An Introduction to Netgen and NGSolve
Session 2
Programming with NGSolve

Christoph Lehrenfeld

Center for Computational Engineering Sciences (CCES)
RWTH Aachen University

University of British Columbia, November 12, 2009

content

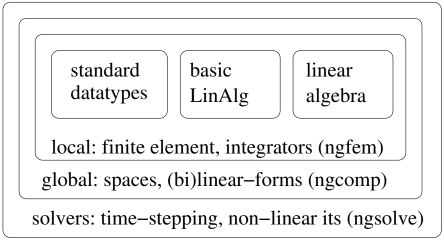
Ingredients to solve FEM-problem

First of all, we want to look at the basic ingredients that are necessary to solve linear elliptic PDE. Our guiding example will be Poissons equation.
For our task we need:

- a Finite Element defining shape functions on a reference domain
- a Finite Element Space handling the connection between the Finite Elements and the global mesh
- Bi- and Linearformintegrators computing elementwise matrices and vectors
- An Algorithm assembling elementwise contributions to a global matrix and/or large vectors
- Linear Algebra Routines solving the linear equation system coming up

Nonlinear, unsteady problems of course need additional things.

The layers of NGSolve



standard datatypes
basic LinAlg
linear algebra
local: finite element, integrators (ngfem)
global: spaces, (bi)linear–forms (ngcomp)
solvers: time–stepping, non–linear its (ngsolve)

- basiclinalg : `namespace ngbla`, `matrix`, `vector`, `bandmatrix`, `...`
- comp : `FESpaces` (`H1HOFESpace`, `HCurlHOFESpace`,...), `Linear-` and `BilinearForms`, `GridFunction`, `MeshAccess`, `Preconditioners`, ...
- fem : `Integrators`, `FiniteElements`, ...
- linalg : `VVectors`, direct solvers, iterative solvers, ...
- multigrid : Smoother, Prolongation, ...
- ngstd : `Array`, `LocalHeap`, `Table`, `AutoDiff`, ...
- parallel : MPI-Implementation of some NGSolve-things
- solve : PDE-Parser, `bvp`, other `numprocs`, ...

NGSolve's basic classes

- `Array<T>` : An array with memory allocation/deallocation
- `FlatArray<T>` : An array without memory handling, initialize with size and pointer
- `DynamicTable<T>` : Table with variable size
- `Table<T>` : A compact table which requires a priori knowledge of entrysizes

For local operation and temporary variable NGSolve uses an efficient memory management called `LocalHeap`:

- `LocalHeap lh(1000);` : Initialize heap memory handler with 1000 bytes
- `int * ip = lh.Alloc<int> (5);` : Allocate memory for 5 integers within the `LocalHeap`
- `lh.Available();` : return how much memory is left in the `LocalHeap`
- `lh.GetPointer();` : returns pointer of the actual position within the `LocalHeap`
- `lh.Cleanup();` : resets the `LocalHeap`-pointer to its origin
- `HeapReset hr(lh);` : The `HeapReset`-destructor resets the `LocalHeap`-pointer to the value of the time when the `HeapReset`-constructor was called (helpful for loops)

You may also take a look at `ngsolve/programming_demos/demo_std.cpp`

NGSolve's basic linear algebra classes

- `Vector<T>` and `Matrix<T>` : Vector and Matrix with memory allocation/deallocation, size is set during runtime
- `Vec<int,T>` and `Mat<int,int,T>` : Vector and Matrix with memory allocation/deallocation, size is known during compile time
- `FlatVector<T>` and `FlatMatrix<T>` : Vector and Matrix without memory handling (`LocalHeap`), size is set during runtime
- `CalcInverse`, `CholeskyFactors<T>` and `CalcEigenSystem` for local computations

You may also take a look at `ngsolve/programming_demos/demo_bla.cpp`

NGSolve's (local) finite element classes

short overview of most often used finite element classes:

- `IntegrationRule` : Information about number of integration points, integration points, and integration weights
- `IntegrationPoint` : Point on reference domain and weight
- `ElementTransformation` : Transformation information form reference to physical domain
- `SpecificIntegrationPoint` : Has an `IntegrationPoint` and the element transformation information
- `FiniteElement` : shape functions on the reference domain for a certain element type (details later)
- `CoefficientFunction` : class for saving space-dependent functions
- `LinearFormIntegrator` and `BilinearFormIntegrator` : class for defining integrator-blocks (details later)

You may also take a look at `ngsolve/programming_demos/demo_fem.cpp`

(1) Defining Finite Elements

Behind every Finite Element Space there has to be a Finite Element, which defines shape functions on the reference domain.

