Function space bases in the dune-functions module

Oliver Sander

with

Christian Engwer, Carsten Gräser, Katja Hanowski, and Steffen Müthing

X-DMS, Umeå, 19. 6. 2017

Dune
Distributed and Unified Numerics Environment

TECHNISCHE UNIVERSITÄT DRESDEN

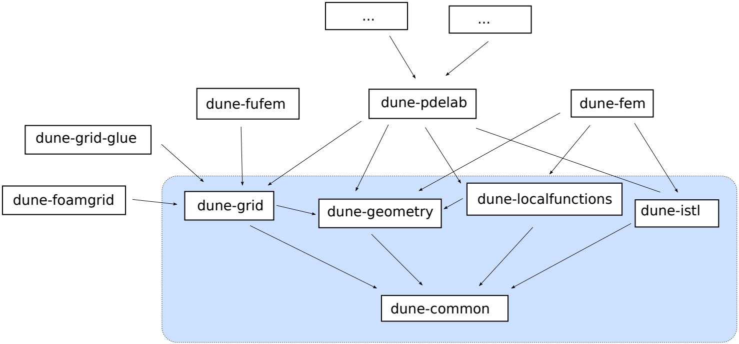
The case for standardization

- Very many FE codes
- Good reasons to have more than one
- Lots of wheels reinvented

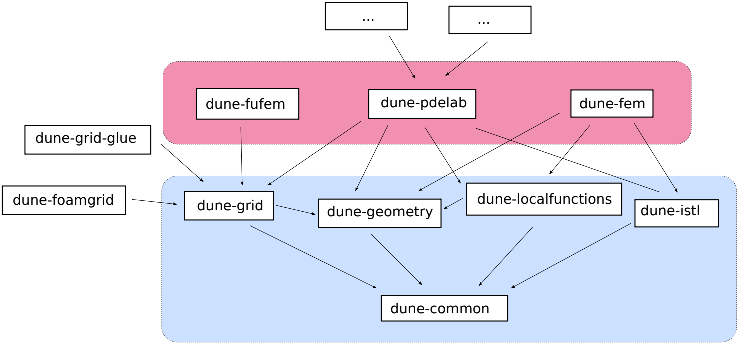
Dune: Agree on low-level components

- Set of libraries for grid-based numerical methods
  - Grids
  - Shape functions
  - Linear algebra
  - etc.
- Open source C++ code
- `www.dune-project.org`

Dune module structure



Dune module structure



Discretization modules

- **dune-fem**: Focus on adaptivity, parallelism, and efficiency
- **dune-pdelab**: Very flexible and powerful—steep learning curve
- **dune-fufem**: Easy to use—less powerful

Functions

- Interface for functions $f : \mathbb{R}^n \to \mathbb{R}^m$, differentiable functions, grid functions, etc.
- Based on callables, concepts and type erasure

Function space bases

- Shape functions and indices
- Independent from any linear algebra
- Hierarchic construction of mixed FE bases

Infrastructure

- Interpolation:

$$\text{function} + \text{basis} \Rightarrow \text{coefficient vector}$$

- VTK output of grid functions

The case for bases

- Grid function spaces are *not* the right abstraction
- More than one basis for the same space
  - E.g., P2 nodal basis vs. hierarchical basis
  - Orthogonal vs. Lagrange DG basis
- Basis + coefficients = discrete function

Functionality of a basis

For any given grid element

- . . . get restrictions of relevant basis functions to this element
  - i.e., the shape functions
  - use dune-localfunctions interfaces
- . . . get local shape function numbers
- . . . get global basis function numbers

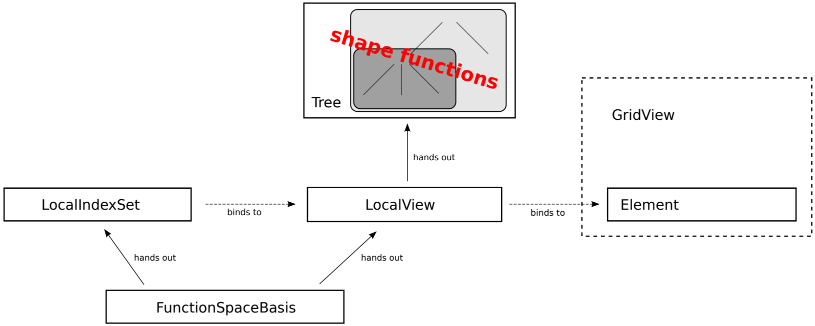


Figure: Overview of the classes making up the interface to finite element space bases

