Netgen/NGSolve Tutorial Part 3 - Programming with Netgen/NGSolve

Christoph Lehrenfeld

March 24, 2015

Outline

Aim of tutorial

In this tutorial we want to explain how to add additional functionality to NGSolve, e.g. by adding your own Finite Element space or your own integrator, etc..

Important components

The basic ingredients that are needed to solve a linear elliptic PDE with NGSolve are:

- a `FiniteElement` defining shape functions on a reference domain
- a `FESpace` (Finite Element Space) handling the connection between the `FiniteElements` and the global mesh
- `BilinearFormIntegrators` and `LinearFormIntegrators` to compute elementwise matrices and vectors

Important components

- An algorithm assembling elementwise contributions to the global system matrix and the vectors (`Bi-` and `LinearForms`)
- Linear Algebra Routines solving the arising linear equation systems
- `numprocs` (numerical procedures) to solve bvp, instationary problem, postprocess results, etc..

An example library where many of these components are introduced and used in a simple setting is the package `my_little_ngsolve` which you can also download from `sourceforge`.

The layers of NGSolve - overview

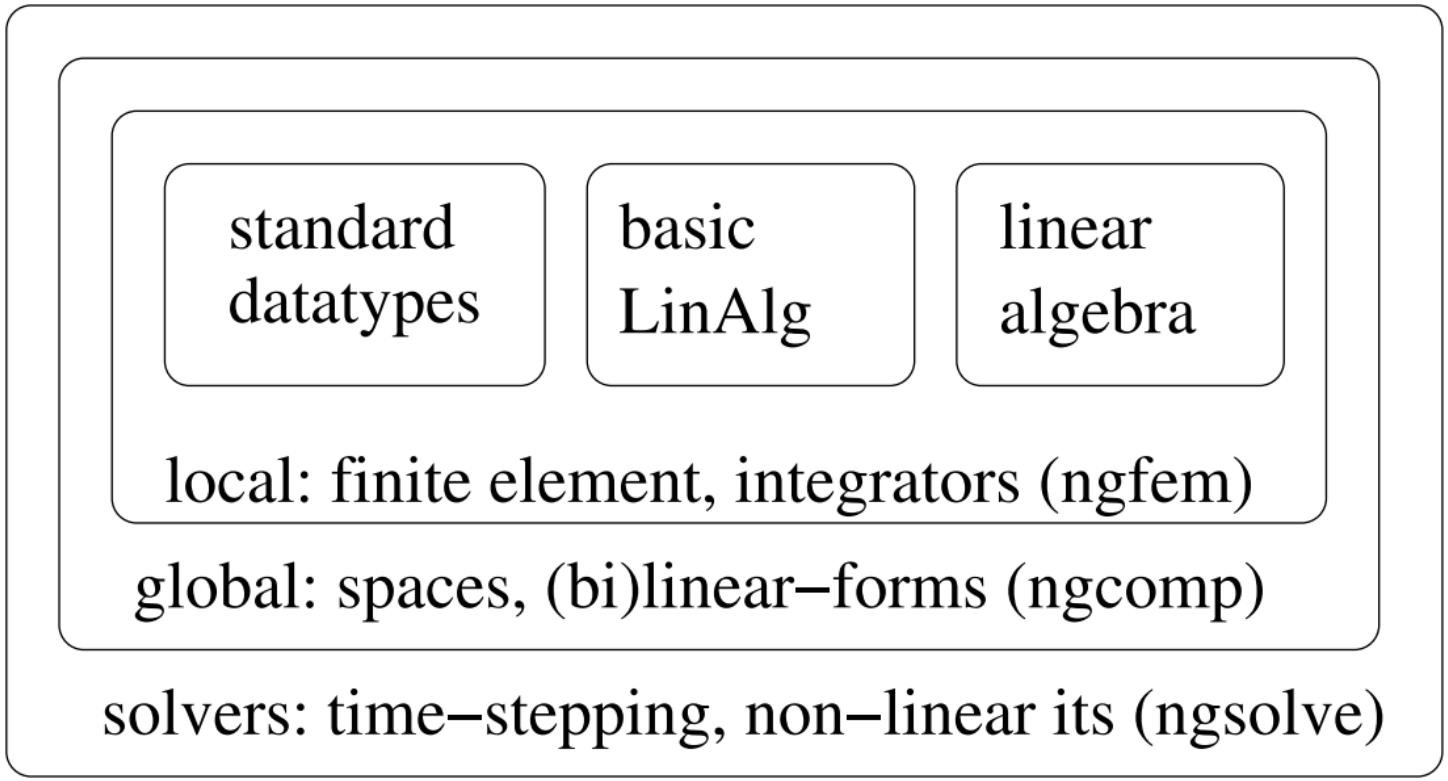

Figure: layers of NGSolve

The layers of NGSolve - levels

There are several layers of NGSolve implementations:

`ngstd`	(general)	.., `Array`, `LocalHeap`, `Table`, ..
`basiclinalg`	(local)	.., `Mat`, `Vec`, ..
`fem`	(local)	.., `Integrators`, `FiniteElements`, ..
`comp`	(global)	.., `FESpaces` (`H1HOFESpace`, `HCurlHOFESpace`,..), ..., `Linear-` and `BilinearForms`, `GridFunction`, ..., `MeshAccess`, `Preconditioners`, ..
`linalg`	(global)	.., `VVectors`, direct solvers, iterative solvers, ..
`multigrid`	(global)	.., `Smoother`, `Prolongation`, ...
`parallel`	(global)	MPI-Implementation of some NGSolve
`solve`	(solvers)	PDE-Parser, `numprocs` as `bvp` ...

NGSolve's basic classes (1) - Arrays and Tables

Arrays and Tables:

- `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 entrysize

NGSolve's basic classes (2a) - LocalHeap

For local operations NGSolve uses the concept of a LocalHeap for memory management. The LocalHeap reserve a certain block of memory and initializes a heap pointer which always points to the end of the "used" part of that block. At the beginning this is the beginning of that memory block because it is not used. "Allocation" on the LocalHeap is done by increasing the heap pointer. Memory is typically freed (i.e. the heap pointer is reset to some old value) after some portion of code (e.g. calculation of an element matrix or an element vector).

NGSolve's basic classes (2b) - LocalHeap - Example usage

- Initializing heap memory handler with 1000 bytes:

```
LocalHeap lh(1000)
```

- Allocate memory for 5 integers on the LocalHeap:

```
int * ip = lh.Alloc<int>(5);
```

- Return how much memory is left on the LocalHeap:

```
lh.Available();
```

- Return heap pointer which points to the first entry within the memory block (which is reserved by the LocalHeap) which is not yet used.

```
lh.GetPointer();
```

NGSolve's basic classes (2c) - LocalHeap - Example usage

- Reset heap pointer to its origin

```
lh.Cleanup();
```

- A HeapReset stores the heap pointer location at its construction and restores it at its destruction, s.t. the allocation on the LocalHeap that takes place during the lifetime of the HeapReset-object is set free afterwards:

```
HeapReset hr(lh);
```

Take a look into standalone/demo_std.cpp in my_little_ngsolve.

NGSolve's basic linear algebra classes

Most of important linear algebra classes in NGSolve:

- `Vector<T>`, `Matrix<T>`: Vectors and matrices with memory (de)allocation, size is set during runtime
- `Vec<int,T>` and `Mat<int,int,T>`: Vector and Matrix with memory (de)allocation, size is known already during compile time
- `FlatVector<T>` and `FlatMatrix<T>`: Vector and Matrix without memory handling (`LocalHeap`), size is set during runtime
- `FlatMatrixFixWidth<int,T>`, `FlatMatrixFixHeight<int,T>`: Vector and Matrix without memory handling (`LocalHeap`), `Width` (`Height`) is set during compile time, `Height` (`Width`) is set during runtime.

You may also want to take a look at `standalone/demo_bla.cpp`

NGSolve's classes for local computations (FE level)

IntegrationRules, Transformations:

- `IntegrationPoint` : Point on reference domain and weight
- `IntegrationRule` : `Array` of `IntegrationPoints`
- `ElementTransformation` : Transformation information from reference to physical domain
- `MappedIntegrationPoint` : Has an `IntegrationPoint` and the `ElementTransformation` information

NGSolve's classes for local computations (FE level)

Finite elements, coefficients and integrators:

- `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 want to take a look at `standalone/demo_fem.cpp` in `my_little_ngsolve`.

Download my_little_ngsolve

Download my_little_ngsolve from sourceforge by cloning the corresponding git repository

```
git clone git://git.code.sf.net/p/ngsolve/ml-ngs mlngs
```

Building

The my-little-ngsolve project uses the `ngscxx`-compiler-wrapper and a `Makefile` to build shared libraries. These are however not installed into another library but just locally. If your environment variables are set correctly, i.e. the `NETGENDIR` point to a location where `ngscxx` is present, you can simply type

```
make
```

to compile the my-little-ngsolve project.

make sure that the `LD_LIBRARY_PATH` points to the local directory `./` or set it manually to point to the location of `my-little-ngsolve`.

