

FUTUREYE USER MANUAL 3.0

YUEMING LIU

CONTENTS

Introduction	2
Functions	4
2.1 Constant functions	6
2.2 Single variable functions	6
2.3 Evaluating Functions	7
2.4 Combinations of Functions	7
2.5 Composition of Functions	8
2.6 Derivative of Functions	8
2.7 Multivariable Functions	9
2.8 Vector and Vector Valued Functions	10
2.9 Class FMath	13
2.10 User Defined Functions	16
2.11 Speedup Evaluation of Functions	19
Basic Finite Element Components	23
3.1 Mesh, Element and Node Classes	23
3.2 Shape Functions and Mappings	33
3.3 Degrees of Freedom	43
3.4 Weak Form Classes	45
3.5 Assembly Process	57
3.6 Solving Linear System	59
Whole Examples	65
4.1 Example 1: Elliptic Problem	65
4.2 Example 2 Plane Stress	69
4.3 Example 3 Plane Stain	70
4.4 Example 4 Stokes: Cylinders	70
4.5 Example 5 Stokes: Box	72
4.6 Example 6 Stokes: U Shape Channel	72
4.7 Example 7 Convection Diffusion	73
4.8 Example 8 Wave	73
4.9 Example 9 3D Elliptic	73
4.10 Example 10 Navier-Stokes Equation	74
Mesh Refinement	78
Handling Exceptions	81
Inverse Problems	81
7.1 Useful Tools	81
7.2 Optical Inverse Problems	85

8scalaFEM		94
8.1	Functions	95
8.2	Meshes and Elements	99
8.3	Weak Form	100
8.4	Matrix and Vector Operations	102

1 Introduction

FuturEye is a Java based Finite Element Method (FEM) Toolkit, providing concise, natural and easy understanding programming interfaces for users who wish to develop researching and/or engineering FEM algorithms for Forward and/or Inverse Problems.

The essential components of FEM are abstracted out, such as nodes, elements, meshes, degrees of freedom and shape functions etc. The data and operations of these classes are encapsulated together properly. The classes that different from other existing object-oriented FEM softwares or libraries are function classes. The behavior of the function classes in FuturEye is very similar to that in mathematical context. For example algebra of functions, function derivatives and composition of functions. Especially in FEM environment, shape functions, Jacobin of coordinate transforms and numerical integration are all based on the function classes. This feature leads to a more close integration between theory and computer implementation.

Traditionally, the finite element analysis software consists of three stages:

- preprocessing, e.g. mesh generation
- problem solution
- postprocessing, e.g. visualization of models and results

Nowadays, there are many pre and post processing softwares available. For example, Gridgen(NASA), GiD, Techplot etc. There is a long list of mesh generators: <http://www-users.informatik.rwth-aachen.de/roberts/software.html>. FuturEye mainly focus on problem solution stage. Interfaces to several pre and post processing softwares are provided already. Because of easily understanding geometry classes in FuturEye, it is easy to write additional interfaces by end users.

FuturEye is designed to solve 1D,2D and 3D partial differential equations(PDE) with scalar and/or vector valued problem unknown functions. The start point of development of this toolkit is solving inverse problems of PDE. In order to solve inverse problems, usually some forward problems must be solved first and many exterior data manipulations should be performed during the solving processes. There are many classes defined for those data operations. However, the data processes are complicated in actual applications, we can not write down all the tools for manipulating data. The design of the basic classes allows operations to all aspect of data structure directly or indirectly in an easily understanding way. This is important for users who need write their own operations or algorithms on the bases of data structure in FuturEye. Some other existing FEM softwares or libraries may over encapsulate the data and operations for such inverse applications.

The toolkit can be used for various purposes:

- Teaching: The feature of close integration between FEM theory and computer implementation of this toolkit helps a student to understand basic FEM concepts, e.g. shape functions, Jacobin and assembly process.

- **Researching:** Helps researchers quickly develop and test their models, experiment data and algorithms. e.g. new equations, new finite elements and new solution methods.
- **Engineering:** The performance and efficiency may be unsatisfied for real applications, if an finite element class defined in a mathematical manner without optimization. Thanks to the interface conception in Java, we can implement the same interface in many different ways, thus a carefully optimized finite element class can be used in applications with huge number of elements.