Example: linear shape functions on a triangle

All FiniteElements are derived from `class FiniteElement`.
In our case we derive from `class ScalarFiniteElement`:

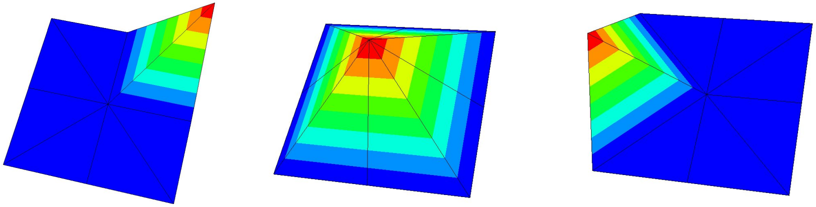
```
class MyLinearTrig :  public ScalarFiniteElement<2>
```

We have to implement:

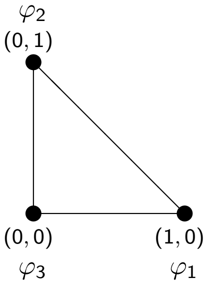
- `virtual void CalcShape`
  `(const IntegrationPoint & ip, FlatVector<> shape) const;`
- `virtual void CalcDShape`
  `(const IntegrationPoint & ip, FlatVector<> shape) const;`

These functions calculate point evaluation of the shape functions or there derivatives on the reference domain and put them into the `FlatVector` shape. shape is an M-dimensional vector or an $M \times N$ matrix, where M is the number of degrees of freedom corresponding to the actual `FiniteElement` and N are the space dimensions.

Linear 'Hat' Finite Elements



We define the linear elements on the reference element:



$$\varphi_1 = x$$
$$\varphi_2 = y$$
$$\varphi_3 = 1 - x - y$$

CalcShape

```
void MyLinearTrig ::  CalcShape
  (const IntegrationPoint & ip,
  FlatVector<> shape) const
{
  double x = ip(0);
  double y = ip(1);
  shape(0) = x;
  shape(1) = y;
  shape(2) = 1-x-y;
}
```

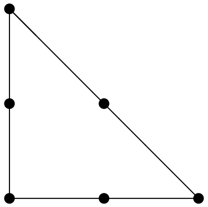
CalcDShape

```
void MyLinearTrig ::  CalcDShape
  (const IntegrationPoint & ip,
  FlatMatrixFixWidth<2> dshape) const
{
  dshape(0,0) = 1;
  dshape(0,1) = 0;
  dshape(1,0) = 0;
  dshape(1,1) = 1;
  dshape(2,0) = -1;
  dshape(2,1) = -1;
}
```

2nd Order Nodal Finite Elements

The vertex basis functions are

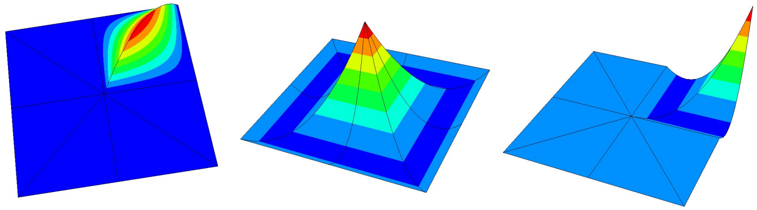
$$\varphi_i^{Vertex} = \lambda_i(2\lambda_i - 1)$$



and the edge basis functions are

$$\varphi_i^{Edge} = 4\lambda_j\lambda_k$$

with j, k the nodes on the corresponding edge i.



