Scientific Computing – Software Concepts for Solving Partial Differential Equations

Joachim Schöberl

WS04/05

Abstract

Many problems in science and engineering are described by partial differential equations. To solve these equations on non-trivial domains, numerical methods such as the finite element method are required. In this lecture, I will shortly introduce models from mechanical and electrical engineering, and present the numerical methods and algorithms. I will focus on the design of finite element software.

1 Partial Differential Equations in Science and Engineering and the Finite Element Method

In this section, models from electrical and mechanical engineering are introduced. The arising partial differential equations are solved with the finite element package NGSolve. We will discuss the computed results and the solution procedures.

1.1 Electrostatics

The full Maxwell equations describe the interaction of electric and magnetic fields. In a stationary limit, the electric fields can be modeled by a scalar equation, only. Electrostatics models for example a charged capacitor.

The involved quantities are:

Symbol	Unit	
Φ	V	electrostatic potential
E	V/m	electric field intensity
D	As/m^2	dielectric displacement current density
ρ	As/m^3	charge density

Here, V is the abbreviation for Volt, and A is short for Ampere.

The quantities are related by

$$E = \nabla\Phi \qquad D = \varepsilon E \qquad \rho = -\operatorname{div} D. \tag{1}$$

The material parameter ε is the dielectric coefficient.

Putting together the equations above, one ends up with the second order scalar equation

$$-\operatorname{div}(\varepsilon\nabla\Phi) = \rho \tag{2}$$

Still, the potential Φ and the charge density ρ are two unknown fields, and we need more information. Assume the domain consists of conductors and insulators. Then

- inside a conductor the voltage is constant. This implies that E, D, and ρ vanish inside the conductor,
- there are no charges inside of an insulator.

This implies that charges are allowed only at the boundary of conductors. We write ρ_S for the surface charge density.

Now, we pose the full model. Assume that the bounded domain Ω contains M separate conductors $\Omega_1^C \dots \Omega_M^C$. Let Ω^I be the complement $\Omega \setminus \cup\Omega_i^C$. On $\Gamma := \partial\Omega$ we assume $\Phi = 0$. Then, the problem is described by the boundary value problem

$$\begin{aligned} -\operatorname{div}(\varepsilon\nabla\Phi) &= 0 && \text{in } \Omega^I, \\ \Phi &= \Phi_i && \text{in } \Omega_i^C, \\ \Phi &= 0 && \text{on } \Gamma. \end{aligned}$$

The scalars Φ_i are assumed to be known. E.g., these are the applied voltages to the plate of a capacitor.

By mean of Gauss´ theorem one obtains

$$D_n = \rho_S,$$

i.e., the Neumann data for the second order equation.

1.1.1 Weak form and discretization

For shorter notation we set $\Gamma_0 := \Gamma$, $\Gamma_i = \partial\Omega_i^c$, and rename $\Omega := \Omega^I$. Then, the bvp is

$$-\operatorname{div}(\varepsilon\nabla\Phi) = 0 \quad \text{in } \Omega \tag{3}$$

with Dirichlet boundary conditions

$$\Phi = \Phi_i \quad \text{on } \Gamma_i. \tag{4}$$

We have not yet defined a function space for the unknown field Φ. It will come out naturally from the weak formulation. For this, we multiply (3) by an arbitrary smooth function v vanishing on $\cup\Gamma_i$, integrate over the domain Ω, and apply integration by parts:

$$-\int_\Omega \operatorname{div}(\varepsilon\Phi)v\,dx = \int_\Omega \varepsilon\nabla\Phi\cdot\nabla v\,dx = 0.$$

This gives now the definition of the boundary value problem in weak form. Define the function space

$$V := \{v \in L_2(\Omega) : \nabla v \in L_2\}.$$

That space is the Sobolev space $H^1(\Omega)$, which is an Hilbert space with inner product $(u,v)_{L_2} + (\nabla u, \nabla v)_{L_2}$. Now, search $\Phi \in V$ such that

$$\Phi = \Phi_i \qquad \text{on } \Gamma_i$$

and

$$\int_\Omega \varepsilon\nabla\Phi\cdot\nabla v\,dx = 0 \qquad \forall\, v \in V \text{ s.t. } v = 0 \text{ on}\Gamma_i.$$

Due to the choice of the space (H^1 has well defined boundary values, trace theorem), this is a well posed formulation. Indeed, there is a unique solution in V, which follows from the inverse trace theorem, and the Lax-Milgram theorem.

An equivalent formulation (to the weak one) is the constrained minimization problem

$$\min_{\substack{v\in V\\ v=\Phi \text{ on } \Gamma_i}} \int \varepsilon\,|\nabla v|^2\,dx. \tag{5}$$

Exercise: Show the equivalence

For two reasons which will become clear later, we replace the constrained minimization problem by a penalty approximation with 'large' penalty parameter α:

$$\min_{v\in V} \int_\Omega \varepsilon\,|\nabla v|^2\,dx + \alpha\int_{\cup\Gamma_i}(v-\Phi_i)^2\,ds. \tag{6}$$

The corresponding variational form is to find $v \in V$ such that

$$\int_\Omega \varepsilon\nabla\Phi\cdot\nabla v\,dx + \alpha\int_\Gamma \Phi v\,ds = \alpha\int_\Gamma \Phi_i v\,ds \qquad \forall\, v \in V. \tag{7}$$

Now, the identity holds for all $v \in V$ without restriction onto the boundary values.

By performing integration by parts again, one obtains the according b.c. in strong form

$$\varepsilon\frac{\partial\Phi}{\partial n} + \alpha\Phi = \alpha\Phi_i,$$

which is a b.c. of Robin type.

1.1.2 Finite Element Discretization

For the numerical treatment of (7) on replaces the infinite dimensional function space V by a space V_N of finite dimension N. The finite element method is one possibility to construct spaces V_N. The domain Ω is sub-divided into simple domains. Most popular are triangles and quadrilaterals in 2D, and tetrahedra and (deformed) cubes in 3D. On these simple domains, the approximation functions are usually polynomials.

.... basis functions, shape functions, dofs to ensure continuity, hat functions ....

Functions in V_N are expanded in the basis $\{p_1, \ldots, p_N\}$

$$\Phi_N(x) = \sum_{i=1}^{N} u_i p_i(x).$$

The so called Galerkin approximation to (7) is to find $\Phi_N \in V_N$ such that

$$\int_\Omega \varepsilon \nabla \Phi_N \cdot \nabla v_N \, dx + \alpha \int_\Gamma \Phi_N v_N \, ds = \alpha \int_\Gamma \Phi_i v_N \, ds \qquad \forall\, v_N \in V_N. \tag{8}$$

We plug in the expansion of Φ_N. Testing for all $v \in V_N$ is equivalent to test for all basis functions. This leads to: Find $u = (u_1, \ldots, u_N) \in \mathbb{R}^N$ such that

$$\sum_{i=1}^{N} \Big\{ \underbrace{\int_\Omega \varepsilon \nabla p_i \cdot \nabla p_j \, dx + \alpha \int_\Gamma p_i p_j \, ds}_{=:A_{ji}} \Big\} u_i = \underbrace{\alpha \int_\Gamma \Phi_i p_j \, ds}_{=:f_j} \qquad \forall\, j \in \{1, \ldots N\}. \tag{9}$$

This is the linear system of equations

$$Au = f.$$

The equation can either be solved by a direct elimination method, or, by an iterative method such as the preconditioned conjugate gradients method. (Sparse) direct methods are appropriate for problems of moderate size (about up to 200 000 unknowns for 2D problems, and 40 000 for 3D problems), but suffer from large memory and CPU requirements for large problems. Iterative methods depend on the type of the applied preconditioner, i.e., an inexact inverse. A simple preconditioners is the diagonal of the matrix, good ones are multigrid and more general Additive Schwarz methods. Preconditioning will be discussed later in Section ..

1.1.3 Simulating a capacitor with NGSolve

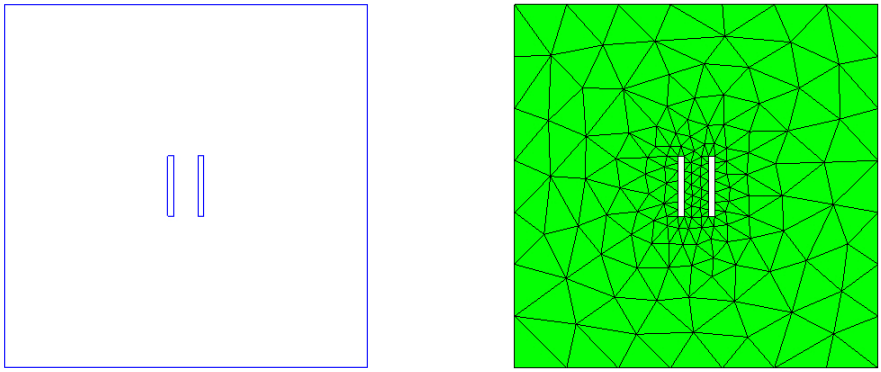
We simulate a plate capacitor as drawn below. The capacity is defined as

$$C = \frac{Q}{U},$$

where $Q = \int_{\Gamma_+} q_S \, ds$ is the total charge on one plate, and $U = \Phi_+ - \Phi_-$ is the voltage (=difference of potential) between the plates.

Exercise: Show that the capacity is related to (twice of) the stored energy

$$CU^2 = \int_\Omega E \cdot D \, dx.$$



The following Netgen input file describes the geometry above. First, a list of 12 points is specified. The entries are x and y coordinates, and a local relative mesh refinement close to this point. Then, a list of 12 line segments is specified. The first parameters give left and right sub-domain number, 0 means outside. The next number specifies the type of curve, 2 means straight line between the next 2 points. The last number is the relative refinement along this line. Finally, with the `-bc=1` flag, a boundary condition number is specified. We set bc=1 on the outer boundary, and 2 and 3 for the two plates, respectively.

