Towards a DSEL for polytopal methods

Rémy Dubois

NEMESIS

New generation methods
for numerical simulations

Towards polytopal meshes in GMSH

Montpellier, 26–27 January 2026

cnrs
Funded by the European Union
erc
European Research Council

Outline

Funded by the European Union

Gradient reconstruction

- Consider

$$\underline{V}_h^k := \Big\{\underline{v}_h = \big((v_T)_{T\in\mathcal{T}_h}, (v_F)_{F\in\mathcal{F}_h}\big) \;:\; v_T \in \mathcal{P}^{k-1}(T) \text{ for all } T \in \mathcal{T}_h,\; v_F \in \mathcal{P}^k(F) \text{ for all } F \in \mathcal{F}_h,\; v_{\partial T}|_F = v_F \text{ for all } F \in \mathcal{F}_T\Big\}$$

- Let $G_T^k : \underline{V}_T^k \to \mathcal{P}^k(T)^3$ be s.t., for all $\underline{v}_T \in \underline{V}_T^k$,

$$\int_T G_T^k \underline{v}_T \cdot \tau = -\int_T v_T \operatorname{div}\tau + \int_{\partial T} v_{\partial T}(\tau \cdot n_{TF}) \quad \forall \tau \in \mathcal{P}^k(T)^d$$

Gradient reconstruction: before (HArDCore)

```
const Cell & T = *mesh().cell(iT);
size_t dim_Pkpo = PolynomialSpaceDimension<Cell>::Poly(degree()+1);
MonomialCellIntegralsType int_mono_2kp2 = IntegrateCellMonomials(T, 2*degree()+2);
Eigen::MatrixXd MGT = GramMatrix(T, *cellBases(iT).Polykd, int_mono_2kp2);
Eigen::MatrixXd BGT = Eigen::MatrixXd::Zero(cellBases(iT).Polykd->dimension(),
dimensionCell(iT));
for (size_t iF = 0; iF < T.n_faces(); iF++) {
  const Face & F = *T.face(iF);
  DecomposePoly dec(F, MonomialScalarBasisFace(F, degree()));
  auto PkdT_dot_nTF_nodes =
scalar_product(evaluate_quad<Function>::compute(*cellBases(iT).Polykd, dec.get_nodes()),
T.face_normal(iF));
  auto PkdT_dot_nTF_family_PkF = dec.family(PkdT_dot_nTF_nodes);
  Eigen::MatrixXd PF = extendOperator(T, F, Eigen::MatrixXd::Identity(dimensionFace(F),
dimensionFace(F)));
  MonomialFaceIntegralsType int_mono_2k_F = IntegrateFaceMonomials(F, 2*degree());
  BGT += GramMatrix(F, PkdT_dot_nTF_family_PkF, *faceBases(F).Polyk, int_mono_2k_F) * PF;
}

DivergenceBasis<HHOSpace::PolydBasisCellType> div_Pkd_basis(*cellBases(iT).Polykd);

BGT.rightCols(numLocalDofsCell()) -= GramMatrix(T, div_Pkd_basis, *cellBases(iT).Polyk,

int_mono_2kp2);

Eigen::MatrixXd GT = MGT.ldlt().solve(BGT);
```

Gradient reconstruction: after (Prometheus)

```
DiscreteSpace Vh(*mesh_ptr, use_threads);
Vh.addScalarPolyCell("v_T", k-1);
Vh.addScalarPolyFace("v_partialT", k);
auto v_T = Vh.ScalarPolyCell("v_T");
auto v_dT = Vh.ScalarPolyFace("v_partialT");

for (auto T : mesh_ptr->get_cells();) {
  auto test_space = TestSpace(*T);
  test_space.addVectorPolyCell("tau", k);
  auto TT = TrialTest(Vh, test_space, *T);
  auto tau = test_space.VectorPolyCell("tau");

  PolynomialFamily Pk3(*T, k, Vector);
  auto GT = reconstruction( Vh, Pk3 );

  auto lhs = integrateCell( dot(GT(v_t), tau), TT);
  auto rhs = integrateBoundary(v_dT*dot(tau,normalT), TT)
           - integrateCell(v_T*divergence(tau), TT);
  GT = solve( lhs, rhs );       }
```

