Draft on Dune-Fem-Functionals

Felix Albrecht (felix.albrecht@uni-muenster.de),
Patrick Henning (patrick.henning@uni-muenster.de) and
Stefan Girke (s_girk01@uni-muenster.de).

March 20, 2011

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.mathematatik.uni-freiburg.de/.

Contents

1 Introduction

The overall goal of DUNE-FEM and DUNE-FEM-FUNCTIONALS is the efficient numerical solution of PDE's. We will present some examples in this section, that may serve as a design motivation.

1.1 finite element solution of elliptic boundary value problems

Assuming standard notation, the following elliptic PDE is one of the simplest sample problem we would like to solve with DUNE-FEM-FUNCTIONALS.

Example 1.1 (elliptic boundary value problem)**.**
Let $\Omega \subset \mathbb{R}$ be a bounded connected lipshitz-domain and let $a,f : \Omega \to \mathbb{R}$ and $g : \partial\Omega \to \mathbb{R}$ be given functions. Find $u : \Omega \to \mathbb{R}$, such that

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

Definition 1.2 (weak formulation)**.**
Let H^1 and H^1_0 be given as usual and let the affine subspace H^1_g be defined as

$$H^1_g := \big\{v \in H^1 \big| v = v_0 + \hat{g} \text{ for a } v_0 \in H^1_0\big\},$$

where $\hat{g} \in H^1$ is a H^1 representation of g. The weak formulation of problem (1.1) *then reads as follows. Find $u \in H^1_g$, such that*

$$\int\limits_\Omega a\nabla u\nabla v \; dx = \int\limits_\Omega fv \; dx \qquad \text{for all } v \in H^1_0. \tag{1.2}$$

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 (operator and functional)**.**
The function f from he original problem 1.1 induces a functional

$$\begin{aligned} F : H^1 &\to \mathbb{R} \\ v &\mapsto F[v] := \int\limits_\Omega fv \; dx. \end{aligned}$$

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

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

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

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

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

Remark 1.4 (variational problem)**.**
Let A and F be as in definition 1.3. The weak formulation (1.2) *can be rewritten as follows. Find $u \in H^1_g$, such that*

$$A(u)[v] = F[v] \qquad \text{for all } v \in H^1_0. \tag{1.3}$$

In order to solve the above problem, one can rewrite equation (1.3) using the definition of the affine subspace.

Remark 1.5 (solution of the variational problem)**.**
With the notation from remark 1.4, find $u_0 \in H^1_0$, such that

$$A(u_0)[v] = F[v] - A(\hat{g})[v] \qquad \text{for all } v \in H^1_0. \tag{1.4}$$

The solution $u \in H^1_g$ of (1.3) *is then given as*

$$u := u_0 + \hat{g}. \tag{1.5}$$

Definition 1.6 (finite element discretization)**.**
With the notation from above, let $\mathcal{T}_h$ be a conform admissable triangulation of the domain Ω with codim 0 elements $T \in \mathcal{T}_h$. The usual finite element lagrange spaces are then given by

$$S^k_h := \big\{ v_h \in C^0(\Omega) \big| v_h|_T \in \mathbb{P}^k(T) \quad \forall T \in \mathcal{T}_h \big\}, \tag{1.6}$$

$$S^k_{h0} := \big\{ v_h \in S^k_h \big| v_h = 0 \text{ on } \partial\Omega \big\} \tag{1.7}$$

and

$$S^k_{hg} := \big\{ v_h \in S^k_h \big| v_{h0} + g_h \text{ for a } v_{h0} \in S^k_{h0} \big\}, \tag{1.8}$$

where $g_h \in S^k_h$ is the projection of $\hat{g}$ onto S^k_h.

With these discrete function spaces at hand, we can define the finite element solution to problem (1.1).

