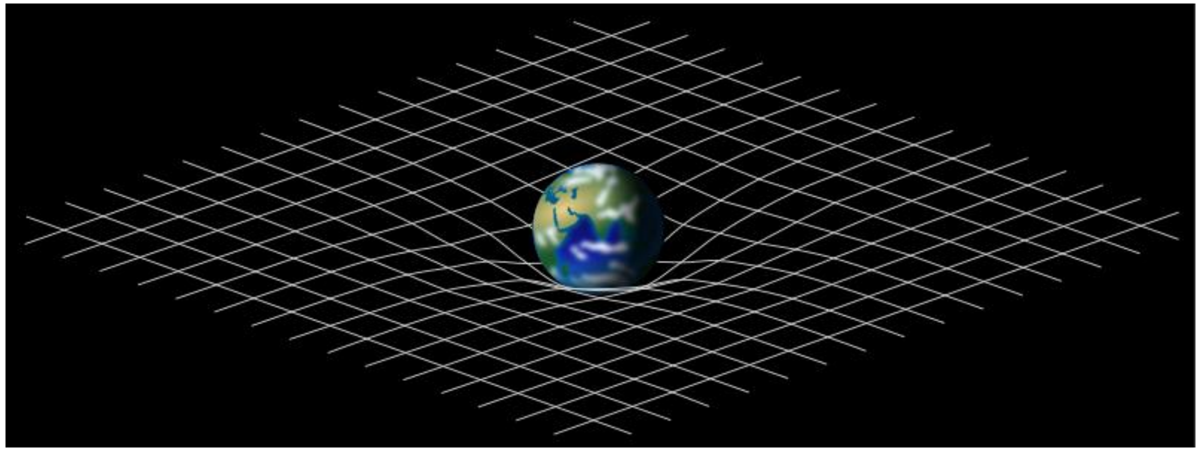
Solver library (amongst much more!). Written in C.

Finite element library. Written in Python, with automatic C code generation.

Goal-oriented error estimation module. Written in Python.

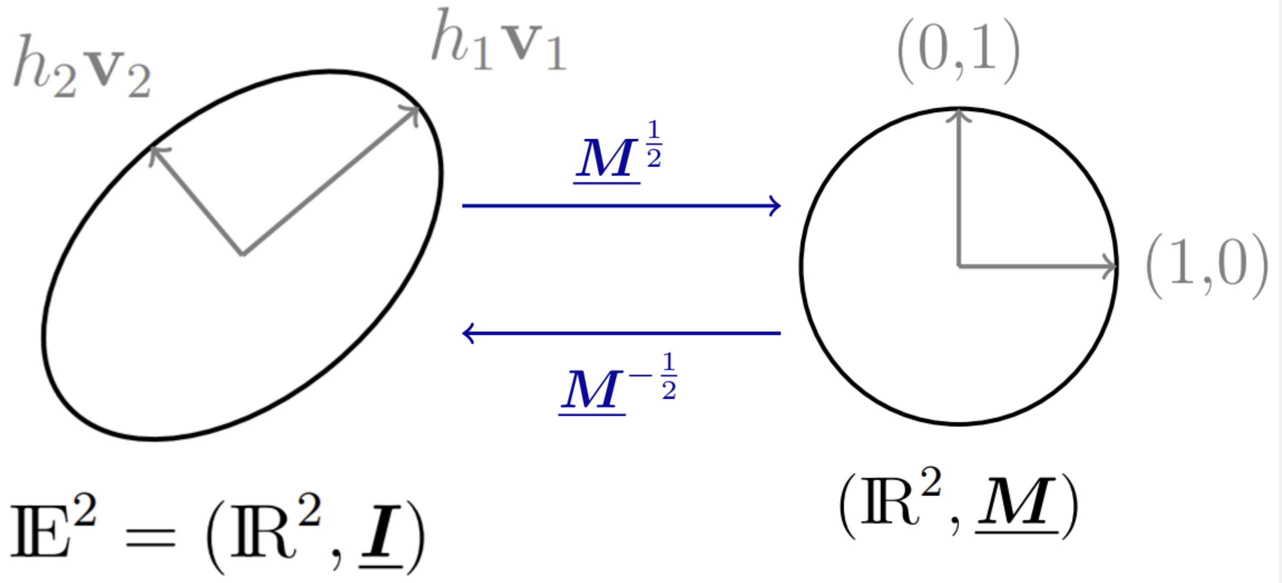
1. Metric-based mesh adaptation in PETSc and Firedrake

Riemannian metric fields


The space-time of general relativity

An nD *Riemannian metric*, $\mathcal{M} = \{\underline{\boldsymbol{M}}(\boldsymbol{x})\}_{\boldsymbol{x}\in\Omega}$, takes the value of an $n \times n$ symmetric positive-definite (SPD) matrix at each point.

Riemannian metric fields



$h_2\mathbf{v}_2$
$h_1\mathbf{v}_1$
$(0,1)$
$(1,0)$
$\underline{\boldsymbol{M}}^{\frac{1}{2}}$
$\underline{\boldsymbol{M}}^{-\frac{1}{2}}$
$\mathbb{E}^2 = (\mathbb{R}^2, \underline{\boldsymbol{I}})$
$(\mathbb{R}^2, \underline{\boldsymbol{M}})$

An *nD Riemannian metric*, $\mathcal{M} = \{\underline{\boldsymbol{M}}(\boldsymbol{x})\}_{x\in\Omega}$, takes the value of an $n \times n$ symmetric positive-definite (SPD) matrix at each point.

Symmetry implies

$$\underline{\boldsymbol{M}}(\boldsymbol{x}) = \underline{\boldsymbol{V}}(\boldsymbol{x})\underline{\boldsymbol{\Lambda}}(\boldsymbol{x})\underline{\boldsymbol{V}}^T(\boldsymbol{x}),$$

where

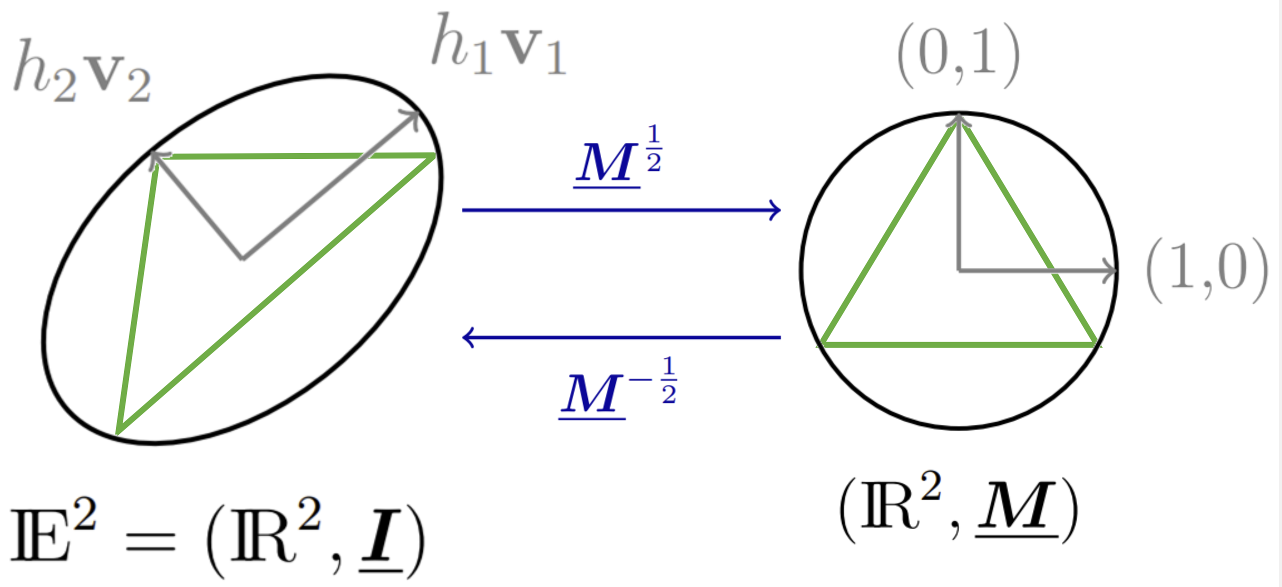
$$\underline{\boldsymbol{\Lambda}}(\boldsymbol{x}) = \operatorname{diag}\big(\lambda_1(\boldsymbol{x}), \dots, \lambda_n(\boldsymbol{x})\big),$$
$$\underline{\boldsymbol{V}}(\boldsymbol{x}) = [\boldsymbol{v}_1(\boldsymbol{x}) \quad \dots \quad \boldsymbol{v}_n(\boldsymbol{x})].$$

Positive-definiteness implies

$$\lambda_i > 0, \forall i = 1{:}n.$$

Define $h_i := {}^1/_{\sqrt{\lambda_i}}, \forall i = 1{:}n.$

Riemannian metric fields



$h_2\mathbf{v}_2$
$h_1\mathbf{v}_1$
$\underline{\boldsymbol{M}}^{\frac{1}{2}}$
$\underline{\boldsymbol{M}}^{-\frac{1}{2}}$
$(0,1)$
$(1,0)$
$\mathbb{E}^2 = (\mathbb{R}^2, \underline{\boldsymbol{I}})$
$(\mathbb{R}^2, \underline{\boldsymbol{M}})$

An n*D Riemannian metric*, $\mathcal{M} = \{\underline{\boldsymbol{M}}(\boldsymbol{x})\}_{x\in\Omega}$, takes the value of an $n \times n$ symmetric positive-definite (SPD) matrix at each point.

Symmetry implies

$$\underline{\boldsymbol{M}}(\boldsymbol{x}) = \underline{\boldsymbol{V}}(\boldsymbol{x})\underline{\boldsymbol{\Lambda}}(\boldsymbol{x})\underline{\boldsymbol{V}}^T(\boldsymbol{x}),$$

where

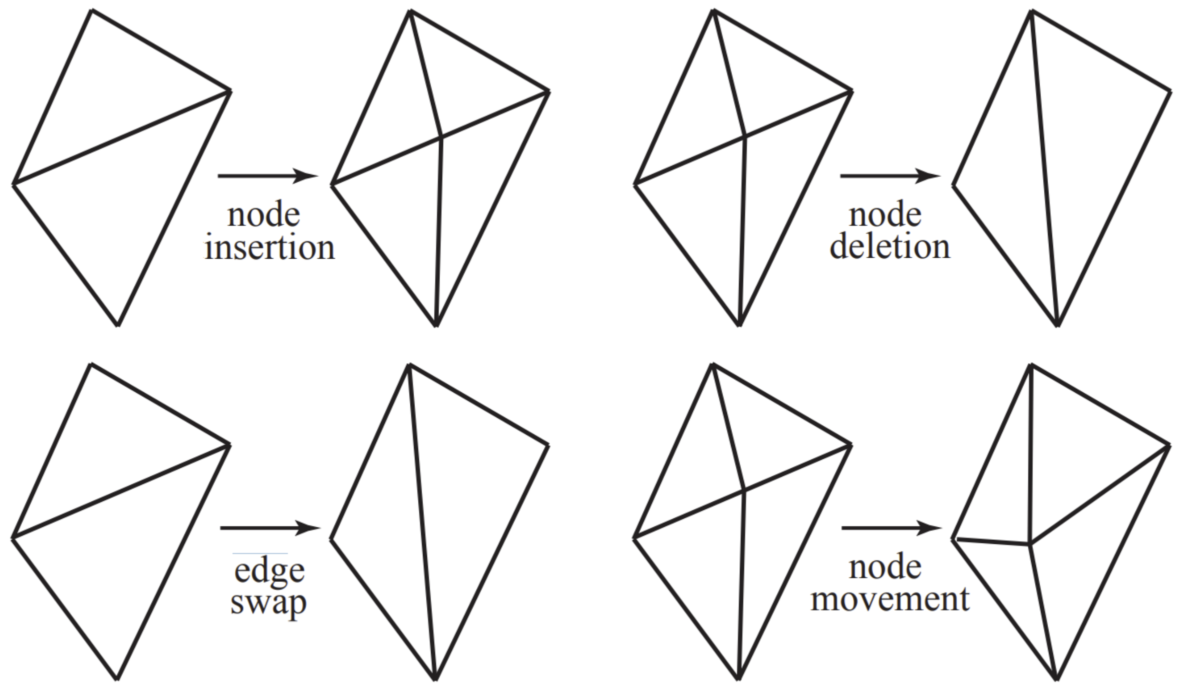
$$\underline{\boldsymbol{\Lambda}}(\boldsymbol{x}) = \mathrm{diag}\big(\lambda_1(\boldsymbol{x}), \dots, \lambda_n(\boldsymbol{x})\big),$$
$$\underline{\boldsymbol{V}}(\boldsymbol{x}) = [\boldsymbol{v}_1(\boldsymbol{x}) \quad \dots \quad \boldsymbol{v}_n(\boldsymbol{x})].$$

Positive-definiteness implies

$$\lambda_i > 0, \forall i = 1{:}n.$$

Define $h_i \coloneqq {}^1/_{\sqrt{\lambda_i}}, \forall i = 1{:}n.$

2D metric-based mesh adaptation



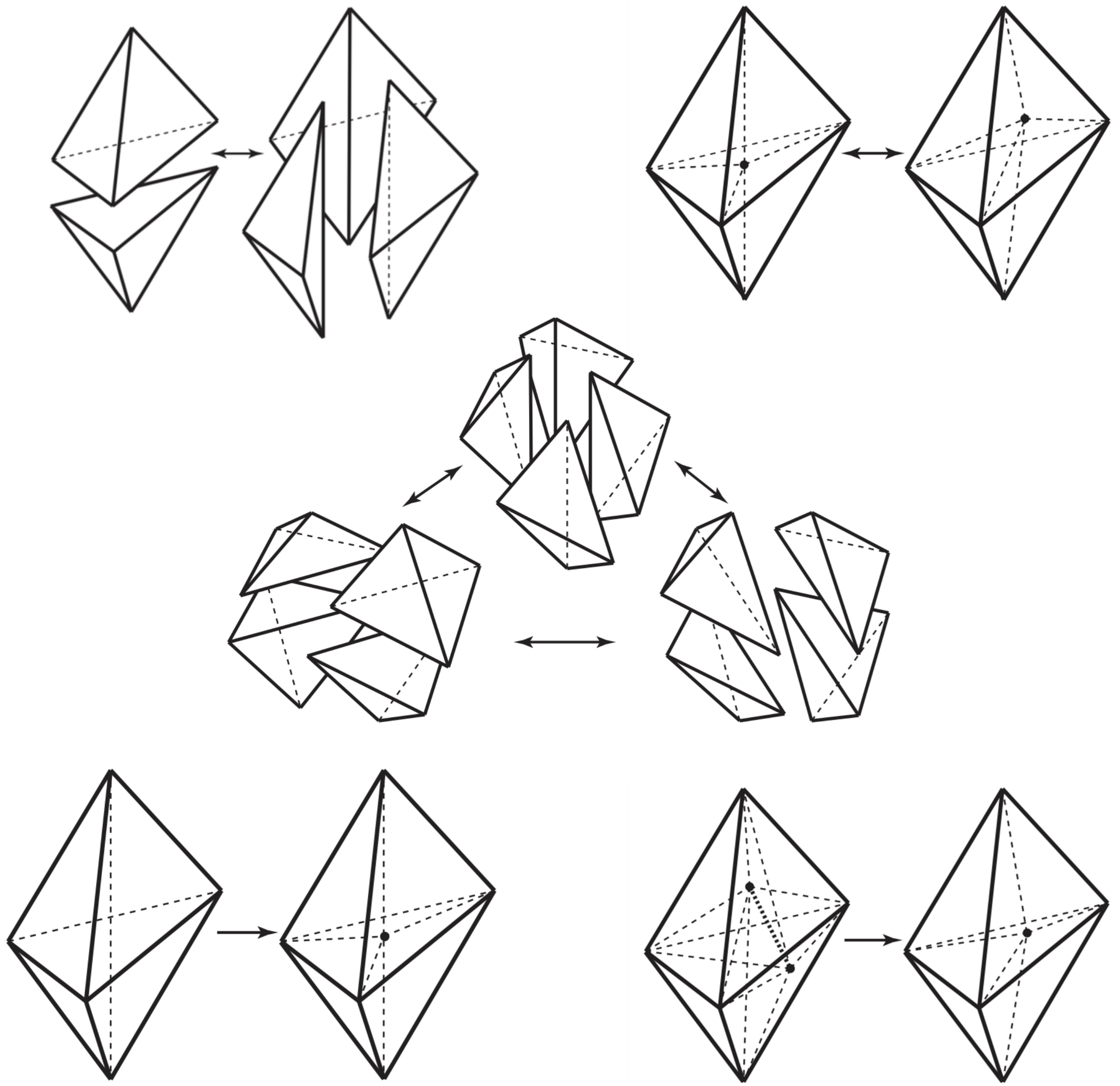
Minimise a quality function,

$$Q_{\mathcal{M}}(K) = \frac{\sqrt{3}}{2} \frac{\sum_{\gamma \in \partial K} \ell_{\mathcal{M}}(\gamma)^2}{|K|_{\mathcal{M}}}$$

for each mesh element K, by applying mesh modification operations.

Here $Q_{\mathcal{M}}(K)$ is minimised if K is an equilateral triangle with unit edges.