Implement the linear functional

$$\ell(v) = \int_\Omega f v \, dx$$

- The function f is given as

```
std::function<double(FieldVector<double,dim>)> f;
```

- The result is written as a container

```
std::vector<double> rhs;
```

- The basis is given as an object

```
basis
```

Assembler for a linear functional

```
{
    rhs.resize(basis.size());
    std::fill(rhs.begin(), rhs.end(), 0.0);

    auto localView = basis.localView();
    auto localIndexSet = basis.localIndexSet();

    // A loop over all elements of the grid
    for (const auto& element : elements(basis.gridView()))
    {
        localView.bind(element);
        localIndexSet.bind(localView);

        // Assemble the local contribution
        std::vector<double> localRhs;
        [ assemble the local contribution ]

        // Add local contribution onto global container
        for (size_t i=0; i<localRhs.size(); i++)
            rhs[localIndex.index(i)] = localRhs[i];
    }
}
```

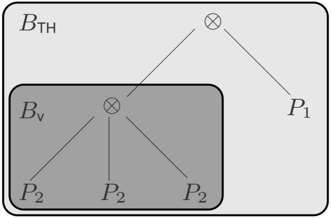
Assembling the local contribution

```
// Set of shape functions for a single element
const auto& localFiniteElement = localView.tree().finiteElement();

// Set all entries to zero
localRhs.resize(localFiniteElement.size());
localRhs = std::fill(localRhs.begin(), localRhs.end(), 0.0);

// A quadrature rule
const auto& quadRule = QuadratureRules<double, dim>::rule(element.type(), order);

// Loop over all quadrature points
for (const auto& quadPoint : quadRule)
{
    // Position of the current quadrature point in the reference element
    const auto quadPos = quadPoint.position();

    // The multiplicative factor in the integral transformation formula
    const double integrationElement = element.geometry().integrationElement(quadPos);

    double functionValue = f(element.geometry().global(quadPos));

    // Evaluate all shape function values at this point
    std::vector<FieldVector<double,1> > shapeFunctionValues;
    localFiniteElement.localBasis().evaluateFunction(quadPos, shapeFunctionValues);

    // Actually compute the vector entries
    for (size_t i=0; i<localRhs.size(); i++)
        localRhs[i] += shapeFunctionValues[i] * functionValue * quadPoint.weight() * integrationElement;
}
```

Systematic construction of basis for vector-valued spaces

- Tensor products of simpler basis
- Taylor–Hood: $B_{\text{TH}} = (P_2 \otimes P_2 \otimes P_2) \otimes P_1$

Tree representation



Systematic construction of

- orderings
- multi-indices

Taylor–Hood basis: lexicographic ordering

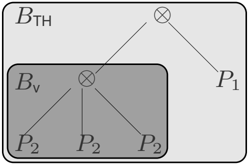
$b_{x,0}$	0	$(0,0)$	$(0,0)$	$(0,0,0)$
$b_{x,1}$	1	$(0,1)$	$(0,1)$	$(0,0,1)$
$b_{x,2}$	2	$(0,2)$	$(0,2)$	$(0,0,2)$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
$b_{y,0}$	n	$(0,n)$	$(1,0)$	$(0,1,0)$
$b_{y,1}$	$n+1$	$(0,n+1)$	$(1,1)$	$(0,1,1)$
$b_{y,2}$	$n+2$	$(0,n+2)$	$(1,2)$	$(0,1,2)$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
$b_{z,0}$	$2n$	$(0,2n)$	$(2,0)$	$(0,2,0)$
$b_{z,1}$	$2n+1$	$(0,2n+1)$	$(2,1)$	$(0,2,1)$
$b_{z,2}$	$2n+2$	$(0,2n+2)$	$(2,2)$	$(0,2,2)$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
p_0	$3n$	$(1,0)$	n	$(1,0)$
p_1	$3n+1$	$(1,1)$	$n+1$	$(1,1)$
p_2	$3n+2$	$(1,2)$	$n+2$	$(1,2)$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$

