Introduction to

Gascoigne 3D

High Performance Adaptive Finite Element Toolkit

Tutorial and manuscript by Thomas Richter

thomas.richter@iwr.uni-heidelberg.de

March 9, 2014

Contents

1 Introduction

When referring to files, the following directories are used:

HOME Your home directory

GAS The base-directory of Gascoigne. For the workshop:

GAS/src The source- and include-files of Gascoigne.

Gascoigne is a software-library written in C++. To solve a partial differential equation with Gascoigne, a small program has to be written using the routines provided by Gascoigne. All functionality is gathered in the library-file FILE: `libGascoigneStd` . This library is *linked* to every application.

1.1 Installation

In this section we explain the usage of Gascoigne as a library for own applications.

Gascoigne requires a configuration file:

FILE: `HOME/.gascoignerc`

Here, some setting like "where is Gascoigne", "where are the libraries" are given. The minimal file looks like:

```
1 ##################################################
2 SET(GASCOIGNE_STD "~/Software/src/Gascoigne")
3
4 ##################################################
5 # Gascoigne Libraries & Binaries
6 SET(GASCOIGNE_LIBRARY_OUTPUT_PATH "~/Software/x86_64/lib")
7 SET(GASCOIGNE_EXECUTABLE_OUTPUT_PATH ".")
```

You need a file like this in the home directory. To build an application using Gascoigne at least two files and directories are required:

FILE: `TEST/bin/`
FILE: `TEST/src/main.cc`
FILE: `TEST/src/CMakeLists.txt`

The first directory will be used to store the executable file for this application. The other two files are the source code of the application and a file to control the compilation of the application. This could look like:

FILE: `TEST/src/CMakeLists.txt`

```
1 INCLUDE($ENV{HOME}/.gascoignerc)
2
3 INCLUDE(${GASCOIGNE_STD}/CMakeGlobal.txt)
4 LINK_LIBRARIES(${GASCOIGNE_LIBS})
5
6 ADD_EXECUTABLE("Test"  main.cc)
```

In the first line, the user's config file is read. The following two lines set global parameters for Gascoigne. The variable `GASCOIGNE_STD` is provided in the config-file, the variable `GASCOIGNE_LIBS` is given in the file `CMakeGlobal.txt`. The last line indicates the name of the application: `Test`, and lists all code-files to be compiled into the application.

A minimal example for a code-file (it does not really calculate something) looks like this:

FILE: `TEST/src/main.cc`

```
1 #include <iostream>
2 #include "stdloop.h"
3
4 int main(int argc, char**argv)
5 {
6   std::cout << "Create a Gascoigne Loop!" << std::endl;
7   Gascoigne::StdLoop Loop;
8 }
```

For building the application we use the program `cmake` to generate Makefiles. In the directory `TEST/bin/` type:

```
1 cmake ../src
2 make
```

The first line creates the FILE: `TEST/bin/Makefile` . It reads the FILE: `CMakeLists.txt` to put everything together. The second command starts the compilation and linking of the application and if everything works out, FILE: `TEST/bin/Test` should be the executable file which then can be started.

The command `cmake` is only needed to create new Makefiles. This is necessary, if new code-files are added to the application. If only source code is changed, e.g. the file FILE: `TEST/src/main.cc` , you only need to call `make` in the directory `TEST/bin`.

1.2 A minimal example solving a partial differential equation

Assume there exists a directory `FIRSTEXAMPLE` with sub-dirs `FIRSTEXAMPLE/bin` and `FIRSTEXAMPLE/src`. In the src-directory you need the files FILE: `main.cc` and FILE: `CMakeLists.txt` . in the bin-directory you call `cmake ../src` for configuration and then `make` for compilation.

FILE: `FIRSTEXAMPLE/src/main.cc`

```
#include "stdloop.h"
#include "problemdescriptorbase.h"
#include "constantrighthandside.h"
#include "zerodirichletdata.h"
#include "laplace2d.h"

using namespace Gascoigne;

class Problem : public ProblemDescriptorBase
{
  public:
    std::string GetName() const { return "First Problem"; }
    void BasicInit(const ParamFile* paramfile)
    {
      GetEquationPointer()   = new Laplace2d;
      GetRightHandSidePointer() = new OneRightHandSideData(1);
      GetDirichletDataPointer() = new ZeroDirichletData;

      ProblemDescriptorBase::BasicInit(paramfile);
    }
};

int main(int argc,char **argv)
{
  ParamFile paramfile("first.param");
  if (argc>1) paramfile.SetName(argv[1]);

  Problem PD;
  PD.BasicInit(&paramfile);

  ProblemContainer PC;
  PC.AddProblem("laplace", &PD);

  StdLoop loop;
  loop.BasicInit(&paramfile, &PC);
  loop.run("laplace");
}
```

The Problem

Here, the problem to be solved is defined. The `Problem` gathers all information on the problem to be solved. This is: which pde, what kind of boundary data, what is the right hand side, and so on.

By `PD.BasicInit()` we initialize the problem (will be described later on). This problem is then added to another class, the `ProblemContainer`. Here, we can gather different problems, each together with a keyword, which is `laplace` in this case. There are applications where it is necessary to solve different problems in the same program.

The problem set pointers to a large variety of classes. Here, we define the equation to be of type `Laplace2d`, an implementation of the two dimensional Laplace equation. For right hand side and Dirichlet data we define self written classes (will be explained later). In GASCOIGNE a very large set of parameters can be set for a problem, all in form of pointers. However these three classes, `Equation`, `RightHandSide` and `DirichletData` are necessary for every instance of GASCOIGNE. The class `Problem` is derived from the base-class `ProblemDescriptorBase`:

FILE: `GAS/src/Problem/problemdescriptorbase.h`

The equation `Laplace2d` is part of the Gascoigne-Library:

FILE: `GAS/src/Problem/laplace2d.h`

This is explained in detail in Section 3.

The Loop

Then, a `Loop` is specified, the loop is the class controlling what is happening at run-time. and what GASCOIGNE is doing: this is usually a sequence like: initialize, solve a pde, write the solution to the disk, evaluate functionals, compute errors, refine the mesh and start over. In `BasicInit()` we initialize the most basic structure and read data from parameter files. Here we also pass the `ProblemContainer` to the different instances.

Finally, the equation is solved by calling `loop.run("laplace")`, where the keyword `laplace` indicates what problem to solve. Let us now have a closer look at this function which is defined in

FILE: `FIRSTEXAMPLE/src/loop.cc`

First, in `run()`, we define two vectors `u,f` for storing the solution and the right hand side. Then, in the `for`-loop we initialize the problem, solve the problem, write out the solution and refine the mesh. This is very typical for every program run, since for a reliable simulation it is necessary to observe the convergence of the solution on a sequence of meshes. Next, we have a closer look at every step of the inner loop:

```
1    GetMultiLevelSolver()->ReInit(problemlabel);
2    GetMultiLevelSolver()->ReInitVector(u);
3    GetMultiLevelSolver()->ReInitVector(f);
4
5    GetSolverInfos()->GetNLInfo().control().matrixmustbebuild() = 1;
```

The first line initializes the basic ingredients of Gascoigne. In particular, we pass the equation to be solved and all the necessary data to different parts like "Solver", "Discretization". Then, by `GetMultiLevelSolver()->ReInitVector(u)` we initialize the vector on the current mesh. This basically means that we resize the vector to match the number of degrees of freedom (which changes with the mesh-size).

Finally, in the last line we tell Gascoigne to assemble the system matrix. This is necessary whenever the system matrix changes, which is the case if the mesh changes, or if the matrix depends on the solution `u` itself.