The traditional programming language of FEM are Fortran(Fortran77) and C, which support procedural programming. In 1990s, there were many object-oriented design of FEM libraries or softwares welling up, especially in structural engineering fields. The limitations of the traditional programming languages are:

- The global access to data structure and/or too many function parameters decrease the flexibility of the system.
- Clear modularity in language level is hard. A particular algorithm usually associate to many data structure and functions. It must be documented clearly for end users.
- Reuse of existing codes to adapt them for new models, new algorithms or slightly different applications is difficult or impossible. Sometimes, a high degree of knowledge of the codes is necessary.

Object-oriented programming doesn't has the limitations listed above. The concepts of class, object, inheritance, polymorphism and encapsulation enables developers build big systems with clear modules and high reusability. It usually provides better data structure and management by the encapsulation of data. The capabilities of adding new models and algorithms can be easily achieved by inheriting from the existing classes in the systems.

Comparing to C++, Java is not widely used in FEM softwares. This is because that Java is very young (from 1995) and Java byte code translation into native instructions leads to a slower operation at the beginning of it's birth. However, Just-In-Time compiler (JIT) and HotSpot Java Virtual Machine (JVM) can significantly speed up the execution of Java applications. For FEM computing, acceptable computational efficiency of the Java code can be achieved.

Java is a simple object-oriented programming language. It's syntax is easier than C++. It has rich collection of libraries implementing various application programming interfaces. Java has built-in garbage collector (GC) relieving the loads of deleting objects by developers selves just like in C++ which is a chief criminal of runtime errors and memory leaks. The portability of Java is excellent. JVMs are developed for all major computer systems. Another advantage of java is that there are very good XDEs(eXtended Development Environments) for developers to use, e.g. Eclipse, NetBeans etc. These XDEs have excellent abilities of code editing, navigating, organization and refactor, especially for Java.

An overview of the process of solving PDEs in FuturEye is given in Fig. 1. There are three classes/interfaces **Function**, **Mesh** and **Vector** exist in the whole process from PDE problems to their solution. This is because they are the fundamental part of solving PDE problems with FEM in mathematical context. Specifically, the domain of interest of a problem is represented by object of **Mesh**; the parameters(coefficients, right hand side, etc.) of problems are represented by objects of **Function** and the discrete solutions of problems are represented by objects of **Vector**. The three basic classes/interfaces are also basic and most useful part in solving inverse problems. They may be used frequently in all stages through out the process of solving PDE forward/inverse problems. Detailed

discussion of how to use them and other classes/interfaces in Fig. 1 will be addressed in the following sections.

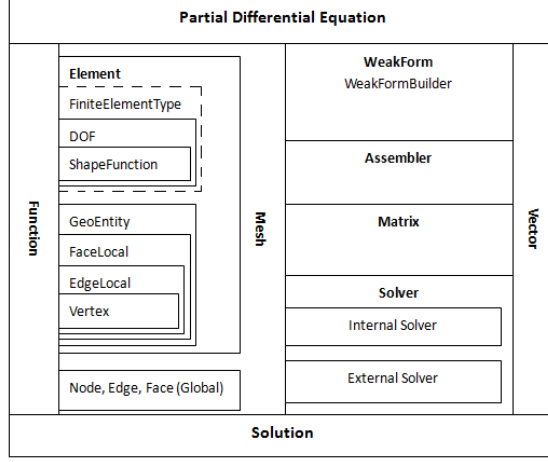


Figure 1: Overview of the process of solving PDEs in FuturEye

Comparing to other existing object-oriented FEM softwares/packages, for example Deal.II, OOFEM, OFELI and LibMesh, abstracted concepts of classes in FuturEye are most similar to the concepts in finite element method described in mathematical context. For example, there are classes corresponding to the concepts of weak forms, shape functions and assemble process. Especially, algebra of functions, function derivatives and composition of functions are distinguishing features in FuturEye. Fig. 2 gives a simple comparison between FuturEye and other several selected FEM softwares/packages written by C++ with object-oriented designs. It shows the basic abstraction of object concepts and inherit structures. The authors have tried their best in presenting the contents of these referenced softwares/packages and hope that all aspects will find themselves correctly quoted.

2 Functions

In mathematical context, function (especially shape function) is one of basic and important concepts of finite element method. How to design an easy-to-use counterpart of function concepts with a computer language is still a challenging problem. Although, there are many commercial softwares that support pure symbol computations, it is still hard to find good and complete packages to solve PDE with FEM based on pure symbol computations. In fact, pure symbol computations in FEM is not a good choice. The alternative solution that FuturEye provides is a good tradeoff between pure symbol computations and hand-coding of function.

