The Skeleton of Modern C++ - Part 2

Matteo Cicuttin
CERMICS
École Nationale des Ponts et Chaussées
Marne-la-Vallée

Nicola Gigante
Università di Udine

INFN-CNAF, Bologna, July 7th, 2017

Intro

Context

Discontinuous Skeletal methods approximate solutions of BVPs by

- using polynomials that are discontinuous in the mesh skeleton $\implies$ “discontinuous”
- attaching unknowns to mesh faces $\implies$ “skeletal”

DiSk methods have advantageous features:

- Dimension-independent construction
- General mesh support (polytopes with or without matching interfaces)
- Arbitrary polynomial order

There are lots of DiSk methods: MFD, HFV, HDG, HHO. At CERMICS we develop HHO [1].

Outline

We needed a software platform for HHO

- *general*, to exploit the features of HHO
- *efficient*, to be able to handle big problems

Today we'll discuss the software platform we built

- Introduction to Hybrid High-Order methods
- Implementing Discontinuous Skeletal methods: goals and challenges
- Some words on generic programming
- The `DiSk++` template library, the infrastructure which supports our implementation of HHO

Introduction to HHO

Introduction to HHO

HHO belongs to the larger class of **Di**scontinuous **Sk**eletal methods.

- Family of arbitrary-order ($k \geqslant 0$) methods $\implies$ *High-order*
- Suitable for general polytopal meshes with matching or non-matching interfaces
- DoFs are polynomials of order k attached to both the mesh cells and the mesh faces $\implies$ *Hybrid*
- Dimension-independent construction
- Devised from *local reconstruction operators* and from a *local stabilization term*
- Succesfully used on diffusion [2, 3], linear elasticity [4], Cahn-Hilliard [5], ...

Setting: Poisson model problem

Let $\Omega \subset \mathbb{R}^d$ with $d \in \{1, 2, 3\}$ be an open, bounded and connected polytopal domain. We will consider the model problem

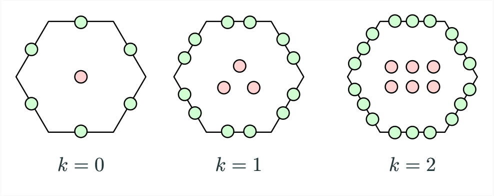
$$\begin{aligned} -\Delta u &= f \quad \text{in } \Omega, \\ u &= 0 \quad \text{on } \partial\Omega, \end{aligned}$$

with $f \in L^2(\Omega)$. By setting $V := H_0^1(\Omega)$, its weak form is

$$\text{Find } u \in V \text{ such that } (\nabla u, \nabla v)_\Omega = (f, v)_\Omega \text{ for all } v \in V.$$

Degrees of freedom

To discretize our problem we need a mesh. Then we choose the DoFs:



Let $\mathcal{M} := (\mathcal{T}, \mathcal{F})$ be the mesh consisting of the set of cells $\mathcal{T}$ and the set of faces $\mathcal{F}$ in which Ω is discretized. For each $T \in \mathcal{T}$ we can define the local space of DoFs

$$U_T^k := \mathbb{P}_d^k(T) \times \left\{ \bigtimes_{F \in \mathcal{F}_T} \mathbb{P}_{d-1}^k(F) \right\}$$

Reconstruction operator

Let $\mathbb{P}^{k+1}_{*d}(T) := \{q \in \mathbb{P}^{k+1}_{d}(T) \mid (q,1)_T = 0\}$.

Local reconstruction operator $R^{k+1}_T : U^k_T \to \mathbb{P}^{k+1}_{*d}(T)$ defined such that

- for all $(v_T, v_{\partial T}) \in U^k_T$
- for all $w \in \mathbb{P}^{k+1}_{*d}(T)$

$$(\nabla R^{k+1}_T(v_T, v_{\partial T}), \nabla w)_T := (\nabla v_T, \nabla w)_T + \sum_{F \in \mathcal{F}_{\partial T}} (v_F - v_T, \nabla w \cdot \boldsymbol{n}_T)_F$$

- Reconstruction operator derives from integration by parts formula
- A local Neumann problem is solved on each cell

Reconstruction operator II

Since:

- $R_T^{k+1}(v_T, v_{\partial T}) \in \mathbb{P}_{*d}^{k+1}(T)$
- bilinear form $(\nabla\bullet, \nabla\bullet)_T$ coercive on $\mathbb{P}_{*d}^{k+1}(T)$

the reconstruction function $R_T^{k+1}(v_T, v_{\partial T})$ is uniquely defined [1].