FILE: `Template1/src/loop.cc`

```
1    Solve(u,f,"Results/u");
2    GetMeshAgent()->global_refine();
```

By calling `Solve(u,f)` we solve the PDE. The subsequent line refines the mesh.

FILE: `Template1/src/loop.cc`

```
1  string Loop::Solve(VectorInterface& u, VectorInterface& f, string
2  name)
3  {
4    GetMultiLevelSolver()->GetSolver()->Zero(f);
5    GetMultiLevelSolver()->GetSolver()->Rhs(f);
6
7    GetMultiLevelSolver()->GetSolver()->SetBoundaryVector(f);
8    GetMultiLevelSolver()->GetSolver()->SetBoundaryVector(u);
9
10   string status = GetMultiLevelSolver()->
11         Solve(u,f,GetSolverInfos()->GetNLInfo());
12
13   Output(u,name);
14 }
```

Here, we first assemble the right hand side `f` of our problem. Then, we need to initialize Dirichlet boundary values. Finally, we tell the multigrid solver to solve the equation.

In the last line, we write the solution into a file for visualization.

Output of Gascoigne

After starting the code, Gascoigne first prints out some information on the current mesh, solver, Discretization and problem data.

Then, for every iteration of the inner loop the output looks like:

```
...
================== 2 ================ [l,n,c] 5 1089 1024
   0: 9.77e-04
M  1: 7.73e-09 [0.00 0.00]   - 6.26e-08 [0.012] {3}
[u.00002.vtk]
...
```

In the first line the current iteration count (2) , the number of mesh-refinemenet levels (5), the number of mesh nodes (1089) and the number of mesh elements (quads) (1024) is given.

Then, Gascoigne prints control and stastic data for the solution of the algebraic systems. Gascoigne solves every equation with a Newton-method, even linear ones. This is for reasons of simplicity as well as having a defect correction method for a better treatment of roundoff errors. On the left side of the output the convergence history of the newton method is printed. Here we only need one Newton step. Before this step, the residual of the equation was $9.77e-4$, after this one step the residual was reduced to $7.73e-9$. The letter `M` indicates that Gascoigne assembled a new system matrix in this step. The folling two numbers `[0.00 0.00]` indicate the convergance rates of the newton iteration by printing the factor, by which the residual was reduced. We only print the first 2 digits, hence Gascoigne gives a 0. The first number is the reduction rate in the current step, the second number is the average reduction rate over all Newton steps.

On the right side, the convergence history of the linear multigrid solver is printed. Here, the values indicate that in newton step 1 we needed {3} iteration of the multigrid solver. In every of these three steps the error was reduced by a factor of `[0.012]` and after these three steps the residual had the value `6.26e-08`.

Finally Gascoigne prints the name of the output file. You can visualize it by calling `visusimple u.00002.vtk` in the terminal window.

The parameter file

In this directory you can also specify the parameter-file
File: `FIRSTEXAMPLE/src/first.param`

```
1  //Block Loop
2  niter 5
3
4  //Block Mesh
5  dimension 2
```

```
6   gridname square.inp
7   prerefine 3
8
9   //Block BoundaryManager
10  dirichlet 4  1 2 3 4
11
12  dirichletcomp 1 1  0
13  dirichletcomp 2 1  0
14  dirichletcomp 3 1  0
15  dirichletcomp 4 1  0
16
17  //Block Nix
```

This minimal example tells Gascoigne the dimension of the problem, which domain to use, where to apply boundary data.

When starting the program, Gascoigne reads in a parameter file. Here, different parameters controlling the behaviour of the code can be supplied. For instance in `//Block Loop` of this file you can change the values of `niter`, which tells, how many times Gascoigne solves the equation and refine the mesh.

In `//Block Mesh` we supply the mesh file as well as the number of refinement steps realized before solving the first problem. You can change this parameter in `prerefine`. Thoe `//Block BoundaryManager` will be explained in detail in Section 3.2

2 The parameter file

Gascoigne can process a parameter file during runtime. Here, we can set some parameters and options that can be changed at run-time without need for recompilation of the program. For the file to be processed, go to the directory containing the parameter file and call

```
../bin/Test test.param
```

The data in the param-file is organized in *blocks* which are read at different points of the code. A sample parameter file could look like: FILE: `TEST/src/test.param`

```
//Block Settings
month         10
year          2009
name          FirstTestOfGascoigne

//Block Solver
...

//Block Nix
```

Each block has to be initialized by the line

```
//Block <blockname>
```

Note that the slashes in front of the blocks are not treated as a comment here. The parameter file has to end with a dummy-block, e.g. `//Block Nix`.

In Gascoigne, the parameter file is stored in the class `ParamFile`. Reading values from the parameter-file is done using the classes `FileScanner` and `DataFormatHandler`. These classes are declared in the header FILE: `filescanner.h` . Here is a modified FILE: `main.cc` reading values from the parameter file:

FILE: `TEST/src/main.cc`

```
#include <iostream>
#include "stdloop.h"
#include "filescanner.h"
#include "paramfile.h"

using namespace Gascoigne;
using namespace std;

```

```
9  int main(int argc, char**argv)
10 {
11   if (argc<2)
12   { cerr << "call: Test paramfile !!" << endl;
13     abort(); }
14   ParamFile paramfile;
15   paramfile.SetName(argv[1]);
16
17   int month,year;
18   string name;
19   DataFormatHandler DFH;
20   DFH.insert("month",&month);
21   DFH.insert("year", &year, 2009 );
22   DFH.insert("name",&name, "test");
23   FileScanner FS(DFH, &paramfile, "Settings");
24   cout << month << "/" << year << "\t" << name << endl;
25 }
```

To read in one block, we first initialize a DataFormatHandler (line 19). Next, we tell the DataFormatHandler which keywords to look for using the function `insert`. The first argument of the `insert` function is the keyword we are searching, the second argument the variable that its value will be written to. Optionally, we can specify a default value as a third argument which is used if the keyword is not present in the parameter file. Finally, we initialize an object `FileScanner` passing the DataFormatHandler, a parameter file and the block to be read. Note that only this block is read! To read from a different block within one file, we have to define a second `DataFormatHandler`.

3 Description of the problem

FILE: `GAS/src/Problem/problemdescriptorbase.h`

The class `ProblemDescriptor` provides all information necessary to set up an application. This includes all problem data like: equation, boundary data, right hand side, initial condition, etc. The most important function is `ProblemDescriptor::BasicInit()`, where the problem is defined by setting pointers to the describing classes. A minimal example might be:

```
1  void LocalProblemDescriptor::BasicInit(const ParamFile* pf)
2  {
3    GetEquationPointer()          = new LocalEquation;
4    GetRightHandSidePointer()     = new LocalRightHandSide;
5    GetDirichletDataPointer()     = new LocalDirichletData;
6
7    ProblemDescriptorBase::BasicInit(pf);
8   }
```

For an problem to be well defined, at least the equation and the Dirichlet data have to be specified. All possible pointers to be set are listed in
FILE: `GAS/src/Problem/problemdescriptorbase.h`

For the most basic problems GASCOIGNE provides predefined classes, e.g.

```
1 GetEquationPointer()   = new Laplace2d;
2 GetRightHandSidePointer() = new OneRightHanSideData(1);
3 GetDirichletDataPointer() = new ZeroDirichletData;
```

which defines the Laplace problem with homogeneous Dirichlet data and constant right-hand side $f = 1$.

In the following we describe the classes needed to specify the most common ingredients of a problem.