A run of Netgen with this input file generates the triangular mesh drawn above.

```
splinecurves2d
5

12
-3      -3      1
3       -3      1
3       3       1
-3      3       1

0.2     -0.5    10
0.3     -0.5    10
0.3     0.5     10
0.2     0.5     10
-0.3    -0.5    10
-0.2    -0.5    10
-0.2    0.5     10
-0.3    0.5     10
```

```
12
1       0       2       1       2       1  -bc=1
1       0       2       2       3       1  -bc=1
1       0       2       3       4       1  -bc=1
1       0       2       4       1       1  -bc=1

0       1       2       5       6       1  -bc=2
0       1       2       6       7       1  -bc=2
0       1       2       7       8       1  -bc=2
0       1       2       8       5       1  -bc=2

0       1       2       9       10      1  -bc=3
0       1       2       10      11      1  -bc=3
0       1       2       11      12      1  -bc=3
0       1       2       12      9       1  -bc=3
```

The pde we want to solve involves the bilinear form

$$A(\Phi, v) = \int_{\cup\Omega_i} \varepsilon_i \nabla\Phi \cdot \nabla v + \int_{\cup\Gamma_i} \alpha_i \Phi v.$$

The coefficients ε_i and α_i can be specified for each sub-domain, and each piece of the boundary, respectively. There is just one sub-domain, i.e., $\varepsilon = (\varepsilon_1) = (1)$. There are three parts of the boundary, thus $\alpha = (0, 1e5, 1e5)$, where 10^5 is the chosen 'large' penalty parameter.

We apply $+1V$ and $-1V$ onto the electrodes. Thus the right hand side functional is

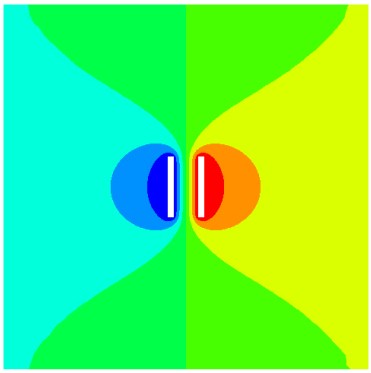
$$f(v) = 10^5 \Big\{ \int_{\Gamma_2} 1\, v\, ds + \int_{\Gamma_3} -1\, v\, ds \Big\} = \int_{\cup\Gamma_i} g_i v\, ds.$$

The coefficient g takes the values $(0, 10^5, -10^5)$ on the pieces Γ_i of the boundary.

The NGSolve input file specifiing that variational problem is given below. First, the filenames of the prepared geometry and mesh files must be specified. Then, one defines coefficient functions, finite element spaces, gridfunctions, bilinear-forms, and linearforms as required by the weak formulation. Several flags are possible to adjust the components. A preconditioner defines an (inexact) inverse. The action starts with the `numproc` . The numproc `bvp` takes the matrix provided by the bilinear-form, a right hand side vector provided from the linear-form,and the vector from the gridfunction, and solves the linear system of equations. Each numproc has a unique name such as `np1`.

```
geometry = demos/capacitor.in2d
mesh = demos/capacitor.vol

define constant geometryorder = 1
# define constant refinep = 1
```

```
define coefficient coef_eps
1,

define coefficient coef_alpha
0, 1e5, 1e5,

define coefficient coef_g
0, 1e5, -1e5,

define fespace v -order=1
define gridfunction u -fespace=v

define bilinearform a -fespace=v -symmetric
laplace coef_eps
robin coef_alpha

define linearform f -fespace=v
neumann coef_g

# define preconditioner c -type=direct -bilinearform=a
# define preconditioner c -type=local -bilinearform=a
define preconditioner c -type=multigrid -bilinearform=a -smoothingsteps=1


numproc bvp np1 -bilinearform=a -linearform=f -gridfunction=u -preconditioner=c -maxsteps=1000

numproc drawflux np2 -bilinearform=a -solution=u -label=flux


# evaluate energy:

define bilinearform aeval -fespace=v -symmetric -nonassemble
laplace coef_eps
numproc evaluate npeval  -bilinearform=aeval -gridfunction=u -gridfunction2=u


# error estimator:

define fespace verr -l2 -order=0
define gridfunction err -fespace=verr

numproc zzerrorestimator np3 -bilinearform=a -linearform=f -solution=u -error=err -minlevel=1
numproc markelements np4 -error=err -minlevel=1 -factor=0.5
```



Computed (double) energy $\int D \cdot E \, dx = \int \varepsilon |\nabla \Phi|^2 \, dx$ is 16.069, which gives a capacity of

$$C = \frac{16.069}{2^2} = 4.017 \frac{As}{V}.$$

1.2 Elasticity

We want to model the mechanical deformation of a body due to applied forces. A body is called elastic, if the deformation returns to the initial state after removing the forces. Otherwise, it is called elasto-plastic. We restrict ourself to the elastic behavior.

1.2.1 One dimensional elasticity

We start with a one-dimensional model. Take a beam (a, b) which is loaded by a force density f in longitudinal (x) direction. We are interested in the displacement $u(x)$ in x-direction.

The involved variables are

- The *deformation* Φ, unit is [m]. The point x of the initial configuration is moved to the point $\Phi(x)$. The *displacement* u is the difference $\Phi(x) - x$.
- The *force density* f, unit [Newton/m] is the applied load inside the body, e.g., the gravity. A *boundary load* g, unit [Newton] can be applied at the end of the beam. Its orientation is in outward direction.
- The *strain* ε, unit [1]: It describes the elongation. Take two points x and y on the beam. After deformation, their distance is $\Phi(y) - \Phi(x)$. The relative elongation of the beam is
$$\frac{(\Phi(y) - \Phi(x)) - (y - x)}{y - x} = \frac{u(y) - u(x)}{y - x}.$$
In the limit $y \to x$, this is u'. We define the strain ε as
$$\varepsilon = u'.$$

- The *stress* σ, unit [Newton]: It describes internal forces. If we cut the piece (x, y) out of the beam, we have to apply forces at x and y to keep that piece in equilibrium. This force is called stress σ. Equilibrium for an internal interval is

$$\sigma(y) - \sigma(x) + \int_x^y f(s)\, ds = 0,$$

which leads in differential form to

$$\sigma' = -f.$$

Equilibrium on an interval including the boundary (e.g., the point b) is

$$g(b) - \sigma(x) + \int_x^b f(s)\, ds = 0.$$

This leads to $\sigma n = g$, where n is the outward unit-vector.

Hook's law postulates a linear relation between the strain and the stress:

$$\sigma = E\varepsilon.$$

Combining the three equations

$$\varepsilon = u' \qquad \sigma = E\varepsilon \qquad \sigma' = -f$$

leads to the second order equation for the displacement u:

$$-(Eu')' = f.$$

Boundary conditions are

- Dirichlet b.c.: Prescribe the displacement at the boundary
- Neumann b.c: Prescibe the boundary load

The weak form is the minimization problem

$$\min_{\substack{v \in H^1 \\ v = g \text{ on } \Gamma_D}} \frac{1}{2} \int_a^b E\,(v')^2 dx - \int_a^b fv\, dx - \int_{\{a,b\}} gv\, ds.$$

The first term can be considered as energy stored due to the deformation of the body, the second and third term is work applied against external forces.

1.2.2 Elasticity in more dimensions

Now, a body $\Omega \subset \mathbb{R}^d$ is deformed due to volume and surface loads. The fields are now

- The *deformation* $\Phi : \Omega \to \mathbb{R}^d$ and the *displacement* $u = \Phi(x) - x$.
- The *volume load density* $f : \Omega \to \mathbb{R}^d$, unit $[N/m^d]$ and the *surface load density* $g : \partial\Omega \to \mathbb{R}^d$, unit $[N/m^{d-1}]$.
- The *strain* is now measured in squared relative distances:

$$\frac{\|\Phi(x+\Delta x) - \Phi(x)\|^2}{\|\Delta x\|^2} = \frac{\|\Phi'(x)\Delta x\|^2}{\|\Delta x\|^2} + O(|\Delta x|)$$

  The *Cauchy Green strain tensor*

$$C(x) := \Phi'(x)^T \Phi'(x)$$

  measures the stretching in a direction n via

$$n^T C n = \lim_{\varepsilon \to 0} \frac{\|\Phi(x+\varepsilon n) - \Phi(x)\|^2}{\|\varepsilon n\|^2}.$$

  A body is undergoing a rigid body motion (i.e., distances are kept constant) if and only if $C = I$, i.e., Φ' is an orthogonal matrix. This means $\det \Phi' \in \{+1, -1\}$. By continuity in time of the deformation process, one excludes -1. Thus a rigid body motions implies that Φ' is a rotation matrix. Φ can be a rotation plus a translation.

Now, we start from the weak formulation, i.e., the principle of minimal energy. For this, let

$$W : \mathbb{R}^{d\times d} \to \mathbb{R}$$

be a function measuring the internal energy density caused by the strain C. The total energy due to a displacement v is

$$V(v) = \int_\Omega W(C(v)) - \int_\Omega fv - \int_{\Gamma_N} gv, \tag{10}$$

and the problem is now

$$\min_{\substack{v \\ v = u_g \text{ on } \Gamma_D}} V(v). \tag{11}$$

A rigid body displacement does not cause internal energy, i.e.,

$$W(C) = 0 \qquad \text{for } C = I.$$

A simple energy funtional is the quadratic one, called Hook's law

$$W(C) = \frac{1}{8} \sum_{i,j,k,l=1}^{d} D_{ijkl}(C-I)_{ij}(C-I)_{kl} = \frac{1}{8} D(C-I) : (C-I).$$

It envolves the fourth order material tensor D. An isotropic material (same properties in all direction) has the special form

$$W(C) = \frac{\mu}{4}|C-I|^2 + \frac{\lambda}{8}(\mathrm{tr}\,(C-I))^2,$$

where μ and λ are called Lamé parameters. Here $A : B = \sum_{i,j} A_{ij}B_{ij}$ is the inner product of tensors, $|A| := (A : A)^{1/2}$ is the norm, and $\mathrm{tr}\,A = \sum_i A_{ii}$ is the trace of the tensor.