Definition 1.7 (finite element solution).
With the notation from above, find $u_{h0} \in S^1_{h0}$*, such that*

$$A(u_{h0})[v_h] = F[v_h] - A(g_h)[v_h] \qquad \text{for all } v_h \in S^1_{h0}. \tag{1.9}$$

The finite element solution $u_h \in S^1_{hh}$ *of* (1.1) *is then given as*

$$u_h := u_{h0} + g_h. \tag{1.10}$$

The following algorithm gives rise to the corresponding constructs that we will nedd in Dune-Fem-Functionals.

Definition 1.8 (algorithm to solve the elliptic boundary value problem).
This algorithm computes the finite element solution of problem (1.1).

(*i*) *define the finite element space* S^1_h

(*ii*) *define the test space* $S^1_{h0} \subset S^1_h$ *as a linear subspace*

(*iii*) *define the ansatz space* $S^1_{hg} \subset S^1_h$ *as an affine subspace*

(*iv*) *define the operator* $A : S^1_h \rightarrow S^{1^{-1}}_h$

(*v*) *define the functional* $F \in S^{1^{-1}}_h$

(*vi*) *assemble a matrix with entries*

$$(A)_{i,j} := A(\varphi_i)[\psi_j] \tag{1.11}$$

for all basefunctions of the ansatz-space $\varphi_i \in S^1_{hg}$ *and all basefunctions of the test-space* $\psi_j \in S^1_{h0}$

(*vii*)) *assemble a vector with entries*

$$(F)_j := F[\psi_j] \tag{1.12}$$

for all basefunctions of the test-space $\psi_j \in S^1_{h0}$

(*viii*) *solve the algebraic system*

$$Ax = F \tag{1.13}$$

for x

(*ix*) *compute the solution* $u \in S^1_{hg}$ *as*

$$u_h := u_{h0} + \hat{g}, \tag{1.14}$$

where $u_{h0} \in S^1_{h0}$ *is the discrete function belonging to the dof vector* x.

2 Abstract concept

2.1 Functionals

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 $v : \Omega \to \mathbb{R}^d$ is called a function, *the set*

$$V := \left\{ v : \Omega \to \mathbb{R}^d \right\}$$

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

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

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*

$$V' := \left\{ F : V \to \mathbb{R} \middle| F \text{ is a linear functional } \right\} \tag{2.2}$$

is called the dual space of V.

Lemma 2.3 (Localization property of discrete linear functionals).
Let V_G be a discrete function space ([1, Def. 18]) and $F \in V_G'$ a discrete linear functional. Let further be

$$u = \sum_{E \in G} u_E \tag{2.3}$$

the representation for a $u \in V_G$ in terms of its local functions $u_E := u_{|E}$ and

$$u_E = \sum_{i \in I_E} u_i^E \varphi_i^E \tag{2.4}$$

the representation of a local function in terms of its local DoFs u_i^E *and the local base functions* φ_i^E *([1, Def. 20]). Thus,* u_E *can be written as*

$$u = \sum_{E \in G} \sum_{i \in I_E} u^E_{\mu_G(i)} \varphi^E_{\mu_G(i)},$$

where μ_G *is local–to–global DoF mapping ([1, Def. 18]). Since* F *is a discrete linear functional, it holds that*

$$F[u] = \sum_{E \in G} \sum_{i \in I_E} u^E_{\mu_G(i)} F\left[\varphi^E_{\mu_G(i)}\right], \tag{2.5}$$

which can also be written as

$$F[u] = \sum_{E \in G} u^E \cdot F[B_E]^E, \tag{2.6}$$

where $u^E := (u^E_{\mu_G(i)})_{i \in I_E}$ *is the local DoF vector of* u_E *(mapped to global) and* $F[B_E]^E$ *is defined as the vector*

$$F[B_E]^E := \left(F\left[\varphi^E_{\mu_G(i)}\right] \right)_{i \in I_E} \tag{2.7}$$

for a local basefunction set B_E*.*

An important set of linear functionals is the set of those functionals that are associated with integration, e.g the functional F, induced by the function f, which arises as a right hand side in the introductory example (see definition 1.3). To formulate the abstract idea of an "integral functional", we first have to introduce a "codim c functional", which is associated with integration over a set of codimension c.

