

# Draft on DUNE-FEM-FUNCTIONALS

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

December 16, 2010

## Abstract

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

---

<b>Contents</b>		4.1 Functionals . . . . .	6
<b>1 Introduction</b>	<b>1</b>	4.2 Required classes . . . . .	7
<b>2 Analytical concept</b>	<b>3</b>	4.3 Draft . . . . .	9
<b>3 Examples</b>	<b>5</b>	<b>5 Realization of Functionals</b>	<b>10</b>
3.1 Elliptic PDE . . . . .	5	5.1 Linear Functionals . . . . .	11
		5.2 Integral Functionals . . . . .	12
<b>4 Programming concept</b>	<b>6</b>	<b>6 Realization of Constraints</b>	<b>14</b>

---

## 1 Introduction

The overall goal of DUNE-FEM and DUNE-FEM-FUNCTIONALS is the efficient numerical solution of PDE's. Assuming standard notation, the following elliptic PDE may serve as a sample problem.

**Example 1.1** (Elliptic PDE in one dimension).

Let  $\Omega \subset \mathbb{R}$  be a domain and  $a, f : \Omega \rightarrow \mathbb{R}$  and  $g : \partial\Omega \rightarrow \mathbb{R}$  be given functions. Find  $u : \Omega \rightarrow \mathbb{R}$ , such that

$$\begin{aligned} -\nabla \cdot (a \nabla u) &= f && \text{in } \Omega, \\ u &= g && \text{on } \partial\Omega. \end{aligned} \tag{1.1}$$

**Definition 1.2.** *Weak formulation*

Let  $H^1$  and  $H_0^1$  be given as usual and let  $H_g^1$  for  $g \in H^1$  be defined as

$$H_g^1 := \{v \in H^1 \mid v = w + g \text{ for a } w \in H_0^1\}.$$

The weak formulation of problem (1.1) then reads as follows. Find  $u \in H_g^1$ , such that

$$\int_{\Omega} a \nabla u \nabla v \, dx = \int_{\Omega} f v \, dx \quad \text{for all } v \in H_0^1. \quad (1.2)$$

The weak formulation (1.2) gives rise to the introduction of functionals and operators. A rigorous mathematical definition of these can be found in the next section. The following is only intended to give the basic idea.

**Definition 1.3** (Operators and functionals).

The function  $f$  from the original problem 1.1 induces a functional

$$\begin{aligned} F : H_0^1 &\rightarrow \mathbb{R} \\ v &\mapsto F[v] := \int_{\Omega} f v \, dx. \end{aligned}$$

Accordingly the function  $a$  from the original problem 1.1 induces an operator

$$\begin{aligned} A : H_g^1 &\rightarrow H^{-1} \\ u &\mapsto A(u), \end{aligned}$$

where  $A(u)$  itself is a functional, defined by

$$\begin{aligned} A(u) : H_0^1 &\rightarrow \mathbb{R} \\ v &\mapsto A(u)[v] := \int_{\Omega} a \nabla u \nabla v \, dx. \end{aligned}$$

With these definitions at hand the weak formulation (1.2) can be rewritten in the following way.

**Remark 1.4** (Variational formulation using functionals and operator).

Let  $A$  and  $F$  be as in definition 1.3. The weak formulation (1.2) can be rewritten as follows. Find  $u \in H_g^1$ , such that

$$A(u)[v] = F[v] \quad \text{for all } v \in H_0^1. \quad (1.3)$$

Our postulate is, that a wide range of interesting problems can be written in this form. For detailed examples of linear and nonlinear problems see section ?.

## 2 Analytical concept

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 2.1** (Function and Functionspace).

Let  $n, d \in \mathbb{N}^{\geq 1}$  be integers and  $\Omega \subseteq \mathbb{R}^n$  a subset. A mapping  $v : \Omega \rightarrow \mathbb{R}^d$  is called a function, the set

$$V := \{v : \Omega \rightarrow \mathbb{R}^d\}$$

is called a functionspace. If  $V$  is an  $\mathbb{R}$ -vector space,  $V$  is called a linear functionspace.

**Definition 2.2** (Functional).

Let  $V$  be a function space. A map

$$\begin{aligned} F : V &\rightarrow \mathbb{R}, \\ v &\mapsto F[v] \end{aligned} \tag{2.1}$$

is called a Functional. If

$$F[\alpha v_1 + \beta v_2] = \alpha F[v_1] + \beta F[v_2]$$

holds for all  $\alpha, \beta \in \mathbb{R}$  and all  $v_1, v_2 \in V$ ,  $F$  is called a Linear Functional. The vector space

$$V' := \{F : V \rightarrow \mathbb{R} \mid F \text{ is a linear functional} \} \tag{2.2}$$

is called the dual space of  $V$ .