We will now evaluate the first order minimum conditions for the minimization probem (11). The directional derivative of the functional $V(u)$ into the direction v is

$$\langle V'(u), v\rangle := \lim_{t\to 0} \frac{1}{t}\left\{V(u+tv) - V(u)\right\}.$$

The first order minimum conditions claim that $V'(u) = 0$, i.e., $\langle V'(u), v\rangle = 0$ for all directions v. We use the chain rule to evaluate the derivatives:

$$\begin{aligned}\langle V'(u), v\rangle &= \int_\Omega \frac{dW(C(u))}{dC} : \langle C'(u), v\rangle - \int_\Omega fv - \int_{\Gamma_N} gv \\ &= \int_\Omega \frac{dW(C(u))}{dC} : \left\{(I+\nabla u)^T\nabla v + (\nabla v)^T(I+\nabla u)\right\} dx - \int_\Omega fv - \int_{\Gamma_N} gv\end{aligned}$$

Since $\frac{dW}{dC}$ is symmetric, and $A : (B + B^T) = 2A : B$ for symmetric tensors A, the integrand is equal to

$$2\frac{dW}{dC} : (I+\nabla u)^T\nabla v = 2(I+\nabla u)\frac{dW}{dC} : \nabla v = 2\nabla\Phi\frac{dW}{dC} : \nabla v.$$

The variational formulation is now to find u such that $u = u_g$ on Γ_D and

$$\int_\Omega 2(\nabla\Phi)\frac{dW}{dC} : \nabla v\, dx = \int_\Omega fv + \int_{\Gamma_N} gv \qquad \forall\, v \text{ s.t. } v = 0 \text{ on } \Gamma_D. \tag{12}$$

First and second Piola Kirchhoff stress tensor ...

If we plug in Hook's law $W = \frac{1}{8}D(C-I) : (C-I)$, then $\frac{dW}{dC} = \frac{1}{4}\,D(C-I)$, and we observe for the first factor

$$\begin{aligned}\nabla\Phi\frac{dW}{dC} &= \frac{1}{4}(I+\nabla u)D((I+\nabla u)^T(I+\nabla u) - I) \\ &= \frac{1}{4}D(\nabla u + (\nabla u)^T) + O((\nabla u)^2).\end{aligned}$$

In linear elasticity, one neglects the higher oder terms in ∇u. The left hand side becomes

$$\int \frac{1}{2}\{D(\nabla u + (\nabla u)^T)\} : \nabla v\, dx.$$

Again, we use that $D(\nabla u + (\nabla u)^T)$ is symmetric to use also the symmetric form for v:

$$\int \frac{1}{4}\{D(\nabla u + (\nabla u)^T)\} : \{\nabla v + (\nabla v)^T\}\, dx.$$

We introduce a new symbol, called the linearized strain operator, or the symmetric gradient operator

$$\varepsilon(u) = \frac{1}{2}\{\nabla u + (\nabla u)^T\}.$$

With this we arrived at the linear elasticity model: find $u \in [H^1(\Omega)]^d$ such that $u = u_g$ on Γ_D and

$$\int_\Omega D\varepsilon(u) : \varepsilon(v) = \int_\Omega fv\, dx + \int_{\Gamma_N} gv\, ds \qquad \forall\, v \in [H^1(\Omega)]^d \text{ s.t. } v = 0 \text{ on } \Gamma_D.$$

The symmetric tensor

$$\sigma := D\varepsilon(u)$$

is called strain tensor. It satisfies

$$\int \sigma\varepsilon(v)\, dx = \int \sigma : \nabla v\, dx = \int -(\text{div } \sigma)\cdot v + \int_{\partial\Omega} \sigma n \cdot v = \int fv + \int_{\Gamma_N} gv.$$

Thus, we have derived the strong from

$$\text{div } \sigma = f$$

and the natural boundary conditions

$$\sigma n = g \qquad \text{on } \Gamma_N$$

1.2.3 Elasticity with NGSolve

```
geometry = ngsolve/pde_tutorial/beam.geo
mesh = ngsolve/pde_tutorial/beam.vol

define constant heapsize = 100000000

define coefficient E
1,

define coefficient nu
0.2,


define coefficient penalty
```

```
1e6, 0, 0, 0, 0, 0

define coefficient coef_force
5e-5,

# finite element space with 3 components
define fespace v -dim=3 -order=5 -eliminate_internal -augmented=1
define gridfunction u -fespace=v

define linearform f -fespace=v
source coef_force -comp=3

define bilinearform a -fespace=v -symmetric  -eliminate_internal -linearform=f
elasticity E nu
robin penalty  -comp=1
robin penalty  -comp=2
robin penalty  -comp=3

# define preconditioner c -type=direct -bilinearform=a
define preconditioner c -type=multigrid -bilinearform=a

numproc bvp np1 -bilinearform=a -linearform=f -gridfunction=u -preconditioner=c -maxstep

# compute stresses:
define fespace vp -dim=6  -order=5
define gridfunction stress -fespace=vp
numproc calcflux np2 -bilinearform=a -solution=u -flux=stress -applyd
```

1.3 Magnetostatics

A second limit problem of Maxwell equations are the equations for a stationary magnetic field. Here, the involved quantities are

Symbol	Unit	
j	A/m^2	a given current density such that div $j = 0$
B	Vs/m^2	magnetic flux density (German: "Induktion")
H	A/m	magnetic field intensity (German: "Magnetische Feldstärke")

All these quantities are vector fields. It is assumed that the currents have no sources, i.e.,

$$\text{div } j = 0.$$

Ampere's law is that for every smooth surface S there holds

$$\int_S j \cdot n \, ds = \int_{\partial S} H \cdot \tau ds.$$

By Stokes´ theorem, the right hand side can be rewritten as $\int_S \text{curl } H \cdot n \, ds$. Thus, there holds

$$\text{curl } H = j$$

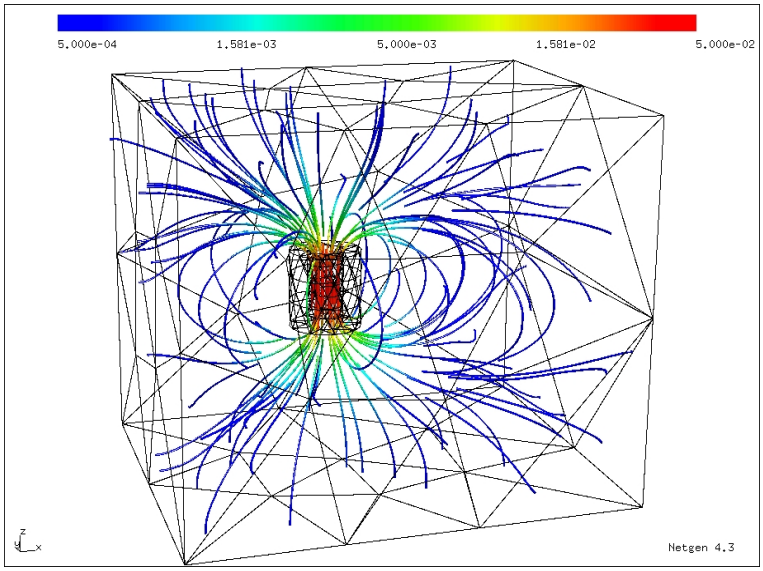
The hypotheses onto the magnetic flux B is that it has no sources, i.e.,

$$\text{div } B = 0$$

The two magnetic quantities B and H are related by the material law

$$B = \mu H,$$

where μ is called permeability. In general, the relation is non-linear, and may depend also on the history. The magnetic flux density B induced by a prescribed current j in a coil is drawn below:



The usual approach to handle these equations is to introduce a vector-potential A such that

$$B = \operatorname{curl} A$$

Since div curl $= 0$, this implies div $B = 0$. On the other hand, $\operatorname{div} B = 0$ holds in $\mathbb{R}^3$, which is simply connected, and thus allows to introduce the potential. Combining the equations leads to the second order problem for A:

$$\operatorname{curl} \mu^{-1} \operatorname{curl} A = j$$

As usual, we go over to the weak form. Multiply with test functions v and using the integration by parts rule $\int_\Omega \operatorname{curl} u \cdot v = \int_\Omega u \cdot \operatorname{curl} v + \int_{\partial\Omega} (n \times u) \cdot v \, ds$:

$$\int_\Omega \mu^{-1} \operatorname{curl} A \cdot \operatorname{curl} v \, ds + \int_{\partial\Omega} (n \times \mu^{-1} \operatorname{curl} A) \cdot v \, ds = \int_\Omega j \cdot v \, dx$$

This variational form shows two canonical boundary conditions:

- Posing the equation for arbitrary v implies that

  $$n \times \mu^{-1} \operatorname{curl} A = n \times H = 0,$$

  i.e., the tangential components of H vanish at the boundary. These are the natural Neumann boundary conditions.

- Prescribe tangential boundary conditions for A, and put the tangential components of v to 0, then

  $$(n \times \mu^{-1} \operatorname{curl} A) \cdot v = \mu^{-1} \operatorname{curl} A \cdot (v \times n) = 0$$

  These are the essential Dirichlet boundary conditions. They imply that $B \cdot n = \operatorname{curl}_n A_\tau = 0$ at the boundary.

The problem is: find A such that

$$\int_\Omega \mu^{-1} \operatorname{curl} A \cdot \operatorname{curl} v \, ds = \int_\Omega j \cdot v \, dx \qquad \forall\, v \tag{13}$$

The solution is not unique. Since $\operatorname{curl} \nabla = 0$, adding an arbitrary gradient field to A gives another solution. One possibility is to ignore the non-uniqueness, and work directly on the factor space. An other approach is to select the unique vector potential being orthogonal to gradients of arbitrary scalar fields, i.e.

$$\int A \cdot \nabla \varphi = 0 \qquad \forall\, \varphi \tag{14}$$

This additional condition is called gauging. The b.c. onto φ have to correspond to the ones of A: If A has 0-tangential b.c., then φ must have 0 b.c. as well.

1.3.1 The classical $\operatorname{curl}\operatorname{curl} + \operatorname{div}\operatorname{div}$ approach

One idea is to perform integration by parts of the gauging-equation to obtain

$$\int_\Omega A\nabla\varphi\,dx = -\int_\Omega \operatorname{div} A\,\varphi\,dx + \int_{\partial\Omega} A\cdot n\varphi\,ds = 0$$

This implies $\operatorname{div} A = 0$, and, for the case of Neumann boundary conditions also $A\cdot n = 0$. The classical approach is to utilize $\operatorname{div} A = 0$ and add a consistent term to the variational equation to obtain

$$\int \mu^{-1}\operatorname{curl}\, A\cdot\operatorname{curl}\, v\,dx + \int \mu^{-1}\operatorname{div}\, A\cdot\operatorname{div}\, v\,dx = \int f\cdot v\,dx \qquad \forall\, v,$$

with boundary conditions either $A\cdot\tau = 0$, or $A\cdot n = 0$. For this problem, standard continuous finite elements can be used. This approach was most popular until the 1990s. But, it has serios disadvantages:

- For the case of highly different material parameters μ, the stability of the equation gets really lost. Iteration numbers of iterative solvers go up, there are no robust methods available.