Definition 2.4 (Codim c functional)**.**
Let V_G *be a discrete function space and* $F \in V_G'$ *a discrete linear functional. If* $F[v]$ *can be written as*

$$F[v] = \int_{\omega^c} \tilde{f}[v](x)\, dx$$

for a function $v \in V_G$*, a subset* $\omega^c \subset \Omega$ *of codimension* c *and a map*

$$\tilde{f} : V_G \to V_G,$$

F *is called a* codim c functional*. The map* $\tilde{f}$ *is called a* local operation provider*.*

Given suitable codim c functionals for all codimensions of interest we can now define an integral functional as a combination of those codim c functionals.

Definition 2.5 (Integral functional).
Let V_G be a discrete function space, $F^{c_1}, \ldots, F^{c_C} \in V_G'$ codim functionals for the codimensions $c_1, \ldots, c_C \in \mathbb{N}^{\geq 0}$ and $F \in V_G'$ a discrete linear functional. If $F[v]$ can be written as

$$F[v] = \sum_{c=c_1}^{c_C} F^c[v]$$

for a function $v \in V_G$, F is called an integral functional.

Remark 2.6 (Localization property of integral and codim c functionals).
Since integral functionals and codim c functionals are themselves discrete linear functionals we can localize the evaluation of these functionals as in lemma 2.3.

2.2 Constraints and subspaces

Definition 2.7 (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 linear functionals C by

$$C : \{1, ..., M\} \times V \to \mathbb{R} \ \textit{with} \ (i, v) \mapsto C[i][v] := F_i[v]. \tag{2.8}$$

The condition:

$$C[i][v] = 0 \ \forall 1 \leq i \leq M \tag{2.9}$$

is called a Constraint *for v.*

In particular each linear functional implies a constraint.

Definition 2.8 (Linear subspace).
Let V be a linear function space and $C[\cdot][\cdot] = 0$ a constraint on V. Then we call

$$V_C := \{v \in V | \ C[i][v] = 0 \ \forall i \in \{1, ..., M\}\} \tag{2.10}$$

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

$$V_g := \{v + g | \ v \in V_C\} \subset V \tag{2.11}$$

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

In general, V_g is only a subspace of V and not a linear subspace (in the sense, that V_G is not a vector space itself in general). It will be the space of solutions in our later problem.

2.3 Operators

Definition 2.10 (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 subspace. Then we call

$$G : V_g \to V_C' \tag{2.12}$$

an operator *on V_g. If*

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

holds for all α,$\beta \in \mathbb{R}$ 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 2.11 (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 subspace. Find $u \in V_g$, such that

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

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, $\mathcal{T}_H = \{T_1, ..., T_N\}$ 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

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

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

$$\int\limits_{\Omega} \nabla u \cdot \nabla v \; dx = \int\limits_{\Omega} f v \; dx \qquad \text{for all } v \in V_0.$$

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

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

$$F[v] := \int\limits_{\Omega} f v \qquad \text{for } v \in V_0.$$

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. $\mathcal{N}_H \cap \partial\Omega = \{x_1^b, ..., x_{\bar{N}}^b\}$). We can therefore identify

$$V_0 = \{v \in V | \; C[i][v] = 0 \;\; \forall 1 \leq i \leq \bar{N}\} =: V_C.$$

The affine subspace V_{g_H} is given by

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

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

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

With these notations the original problem reads:

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

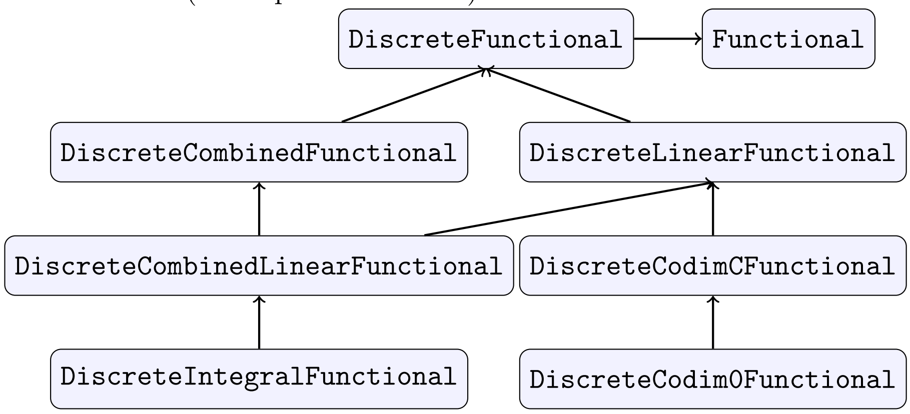
4 Realization in Dune-Fem-Functionals

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

The following diagram is intended to be a conceptual overview only. It is not intended to display a class hierarchy.

Remark 4.1 (Conceptual class tree)**.**



The class `Functional` will not really be used. It is only there to provide us with the possibility to have all the following classes in a non–discrete way in the future. The only thing this class does is to enforce the `operator()` for all functionals. The only method, the function v has to provide, is a method `evaluate( xGlobal )`.

Class 4.2 (`Functional`)**.**
This class represents a functional F (see definition 2.2). It provides the base class for discrete and what–ever–else–there–will–be functionals.

Class definition:

```
class Functional< FunctionSpaceType >
```

Methods:

`RangeFieldType ret = operator( FunctionType function )`

description:	*This method represents the functional, applied to a function.*
in:	`FunctionType function` v
out:	`RangeFieldType ret` $F[v]$

The class `DiscreteFunctional` is the actual base class for all functionals we use at the time being.

Class 4.3 (`DiscreteFunctional`).
This class represents a functional F (see definition 2.2), which can be applied to a discrete function.

Class definition:

```
class DiscreteFunctional< DiscreteFunctionSpaceType >
```

Methods:

```
RangeFieldType ret = operator(
 DiscreteFunctionType discreteFunction )
```

description:	*This method represents the functional, applied to a function.*
in:	`FunctionType function` v
out:	`RangeFieldType ret` $F[v]$

The `DiscreteCombinedFunctional` is more or less a pair of two `DiscreteFunctionals`. When its `operator()` is called, it just calls each `operator()` and adds the results.

Class 4.4 (`DiscreteCombinedFunctional`).
Given two functionals $F,G \in V'$, this class represents the functional

$$\begin{aligned} F + G : V &\to \mathbb{R} \\ v &\mapsto F[v] + G[v]. \end{aligned}$$

This class is derived from `DiscreteFunctional`.

Class definition:

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

Methods:

```
RangeFieldType ret = operator(
 DiscreteFunctionType discreteFunction )
```

description:	*This method redefines* `DiscreteFunctional::operator()`.
in:	`DiscreteFunctionType discreteFunction` v
out:	`RangeFieldType ret` $F[v] + G[v]$

The `DiscreteLinearFunctional` is the base class for a wide range of interesting functionals – linear functionals. As stated in lemma 2.3, the most important property of linear functionals is, that its application to a discrete function can be split up and carried out by some kind of local application of the functionals to a local function.

Class 4.5 (`DiscreteLinearFunctional`).
This class represents a linear functional F (see definition 2.2). All linear functionals have to provide the method `applyLocal()` *in addition to the method* `operator()`*. This class is derived from* `DiscreteFunctional`*.*

Class definition:

```
class DiscreteLinearFunctional< DiscreteFunctionSpaceType >
: DiscreteFunctional
```