Moreover, let $I_T^k : H^1(T) \to U_T^k$ be the reduction map

$$I_T^k(v) = (\Pi_T^k(v), \Pi_F^k(v)),$$

where Π_T^k and Π_F^k are the L_2-orthogonal projectors on the cell and on the faces respectively. The following consistency property holds:

$$\nabla R_T^{k+1}(I_T^k(q)) = \nabla q, \qquad \forall q \in \mathbb{P}_d^{k+1}(T).$$

Reconstruction operator III

Reconstruction operator $R_T^{k+1}(v_T, v_{\partial T})$ is used to build the following bilinear form on $U_T^k \times U_T^k$:

$$a_T^{(1)}((v_T, v_{\partial T}), (w_T, w_{\partial T})) = (\nabla R_T^{k+1}(v_T, v_{\partial T}), \nabla R_T^{k+1}(w_T, w_{\partial T}))_T$$

Note how $(\nabla R_T^{k+1}(v_T, v_{\partial T}), \nabla R_T^{k+1}(w_T, w_{\partial T}))_T$ mimics *locally* the l.h.s. of our original problem

$$\text{Find } u \in H_0^1(\Omega) \text{ s.t. } (\nabla u, \nabla v)_\Omega = (f, v)_\Omega, \quad \forall v \in H_0^1(\Omega)$$

Stabilization operator I

For $(v_T, v_{\partial T}) \in U_T^k$, the reconstructed gradient $\nabla R_T^{k+1}(v_T, v_{\partial T})$ is not stable: $\nabla R_T^{k+1}(v_T, v_{\partial T}) = 0$ *does not imply* that v_T and $v_{\partial T}$ are constant functions taking the same value.

We introduce a least-squares penalization of the difference between functions in the faces and function in the cell

$$S_T^k(v_T, v_{\partial T}) := \Pi_{\partial T}^k \left(v_{\partial T} - (v_T + r_T^{k+1} - \Pi_T^k r_T^{k+1})|_{\partial T} \right),$$

where we use the shorthand notation $r_T^{k+1} := R_T^{k+1}(v_T, v_{\partial T})$.

Stabilization operator II

Using the stabilization operator just defined, we build a second bilinear form on $U_T^k \times U_T^k$:

$$a_T^{(2)}((v_T, v_{\partial T}), (w_T, w_{\partial T})) = \sum_{F \in \mathcal{F}_{\partial T}} h_F^{-1} (S_T^k(v_T, v_{\partial T}), S_T^k(w_T, w_{\partial T}))_F,$$

where h_F denotes the diameter of the face F.

- The stabilization as defined allows HHO to converge as $k + 2$ in L_2 norm
- The simpler stabilization considering the difference $v_{\partial T} - v_T$ would limit convergence to $k + 1$

Global spaces

Local discrete spaces U_T^k, for all $T \in \mathcal{T}$, are collected into a global discrete space

$$U_{\mathcal{M}}^k := U_{\mathcal{T}}^k \times U_{\mathcal{F}}^k,$$

where

$$U_{\mathcal{T}}^k := \mathbb{P}_d^k(\mathcal{T}) := \{v_{\mathcal{T}} = (v_T)_{T\in\mathcal{T}} \mid v_T \in \mathbb{P}_d^k(T),\ \forall T \in \mathcal{T}\},$$
$$U_{\mathcal{F}}^k := \mathbb{P}_{d-1}^k(\mathcal{F}) := \{v_{\mathcal{F}} = (v_F)_{F\in\mathcal{F}} \mid v_F \in \mathbb{P}_{d-1}^k(F),\ \forall F \in \mathcal{F}\}.$$

For a pair $v_{\mathcal{M}} := (v_{\mathcal{T}}, v_{\mathcal{F}})$ in the global discrete space $U_{\mathcal{M}}^k$, we denote $(v_T, v_{\partial T})$, for all $T \in \mathcal{T}$, its restriction to the local discrete space U_T^k, where $v_{\partial T} = (v_F)_{F\in\mathcal{F}_{\partial T}}$