- This approach cannot be used in electro-dynamics, where more terms come in. There, the equation becomes (e.g., for the time harmonic setting in the so called A^* formulation)

  $$\int \mu^{-1}\operatorname{curl}\, A\operatorname{curl} v\,dx - \omega^2\int \varepsilon(x)Av\,dx = \int j\cdot v\,dx,$$

  where ε is the dielectric material parameter. Now, choosing test functions $v = \nabla\varphi$, integration by parts gives the equation

  $$\operatorname{div}(\varepsilon A) = 0,$$

  there is no choice for gauging. In sub-domains with constant coefficients ε, this implies $\operatorname{div} A = 0$ in the sub-domains, but not globally. One could use this hidden equation for stabilization:

  $$\int \mu^{-1}\operatorname{curl}\, A\operatorname{curl} v\,dx + \int \mu^{-1}\operatorname{div}(\varepsilon A)\operatorname{div}(\varepsilon v)\,dx - \omega^2\int \varepsilon Av\,dx = \int j\cdot v\,dx.$$

  But now, one cannot use continuous elements: The $\operatorname{div}-\operatorname{div}$-term is not finite in the case of non-continuous coefficients ε. There are fixes by introducing an additional scalar potential, but things don't get simpler.

1.3.2 The $H(\mathrm{curl})$ formulations

The modern approach is not to stabilize with the $\mathrm{div}-\mathrm{div}$ term. The theoretical setting is to add the gauging condition as given in (14). In general, when adding equations, one has to add also additional unknowns. We pose the mixed formulation to find the vector field A and the scalar field φ such that

$$\begin{aligned} \int \mu^{-1} \operatorname{curl} A \operatorname{curl} v \, dx \;+\; \int v \nabla \varphi &= \int j \cdot v \, dx \qquad \forall\, v \\ \int A \nabla \psi \, dx &= 0 \qquad \forall\, \psi \end{aligned}$$

The proper function space for A is the space

$$H(\mathrm{curl}) = \{v \in [L_2]^3 : \operatorname{curl} v \in [L_2]^3\},$$

with the norm

$$\|v\|^2_{H(\mathrm{curl})} = \|v\|^2_{L_2} + \|\operatorname{curl} v\|^2_{L_2}.$$

The space for the Lagrange parameter φ is the $(H^1, \|\nabla \cdot\|_{L_2})$. Continuity of the mixed formulation is immediate. The important LBB-condition

$$\sup_{v \in H(\mathrm{curl})} \frac{(v, \nabla \psi)}{\|A\|_{H(\mathrm{curl})}} \geq c\, |\nabla \psi|_{L_2}$$

is also simple to verify: Take $v = \nabla \psi$. This is possible, since

$$\nabla H^1 \subset H(\mathrm{curl}).$$

That property is essential, and will also be inherited onto the discrete level. Kernel-ellipticity is more involved, but also true.

We usually do not want to solve the mixed problem, but a positive definite one. This can be obtained by regularization: Adding a 'small' term

$$\varepsilon \int A v \, dx$$

to the mixed formulation is a regular perturbation, i.e., we change the solution of $O(\varepsilon)$. The perturbed problem is now:

$$\begin{aligned} \int \mu^{-1} \operatorname{curl} A \operatorname{curl} v \, dx + \varepsilon \int A v \, dx \;+\; \int v \nabla \varphi &= \int j \cdot v \, dx \qquad \forall\, v \\ \int A \nabla \psi \, dx &= 0 \qquad \forall\, \psi \end{aligned}$$

We choose test functions $v = \nabla \varphi$ for the first line:

$$\underbrace{\int \mu^{-1} \operatorname{curl} A \operatorname{curl} \nabla \varphi \, dx}_{=0} + \varepsilon \underbrace{\int A \nabla \varphi \, dx}_{=0} + \int \nabla \varphi \nabla \varphi = \underbrace{\int j \nabla \varphi \, dx}_{=0},$$

i.e, $\nabla \varphi = 0$, and we can solve the regularized problem in A, only:

$$\int \mu^{-1} \operatorname{curl} A \operatorname{curl} v \, dx + \varepsilon \int A v \, dx = \int j \cdot v \, dx \qquad \forall\, v \in H(\mathrm{curl})$$

With the proper methods, everything (discretization, solvers, error estimators) are robust in the regularization parameter ε.

1.3.3 Finite elements in H(curl)

We first derive the natural continuity properties of $H(\mathrm{curl})$. Let $u \in H(\mathrm{curl})$, and $q = \mathrm{curl}\, u$. Then, for all smooth vector functions φ vanishing at the boundary there holds $\int q \cdot \varphi = \int \mathrm{curl}\, u \cdot \varphi = \int u \cdot \mathrm{curl}\, \varphi$. This relation is used to define the *weak* curl operator: q is called the weak curl of u if there holds

$$\int_\Omega q \cdot \varphi = \int_\Omega u \cdot \mathrm{curl}\, \varphi \qquad \forall \varphi \in [C_0^\infty]^3$$

An L_2 function u is in $H(\mathrm{curl})$ if curl u is in L_2. A sufficient condition is the following:

Theorem 1. *Let $\Omega = \bigcup \Omega_i$ be a domain decomposition. Assume that*

$$u_i = u|_{\Omega_i} \qquad \textit{is smooth}$$

and the tangential components are continuous over sub-domain interfaces, i.e.,

$$u_i \times n_i = -u_j \times n_j \qquad \textit{on } \overline{\Omega_i} \cap \overline{\Omega_j}$$

Then $u \in H(\mathrm{curl}, \Omega)$, and the locally defined curl u *is the global, weak* curl u.

Proof. We check that the local curl satisfies the definition of the weak curl:

$$\begin{aligned}
\int_\Omega (\mathrm{curl}\, u)_i \varphi \, dx &= \sum_{\Omega_i} \int_{\Omega_i} \mathrm{curl}\, u_i \varphi \, dx \\
&= \sum_{\Omega_i} \int_{\Omega_i} u_i \,\mathrm{curl}\, \varphi \, dx + \int_{\partial\Omega_i} (n_i \times u_i)\varphi \, ds \\
&= \int_\Omega u \,\mathrm{curl}\, \varphi \, dx + \underbrace{\sum_{\overline{\Omega_i} \cap \overline{\Omega_j}} \int_{\overline{\Omega_i} \cap \overline{\Omega_j}} [n_i \times u_i + n_j \times u_j]\varphi \, ds}_{=0} .
\end{aligned}$$

□

The opposite is true is well. If $u \in H(\mathrm{curl}, \Omega)$, then u has continuous tangential components.

This characterization motivates the definition of finite element spaces for $H(\mathrm{curl})$ on the mesh $\{T\}$. The type-II Nédélec elements of order k generate the space

$$V_h^k = \{v : v|_T \in [P^k]^3,\ v \cdot t \text{ continuous over element-interfaces}\}$$

The space is constructed by defining a basis. We start with a P^1-triangular element. In 2D, the $H(\mathrm{curl})$ has 2 vector components. Each component is an affine linear function on the triangle, i.e., has 3 parameters. Totally, the P^1 triangle has 6 parameters. We demand continuity of the tangential component over element boundaries, i.e., the edges.

The tangential component is a linear function on the edge, thus, it is specified by two 2 degrees of freedom. 3 edges times 2 degrees of freedom specifiy the 6 parameters on the element.

We now give a basis. Let E_{ij} be an edge from vertex V_i to vertex V_j. Let φ_i^V and φ_j^V be the corresponding vertex basis functions. Then we define the two basis functions

$$\varphi_{ij}^{E,0} := \varphi_i^V \nabla \varphi_j^V - \nabla \varphi_i^V \varphi_j^V$$

as well as

$$\varphi_{ij}^{E,1} := \nabla(\varphi_i^V \varphi_j^V) = \varphi_i^V \nabla \varphi_j^V + \nabla \varphi_i^V \varphi_j^V$$

associated with the edge E_{ij}.
Exercise: Show that this is a basis for $V_h^1 \subset H(\mathrm{curl})$. Verify that

- $\varphi_{ij}^{E,0}$ and $\varphi_{ij}^{E,1} \subset V_h^1$
- $\varphi_{ij}^{E,0} \cdot \tau = 0$ and $\varphi_{ij}^{E,1} \cdot \tau = 0$ on edges $E \neq E_{ij}$
- $\varphi_{ij}^{E,0} \cdot \tau$ and $\varphi_{ij}^{E,1} \cdot \tau$ are linearly independent on the edge E_{ij}

In magnetostatics, we are only interested in the magnetic flux $B = \mathrm{curl}\, A$, but not in the vector potential A itself. The basis-functions $\varphi_{ij}^{E,1}$ are gradients, so do not improve the accuracy of the B-field. Thus, we might skip them, and work with the $\varphi_{ij}^{E,0}$, only. This are the Nédélec elements of the first type, also known as edge elements.

1.3.4 Magnetostatics with NGSolve

We first give the definition of a 3D geometry as drawn in the field-lines plot above. We specify the cylindrical coil by cutting out a smaller cylinder from a larger cylinder. The air domain is bounded by a rectangular box. In Netgen, one can specify geometric primitives such as infinite cylinders, half-spaces (called planes), and so on. One can construct more complicated solids by forming the union (or) , intersection (and) or complements (not) of simpler ones. One can specify boundary conditions with the `-bc=xxx` flag. The final sub-domains are called Top-Level-Objects (tlo) and have to been specified. The `-col=[red,green,blue]` flag gives the color for the visualization.

```
algebraic3d
solid coil = cylinder (0, 0, -1; 0, 0, 1; 0.4)
        and not cylinder  (0, 0, -1; 0, 0, 1; 0.2)
        and plane (0, 0, 0.4; 0, 0, 1)
        and plane (0, 0, -0.4; 0, 0, -1);

solid box = orthobrick (-2, -2, -2; 3, 2, 2) -bc=1;
solid air =  box and not coil -bc=3;
```

```
tlo coil -col=[0, 1, 0];
tlo air  -col=[0, 0, 1] -transparent;
```

The input-file for the solver is the tutorial example 'd7_coil.pde': The current source in the sub-domain of the coil is prescribed as function $(y, -x, 0)$ in angular direction. An $H(\mathrm{curl})$ finite element space of arbitrary order is defined by specifiing the `-hcurlho` flag. The flag `-nograds` specifies to skip all gradient basis functions. The keywords for the integrators are

`sourceedge jx jy jz`	$\int j \cdot v \, dx$
`curlcurledge nu`	$\int \nu \, \mathrm{curl}\, u \, \mathrm{curl}\, v \, dx$
`massedge sigma`	$\int \sigma u \cdot v \, dx$