CalcShape (Vertex and Edges Basis Functions)

```
 void MyQuadraticTrig ::  CalcShape
    (const IntegrationPoint & ip, FlatVector<> shape) const
{
  // use barycentric coordinates x, y, 1-x-y:
  double lam[3] = { ip(0), ip(1), 1-ip(0)-ip(1) };
  // vertex basis functions:
  for (int i = 0; i < 3; i++)
      shape(i) = lam[i] * (2 * lam[i] - 1);
  // edge basis functions:
  const EDGE * edges = ElementTopology::GetEdges (ET_TRIG);
  // table provides connection of edges and vertices
  // i-th edge is between vertex
  // edges[i][0] and edges[i][1]
  for (int i = 0; i < 3; i++)
      shape(3+i) = 4 * lam[edges[i][0]] * lam[edges[i][1]];
}
```

CalcDShape (using Automated Differentiation)

```
void MyQuadraticTrig ::  CalcDShape
  (const IntegrationPoint & ip,
  FlatMatrixFixWidth<2> dshape) const
{
  // Use automatic (exact !)  differentiation
  // with overloaded data-types
  // value of x, gradient is 0-th unit vector (1,0)
  AutoDiff<2> x (ip(0), 0);
  // value of y, gradient is 1-th unit vector (0,1)
  AutoDiff<2> y (ip(1), 1);
  AutoDiff<2> lam[3] =  x, y, 1-x-y ;
  // vertex basis functions:
  for (int i = 0; i < 3; i++) {
    AutoDiff<2> shape = lam[i] * (2 * lam[i] - 1);
    dshape(i,0) = shape.DValue(0); // x-derivative
    dshape(i,1) = shape.DValue(1); // y-derivative
  }
⋮
```

```
 ⋮
 // edge basis functions:
 const EDGE * edges = ElementTopology::GetEdges (ET_TRIG);
 for (int i = 0; i < 3; i++) {
   AutoDiff<2> shape = 4 * lam[edges[i][0]] * lam[edges[i][1]];
   dshape(3+i,0) = shape.DValue(0); // x-derivative
   dshape(3+i,1) = shape.DValue(1); // y-derivative
 }
}
```

(2) Finite Element Spaces

The Finite Element Space has to provide:

- `Update()` : calculates number of degrees of freedoms
- `GetNDof()` : gives the total number of degrees of freedoms of the whole space
- `GetDofNrs(..)` : gives the indices of all degrees of freedom that are coupled with one element
- `GetSDofNrs(..)` : gives the indices of all degrees of freedom that are coupled with one surface element
- `GetFE(..)` : gives the finite element of an element
- `GetSFE(..)` : gives the finite element of a surface element

Remark: If you use FESpaces with `dim`$= M > 1$, then one degree of freedom is an M-dimensional vector!

Update

```
 void MyFESpace ::  Update(LocalHeap & lh)
{
  ⋮
  nvert = ma.GetNV();
  // number of dofs:
  if (!secondorder)
      // number of vertices
      ndof = nvert;
  else
      // num vertices + num edges
      ndof = nvert + ma.GetNEdges();
}
```

Connecting local to global dofs

```
void MyFESpace ::  GetDofNrs
  (int elnr, Array<int> & dnums) const
{   // returns dofs of element number elnr
  dnums.SetSize(0);
  Array<int> vert_nums;
  ma.GetElVertices (elnr, vert_nums); // global vertex numbers
  // first 3 dofs are vertex numbers:
  for (int i = 0; i < 3; i++)
    dnums.Append (vert_nums[i]);
  if (secondorder)
  {
  // 3 more dofs on edges:
  Array<int> edge_nums;
  ma.GetElEdges (elnr, edge_nums); // global edge numbers
  for (int i = 0; i < 3; i++)
    dnums.Append (nvert+edge_nums[i]);
  }
}
```

Connecting local to global dofs on the boundary

```
void MyFESpace :: GetSDofNrs
  (int elnr, Array<int> & dnums) const
{   // returns dofs of surface element number elnr
  dnums.SetSize(0);
  Array<int> vert_nums;
  ma.GetSElVertices (elnr, vert_nums); // global vertex numbers
  // first 2 dofs are vertex numbers:
  for (int i = 0; i < 2; i++)
    dnums.Append (vert_nums[i]);
  if (secondorder)
  {
  // 1 more dofs on the edges:
  Array<int> edge_nums;
  ma.GetSElEdges (elnr, edge_nums); // global edge number
  dnums.Append (nvert+edge_nums[0]);
  }
}
```