3D metric-based mesh adaptation



Minimise a quality function,

$$Q_{\mathcal{M}}(K) = \frac{\sqrt{3}}{216} \frac{\left(\sum_{\gamma\in\partial K} \ell_{\mathcal{M}}(\gamma)^2\right)^{\frac{3}{2}}}{|K|_{\mathcal{M}}}$$

for each mesh element K, by applying mesh modification operations.

Here $Q_{\mathcal{M}}(K)$ is minimised if K is a regular tetrahedron with unit edges.

Creating metric fields in PETSc

```
DMPlexMetricCreate
DMPlexMetricCreateIsotropic
DMPlexMetricCreateUniform
DMPlexComputeGradientClementInterpolant


DMPlexMetricEnforceSPD
DMPlexMetricSetMinimumMagnitude
DMPlexMetricSetMaximumMagnitude
DMPlexMetricSetMaximumAnisotropy
DMPlexMetricSetTargetComplexity
DMPlexMetricSetNormalizationOrder
DMPlexMetricNormalize
DMPlexMetricSetGradationFactor
DMPlexMetricSetHausdorffNumber


DMPlexMetricAverage
DMPlexMetricIntersection
```

Isotropic metrics can be constructed easily:

$$\mathcal{M} = \varepsilon \underline{\boldsymbol{I}}.$$

A common choice of anisotropic metric is to use the Hessian of a field of interest, u:

$$\underline{\boldsymbol{H}}(u) = \underline{\boldsymbol{V\Lambda V}}^T.$$

There are bounds on the interpolation error in u involving the Hessian.

Processing metric fields in PETSc

```
DMPlexMetricCreate
DMPlexMetricCreateIsotropic
DMPlexMetricCreateUniform
DMPlexComputeGradientClementInterpolant
```

```
DMPlexMetricEnforceSPD
DMPlexMetricSetMinimumMagnitude
DMPlexMetricSetMaximumMagnitude
DMPlexMetricSetMaximumAnisotropy
DMPlexMetricSetTargetComplexity
DMPlexMetricSetNormalizationOrder
DMPlexMetricNormalize
DMPlexMetricSetGradationFactor
DMPlexMetricSetHausdorffNumber
```

```
DMPlexMetricAverage
DMPlexMetricIntersection
```

To make sure we have a metric, Hessians should be made SPD:

$$|\underline{\boldsymbol{H}}| = \underline{\boldsymbol{V}}|\underline{\boldsymbol{\Lambda}}|\underline{\boldsymbol{V}}^T.$$

We can (approximately) enforce minimum/maximum element sizes by modifying $\underline{\boldsymbol{\Lambda}}$.

It is usually a good idea to normalise the metric, too.

Combining metric fields in PETSc

```
DMPlexMetricCreate
DMPlexMetricCreateIsotropic
DMPlexMetricCreateUniform
DMPlexComputeGradientClementInterpolant
```

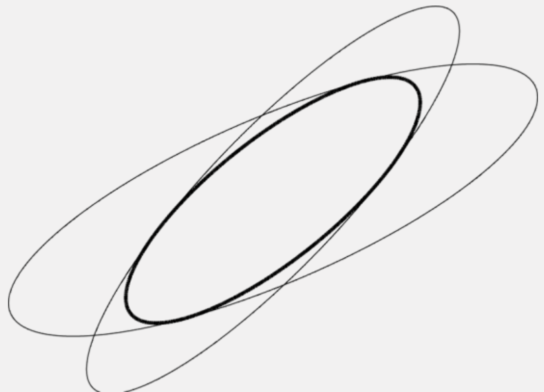
```
DMPlexMetricEnforceSPD
DMPlexMetricSetMinimumMagnitude
DMPlexMetricSetMaximumMagnitude
DMPlexMetricSetMaximumAnisotropy
DMPlexMetricSetTargetComplexity
DMPlexMetricSetNormalizationOrder
DMPlexMetricNormalize
DMPlexMetricSetGradationFactor
DMPlexMetricSetHausdorffNumber
```

```
DMPlexMetricAverage
DMPlexMetricIntersection
```

Often there are multiple metrics that we want to take account of.

Metric averaging: mathematically simple, but not geometrically intuitive.

Metric intersection: geometrically intuitive, but not mathematically simple.



Using metric utilities in petsc4py

`DMPlexMetricCreate`	`plex.metricCreate`
`DMPlexMetricCreateIsotropic`	`plex.metricCreateIsotropic`
`DMPlexMetricCreateUniform`	`plex.metricCreateUniform`
`DMPlexComputeGradientClementInterpolant`	`plex.computeGradientClementInterpolant`

`DMPlexMetricEnforceSPD`	`plex.metricEnforceSPD`
`DMPlexMetricSetMinimumMagnitude`	`plex.metricSetMinimumMagnitude`
`DMPlexMetricSetMaximumMagnitude`	`plex.metricSetMaximumMagnitude`
`DMPlexMetricSetMaximumAnisotropy`	`plex.metricSetMaximumAnisotropy`
`DMPlexMetricSetTargetComplexity`	`plex.metricSetTargetComplexity`
`DMPlexMetricSetNormalizationOrder`	`plex.metricSetNormalizationOrder`
`DMPlexMetricNormalize`	`plex.metricNormalize`
`DMPlexMetricSetGradationFactor`	`plex.metricSetGradationFactor`
`DMPlexMetricSetHausdorffNumber`	`plex.metricSetHausdorffNumber`

+

`DMPlexMetricAverage`	`plex.metricAverage`
`DMPlexMetricIntersection`	`plex.metricIntersection`

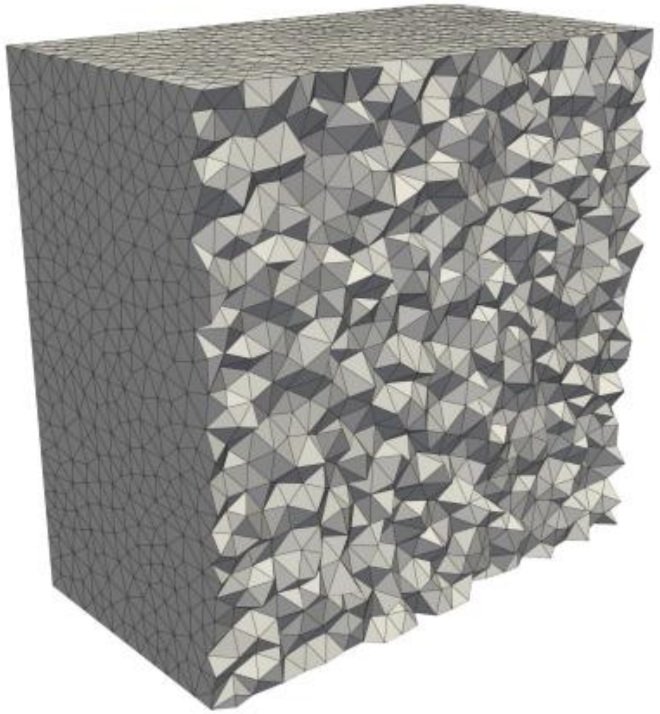
Metric fields in Firedrake

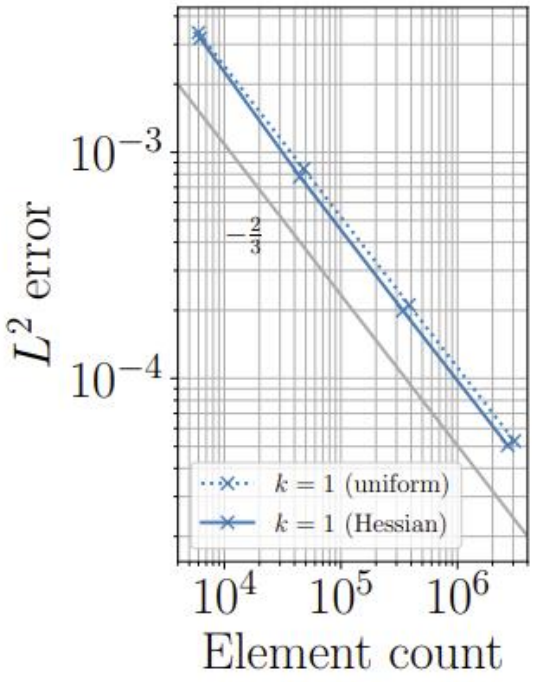
Proposed implementation gathers petsc4py metric utilities in a Riemannian metric class.

The underlying field data can be accessed and modified on either the Firedrake level (`Function`) or PETSc level (`Vec`).

```
class RiemannianMetric(Metric):

    def set_parameters(self, **kwargs):
        …

    def enforce_spd(self, **kwargs):
        …

    def normalise(self, **kwargs):
        …

    def average(self, *metrics):
        …

    def intersect(self, *metrics):
        …

    …
```

Poisson test case in PETSc





$\Delta x \approx (\text{element count})^{-\frac{1}{3}}$

Consider the Poisson problem

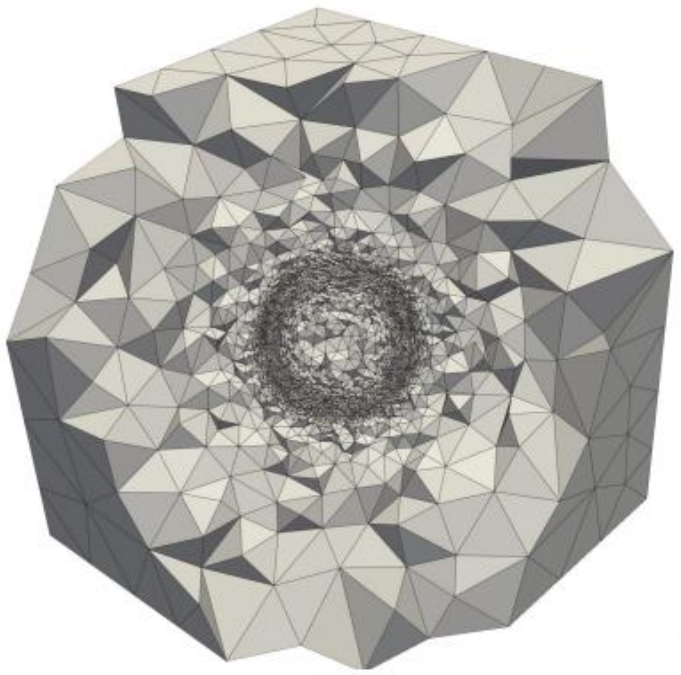
$$\Delta u = f$$

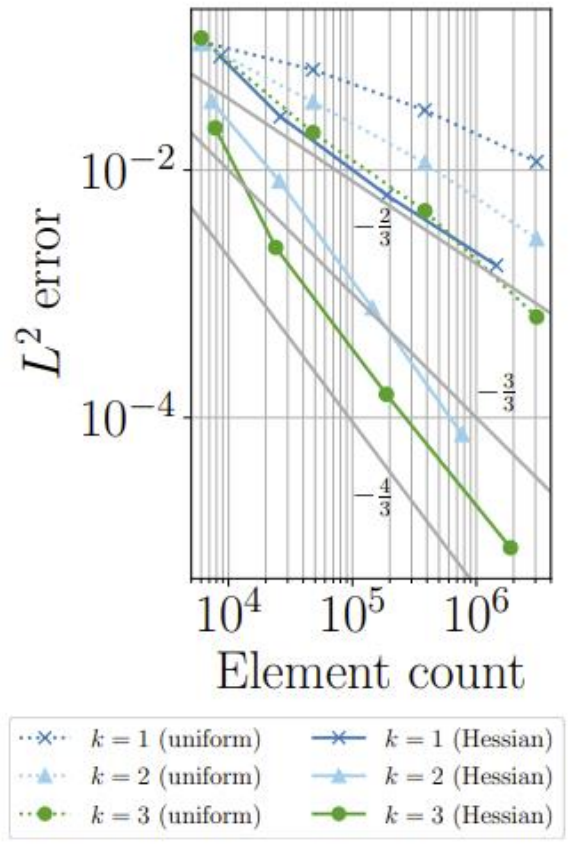
on the unit cube with Dirichlet conditions.

We can manufacture an analytical solution by choosing f appropriately.

Example 1: $u(x, y, z) = \frac{4}{3}(x^2 + y^2 + z^2)$.

Poisson test case in PETSc





$$\Delta x \approx (\text{element count})^{-\frac{1}{3}}$$

Consider the Poisson problem

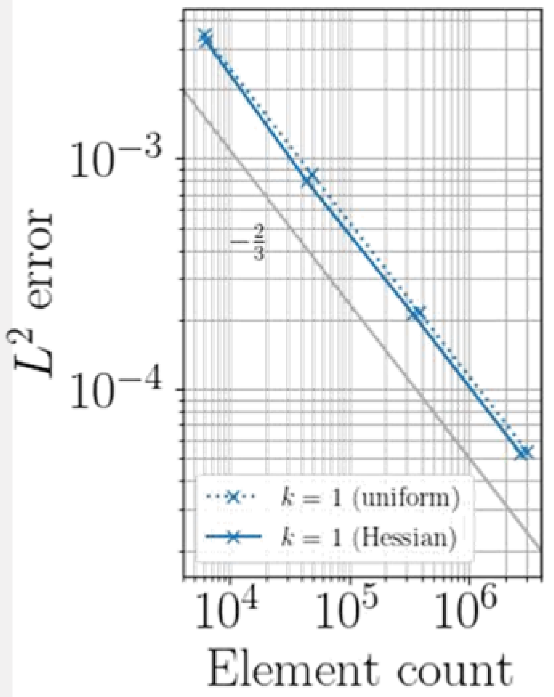
$$\Delta u = f$$

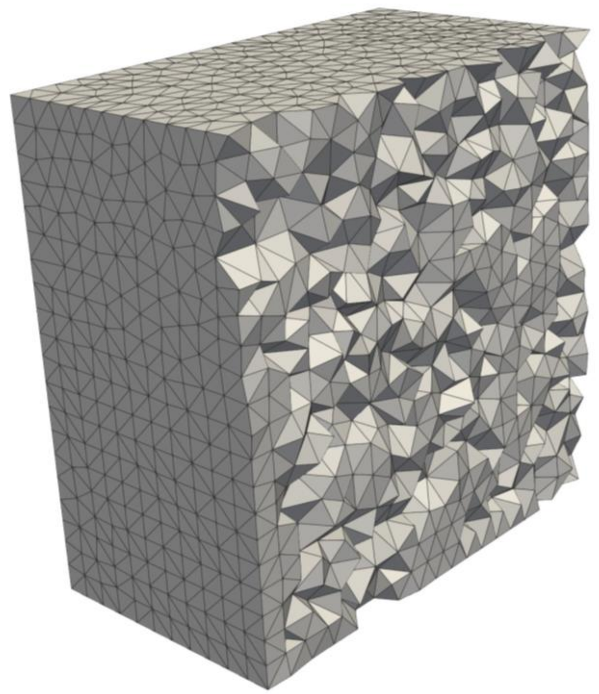
on the unit cube with Dirichlet conditions.

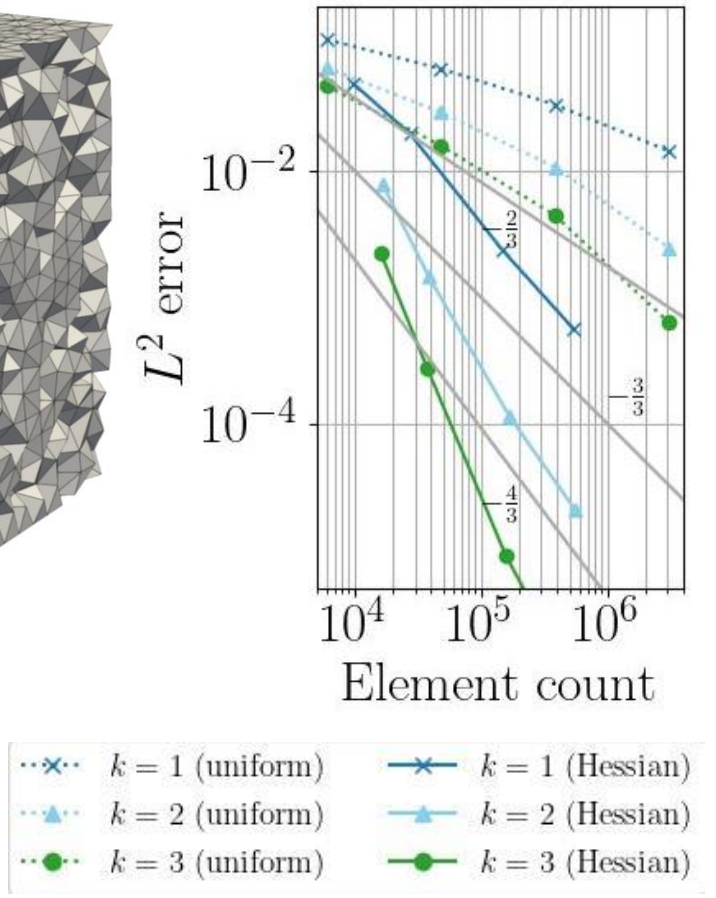
We can manufacture an analytical solution by choosing f appropriately.