Boundary conditions

Homogeneous Dirichlet BCs are enforced strongly by considering the subspace

$$U^k_{\mathcal{M},0} := U^k_{\mathcal{T}} \times U^k_{\mathcal{F},0},$$

where

$$U^k_{\mathcal{F},0} := \{v_{\mathcal{F}} \in U^k_{\mathcal{F}} \mid v_F \equiv 0,\ \forall F \in \mathcal{F}^{\mathrm{b}}\}.$$

Discrete problem

For all $T \in \mathcal{T}$, we combine reconstruction and stabilization bilinear forms into a_T on $U_T^k \times U_T^k$ such that

$$a_T := a_T^{(1)} + a_T^{(2)}.$$

We then do a standard cell-wise assembly

$$a_{\mathcal{M}}(u_{\mathcal{M}}, w_{\mathcal{M}}) := \sum_{T \in \mathcal{T}} a_T((u_T, u_{\partial T}), (w_T, w_{\partial T})),$$
$$\ell_{\mathcal{M}}(w_{\mathcal{M}}) := \sum_{T \in \mathcal{T}} (f, w_T)_T.$$

Finally we search for $u_{\mathcal{M}} := (u_{\mathcal{T}}, u_{\mathcal{F}}) \in U_{\mathcal{M},0}^k$ such that

$$a_{\mathcal{M}}(u_{\mathcal{M}}, w_{\mathcal{M}}) = \ell_{\mathcal{M}}(w_{\mathcal{M}}), \qquad \forall w_{\mathcal{M}} := (w_{\mathcal{T}}, w_{\mathcal{F}}) \in U_{\mathcal{M},0}^k,$$

It is possible (and we do it) to use static condensation to remove cell-based DoFs.

Implementing DiSk methods, goals

As emerges by previous discussion, HHO is:

- Dimension-independent
- Cell-shape-independent

These features are shared by most numerical methods and, mathematically, this is rather natural.

Implementing DiSk methods, goals (2)

The corresponding software implementation then

- should be able to run on any kind of mesh
- should be as efficient as possible, on any kind of mesh
- should allow the user to write its code without caring about the details of the underlying mesh (element shape/space dimension)

Goal in one sentence:
write the method once, run it on any kind of mesh.

Implementing DiSk methods, practice

Unfortunately, in practice there are some issues

- devising an implementation which is *efficient* and *general* at the same time is not trivial
- sometimes, with languages like Fortran or Matlab, is not even feasible

What usually happens is that one ends up writing different codes for 1D, 2D and 3D, Or different codes for different element shapes. Or codes that run on any element shape, but have sub-optimal performance.

DiSk++

We propose a C++ library realizing all the desirable features we listed a couple of slides ago.

The library is not HHO specific, it can be used to implement any method!

Our goals are pursued by employing

- generic programming (templates) to build zero-cost abstractions
- as much information as possible about the kind of mesh, to have always the most efficient possible representation and operations

Introduction to GP

Generic programming

Generic programming is a way to write algorithms and data structures leaving the types (in some sense) unspecified. Think about pseudo-code.

Generic programming is about

- Code reuse
- Performance
- Readability
- Maintainability
- Correctness
- ...

Code reuse I

Let's consider the `abs()` function.

```
int abs(int a)
{
  return (a < 0) ? -a : a;
}
```

Code reuse I

Let’s consider the `abs()` function.

```
int abs(int a)
{
  return (a < 0) ? -a : a;
}
```

What if I need to compute absolute values of `float`s or `double`s?

Code reuse I

Let's consider the `abs()` function.

```
int abs(int a)
{
  return (a < 0) ? -a : a;
}
```

What if I need to compute absolute values of `float`s or `double`s?

```
float abs(float a)
{
  return (a < 0) ? -a : a;
}
```

Code reuse I

Let's consider the `abs()` function.

```
int abs(int a)
{
  return (a < 0) ? -a : a;
}
```

What if I need to compute absolute values of `float`s or `double`s?

```
float abs(float a)
{
  return (a < 0) ? -a : a;
}
```

```
double abs(double a)
{
  return (a < 0) ? -a : a;
}
```

Code reuse I

Let’s consider the `abs()` function.

```
int abs(int a)
{
  return (a < 0) ? -a : a;
}
```

What if I need to compute absolute values of `float`s or `double`s?

```
float abs(float a)
{
  return (a < 0) ? -a : a;
}
```

```
double abs(double a)
{
  return (a < 0) ? -a : a;
}
```

Do you notice the problem?