Gradient reconstruction: after (Prometheus)

$$\underline{V}_h^k := \Big\{ \underline{v}_h = \big((v_T)_{T\in\mathcal{T}_h}, (v_F)_{F\in\mathcal{F}_h}\big) \;:\; v_T \in \mathcal{P}^{k-1}(T) \text{ for all } T \in \mathcal{T}_h,$$
$$v_F \in \mathcal{P}^{k}(F) \text{ for all } F \in \mathcal{F}_h$$
$$v_{\partial T}|_F = v_F \text{ for all } F \in \mathcal{F}_T \Big\}$$

```
DiscreteSpace Vh(*mesh_ptr, use_threads);
Vh.addScalarPolyCell("v_T", k-1);
Vh.addScalarPolyFace("v_partialT", k);
auto v_T = Vh.ScalarPolyCell("v_T");
auto v_dT = Vh.ScalarPolyFace("v_partialT");
```

Gradient reconstruction: after (Prometheus)

$$\forall \tau \in \mathcal{P}^k(T)^d$$

```
auto test_space = TestSpace(*T);

test_space.addVectorPolyCell("tau", k);

auto tau = test_space.VectorPolyCell("tau");
auto TT = TrialTest(Vh, test_space, *T);
```

Gradient reconstruction: after (Prometheus)

$$G_T^k : \underline{V}_T^k \to \mathcal{P}^k(T)^3$$

```
PolynomialFamily Pk3(*T, k, Vector);

auto GT = reconstruction( Vh, Pk3 );
```

Gradient reconstruction: after (Prometheus)

$$\int_T G_T^k \underline{v}_T \cdot \tau = -\int_T v_T \operatorname{div} \tau + \int_{\partial T} v_{\partial T} (\tau \cdot n_{TF})$$

```
auto lhs = integrateCell( dot(GT(v_t), tau), TT);

auto rhs = - integrateCell(v_T*divergence(tau), TT)
           + integrateBoundary(v_dT*dot(tau,normalT), TT);

GT = solve( lhs, rhs );
```

Potential reconstruction

- Space: $\underline{V}_T^k = \{\underline{v}_T = (v_T, (v_F)_{F\in\mathcal{F}_T}) \,:\, v_T \in \mathcal{P}^{k-1}(T)\,,\ v_F \in \mathcal{P}^k(F)\}$.
- Potential reconstruction ($k \geq 1$):

$$\int_T \nabla p_T^{k+1}\underline{v}_T \cdot \nabla w = -\int_T v_T \Delta w + \int_{\partial T} v_{\partial T}(\nabla w \cdot n_{TF}) \quad \forall w \in \mathcal{P}^{k+1}(T),$$

Recast as: for all $w \in \mathcal{P}^{k+1}(T)$,

$$(\nabla p_T^{k+1}\underline{v}_T, \nabla w)_T + (p_T^{k+1}\underline{v}_T, 1)_T(w,1)_T = -(v_T, \Delta w)_T + (v_{\partial T}, \nabla w \cdot n_{TF})_{\partial T} + (v_T, 1)_T(w,1)_T.$$