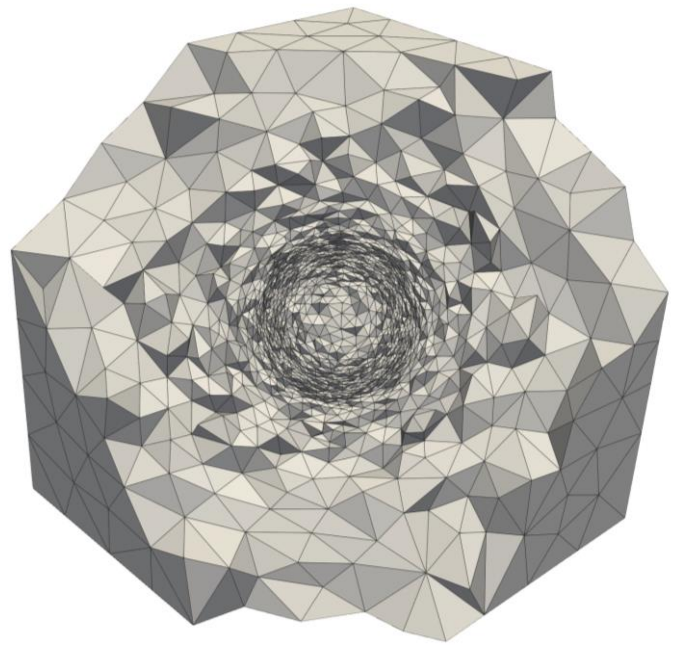
Example 2: (smoothed) spherical indicator function.

Poisson test case in Firedrake





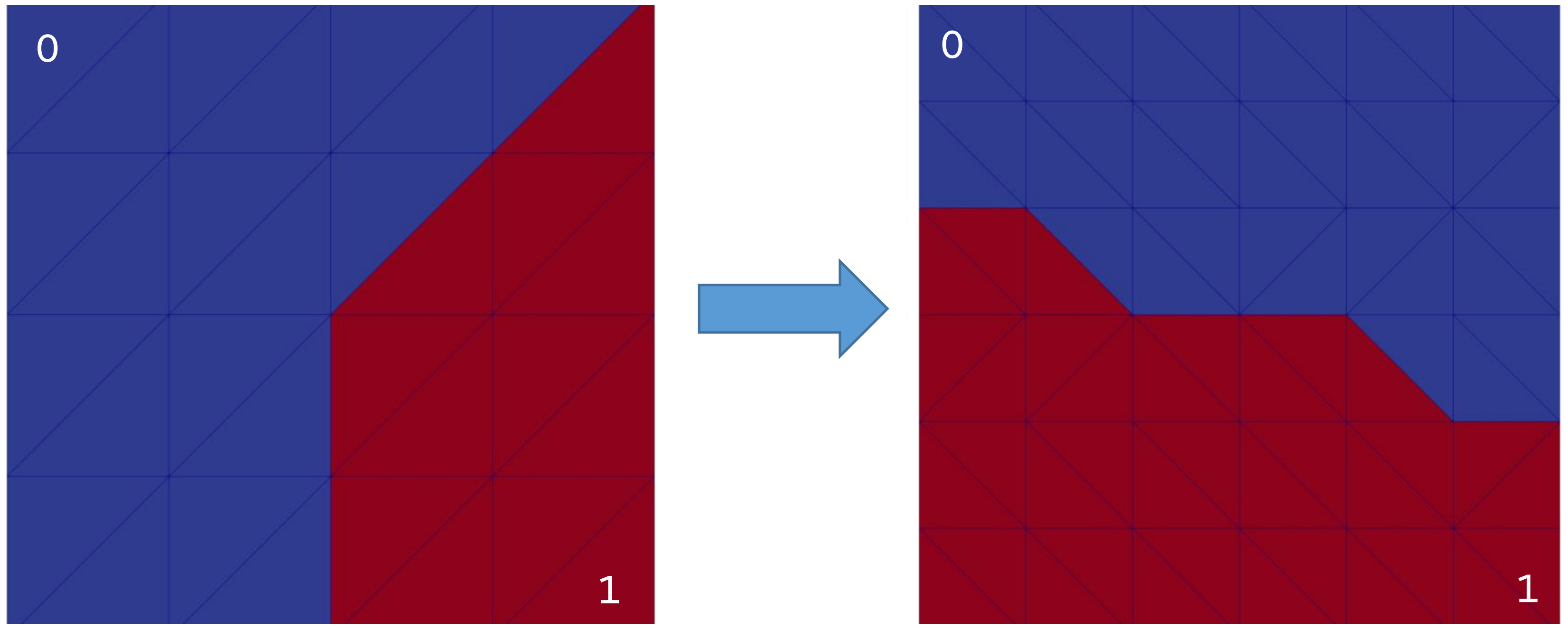




$$\Delta x \approx (\text{element count})^{-\frac{1}{3}}$$

Parallel mesh-to-mesh interpolation

PETSc's mesh-to-mesh interpolation functionality does not currently work in parallel. Firedrake is similarly limited.

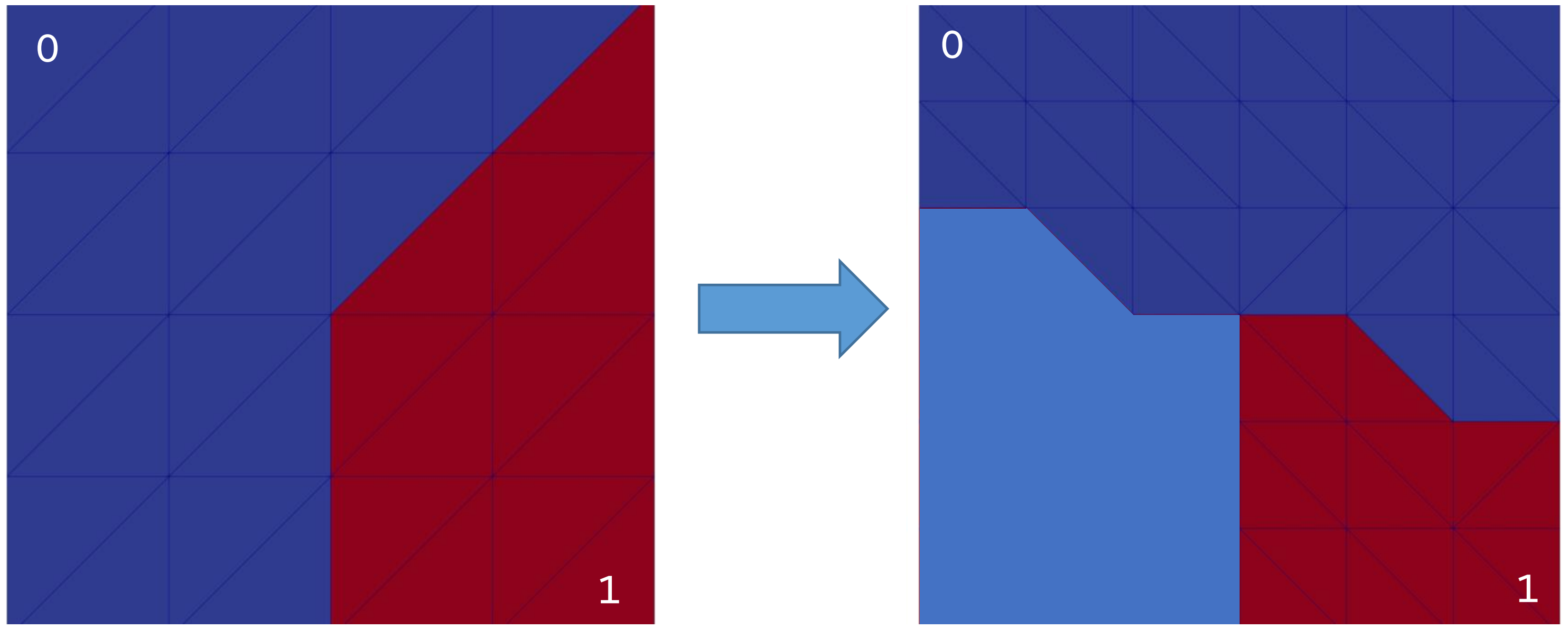


Source mesh

Target mesh

Parallel mesh-to-mesh interpolation

PETSc's mesh-to-mesh interpolation functionality does not currently work in parallel. Firedrake is similarly limited.

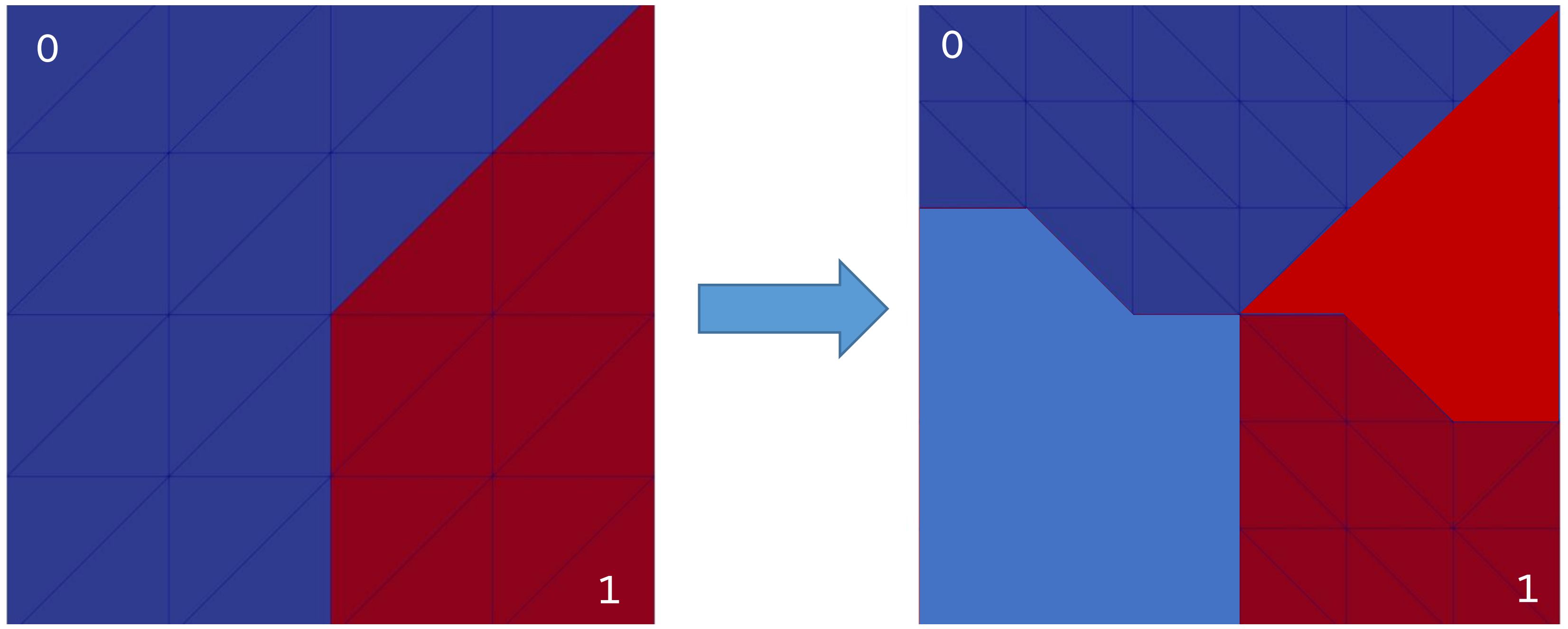


Source mesh

Target mesh

Parallel mesh-to-mesh interpolation

PETSc’s mesh-to-mesh interpolation functionality does not currently work in parallel. Firedrake is similarly limited.

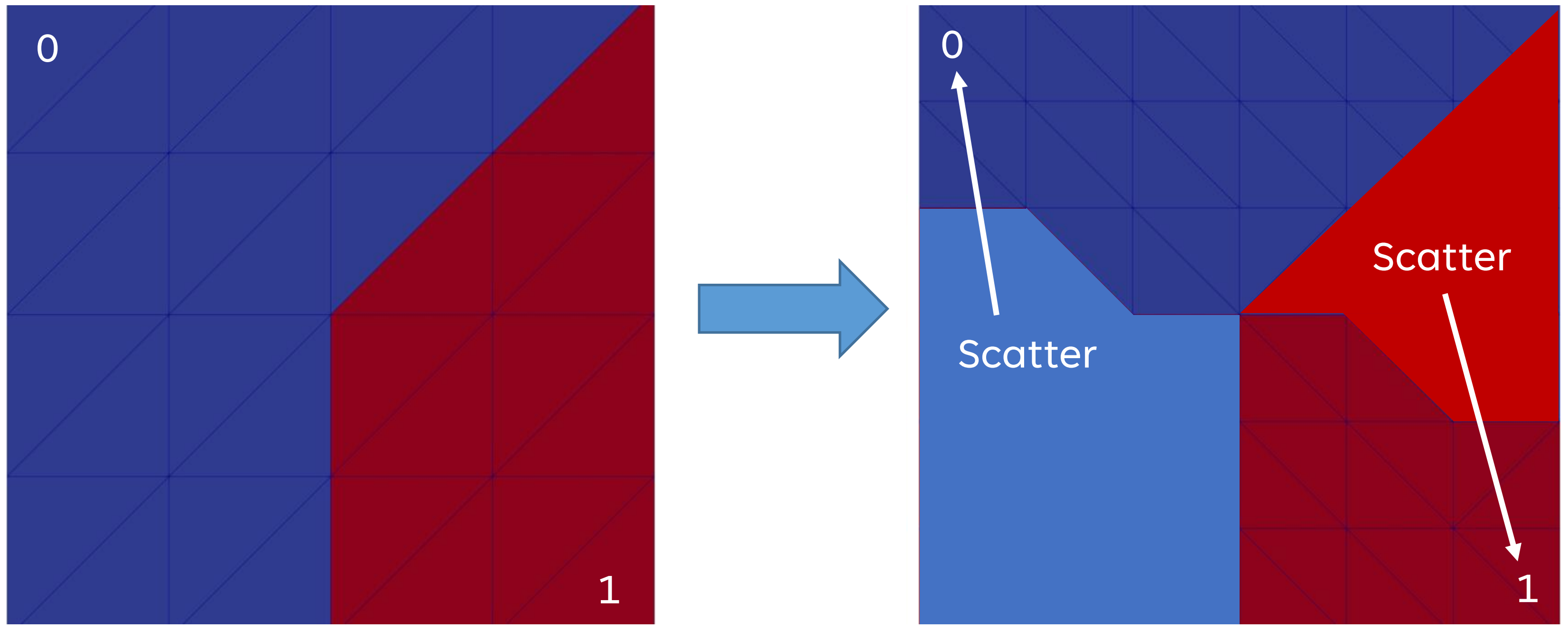


Source mesh

Target mesh

Parallel mesh-to-mesh interpolation

PETSc's mesh-to-mesh interpolation functionality does not currently work in parallel. Firedrake is similarly limited.



Source mesh

Target mesh

1. Metric-based mesh adaptation: Summary of achievements

- Coupled Mmg and ParMmg mesh adaptation tools to PETSc.
- Implemented routines for creating, modifying and combining Riemannian metrics in PETSc, plus Python bindings.
- Added new PETSc tests and tutorials.
- Open pull request for piping functionality from PETSc to Firedrake.
  - Proposed RiemannianMetric class.
  - Basic tests of functionality.
- Open merge request for parallel mesh-to-mesh interpolation in PETSc.
  - Basic tests of functionality.

2. Goal-oriented error estimation and mesh adaptation using Pyroteus

Goal-oriented error estimation

Variational form:

$$a(u, v) = L(v), \qquad \forall v \in V.$$

Weak form:

$$a(u_h, v) = L(v), \qquad \forall v \in V_h \subset V.$$

Weak residual:

$$\rho(u_h, v) = L(v) - a(u_h, v), \qquad v \in V.$$

Goal-oriented error estimation

Variational form:

$$a(u, v) = L(v), \qquad \forall v \in V.$$

Weak form:

$$a(u_h, v) = L(v), \qquad \forall v \in V_h \subset V.$$

Weak residual:

$$\rho(u_h, v) = L(v) - a(u_h, v), \qquad v \in V.$$

Adjoint in variational form:

$$a(v, u^*) = J(v), \qquad \forall v \in V.$$

Adjoint weak form:

$$a(v, u_h^*) = J(v), \qquad \forall v \in V_h.$$

Goal-oriented error estimation

Variational form:

$$a(u, v) = L(v), \qquad \forall v \in V.$$

Weak form:

$$a(u_h, v) = L(v), \qquad \forall v \in V_h \subset V.$$

Weak residual:

$$\rho(u_h, v) = L(v) - a(u_h, v), \qquad v \in V.$$

Adjoint in variational form:

$$a(v, u^*) = J(v), \qquad \forall v \in V.$$

Adjoint weak form:

$$a(v, u_h^*) = J(v), \qquad \forall v \in V_h.$$

Dual weighted residual:

$$\boxed{J(u) - J(u_h) \approx \rho(u_h, u^* - u_h^*).}$$

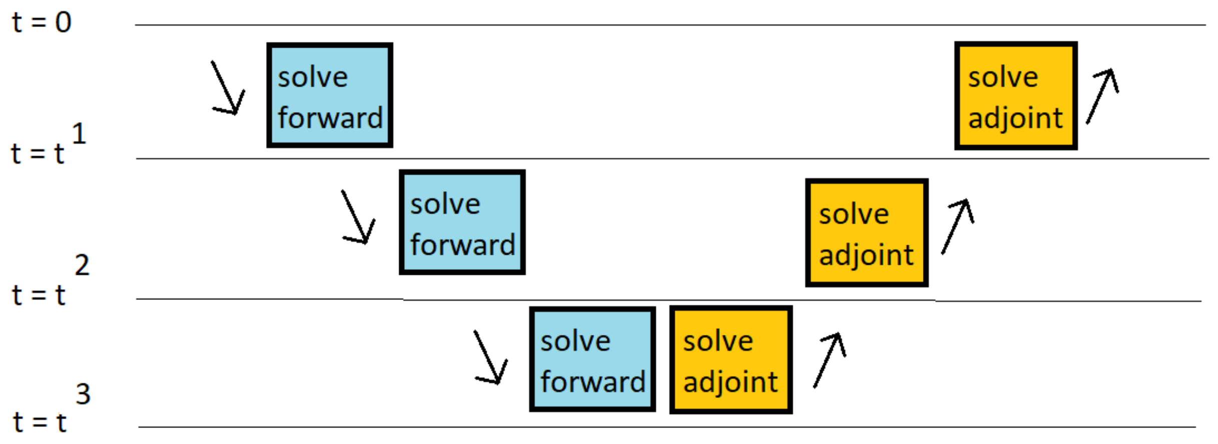
Goal-oriented error estimation

Main difficulties:

– The true adjoint solution is generally unknown.