Code reuse II

To avoid writing a function with the same body for each type, I can write

```
template<typename T>
T abs(T a)
{
  return (a < 0) ? -a : a;
}
```

Read “`template<typename T>`” as “ $\forall T$ ”

Code reuse II

To avoid writing a function with the same body for each type, I can write

```
template<typename T>
T abs(T a)
{
  return (a < 0) ? -a : a;
}
```

Read “`template<typename T>`” as “ $\forall$T ”

- T gets substituted by the compiler with the right type $\implies$ it is the compiler doing the boring job of writing the needed versions, not you!
- As long as $<$ and $-$ are defined for T, everything works just fine.
- This happens at *compile time!!* In the machine code you will not find any trace of this mechanism. For this reason this kind of things has zero overhead.

Code reuse III

What if I want `abs()` working also on complex numbers, quaternions, etc? Just add an *overload* of `abs()`!

```
template<typename T>
T abs(std::complex<T> a)
{
  return sqrt( real(val)*real(val) + imag(val)*imag(val) );
}
```

Code reuse III

Result: for the user the call to `abs()` is transparent w.r.t. the type of the argument. Forget things like `fabs()/fabsl()/fabsf()`!!

```
int main(void)
{
    std::cout << abs(-10) << std::endl;
    std::cout << abs(-10.0) << std::endl;
    std::cout << abs(10.0 - 10.0i) << std::endl;
}
```

All this has obvius advantages in terms of user friendliness and maintainability of the code.

Performance I

Example: The C `qsort()` function (applies also to Fortran)

```
void qsort(void *base, size_t nel, size_t width,
           int (*compar)(const void *, const void *));
```

`qsort()` lives in the standard C library

- is immutable *binary* code (kills any chance of inlining)
- it will call into your code at each comparison! Can you count the number of function calls for an array of 1M integers? [$O(n \log n)$]

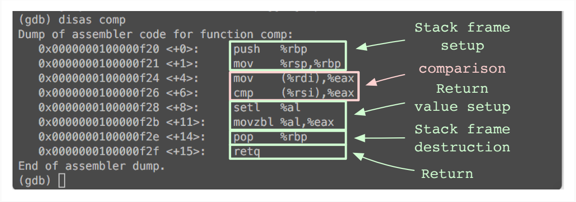
Performance I

Example: The C `qsort()` function (applies also to Fortran)

```
void qsort(void *base, size_t nel, size_t width,
           int (*compar)(const void *, const void *));
```

`qsort()` lives in the standard C library

- is immutable *binary* code (kills any chance of inlining)
- it will call into your code at each comparison! Can you count the number of function calls for an array of 1M integers? [$O(n \log n)$]



```
(gdb) disas comp
Dump of assembler code for function comp:
   0x0000000100000f20 <+0>:     push   %rbp
   0x0000000100000f21 <+1>:     mov    %rsp,%rbp
   0x0000000100000f24 <+4>:     mov    (%rdi),%eax
   0x0000000100000f26 <+6>:     cmp    (%rsi),%eax
   0x0000000100000f28 <+8>:     setl   %al
   0x0000000100000f2b <+11>:    movzbl %al,%eax
   0x0000000100000f2e <+14>:    pop    %rbp
   0x0000000100000f2f <+15>:    retq
End of assembler dump.
(gdb)
```

Of 8 machine instructions, 6 of them are overhead (75%)

Performance II

The way how generic programming is implemented in C++ allows the compiler to do very aggressive inlining.

```
std::vector<T> v;
std::sort(v.begin(), v.end());
```

- as `abs()`, previous code works for any `T`, as long as $<$ defined for `T`
- if not, you can define your own $<$
- `std::sort()` lives in a header, and not in a binary
- it is source code and not binary code: the compiler can embed the $<$ (2 instructions for integers) directly inside the code of `std::sort()`

Because of this, you normally see that sorting in C++ is much faster than sorting in C.

DiSk++

DiSk++

DiSk++ uses generic programming techniques to expose to the user a simple interface to code numerical methods.

The machinery of DiSk++ is divided in different functional areas:

- Mesh loading (from different file formats)
- Mesh representation [biggest issue]
- Geometric operations
- Quadratures/Basis functions
- Numerical methods primitives (i.e. HHO gradient reconstruction)

Mesh representation I

Remember our goal: HHO (and others) is formulated in dimension-independent and cell-shape-independent way, we want the software to be the same. And to be fast.