3.1 The right hand side

FILE: `GAS/src/Interface/domainrighthandside.h`

The interface class to specify a function $f : \Omega \to \mathbb{R}$ as right hand side for the problem is `DomainRightHandSide`. Let $f(x, y) = \sin(x)\cos(y)$ be the right hand side:

```
1  class LocalRightHandSide : public DomainRightHandSide
2  {
3    public:
4      int GetNcomp() const { return 1; }
5
6      std::string GetName() const { return "Local Right Hand Side"; }
7
8      double operator()(int c, const Vertex2d& v) const
9      {
10       return sin(v.x()) * cos(v.y());
11     }
12 }
```

The most important function of this class is `double operator()(int c, const Vertex2d& v) const`, here, the value of the right hand side function f at point $v \in \Omega$ is returned. The class `Vertex2d` is explained in FILE: `GAS/src/Common/vertex.h` . For three dimensional problems the operator `double operator()(int c, const Vertex3d& v) const` has to be written.

The second parameter `int c` as well as the function `int GetNcomp() const` is only needed if we solve a system of equations. This is explained in detail in Section 3.5.

A function called `GetName()` is necessary in every class used in the `ProblemDescriptor` to provide a label.

3.2 Boundary Data

In the geometry file, a certain color is assigned to every boundary part of the mesh, see Section 5. You can visualize meshes (inp-files) with the program `visusimple`. Here you can also look at the boundary data, use `Select->Scalar`. In the parameter file we specify all boundary colors where Dirichlet boundary conditions are used.

To assign Dirichlet boundary condition to a certain boundary color, the parameter file has to contain:

```
1 //Block BoundaryManager
2 dirichlet      2  4 8
3 dirichletcomp  4  1 0
4 dirichletcomp  8  1 0
```

This definition lets the boundaries with colors `4` and `8` have Dirichlet condition. The parameter `dirichlet` is given as a vector: the first value is the number of Dirichlet colors, then the colors follow. If only the color `4` is picked for Dirichlet it would look like:

```
1 //Block BoundaryManager
2 dirichlet      1  4
3 dirichletcomp  4  1 0
```

We have to specify a second parameter `dirichletcomp` for every boundary color used with Dirichlet values. Here, we choose the components of the solution where Dirichlet conditions shall be used. These values are again given as vectors: First the color is specified, then the number of solution components that have Dirichlet values on this color. Finally the solution components are listed. For scalar equations with only one component, the last two values are always `1 0`. See Section 3.5 for using Dirichlet colors for systems of partial differential equations with more than one solution component.

Every boundary color, that is not listed as Dirichlet boundary uses *natural boundary conditions* given due to integration by parts. For the Laplace Equation

$$(-\Delta u, \phi)_\Omega + \langle \partial_n u, \phi \rangle_\Gamma = (\nabla u, \nabla \phi)_\Omega,$$

this is the homogeneous Neumann condition $\partial_n u = 0$. How to use non-homogeneous Neumann is explained in Section 3.2.2 and details on Robin-boundary conditions are given in Section 3.2.3.

3.2.1 Dirichlet boundary data

FILE: `src/Problem/diricheltdata.h`

Assume, that the colors `4` and `8` are picked as Dirichlet boundary colors in the parameter file.

The Dirichlet values to be used for the boundaries with colors specified in the parameter file are described in the class `DirichletData`, see FILE: `GAS/src/Interface/dirichletdata.h`

Here we show an example how to define a class `LocalDirichletData` that sets Dirichlet values

```
1
2  #include "dirichletdata.h"
3
4  class LocalDirichletData : public DirichletData
5  {
6    public:
7      std::string GetName() const { return "Local Dirichlet Data;" }
8      void operator()(DoubleVector& b, const Vertex2d& v, int col) const
9      {
10       b.zero();
11       if (col==4) b[0] = 0.0;
12       if (col==8) b[0] = v.x() + v.y();
13     }
14 }
```

This example sets the boundary value to zero on the boundary with color `4` and to the function $g(x, y) = x+y$ on color 8. The values are not returned as in the case of the right hand side, but written in the vector `b`. The value `b[i]` defines the Dirichlet value of the i-th solution

component. For scalar pde's we only set `b[0]`. The parameter `Vertex2d& v` gives the coordinate and the parameter `col` gives the color of the node. In three dimensions, the same operator exists using a `Vertex3d& v` to indicate the coordinate. In the `ProblemDescriptor` we need to set a pointer to this new class: FILE: `TEST/src/problem.h`

```
1  class ProblemDescriptor : public ProblemDescriptorBase
2  {
3     // ...
4     void BasicInit( const ParamFile* paramfile)
5     {
6       // ...
7       GetDirichletDataPointer() =
8         new LocalDirichletData;
9     }
10    // ...
11 };
```

3.2.2 Neumann conditions

For non-homogenous Neumann conditions of the type

$$\langle \partial_n u, \phi \rangle_{\Gamma^N} = \langle g^N, \phi \rangle_{\Gamma^N},$$

we have to add the term (g^N, ϕ) to the right hand side. In Gascoigne, this Neumann right hand side is derived from the class `BoundaryRightHandSide` specified in the FILE: `GAS/src/Interface/boundaryrighthandside.h` . An example for a 2d scalar problem is given by

```
1  #include "boundaryrighthandside.h"
2
3  class LocalBoundaryRightHandSide : public BoundaryRightHandSide
4  {
5    public:
6      int GetNcomp() const { return 1; }
7      string GetName() const { return "Local B-RHS"; }
8
9      double operator()(int c, const Vertex2d& v, const Vertex2d& n,
10                       int color) const
11     {
12       if (color==0) return 1.0;
13       if (color==1) return v.x() * n.x() + v.y() * n.y();
14     }
15
16 };
```

Here, we define Neumann data on two different parts of the boundary with colors `0` and `1`:

$$g_0(x) = 1, \quad g_1(x) = n(x) \cdot x,$$

where $n(x)$ is the outward unit-normal vector in the point x.

Boundary right hand sides need to be specified in the parameter file in `//Block BoundaryManager`. Otherwise these boundary terms are not taken into account: FILE: `TEST/src/test.param`

```
1  //Block BoundaryManager
2  dirichlet      1  4
3  dirichletcomp  4  1 0
4  righthandside  2  0 1
```

The new parameter `righthandside` specifies a vector. Here it tells Gascoigne that `2` boundary colors have Neumann data. These are the colors `0` and `1`. In the `ProblemDescriptor` we need to set a pointer to this new class: FILE: `TEST/src/problem.h`

```
1  class ProblemDescriptor : public ProblemDescriptorBase
2  {
3     // ...
4     void BasicInit( const ParamFile* paramfile)
5     {
6       // ...
7       GetBoundaryRightHandSidePointer() =
8         new LocalBoundaryRightHandSide;
9     }
10    // ...
11 };
```

3.2.3 Robin conditions

Robin boundary data includes very general conditions to be fulfilled on the boundary of the domain. We can have:

$$\langle G(u), \phi \rangle_{\Gamma^R} = 0,$$

where $G(\cdot)$ can be some operator, e.g. $G(u) = \partial_n u + u^2$. Hence, Robin boundary data means that we have an additional equation that is valid on the boundary. We will explain this concept in detail after explaining how to specify equations. See Section 3.3.2.