– We need to approximate it, e.g. by solving the adjoint problem again in an enriched finite element space.

– For time-dependent problems, we need to be able to solve both forward and adjoint problems across sequences of meshes.



Pyroteus goal-oriented error estimation toolkit

To tackle the time-dependent adaptation problem, we make use of a “mesh sequence”, `MeshSeq`. The user just needs to provide:

- `TimePartition` object
- list of meshes
- `get_function_spaces(mesh)`
- `get_form(mesh_seq)`
- `get_solver(mesh_seq)`
- `get_initial_condition(mesh_seq)`

2. Goal-oriented error estimation and mesh adaptation using Pyroteus: Summary of achievements

Provided this information and a `get_qoi(mesh_seq)` function, the `GoalOrientedMeshSeq` subclass allows us to:

– Solve the forward problem over all meshes.

– Solve the adjoint problem over all meshes.

– Solve the adjoint problem over all “enriched” meshes.

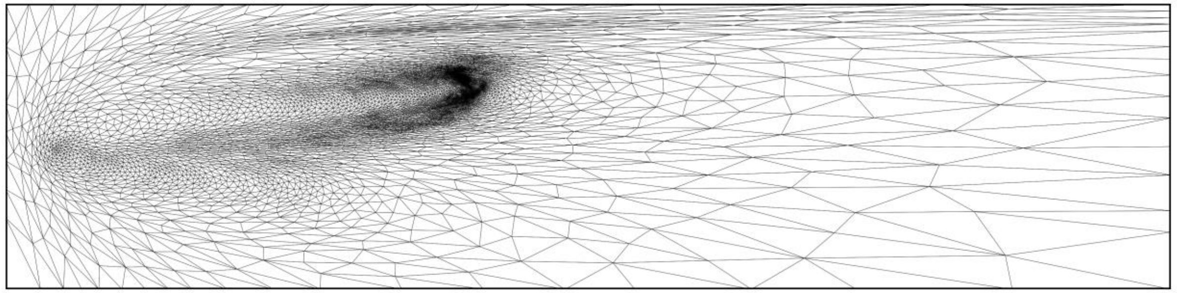
– Compute goal-oriented error estimators based on this information.

– Perform goal-oriented mesh adaptation in a “fixed point iteration” type approach.

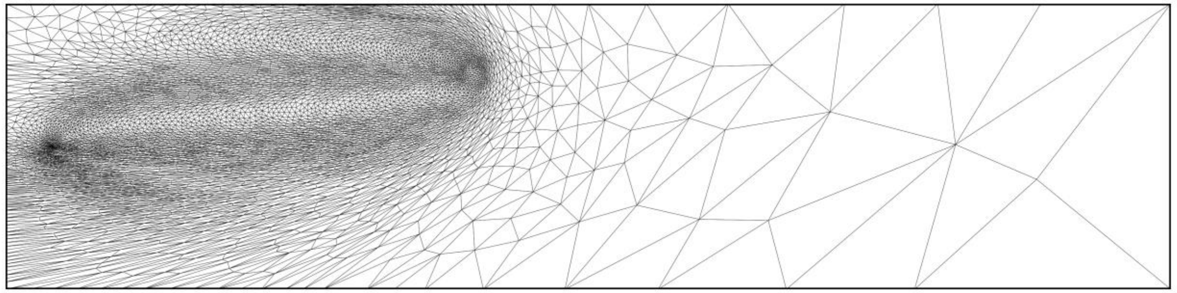
3. Applications and documentation

2D steady-state tracer transport test case

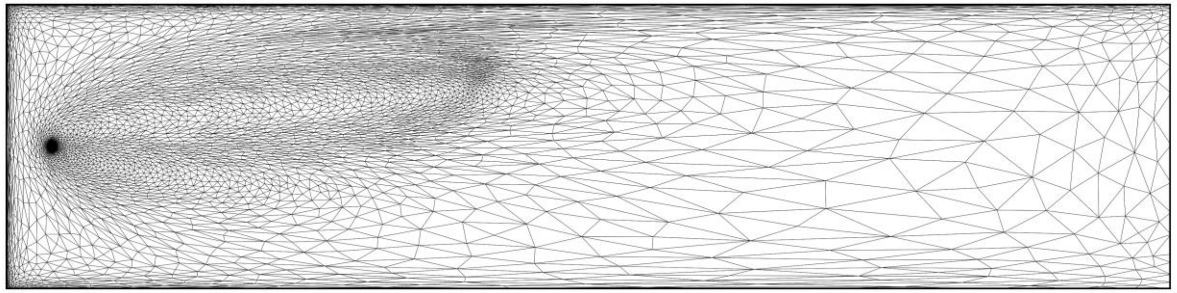
“Anisotropic DWR”



“Weighted Hessian”



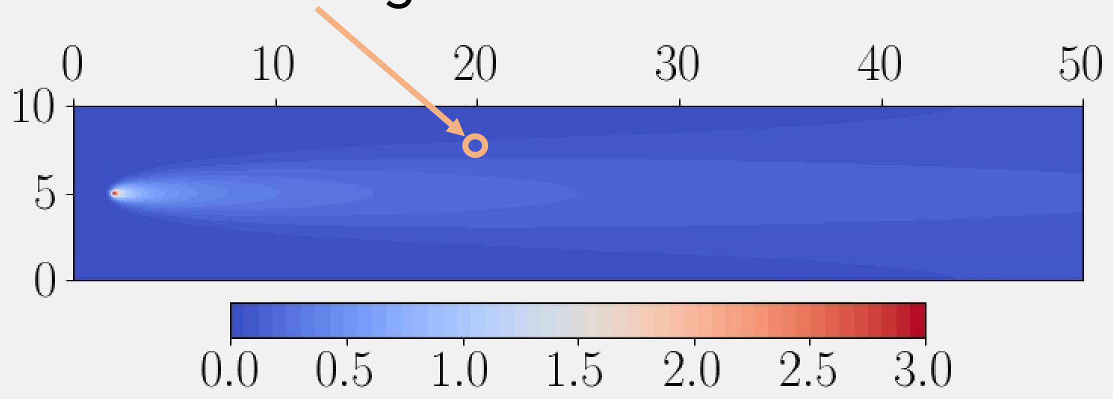
“Weighted gradient”

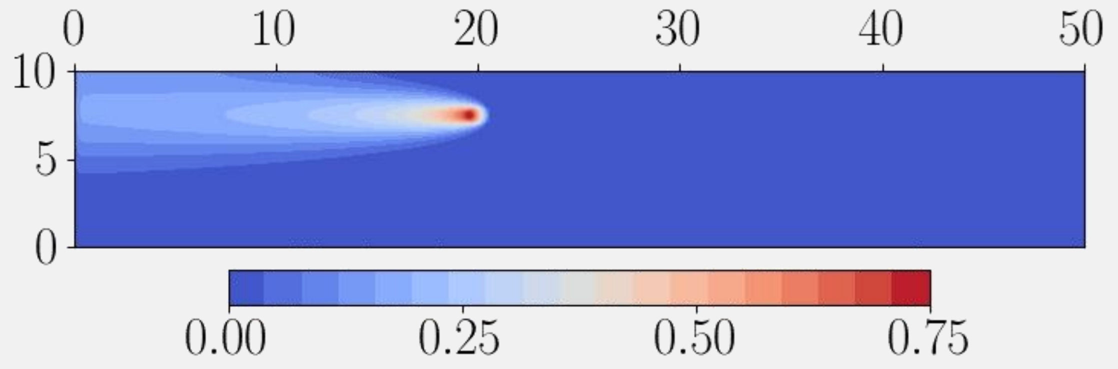


Consider the advection-diffusion problem

$$\mathbf{u} \cdot \nabla c - \nabla \cdot (D \nabla c) = f, \qquad c \in P1$$

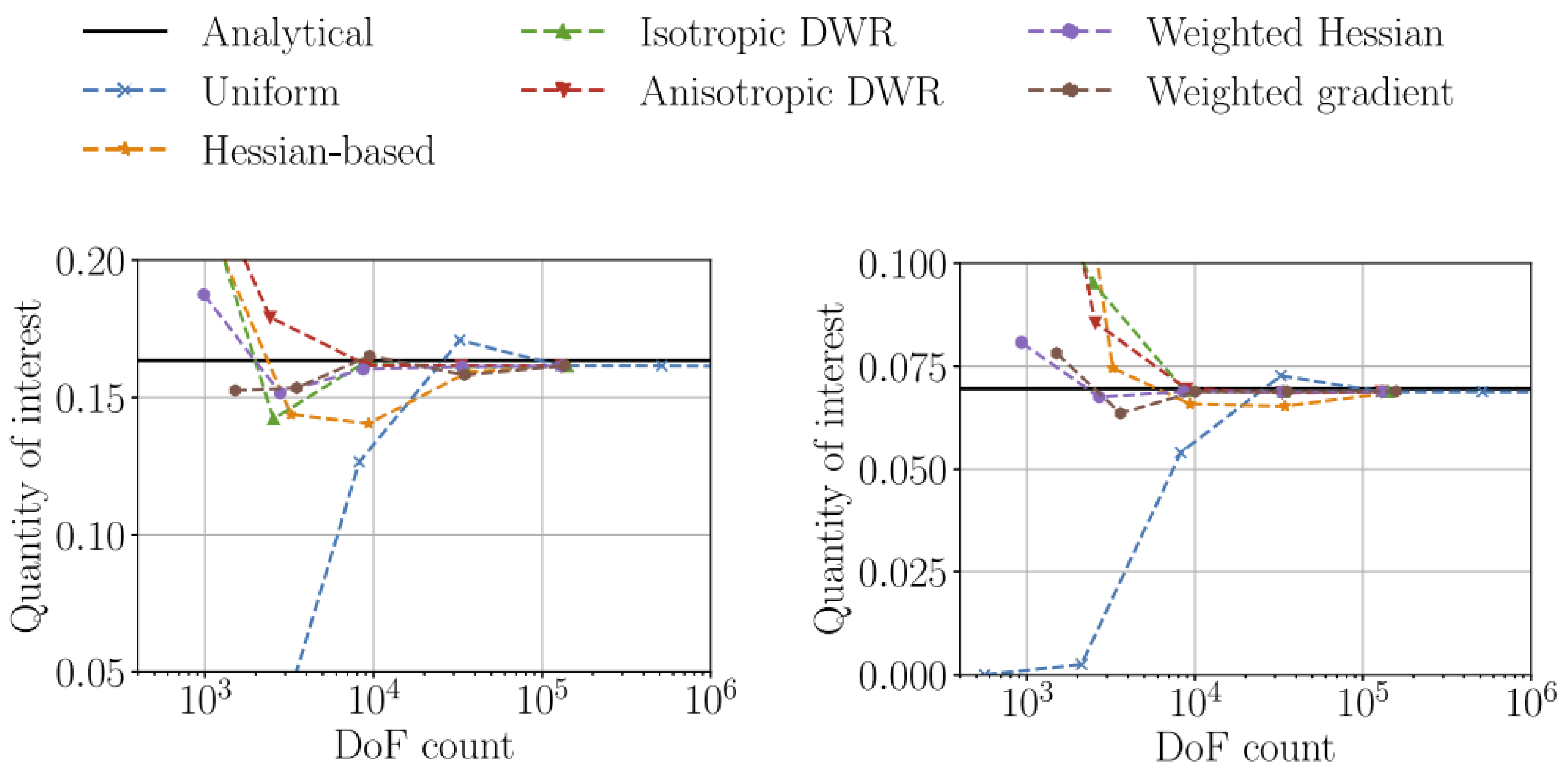
in a rectangular domain, where f is a point source. QoI: tracer concentration c in “receiver region”.



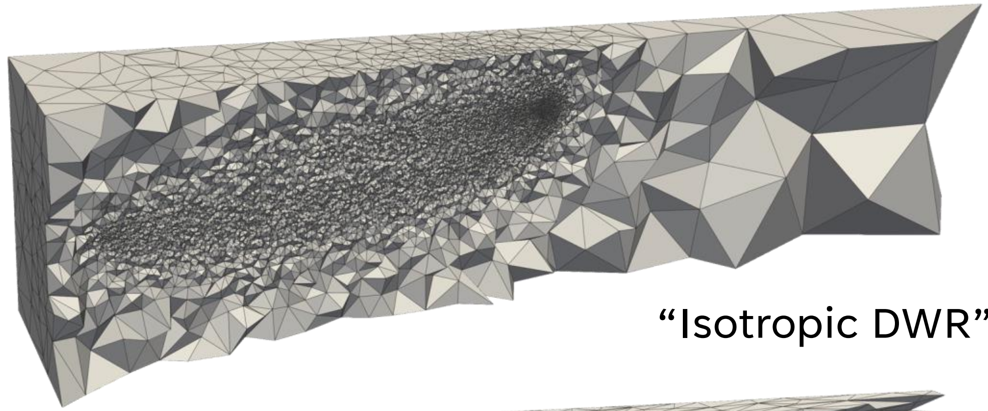


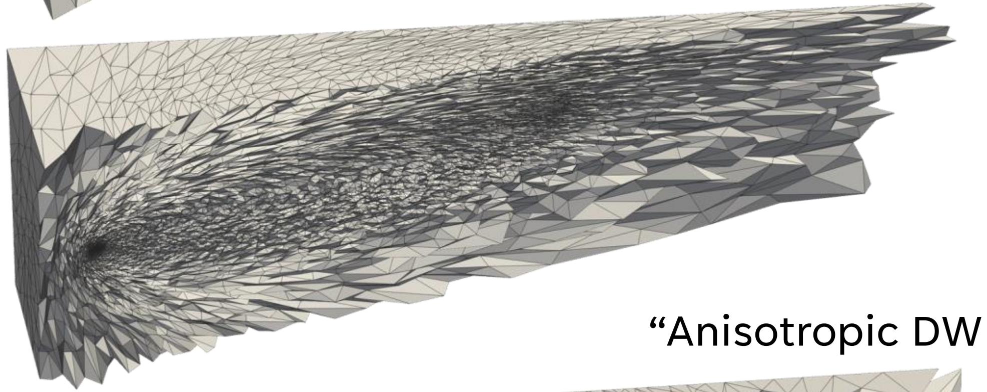
+

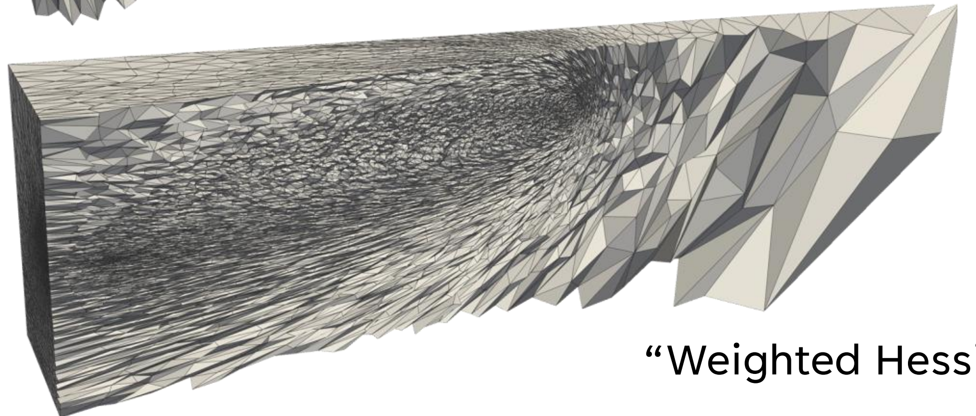
2D steady-state tracer transport test case



3D steady-state tracer transport test case


"Isotropic DWR"

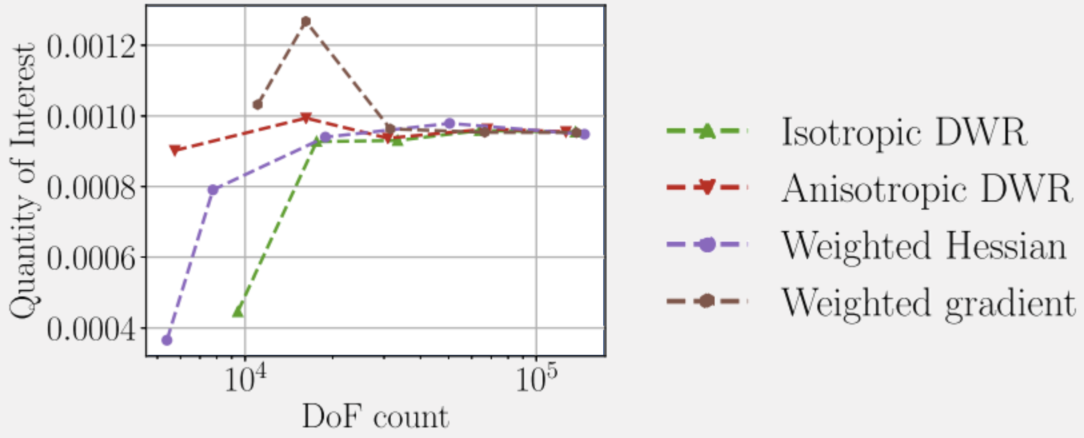

"Anisotropic DWR"


"Weighted Hessian"

Consider the same advection-diffusion problem

$$\mathbf{u}\cdot\nabla c-\nabla\cdot(D\nabla c)=f,\qquad c\in P1,$$

but extended to 3D in a cuboid domain. Again, QoI: tracer concentration c in "receiver region".