Possible index types for lexicographic ordering of the velocity basis functions

Taylor–Hood basis: interleaved ordering

$b_{x,0}$	0	$(0,0)$	$(0,0)$	$(0,0,0)$
$b_{y,0}$	1	$(0,1)$	$(0,1)$	$(0,0,1)$
$b_{z,0}$	2	$(0,2)$	$(0,2)$	$(0,0,2)$
$b_{x,1}$	3	$(0,3)$	$(1,0)$	$(0,1,0)$
$b_{y,1}$	4	$(0,4)$	$(1,1)$	$(0,1,1)$
$b_{z,1}$	5	$(0,5)$	$(1,2)$	$(0,1,2)$
$b_{x,2}$	6	$(0,6)$	$(2,0)$	$(0,2,0)$
$b_{y,2}$	7	$(0,7)$	$(2,1)$	$(0,2,1)$
$b_{z,2}$	8	$(0,8)$	$(2,2)$	$(0,2,2)$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
p_0	$3n$	$(1,0)$	n	$(1,0)$
p_1	$3n+1$	$(1,1)$	$n+1$	$(1,1)$
p_2	$3n+2$	$(1,2)$	$n+2$	$(1,2)$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$

Possible index types for interleaved ordering of the velocity basis functions



Leaf nodes

- `const FiniteElement& finiteElement() const`
- `size_type localIndex(size_type i) const`

Inner nodes

- `PowerNode`: Combines identical subtrees
- `CompositeNode`: Combines differing subtrees

Node access

- `tree.child(a,b,c,...)`,
  with `a,b,c,...` either `int` or `std::integral_constant<size_type,.>`
- Example: `tree.child(_0,0)`: first component of velocity basis

How to set up a Taylor–Hood space

```
using namespace Functions::BasisBuilder;
static const std::size_t k = 1; // pressure order

auto taylorHoodBasis = makeBasis(
  gridView,
  composite(
    power<dim>(
      lagrange<k+1>(),
       flatInterleaved ()),
    lagrange<k>()
  ));
```

How to set up a Taylor–Hood space

```
using namespace Functions::BasisBuilder;
static const std::size_t k = 1; // pressure order

auto taylorHoodBasis = makeBasis(
  gridView,
  composite(
    power<dim>(
      lagrange<k+1>(),
       flatInterleaved ()),
    lagrange<k>()
  ));
```

Basis implementations

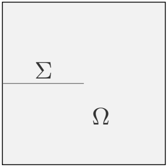
- PQkNodalBasis
- LagrangeDGBasis
- BSplineBasis
- RannacherTurekBasis
- . . . more to come

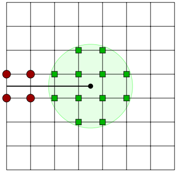
XFEM: eXtended Finite Element method

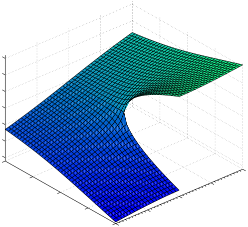
- Extend standard basis by additional problem-specific basis functions

XFEM for fracture mechanics problems:

- Lagrange basis
- Heaviside enrichment along the crack
- Singularity enrichment near the crack tip







Generic basis enhancement

- Example: enhance second-order Lagrange basis by fracture XFEM functions

```
using XFEMBasis = Functions::FractureXFEMBasis<PQkNodalBasis<GridView,2>,
                                               InteractionDetector>;
```

Generic basis enhancement

- Example: enhance second-order Lagrange basis by fracture XFEM functions

```
using XFEMBasis = Functions::FractureXFEMBasis<PQkNodalBasis<GridView,2>,
                                               InteractionDetector>;
```

Interface extensions for shape functions

- Direct access to jumps $[\![\phi_i]\!]$ and averages $\{\phi_i\}$

```
void LocalBasis::jump(const DomainType& localPoint,
                      std::vector<RangeType>& out) const;

void LocalBasis::average(const DomainType& localPoint,
                         std::vector<RangeType>& out) const;
```

- Assume that evaluation will only occur on the fracture, e.g.,

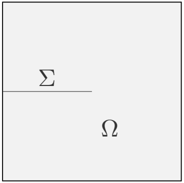
$$[\![H\phi_i]\!] = 2\phi_i$$

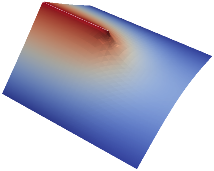
Darcy flow on fractured porous medium [Martin, Jaffré, Roberts]