Mesh representation is a big issue:

- On one side we have different file formats. Each format represents the mesh in its own way. And the mesh can be 1D, 2D or 3D.
- On the other side we have the HHO method, in which we have only the concepts of *cells* and *faces*. And we forget about dimension.

How to match the two sides?

Mesh representation II

A mesh is a collection of

- 0-polytopes: Nodes
- 1-polytopes: Edges
- 2-polytopes: Triangles, quadrangles, …
- 3-polytopes: Tetrahedra, hexahedra, …
- d-polytopes: $d > 3$ supported, but do we really need them?

Mesh representation II

A mesh is a collection of

- 0-polytopes: Nodes
- 1-polytopes: Edges
- 2-polytopes: Triangles, quadrangles, ...
- 3-polytopes: Tetrahedra, hexahedra, ...
- d-polytopes: $d > 3$ supported, but do we really need them?

Easy! Since HHO needs cells and faces, load the mesh in some data structure and, for dimension d, just map d-polytopes to cells and $(d-1)$-polytopes to faces!

Mesh representation II

A mesh is a collection of

- 0-polytopes: Nodes
- 1-polytopes: Edges
- 2-polytopes: Triangles, quadrangles, ...
- 3-polytopes: Tetrahedra, hexahedra, ...
- d-polytopes: $d > 3$ supported, but do we really need them?

Easy! Since HHO needs cells and faces, load the mesh in some data structure and, for dimension d, just map d-polytopes to cells and $(d-1)$-polytopes to faces!

Yeah, that's the idea, but not so fast!

Mesh representation III

We can have very different kinds of mesh

- with only simplicial elements
- with only cartesian elements
- with mixed elements, but with a fixed number of them
- with fully general polygonal/polyhedral elements

and we want to run as fast as possible on all those kinds of mesh.

Mesh representation III

We can have very different kinds of mesh

- with only simplicial elements
- with only cartesian elements
- with mixed elements, but with a fixed number of them
- with fully general polygonal/polyhedral elements

and we want to run as fast as possible on all those kinds of mesh.

This is not possible with a single, “conventional” data structure. For example:

- tetrahedron fully described by 4 integers
- generic polyhedra needs more complex description
- quadratures? If element shape is known, they can be optimized

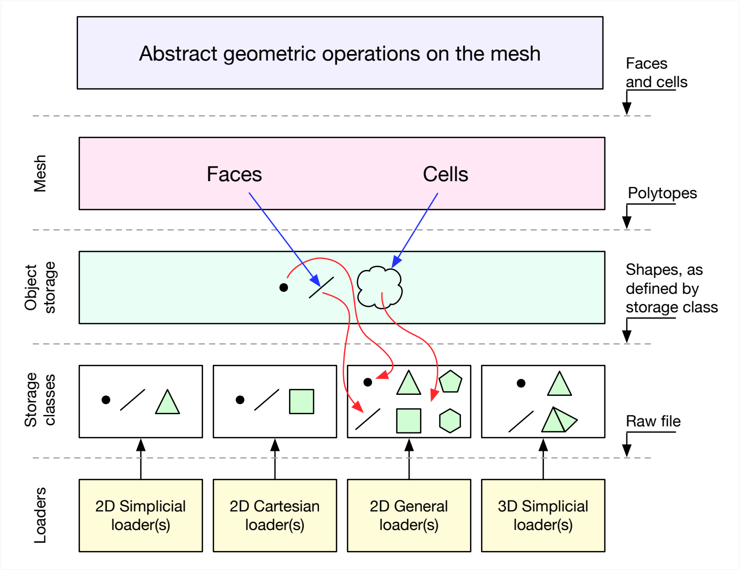
Storage classes

- We introduce the *storage classes*, which encode the optimal representation for each kind of element.
- The data structure for the mesh is a template instantiated on a specific storage class...
- ...and thus becomes a collection of objects of a *specific* class.

DiSk++ provides different storage classes: simplicial 2D/3D, cartesian 2D/3D, general 2D/3D, ...

Let me use a picture to show you how things work...

The overall architecture of DiSk++



Where’s the deal?

All this looks complicated, but...