GetFE and GetSFE

```
MyFESpace :: MyFESpace (const MeshAccess & ama,
       const Flags & flags) : FESpace (ama, flags)
{
  ...
  reference_element = new MyLinearTrig;
  reference_surface_element = new FE_Segm1;
  ...
}

const FiniteElement & MyFESpace :: GetFE (int elnr, LocalHeap & lh) const
{
  if (ma.GetElType(elnr) == ET_TRIG)
    return *reference_element;
  throw Exception ("Sorry, only triangular elements are supported");
}

const FiniteElement & MyFESpace :: GetSFE (int selnr, LocalHeap & lh) const
{
  return *reference_surface_element;
}
```

numproc shapetester

With the help of the `numproc shapetester` you are able to look at your finite element shape functions in the physical domain very easily:

```
geometry = square.in2d
mesh = square.vol

shared = liblilngsolve

define fespace v -lilmyfespace

define gridfunction u -fespace=v

numproc shapetester np1 -gridfunction=u
```

(3) Bilinear- and LinearFormIntegrators

- The Bilinear- or LinearFormIntegrator (as well as all other objects that shall be used from the PDE-level) is registered with the help of the constructor of the class `Init`.
- The integer parameters of `AddBFIntegrator` are (in order): space dimension, number of coefficients
- When a Bilinear- or a LinearFormIntegrator is used in a PDE-File the `Create`-Function is called with an Array of `CoefficientFunctions`.

```
class MyLaplaceIntegrator : public BilinearFormIntegrator
{
  CoefficientFunction * coef_lambda;
public:
  static Integrator * Create (Array<CoefficientFunction*> & coeffs)
  { return new MyLaplaceIntegrator (coeffs[0]);  }
};
...
namespace init_mylaplace
{
  class Init { public: Init (); };

  Init::Init(){
    GetIntegrators().AddBFIntegrator ("lilmylaplace", 2, 1,
                                      MyLaplaceIntegrator::Create);}
  Init init;
}
```

Every Integrator essentially just needs to provide a Function which computes the element matrices or vectors.
In NGSolve this method is still called `AssembleElementMatrix` or `AssembleElementVector`.
Those methods get

- The Finite Element (the reference element and the shape functions)
- The Transformation from the reference element to the physical element
- A reference to a (small) matrix/vector which has as much entries as the element has degrees of freedom
- A reference to the LocalHeap to assign temporary memory efficiently

```
void My...Integrator ::
AssembleElementMatrix (const FiniteElement & base_fel,
                       const ElementTransformation & eltrans,
                       FlatMatrix<double> & elmat,
                       LocalHeap & lh) const
{ ... }
```

An implementation of the laplace-integrator

$$\int_T \lambda(x)(\nabla\phi_j)^T(\nabla\phi_i)\,dx \approx \sum_k \lambda(x_k)w_k|\det(F)|((\hat{\nabla}\hat{\phi}_j)^T F^{-1})(F^{-T}\hat{\nabla}\hat{\phi}_i)$$

```
AssembleElementMatrix (...)
{
  const ScalarFiniteElement<2> & fel =
    dynamic_cast<const ScalarFiniteElement<2> &> (base_fel);
  int ndof = fel.GetNDof();
  elmat = 0;
  Matrix<> dshape_ref(ndof, 2); // gradient on reference element
  Matrix<> dshape(ndof, 2);     // gradient on mapped element
  const IntegrationRule & ir =
    SelectIntegrationRule (fel.ElementType(), 2*fel.Order());
  for (int i = 0 ; i < ir.GetNIP(); i++)
    {
      SpecificIntegrationPoint<2,2> sip(ir[i], eltrans, lh);
      double lam = coef_lambda -> Evaluate (sip);
      fel.CalcDShape (ir[i], dshape_ref);
      dshape = dshape_ref * sip.GetJacobianInverse();
      double fac = sip.IP().Weight() * fabs (sip.GetJacobiDet());
      elmat += (fac*lam) * dshape * Trans(dshape);
    }
}
```