Potential reconstruction: implementation

```
DiscreteSpace Vh(*mesh_ptr, use_threads);
Vh.addScalarPolyCell("v_T", k-1);
Vh.addScalarPolyFace("v_partialT", k);
auto v_T = Vh.ScalarPolyCell("v_T");
auto v_dT = Vh.ScalarPolyFace("v_patrialT");

for (auto T : mesh_ptr->get_cells();) {
  //test space
  auto test_space = TestSpace(*T);
  test_space.addScalarPolyCell("w", k+1);
  auto w = test_space.ScalarPolyCell("w");
  auto TT = TestTrial(Vh, test_space, *T);

  // Instantiation of local operator
  PolynomialFamily Pkpo(*T, k+1, Scalar);
  auto PT = reconstruction( Vh, Pkpo );

  // Definition of local operator
  auto RHS_PT = integrateBoundary( scal( v_dT, dot( gradient(w), normalT) ), TT)
              + Gram(integrate_T(v_T), integrate_T(w), TT)
              - integrateCell( v_T * divergence(gradient(w)) , TT);
  auto LHS_PT = integrateCell( scal(gradient(PT) , gradient(w) ) , TT)
              + Gram( integrate_T(PT), integrate_T(w), TT);

  PT = solve( LHS_PT, RHS_PT );              }
```

Potential reconstruction: implementation

$$\underline{V}_T^k = \{\underline{v}_T = (v_T, (v_F)_{F\in\mathcal{F}_T}) \,:\, v_T \in \mathcal{P}^{k-1}(T)\,,\; v_F \in \mathcal{P}^k(F)\}.$$

```
DiscreteSpace Vh(*mesh_ptr, use_threads);
Vh.addScalarPolyCell("v_T", k-1);
Vh.addScalarPolyFace("v_F", k);
auto v_T = Vh.ScalarPolyCell("v_T");
auto v_dT = Vh.ScalarPolyFace("v_F");
```

Potential reconstruction: implementation

$$\forall w \in \mathcal{P}^{k+1}(T)$$

```
auto test_space = TestSpace(*T);
test_space.addScalarPolyCell("w", k+1);
auto w = test_space.ScalarPolyCell("w");
auto TT = TestTrial(Vh, test_space, *T);
```

Potential reconstruction: implementation

$$p_T^{k+1} : \underline{V}_T^k \to \mathcal{P}^{k+1}(T)$$

```
PolynomialFamily Pkpo(*T, k+1, Scalar);
auto PT = reconstruction( Vh, Pkpo );
```

Potential reconstruction: implementation

$$(\nabla p_T^{k+1}\underline{v}_T, \nabla w)_T + (p_T^{k+1}\underline{v}_T, 1)_T(w,1)_T$$
$$= -(v_T, \Delta w)_T + (v_{\partial T}, \nabla w \cdot n_{TF})_{\partial T} + (v_T, 1)_T(w,1)_T.$$

```
auto LHS_PT = integrateCell( gradient(PT)*gradient(w), TT)
            + Gram( integrate_T(PT), integrate_T(w), TT);

auto RHS_PT = integrateBoundary( v_dT*dot(gradient(w),normalT) ), TT)
            + Gram(integrate_T(v_T), integrate_T(w), TT)
            - integrateCell( v_T * divergence(gradient(w)) , TT);

PT = solve( LHS_PT, RHS_PT );
```

Potential reconstruction 2

- Space: $\underline{V}_T^k = \{\underline{v}_T = (v_T, (v_F)_{F\in\mathcal{F}_T}) \ : \ v_T \in \mathcal{P}^{k-1}(T)\,,\ v_F \in \mathcal{P}^k(F)\}$.
- Potential reconstruction $(k \geq 1)$:

$$\mathcal{P}_0^{k+1}(T) = \{\mathcal{P}^{k+1}(T) : \int_T w = 0\}$$

$$\forall w \in \mathcal{P}_O^{k+1}(T)\,, \lambda \in \mathcal{P}^0(T)$$

$$\int_T \nabla p_T^{k+1}\underline{v}_T \cdot \nabla w + \int_T p_T^{k+1}\underline{v}_T\lambda = -\int_T v_T\Delta w + \int_{\partial T} v_{\partial T}\,(\nabla w \cdot n_{TF}) + \int_T \underline{v}_T\lambda$$