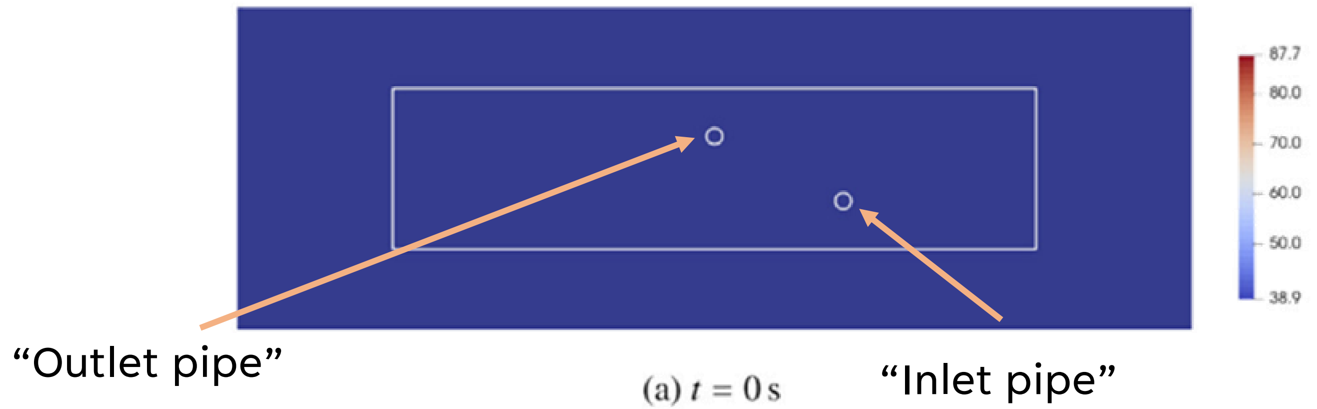


2D time-dependent desalination test case

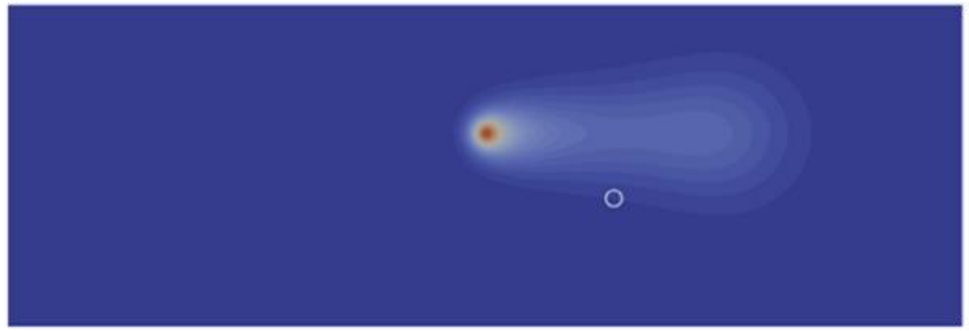
Time-dependent advection-diffusion,

$$\frac{\partial s}{\partial t} + \mathbf{u} \cdot \nabla s - \nabla \cdot (D \nabla s) = f, \qquad s \in P1.$$

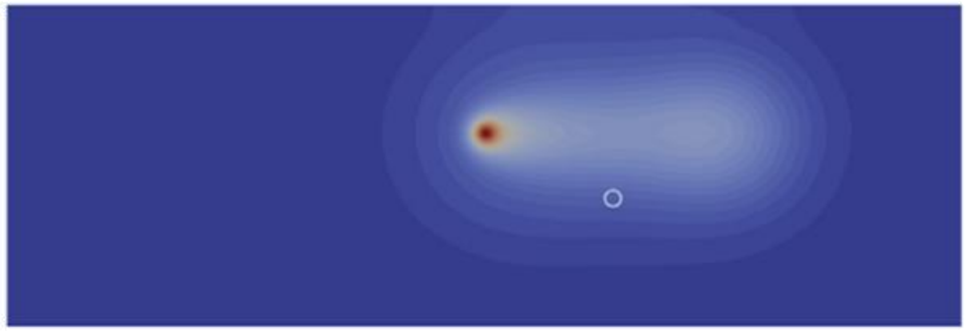
QoI: salinity s at inlet pipe.



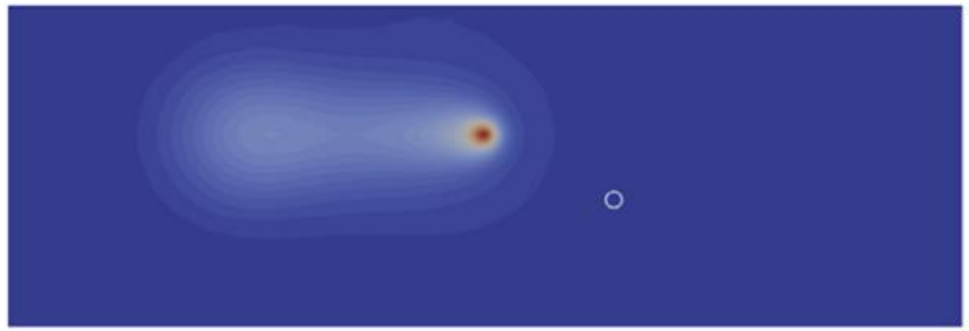
(a) $t = 0\,\mathrm{s}$



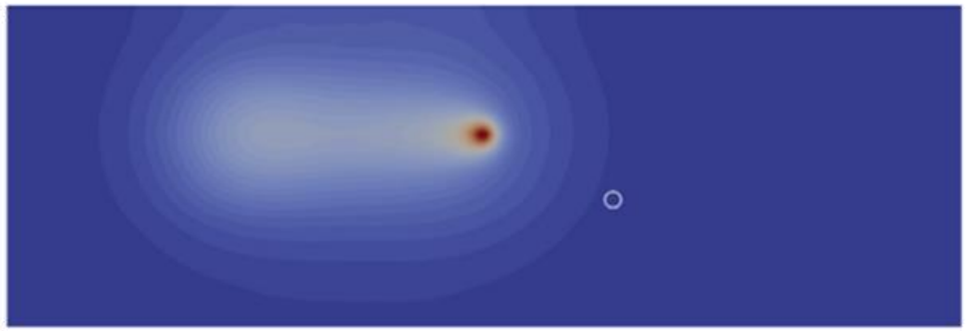
(b) $t = 0.5\,T_{\mathrm{tide}}$



(c) $t = 1.5\,T_{\mathrm{tide}}$

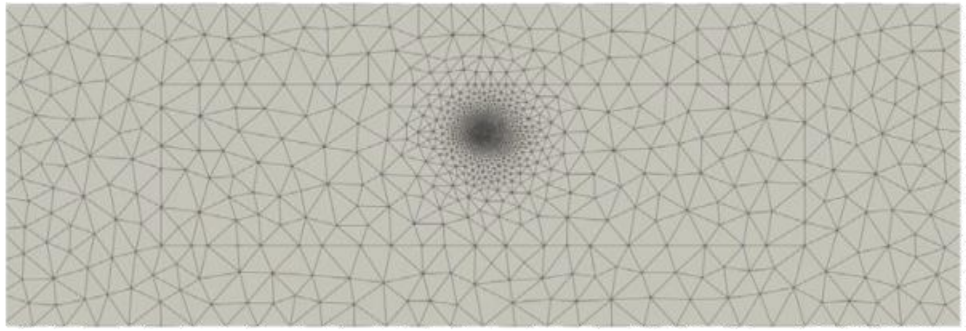


(d) $t = T_{\mathrm{tide}}$

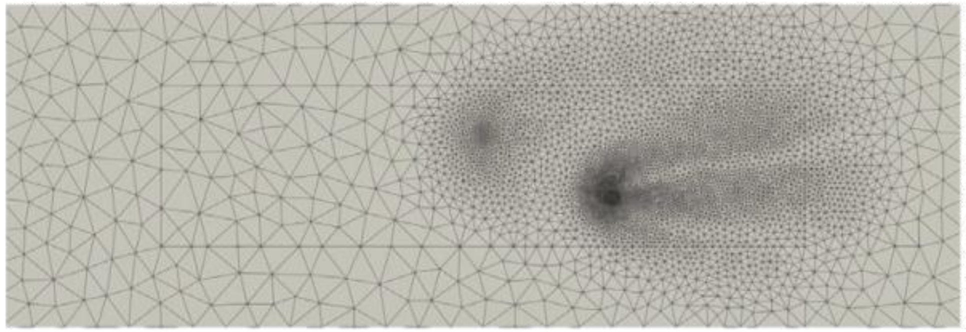


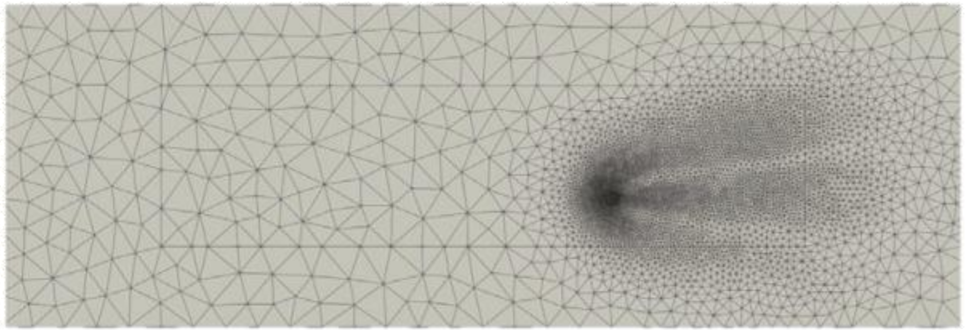
(e) $t = 2\,T_{\mathrm{tide}}$

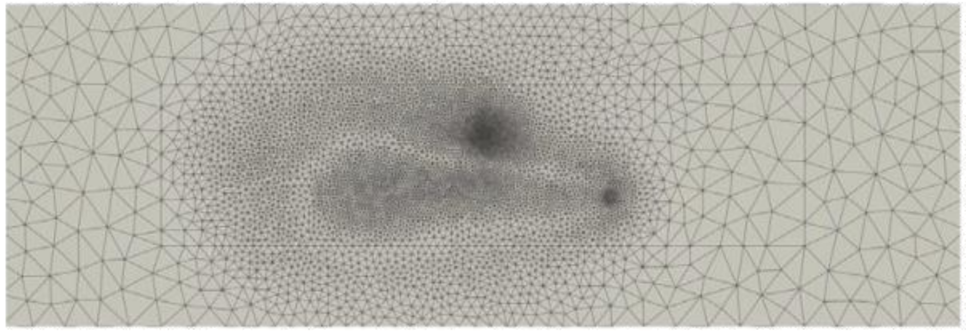
2D time-dependent desalination test case

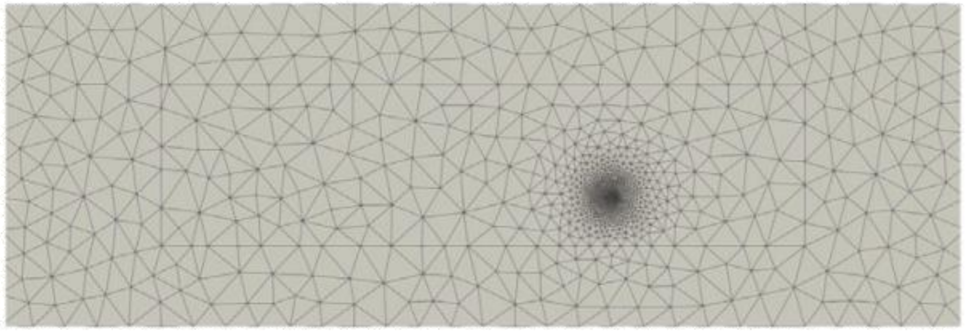

(a) $t = 0$

t/T_{tide}	Elements	Vertices	Mean AR	Max. AR
0.0	2,482	1,180	1.2	1.6
0.5	10,598	5,346	1.3	1.7
1.0	12,500	6,304	1.2	1.7
1.5	12,104	6,099	1.3	3.0
2.0	2,278	1,180	1.3	3.0


(b) $t = 0.5\,T_{\text{tide}}$

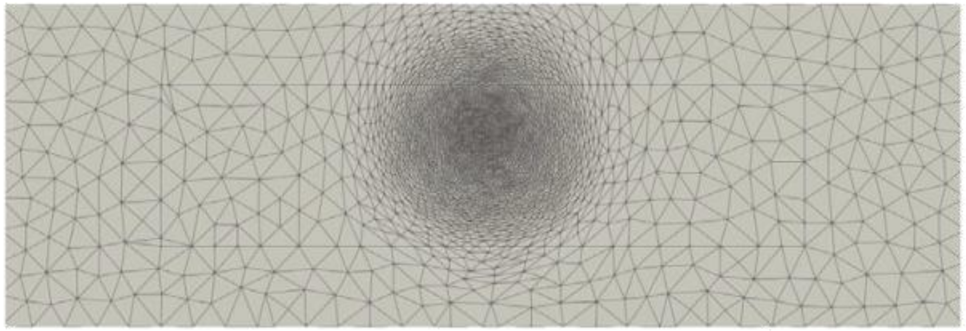

(c) $t = 1.5\,T_{\text{tide}}$


(d) $t = T_{\text{tide}}$


(e) $t = 2\,T_{\text{tide}}$

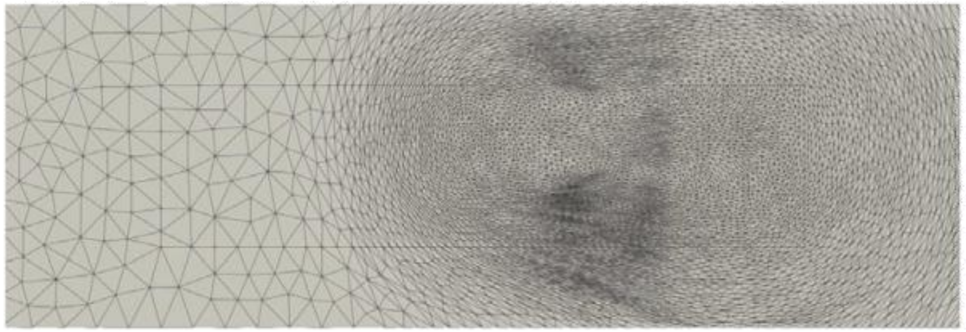
+

2D time-dependent desalination test case

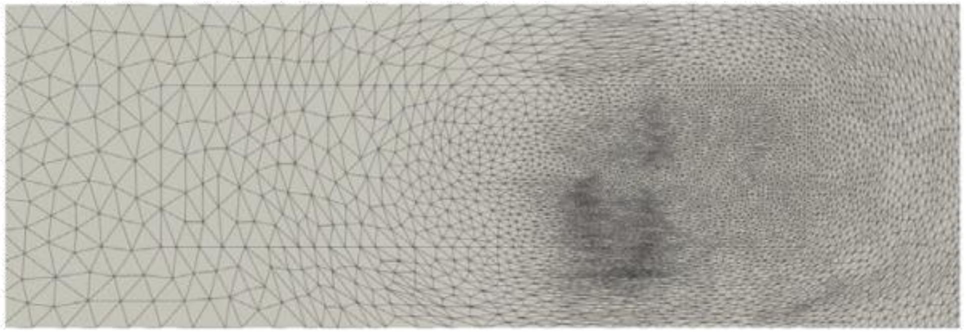


(a) $t = 0\,\mathrm{s}$

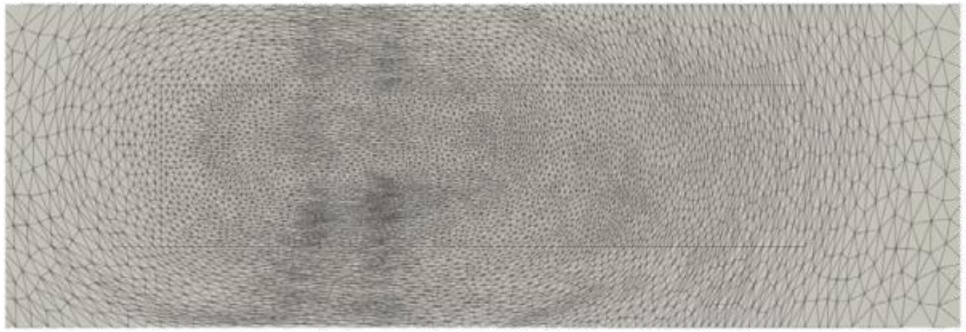
t/T_{tide}	Elements	Vertices	Mean AR	Max. AR
0.0	5,792	2,942	2.2	5.8
0.5	11,556	5,862	2.9	23.5
1.0	12,555	6,365	2.3	13.0
1.5	10,231	5,191	2.3	18.9
2.0	4,931	2,515	1.7	5.4