The numproc `drawflux` inserts a field into the visualization dialog-box. It applies the differential operator of the specified bilinear-form to the specified grid-function. The `-applyd` flag specifies whether the field is multiplied with the coefficient function, or not. Recall that the B-field is curl A, and the H-field is μ^{-1} curl A.

```
geometry = ngsolve/pde_tutorial/coil.geo
mesh = ngsolve/pde_tutorial/coil.vol

define constant geometryorder = 4
define constant secondorder = 0

define coefficient nu
1, 1,

define coefficient sigma
1e-6, 1e-6,

define coefficient cs1
( y ), 0,
define coefficient cs2
( -x ), 0,
define coefficient cs3
0, 0, 0

define fespace v -hcurlho -order=4 -nograds

define gridfunction u -fespace=v -novisual

define linearform f -fespace=v
sourceedge cs1 cs2 cs3 -definedon=1
```

```
define bilinearform a -fespace=v -symmetric
curlcurledge nu
massedge sigma -order=2


define bilinearform acurl -fespace=v -symmetric -nonassemble
curlcurledge nu


define preconditioner c -type=multigrid -bilinearform=a -cylce=1 -smoother=block -coarse

numproc bvp np1 -bilinearform=a -linearform=f -gridfunction=u -preconditioner=c -maxstep

numproc drawflux np5 -bilinearform=acurl -solution=u  -label=B-field
numproc drawflux np6 -bilinearform=acurl -solution=u  -label=H-field  -applyd
```

2 Mathematical Objects and their implementation

In this chapter, we discuss the building blocks of the finite element method, and how they are implemented in the object-oriented C++ code NGSolve.

2.1 Finite Elements

One has to build a basis for the finite element function space

$$V_h = \{v \in C^0 : v|_T \in P^k \text{ for all elements } T \text{ in the mesh}\},$$

where C^0 are the continuous functions. On triangles (and tetrahedra), the space P^k is the space of polynomials up to the total order k. On quadrilaterals (and 3D hexahedra), the space $P^k = P^{k,k}$ contains all polynomials up to order k in each variable.

The global basis is constructed by defining an local basis on each element such that the local basis functions match at the element boundaries. To define the element basis, the element T is considered as the mapping of one reference element, i.e.,

$$T = F_T(T^R).$$

This reduces the task to define a basis $\{\varphi_1, \ldots, \varphi_{N_T}\}$ on the reference element. The basis functions on the mapped element are

$$\varphi_{T,i}(F_T(x)) := \varphi_i(x) \qquad \forall\, x \in T^R$$

The basis functions on the reference element are called shape functions.

2.1.1 One dimensional finite elements

The lowest order finite element space consists of continuous and piecewise affine-linear functions. A basis for the 1D reference element $T^R = (-1, 1)$ is

$$\varphi_1 = \frac{1+x}{2}, \quad \varphi_2 = \frac{1-x}{2}.$$

These functions are 1 in one vertex, and vanish in the other one. The global basis function associated with the vertex V is $\varphi_{T_l,2}$ on the left element, and $\varphi_{T_r,1}$ for the right element.

To build C^0-continuous elements of higher order, one adds more basis functions which vanish at the boundary $\{-1, +1\}$ to ensure continuity. These functions are called bubble functions. A simple basis is the monomial one

$$\varphi_{i+3} = x^i(1-x^2) \qquad \forall\, i = 0, \ldots p-2.$$

Later, we will have to evaluate matrices like $A_{ij} := \int_{-1}^{+1} \varphi_i \varphi_j \, dx$. When using this basis, the matrix is very ill conditioned (comparable to the Hilbert matrix). Thus, one usually chooses orthogonal polynomials as basis functions.

The k^{th} Legendre polynomial P_k is a polynomial of order k which is $L_2(-1,1)$-orthogonal to all polynomials of order $l < k$, i.e.,

$$\int_{-1}^{1} P_k(x) P_l(x)\, dx = 0 \qquad \forall\, l \neq k.$$

They are normalized such that $P_k(1) = 1$. Legendre polynomials can be evaluated efficiently by the three-term reccurency

$$\begin{aligned} P_0(x) &= 1, \\ P_1(x) &= x, \\ P_k(x) &= \frac{2j-1}{j} x P_{k-1}(x) - \frac{j-1}{j} P_{k-2}(x) \qquad k \geq 2. \end{aligned}$$

Legendre polynomials do not vanish at the element boundaries. One possibility to ensure this is to multiply with the quadratic bubble, i.e., to take $(1-x^2)P_k(x)$. A different one (and the most popular one) is to introduce integrated Legendre polynomials

$$L_k(x) := \int_{-1}^{x} P_{k-1}(s)\, ds.$$

For $k \geq 2$, these polynomials vanish in $\{-1, 1\}$. The left end is clear, for the right end there holds

$$L_k(1) = \int_{-1}^{1} P_{k-1}\, ds = \int_{-1}^{1} P_0 P_{k-1}\, dx = 0 \qquad \forall\, k \geq 2.$$

The idea behind the integrated Legendre polynomials is that they are orthogonal with respect to the inner product $(u', v')_{L_2(-1,1)}$.

Legendre polynomials belong to the more general family of Jacobi polynomials $P_k^{(\alpha,\beta)}$ which are orthogonal polynomials with respect to the weighted inner product

$$(u, v) := \int_{-1}^{1} (1-x)^\alpha (1+x)^\beta uv\, dx.$$

There is a three term reccurency for the Jacobi polynomials as well. The bubble functions

$$\varphi_k = (1-x^2) P_k^{(2,2)}$$

are L_2-orthogonal.

2.1.2 Quadrilateral finite elements

We take the reference square $(-1,1)^2$. The lowest order basis functions are the bilinear ones

$$\varphi_1 = \frac{(1+x)(1+y)}{4} \quad \varphi_2 = \frac{(1-x)(1+y)}{4} \quad \varphi_3 = \frac{(1-x)(1-y)}{4} \quad \varphi_4 = \frac{(1+x)(1-y)}{4}$$

These are functions which are 1 in one vertex, and vanish on all edges not containing this vertex, in particular, in all other vertices. These basis functions are associated with the vertices of the mesh. The restriction to the edges are linear functions. They are continuous since they coincide in the vertices.

Next, we add functions to obtain polynomials of order p on the edges. These additional basis functions must have support only on the elements containing the edge. This is obtained by choosing edge bubble-functions vanishing in the vertices. For example, the edge bubble functions for the edge $(-1,-1)$-$(1,-1)$ on the reference element are defined as

$$\varphi_{E_1,i} := L_{i+2}(x)\frac{y+1}{2} \qquad i = 0,\dots,p-2$$

These functions vanish on all other edges.

To obtain the full space $P^{k,k}$, one has to add the element bubble functions

$$L_i(x)L_j(y) \qquad i,j = 2,\dots,p.$$

These functions vanish on the boundary ∂T, and thus are always continuous to the neighbor elements.

There appears one difficulty with the continuity of the edge basis functions: There are even functions and odd functions on the edge. For the odd functions, the orientation of the edge counts. When mapping the reference elements onto the domain, the orientation of the edges do not necessarily match, and thus, the functions would not be continuous. One possibility to resolve the conflict is to transform the basis functions. The odd functions must change sign, if the edge of the reference element is oriented opposite to the global edge. A second possibility is to parameterize the reference element: For each specific element, take the global orientation of the edge onto the reference element. This is a transformation of the arguments x or y. A simple possibility to orient edges is to define the direction from the smaller vertex to the larger one. On the reference element, one has to know the global vertex number to handle the orientation. This second approach is simpler for 3D elements (in particular tetrahedral ones) and thus taken in NGSolve.

2.1.3 Triangular finite elements

The construction of a basis on triangles follows the same lines. It is useful to work with barycentric coordinates λ_1, λ_2, and λ_3. The vertex basis functions are exactly the barycentric coordinates.

The edge-based basis functions on the edge between vertex i and vertex j are defined as

$$\varphi_{E,k} = L_{k+2}\Big(\frac{\lambda_i - \lambda_j}{\lambda_i + \lambda_j}\Big)(\lambda_i + \lambda_j)^{k+2} \qquad k = 0,\dots,p-2.$$

There holds

1. This is a polynomial of order $k+2$. The first factor is rational of order $k+2$ in the denominator, and this is compensated by the second factor. The implementation of these functions is possible by a division free three-term reccurency.

2. On the edge E_{ij} there holds $\lambda_i+\lambda_j = 1$, and thus, the function simplifies to $L_{k+2}(\lambda_i - \lambda_j)$.

3. On the edge E_{jk} there holds $\lambda_i = 0$, and the function is $L_{k+2}(-1)\lambda_j^{k+2} = 0$. Similar for the edge E_{ik}.

Again, the element bubble functions are defined by a tensor product construction. For this, let

$$\begin{aligned} u_k &= L_{k+2}\Big(\frac{\lambda_1-\lambda_2}{\lambda_1+\lambda_2}\Big)(\lambda_1+\lambda_2)^{k+2} \qquad k = 0,\ldots,p-2, \\ v_l &= \lambda_3 P_l(2\lambda_3-1) \qquad l = 0,\ldots,p-1 \end{aligned}$$

Then, the element bubble functions are

$$\varphi_{k,l} = u_k v_l \qquad \forall\, k \geq 0, l \geq 0, l+k \leq p-3.$$

The first factor vanishes for the edges $\lambda_1 = 0$ and $\lambda_2 = 0$, and the second factor vanishes for the edge $\lambda_3 = 0$.

2.1.4 Tangential continuous finite elements

To approximate Maxwell equations (in $H(\mathrm{curl})$), we need vector valued finite elements whose tangential components are continuous over element boundaries. We construct such elements for triangles.