Potential reconstruction: implementation 2

```
DiscreteSpace Vh(*mesh_ptr, use_threads);
Vh.addScalarPolyCell("v_T", k-1);
Vh.addScalarPolyFace("v_dT", k);
auto v_T = Vh.ScalarPolyCell("v_T");
auto v_dT = Vh.ScalarPolyFace("v_partialT");

for (auto T : mesh_ptr->get_cells();) {
  auto test_space = TestSpace(*T);
  std::function< ScalarPolyCellType (const Cell &, const size_t,
                 const scalarGenerator &) > constructPolynomial
  = [](const Cell & T, const size_t k, const scalarGenerator & gen)
     { return zeroAveragePoly(T, k+1, gen); };
  test_space.addPolyFamily(constructPolynomial, "w", k);
  test_space.addScalarPolyCell("lambda", 0);
  auto w = test_space.PolyCell("w");
  auto lambda = PolyCell("lambda");

  auto TT = TrialTest(Vh, test_space, *T);
  PolynomialFamily Pk3(*T, k+1, Vector);
  auto PT = reconstruction( Vh, Pk3 );
  auto lhs = integrateCell( dot(gradient(PT(v_T)), gradient(w)), TT)
           + integrateCell(dot(PT(v_T),lambda));
  auto rhs = integrateBoundary(v_dT*dot(gradient(w),normalT), TT)
           - integrateCell(v_T*divergence(gradient(w))
           + integrateCell(v_T*lambda), TT);
  PT = solve( lhs, rhs );
}
```

Outline

Funded by the European Union

Frontend for polynomial bases

PolynomialFamily(element, degree, Type, [generator], [orth]);

- ★ element: a mesh element (cell, face, edge, vertex)
- ★ degree: polynomial degree
- ★ Type: either Scalar, Vector, Matrix or user defined generator
- ★ generator: double, Eigen::VectorXd or Eigen::MatrixXd (optional parameter)
- ★ orth: perfoms orthonormalization or not (optional parameter)

Simplification of basis and integration

	HC2D	*HC3D*	*Prometheus*
basis	2 files, 3421 lines	2 files, 3418 lines	2 files, 1190 lines
integration	9 files, 1888 lines	14 files, 3622 lines	2 files, 518 lines

Features

- Methods acting on polynomial families return a polynomial family

```
auto PF2 = gradient(PF1);
```

Let PF1 being of type PolynomialFamily(T, k, Scalar)
Then PF2 is of type PolynomialFamily(T, k-1, Vector)

`gradient(PF)`, `divergence(PF)`, `curl(PF)`, `sum(PF1, PF2)`, `applyLinearTransform(PF, std::function)` (linear map acting on generators)

It allows to concatenate methods.

- Canonical methods

$$\mathcal{R}^{c,\ell}(T) = (\boldsymbol{x} - \boldsymbol{x}_T)\mathcal{P}^{\ell-1}(T), \quad \mathcal{G}^{c,\ell}(T) = (\boldsymbol{x} - \boldsymbol{x}_T) \times \mathcal{P}^{\ell-1}(T)^3.$$

`koszulGrad(T, K)`, `koszulCurl(T, K)`

Outline

Frontend for trial space

Discrete spaces are meant to create/store polynomial families defined on every element of a mesh (type of the polynomial family define the element). They will hold methods to retrieve local/global DOFs (hidden within DSEL's expression).

- Example

```
DiscreteSpace Vh(*mesh_ptr, use_threads);
Vh.addScalarPolyCell("v_T", degree);
Vh.addScalarPolyEdge("v_h", degree);
```

- Methods to create polynomial families

```
addScalarPolyCell("name", degree)
addVectorPolyCell("name", degree)
addMatrixPolyCell("name", degree)
addScalarPolyFace("name", degree)
...
addPolyFamily(std::function, "name", degree)
```

Frontend for trial space

- Koszul complements:

$$\mathcal{R}^{c,\ell}(T) = (\boldsymbol{x} - \boldsymbol{x}_T)\mathcal{P}^{\ell-1}(T)\,, \quad \mathcal{G}^{c,\ell}(T) = (\boldsymbol{x} - \boldsymbol{x}_T) \times \mathcal{P}^{\ell-1}(T)^3.$$