3.3 Definition of the partial differential equations

FILE: `GAS/src/Interface/equation.h`

Let the pde to be solved be given in the weak formulation

$$a(u)(\phi) = (f, \phi) \quad \forall \phi,$$

where $a(\cdot)(\cdot)$ is a semi-linear form, linear in the second argument. Gascoigne solves every problem with a Newton method (also linear problems). With an initial guess u^0 updates w^k are defined by

$$a'(u^k)(w^k, \phi) = (f, \phi) - a(u^k)(\phi) \quad \forall \phi, \quad u^{k+1} := u^k + w^k.$$

The Jacobi matrix is the matrix of the directional derivatives of $a(\cdot)(\cdot)$ in the point u^k and defined by

$$a'(u)(w, \phi) := \frac{d}{ds} a(u + sw)(\phi) \Big|_{s=0}.$$

For linear problems, it holds

$$a(u)(w, \phi) = a(w)(\phi).$$

To solve, Gascoigne needs to know about the form $a(u^k)(\phi)$ and its derivative, the matrix $a'(u^k)(w^k, \phi)$. Form and matrix are given in the class `Equation`:

```
1  class LocalEquation : public Equation
2  {
3    public:
4
5      int GetNcomp() const;
6      string GetName() const;
7
8      void point(double h, const Vertex2d& v) const;
9      void point(double h, const Vertex3d& v) const;
10
11     void Form(VectorIterator b, const FemFunction& U,
12               TestFunction& N) const;
13     void Matrix(EntryMatrix& A, const FemFunction& U,
14                 const TestFunction& M, const TestFunction& N) const;
15
16 };
```

The first function `GetNcomp()` returns the number of solution components for systems of partial differential equations. For scalar equations, this function returns 1. `GetName()` is a label for the equation. The most important functions are `Form(b,U,N)`, which defines $a(u)(\phi)$ and `Matrix(A,U,M,N)` which gives the derivative $a'(u)(w, \phi)$. The parameters `b` and `A` are the return values, `U` is the last approximation u, `N` is the test-function ϕ and `M` stands for the direction w. The function `point()` is meant to set parameters depending on the mesh size `h` or on the coordinate `v`. `point` is called before each call of `Form()` or `Matrix()` and can be used to set local variables.

For the Laplace equation, the implementation is given by

```
1  void Form(VectorIterator b, const FemFunction& U,
2            const, TestFunction& N) const
3  {
```

```
4     b[0] += U[0].x() * N.x() + U[0].y() * N.y();
5   }
6
7   void Matrix(EntryMatrix& A, const FemFunction& U,
8               const TestFunction& M, const TestFunction& N) const
9   {
10    A(0,0) += M.x() * N.x() + M.y() * N.y();
11  }
```

The class `TestFunction` describes the values and derivatives of a discrete function in a certain point. By `N.m()` the value is accessed, by `N.x()`, `N.y()` and `N.z()` the directional derivatives. The class `FemFunction` is a vector of `TestFunctions` and used for the solution function u. For systems of partial differential equations, the index gives the number of the solution component. For scalar equations, it is always `U[0].m()`.

3.3.1 Nonlinear equations

As example we now consider the nonlinear partial differential equation given by

$$a(u)(\phi) = (\nabla u, \nabla\phi) + (\langle\nabla u, \nabla u\rangle, \phi), \quad \langle x, y\rangle := \sum x_i y_i.$$

The form is given by

```
1   void Form(VectorIterator b, const FemFunction& U,
2             const, TestFunction& N) const
3   {
4     b[0] += U[0].x() * N.x() + U[0].y() * N.y()
5             + (U[0].x() * U[0].x() + U[0].y() * U[0].y()) * N.m();
6   }
```

To define the matrix we first build the derivative:

$$\begin{aligned} a'(u)(w,\phi) &= \frac{d}{ds} a(u+sw)(\phi)|_{s=0} \\ &= \left(\frac{d}{ds}\nabla(u+sw), \nabla\phi\right) + \left(\frac{d}{ds}\langle\nabla(u+sw), \nabla(u+sw)\rangle, \phi\right)|_{s=0} \\ &= (\nabla w, \nabla\phi) + (\langle\nabla(u+sw), \nabla w\rangle, \phi) + (\langle\nabla w, \nabla(u+sw)\rangle, \phi)\,|_{s=0} \\ &= (\nabla w, \nabla\phi) + 2(\langle\nabla u, \nabla w\rangle, \phi). \end{aligned}$$

The matrix function is then given by

```
1   void Matrix(EntryMatrix& A, const FemFunction& U,
2               const TestFunction& M, const TestFunction& N) const
3   {
4     A(0,0) += M.x() * N.x() + M.y() * N.y();
5     A(0,0) += 2.0 * (U[0].x() * M.x() + U[0].y() * M.y) * N.m();
6   }
```

3.3.2 Equations on the boundary & Robin boundary data

Robin boundary data is realized by setting an equation on the boundary. First, we need to tell Gascoigne to include boundary equations. Herefore, section `//Block BoundaryManager` of the parameter file needs to know on what boundary colors we set an equation: File: `TEST/src/test.param`

```
1  //Block BoundaryManager
2  dirichlet      1  4
3  dirichletcomp  4  1 0
4  equation       1  1
```

Here we tell Gascoigne to use the equation on `1` color, the color `1`. The actual equation is of type `BoundaryEquation` defined in the File: `GAS/src/Interface/boundaryequation.h` The boundary equation is declared as the equation: you need to specify the `Form` and the `Matrix`. Major difference is that these two function get the outward unit normal vector and the color of the boundary line in addition. Boundary equations are then defined in the `ProblemDescriptor` by setting: File: `TEST/src/problem.h`

```
1  class ProblemDescriptor : public ProblemDescriptorBase
2  {
3     // ...
4     void BasicInit( const ParamFile* paramfile)
5     {
6       // ...
7       GetBoundaryEquationPointer() =
8         new LocalBoundaryEquation;
9     }
10    // ...
11 };
```

We next give an example of a boundary equation which will realize the Robin data $\partial_n u + u^2 = 0$, see File: `TEST/src/problem.h`

```
1  #include "boundaryequation.h"
2
3
4  class LocalBoundaryEquation : public BoundaryEquation
5  {
6    public:
7
8      int GetNcomp() const {return 1;}
9      string GetName() const { return "Local_B-EQ"; }
10
11     void pointboundary(double h, const Vertex2d& v, const Vertex2d& n)
12 const
13     {}
14
```

```
15      void Form(VectorIterator b, const FemFunction& U,
16                TestFunction& N, int col) const
17      {
18        if (col==1) b[0] += U[0].m() * U[0].m() * N.m();
19      }
20
21      void Matrix(EntryMatrix& A, const FemFunction& U,
22                  const TestFunction& M, const TestFunction& N, int col)
23  const
24      {
25        if (col==1) A(0,0) += 2.0 * U[0].m() * M.m() * N.m();
26      }
27
28  };
```

3.4 Exact Solution and Evaluation of Errors

If an analytic solution is known it can be added to the `ProblemDescriptor` and then be used for evaluating the error in the L^2, H^1 and L^∞ norms. The `operator` in the class `ExactSolution` is used to define the solution function. The parameter `int c` specifies the solution component for systems of pdes.