- p^Ω, r^Ω: bulk pressure and test function
- p^Σ: Fracture pressure

(Some of the) weak coupling terms:

$$\frac{1}{2\xi - 1}\left(\frac{4K^\nu}{b}\{p^\Omega\}, \{r^\Omega\}\right)_\Sigma + \left(\frac{K^\nu}{b}[\![p^\Omega]\!], [\![r^\Omega]\!]\right)_\Sigma - \frac{1}{2\xi - 1}\left(\frac{4K^\nu}{b}p^\Sigma, \{r^\Omega\}\right)_\Sigma$$





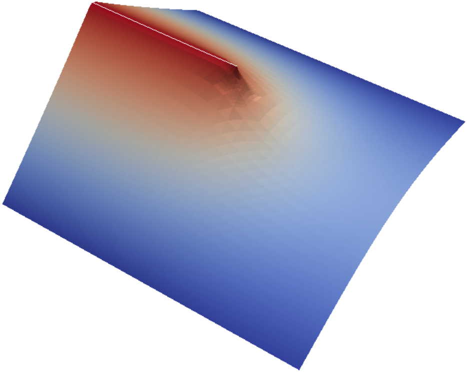
```
auto bulkLocalView = bulkBasis.localView();
auto bulkLocalIndexSet = bulkBasis.localIndexSet();

auto fractureLocalView = fractureBasis.localView();
auto fracturelocalIndexSet = fractureBasis.localIndexSet();

// loop over all intersections between bulk grid and fracture grid
for (const auto& intersection : intersections(glue))
{
  bulkLocalView.bind(intersection.inside());
  bulkLocalIndexSet.bind(bulkLocalView);
  const auto& bulkLocalFiniteElement = bulkLocalView.tree().finiteElement();

  fractureLocalView.bind(intersection.outside());
  fractureLocalIndexSet.bind(fractureLocalView);
  const auto& fractureLocalFiniteElement = fractureLocalView.tree().finiteElement;

  // — Set up data structures for shape function values, jumps, and averages
  // — Initialize the local matrix
  // — Request a quadrature rule
```

```
// Loop over all quadrature points
for (const auto& quadPoint : quadRule)
{
  // get quadrature point coordinates in the bulk and fracture elements
  const auto localInBulk = intersection.geometryInInside().global(quadPoint.position());
  const auto localInInterface = intersection.geometryInOutside().global(quadPoint.position());

  // Determinant term from the integral transformation formula
  double integrationElement = intersection.geometry().integrationElement(quadPoint.position());

  // Get jumps and averages of bulk shape functions
  bulkLocalFiniteElement.localBasis().evaluateJump(localInBulk, jump);
  bulkLocalFiniteElement.localBasis().evaluateAverage(localInBulk, average);

  // Get values of fracture shape functions
  fractureLocalFiniteElement.localBasis().evaluateFunction(localInInterface, values_fracture);
```

Example

$$\frac{1}{2\xi - 1}\left(\frac{4K^\nu}{b}\{p^\Omega\}, \{r^\Omega\}\right)_\Sigma + \left(\frac{K^\nu}{b}[\![p^\Omega]\!], [\![r^\Omega]\!]\right)_\Sigma - \frac{1}{2\xi - 1}\left(\frac{4K^\nu}{b}p^\Sigma, \{r^\Omega\}\right)_\Sigma$$

```
double z = quadPoint.weight() * integrationElement;

// Bulk i
for (int i = 0; i < volDOF; ++i)
{
  // Bulk j
  for (int j = 0; j < volDOF; ++j)
  {
    localMatrix[i][j] += z * (1.0 / (2.0*xi-1.0)) + (4.0*K_nu/b) * average[i] * average[j];

    localMatrix[i][j] += z * K_nu/b * jump[i] * jump[j];
  }

  // Interface j
  for (int j = volDOF; j < totalDOF; ++j)
  {
    double zij = z * (1.0 / (2.0*xi-1.0)) * (4.0*K_nu/b) * average[i] * values_fracture[j-volDOF];

    localMatrix[i][j] -= zij;
    localMatrix[j][i] -= zij;
  }
}
```



Further information

- www.dune-project.org/modules/dune-functions