The simplest type of basis functions are the high order edge-based basis functions. Take the gradients of the H^1 edge-based basis functions of one order higher:

$$\Phi_{E,i} = \nabla\varphi_{E,i+1} \qquad i = 1,\ldots,p$$

Since the global H^1 basis functions are continuous over element boundaries, the tangential component of their derivatives is continuous, as well. Above, we have defined p basis functions up to order p. There is missing one more function. Since for edge bubble functions there holds

$$\int_E \Phi_{E,i}\cdot\tau\,ds = \int_E \nabla\varphi_{E,i+1}\cdot\tau\,ds = \varphi_{E,i+1}(V_{E_1}) - \varphi_{E,i+1}(V_{E_2}) = 0,$$

all these functions are $(\cdot\tau,\cdot\tau)_{L_2(E)}$-orthogonal to the constant. One has to add the lowest order edge-element basis function as defined in Section 1.3:

$$\Phi_{E,0} = \nabla\varphi_{E_1}\varphi_{E_2} - \varphi_{E_1}\nabla\varphi_{E_2}$$

The tangential boundary values of the element-based basis functions must vanish. Recall the construction $\varphi_{T,ij} = u_i v_j$ for the H^1-case. The first factor vanishes on the two edges $\lambda_1 = 0$ and $\lambda_2 = 0$, while the second factor vanishes for $\lambda_3 = 0$. The functions

$$\Phi^1_{T,kl} := (\nabla u_k) v_l$$

have vanishing tangential traces on all edges, since the tangential derivative of a bubble function vanishes. The same holds for

$$\Phi^2_{T,kl} := u_k (\nabla v_l)$$

A third class of functions with vanishing tangential traces are

$$\Phi^2_{T,kl} := \Phi_{E,0} v_l,$$

where $\Phi_{E,0}$ is the lowest order edge-basis function for the edge $\lambda_3 = 0$. One can show that these functions form a basis for $\{v \in [P^p]^2 : v \cdot \tau = 0 \text{ on } \partial T\}$. Instead of taking the first two types a above, one may take the sum and the difference of them. This has the advantage to include gradient basis functions explicitely (since $\nabla(u_k v_l) = (\nabla u_k) v_l + u_k \nabla v_l$).

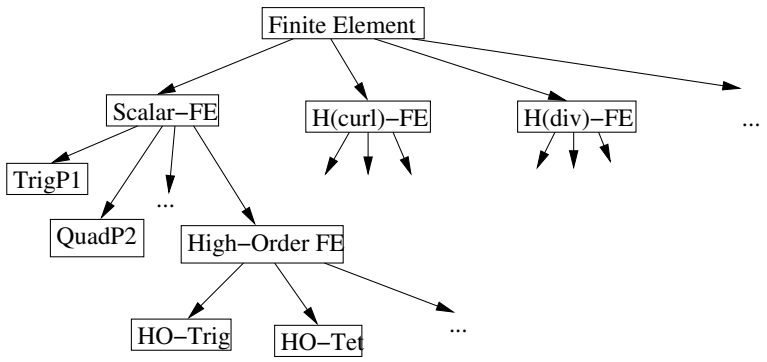
We were sloppy in defining the gradient. We need tangential continuity on the mapped element, so the basis functions must be gradients on the mapped elements. But, on the other hand, we want to define the $H(\mathrm{curl})$ basis functions on the reference element. The remedy is to define the transformation for $H(\mathrm{curl})$-basis functions compatible with gradients by

$$u(F(x)) := (F')^{-T} u^R(x) \qquad \forall\, x \in T^R.$$

If u^R is the gradient ∇w^R on the reference element, then $u = \nabla w$ on the mapped element, where $w(F(x)) = w^R(x)$. This transformation is called covariant transformation.

2.1.5 A class hierarchy for finite elements

In NGSolve, a hierarchy is designed to extract common properties of families of finite elements:



All these types correspond to C++ classes.

The base class must be general enough to include common properties of all type of elements (currently in the mind of the author). These include the type of element, space dimension, number of shape functions, and polynomial order. The C++ realization is:

```
class FiniteElement
{
protected:
  int dimspace;         // space dimension (1, 2, or 3)
  ELEMENT_TYPE eltype;  // element geometry (trig, quad, ...)
  int ndof;             // number of degrees of freedom
  int order;            // polynomial order

public:
  FiniteElement (int adimspace, ELEMENT_TYPE aeltype, int andof, int aorder)
    : dimspace(adimspace), eltype(aeltype), ndof(andof), order(aorder) { ; }
  virtual ~FiniteElement () { ; }

  int SpatialDim () const { return dimspace; }
  int GetNDof () const { return ndof; }
  int Order () const { return order; }
  ELEMENT_TYPE ElementType() const { return eltype; }
};
```

The properties are chosen to be stored as member variables (instead of, e.g., provided by virtual function calls) for faster access. A variable of each specific finite element type is just defined once for the reference element, so additional memory cost is not an issue. The base class has no methods for shape function evaluation since it is not clear whether the shape functions are scalars or vectors.

Finite elements for scalar H^1-problems have in common that they have scalar shape functions, and the gradient is well defined. For historical reasons, H^1 finite elements are called `NodalFiniteElement`. These class inherits the properties of the base `FiniteElement`, and adds methods for shape function evaluation:

```
class NodalFiniteElement : public FiniteElement
{
  ...
  virtual void CalcShape (const IntegrationPoint & ip,
                          Vector<> & shape) const = 0;
  virtual void CalcDShape (const IntegrationPoint & ip,
                           Matrix<> & dshape) const;
};
```

A NodalFiniteElement offers just the interface to compute shape functions, but, it does not know about the specific element, so it cannot compute shape functions. This is provided by the mechanism of virtual functions. The function `CalcShape` computes the vector of all shape functions in a given point `ip` on the reference element. It is a pure virtual function (specified by the syntax `= 0` at the end of the line) which means that every specific finite element must overload the `CalcShape` function. The function `CalcDShape` computes all partial derivatives of shape functions and stores them in the $ndof \times spacedim$-matrix dshape. There is a default implementation by numerical differentiation, but, the specific finite element class may overload this function by computing the derivatives analytically.

A specific finite element is a triangular element with shape functions in P^1. Indeed, also the CalcDShape method is overloaded:

```
class FE_Trig1 : public NodalFiniteElement
{
  FE_Trig1() : NodalFiniteElement (2, ET_TRIG, 3, 1) { ; }
  virtual ~FE_Trig1() { ; }
  virtual void CalcShape (const IntegrationPoint & ip,
                          Vector<> & shape) const
  {
    shape(0) = ip(0);
    shape(1) = ip(1);
    shape(2) = 1-ip(0)-ip(1);
  }
};
```

There were a plenty of elements implemented for fixed order up to 2 or 3.

The newer elements are high order elements of variable order. One can specify a polynomial order for each edge, (and each face for 3D,) and the interior of the element separately. The parametric reference element also contains the global vertex number to compute shape functions with the right orientation. The following class collects the additional properties for all scalar high order elements:

```
class H1HighOrderFiniteElement : public NodalFiniteElement
{
public:
  int vnums[8];   // global vertex number
  int order_inner;
  int order_face[6];
  int order_edge[12];
public:
  H1HighOrderFiniteElement (int spacedim, ELEMENT_TYPE aeltype);
```

```
  void SetVertexNumbers (Array<int> & vnums);
  void SetOrderInner (int oi);
  void SetOrderFace (const Array<int> & of);
  void SetOrderEdge (const Array<int> & oe);

  virtual void ComputeNDof () = 0;
};
```

Now, for each element geometry (trig, quad, tet, hex, ..), one high order finite element class is defined. It computes the number of shape functions and the shape functions for a specified order in each edge and the interior:

```
class H1HighOrderTrig : public H1HighOrderFiniteElement
{
  H1HighOrderTrig (int aorder);
  virtual void ComputeNDof();
  virtual void CalcShape (const IntegrationPoint & ip,
                          Vector<> & shape) const;
};
```

In contrast to the `NodalFiniteElement`, the `HCurlFiniteElement3D` computes a matrix of shape functions, and a matrix of the curl of the shape functions:

```
class HCurlFiniteElement3D : public FiniteElement
{
  ...
  virtual void CalcShape (const IntegrationPoint & ip,
                          Matrix<> & shape) const = 0;
  virtual void CalcCurlShape (const IntegrationPoint & ip,
                              Matrix<> & dshape) const;
};
```

2.1.6 The shape tester

The shape tester is a tool to visualize the basis-functions. It was written for debugging the shape functions. There pops up a dialog box to select the index of the basis function.

```
geometry = examples/cube.geo
mesh = examples/cube.vol

define fespace v -h1ho -order=3
define gridfunction u -fespace=v

numproc shapetester np1 -gridfunction=u
```

2.2 Integration of bilinear-forms and linear-forms

After choosing the basis $\{\varphi_1, \ldots, \varphi_N\}$ for the finite element space, the system matrix $A \in \mathbb{R}^{N \times N}$ and right hand side vector $f \in \mathbb{R}^N$ are defined by

$$A_{i,j} := A(\varphi_i, \varphi_j) \qquad \text{and} \qquad f_j := f(\varphi_j).$$

For now, we assume that $A(.,.)$ has the structure

$$A(u,v) = \int_\Omega B(v)^t D B(u)\, dx,$$

where $B(u)$ is some differential operator such as $B(u) = \nabla u$, $B(u) = u$, $B(u) = \operatorname{curl} u$, etc. etc. The matrix D is a coefficient matrix, e.g., $D = \lambda(x) I$. Similar, the linear-form has the structure

$$f(v) = \int_\Omega d^t B(v)\, dx,$$

where d is the coefficient vector. The case of boundary-integrals follows the same lines.

The convenient way to compute the global matrix is to split the integral over Ω into integrals over the elements T, and use that the restriction of the global basis function φ_i onto the element T is a shape function φ_α^T:

$$A_{i,j} = \int_\Omega B(\varphi_j)^t D B(\varphi_i)\, dx = \sum_{T \in \mathcal{T}} \int_T B(\varphi_\beta^T)^t D B(\varphi_\alpha^T)$$

One computes local *element matrices* $A^T \in \mathbb{R}^{N_T \times N_T}$ by

$$A^T_{\alpha,\beta} = \int_T B(\varphi_\beta^T)^t D B(\varphi_\alpha^T)\, dx,$$

and assembles them together to the global matrix

$$A = \sum_T (C^T)^t A^T C^T,$$

where $C_T \in \mathbb{R}^{N_T, N}$ is the *connectivity matrix* relating the restriction of the global basis function φ_i to the local ones via

$$\varphi_i|_T = \sum_{\alpha=1}^{N_T} C^T_{\alpha,i} \varphi^T_\alpha.$$

Usually, the C^T consists mainly of 0s, and has N_T entries 1. E.g., the orientation of edges can be included into the connectivity matrices by -1 entries.

For many special cases, the integrals can be computed explicitely. But for more general cases (e.g., with general coefficients), they must be computed by numerical integration.