FILE: `../src/problem.h`

```
1   #include "exactsolution.h"
2
3   class MyExactSolution : public ExactSolution
4   {
5     public:
6       std::string GetName() const {return "My exact solution";}
7       double operator()(int c, const Vertex2d& v)const
8       {
9         return v.x()*v.y();
10      }
11  };
12  class Problem : public ProblemDescriptorBase
13  {
14  ...
15    public:
16      void BasicInit(const ParamFile* pf)
17      {
18        ...
19        GetExactSolutionPointer() = new MyExactSolution;
20      }
21  };
```

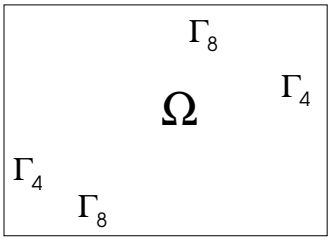
The function `ComputeGlobalErrors` which calculates the L^2, L^∞ and H^1 norm error has to be called in the `Loop` after solving the equation with: FILE: `../src/myloop.cc`

```
1 ...
2     Solve(u,f);
3     ComputeGlobalErrors(u);
4 ...
```

3.5 Systems of partial differential equations

One of the main features of GASCOIGNE is the efficient solution of large systems of partial differential equations. We call the number of equations in a system (and thus the number of solution variables) the *number of components*, denoted by `ncomp` in Gascoigne. Nearly all classes have the number of components as member. As example we discuss the two-equation system in Ω:

$$
\begin{aligned}
(\nabla u_0, \nabla \phi_0) + (u_0 u_1, \phi_0) &= (f_0, \phi_0), \\
(\nabla u_1, \nabla \phi_1) + (u_0 \sin(u_1), \phi_1) &= (f_1, \phi_1), \\
u_0 &= 0 \quad \text{on } \Gamma_4 \cup \Gamma_8, \\
u_1 &= 1 \quad \text{on } \Gamma_4, \\
\partial_n u_1 &= 0 \quad \text{on } \Gamma_8.
\end{aligned}
$$



Several modifications are necessary:

3.5.1 Modification of the right hand side

See 3.1 for comparison. The right hand side has to know the number of equations, in this case:

```
1 int GetNcomp() const { return 2; }
```

The operator for specifying the functions f_0 and f_1 gets the current component as first argument. As an example, let us set $f_0(x, y) = 5$ and $f_1(x, y) = xy$:

```
1 double operator()(int c, const Vertex2d& v) const
2 {
3   if (c==0) return 5;
4   if (c==1) return v.x() * v.y();
5   return 0.0;
6 }
```

3.5.2 Modification of Dirichlet data

For Dirichlet data two separate issues have to be considered: first the declaration in the parameter file has to be modified. Second, the function specifying the Dirichlet data has to be altered. In this example, where the component u_0 has Dirichlet data on both and u_1 just on one boundary color, we set:

```
1 //Block BoundaryManager
2 dirichlet 2    4 8
3 dirichletcomp 4    2     0  1
4 dirichletcomp 8    1     0
```

This is to be read as: there are 2 dirichlet colors, the colors 4 and 8. Dirichlet color 4 is applied to 2 components, components 0 and 1. And so on. Gascoigne uses 0 as index for the first component. The implementation of the boundary data is as follows:

```
1  class LocalDirichletData : public DirichletData
2  {
3    public:
4      std::string GetName() const { return "Local Dirichlet Data;" }
5      void operator()(DoubleVector& b, const Vertex2d& v, int col) const
6      {
7        b.zero();
8        if (col==4) b[0] = 0.0;
9        if (col==4) b[1] = 1.0;
10       if (col==8) b[0] = 0.0;
11     }
12 };
```

Here, the values are sorted into a vector whose components correspond to the components of the solution u.

3.5.3 Modification for the equation, form and matrix

Like the right hand side, the equation needs to know about the number of solution components given in the header file as

```
1 int GetNcomp() const { return 2; }
```

In both functions, `Form` and `Matrix` the last approximate solution `U` is given as a parameter. This parameter is a vector and `U[c]` specifies component `c` of the solution. The modifications for the `Form` are straight-forward, the values for i-th equation are written to the component `b[i]`. The test function `N` is the same for all equations. We give an example:

```
1 void Form(VectorIterator& b, const FemFunction& U,
2           const TestFunction& N) const
3 {
```

```
4     b[0] += U[0].x() * N.x() + U[0].y() * N.y();
5     b[0] += U[0].m() * U[1].m() * N.m();
6
7     b[1] += U[1].x() * N.x() + U[1].y() * N.y();
8     b[1] += U[0].m() * sin(U[1].m()) * N.m();
9   }
```

The `Matrix` works in a simular way. We assemble a local matrix of size `ncomp` times `ncomp`, where `A(i,c)` specifies the derivative of equation `i` with respect to the component `U[c]`. In this example:

$$
\begin{aligned}
A_{00} &= \frac{d}{ds}(\nabla(u_0+sw),\nabla\phi_0) + ((u_0+sw)u_1,\phi_0)\big|_{s=0} &&= (\nabla w,\nabla\phi_0) + (wu_1,\phi_0).\\
A_{01} &= \frac{d}{ds}(\nabla u_0,\nabla\phi_0) + (u_0(u_1+sw),\phi_0)\big|_{s=0} &&= (u_0 w,\phi_0)\\
A_{10} &= \frac{d}{ds}(\nabla u_1,\nabla\phi_1) + ((u_0+sw)\sin(u_1),\phi_1) &&= (w\sin(u_1),\phi_1),\\
A_{11} &= \frac{d}{ds}(\nabla u_1,\nabla\phi_1) + (u_0\sin(u_1+sw),\phi_1) &&= (\nabla w,\nabla\phi_1) + (u_0\cos(u_1)w,\phi_1),
\end{aligned}
$$

and the implementation:

```
1   void Matrix(EntryMatrix& A, const FemFunction& U,
2               const TestFunction& M, const TestFunction& N) const
3   {
4     A(0,0) += M.x() * N.x() + M.y() * N.y();
5     A(0,0) += M.m() * U[1].m() * N.m();
6     A(0,1) += U[0].m() * M.m() * N.m();
7
8     A(1,0) += M.m() * sin(U[0].m()) * N.m();
9     A(1,1) += M.x() * N.x() + M.y() * N.y();
10    A(1,1) += U[0].m() * cos(U[1].m()) * M.m() * N.m();
11  }
```

4 Time dependent problems

Gascoigne provides a simple technique for the modeling of time-dependent partial differential equations like

$$\partial_t u + A(u) = f \text{ in } [0,T] \times \Omega,$$

with a spatial differential operator $A(u)$. Time-discretization is accomplished with Rothe's method by first discretizing in time with simple time-stepping methods. For time-dependent problems, we must provide some additional information:

- We must supply an initial value $u(x,0) = u^0(x)$.
- All problem data like the right hand side, boundary values and differential operator can depend on the time $t \in [0,T]$.
- We must describe the time-stepping scheme and choose a time-step size.

4.1 Time discretization of the heat equation

Here, we examplarily demonstrate the time discretization of the heat equation

$$\partial_t u - \Delta u = f \text{ in } [0,T] \times \Omega,$$

with Dirichlet boundary

$$u = g \text{ on } [0,T] \times \Omega,$$

and the initial value

$$u(\cdot,0) = u^0(\cdot) \text{ on } \Omega.$$

In Gascoigne, time discretization is based on the general θ-scheme

$$\frac{1}{\theta k}(u^m,\phi) + a(u^m,\phi) = (f(t_m),\phi) + \frac{1-\theta}{\theta}(f(t_{m-1}),\phi) + \frac{1}{\theta k}(u^{m-1},\phi) - \frac{1-\theta}{\theta}a(u^{m-1},\phi),$$

where $\theta \in (0,1]$. For $\theta = \frac{1}{2}$, this scheme corresponds to the Crank-Nicolson method, for $\theta = 1$ to the implicit Euler method.

The class `StdTimeLoop` includes a simple loop for time-stepping problems and replaces the `StdLoop`:

```
1  void StdTimeLoop::run(const std::string& problemlabel)
2  {
3  [...]
4    for (_iter=1; _iter<=_niter; _iter++)
5    {
6      [...]
7       _timeinfo.SpecifyScheme(_iter);
8       TimeInfoBroadcast();
9
10      // RHS
11      GetMultiLevelSolver()->GetSolver()->GetGV(f).zero();
12      GetMultiLevelSolver()->GetSolver()->TimeRhsOperator(f,u);
13      GetMultiLevelSolver()->GetSolver()->TimeRhs(1,f);
14
15      _timeinfo.iteration(_iter);
16      TimeInfoBroadcast();
17
18      GetMultiLevelSolver()->GetSolver()->TimeRhs(2,f);
19
20      // SOLVE
21      SolveTimePrimal(u,f);
22
23      [...]
24   }
25 }
```

Every time-step is treated as a stationary problem. The differential operator evaluated at the old time step is known in advance and thus, GASCOIGNE automatically includes it when assembling the right-hand side, if $\theta < 1$. The function `TimeInfoBroadcast()` announces the current time t and the time-step k.