Testing

Start netgen in the directory of `my_little_ngsolve` and load a pde-file from the same directory.

A new FiniteElement (1) - myElement.*pp

A FiniteElement defines shape functions on a reference domain, e.g. on the reference triangle, the reference hexahedra, ...
In myElement.hpp and myElement.cpp a the standard linear and quadratic FiniteElements on a triangle are defined.
The corresponding FiniteElements are scalar valued and "live" in two space dimension. Some functionality is inherited from ScalarFiniteElement<2>:

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

A new FiniteElement (2a) - myElement.*pp

As a ScalarFiniteElement the new FiniteElement has to implement the following four member functions:

- a constructor, e.g.:

```
MyLinearTrig ();
```

- a Getter for the geometrical information of the element type, e.g. a ET_TRIG

```
virtual ELEMENT_TYPE ElementType() const { return ET_TRIG; }
```

A new FiniteElement (2b) - myElement.*pp

- a function which evaluates the function values of the finite element. Note that this evaluation is w.r.t. the reference element.

```
virtual void CalcShape (const IntegrationPoint & ip,
                        SliceVector<> shape) const;
```

  The result is stored in dshape which is M-dimensional vector.

- a function which evaluates the gradients of the FiniteElement. Note that this evaluation is w.r.t. the reference element.

```
virtual void CalcDShape (const IntegrationPoint & ip,
                         SliceMatrix<> dshape) const;
```

  The result is stored in dshape which is an M x N matrix, where M is the number of degrees of freedom and N the space dimension (here 2).

A new FiniteElement (3) - myElement.*pp

In myElement.cpp you can find the according implementation of the above mentioned functions. The code is pretty self-explanatory, we just make one remark:

- The constructor essentially only calls the base constructor of ScalarFiniteElement<2> which gets the arguments
  - "number of degrees of freedoms" which is 3 for the linear case
  - "order" of finite elements which is 1 here.

In myElement.*pp quadratic finite elements are also defined. Try to get the idea for those elements as well.

FESpace (responsibilities)

- How many dofs exist?
- Which dofs are relevant on a specific (boundary Element)?
- Which dofs correspond to which basis function (`FiniteElement`)?

FESpace (1) - myFESpace.*pp

The functionality of the FESpace is provided by the following functions:

- constructor/destructor: In the constructor flags are parsed and some preliminary information is stored.

```
MyFESpace (shared_ptr<MeshAccess> ama, const Flags & flags);
virtual ~MyFESpace ();
```

- ClassName() just defines the name of the space:

```
virtual string GetClassName () const;
```

FESpace (2) - myFESpace.*pp

- `Update()` takes care of the calculation of the total number of degrees of freedoms and prepares data structures for the mapping between (surface) elements and degrees of freedoms.

```
virtual void Update(LocalHeap & lh);
```

- Getter for the global number of degrees of freedom:

```
virtual int GetNDof () const;
```

FESpace (3) - myFESpace.*pp

- Mapping between (surface) elements and the corresponding degrees of freedom

```
virtual void GetDofNrs (int elnr, Array<int> & dnums) const;
virtual void GetSDofNrs (int selnr, Array<int> & dnums) const;
```

- Mapping between (surface) elements and the corresponding finite elements

```
virtual const FiniteElement & GetFE (int elnr, LocalHeap & lh)
  const;
virtual const FiniteElement & GetSFE (int selnr, LocalHeap & lh)
  const;
```

FESpace (4) - myFESpace.*pp

In myElement.cpp you can find the according implementation of the above mentioned functions. The code is pretty self-explanatory.

FESpace (5a) - Dirichlet boundaries

We briefly remark on how Dirichlet conditions are treated in NGSolve:

- In terms of the numbering:
  - the FESpace does not distinguish between Dirichlet dof and other dofs, i.e. `GetFE()` and `GetDofNrs()` does not care about Dirichlet degrees of freedoms.
  - Element and globale matrices and vectors are set up for more dofs than actually needed.
- Additional information provided by `GetFreeDofs()` which is defined in the `FESpace` base class `FESpace`. The marking of corresponding dofs as "Dirichlet" is implemented in the base class `FESpace` (in `FinalizeUpdate()`)

FESpace (5b) - Dirichlet boundaries

Using the information from `GetFreeDofs()`, we partition the linear system as:

$$\begin{pmatrix} A_{FF} & A_{FD} \\ A_{DF} & A_{DD} \end{pmatrix} \cdot \begin{pmatrix} u_F \\ u_D \end{pmatrix} \text{„} = \text{„} \begin{pmatrix} f_F \\ f_D \end{pmatrix}$$

with u_F the coefficients corresponding to the "free" dofs and u_D to the Dirichlet dofs.