- all the machinery is hidden inside the library, the user does not need (and shouldn’t) come in contact with it or even know it...
- ...except if he wants to extend the library to support a new kind of mesh.
- In that case he needs to write only the loader and possibly a new storage class.

Actually, it is really easy for the user to manipulate the mesh and do its own computations.

Coding example

Suppose we want to compute some properties of the cells and the faces of a mesh:

```
for (auto& cl : msh) {
  auto cell_meas = measure(msh, cl);
  auto cell_bar = barycenter(msh, cl);
  auto fcs = faces(msh, cl);
  for (auto& fc : fcs) {
    auto face_meas = measure(msh, fc);
    auto face_bar = barycenter(msh, fc);
  }
}
```

This code will work on any mesh you will throw at it as efficiently as possible!

Geometric operations on the mesh I

In the previous code snippet we met some functions

- `barycenter()`
- `measure()`
- `faces()`

They are part of the set of the numerous geometric operations provided by DiSk++.

Let's take a look at them and understand how generic programming helps us.

```
template<typename Mesh, typename Element>
typename Mesh::point_type
barycenter(const Mesh& msh, const Element& elm)
{
  auto pts = points(msh, elm);
  auto bar = accumulate(next(pts.begin()), pts.end(), pts.front());
  return bar / pts.size();
}
```

Geometric operations on the mesh II

Sometimes an operation is not generalizable to every element.

```
template<typename T>
T measure(const generic_mesh<T,2>& msh,
          const typename generic_mesh<T,2>::face& fc);


template<typename T>
T measure(const simplicial_mesh<T, 3>& msh,
          const typename simplicial_mesh<T, 3>::face& surf);
```

- in that case is sufficient to write the correct specializations for a given element/storage class (exactly like `abs()`)
- the compiler will pick the right one at *compile time*
- since they have all the same name, the user sees only one `measure()`
- we (re)wrote only the code we actually needed

Quadratures and basis functions

As for the geometric operations, also quadratures and basis functions are generic:

```
template<typename mesh_type, typename cell_type>
class scaled_monomial_basis<mesh_type, cell_type> {...};

template<typename mesh_type, typename cell_type>
class quadrature<mesh_type, cell_type> {...};
```

- Simplicial mesh $\implies$ simplicial quadratures
- Cartesian mesh $\implies$ tensorized Gauss points
- General mesh $\implies$ split in simplices

Again, the user does not need to know anything about this. You'll get automatically the right thing you need.

Example: compute $\int_T pq$ on each T

```
template<typename Mesh>
void compute_integrals(const Mesh& msh)
{ /* assumes that degree, p and q are correctly defined */
  typedef Mesh                                   mesh_type;
  typedef typename mesh_type::cell_type          cell_type;

  scaled_monomial_basis<mesh_type, cell_type> cb(degree);
  quadrature<mesh_type, cell_type> cq(2*degree);
  for (auto& cl : msh) {
    auto quadpoints = cq.integrate(msh, cl);
    for (auto& qp : quadpoints) {
      auto phi  = cb.eval_functions(msh, cl, qp.point());
      mass_matrix += qp.weight() * phi * phi.transpose();
    }
  std::cout << "(p,q) = " << dot(q, mass_matrix*p) << std::endl;
  }
}
```

HHO and DiSk++

DiSk++ allowed to implement HHO stuff in a very simple and compact way

- Gradient reconstruction ($\approx$ 80 LOCs)
- Divergence reconstruction ($\approx$ 90 LOCs)
- Stabilization ($\approx$ 70 LOCs)

They are written in a completely mesh-independent fashon.

These basic building blocks live in three template classes

- `gradient_reconstuction`
- `divergence_reconstuction`
- `diffusion_like_stabilization`

To solve a problem with HHO, just combine them!

Assembly of diffusion problem

```
gradient_reconstruction<mesh_type, basis_type> gradgrad;
diffusion_like_stabilization<mesh_type, basis_type> stab;

for (auto& cl : msh) {
  /* build $a_T^{(1)}$ and $a_T^{(2)}$*/
  gradgrad.compute(msh, cl);
  stab.compute(msh, cl, gradgrad.oper);
  auto cell_rhs =
    compute_rhs<cell_basis_type,
                cell_quadrature_type>(msh, cl, load, degree);
  /* local cell contribution: $a_T^{(1)} + a_T^{(2)}$ */
  matrix_type loc = gradgrad.data + stab.data;
  /* do static condensation */
  auto sc = statcond.compute(msh, cl, loc, cell_rhs);
  assembler.assemble(msh, cl, sc);
}
assembler.impose_boundary_conditions(msh, bc_func);
assembler.finalize();
```

Overall performance of DiSk++

Having specialized data structures for particular types of mesh pays off.