The class `StdTimeSolver` is used instead of `StdSolver`. It has to be invoked by the option `solver instat` in block `//Block MultiLevelSolver` of the parameter file. Gascoigne assembles the time-dependent matrix

$$\frac{1}{\theta k}\mathbf{M} + \mathbf{A},$$

where $\mathbf{M}$ is the mass matrix and $\mathbf{A}$ the stiffness matrix which has to be specified as usual in the class `Equation`.

```
1  void StdTimeSolver::AssembleMatrix(const VectorInterface& gu, double d)
2  {
3    StdSolver::AssembleMatrix(gu,d);
4
5    double scale = d/(_dt*_theta);
6    GetMatrix()->AddMassWithDifferentStencil(GetMassMatrix(),
7                                             GetTimePattern(),scale);
```

```
8 }
```

First, the main part regarding $a(\cdot,\cdot)$ is assembled. Then, the part regarding the time-derivative is added with the proper scaling depending on the time-step k and parameter θ. The parameters θ and the time-step k are provided in the parameter file within the `//Block Loop`

```
1 //Block Loop
2 scheme   Theta
3 theta    0.5
4 dt       0.01
5 neuler   0
```

By the option `neuler` we can tell GASCOIGNE to start with a number of `neuler` initial steps using the implicit Euler scheme. In this way irregular initial data may be smoothened which can be necessary for stability reasons. By setting `neuler 0`, no Euler steps are used and the loop directly starts with the θ-time-stepping scheme.

The function `GetTimePattern()` tells Gascoigne, on which solution components the mass-matrix will act. This `TimePattern` is specified in the Equation by a further function:

```
1 void HeatEquation::SetTimePattern(TimePattern& TP) const
2 {
3   TP(0,0) = 1.;
4 }
```

`TimePattern TP` is a matrix of size `ncomp` times `ncomp`. Typically, this matrix is the diagonal unit-matrix. See Section 4.3 for special choices of the `TimePattern`.

Time dependent problems usually are initial-boundary value problems and require some initial data, like

$$u(x,0) = u^0(x),$$

at time $t = 0$. The initial data is specified via the function `GetInitialConditionPointer()` in the class `ProblemDescriptor`. The initial condition can be set in the same way as the right-hand side:

```
1  class MyInitial : public DomainRightHandSide
2  {
3  public:
4    std::string GetName() const {return "Initial Condition";}
5    int GetNcomp() const { return 1; }
6
7    double operator()(int c, const Vertex2d& v) const
8    {
9      return v.x() * v.y();
10   }
11 };
12
```

```
13 [...]
14
15 class ProblemDescriptor : public ProblemDescriptorBase
16 {
17 public:
18     std::string GetName() const {return "Time-Dependent";}
19     void BasicInit(const ParamFile* pf)
20     {
21       [...]
22       GetInitialConditionPointer() = new MyInitial;
23       ProblemDescriptorBase::BasicInit(pf);
24     }
25 };
```

4.2 Time dependent problem data

The problem data like boundary data and right hand side can depend on time. Let us for example consider the right hand side

$$f(x,t) = \sin(\pi t)(1-x^2)(1-y^2).$$

All data classes like `DomainRightHandSide`, `DirichletData` or `Equation` are derived from the class `Application`. We can access the current time and the time step via the functions `double GetTime() const` and `double GetTimeStep() const`.

```
1  class RHS : public DomainRightHandSide
2  {
3  public:
4    int GetNcomp() const { return 1; }
5    string GetName() const { return "RHS"; }
6
7    double operator()(int c, const Vertex2d& v) const
8    {
9      double t = GetTime();
10     return sin(M_PI * t) * (1.0 - v.x()*v.x()) * (1.0-v.y() * v.y());
11   }
12 };
```

4.3 Non standard time-discretizations

As example for a non-standard time-depending problem we consider the following system of partial differential equations

$$\begin{aligned} \partial_t u_1 + \partial_t u_2 - \Delta u_1 &= f_1 \\ \partial_t u_2 + u_1 - \Delta u_2 &= f_2 \end{aligned}$$

For the implementation of this system, `Matrix`, `Form` and `SetTimePattern` must be given as:

```
1  void EQ::Form(...)
2  {
3    b[0] += U[0].x() * N.x() + U[0].y() * N.y();
4
5    b[1] += U[0].m() * N.m();
6    b[1] += U[1].x() * N.x() + U[1].y() * N.y();
7  }
8
9  void EQ::Matrix(...)
10 {
11   A(0,0) += M.x() * N.x() + M.y() * N.y();
12
13   A(1,0) += M.m() * N.m();
14   A(1,1) += M.x() * N.x() + M.y() * N.y();
15 }
16
17 void EQ::SetTimePattern(TimePattern& TP) const
18 {
19   TP(0,0) = 1.;
20   TP(0,1) = 1.;
21   TP(1,1) = 1.;
22 }
```

5 Mesh handling

Mesh handling includes definition of the domain Ω, refinement of meshes, managing of curved boundaries, input and output of meshes.

The mesh to be used in the calculation is indicated in the parameter file. In the Section `//Block Mesh` the mesh to be read is specified:

```
1 //Block Mesh
2
3 dimension   2
4 gridname    mesh.inp
5 prerefine   3
```

Gascoigne distinguishes two types of meshes: `inp`-files are coarse meshes describing the domain Ω. They are user-specified and the file format is explained in Section 5.1. These meshes are usually as coarse as possible. Refinement is applied during run-time. The parameter `prerefine` declares global refinements to be executed before the first solution cycle. It is also possible to write out and read in locally refines meshes. These meshes are given in the `gup`-format:

```
1 //Block Mesh
2
3 dimension   2
4 gridname    mesh.gup
5 prerefine   0
```

During run-time all mesh handling like I/O and refinement is done by the class `MeshAgent`. The `MeshAgent` is a member of the class `Loop` and is created in the `Loop::BasicInit`. The `MeshAgent` takes care of the mesh refinement hierarchy `HierarchicalMesh`, it creates the sequence of multigrid meshes `GascoigneMultigridMesh` and it finally provides the meshes on every level used for the computation, the `GascoigneMeshes`. See the files with the same names in FILE: `GAS/src/Mesh/*` .

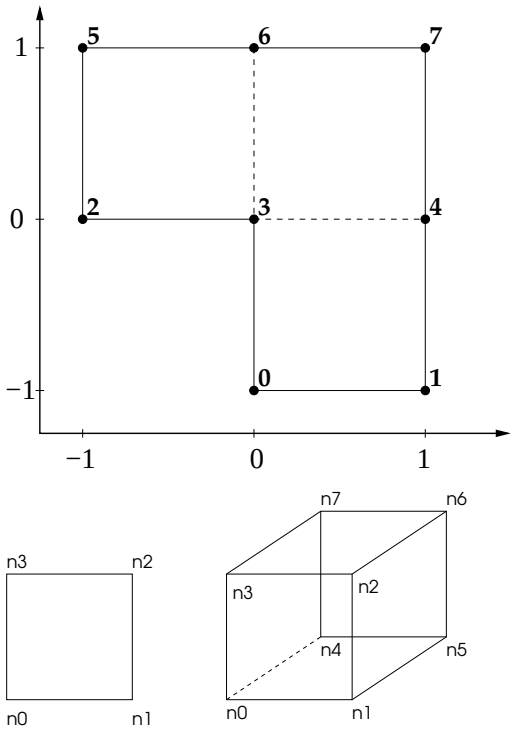
If domains with curved boundaries are used, the shape definition is also done in the class `MeshAgent`.