We prescribe the Dirichlet coefficients, such that we are left with solving the first equation:

$$\begin{pmatrix} A_{FF} & A_{FD} \end{pmatrix} \cdot \begin{pmatrix} u_F \\ u_D \end{pmatrix} = \begin{pmatrix} f_F \end{pmatrix}$$

$$A_{FF} u_F = f_F - A_{FD} u_D$$

FESpace (5c) - Dirichlet boundaries

Given an initial guess u^0, we can always write the solution as $u = u^0 + \Delta u$, where Δu is the solution to:

$$\Delta u_F = A_{FF}^{-1}(f_F - A_{FF}\,u_F^0 - A_{FD}\,u_D^0)$$

and $\Delta u_D = 0$ if u^0 fulfills the boundary conditions.
This is typically implemented as

$$u = u^0 + \underbrace{\begin{pmatrix} A_{FF}^{-1} & 0 \\ 0 & 0 \end{pmatrix}}_{\text{"}Inverse(freedofs)\text{"}} \cdot \left(\begin{pmatrix} f_F \\ f_D \end{pmatrix} - \begin{pmatrix} A_{FF} & A_{FD} \\ A_{DF} & A_{DD} \end{pmatrix} \begin{pmatrix} u_F^0 \\ u_D^0 \end{pmatrix} \right)$$

FESpace (6) - Registration

The FESpace has to be registered to NGSolve in order to make it available from the pde-file. This is done using

```
static RegisterFESpace<MyFESpace> initifes ("myfespace");
```

in myFESpace.hpp.

Testing FiniteElement/FESpace

using the numproc shapetester you can easily check the new basis functions:
test_fespace.pde:

```
1  geometry = square.in2d
2  mesh = square.vol
3  shared = libmyngsolve
4  fespace v -type=myfespace #-secondorder
5  gridfunction u -fespace=v
6  numproc shapetester np1 -gridfunction=u
```

Integrators

To define new partial differential equations you may also want to define different integrals in the weak formulation, this is done using `BilinearFormIntegrators` / `LinearFormIntegrators`.

In `myIntegrator.hpp` and `myIntegrator.cpp` the `laplace` integrator for $\int_\Omega \alpha \nabla u \nabla v dx$ and the `source` integrator for $\int_\Omega f v dx$ are reimplemented.

Both are good (and simple) examples explaining how these kind of integrators can be implemented.

BilinearFormIntegrators (1) - myIntegrator.*pp

Let us start with BilinearFormIntegrators. The essential functions that have to implemented are:

- The constructor which gets an array of CoefficientFunctions as input. These can describe the weights in integral but also constants like stabilization parameters or similar things.

```
MyLaplaceIntegrator (const
  Array<shared_ptr<CoefficientFunction>> & coeffs);
```

- A name:

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

BilinearFormIntegrators (2) - myIntegrator.*pp

- The dimension of the element, for volume elements this coincides with the dimension of the space, for boundary integrals it is one dimension smaller, here it is fixed to 2:

```
virtual int DimElement () const { return 2; }
```

- The dimension of the space

```
virtual int DimSpace () const { return 2; }
```

- Is the integrator implementing a boundary integral ?

```
virtual bool BoundaryForm () const { return false; }
```

BilinearFormIntegrators (3) - myIntegrator.*pp

- The core function of a BilinearFormIntegartion is CalcElementMatrix which calculates the element matrix and gets the information about the local finite elements (FiniteElement), the geometry-information in terms of the transformation from reference element to the physical domain. elmat is the result matrix and LocalHeap lh the Heap-memory which is provided.

```
virtual void
CalcElementMatrix (const FiniteElement & fel,
                   const ElementTransformation & eltrans,
                   FlatMatrix<double> elmat,
                   LocalHeap & lh) const;
```

BilinearFormIntegrators (4) - myIntegrator.*pp

- For the evaluation of fluxes the dimension has to be provided. For `laplace` this is 2 (as dimension is 2) and the flux is $(\alpha)\nabla u \in \mathbb{R}^2$.

```
virtual int DimFlux () const { return 2; }
```

BilinearFormIntegrators (5a) - myIntegrator.*pp (CalcFlux)

- `CalcFlux`: The computation of the flux (e.g. $(\alpha)\nabla u$) at one point (`BaseMappedIntegrationPoint`).
  - The combination of the `FiniteElement` and the coefficients `elx` provide a representation of a discrete solution on one element.
  - The result vector is the vector `flux`.
  - The boolean `applyd` asks prescribes if an additional operator (typically a simple scaling) should be applied, here this would be the scaling with α.
  - Again, the `LocalHeap` provides the memory for local computations.