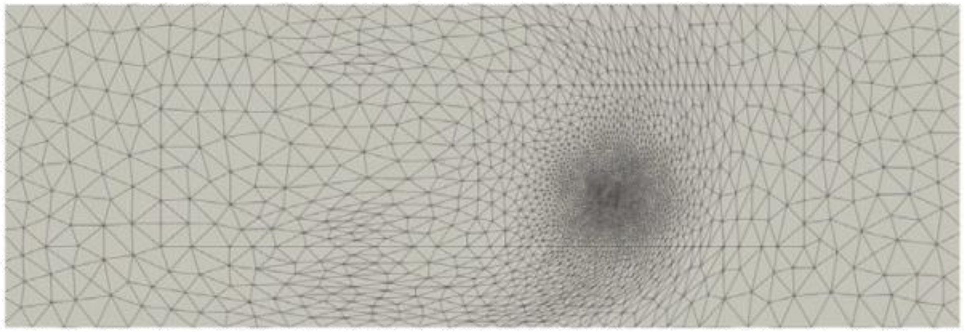
(b) $t = 0.5\,T_{\mathrm{tide}}$



(c) $t = 1.5\,T_{\mathrm{tide}}$

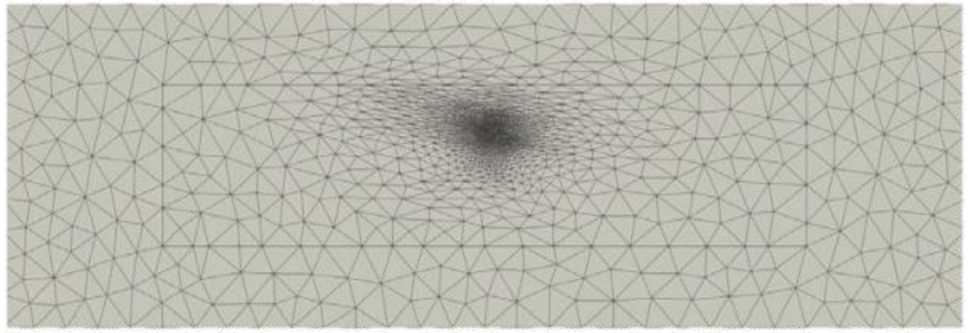


(d) $t = T_{\mathrm{tide}}$

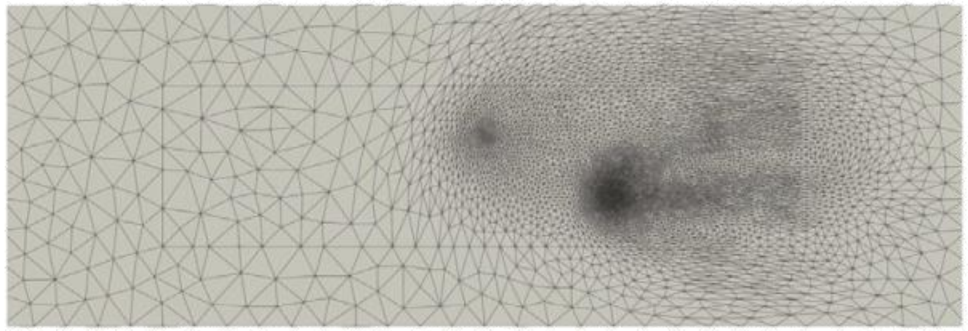


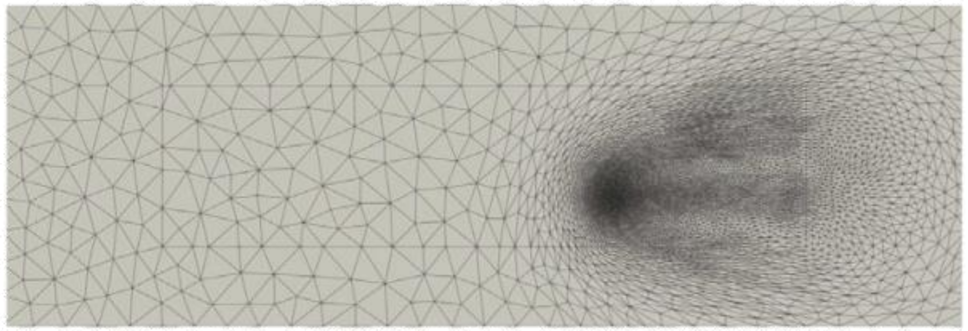
(e) $t = 2\,T_{\mathrm{tide}}$

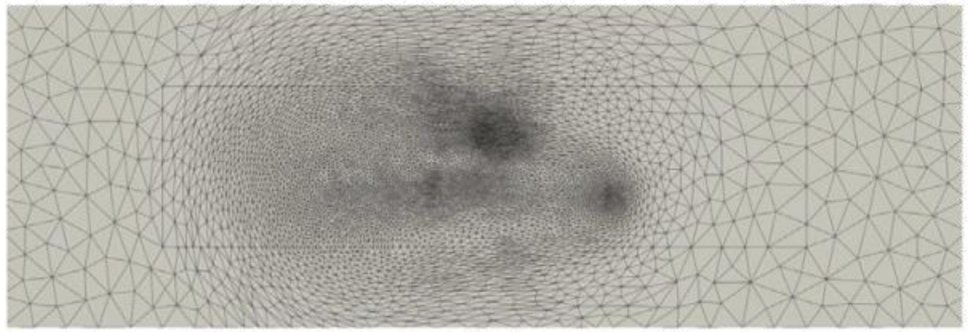
2D time-dependent desalination test case

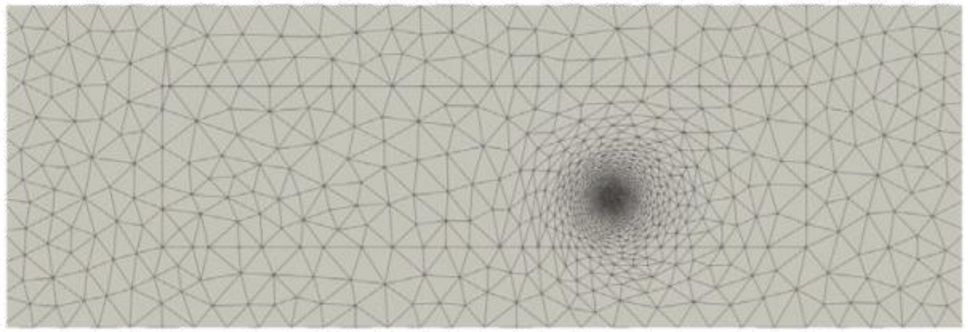

(a) $t = 0\,\text{s}$

t/T_{tide}	Elements	Vertices	Mean AR	Max. AR
0.0	2,426	1,254	2.7	11.3
0.5	12,473	6,280	1.8	11.1
1.0	12,243	6,169	2.1	14.0
1.5	13,523	6,809	2.0	17.7
2.0	2,708	1,395	1.5	3.3

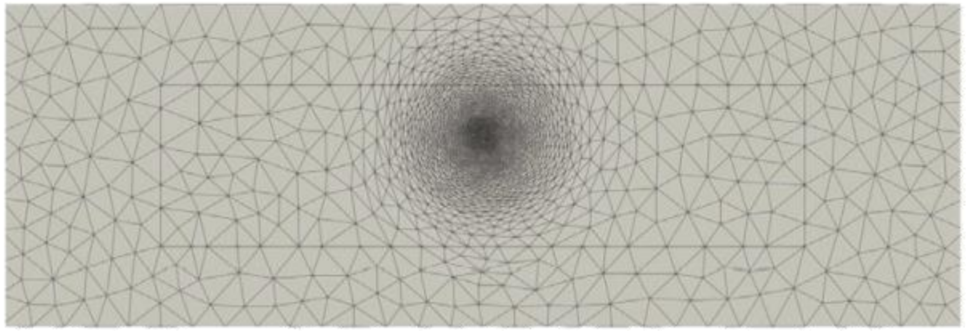

(b) $t = 0.5\,T_{\text{tide}}$


(c) $t = 1.5\,T_{\text{tide}}$

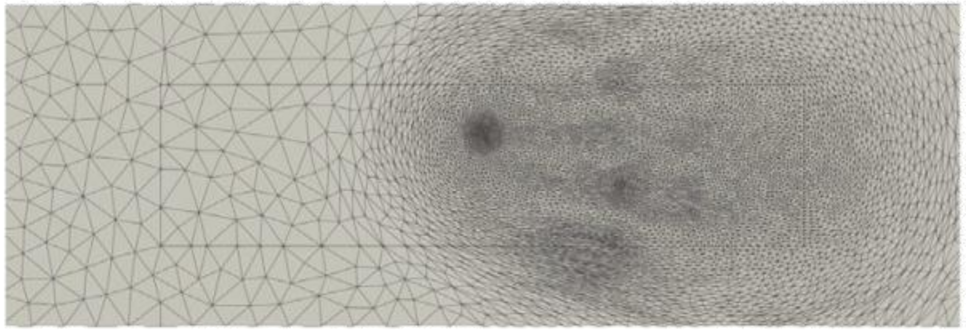

(d) $t = T_{\text{tide}}$

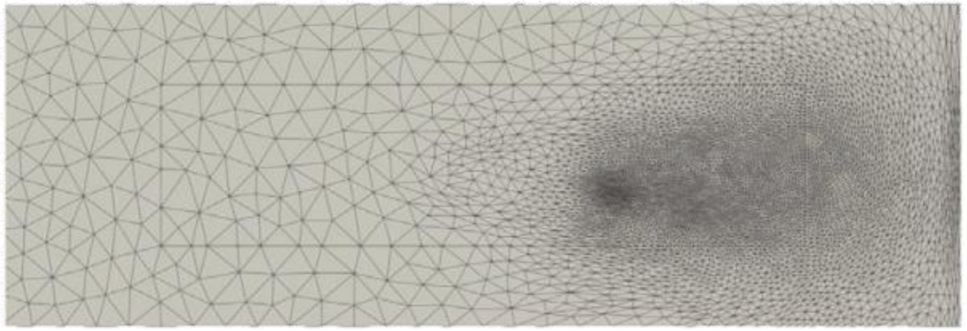

(e) $t = 2\,T_{\text{tide}}$

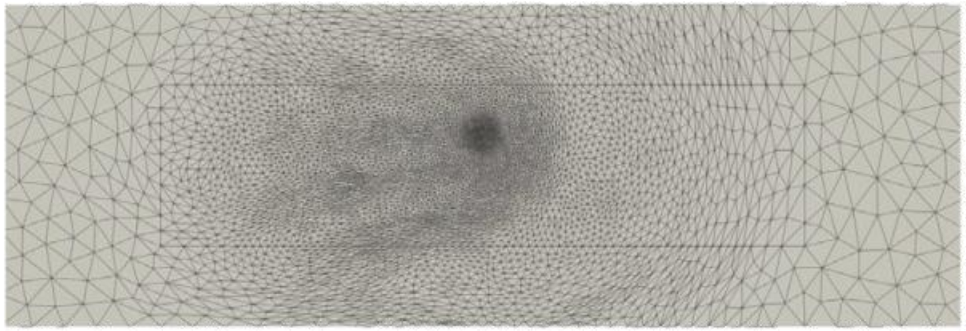
2D time-dependent desalination test case

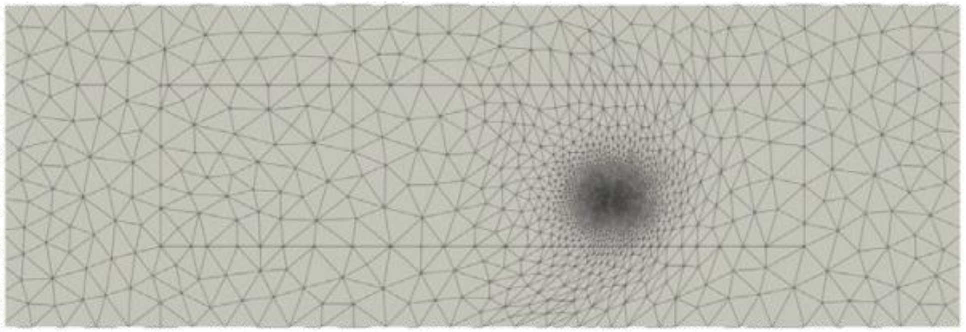

(a) $t = 0\,\mathrm{s}$

t/T_{tide}	Elements	Vertices	Mean AR	Max. AR
0.0	4,794	2,438	1.8	4.8
0.5	13,755	6,947	2.1	19.8
1.0	10,826	5,474	1.8	7.0
1.5	11,560	5,839	1.8	13.4
2.0	4,077	2,081	1.6	3.7