- Implementation

```
std::function< VectorPolyCellType (const Cell &, const size_t) >
constructKoszulGrad
= [](const Cell & T, const size_t K) { return koszulGrad(T, K); };
Vh.addPolyFamily(constructPolynomial, "Gck", degree);

std::function< ScalarPolyCellType (const Cell &, const size_t) >
constructKoszulCurl
= [](const Cell & T, const size_t K) {return koszulCurl(T, K); };
Vh.addPolyFamily(constructPolynomial, "Rck", degree);
```

Frontend for test space

Test spaces are meant to create/store polynomial families defined on a specific element and all sub-elements.

- Instantiation

```
auto test_space = TestSpace( element );
```

Element is a mesh element (Cell, Face, Edge, Vertex).

- Methods (similar ones as DiscreteSpace)

```
test_space.addVectorPolyCell("phi", degree);
```

The element of a TestSpace can restrict the methods.

Outline

Funded by the European Union

Purpose

- Let's consider:

$$\int_{\partial T} v_{\partial T}(\tau \cdot n_{TF})$$

  If we wish to stick closer to mathematical form and to translate it as:

```
auto rhs = integrateBoundary(v_dT*dot(tau,normalT), TT);
```

  or:

```
auto rhs = integrateBoundary(dot(tau,normalT)*v_dT, TT);
```

- Prior calculation of `v_F*dot(tau,normalT)` would not be meaningful as no specific element is provided (`v_F` and `normalT` are undefined ...).
- Lazy evaluation and using boost::proto as expression parser were selected to tackle it.

Expressions

- $$\int_T v_T \ \tau$$

```
auto ret = integrateCell(v_T*tau, TT);
```

- $$\int_{\partial T} v_{\partial T}(\tau \cdot n_{TF})$$

```
auto ret = integrateBoundary(v_dT*dot(tau,normalT), TT);
```

- $$(\int_T v_T)(\int_T w), (\int_F v_F)(\int_F w)$$

```
auto ret = Gram(integrateT(v_T),integrateT(w), TT);
auto ret = Gram(integrateF(v_F),integrateF(w), TT);
```

- Return is automatically assemble with respect to local DOFs.

Local operators

- Instantiation:

```
auto Op = reconstruction(domain, codomain);
```

  ⋆ `domain` is a discrete space

  ⋆ `codomain` is a polynomial family

- Definition:

```
auto op = solve( lhs, rhs );
```

  With rhs and lhs being either results of Prometheus expressions or Eigen matrix.

- Direct use within Prometheus expressions:

```
auto ret = integrateCell( gradient(Op) * w, TT);
```

Local operators: example

Let $G_T^k : \underline{V}_T^k \to \mathcal{P}^k(T)^3$ be s.t., for all $\underline{v}_T \in \underline{V}_T^k$,

$$\int_T G_T^k \underline{v}_T \cdot \tau = -\int_T v_T \operatorname{div} \tau + \int_{\partial T} v_{\partial T}(\tau \cdot n_{TF}) \quad \forall \tau \in \mathcal{P}^k(T)^d$$

```
//Instantiation
PolynomialFamily Pk3(*T, degree, Vector);
auto GT = reconstruction( Vh, Pk3 );
//Definition
auto lhs = integrateCell( dot(GT(v_T), tau), TT);
auto rhs = integrateBoundary( v_dT*dot(tau,normalT), TT)
         - integrateCell( v_T*divergence(tau) , TT);
GT = solve( lhs, rhs );


//Usage example (once defined)
auto ret = integrateCell( gradient(GT(v_T)) * gradient(tau)), TT);
```

Funded by the European Union (ERC Synergy, NEMESIS, project number 101115663). Views and opinions expressed are however those of the authors only and do not necessarily reflect those of the European Union or the European Research Council Executive Agency. Neither the European Union nor the granting authority can be held responsible for them.

Thank you for your attention!