Methods:

```
RangeFieldType ret = operator(
 DiscreteFunctionType discreteFunction )
```

description:	*This method redefines* `DiscreteFunctional::operator()`*. It makes use of the localization property of linear functionals (see lemma 2.3). It is implemented as a grid walk over all codim 0 entities which calls the method* `applyLocal()` *on each entity. Thus each linear functional only has to implement the method* `applyLocal()`*.*
in:	`DiscreteFunctionType discreteFunction` v
out:	`RangeFieldType ret` $F[v]$

```
void applyLocal(
 LocalBasefunctionSetType localBasefunctionSet,
 LocalDoFVectorType returnVector )
```

description:	*Given the localization property of a linear functional (see lemma 2.3), this method implements the vector $F[B_E]^E$. This is still only a rough idea of this methods signature. It is highly possible that we will need to extend this method to take additional arguments, such as the neighbors local basefunction set etc...*
in:	`LocalBasefunctionSetType localBasefunctionSet` B_E
out:	`LocalDoFVectorType returnVector` $F[B_E]^E := \left(F\left[\varphi^E_{\mu_G(i)}\right]\right)_{i\in I_E}$

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, 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, DiscreteFunctionType >
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 $\to$ (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
- *Algebraic representation:* get system matrix with `operator.systemMatrix();` or `operator.algebraic();`
- incorporates something like:
  `SystemMatrix(); constraints.apply( systemMatrix() );`, `algebraic(); constraints.apply( algebraic() );` respectively

```
typedef FunctionalAssembler< FunctionalType, AffineSubspace >
FunctionalAssemblerType;
```

- assembles the right hand side in our system of equations
- *Algebraic representation:* get assembled functional with `functional.algebraic()`

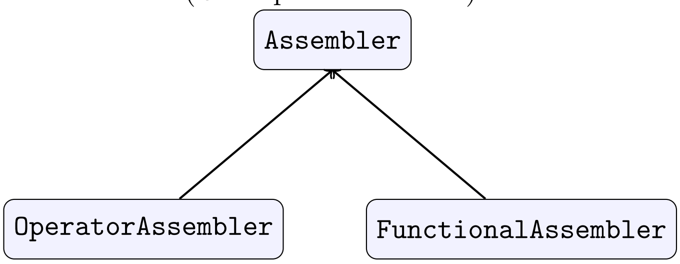
- incorporates something like:
  `algebraicl(); constraints.apply( functional );`

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

4.2.1 Suggestion for an alternative ”Assembler“ design

Maybe we should think about the operator and ”assembled functional” design again. Here is another abstract approach:

Remark 4.6 (Conceptual class tree)**.**




```
typedef Assembler < AssemblerTraits > AssemblerType;
```

- Assembler interface class.
- Each class derived from this interface defines a type `AlgebraicRepresentationType`, which defines how the algebraic representation is stored - for example a vector or matrix.
- A method `algebraic()` with return type `AlgebraicRepresentationType` is introduced here.

```
typedef OperatorAssembler< LinearSubspaceType, AffineSubspaceType>
OperatorAssemblerType;
```

- Derived from `Assembler`.
- `typedef MatrixObject AlgebraicRepresentationType`.
- The method `algebraic()` represents the algebraic representation for the system matrix, i.e. the assembled system matrix.

```
typedef FunctionalAssembler < FunctionalType, AffineSubspace >
FunctionalAssemblerType;
```

- Derived from `Assembler`.

- typedef VectorObject AlgebraicRepresentationType.
- The method algebraic() represents the algebraic representation, i.e. the assembled functional.

No we can define the Operator and FunctionalAssembler in a different way:
typedef Operator< FunctionalType, AffineSubspace, AssemblerType > DifferentialOperatorType;
Note: We can also define typedef Operator< AssemblerType > DifferentialOperatorType by extracting the space types from the assembler.

- Same properties as above...
- Define a private method void applyConstraints(algebraicRepresentation) where the constraints are extracted and applied to the algebraic representation. Now we can implement a general assemble method just by writing applyConstraints(operator.algebraic()).

typedef RHSFunctional< FunctionalType, AffineSubspace, AssemblerType >
RHSFunctionalType;
Note: We can also define typedef RHSAssembler< AssemblerType > RHSAssemblerType by extracting the space types from the assembler.