The behavior of the function classes in FuturEye is very similar to that in mathematical context. For example algebra of functions, function derivatives and composition of functions. These features will be discussed in subsequent sections.

The java interface **Function** provides an approach to define any mathematical functions. A function is composed by an expression and variables in it.

Mathematical Concept		FuturEye	Deal.II	OOFEM	OFELI	LibMesh
Domain	Mesh	Mesh	Triangulation	Domain	Domain Mesh, Gird	Mesh
	Geometric Entities: Vertex, Edge, Face, Volume	Classes implement GeoEntity	GeometryInfo<dim>	BasicGeometry FEICellGeometry	Classes inherit Figure Element	Classes inherit Elem
Finite Element	Node	Node	-	Node	Node	Node
	Element	Element	Classes inherit FiniteElement	Classes inherit Element	Element	Classes inherit FEBase
	Element Type	Classes implement FiniteElementType	-	Classes inherit FEInterpolation instead	-	FEType
	Degree of Freedom(DOF)	DOF Element	DOFAccessor DOFHandler	DOF DOFManager (e.g. Node, Element)	Node Element	DOFObject (e.g. Node, Elem) DOFMap
	Shape Function	Classes implement ShapeFunction	Member function of FEValuesBase<dim, spacedim>	Member function of Element	Classes inherit FEShape	Member functions of FE and FEInterface
	Coordinate Transformation Jacobian Jacobian Matrix	CoordinateTransform	Classes inherit Mapping<...>	Classes inherit FEInterpolation	FEShape	Member functions of FEBase
	Vector Valued Element Shape Function	VectorShapeFunction	FESystem	DOFManager (e.g. Node, Element)	-	FEBase
Weak Form		WeakFormBuilder Classes implement WeakForm	Implement by hand	-	Classes inherit Equation	Implement by hand
Assemble Process		Classes implement Assembler	MeshWorker framework	-	Implement by hand	Implement by hand
Quadrature		Class FOIntegrate	Classes inherit Quadrature	GaussPoint Classes inherit IntegrationRule	Classes inherit FEShape	FEBase
Matrix, Vector		Classes implement Matrix, AlgebraMatrix, BlockMatrix, Vector, AlgebraVector and BlockVector	Linear Algebra Classes Module: Full, Block, Sparse Matrix and Vector etc.	Classes inherit Matrix and SparseMatrix FloatArray	Classes inherit Matrix<T_> Vect	Classes inherit SparseMatrix<T> DenseMatrix<T> DenseVector<T>
Solver		Internal solver classes External solver classes	Linear solver classes Preconditioners and Relaxation Operators SLEPcWrappers PETScWrappers TrilinosWrappers	Classes inherit NumericalMethod	Eigen, Iter, Prec, Precond, Reconstruction	Classes inherit Solver LinearSolver NonlinearSolver TimeSolver EigenSolver DiffSolver
Functions	Basic Functions	Classes implement Function and VectorFunction	Classes inherit Function<dim>	Classes inherit EnrichmentFunction	Funct	-
	Function Algebra, Combination, Composition, Derivative	Classes implement Function Class FMath	-	-	-	-

Figure 2: Comparison of basic concepts

2.1 Constant functions

The simplest function in FuturEye is constant function, which is a predefined class **FC**. The following code segment defines a constant function $f=2.0$:

```
1  Function c2 = new FC(2.0);
```

There are several predefined constant function objects, which are often used in real applications. Considering memory consumption and speed, these constant functions are define in **FMath** as static objects. One can use "import static" to omit the class name **FMath**:

```
1  import static edu.uta.futureeye.function.operator.FMath.*;
2  ...
3      Function c0 = C0; //f=0.0
4      Function c1 = C1; //f=1.0
5      Function cm1 = Cm1; //f=-1.0
6      Function PI = PI; //f=Math.PI
7      Function E = E; //f=Math.E
8      Function aConst = C(80) //f=80
```

For the constants which are used frequently in your code and which are not predefined static objects in **FMath**, you can use the static function **C(double v)** to get the instance of the constant function. The object will be cached in an inner map, when you call **C(double v)** again for the same constant function, it will be retrieved from the map. By this way, operations of creating a new object of the same constant each time in a loop can be easily avoided. For example:

```
1      for(int i=1; i<=10; i++) {
2          Function c = C(i%3);
3          //do something...
4      }
```

2.2 Single variable functions

Class **FX** represents the identity function $f(x) = x$. The variable name of the identity function can be different when you create an instance. The predefined static object **FMath.X** is an static instance of **FX** which represents identity function $f(x) = x$ with variable name x . It is a quicker and better way of using this static object to express complicated functions than creating an new object of class **FX**. Because this object is immutable in most application cases duplicate objects can lead to memory waste. The following table is all the predefined static objects of **FX** with different variable names.