**Definition 2.3** (Constraint).

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

$$C : \{1, \dots, M\} \times V \rightarrow \mathbb{R} \text{ with } (i, v) \mapsto C[i][v] := F_i[v]. \tag{2.3}$$

The condition:

$$C[i][v] = 0 \quad \forall 1 \leq i \leq M \tag{2.4}$$

is called a Constraint for  $v$ .

In particular each linear functional implies a constraint.

**Definition 2.4** (Linear Subspace).

Let  $V$  be a linear function space and  $C[\cdot][\cdot] = 0$  a constraint on  $V$ . Then we call

$$V_C := \{v \in V \mid C[i][v] = 0 \ \forall i \in \{1, \dots, M\}\} \quad (2.5)$$

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

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

**Definition 2.5** (Affine space).

Let  $V$  be a function space,  $V_C$  a linear subspace and  $g \in V$ . Then we call

$$V_g := \{v + g \mid v \in V_C\} \subset V \quad (2.6)$$

an affine space with respect to  $g$  and  $V_C$ .

In general,  $V_g$  is only a subset of  $V$  and not a subspace. It will be the space of solutions in our later problem.

**Definition 2.6** (Operator).

Let  $V$  be a linear function space,  $V_C \subset V$  a linear subspace,  $V'_C$  its dual and  $V_g \subset V$  an affine space. Then we call

$$G : V_g \rightarrow V'_C \quad (2.7)$$

an operator on  $V_g$ . If

$$G(\alpha v + \beta w) = \alpha G(v) + \beta G(w)$$

holds for all  $\alpha, \beta \in \mathbb{R}$  and for all  $v, w \in V_g$  the operator  $G$  is called linear.

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

**Problem 2.7.** *Sample problem*

Let  $V$  be a linear space,  $V_C \subset V$  a linear subspace,  $F \in V'_C$  a functional and  $V_g \subset V$  an affine space. Find  $u \in V_g$ , such that

$$G(u)[v] = F[v] \quad \text{for all } v \in V_C.$$

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.

### 3 Examples

#### 3.1 Elliptic PDE

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

$$V := \left\{ v_H \in C^0(\Omega) \mid (v_H)|_T \in \mathbb{P}^1(T) \quad \forall T \in \mathcal{T}_H \right\}$$

and the linear subspace of testfunctions by  $V_0 := V \cap H_0^1(\Omega)$ . Now, let us consider the following discrete problem.

**Problem 3.1.**

For  $g \in V$  find  $u \in V$  with  $u = g$  on  $\partial\Omega$  and

$$\int_{\Omega} \nabla u \cdot \nabla v \, dx = \int_{\Omega} f v \, dx \quad \text{for all } v \in V_0.$$

(Note:  $V_0$  can not be replaced by  $V$ ).

Putting this into the general framework above, we define the (linear) functional  $F : V_0 \rightarrow \mathbb{R}$  by

$$F[v] := \int_{\Omega} f v \quad \text{for } v \in \mathring{V}.$$

$V_0$  is a constraint subspace with the constraint  $C[i][v] := v(x_i^b) = 0$  for any boundary node  $x_i^b$  (i.e.  $\mathcal{N}_H \cap \partial\Omega = \{x_1^b, \dots, x_{\tilde{N}}^b\}$ ). We can therefore identify

$$V_0 = \{v \in V \mid C[i][v] = 0 \quad \forall 1 \leq i \leq \tilde{N}\} =: V_C.$$

The affine space  $V_{g_H}$  is given by

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

and the (differential) operator  $G : V_{g_H} \rightarrow V'_C$  by

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

With these notations the original problem reads:

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

## 4 Programming concept

In this section we describe the general programming concept. For a detailed description of these classes, see section ? and the doxygen documentation. These classes are realized in a namespace `Functionals` in order to avoid conflicts with other existing DUNE-FEM-classes. We do not distinguish between interfaces and realizations here, this is only intended as a conceptual overview.

### 4.1 Functionals

**Remark 4.1** (Class tree).

```
Functional
  → CombinedFunctional
  → LinearFunctional
    → CombinedLinearFunctional
    → CodimCFunctional
    → IntegralFunctional
```

**Class 4.2** (`Functional`).

*This class represents a functional  $F$  (see definition 2.2). It provides the base class for linear and nonlinear functionals.*

***Class definition:***

---

---

```
class Functional< FunctionSpaceType >
```

***Methods:***

---

---

```
RangeFieldType ret = operator( FunctionType function )
```

<i>description:</i>	<i>This method represents the functional, applied to a function.</i>
<i>in:</i>	<code>FunctionType function</code> $v$
<i>out:</i>	<code>RangeFieldType ret</code> $F[v]$