- Same properties as above...
- Define a private method applyConstraints(algebraicRepresentation) where the constraints are extracted and applied to the algebraic representation. Now we can implement a general assemble method just by writing applyConstraints(functional.algebraic()).

4.3 Draft

```
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 x N
typedef Dune::SparseRowMatrixTraits < DiscreteFunctionSpace, DiscreteFunctionSpace >
  MatrixObjectTraits;
typedef Operator< LinearSubspaceType, AffineSubspaceType, MatrixObjectTraits >
  DiffOperatorType;

// algebraic system assembler:
```

```
typedef FunctionalAssembler < FunctionalType , AffineSubspace >
  FunctionalAssemblerType;

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

Main Code:

```
//constraints, for example dirichlet constraints
ConstraintsType constraints( gridPart );

//subspaces
LinearSubspaceType linearSubspace( constraints );
AffineSubspaceType affineSubspace( linearSubspace, discretefunction );

// operator (behaves like the old Operator-class of dunefem)
DiffOperatorType differentialOperator ( linearSubspace, affineSubspace );

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.algebraic( rhs );

// 'differentialOperator' contains correct 'systemMatrix()':}
InverseOperatorType cg( differentialOperator, 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 to the old main code (for laplace operator and zero boundary condition):

```
DiscreteFunctionType rhs( "rhs", discreteFunctionSpace );
typedef AssembledFunctional< FunctionalType > AssembledFunctionalType;
AssembledFunctionalType rhsFunctional ( discretFunctionSpace, 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

5.1 Integral Functionals

Definition 5.1 (Integral functional)**.**
Let V be a vector space, $u \in V$ and $f \in V^*$. If $f[u]$ can be decomposed as

$$f[u] = \sum_{c=0}^{dim} f^c[u], \tag{5.1}$$

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

$$f^c[u] = \int_{\omega^c} \tilde{f}^c[u] \tag{5.2}$$

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

Lemma 5.2 (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

$$\begin{aligned} f[u] &= \sum_{E \in G^0} \sum_{i \in I_E} u_i^E \sum_{c=0}^{dim} f^c[\varphi_i^E] \\ &= \sum_{E \in G^0} u^E \cdot \left(\sum_{c=0}^{dim} \int_{G_E^0} \tilde{f}^c[B_E]^E \right), \end{aligned} \tag{5.3}$$

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

$$\left(\sum_{c=0}^{dim} \int_{G_E^c} \tilde{f}^c[B_E]^E \right) := \left(\sum_{c=0}^{dim} \int_{G_E^c} \tilde{f}^c[\varphi_i^E] \right)_{i \in I_E}, \tag{5.4}$$

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

Class 5.3 (`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 5.4 (`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`*.*

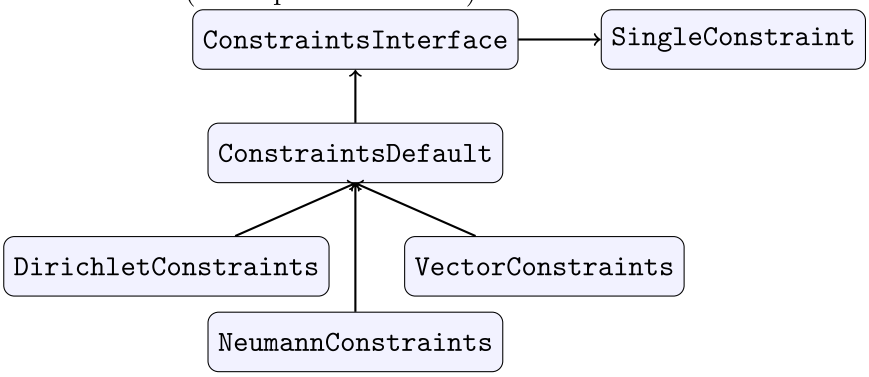
Class 5.5 (`IntegralFunctional< CodimIntegralFunctionals >:LinearFunctional`).
Represents an integral functional. This is like a `CombinedLinearFunctional` *(see somewhere), but the integration in* `applyLocal()` *is only done once, calling*
`prepareLocalIntegration()` *on each* `CodimIntegralFunctional`*.*

`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.*

6 Realization of Constraints

We should discuss about the general concept, especially about the methods `operator()`, `apply()` and `applyLocal()`.

Remark 6.1 (Conceptual class tree)**.**



Remark 6.2 (SingleConstraint vs. ConstraintsInterface)**.**