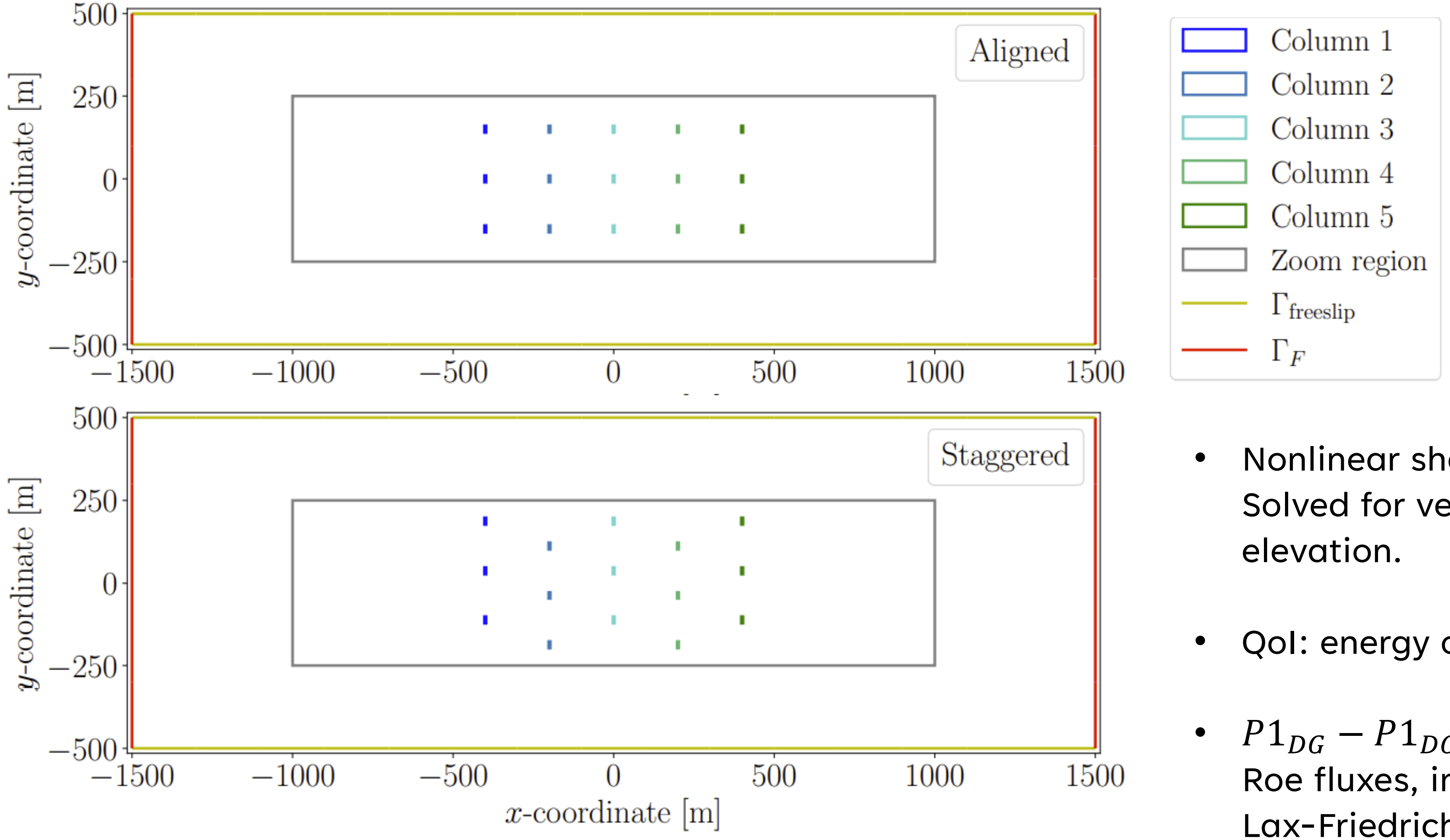

(b) $t = 0.5\,T_{\mathrm{tide}}$


(c) $t = 1.5\,T_{\mathrm{tide}}$


(d) $t = T_{\mathrm{tide}}$

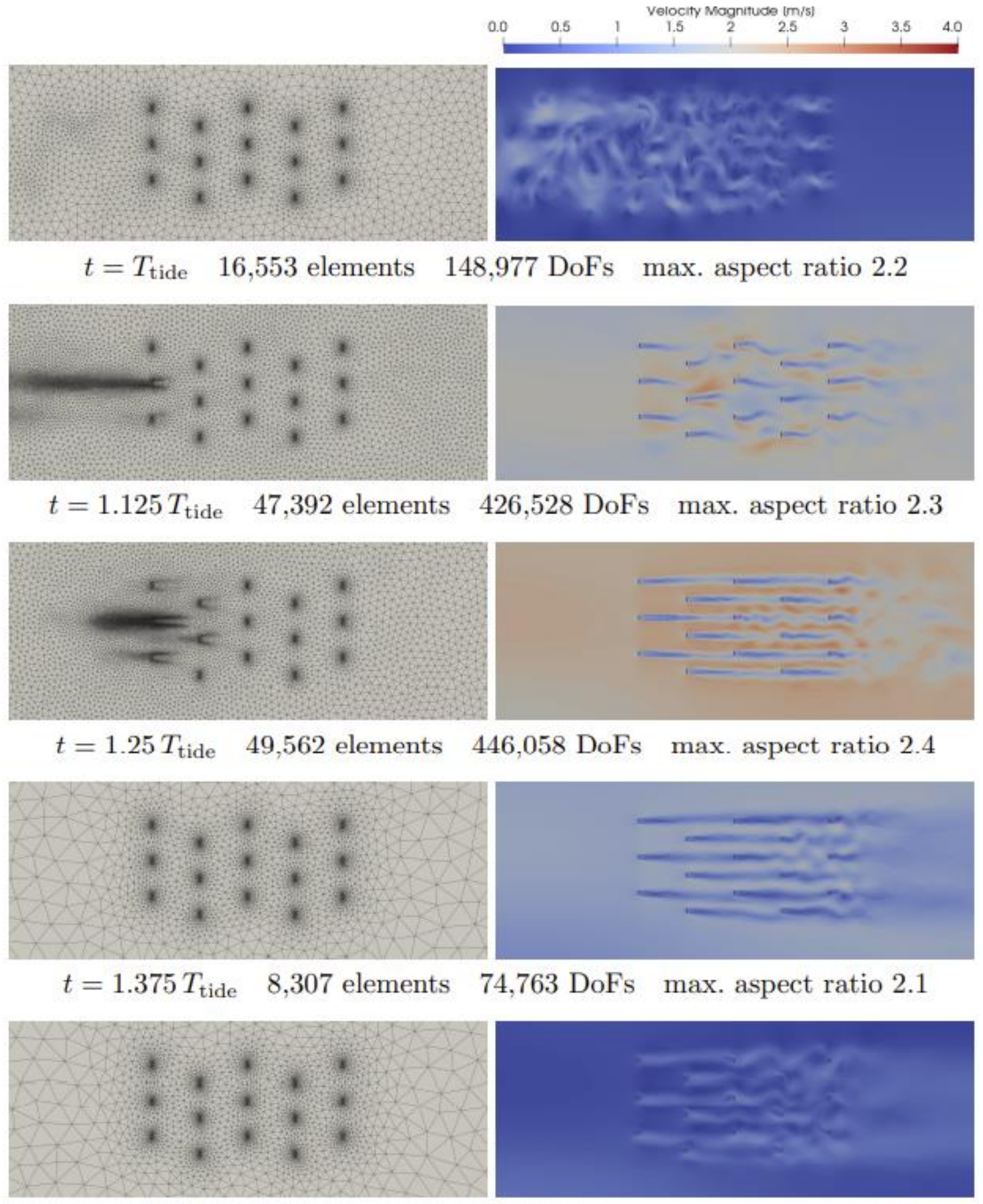

(e) $t = 2\,T_{\mathrm{tide}}$

2D time-dependent tidal farm test case



- Nonlinear shallow water equations. Solved for velocity and free surface elevation.
- QoI: energy output of tidal farm.
- $P1_{DG} - P1_{DG}$ spatial discretisation with Roe fluxes, interior penalty method and Lax-Friedrichs stabilisation.
- Crank-Nicolson timestepping.

2D time-dependent tidal farm test case



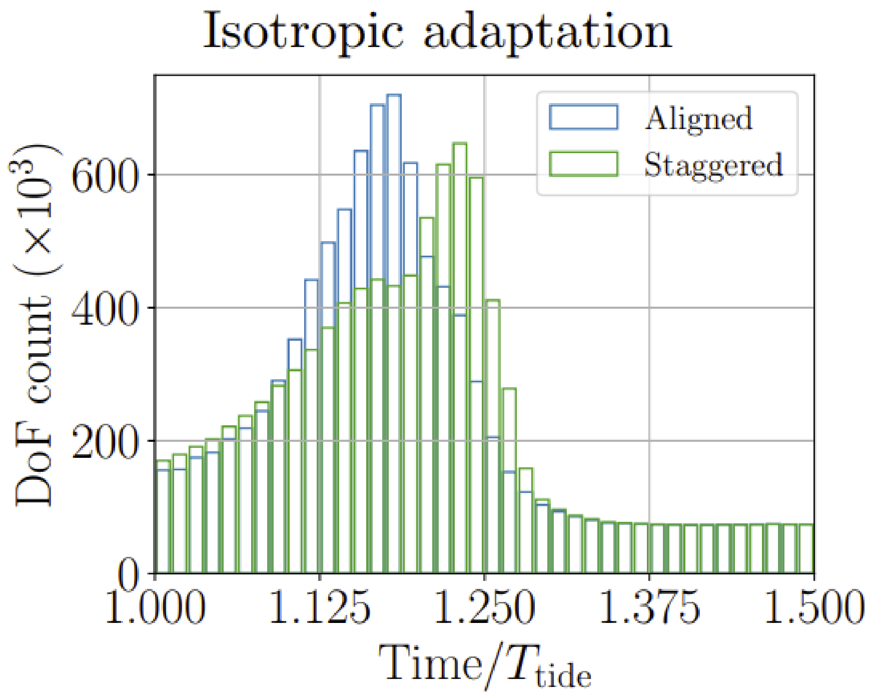
$t = T_{\mathrm{tide}}$ 16,553 elements 148,977 DoFs max. aspect ratio 2.2

$t = 1.125\,T_{\mathrm{tide}}$ 47,392 elements 426,528 DoFs max. aspect ratio 2.3

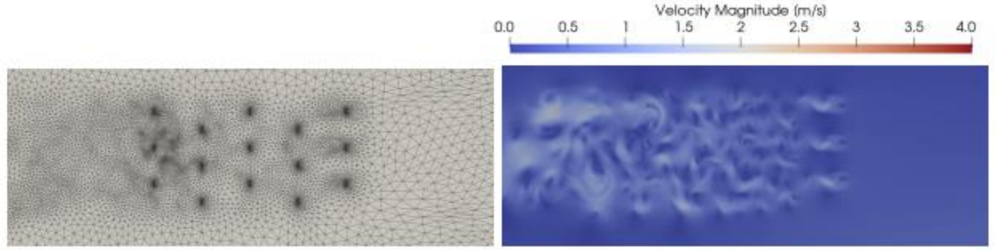
$t = 1.25\,T_{\mathrm{tide}}$ 49,562 elements 446,058 DoFs max. aspect ratio 2.4

$t = 1.375\,T_{\mathrm{tide}}$ 8,307 elements 74,763 DoFs max. aspect ratio 2.1

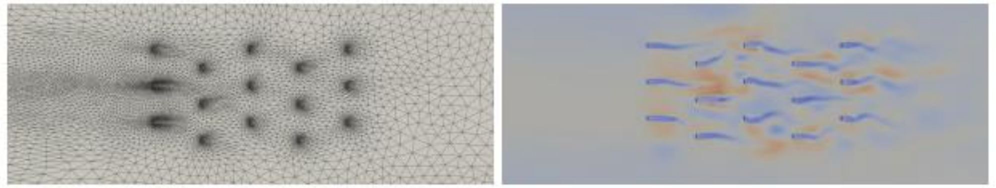
$t = 1.5\,T_{\mathrm{tide}}$ 8,282 elements 74,538 DoFs max. aspect ratio 2.4



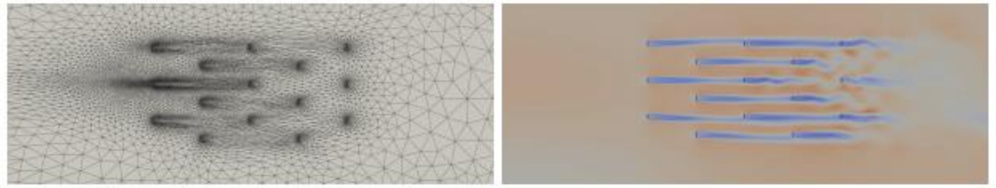
2D time-dependent tidal farm test case



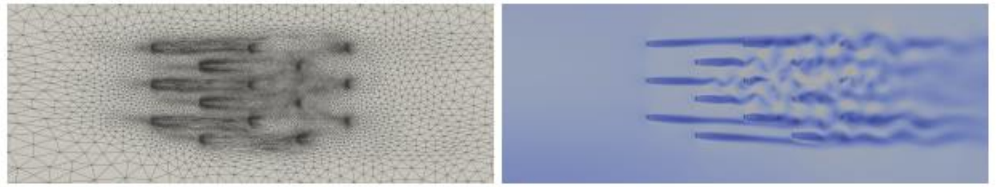
$t = T_{\text{tide}}$ 35,381 elements 318,429 DoFs max. aspect ratio 8.3



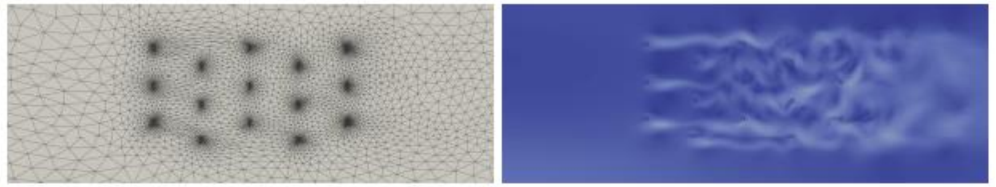
$t = 1.125\,T_{\text{tide}}$ 27,772 elements 249,948 DoFs max. aspect ratio 10.4



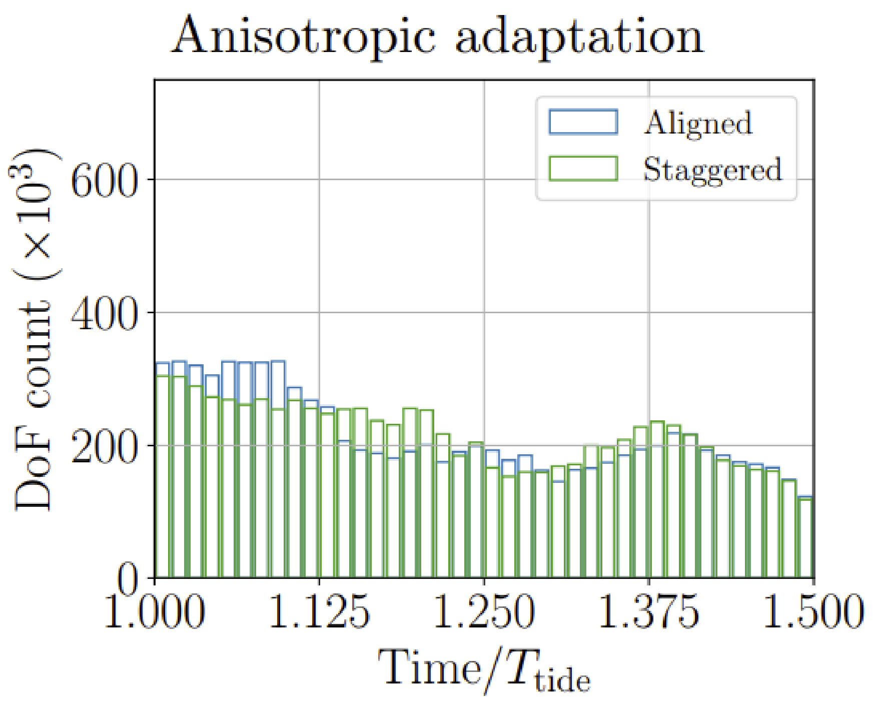
$t = 1.25\,T_{\text{tide}}$ 23,503 elements 211,527 DoFs max. aspect ratio 21.8



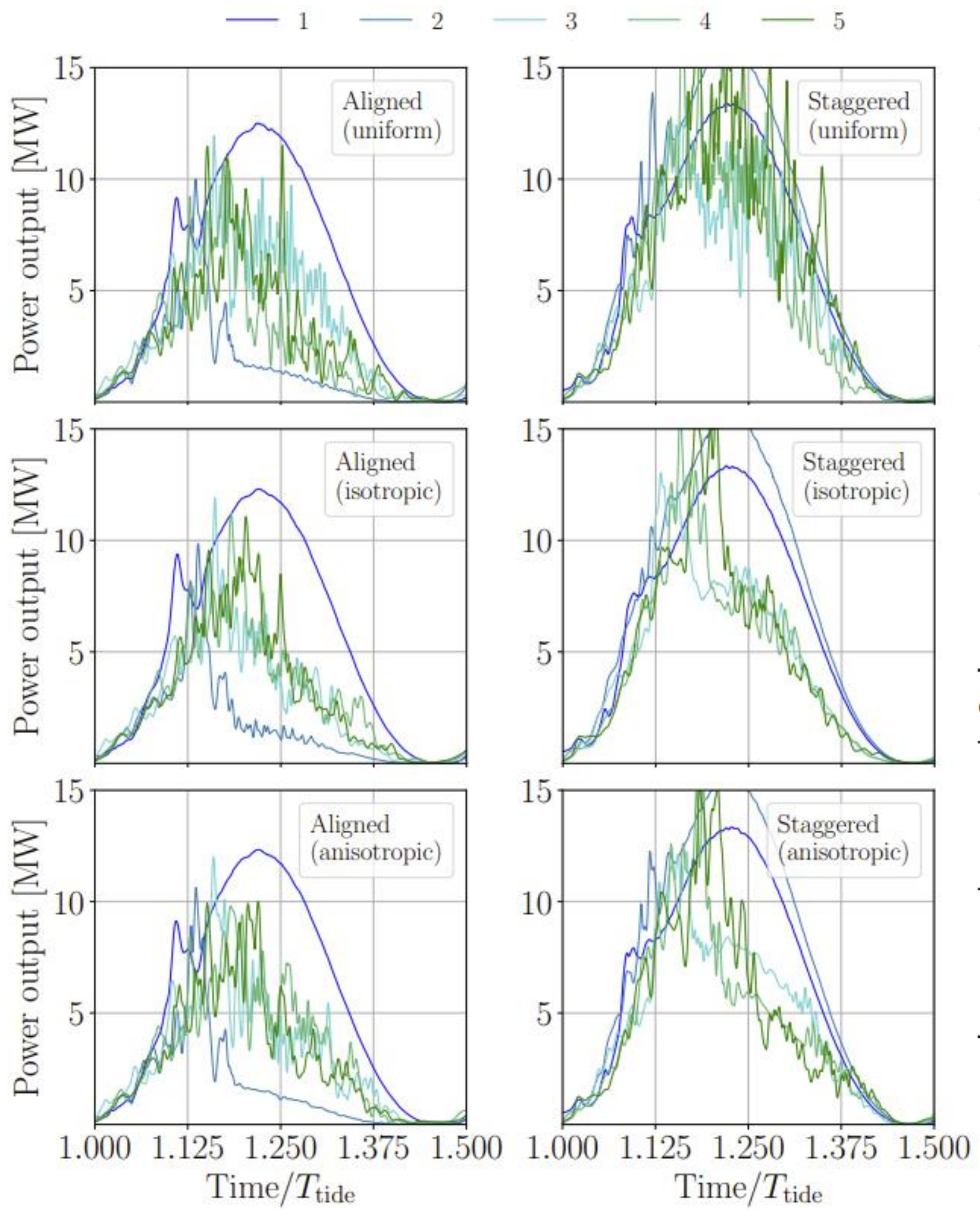
$t = 1.375\,T_{\text{tide}}$ 27,475 elements 247,275 DoFs max. aspect ratio 26.2