For this, let $IR^k = \{(x_i, \omega_i)\}$ be an integration rule or order k for the reference element $\widehat{T}$, i.e.,

$$\int_{\widehat{T}} f(x)\,dx \approx \sum_{(x_i,\omega_i)\in IR^k} \omega_i f(x_i),$$

and the formula is exact for polynomials up to order k. The best integration rules for the 1D reference element are Gauss-rules, which can be generated for an arbitrary order k. On other elements (quads, trigs, tets, hexes, ...) integration rules are formed by tensor product construction.

On a general element T, which is obtained by the transformation F_T, i.e., $T = F_T(\widehat{T})$, the transformed integration rule is

$$\int_T f(x)\,dx \approx \sum_{(x_i,\omega_i)\in IR^k} \omega_i f(F_T(x_i)) \det(F_T'(x_i)).$$

Computing the element matrices by the integration rule gives

$$A^T_{\alpha,\beta} \approx \sum_{(x_i,\omega_i)} \omega_i B(\varphi^T_\beta)(F_T(x_i))^t D B(\varphi^T_\alpha)(F_T(x_i))\ \det(F_T'(x_i)).$$

By combining the vectors $B(\varphi_i^T)(F_T(x_i))$ to a matrix (called B-matrix), the whole matrix A^T can be written as

$$\begin{aligned} A^T &\approx \sum_{(x_i,\omega_i)} \omega_i \begin{pmatrix} B(\varphi_1^T)^t \\ B(\varphi_2^T)^t \\ \vdots \\ B(\varphi_{N_T}^T)^t \end{pmatrix} D\Big(B(\varphi_1^T), \ldots, B(\varphi_{N_T}^T)\Big) \det(F_T') \\ &= \sum_{(x_i,\omega_i)} \omega_i B^t D B\ \det(F_T') \end{aligned}$$

The advantage of the matrix form is that for the implementation optimized matrix-matrix operations can be used instead of hand-written loops.

2.2.1 Examples of integrator

In the case of the bilinear-form

$$\int \rho(x) u v\,dx,$$

i.e., $B(u) = u$ and $D = \rho(x)$, the B-matrix in $\mathbb{R}^{1\times N_T}$ is

$$B_{j,1} = (\varphi_j^T(x_i))_{i=1,\ldots N_T}.$$

The shape functions φ^T on the mapped element $T = F_T(\widehat{T})$ are defined by means of the shape functions $\hat{\varphi}$ on the reference element

$$\varphi^T(F_T(\hat{x})) := \hat{\varphi}(\hat{x}) \qquad \forall\, \hat{x} \in \widehat{T}.$$

The bilinear-form of a scalar 2^{nd} order problem with general coefficient matrix $a \in \mathbb{R}^{d\times d}$ is

$$\int_\Omega (a\nabla u)\cdot\nabla v\, dx.$$

In this case, the B-matrix is of dimension $d \times N_T$ and has the components

$$B_{k,j} = \frac{\partial \varphi_j^T}{\partial x_k}$$

To express these derivatives by derivatives on the reference element, the chain rule is involved (where $\frac{d}{dx}$ gives a row vector):

$$\frac{d\varphi^T}{dx} = \frac{d}{dx}\hat{\varphi}(F_T^{-1}(x)) = \frac{d\hat{\varphi}}{dx}(F_T^{-1}(x))\frac{dF_T^{-1}(x)}{dx} = \frac{d\hat{\varphi}}{dx}(F_T^{-1}(x))(F')^{-1}(F_T^{-1}(x)).$$

Rewriting for column-vectors $\nabla\varphi$ gives

$$\nabla\varphi^T(F_T(\hat{x})) = (F')^{-T}(\hat{x})(\nabla\hat{\varphi})(\hat{x}).$$

By first setting up the $\widehat{B}$-matrix for the reference element, i.e.,

$$\widehat{B}_{k,j} = \frac{\partial \varphi_j}{\partial x_k},$$

the B-matrix is computed by the matrix-matrix operation

$$B = (F')^{-T}\widehat{B}. \tag{15}$$

The bilinear-form for *linear elasticity* (e.g., for 2D) involves the strain operator $\varepsilon(u) = \frac{1}{2}\{(\nabla u) + (\nabla u)^T\}$. We rewrite this symmetric strain-matrix as a strain-vector in the form

$$B(u) = \begin{pmatrix} \varepsilon_{11}(u) \\ \varepsilon_{22}(u) \\ 2\varepsilon_{12}(u) \end{pmatrix} = \begin{pmatrix} \frac{\partial u_1}{\partial x_1} \\ \frac{\partial u_2}{\partial x_2} \\ \frac{\partial u_1}{\partial x_2} + \frac{\partial u_2}{\partial x_1} \end{pmatrix}.$$

A basis for the vector-field (u_x, u_y) is built by taking 2 copies of the scalar basis and arranging it as

$$\left\{ \begin{pmatrix} \varphi_1^T \\ 0 \end{pmatrix}, \begin{pmatrix} 0 \\ \varphi_1^T \end{pmatrix}, \begin{pmatrix} \varphi_2^T \\ 0 \end{pmatrix}, \begin{pmatrix} 0 \\ \varphi_2^T \end{pmatrix}, \ldots, \begin{pmatrix} \varphi_{N_T}^T \\ 0 \end{pmatrix}, \begin{pmatrix} 0 \\ \varphi_{N_T}^T \end{pmatrix} \right\}.$$

Thus, the B-matrix in $\mathbb{R}^{3\times 2N_T}$ takes the form

$$B = \begin{pmatrix} \frac{\partial \varphi_1^T}{\partial x_1} & 0 & \frac{\partial \varphi_2^T}{\partial x_1} & 0 & \cdots \\ 0 & \frac{\partial \varphi_1^T}{\partial x_2} & 0 & \frac{\partial \varphi_2^T}{\partial x_2} & \\ \frac{\partial \varphi_1^T}{\partial x_2} & \frac{\partial \varphi_1^T}{\partial x_1} & \frac{\partial \varphi_2^T}{\partial x_2} & \frac{\partial \varphi_2^T}{\partial x_1} & \end{pmatrix}$$

2.2.2 A template family of integrators

The element-matrix (and element-vector) integrators are implemented as classes. The base class provides the interface to a function for computing the element matrix. It cannot be implemented for the base-class, so it is a pure virtual function:

```
class BilinearFormIntegrator
{
public:
  virtual
  void AssembleElementMatrix (const FiniteElement & fel,
                              const ElementTransformation & eltrans,
                              Matrix<double> & elmat) = 0;
  ...
};
```

One has to provide a `FiniteElement`, which can evaluate the shape functions on the reference element. The `ElementTransformation` knows about the mapping F_T, and can compute metric quantities such as F_T'. The result is returned in the matrix `elmat`.

The common properties for integrators of the form $\int B(v)^t D B(u) dx$ are combined into the family of classes `T_BDBIntegrator`. Each member of the family has its own differential operator $B(.)$, and coefficient-matrix D. Such families of classes can be realized by class-templates:

```
template <class DIFFOP, class DMATOP>
class T_BDBIntegrator : public BilinearFormIntegrator
{
protected:
  DMATOP dmatop;
public:

  virtual void
  AssembleElementMatrix (const FiniteElement & fel,
                         const ElementTransformation & eltrans,
                         Matrix<double> & elmat) const
  {
     int ndof = fel.GetNDof();
     elmat.SetSize (DIFFOP::DIM * ndof);
     elmat = 0;

     Matrix<double> bmat (DIFFOP::DIM_DMAT, DIFFOP::DIM * ndof);
     Matrix<double> dbmat (DIFFOP::DIM_DMAT, DIFFOP::DIM * ndof);
     Mat<DIFFOP::DIM_DMAT, DIFFOP::DIM_DMAT, double> dmat;
```

```
      const IntegrationRule & ir = GetIntegrationRule (fel);

      for (int i = 0 ; i < ir.GetNIP(); i++)
      {
        SpecificIntegrationPoint  ip(ir[i], eltrans);

        DIFFOP::GenerateMatrix (fel, ip, bmat);
        dmatop.GenerateMatrix (fel, ip, dmat);

        double fac = ip.GetJacobiDet() * ip.Weight();

        dbmat = fac * (dmat * bmat);
        elmat += Trans (bmat) * dbmat;
      }
    }
}
```

The dimension of the element matrices depend on the number of shape functions provided by finite element, and of the differential operator. The diff-op provides the dimension of the D-matrix (called `DIFFOP::DIM_DMAT`), as well as the number of copies of the finite element, e.g., 2 for linear elasticity in 2D, (called `DIFFOP::DIM`). While the size of the B-matrix depends on the finite element, the size of the D-matrix is fixed at compile-time. For this, once the matrix-class `Matrix` of dynamic size, and once, the matrix class with fixed size `Mat<H,W>` is used. The integration rule depends on the geometry of the element, and the polynomial order of the shape functions. The `SpecificIntegrationPoint` stores the Jacobi matrix, which is computed by the `ElementTransformation eltrans`.

The `DIFFOP` class has the function `GenerateMatrix` to compute the B-matrix. It does not need additional data, so a static function of the class is called. Similar, the `DMATOP` class computes the D-matrix. But now, data (such as the value of coefficients) are involved, and a function for the variable `dmatop` is called. The linear algebra expressions compute the element matrix as derived above.

Some more secrets:

- Memory management to avoid many allocate/delete-operations
- combining several integration points for longer inner loops
- B-matrix of fixed height at compile-time

A differential operator has to provide some values such as the dimension of the D-matrix, and has to compute the B-matrix. A gradient differential operator for D dimensions is `DiffOpGradient<D>`. The B-matrix is computed according to equation (15):

```
/// Gradient operator of dimension D
template <int D> class DiffOpGradient
```

```
{
public:
  enum { DIM = 1 };
  enum { DIM_SPACE = D };
  enum { DIM_ELEMENT = D };
  enum { DIM_DMAT = D };

  static void GenerateMatrix (const FiniteElement & fel,
                              const SpecificIntegrationPoint & ip,
                              Matrix<double> & mat)
  {
    mat = Trans (ip.GetJacobianInverse ()) * Trans (fel.GetDShape(ip));
  }
};
```

To provide a simpler use of the integrators, we have defined classes combining differential operators and coefficient matrices, such as a D-dimensional `LaplaceIntegrator<D>`:

```
template <int D>
class LaplaceIntegrator : public T_BDBIntegrator<DiffOpGradient<D>, DiagDMat<D> >
{
public:
  LaplaceIntegrator (CoefficientFunction * coeff)
    : T_BDBIntegrator<DiffOpGradient<D>, DiagDMat<D> > (DiagDMat<D> (coeff))
  {
    ;
  }
};
```

A tutorial showing how to use the finite elements and integrators is `netgen/ngsolve/tutorial/demo_fem.cpp`

2.3 Linear algebra concepts

Vectors and matrices are central data types for any mathematical simulation software. There are different competing design choices. E.g., if one needs many operations with small matrices, or little operations with large ones. The first type is the one I have in mind for the (dense) element matrix computations, while the second one is the type for the assembled global (sparse) matrices. A related choice is whether matrix-matrix operations are useful, or if just matrix-vector multiplications are efficiently possible.