The next figure shows the class concept again - concentrating on the difference between a single constraint and classes derived from the class `ConstraintsInterface`*. Here we should discuss about some implementation details, especially about the Interface class. (Is it convient to implement a method like* `applyLocal(en)`*?)*

Class 6.3 (`SingleConstraint< FunctionalImp >`)**.** *Represents a single constraint. Mainly, this class is the formal implementation of 2.7. This class is the basic class for all other classes dealing with constraints. Normally this class is not used directly because an efficient implementation is not possible.*

Shortly said, a single constraint presents a pair of a functional *(defining the constraint) and* row number *determing the row which has to be deleted in the system matrix.*
This means, that a class (for example representing the dirichlet constraints) can be (formally) seen as a vector of `SingleConstraints`*.*

`number` `= operator( discretefunction )`	*Computes $F[v]$, where F is the functional and v the discrete function.*
`vector` `= apply( discreteFunction)`	*just calls* `applyLocal()` *for each entity.*

`localVector`
`= applyLocal( entity,`
`localBasefunctionSet )`	

`int` `= getRow()`	*returns the row number which has to be deleted in the system matrix.*
`functional` `= getFunctional()`	*returns the functional representing the constraint, i.e. $F[v] \stackrel{!}{=} 0$ for the functional F returned by this function.*

Class 6.4 (`ConstraintsInterface< FunctionalImp >`). *This class represents an interface for all classes. Mention that it is possible to write something like this:* `ConstraintsInterfaceImp C; C[i].apply();`*.*

`number` `= operator( discretefunction )`	*Computes $F[v]$, where F is the functional and v the discrete function.*
`vector` `= apply( discreteFunction)`	*just calls* `applyLocal()` *for each entity.*

`localVector`
`= applyLocal( entity,`
`localBasefunctionSet )`	

`int` `= size()`	*Returns number of constraints.*
`singleconstraint` `= operator[i]`	*Returns a single constraint represented by the class* `SingleConstraint`*. This method will not be used in general.*

Class 6.5 (`ConstraintsDefault< FunctionalImp >`). *Default implementation of* `ConstraintsInterface`*.*

Class 6.6 (`DirichletConstraints< FunctionalImp >`). *An example for constraints: Implementation of Dirichlet constraints.*

Derived from `ConstraintsDefault`. *Here we will* overwrite *the methods* `operator()`, `apply()` *and* `applyLocal()` *without using the single constraints from* `SingleConstraint`.

`number = operator( discretefunction )`	*Computes $F[v]$, where F is the functional and v the discrete function.*
`vector = apply( discreteFunction)`	*just calls* `applyLocal()` *for each entity.*
`localVector = applyLocal( entity, localBasefunctionSet )`	
`int = size()`	*Returns number of constraints.*
`singleconstraint = operator[i]`	*Returns a single constraint represented by the class* `SingleConstraint`. *This method will not be used in general.*

Class 6.7 (`NeumannConstraints< FunctionalImp >`). *An example for constraints: Implementation of Neumann constraints.*
Derived from `ConstraintsDefault`. *See above.*

Class 6.8 (`VectorConstraints< FunctionalImp >`). *An example for constraints: Here the constraints are stored explicitly in a vector of elements from the class* `SingleConstraint`. *Derived from* `ConstraintsDefault`. In general we do not want to have such a class.

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.