(4) Assembling

So far we can assemble element matrices and vectors. We just have to put them together into a (sparse) matrix before solving the linear system.
This is done by the BilinearForm and the LinearForm. The according function is `Assemble` One possible implementation could look like:

```
template <typename SCAL = double>
void BilinearForm :: Assemble (LocalHeap & lh) {
  Array<int> dnums;
  Matrix<SCAL> Matrix elmat;
  ElementTransformation eltrans;
  int ndof = fespace.GetNDof();
  BaseMatrix & mat = GetMatrix();
  mat = 0.0;
  for (int i = 0; i < ma.GetNE(); i++)  {
      HeapReset hr(lh);
      if (!fespace.DefinedOn (ma.GetElIndex (i))) continue;
      ma.GetElementTransformation (i, eltrans);
      const FiniteElement & fel = fespace.GetFE (i);
      fespace.GetDofNrs (i, dnums);
      for (int j = 0; j < parts.Size(); j++)  {
          const BilinearFormIntegrator & bfi = *parts.Get(j);
          if (bfi.BoundaryForm()) continue;

          bfi.AssembleElementMatrix (fel, eltrans, elmat, lh);
          mat->AddElementMatrix (dnums, elmat);
      }
  }
  for (int i = 0; i < ma.GetNSE(); i++) ...
}
```

(5) Solving the System (all in one)

instead of defining FESpaces, Bilinearforms, Linearforms and Integrators on the PDE-level, we can do everything directly in the numproc.

```
class NumProcAllInOne : public NumProc
{
protected:
  FESpace * fes;
  GridFunction *gfu;
  BilinearForm *bfa;
  LinearForm * lff;

public:

  NumProcAllInOne (PDE & apde, const Flags & flags)
    : NumProc (apde)
  {
    ...
```

```
...
Flags flags_fes;
flags_fes.SetFlag ("order", 4);
fes = new H1HighOrderFESpace (ma, flags_fes);

Flags flags_gfu;
gfu = new T_GridFunction<double> (*fes, "u", flags_gfu);

Flags flags_bfa;
bfa = new T_BilinearFormSymmetric<double>
       (*fes, "a", flags_bfa);

BilinearFormIntegrator * bfi;
bfi = new LaplaceIntegrator<2>
       (new ConstantCoefficientFunction (1));
bfa -> AddIntegrator (bfi);
...
```

```
  ...
  Array<double> penalty(ma.GetNBoundaries());
  penalty = 0.0;
  penalty[0] = 1e10;

  bfi = new RobinIntegrator<2>
          (new DomainConstantCoefficientFunction (penalty));
  bfa -> AddIntegrator (bfi);

  bfi = new MassIntegrator<2>
          (new ConstantCoefficientFunction (1));
  bfa -> AddIntegrator (bfi);

  Flags flags_lff;
  lff = new T_LinearForm<double> (*fes, "f", flags_lff);

  LinearFormIntegrator * lfi;
  lfi = new SourceIntegrator<2>
          (new ConstantCoefficientFunction (5));
  lff -> AddIntegrator (lfi);
}
```

```
virtual void Do (LocalHeap & lh)
{
  fes -> Update(lh);
  gfu -> Update();
  bfa -> Assemble(lh);
  lff -> Assemble(lh);

  const BaseMatrix & mata = bfa -> GetMatrix();
  const BaseVector & vecf = lff -> GetVector();
  BaseVector & vecu = gfu -> GetVector();

  BaseMatrix * inverse = dynamic_cast<const BaseSparseMatrix&>
         (mata).InverseMatrix();

  vecu = (*inverse) * vecf;

  delete inverse;
}
```

(6) Using B^{T}DB Integrators

If we assume to have integrators of the Form

$$A(u,v) = \int_\Omega B(v)^T D B(u)\, dx \quad \text{respectively} \quad f(v) = \int_\Omega d^T B(v)\, dx$$

- $B(u)$ is some differential operator, like $B(u) = \operatorname{curl} u$, $B(u) = \nabla u$, ...
- D is a coefficient matrix like $\lambda(x) I$
- d is a coefficient vector like $\lambda(x) e_x$