$t = 1.5\,T_{\text{tide}}$ 13,482 elements 121,338 DoFs max. aspect ratio 5.8



2D time-dependent tidal farm test case



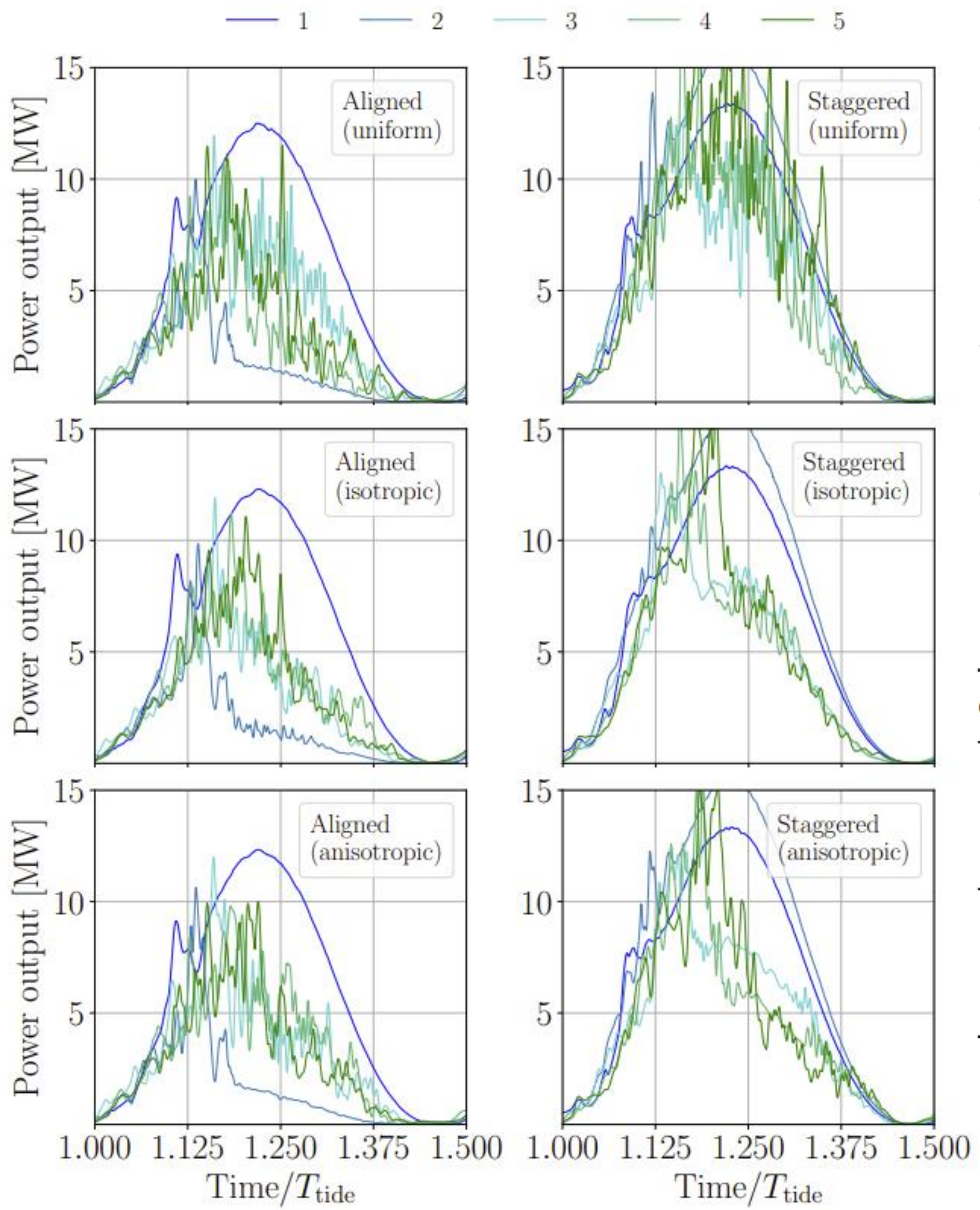
Configuration	Run	$\mathcal{C}^1$	$\mathcal{C}^2$	$\mathcal{C}^3$	$\mathcal{C}^4$	$\mathcal{C}^5$	$\mathcal{F}$
Aligned	Uniform mesh	2.75	0.82	1.76	1.27	1.48	8.07
	Isotropic	2.73	0.79	1.39	1.52	1.52	7.93
	Anisotropic	2.72	0.80	1.51	1.58	1.46	8.07
Staggered	Uniform mesh	3.07	3.61	2.49	2.47	2.98	14.62
	Isotropic	3.06	3.54	2.23	2.19	2.30	13.33
	Anisotropic	3.06	3.57	2.27	2.15	2.25	13.30

Energy output comparison

Configuration	Run	$\mathcal{C}^1$	$\mathcal{C}^2$	$\mathcal{C}^3$	$\mathcal{C}^4$	$\mathcal{C}^5$	$\mathcal{F}$
Aligned	Iso. – Fixed	-0.9%	-4.1%	-21.1%	19.7%	2.9%	-1.7%
	Aniso. – Fixed	-1.2%	-2.1%	-13.9%	24.5%	-1.1%	-0.0%
	Iso. – Aniso.	0.3%	-2.0%	-7.3%	-4.8%	4.0%	-1.7%
Staggered	Iso. – Fixed	-0.2%	-1.8%	-10.7%	-11.3%	-22.6%	-8.8%
	Aniso. – Fixed	-0.1%	-1.0%	-9.1%	-13.2%	-24.5%	-9.0%
	Iso. – Aniso.	-0.1%	-0.8%	-1.6%	2.0%	2.0%	0.2%

Relative differences in energy output

2D time-dependent tidal farm test case



Configuration	Run	$\mathcal{C}^1$	$\mathcal{C}^2$	$\mathcal{C}^3$	$\mathcal{C}^4$	$\mathcal{C}^5$	$\mathcal{F}$
Aligned	Uniform mesh	2.75	0.82	1.76	1.27	1.48	8.07
	Isotropic	2.73	0.79	1.39	1.52	1.52	7.93
	Anisotropic	2.72	0.80	1.51	1.58	1.46	8.07
Staggered	Uniform mesh	3.07	3.61	2.49	2.47	2.98	14.62
	Isotropic	3.06	3.54	2.23	2.19	2.30	13.33
	Anisotropic	3.06	3.57	2.27	2.15	2.25	13.30

Energy output comparison

Configuration	Run	$\mathcal{C}^1$	$\mathcal{C}^2$	$\mathcal{C}^3$	$\mathcal{C}^4$	$\mathcal{C}^5$	$\mathcal{F}$
Aligned	Iso. – Fixed	-0.9%	-4.1%	-21.1%	19.7%	2.9%	-1.7%
	Aniso. – Fixed	-1.2%	-2.1%	-13.9%	24.5%	-1.1%	-0.0%
	Iso. – Aniso.	0.3%	-2.0%	-7.3%	-4.8%	4.0%	-1.7%
Staggered	Iso. – Fixed	-0.2%	-1.8%	-10.7%	-11.3%	-22.6%	-8.8%
	Aniso. – Fixed	-0.1%	-1.0%	-9.1%	-13.2%	-24.5%	-9.0%
	Iso. – Aniso.	-0.1%	-0.8%	-1.6%	2.0%	2.0%	0.2%

Relative differences in energy output

Metric-based mesh adaptation

DMPlex supports mesh adaptation using the *Riemmanian metric framework*. The idea is to use a Riemannian metric space within the mesher. The metric space dictates how mesh resolution should be distributed across the domain. Using this information, the remesher transforms the mesh such that it is a *unit mesh* when viewed in the metric space. That is, the image of each of its elements under the mapping from Euclidean space into the metric space has edges of unit length.

One of the main advantages of metric-based mesh adaptation is that it allows for fully anisotropic remeshing. That is, it provides a means of controlling the shape and orientation of elements in the adapted mesh, as well as their size. This can be particularly useful for advection-dominated and directionally-dependent problems.

See [Ala10] for further details on metric-based anisotropic mesh adaptation.

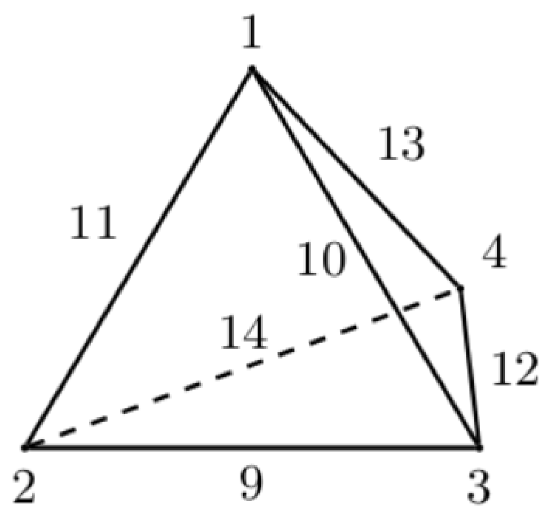
The two main ingredients for metric-based mesh adaptation are an input mesh (i.e. the DMPlex) and a Riemannian metric. The implementation in PETSc assumes that the metric is piecewise linear and continuous across elemental boundaries. Such an object can be created using the routine

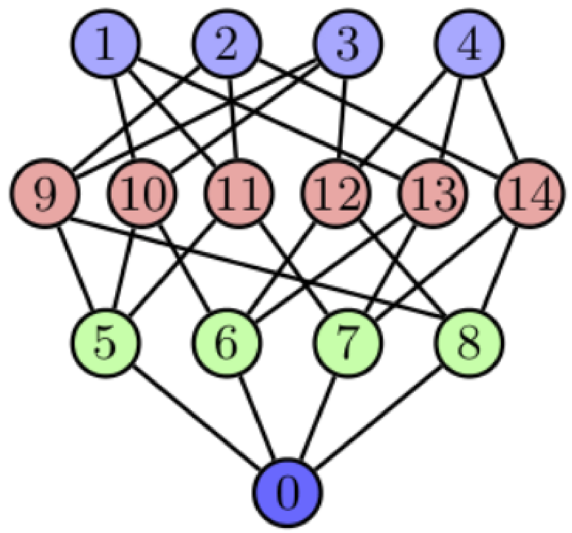
```
DMPlexMetricCreate(DM dm, PetscInt f, Vec *metric);
```

A metric must be symmetric positive-definite, so that distances may be properly defined. This

Accessing the PETSc mesh representation

Under the hood, Firedrake uses PETSc's DMPlex unstructured mesh representation. It uses a hierarchical approach, where entities of different dimension are put on different levels of the hierarchy. The single tetrahedral element shown on the left below may be interpreted using the graph representation on the right. Entities of dimension zero (vertices) are shown at the top. Entities of dimension one (edges) are shown on the next level down. Entities of dimension two (faces) are shown on the penultimate level and the (dimension three) element itself is on the bottom level. Edges in the graph indicate which entities own/are owned by others.





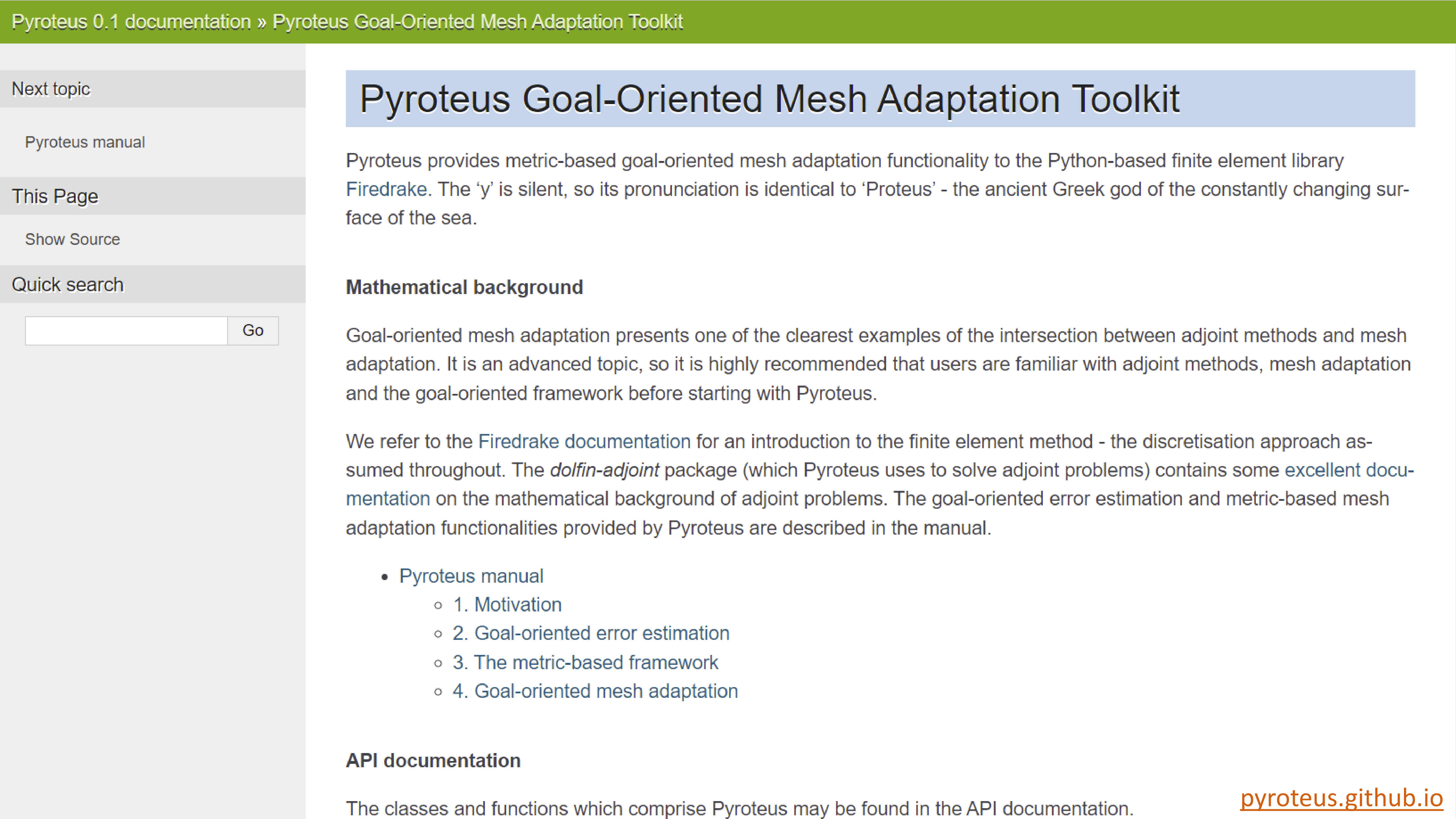
The DMPlex associated with a given `mesh` may be accessed via its `topology_dm` attribute:

```
plex = mesh.topology_dm
```

All entities in a DMPlex are given a unique number. The range of these numbers may be deduced using the method `plex.getDepthStratum`, whose only argument is the entity dimension sought. For example, 0 for vertices, 1 for edges, etc. Similarly, the method `plex.getHeightStratum` can be used for codimension access. For example, height 0 corresponds to cells. The hierarchical DMPlex structure may be traversed using other methods, such as `plex.getCone`, `plex.getSupport` and `plex.getTransitiveClosure`. See the Firedrake DMPlex paper and the PETSc manual for details.

If vertex coordinate information is to be accessed from the DMPlex then we must first establish a mapping between its numbering and the coordinates in the Firedrake mesh. This is done by establishing a 'section'. A section provides a way of associating data with the mesh - in this case,

firedrakeproject.org



Pyroteus Goal-Oriented Mesh Adaptation Toolkit

Pyroteus provides metric-based goal-oriented mesh adaptation functionality to the Python-based finite element library Firedrake. The 'y' is silent, so its pronunciation is identical to 'Proteus' - the ancient Greek god of the constantly changing surface of the sea.

Mathematical background

Goal-oriented mesh adaptation presents one of the clearest examples of the intersection between adjoint methods and mesh adaptation. It is an advanced topic, so it is highly recommended that users are familiar with adjoint methods, mesh adaptation and the goal-oriented framework before starting with Pyroteus.

We refer to the Firedrake documentation for an introduction to the finite element method - the discretisation approach assumed throughout. The *dolfin-adjoint* package (which Pyroteus uses to solve adjoint problems) contains some excellent documentation on the mathematical background of adjoint problems. The goal-oriented error estimation and metric-based mesh adaptation functionalities provided by Pyroteus are described in the manual.

API documentation

The classes and functions which comprise Pyroteus may be found in the API documentation.

pyroteus.github.io

3. Applications and documentation: Summary of achievements

- Tested functionality in two research papers.
- Metric-based mesh adaptation section in the PETSc manual.
- Extension of existing Firedrake documentation.
- Created a webpage for Pyroteus: pyroteus.github.io, including plenty of documentation and demos.

Papers

About eCSE work

– JGW, M. G. Knepley, N. Barral, M. D. Piggott, *Parallel metric-based mesh adaptation in PETSc using ParMmg*, 30th International Meshing Roundtable (2022), doi:10.48550/arXiv.2201.02806.

– JGW, A. K. Mohan, M. D. Piggott, *Pyroteus Goal-Oriented Error Estimation Toolkit*, 31st International Meshing Roundtable (in preparation).

Using eCSE work

– JGW, N. Barral, D. A. Ham, M. D. Piggott, *Goal-oriented error estimation and mesh adaptation for tracer transport modelling*, Computer-Aided Design 145 (2022): 103187, doi:10.1016/j.cad.2021.103187.

– JGW, A. Angeloudis, N. Barral, L. Mackie, S. C. Kramer, M. D. Piggott, *Tidal array modelling using goal-oriented mesh adaptation*, Journal of Ocean Engineering and Marine Energy (under review), doi:10.31223/X5H06B.

Thanks for listening!