5.1 Definition of coarse meshes

Coarse meshes are specified in the `inp`-format. We indicate the nodes of the mesh, the elements (that are quadrilaterals in two and hexahedrals in three dimensions) and finally

the boundary elements. In two dimensions, these are the lines of quadrilaterals that touch the boundary, in three dimensions we lists the quadrilaterals that are boundaries of the domain and of elements. We never list interior lines (or quadrilaterals in three dimensions), as these are not needed for the calculation. The following example is the coarse mesh for the L-shaped domain:

```
8 11 0 0 0
0 0 -1 0
1 1 -1 0
2 -1 0 0
3 0 0 0
4 1 0 0
5 -1 1 0
6 0 1 0
7 1 1 0
0 0 quad 0 1 4 3
1 0 quad 2 3 6 5
2 0 quad 3 4 7 6
0 4 line 0 1
1 4 line 1 4
2 4 line 4 7
3 4 line 0 3
4 2 line 2 3
5 2 line 2 5
6 8 line 5 6
7 8 line 6 7
```



The file format is very simple: in the row, first the number of nodes in the coarse mesh is given, followed by the sum of mesh elements plus elements on the boundary. The last three values are always zero. This example has 8 nodes, 3 quads and 8 lines on the boundary.

After the header line all nodes of the mesh are declared in the format:

```
n x y z
```

where `n` is the index of the node (starting with zero) and `x y z` are the coordinates. We always have to give all three coordinates (also in two dimensions).

The nodes are followed by the mesh elements. In two dimensions the format is:

```
n 0 quad n0 n1 n2 n3
```

where `n` is a consecutive index starting with 0. The second parameter is not used, thus always 0. The mesh element, always `quad` is indicated by the third entry. The last four values give the nodes describing the quadrilateral in counter-clockwise sense.

In three spatial dimension, the format is:

```
n 0 hex n0 n1 n2 n3 n4 n5 n6 n7
```

Here, first the nodes `n0` to `n3` in the front are given in counter-clock wise sense followed by the nodes in the back.

Finally, all the boundary elements are given. In two dimensional meshes, these are all lines, where both nodes are on the boundary. In three dimensions the boundary elements are all quads with four nodes on the boundary. The format is:

```
n c line n0 n1
```

in two dimensions and

```
n c quad n0 n1 n2 n3
```

in three dimensions. `n` is again an iteration index. This index starts with 0. The value `c` defines a *color* for the boundary element. These colors are necessary to distinguish between different types of boundary conditions, see Sections 3.2.1 and 3.2.2. In two dimensions, the order of the two nodes `n0 n1` is arbitrary. In three dimensions however the four nodes have to be given in a counter clock-wise sense seen from the outside of the domain! This is necessary to get a unique definition of the normal vectors on the boundary. Thus, for the hex shown in Picture above, the `inp`-file would be:

```
8 7 0 0 0
0 0 0 0
1 1 0 0
2 1 1 0
3 0 1 0
4 0 0 1
5 1 0 1
6 1 1 1
7 0 1 1
```

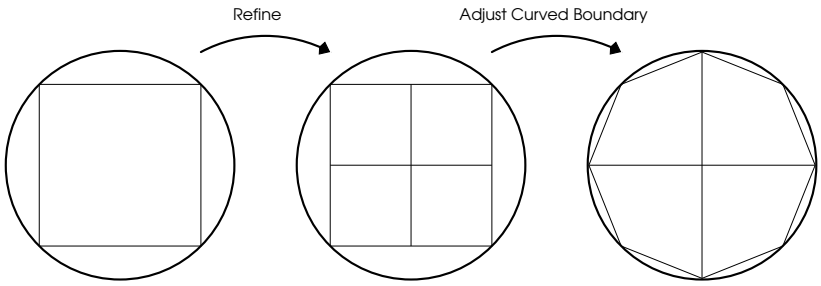
```
0 0 hex 0 1 2 3 4 5 6 7
0 4 quad 0 1 2 3
1 4 quad 1 5 6 2
2 4 quad 3 2 6 7
3 4 quad 4 0 3 7
4 4 quad 5 4 7 6
5 4 quad 4 5 1 0
```

5.2 Curved boundaries

A part of the boundary can be assigned a curved boundary function. The idea is easy: Let the boundary with color `col` be a curved boundary. We specify a function $\mathbb{R}^d \to \mathbb{R}$ which has the desired curved boundary as zero-iso-contour. For instance if the boundary Γ of the domain should have the shape of a circle with radius r and midpoint (m_x, m_y), one possible function f_{circle} would be

$$f_{\text{circle}}(x, y) = (x - m_x)^2 + (y - m_y)^2 - r^2.$$

Whenever an element with boundary nodes on a line with color `col` is refined, the newly generated nodes on the boundary line are then *pulled* onto the zero-iso-contour of the boundary function. This is done by solving for the root of f_{circle} with Newton's method.



In Gascoigne boundary functions are described by the class `BoundaryFunction<DIM>` in FILE: `GAS/src/Mesh/BoundaryFunction` . The circle is given by:

```
1  class Circle : public BoundaryFunction<2>
2  {
3      double __r,__mx,__my;
4    public:
5      Circle(double r, double mx, double my) : __r(r), __mx(mx), __my(my)
6      { }
7
8      double operator()(const Vertex2d& v) const
9      {
10       return (v.x()-__mx)*(v.x()-__mx)+(v.y()-__my)*(v.y()-__my)-__r*__r;
11     }
12 }
```

The boundary function object has to be passed to the `MeshAgent`. For this, a derived mesh agent class has to be specified and created in the `Loop`. The following minimal steps have to be taken:

```
1  class LocalMeshAgent : public MeshAgent
2  {
3      Circle _circle;
4    public:
5      LocalMeshAgent() : MeshAgent()
6      {
7        AddShape(4, &_circle);
8      }
9  };
10
11 class LocalLoop : public StdLoop
12 {
13   public:
14
15     void BasicInit(const ParamFile* paramfile,
16                    const ProblemContainer* PC,
17                    const FunctionalContainer* FC=NULL)
18     {
```

```
19        GetMeshAgentPointer() = new LocalMeshAgent();
20        StdLoop::BasicInit(paramfile, PC, FC);
21      }
22  };
```

In this example, the boundary with color `4` is used as curved boundary.

5.3 Adaptive refinement of meshes

For local refinement, the `MeshAgent` has the function `refine_nodes(nodes)`. The `nvector<int> nodes` contains indices of mesh nodes to be refined. If a node is to be refined, all elements having this node as a corner node will be refined.

FILE: `../src/myloop.cc`

```
1   for (_iter=1;_iter<_niter;++_iter)
2   {
3  ...
4       Solve(u,f);
5  ...
6       // Estimate Error
7       nvector<int> ref;
8       // pick nodes for refinement in ref
9       GetMeshAgent()->refine_nodes(ref);
10  }
```

5.3.1 Estimating the Energy-Error

The class `DwrQ1Q2` contains a function for estimation of an energy-error. It can be used in the following way:

FILE: `../src/myloop.cc`

```
1  #include "dwrq1q2.h"
2  ...
3   for (_iter=1;_iter<_niter;++_iter)
4   {
5  ...
6       Solve(u,f);
7  ...
8       // Estimate Error
9       DoubleVector eta;
10      DwrQ1Q2 dwr(*GetMultiLevelSolver()->GetSolver());
11      dwr.EstimatorEnergy(eta, f,u);
12 ...
13  }
```

Now, for each node of the mesh the vector η contains a measure for the local contribution to the total energy error.