We can use this structure for our element matrix/vector computation. We define the integration for all Bilinear- and Linearforms of that type and then we just have to define the differential operators and coefficient matrices or vectors to build up the integrators.

```
template <int D, typename FEL = ScalarFiniteElement<D> >
class LaplaceIntegrator
  : public T_BDBIntegrator<DiffOpGradient<D>, DiagDMat<D>, FEL>
...
template <int D>
class MassIntegrator
  : public T_BDBIntegrator<DiffOpId<D>, DiagDMat<1>, ScalarFiniteElement<D> >
...
template <int D>
class ElasticityIntegrator
  : public T_BDBIntegrator<DiffOpStrain<D>, ElasticityDMat<D>, ScalarFiniteElement<D> >
```

B^TDB-Integrators: computing element matrices/vectors

(taken from bdbintegrator.hpp)

```
virtual void  AssembleElementMatrix (...) {
  const FEL & fel = static_cast<const FEL&> (bfel);
  int ndof = fel.GetNDof();
  elmat = 0;
  FlatMatrixFixHeight<DIM_DMAT, double> bmat (ndof * DIM, locheap);
  FlatMatrixFixHeight<DIM_DMAT, double> dbmat (ndof * DIM, locheap);
  Mat<DIM_DMAT,DIM_DMAT> dmat;
  const IntegrationRule & ir = GetIntegrationRule (fel,eltrans.HigherIntegrationOrde
  for (int i = 0 ; i < ir.GetNIP(); i++)
  {
    HeapReset hr(locheap);
    SpecificIntegrationPoint<DIM_ELEMENT,DIM_SPACE>
      sip(ir[i], eltrans, locheap);
    dmatop.GenerateMatrix (fel, sip, dmat, locheap);
    DIFFOP::GenerateMatrix (fel, sip, bmat, locheap);
    double fac =
      fabs (sip.GetJacobiDet()) * sip.IP().Weight();
    dbmat = fac * (dmat * bmat);
    elmat += Trans (bmat) * dbmat;
  }
}
```

We now want to solve $-\operatorname{div}(\alpha \nabla u) = f$ in mixed formulation:

$$\begin{array}{rclr} \operatorname{div}(\sigma) & = & -f & \text{on } \Omega \\ \frac{1}{\alpha}\sigma & = & \nabla u & \text{on } \Omega \end{array} \Rightarrow \begin{array}{rcrcll} \frac{1}{\alpha}(\sigma,\tau) & + & (\operatorname{div}(\tau),u) & = & (u_D,\tau_n)_\Gamma & \forall \tau \\ (\operatorname{div}(\sigma),v) & & & = & (-f,v) & \forall v \end{array}$$

We choose $u, v \in \mathcal{P}^n$ and $\sigma, \tau \in \mathcal{W}^{n+1}$.

In NGSolve
$\mathcal{P}^n =$ `L2HighOrderFESpace`(order$= n$) and
$\mathcal{W}^{n+1} =$ `HDivHighOrderFESpace`(order$= n+1$)

We call the direct sum of both Spaces (the “Compound”) C and $U = (u, \sigma)$ and get:

$$A(U,V) = F(V)$$

(7a) Using Compound Finite Element Spaces

```
 class MixedHDivL2FESpace : public CompoundFESpace
{
  public:
    MixedHDivL2FESpace (const MeshAccess & ama,
           const Array<const FESpace*> & aspaces, const Flags & flags)
    : CompoundFESpace (ama, aspaces,flags)
    { }
    ...
    static FESpace * Create (const MeshAccess & ma, const Flags & flags)
    {
      Array<const FESpace*> spaces(2);

      Flags l2flags(flags), hdivflags(flags);
      int order = int (flags.GetNumFlag ("order", 0));
      l2flags.SetFlag ("order", order);
      hdivflags.SetFlag("order", order+1);
      spaces[0] = new L2HighOrderFESpace (ma, l2flags);
      spaces[1] = new HDivHighOrderFESpace (ma, hdivflags);
      Flags emptyflags;
      MixedHDivL2FESpace * fes = new MixedHDivL2FESpace (ma, spaces, emptyflags);
      return fes;
    }
    ...
};
```