Assembly time (seconds, single thread) for 1 million DoFs in different cases: [S] = specialized, [G] = general

k	Tet [S]	Tet [G]	Hex [S]	Hex [G]	Poly [G]
0	20	55	23	99	160
1	20	60	25	102	171
2	22	60	32	105	188
3	37	98	53	167	300

- CPU: Core i7 3615QM/16 GB RAM.
- Compiler: Clang 8.0.0 (Apple)
- Linear algebra: Eigen 3.3.2.

Preliminary information indicates DiSk++ is 5x-10x faster than a competing Fortran implementation.

Effectiveness of storage-class-parametrized mesh (tet)

Speedup of Reconstruction and Stabilization due to the usage of a specialized data structure. Tetrahedral meshes, from left to right and from top to bottom: $k = 0$, $k = 1$, $k = 2$, $k = 3$.

DOFs	Rec	Stab
4912	3.60x	2.45x
9152	3.66x	2.51x
17600	3.21x	2.18x
34009	3.91x	2.69x

DOFs	Rec	Stab
14736	2.91x	2.68x
27456	2.61x	2.36x
52800	2.91x	2.66x
102027	2.89x	2.62x

DOFs	Rec	Stab
29472	2.68x	2.29x
54912	2.54x	2.18x
105600	2.73x	2.35x
204054	2.86x	2.46x

DOFs	Rec	Stab
49120	2.58x	2.11x
91520	2.67x	2.21x
176000	2.70x	2.22x
340090	2.77x	2.25x

Effectiveness of storage-class-parametrized mesh (hex)

Speedup of Reconstruction and Stabilization due to the usage of a specialized data structure. Hexahedral meshes, from left to right and from top to bottom: $k = 0$, $k = 1$, $k = 2$, $k = 3$.

DOFs	Rec	Stab
60	4.84x	3.99x
336	4.23x	3.21x
2112	4.47x	3.33x
14592	4.72x	3.45x

DOFs	Rec	Stab
180	5.53x	4.77x
1008	3.74x	3.29x
6336	3.57x	3.06x
43776	4.06x	3.54x

DOFs	Rec	Stab
360	3.70x	2.94x
2016	3.49x	2.83x
12672	3.29x	2.70x
87552	3.60x	2.95x

DOFs	Rec	Stab
600	3.13x	2.44x
3360	3.10x	2.41x
21120	3.36x	2.66x
145920	3.61x	2.82x

Conclusions

Conclusions

- We built a library of primitives to implement DiSk methods
- We implemented HHO on top of it
- Generic programming allowed to hide the complexity and to provide an user-friendly interface that matches the dimension-independence and mesh-independence of the method
- Performance numbers look very good

Future directions

DiSk++ HHO layer is currently growing. It is used at CERMICS for

- Multiscale problems (Matteo)
- Nonlinear problems and mesh adaptation (Karol)
- Elasticity (Nicolas)

DiSk++ is quite stable at the moment, however is not yet as user friendly as I want...I'm working on it!

- Further simplification of the syntax and the interface
- Better integration with visualization tools
- Further optimization of general polyhedra support
- Implementation of other methods to ease comparisons

Thank you!

Thank you!

e-mail: `matteo.cicuttin@enpc.fr`

code: `https://github.com/datafl4sh/diskpp`

Bibliography

- D. Di Pietro, A. Ern and S. Lemaire, A review of Hybrid High-Order methods: formulations, computational aspects, comparison with other methods
- D. Di Pietro and A. Ern, Hybrid high-order methods for variable-diffusion problems on general meshes
- D. Di Pietro, J. Droniou and A. Ern, A discontinuous-skeletal method for advection-diffusion-reaction on general meshes
- D. Di Pietro and A. Ern, A hybrid high-order locking-free method for linear elasticity on general meshes
- F. Chave, D. A. Di Pietro, F. Marche, F. Pigeonneau, A Hybrid High-Order Method for the Cahn–Hilliard Problem in Mixed Form