Draft on the 'functionals-concept' in DUNE-FEM

Felix Albrecht (felix.albrecht@uni-muenster.de) and
Patrick Henning (patrick.henning@uni-muenster.de)

December 16, 2010

Abstract

This document is a draft about a new concept of DUNE-FEM basing on 'functionals'. DUNE-FEM is part of the Distributed and Unified Numerics Environment (DUNE) and is available from the site http://dune.mathematik.uni-freiburg.de/.

Contents

1 Analytical concept

1.1 Overview

We start with an overview on the mathematical concept which is carried over to a corresponding programming concept. The following notations and definitions are required for the subsequent sections.

Definition 1.1 (Function and Functionspace).
If Ω denotes a subset of $\mathbb{R}^n$, a mapping $v : \Omega \to \mathbb{R}^d$ is called a function. *Any vector space V which only consits functions is called a* functionspace. *If additionally*

$$(\alpha v_1 + \beta v_2) \in V \;\; \forall v_1, v_2 \in V, \forall \alpha, \beta \in \mathbb{R}$$

the space is called a linear functionspace.

Definition 1.2 (Functional).
Let V be a function space. A map

$$F : V \to \mathbb{R} \;\; with \;\; v \mapsto F[v] \tag{1.1}$$

is called a Functional. *If $F(\alpha v_1 + \beta v_2) = \alpha F(v_1) + \beta F(v_2)$ for all $\alpha, \beta \in \mathbb{R}$ and $v_1, v_2 \in V$, F is a* Linear Functional. *The space*

$$V' := \{F : V \to \mathbb{R} | \; is \; linear \; functional\} \tag{1.2}$$

is called the dual space of V.

Definition 1.3 (Constraint).
Let V be a linear function space, $M \in \mathbb{N}_{>0}$ and $\{F_1, ..., F_M\}$ a set of linear functionals on V. We define the corresponding vector of functionals C by

$$C : \{1, ..., M\} \times V \to \mathbb{R} \text{ with } (i, v) \mapsto C[i][v] := F_i[v]. \tag{1.3}$$

The condition:

$$C[i][v] = 0 \ \forall 1 \leq i \leq M \tag{1.4}$$

is called a Constraint *for v.*

In particular every single linear functional implies a constraint.

Definition 1.4 (Linear Subspace).
Let V be a linear function space and $C[i][\cdot] = 0$ a constraint on V. Then we call

$$V_C := \{v \in V | \ C[i][v] = 0 \ \forall i \in \{1, ..., M\}\} \tag{1.5}$$

a linear subspace *of V with respect to C.*

V_C is a linear vector space, since the constraint functionals $C[i]$ are linear. Typically, V_C becomes the space of test functions in our later problem.

Definition 1.5 (Affine Subspace).
Let V be a linear function space, V_C a linear subspace and $g \in V$. Then we call

$$V_g := \{v + g | \ v \in V_C\} \subset V \tag{1.6}$$

an affine subspace *of V with respect to g and V_C.*

In general, V_g is a nonlinear function space. It becomes the space of solution in our later problem.

Definition 1.6 (Operator).
Let V_C be a linear (constraint) function space with dual space V_C' and V_g a constraint subspace. Then we call

$$G : V_g \to V_C' \tag{1.7}$$

an operator *on V_g.*

In the subsequent sections, we are dealing with the following problem:

Problem 1.7. *For a linear space V, a linear (constraint) subspace V_C, a functional F and an affine subspace V_g of V, find $u \in V_g$ with*

$$G(u)[v] = F[v] \ \forall v \in V_C.$$

In general, the functional F on the right hand side of our problem is linear. The (differential) operator G can be either linear or nonlinear.