2.3.1 Matrix and Vector data types

The cheapest useful vector data type stores the size of the vector, and has a pointer to the data. In NGSolve, such a vector is called `FlatVector`. The prefix `Flat` is always used for classes not taking care about memory-management. Such a `FlatVector` is used, when one wants to use ones own allocation, or, if the data already exists in memory, and one wants to put a Vector - data-structure over this array. The `FlatVector` is a template-class, where the template argument specifies the data type for the elements of the vector. Useful types are, e.g., `double` or `std::complex<double>` from the C++ standard library.

In the constructor, the vector-size and the pointer to the memory are set. One can access the size, and one can access the vector's elements with the bracket-operator, for example `u(i) = v(i) + w(i)`:

```
template <typename T = double>
class FlatVector
{
protected:
  int size;
  T * data;
public:
  FlatVector (int as, T * adata) : size(as), data(adata) { ; }

  int Size () const { return size; }

  T & operator() (int i) { return data[i]; }
  const T & operator() (int i) const { return data[i]; }
};
```

The `Vector` class extends the `FlatVector` by memory management. In the constructor, the vector size is given. The `Vector` allocates the required memory, and initializes the base class with it. The `Vector` has a destructor cleaning up the memory:

```
template <typename T = double>
class Vector : public FlatVector<T>
{
```

```
public:
 Vector (int as) : FlatVector<T> (as, new T[as]) { ; }
  ~Vector() { delete [] data; }
};
```

Possible uses of `Vector` and `FlatVector` are:

```
  Vector<double> u(10);
  FlatVector<double> subvec(&u(6), 4);
```

Often, the size of the vector is known at compile-time. For this case, there is the `Vec` template class. It can use static memory, instead of dynamic:

```
template <int SIZE, typename T = double>
class Vec
{
protected:
  T data[SIZE];
public:
  Vec () { ; }

  int Size () const { return SIZE; }

  T & operator() (int i) { return data[i]; }
  const T & operator() (int i) const { return data[i]; }
};
```

Possible uses are for 3D - point coordinates. One can also build `Vector`s, where the elements are `Vec`s:

```
  Vector<Vec<3,double> > u(10);
  u(10)(0) = 5;
```

In a similar way, there exist the matrix types `FlatMatrix`, `Matrix`, and `Mat`. The access operators are of the form `m(i,j)`.

2.3.2 Vector operations

Vectors should provide the vector space operations 'sum of vectors' and 'multiplication with a scalar'. A nice way to code vector-operations is like

```
  Vector<double> u(10), v(10), w(10);
  double alpha;
  u = alpha * v + w;
```

The C++ beginners implementation is to implement the '+' operator, the '*' operator, and the assignment operator '=' as follows:

```
template<typename T>
Vector<T> operator+ (const FlatVector<T> & a, const FlatVector<T> & b)
{
  Vector<T> temp(a.Size());
  for (int i = 0; i < a.Size(); i++) temp(i) = a(i) + b(i);
  return temp;
}

template<typename T>
Vector<T> operator* (double a, const FlatVector<T> & b)
{
  Vector<T> temp(a.Size());
  for (int i = 0; i < b.Size(); i++) temp(i) = a * b(i);
  return temp;
}

class FlatVector<T>
{
  ...
  FlatVector<T> & operator= (const FlatVector<T> & v)
  {
     for (int i = 0; i < size; i++) data[i] = v(i);
     return *this;
  }
};
```

This vector implementation allows the nice notation, but at the costs of low performance: Temporary objects must be allocated inside the operator functions, and, additionally for the return values.

To avoid such temporary objects, one can provide the following assignment methods:

```
class FlatVector<T>
{
  ...

  FlatVector<double> & void Set (double alpha, FlatVector<T> & v)
  {
     for (int i = 0; i < size; i++) data[i] = alpha * v(i);
     return *this;
  }

  FlatVector<double> & Add (double alpha, FlatVector<T> & v)
  {
    for (int i = 0; i < size; i++) data[i] += alpha * v(i);
```

```
    return *this;
  }
```

The use of these methods look like:

```
u . Set (alpha, v) . Add (1, w);
```

... not that nice, but more efficient.

The remedy for this performance-readability conflict are *expression templates*. The above operator notation gets inefficient, since the memory to store the result is not available when just knowing the vector arguments. The idea is to return a symbolic object representing the sum of two vectors. This representation knows how to evaluate the elements of the sum:

```
template <typename VA, typename VB>
class SumVector
{
  const VA & veca;
  const VB & vecb;
public:
  SumVector (const VA & a, const VB & b) : veca(a), vecb(b) { ; }
  VA::TELEM operator() (int i) { return veca(i)+vecb(i); }
};

template <typename VA, typename VB>
SumVector<VA, VB> operator+ (const VA & a, const VB & b)
{ return SumVector<VA, VB> (a, b); }
```

The computation can only happen at a stage, when the memory for the result is known. This is the case in the assignment operator:

```
class FlatVector
{
  ...
  template <typename VA, typename VB>
  void operator= (const SumVector<VA, VB> & sum)
  {
    for (int i = 0; i < size; i++)  data[i] = sum(i);
  }
}
```

Now, a construct like

```
u = v + w;
```

is possible. The '+' operator returns the object `SumVector<Vector<double>, Vector<double> >`. The '*' operator is defined similar:

```
template <typename VB>
class ScaleVector
{
  double scal;
  const VB & vecb;
public:
  ScaleVector (double a, const VB & b) : scal(a), vecb(b) { ; }
  VB::TELEM operator() (int i) { return scal*vecb(i); }
};

template <typename VB>
ScaleVector<VB> operator* (double, const VB & b)
{ return ScaleVector<VB> (a, b); }
```

It is also possible to combine expressions:

```
u = alpha * v + w;
```

This results in a type `SumVector<ScaleVector<Vector<double> >, Vector<double> >`.

The above concept has one problem: The operators (e.g., '+') have un-specialized template arguments, which define the operator for every type. This may conflict with other '+' operators (e.g., for connecting strings). One has to introduce a joint base class (called, e.g., `VecExpr`) for all vector classes (including SumVector etc.), and defines the operators for members of the `VecExpr` family, only. To find back from the base-class to the specific vector class, the so called *Barton and Nackman*-trick is applied: The base class is a template family, and the derived class instantiates the base-class template argument with itself:

```
template <typename T>
class VecExpr { };

template <typename T>
class FlatVector<T> : public VecExpr<FlatVector<T> >
{ ... };

template <typename VA, typename VB>
class SumVector : public VecExpr<SumVector<VA, VB> >
{ ... };
```

Now, the '+' operator can be defined for members of the `VecExpr` family, only:

```
  template <typename VA, typename VB>
  SumVector<VA, VB> operator+ (const VecExpr<VA> & a, const VecExpr<VB> & b)
  {
    return SumVector<VA,VB> (static_cast<VA> (a), static_cast<VB> & b);
  }
```

What happens for `u + v`, where the vectors are of type `FlatVector<double>` ? The elements `u` and `v` are derived from `VecExpr<FlatVector<double> >`, thus, the above '+' operator can be applied. The template parameters `VA` and `VB` are initialized with `FlatVector<double>`. Inside the function, the elements `a` and `b`, which are known to be of the base type `VecExpr<FlatVector<double> >`, are up-casted to the derived type `FlatVector<double>`, what is indeed a valid cast. The result will be of the type `SumVector<FlatVector<double>, FlatVector<double> >`.

The assignment operator also takes profit of the `VecExpr` family:

```
template <typename T> class FlatVector : public VecExpr<..>
{
  ...
  template <typename TV>
  FlatVector & operator= (const VecExpr<TV> & v)
  {
    for (int i = 0; i < size; i++) data[i] = static_cast<TV>(v) (i);
    return *this;
  }
}
```

This programming style is called *expression templates*. In NGSolve, these expression templates are implemented in the basic linear algebra for vectors and dense matrices. Vectors are considered to be matrices of width 1. Matrix expressions include matrix-matrix products, sums, differences, negative matrices, and transpose matrices.

Expression templates are a challenge for compilers. Newer compilers (e.g. gcc3.x, Visual.net) are able to generate code comparable to hand-written loops for the matrix-vector operations. It is important to declare all the involved functions as inline to combine everything into one block.

An example file showing the use of the basic linear algebra is *ngsolve/tutorial/demo_bla.cpp*.

2.3.3 Linalg library based on Matrix-Vector multiplication

The above concept applies well to dense matrices, but cannot be used this way for all matrix operations providing just a matrix-vector multiplication. Examples for such matrices are sparse matrices, or linear operators defined by iterative methods like Gauss-Seidel iteration.

Here, I am thinking about matrices of large dimension, where one does not worry about a few virtual function calls.

In this case, we have a base class `BaseMatrix` providing a matrix-vector multiplication:

```
class BaseMatrix
{
  virtual void MultAdd (double s, BaseVector & x, BaseVector & y) = 0;
     // y += s * Mat * x
};
```

The specific matrices are derived from `BaseMatrix` and overload the `MultAdd` method. For virtual functions, template-parameterized arguments are not allowed. Thus, one needs also a `BaseVector` class. For the large matrices, I decided for a second family of matrices and vectors independent of the `Vector` from the dense library. Vectors derived from `BaseVector` were called `VVector` with `V` like in virtual. This vector family provides the `Set` and `Add` functions:

```
class BaseVector
{
  virtual BaseVector & Set (double s, BaseVector & v) = 0;   // *this = s * v
  virtual BaseVector & Add (double s, BaseVector & v) = 0;   // *this += s * v
};
```

Also for the matrix-vector library, expression templates are defined to allow the 'nice' notation. But now, the evaluation does not access vector/matrix elements, but calls the virtual functions `Set`, `Add`, or `MultAdd`.