Static Objects	Functions with different variable names
FMath.X	$f(x) = x$
FMath.Y	$f(y) = y$
FMath.Z	$f(z) = z$
FMath.R	$f(r) = r$
FMath.S	$f(s) = s$
FMath.T	$f(t) = s$

Again, you can use "import static edu.uta.futureeye.function.operator.FMath.*" at the beginning of your class file to omit prefix "FMath" in user code. These predefined static objects are useful when you creating new functions through combinations of functions(see

Section ??). The suggested way to defined an identity function with user specified variable name that not listed above is to using an static member

```
1 class Cls {
2     static Function fu = new FX("u");
3     ...
4 }
```

There are some basic classes represent single variable functions defined in package **edu.uta.futureye.function.basic**. The following table is a list of these classes:

Classes	Math Functions
FX	$f(x) = x$
FAx	$f(x) = ax$
FAxpb	$f(x) = ax + b$
FPolynomial1D	$f(x) = \sum_{i=0}^n a_i x^i$
FLinear1D	$f(x) = \frac{y_2 - y_1}{x_2 - x_1}(x - x_1) + y_1, x_1 \neq x_2$

2.3 Evaluating Functions

To evaluate a function, the value of variable must be given first. Class **Variable** represents function variable(s).

```
1 Variable v = new Variable(2.0);
2 //f(x)=2x + 3
3 Function f = new FAxpb(2.0,3.0);
4 System.out.println("f(x)="+f);
5 System.out.println("f("+v.get()+")="+f.value(v));
```

The output is:

```
1 f(x)=2.0*x+3.0
2 f(2.0)=7.0
```

The constructor *new Variable(2.0)* will construct a variable with default variable name *x*. Call *new Variable("r",2.0)* can create a variable with name *x*.

2.4 Combinations of Functions

We just demonstrated how to form function $f(x) = 2.0 * x + 3.0$, but this is not the only way. Through algebra of functions, we can also get the same function:

```
1 import static edu.uta.futureye.function.operator.FMath.*;
2 ...
3 //f(x)=2x+3
4 Function f = X.M(2.0).A(3.0);
5 System.out.println("f(x)="+f);
```

The output is:

```
1 f(x)=x * 2.0 + 3.0
```

Every function inherited from class **AbstractFunction** will have the basic algebra operations. These operations defined in **AbstractFunction** shown in the following table:

Operations defined in AbstractFunction	Operations
Function A (Function g)	$f(x) + g(x)$
Function S (Function g)	$f(x) - g(x)$
Function M (Function g)	$f(x) * g(x)$
Function D (Function g)	$f(x)/g(x)$

Unlike that in C++, Smalltalk, Ruby and Scala etc. there are operator overloading, Java does not have it. However, we can use A,S,M and D to represent +,-,* and /.

2.5 Composition of Functions

Through composition, we can get new functions. This is useful in case of defining finite element shape functions. The best way to describe how to use the composition of functions is by example. The following code segment gives a demo

```

1 import static edu.uta.futureeye.function.operator.FMath.*;
2 ...
3 Function fOut = X.M(X).S(C1); //Algebra of Functions
4 System.out.println(fOut); //fOut(x) = x*x - 1.0
5 Function fIn = R.M(R).A(C1); //Algebra of Functions
6 System.out.println(fIn); //fIn(r) = r*r + 1.0
7
8 //We need a map: x = r*r + 1.0
9 Map<String,Function> map = new HashMap<String,Function>();
10 map.put("x", fIn); //x = fIn(r) = r*r + 1.0
11 Function fComp = fOut.compose(map);
12 System.out.println(fComp); //x(r) * x(r) - 1.0
13
14 //evaluate with respect to x
15 System.out.println(fComp.value(new Variable("x",2.0)));
16 //evaluate with respect to r
17 System.out.println(fComp.value(new Variable("r",2.0)));

```

The output is:

```

1 x * x - 1.0
2 r * r + 1.0
3 x(r) * x(r) - 1.0
4 3.0
5 24.0