1.2 Example

Let $\Omega \subset \mathbb{R}^d$ denote a polygonal bounded domain, $\mathcal{T}_H = \{T_1, ..., T_N\}$ a corresponding regular triangulation, $\mathcal{N}_H = \{x_1, ..., x_{\tilde{N}}\}$ the set of nodes and $\{\Phi_1, ..., \Phi_{\tilde{N}}\}$ the associated Lagrange basis of order 1. The (discrete) linear space of solutions is given by

$$V := \{v_H \in C^0(\Omega) | \; (v_H)_{|T} \in \mathbb{P}^1(T) \forall T \in \mathcal{T}_H\}$$

and the linear subspace by $\overset{\circ}{V} := V \cap \overset{\circ}{H}{}^1(\Omega)$. Now, let us consider the following discrete problem:

Problem 1.8. *For $g \in V \subset C^0(\Omega)$ find $u \in V$ with $u = g$ on $\partial\Omega$ and*

$$\int_\Omega \nabla u \cdot \nabla v = \int_\Omega fv \quad \forall v \in \overset{\circ}{V}.$$

(Note: $\overset{\circ}{V}$ can not be replaced by V).

Putting this into the general framework above, we idenitfy the (linear) functional $F : \overset{\circ}{V} \to \mathbb{R}$ by

$$F[v] := \int_\Omega fv \text{ for } v \in \overset{\circ}{V}.$$

$\overset{\circ}{V}$ is a constraint subspace with the constraint $C[i][v] := v(x_i^b) = 0$ for any boundary node x_i^b (i.e. $\mathcal{N}_H \cap \partial\Omega = \{x_1^b, ..., x_{\bar{N}}^b\}$). We can therefore identify

$$\overset{\circ}{V} = \{v \in V | \; C[i][v] = 0 \;\; \forall 1 \le i \le \bar{N}\} =: V_C.$$

The affine space is given by:

$$V_{g_H} := \{v + g_H | \; v \in V \text{ and } g_H := \sum_{i=1}^{\tilde{N}} g(x_i)\Phi_i\}.$$

and the (differential) operator $G : V_{g_H} \to V_C'$ by:

$$G(u)[v] := \int_\Omega \nabla u \cdot \nabla v.$$

With these notations the problem reads:

$$\text{Find } u \in V_{g_H} \text{ with } G(u) = F \text{ on } V_C.$$

2 Programming concept

In this section we describe the general programming concept. Details on the implementation of the various classes are given later. We assume that we use a `namespace Functionals` in order to avoid conflicts with other existing Dune-Fem-classes.

2.1 Required classes

First of all we give an overview on the various classes that are required in our concept. In particuler we comment on the functionality of each class.

`typedef Functional < DiscreteFunctionSpace > FunctionalType;`

- ◦ from the the general type `Functional` we can derive various realizations of functionals
- ◦ at first, we restrict ourselves to linear functionals
- ◦ we might distinguish the types of functionals according to the codim: Codim-Functionals, for example `FunctionalType::CodimFunctional<codim>`; combinations of functionals for different codims must be possible (for instance $F(\Phi) = \int_\Omega f\Phi + \int_{\partial\Omega} g\Phi$)
- ◦ required methods:
- · method: `apply( function )` $\leftrightarrow F[v]$, v analytical function
- · method: `apply( discreteFunction )` $\leftrightarrow F[v_H]$, v_H discrete function
- · method: `applyLocal( localBasefunctionSet )` $\rightarrow$ an abstract method depending on the specific realisation of a functional; we can say it returns a vector of local contributions for a specific grid element; it is required for assembling the right hand side in our system of equations; details are given later

`typedef Constraint < FunctionalType > ConstraintType;`

- ◦ various realizations of constraints C are possible (boundary conditions, periodicity, zero-average, ...); they are derived from the general `Constraint` class
- ◦ mapping an element $v \in V$ on an element $v_C \in V_C$ is not unique, therefore 'applying a constraint' to a general function v means that we project v on V_C with respect to certain scalar product; in the discrete setting these projections are typically straight forward
- ◦ required methods:
- · method: `apply( numberOfConstraint, function )` $\leftrightarrow$ find $v_C \in V_C$ which is 'close' to v and which fulfills $C[i][v_C] = 0$, i is the index of the functional (in our functional vector), v is an analytical function
- · method: `apply( numberOfConstraint, discreteFunction )` $\leftrightarrow$ find $v_C \in V_C$ which is 'close' to v_H and which fulfills $C[i][v_C] = 0$, i is the index of the functional (in our functional vector), v_H is a discrete function; typically we simply change the value of v_H in a certain number of nodes
- · method: `applyLocal( numberOfConstraint, localBasefunctionSet, localBasefunctionSet )` $\rightarrow$ again, an abstract method depending on the specific type of the constraint; it returns local contributions for a specific grid element; it is required for assembling the system matrix in our system of equations; details are given later