(7b) B^{T}DB-Integrator for mixed laplace problem

back to the mixed laplace example

$$A(U,V) = F(V)$$

The Linearform $F(V) = F(v,\tau) = (-f, v) + (u_D, \tau_n)_\Gamma$ can be handled on the PDE-level directly with the help of the -comp-Flag:

```
define linearform rhs -fespace=v
  neumannhdiv u_D -comp=2
  source minusf -comp=1
```

We write the Bilinearform in B^{T}DB-Form: $A(U,V) = (DB(u), B(v))$ with:

$$B(U) = B(u,\sigma) = \begin{pmatrix} u \\ \sigma_x \\ \sigma_y \\ \sigma_z \\ \operatorname{div}(\sigma) \end{pmatrix} \quad D = \begin{pmatrix} & & & & 1 \\ & \frac{1}{\alpha} & & & \\ & & \frac{1}{\alpha} & & \\ & & & \frac{1}{\alpha} & \\ 1 & & & & \end{pmatrix}$$

DiffOpMixedLaplace

Now we just have to implement `DiffOpMixedLaplace` and `MixedLaplaceDMat`. Here as an 2D-example:

```
class DiffOpMixedLaplace : public DiffOp<DiffOpMixedLaplace>
{

public:
  enum { DIM = 1 };           // just one copy of the spaces
  enum { DIM_SPACE = 2 };     // 2D space
  enum { DIM_ELEMENT = 2 };   // 2D elements
  enum { DIM_DMAT = 4 };      // D-matrix is 5x5
  enum { DIFFORDER = 0 };     // minimal differential order
...
```

```
template <typename FEL, typename SIP, typename MAT>
static void GenerateMatrix (const FEL & bfel, const SIP & sip,
                            MAT & mat, LocalHeap & lh)
{
  const CompoundFiniteElement & cfel =
    dynamic_cast<const CompoundFiniteElement&> (bfel);
  const L2HighOrderFiniteElement<2> & fel_u =
    dynamic_cast<const L2HighOrderFiniteElement<2>&> (cfel[0]);
  const HDivHighOrderFiniteElement<2> & fel_sigma =
    dynamic_cast<const HDivHighOrderFiniteElement<2>&> (cfel[1]);
  int nd_l2 = fel_u.GetNDof(), nd_hdiv = fel_sigma.GetNDof();

  FlatVector<> shape_u(nd_l2, lh);
  shape_u = fel_u.GetShape(sip.IP(),lh);

  FlatMatrix<> shape_sigma(nd_hdiv,2,lh);
  FlatVector<> shape_divsigma(nd_hdiv, lh);
  fel_sigma.CalcMappedShape(sip,shape_sigma);
  fel_sigma.CalcMappedDivShape (sip, shape_divsigma);

  mat = 0.0;
  mat.Row(0).Range(0,nd_l2) = shape_u;
  mat.Row(1).Range(nd_l2,nd_l2+nd_hdiv) = shape_sigma.Col(0);
  mat.Row(2).Range(nd_l2,nd_l2+nd_hdiv) = shape_sigma.Col(1);
  mat.Row(3).Range(nd_l2,nd_l2+nd_hdiv) = shape_divsigma;
}
```

MixedLaplaceDMat

```
class MixedLaplaceDMat : public DMatOp<MixedLaplaceDMat>
{
  CoefficientFunction * alpha;
public:

  enum { DIM_DMAT = 4 };

  MixedLaplaceDMat (CoefficientFunction * aalpha) : alpha(aalpha) { ; }

  template <typename FEL, typename SIP, typename MAT>
  void GenerateMatrix (const FEL & fel, const SIP & sip,
       MAT & mat, LocalHeap & lh) const
  {
    mat = 0;
    double val = alpha -> Evaluate (sip);
    for (int i = 1; i < 3; i++)
      mat(i, i) = 1.0/val;
    mat(0,3) = 1.0;
    mat(3,0) = 1.0;
  }
};
```