**Class 4.3** (`CombinedFunctional`).

*Given two functionals  $F, G \in V'$ , this class represents the functional*

$$F + G : V \rightarrow \mathbb{R}$$

$$v \mapsto F[v] + G[v].$$

*This class is derived from `Functional`.*

***Class definition:***

---

---

```
class CombinedFunctional< FirstFunctionalType, SecondFunctionalType >
: Functional
```

### *Methods:*

<code>RangeFieldType ret = operator( FunctionType function )</code>	
<i>description:</i>	<i>This method redefines <code>Functional::operator()</code>.</i>
<i>in:</i>	<code>FunctionType function    <math>v</math></code>
<i>out:</i>	<code>RangeFieldType ret    <math>F[v] + G[v]</math></code>

## 4.2 Required classes

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

`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.

### 4.3 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  $N \times N$ 
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 'FunctionalSolver-Interface', 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 ( discreteFunctionSpace, 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).

## 5 Realization of Functionals

**Class 5.1** (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 )	<i>Given <math>u</math>, returns <math>f[u]</math>.</i>
-------------------------------	---

## 5.1 Linear Functionals

**Definition 5.2** (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  $\lambda, \mu \in \mathbb{K}$ ,

$$f[\lambda u] + f[\mu v] = \lambda f[u] + \mu f[v] \quad (5.1)$$

holds,  $f$  is called a linear functional.

**Definition 5.3** (Dual Space).

Let  $V$  be a vector space and  $\mathbb{K}$  its underlying scalar field. The space

$$V^* := \{f : V \rightarrow \mathbb{K} \mid f \text{ linear functional}\}$$

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

**Lemma 5.4** (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

$$u = \sum_{E \in G} \sum_{i \in I_E} u_i^E \varphi_i^E \quad (5.2)$$

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

$$f[u] = \sum_{E \in G} \sum_{i \in I_E} u_i^E f[\varphi_i^E], \quad (5.3)$$

which can also be written as

$$f[u] = \sum_{E \in G} u^E \cdot f[B_E]^E, \quad (5.4)$$

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

$$f[B_E]^E := \left( f[\varphi_i^E] \right)_{i \in I_E} \quad (5.5)$$

for a local basfunction set  $B_E$ .

**Class 5.5** (LinearFunctional< Space, DiscreteFunctionSpace >:Functional).

*Represents a linear functional  $f$ .*

<b>number</b> = operator( function )	<i>Redefines Functional::operator().          Given <math>u</math>, computes <math>\sum_{E \in G} u^E \cdot f[B_E]^E</math>          by doing a gridwalk and calling          applyLocal() on each entity.</i>
<b>vector</b> = applyLocal( localBasefunctionSet )	<i>Implements <math>f[B_E]^E</math>.          Given <math>B_E</math>, computes <math>(f[\varphi_i^E])_{i \in I_E}</math></i>

## 5.2 Integral Functionals

**Definition 5.6** (Integral functional).

Let  $V$  be a vector space,  $u \in V$  and  $f \in V^*$ . If  $f[u]$  can be decomposed as

$$f[u] = \sum_{c=0}^{\dim} f^c[u], \quad (5.6)$$

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

$$f^c[u] = \int_{\omega^c} \tilde{f}^c[u] \quad (5.7)$$

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

**Lemma 5.7** (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

$$\begin{aligned} f[u] &= \sum_{E \in G^0} \sum_{i \in I_E} u_i^E \sum_{c=0}^{\dim} f^c[\varphi_i^E] \\ &= \sum_{E \in G^0} u^E \cdot \left( \sum_{c=0}^{\dim} \int_{G_E^0} \tilde{f}^c[B_E]^E \right), \end{aligned} \quad (5.8)$$

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

$$\left( \sum_{c=0}^{\dim} \int_{G_E^0} \tilde{f}^c[B_E]^E \right) := \left( \sum_{c=0}^{\dim} \int_{G_E^c} \tilde{f}^c[\varphi_i^E] \right)_{i \in I_E}, \quad (5.9)$$

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

**Class 5.8** (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).

<b>number</b> <b>= apply( function,</b> <b>  localPoint,</b> <b>  functionalFunction = 1 )</b>	<i>Given a point <math>x</math> in local coordinates, returns <math>\tilde{f}^c[\varphi_i^E](x)</math>, where <math>\varphi_i^E</math> is given as function and some function associated with the functional can be given as functionalFunction</i>
---	---

**Class 5.9** (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.

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

**Class 5.10** (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**.

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

## 6 Realization of Constraints

Maybe, we should discuss the general concept first.

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

## References

- [1] A. Dedner, R. Klöforn, 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.