5.3.2 Picking Nodes for Refinement

There are several methods for picking nodes for refinement. Common to all is to choose nodes `n` with large estimator values `eta[n]`. The class `MalteAdaptor` uses a global optimization scheme for picking nodes:

FILE: `../src/myloop.cc`

```
#include "malteadaptor.h''
...
  for (_iter=1;_iter<_niter;++_iter)
  {
...
    Solve(u,f);
...
    // Estimate Error
    DoubleVector eta;
    DwrQ1Q2 dwr(GetMultiLevelSolver()->GetSolver());
    dwr.EstimatorEnergy(eta, f,u);

    // Pick Elements
    nvector<int> refnodes;
    MalteAdaptor A(_paramfile,eta);
    A.refine(refnodes);

    // Refine
    GetMeshAgent()->refine_nodes(refnodes);
  }
```

6 Flow problems and stabilization

In order to discretize the Stokes or Navier-Stokes systems one can either use *inf-sup-* stable finite element pairs for pressure and velocity which fulfill an inf-sup condition or include stabilization terms in the equations that ensure uniqueness of the solution and stability. Gascoigne always uses equal-order elements which are not inf-sup stable, in combination with different pressure stabilization techniques. If in the Navier-Stokes case the convective term is dominating (high *Reynolds* number) further stabilization is required.

6.1 The function `point`

To set element-wise constants in Gascoigne, the function `point` is needed:

```
1  void Equation::
2       point(double h, const FemFunction& U, const Vertex2d& v) const
3  {
4
5  }
```

This function is called before every call of `Matrix` and `Form` and here the local mesh-size `h` and the coordinates of the node `v` are given as a parameter. That way one can specify certain criteria for the equation based on location or local mesh width.

To stabilize a given equation or system of equations (like Stokes and Navier-Stokes) one usually modifies it by adding certain stabilizing terms. These terms often depend on the local mesh width and have therefore to be defined in this function.

Since the function `point` is defined as `const` no class members can be altered. Thus, every `variable` which shall be modified here has to be defined as `mutable` in the header file:

```
1  mutable double variable;
```

6.2 Stabilization of the Stokes system

The Stokes equations are given by

$$\nabla \cdot v = 0, \quad -\nu \Delta v + \nabla p = f \text{ in } \Omega.$$

The simplest stabilization technique is the addition of an artificial diffusion. Stability is ensured by modifying the divergence equation to

$$(\nabla \cdot v, \xi) + \sum_{K \in \Omega_h} \alpha_K (\nabla p, \nabla \xi)_K,$$

where α_K is an element-wise constant that can be chosen as

$$\alpha_K = \alpha_0 \frac{h_K^2}{6\nu}.$$

To get the local cell size h_K of an element K, we use the function `point` which is called before each call of `Form` and `Matrix`. In `point` we can calculate the value α_K and save it to a local variable:

```
1  void Stokes::
2       point(double h, const FemFunction& U, const Vertex2d& v) const
3  {
4    __alphaK = __alpha0 * h*h/6.0/__nu;
5  }
```

The parameter `__alpha0` can be read from the parameter file. The variable `__alphaK` has to be defined as `mutable` in the header file:

```
1  mutable double __alphaK;
```

6.3 Stabilization of convective terms

The Navier-Stokes system

$$\nabla \cdot v = 0, \quad v \cdot \nabla v - \nu \Delta v + \nabla p = f \text{ in } \Omega,$$

needs additional stabilization of the convective term if the viscosity is low. One popular approach consists of adding diffusion in streamline direction (Streamline Upwind Petrov-Galerkin (SUPG) method). In combination with the artificial diffusion ansatz for pressure stabilization, the full stabilized system reads

$$(\nabla \cdot v, \xi) + \sum_K (\alpha_K \nabla p, \nabla \xi)_K = 0,$$
$$(v \cdot \nabla v, \phi) + \nu(\nabla v, \nabla \phi) - (p, \cdot \nabla \phi) + \sum_K (\delta_K v \cdot \nabla v, v \cdot \nabla \phi)_K = (f, \phi).$$

The parameters α_K and δ_K are set to

$$\alpha_K = \alpha_0 \left(\frac{6\nu}{h_K^2} + \frac{\|v\|_{K,\infty}}{h_K} \right)^{-1}, \quad \delta_K = \delta_0 \left(\frac{6\nu}{h_K^2} + \frac{\|v\|_{K,\infty}}{h_K} \right)^{-1},$$

where $\|v\|_{K,\infty}$ is the norm of the local velocity vector. The parameters might be chosen as

$$\alpha_0 = \delta_0 \approx \frac{1}{2}.$$

For programming the `Matrix` it is usually not necessary to implement all derivatives. It should be sufficient to have:

```
1  void Equation::Matrix(...)
2  {
3
4  ...
5    // derivative of the convection stabilization terms
6    A(1,1) += __delta * (U[1].m()*M.x() + U[2].m() * M.y())
7                      * (U[1].m()*N.x() + U[2].m() * N.y());
8    A(2,2) += __delta * (U[1].m()*M.x() + U[2].m() * M.y())
9                      * (U[1].m()*N.x() + U[2].m() * N.y());
10 ...
11 }
```

6.4 LPS Stabilization

Another way to stabilize these equations is the so called Local Projection Stabilization (LPS). The idea is to suppress local fluctuations by adding the difference between the function itself and its projection onto a coarser grid.

We use the fluctuation operator

$$\pi = id - i_{2h}$$

where i_{2h} is a projection to the coarser grid. The LPS stabilized formulation for the Stokes system reads

$$(\nabla \cdot v, \xi) + \sum_{K \in \Omega_h} \alpha_K (\nabla \pi p, \nabla \pi \xi)_K = 0$$

and for the Navier-Stokes systems

$$(\nabla \cdot v, \xi) + \sum_K (\alpha_K \nabla \pi p, \nabla \pi \xi)_K = 0,$$

$$(v \cdot \nabla v, \phi) + \nu(\nabla v, \nabla \phi) - (p, \cdot \nabla \phi) + \sum_K (\delta_K v \cdot \nabla \pi v, v \cdot \nabla \pi \phi)_K = (f, \phi).$$

α_K and δ_K can be taken from above.

To implement the projection in Gascoigne, one has to use the class `LpsEquation` instead of `Equation`. It provides the additional functions

```
void lpspoint(double h, const FemFunction& U,
              const Vertex2d& v) const { ... }
void StabForm(VectorIterator b, const FemFunction& U,
              const FemFunction& UP, const TestFunction& Np)
              const { ... }
void StabMatrix(EntryMatrix& A, const FemFunction& U,
                const TestFunction& Np, const TestFunction& Mp)
                const { ... }
```

which have to be used for the stabilizing terms.

The variables `UP`, `Np` and `Mp` stand for the fluctuation operator π applied to `N` and `M`. GASCOIGNE provides these fluctuation operators if the discretization specified by the keyword `discname` in the parameterfile is chosen `Q1` or `Q2Lps`.

Similar as in the case of the SUPG method, it should be sufficient to implement the following terms in the `StabMatrix`

```
void LpsEquation::StabMatrix(...)
{

...
  // derivative of the convection stabilization terms
  A(1,1) += __delta * (U[1].m()*Mp.x() + U[2].m() * Mp.y())
                    * (U[1].m()*Np.x() + U[2].m() * Np.y());
  A(2,2) += __delta * (U[1].m()*Mp.x() + U[2].m() * Mp.y())
                    * (U[1].m()*Np.x() + U[2].m() * Np.y());
...
}
```