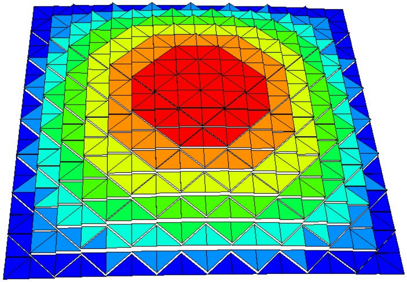
MixedLaplaceIntegrator

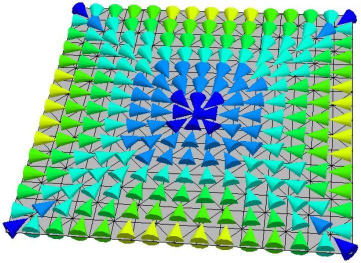
```
class MixedLaplaceIntegrator
  : public T_BDBIntegrator<DiffOpMixedLaplace,
    MixedLaplaceDMat, FiniteElement>
{
public:
  MixedLaplaceIntegrator (CoefficientFunction * coeff)
    :  T_BDBIntegrator<DiffOpMixedLaplace, MixedLaplaceDMat,
       FiniteElement>
  (MixedLaplaceDMat (coeff))
  { ; }

  static Integrator * Create
      (Array<CoefficientFunction*> & coeffs)
  {
    return new MixedLaplaceIntegrator (coeffs[0]);
  }

  virtual string Name () const { return "MixedLaplace"; }
};
```

P0W1-Approximation of the Laplace-Problem:





solution u on the left side and σ on the right side

You may also take a look at `demo_stokes.cpp`

(8) Transient PDEs

Now we also want to solve unsteady problems. Of course the first (easiest) choice is an unsteady heatflow problem:

$$\frac{\partial u}{\partial t} - \Delta u = sin(t) f \qquad + \text{ic's} + \text{bc's}$$

Semidiscrete form:

$$M \frac{\partial u}{\partial t} + Au = sin(t) f$$

We want to discretize this with an implicit euler method:

$$M \frac{\Delta u}{\Delta t} + A \Delta u = sin(t) f - Au^n$$

From the pde-file we get:

- BilinearForm A
- BilinearForm M
- LinearForm f
- GridFunction u
- Δt, t_{end}

Reading objects from pde-files

```
  NumProcParabolic (PDE & apde, const Flags & flags)
  : NumProc (apde)
{
  bfa = pde.GetBilinearForm
        (flags.GetStringFlag ("bilinearforma", "a"));
  bfm = pde.GetBilinearForm
        (flags.GetStringFlag ("bilinearformm", "m"));
  lff = pde.GetLinearForm
        (flags.GetStringFlag ("linearform", "f"));
  gfu = pde.GetGridFunction
        (flags.GetStringFlag ("gridfunction", "u"));

  dt = flags.GetNumFlag ("dt", 0.001);
  tend = flags.GetNumFlag ("tend", 1);
}
```

Solving the unsteady problem

```
virtual void Do(LocalHeap & lh)
{
  const BaseMatrix & mata = bfa->GetMatrix();
  const BaseMatrix & matm = bfm->GetMatrix();
  const BaseVector & vecf = lff->GetVector();
  BaseVector & vecu = gfu->GetVector();
  BaseMatrix & summat = *matm.CreateMatrix();
  BaseVector & d = *vecu.CreateVector();
  BaseVector & w = *vecu.CreateVector();
  summat.AsVector() = (1.0/dt) * matm.AsVector() + mata.AsVector();
  BaseMatrix & invmat = * dynamic_cast<BaseSparseMatrix&> (summat)
       . InverseMatrix();
  vecu = 0;
  for (double t = 0; t <= tend; t += dt)
    {
      d = sin(t) * vecf - mata * vecu;
      w = invmat * d;
      vecu += w;
      Ng_Redraw ();
    }
}
```

(9) Nonlinear PDEs

As soon as you have a nonlinear problem you have to write your own `integrators`, `numprocs`, `etc....` If the nonlinearity of what fits into "BilinearForm" only depends on the solution itself you can use the method `AssembleLinearization` in your numproc:
Use `bfa.AssembleLinearization (vecu, lh);` on a BilinearForm bfa and it will

- Compute and assemble all solution-independent (meaning really bilinear) Integrators
- Compute and assemble all solution-dependent Integrators. On every element `AssembleLinearizedElementMatrix(..,..,FlatVector<double> & elveclin,..,..)const` is called then. So your Integrator has to implement this.

There is also the possib see `demo_nonlinear.cpp`

Thank you for your attention!