  · method: `applyLocal( localBasefunctionSet, localBasefunctionSet )` $\rightarrow$ use `applyLocal( numberOfConstraint, localBasefunctionSet, localBasefunctionSet )` for all `numberOfConstraint`

- other methods depending on the specific type of a constraint (e.g. DirichletConstraint)?
- constraints are used to construct a 'constraint subspace' - for the user, nothing else has to be done with the constraints

`typedef LinearSubspace < DiscreteFunctionSpace, ConstraintType > LinearSubspaceType;`

- derived from `DiscreteFunctionSpace`
- all the information about the constraint is in our subspace
- we can extract the constraint that it was constructed from
- formally the subspace is of the same size as `DiscreteFunctionSpace`
- in particular an object of `LinearSubspace` becomes the space of test functions in our later problem

`typedef AffineSubspace < LinearSubspace > AffineSubspaceType;`

- the space of the solution
- initialized with a fixed discrete function v_H: 'AffineSubspace $= v_H +$ LinearSubspace'
- `AffineSubspace`-class derived from `DiscreteFunctionSpace`

`typedef Operator< LinearSubspaceType, AffineSubspaceType, MatrixObjectTraits > DifferentialOperatorType;`

- can be derived from the dune-fem Operator-class, later it should be implemented independently
- Operator : AffineSubspace $\rightarrow$ (LinearSubspace)$'$
- if required: automatically assembles the correct system matrix (which is a quadratic sparse row matrix) with respect to the subspaces (i.e. with respect to the constraints)
- simplified we can say: the *linear subspace* tells us which lines we must substitute in our later system of equations and the *affine subspace* tells us by what we must substitute these lines.
- usage of a `DifferentialOperatorType`-object identical to the old usage of an `Operator`-object
- get system matrix with `operator.systemMatrix();`

Algebraic classes (assembling of system matrix and right hand side):

To assemble the right hand side in our system of equations:
```
typedef FunctionalAssembler < FunctionalType, AffineSubspace >
   FunctionalAssemblerType;
```

To assemble the correct system matrix (with respect to the subspaces):
```
typedef OperatorAssembler < OperatorType > OperatorAssemblerType;
```

- incorporates something like:
  `assembleSystemMatrix(); constraints.apply( systemMatrix() );`

Both classes might be incorporated in a general `FunctionalSolverInterface`, so that the user does not need to care about the system assemblers.

2.2 Draft

Essential classes:

```
using namespace Functionals;

typedef Functional< DiscreteFunctionSpace > FunctionalType;
typedef Constraint < FunctionalType > ConstraintType;
typedef LinearSubspace < DiscreteFunctionSpace, ConstraintType > LinearSubspaceType;
typedef AffineSubspace < LinearSubspace > AffineSubspaceType;

// sparse row matrix of size N × N
typedef Dune::SparseRowMatrixTraits < DiscreteFunctionSpace, DiscreteFunctionSpace > MatrixObjectTraits;
typedef Operator< LinearSubspaceType, AffineSubspaceType,
   MatrixObjectTraits > DiffOperatorType;

// algebraic system assemblers:
typedef OperatorAssembler < OperatorType > OperatorAssemblerType;
typedef FunctionalAssembler < FunctionalType, AffineSubspace > FunctionalAssemblerType;

// CG scheme
typedef CGInverseOp< DiscreteFunctionType, OperatorAssembler > InverseOperatorType;
```

Main code:

```
DiscreteFunctionType rhs( "right hand side", discreteFunctionSpace );

// use a right hand side assembler class to apply 'functional+constraints' to right hand side vector
FunctionalAssemblerType rhsAssembler ( functional, affineSubspace );
rhsAssembler.assemble( rhs );

// behaves like the old Operator-class of DUNE-FEM:
OperatorAssemblerType systemMatrixAssembler ( differentialOperator );
// 'differentialOperator' contains correct 'systemMatrix()':
InverseOperatorType cg( systemMatrixAssembler, 1e-6, 1e-8 );
cg( rhs, solution );
```