BilinearFormIntegrators (5b) - myIntegrator.*pp (CalcFlux)

```
virtual void
CalcFlux (const FiniteElement & fel,
          const BaseMappedIntegrationPoint & bsip,
          FlatVector<double> elx,
          FlatVector<double> flux,
          bool applyd,
          LocalHeap & lh) const;
```

LinearFormIntegrators - myIntegrator.*pp

The functions for a `LinearFormIntegrator` are pretty much the same except for the fact that the core function is `CalcElementVector` instead of `CalcElementMatrix`.

In `myIntegrator.cpp` the `CalcElementxxx` functions for the laplace and the source integrator are implemented and very well documented. Read through the lines to get the main idea. Note that for the implementation of the integrals quadrature rules are typically used and transformation factors and rules have to be applied to recover the integral on the reference domain.

Assembling of matrices and vectors

The `bilinearform` which stores the `Integrators` and the information about the `FESpace` takes care of the assembling. For most problems it is not necessary to influence the assembling procedure directly. However in `myAssembling.cpp` it is demonstrated how the assembling (including the definition of integrators) can be implemented "from scratch".

Numerical procedures (bvp)

- Problem definition (matrices and vectors):
  - with `FESpaces` , `GridFunctions` , `Linear` - and `BilinearForms` (and `Integrators`)
  - `Linear` - and `BilinearForms` take care about the assembling
- Left to do: solving arising discrete problems. This is done in `numprocs`, numerical procedures.
- The standard numproc is the `numproc bvp` (boundary value problem) which solves the system $Au = f$
  - A defined through a `BilinearForm`
  - f defined through a `LinearForm`
  - u defined through a `GridFunction`

Numerical procedures (other tasks)

A numerical procedure can also be used to do other things, e.g.:

- setting values for a gridfunction (e.g. on the boundary):

  ```
  numproc setvalues npsv -gridfunction=u -coefficient=coef_u (-bounda
  ```

- changing visualization options:

  ```
  numproc visualization npvis ...
  ```

- calculating errors or difference between gridfunctions and other gridfunctions or coefficients

  ```
  numproc difference npdiff
  ```

- evaluating linearform $f(u)$ or solution along lines, ...

Numerical procedures (instationary problems) (1)

To solve simple instationary problems it is also sufficient to define a simple numerical procedure. In `my_little_ngsolve` this is done in `demo_instat.cpp` and `instat.pde`.

We briefly explain the structure of a NumProc:

- The contructor gets the PDE (container which has essentially all the information which already have been processed (from the pde-file)) and a number of flags:

  ```
  MyNumProc (PDE & apde, const Flags & flags)
  ```

  In the constructor reference to other objects like FESpaces, Gridfunctions, etc.. can be accessed and stored and options in terms of Flags can be processed.

Numerical procedures (instationary problems) (2)

- The actual work of the numerical procedure is typically done in the function

  ```
  virtual void Do (LocalHeap & lh)
  ```

  This function is automatically called from the pde-parser after the setup/assembly of FESpaces, GridFunctions, Bi/LinearForms and preconditioners.

Numerical procedures (instationary problems) (3)

Take a look at `demo_instat.cpp` and `instat.pde` to see how self-made `numprocs` can be used to solve your own problems. Which problem is solved in `instat.pde` and which time integration method is used there?

Different tasks for you:

- (simple):
  - solve the PDE: $\partial_t u - \alpha \Delta u = f$ in Ω_i, $i = 1, 2$. Use a geometrical configuration as in `ex2_twodom_poisson` and set $f = 0$ in the outer domain and $f = 1$ in the inner domain. Set the initial condition to 1.0 in the whole domain. Choose $\alpha_1 = 1$ and $\alpha_2 = 0$. Choose boundary conditions as you like.
- (advanced):
  - Define a FESpace which is piecewise linear (or higher order) and continuous but discontinuous across a fixed inner interface (e.g. surface 5 in `ex2_twodom_poisson`).

my_little_ngsolve files and generated files

- my_little_ngsolve
  - standalone/demo_std.cpp
  - standalone/demo_bla.cpp
  - standalone/demo_fem.cpp
  - myElement.hpp / myElement.cpp
  - myFESpace.hpp / myFESpace.cpp
  - myIntegrator.hpp / myIntegrator.cpp
  - myAssembling.cpp
  - demo_instat.cpp, instat.pde
- generated files from this document
  - test_fespace.pde
  - mlngs.pdf
  - mlngs_slides.pdf
  - mlngs.html