We might think about hiding this main code behind a 'FunctionalSolverInterface', so that the user can simply call:

```
    cg( differentialOperator, functional, affineSubspace, solution );
```

(i.e. `FunctionalSolverInterface< Operator, Functional, AffineSubspace>`)

Comparison with 'old' main code (for laplace operator and zero boundary condition):

```
DiscreteFunctionType rhs( "rhs", discreteFunctionSpace );
AssembledFunctional< FunctionalType > rhsFunctional ( disceretFunctionSpace, functional );
rhsFunctional.assemble( rhs );

typedef LaplaceOperator< DiscreteFunctionType, MatrixObjectTraits > LaplaceOperatorType;
// apply constraints
bool hasDirBoundary = constraints.apply( laplaceOperator.systemMatrix(), rhs, solution );

InverseOperatorType cg( laplaceOperator, 1e-6, 1e-8 );
cg( rhs, solution );
```

The essential difference is that the (differential)operator already knows the correct system matrix (due to the subspaces, that know the constraints). Therefore the user does not need some kind of 'constraints.apply' method (this happens internally in the two system assemblers).

3 Realization of Functionals

Class 3.1 (`Functional< Space >`).
Represents a functional f. This class comes without any functionality at the moment, until someone comes up with a reasonable example of nonlinear functionals.

`number = operator( function )`	*Given u, returns $f[u]$.*

3.1 Linear Functionals

Definition 3.2 (Linear functional).
Let V be a vector space, $\mathbb{K}$ its underlying scalar field and f a functional. If, for all $u,v \in V$ and for all $\lambda,\mu \in \mathbb{K}$,

$$f[\lambda u] + f[\mu v] = \lambda f[u] + \mu f[v] \tag{3.1}$$

holds, f is called a linear functional.

Definition 3.3 (Dual Space).
Let V be a vector space and $\mathbb{K}$ its underlying scalar field. The space

$$V^* := \left\{ f : V \to \mathbb{K} \middle| f \textit{ linear functional} \right\}$$

is called the dual space *of V and is a vector space itself.*

Lemma 3.4 (Localization property of linear functionals).
Let V_G be a discrete function space ([1, Def. 18]) and $f \in V_G^*$ a linear functional. Let further be

$$u = \sum_{E \in G} \sum_{i \in I_E} u_i^E \varphi_i^E \tag{3.2}$$

the representation for a $u \in V_G$ in terms of its local DoFs u_i^E and the local base functions φ_i^E ([1, Def. 20]). Then it holds that

$$f[u] = \sum_{E \in G} \sum_{i \in I_E} u_i^E f\left[\varphi_i^E\right], \tag{3.3}$$

which can also be written as

$$f[u] = \sum_{E \in G} u^E \cdot f[B_E]^E, \tag{3.4}$$

where $u^E := (u_i^E)_{i \in I_E}$ is the local DoF vector of u on E and $f[B_E]^E$ is defined as the vector

$$f[B_E]^E := \left(f\left[\varphi_i^E\right] \right)_{i \in I_E} \tag{3.5}$$

for a local basfunction set B_E.

Class 3.5 (`LinearFunctional< Space, DiscreteFunctionSpace >:Functional`).
Represents a linear functional f.

`number`
` = operator( function )`
	Redefines `Functional::operator()`. *Given u, computes* $\sum_{E \in G} u^E \cdot f[B_E]^E$ *by doing a gridwalk and calling* `applyLocal()` *on each entity.*

`vector`
` = applyLocal( localBasefunctionSet )`
	Implements $f[B_E]^E$. Given B_E, computes $(f[\varphi_i^E])_{i \in I_E}$

3.2 Integral Functionals

Definition 3.6 (Integral functional).
Let V be a vector space, $u \in V$ and $f \in V^*$. If $f[u]$ can be decomposed as

$$f[u] = \sum_{c=0}^{dim} f^c[u], \tag{3.6}$$

where $f^c \in V^*$ are codim c integral functionals, *which can be written as*

$$f^c[u] = \int_{\omega^c} \tilde{f}^c[u] \tag{3.7}$$

for a set ω^c of codimension c and a functional $\tilde{f}^c \in V^*$, then f is called an integral functional.

Lemma 3.7 (Localization property of integral functionals).
Let V_G be a discrete function space and $f \in V_G^*$ an integral functional. Then it holds that

$$\begin{aligned} f[u] &= \sum_{E \in G^0} \sum_{i \in I_E} u_i^E \sum_{c=0}^{dim} f^c[\varphi_i^E] \\ &= \sum_{E \in G^0} u^E \cdot \left(\sum_{c=0}^{dim} \int_{G_E^0} \tilde{f}^c[B_E]^E \right), \end{aligned} \tag{3.8}$$

where $(\dots)$ is to be understood as the vector

$$\left(\sum_{c=0}^{dim} \int_{G_E^0} \tilde{f}^c[B_E]^E \right) := \left(\sum_{c=0}^{dim} \int_{G_E^c} \tilde{f}^c[\varphi_i^E] \right)_{i \in I_E}, \tag{3.9}$$

where G_E^c is "the set of all codim c entities, that lie inside E".

Class 3.8 (`LocalOperationProvider`).
Represents the operation $\tilde{f}^c[\varphi_i^E]$, e.g. $\tilde{f}^c[\varphi_i^E] = f(x)\varphi_i^E(x)$. This class has to be provided by the user in order to define a `CodimIntegralFunctional` *(see below).*

`number = apply( function, localPoint, functionalFunction = 1 )`	*Given a point x in local coordinates, returns $\tilde{f}^c[\varphi_i^E](x)$, where φ_i^E is given as* `function` *and some function associated with the functional can be given as* `functionalFunction`

Class 3.9 (`CodimIntegralFunctional< LocalOperationProvider >:LinearFunctional`).
Represents a codim c functional $f^c[u]$. A `CodimIntegralFunctional` *provides an additional method* `prepareLocalIntegration()` *to facilitate the integration in* `IntegralFunctional::applyLocal()` *(see below). There should be derived classes for each codimension, which, together with a suitable* `LocalOperationProvider`*, can be given to an* `IntegralFunctional` *to provide something like an* `L2Functional` *for the user.*

`number = operator( function )`	*Inherited from* `LinearFunctional`*.*
`vector = applyLocal( localBasefunctionSet )`	*Redefines* `LinearFunctional::applyLocal()`*. Given B_E, computes $(f^c[\varphi_i^E])_{i \in I_E}$ by doing a codim c integration by quadrature and calling* `prepareLocalIntegration()` *for each quadrature point.*
`vector = prepareLocalIntegration( localBasefunctionSet localPoint )`	*Given B_E and a point x in local coordiantes, returns $\tilde{f}^c[\varphi_i^E](x)$ by calling the underlying* `LocalOperationProvider`*.*

Class 3.10 (`IntegralFunctional< CodimIntegralFunctionals >:LinearFunctional`)**.** *Represents an integral functional. This is like a* `CombinedLinearFunctional` *(see somewhere), but the integration in* `applyLocal()` *is only done once, calling* `prepareLocalIntegration()` *on each* `CodimIntegralFunctional`*.*

`number = operator( function )`	*Inherited from* `LinearFunctional`*.*
`vector = applyLocal( localBasefunctionSet )`	*Redefinition of* `LinearFunctional::applyLocal()`*. Given B_E, computes $(f^c[\varphi_i^E])_{i\in I_E}$ by doing a codim c integration for each given codim by quadrature and calling* `prepareLocalIntegration()` *of each* `CodimIntegralFunctional` *for each quadrature point.*

4 Realization of Constraints

Maybe, we should discuss the general concept first.

Example: `ConstraintType::DirichletConstraint dirConstraint( function );`

References

[1] A. Dedner, R. Klöfkorn, M. Nolte, and M. Ohlberger. A generic interface for parallel and adaptive discretization schemes: abstraction principles and the dune-fem module. *Computing*, 90(3-4):165–196, 2010.