```

2.6 Derivative of Functions

The derivative of a function $f(x)$ is also a function. Continue with the above code segment:

```

1 System.out.println(fComp._d("x")); //d(fComp)/dx
2 Function dr = fComp._d("r"); //d(fComp)/dr
3 System.out.println(dr);
4 System.out.println(dr.value(new Variable("r",2.0)));

```

The output is:

```

1 x + x
2 (x(r) * x(r) - 1.0) * (r + r)
3 96.0

```


Because function $fComp$ is a composite function, we can compute the derivative of $fComp$ with respect to x or r . That means:

$$\frac{d}{dx}(x^2 - 1) = 2x$$

$$\frac{d}{dr}((r^2 + 1)^2 - 1) = 2r(x^2 - 1) * (2r), x = r^2 + 1$$

The last line of outputs shows the evaluation of $fComp$ with respect to $r = 2.0$.

2.7 Multivariable Functions

A multivariable function is a function with several variables. There are two ways to create functions with several variables. First way is using combinations of existing single or multivariable functions. Second way is using user defined functions. We concentrate on the first approach in this section. For example, define a function $f(x, y) = \frac{1}{4}(1 - x)(1 - y)$:

```
1 import static edu.uta.futureye.function.operator.FMath.*;
2 ...
3 Function f = C(0.25).M(C1.S(X)).M(C1.S(Y));
4 System.out.println(f);
5 Variable v = new Variable("x",0.5).set("y", 0.5);
6 System.out.println(f.value(v));
```

The output is:

```
1 0.25 * (1.0 - x) * (1.0 - y)
2 0.0625
```

The evaluation procedure of multivariable functions is the same as the case of single variable functions. The only difference is that the variable should be defined with several variable names. With the following member function

```
1 public Variable set(String name, double val)
```

of class *Variable* one can set the value for a specified variable name. This function returns the object itself, so it can be called in manner of chain expression

```
1 Variable v = new Variable("x",0.5).set("y", 0.5);
```

It equals

```
1 Variable v = new Variable("x",0.5);
2 v.set("y", 0.5);
```

Composition of multivariable functions is also similar with single variable functions:

```
1 Map<String,Function> map = new HashMap<String,Function>();
2 map.put("x", fInx); //x = fInx(r,s)
3 map.put("y", fIny); //y = fIny(r,s)
4 Function fComp = fOut.compose(map);
```

Finite element shape functions are define in local coordinate system (e.g. barycenter coordinate system). The local variables can be transformed to Cartesian coordinate system. We need composite functions to express the relation between variables in local coordinate system and Cartesian coordinate system.

2.8 Vector and Vector Valued Functions

When we describe some physical quantities, for example displacement, velocity, electric field and magnetic field, vector and vector valued functions are often used. There are two java interfaces **Vector** and **VectorFunction** in FuturEye, all vectors should implement interface **Vector** and all vector valued functions should implement interface **VectorFunction**.

Class **SpaceVector** is an implementation of interface **Vector**. It acts as space vectors in mathematical context. For example:

```

1 //2D vectors
2 SpaceVector a = new SpaceVector(1.0,0.0);
3 SpaceVector b = new SpaceVector(0.0,1.0);
4 System.out.println(a.dot(b));
5
6 //3D vectors
7 SpaceVector a3 = new SpaceVector(1.0,0.0,0.0);
8 SpaceVector b3 = new SpaceVector(0.0,1.0,0.0);
9 a3.crossProduct(b3).print();

```

The output is:

```

1 0.0
2 0.0 0.0 1.0

```

The first line of outputs is the value of dot product $\vec{a} \cdot \vec{b}$, the second line is the value of cross product $\vec{a} \times \vec{b}$. It should be noted that function *crossProduct()* is not declared in interface **Vector**, because this function is mean less in other specific vector implementations, e.g. **SparseVector** and **SparseBlockVector**.

The following table shows the basic functions defined in interface **Vector**

edu.uta.futureeye.algebra.intf.Vector	Comment
int getDim ()	Returns dimension
void setDim (int dim)	Set dimension
double get (int index)	Returns x(index)
void set (int index, double value)	Set x(index)=value
void add (int index,double value)	Set x(index) = x(index) + value

The following table shows the linear algebra operations defined in interface **Vector** and the corresponding operations in Matlab and BLAS package.

edu.uta.futureeye.algebra.intf.Vector	Matlab	BLAS Level 1 (double)
Vector set (Vector y)	x=y	dcopy(...)
Vector set (double a, Vector y)	x=a*y	-
Vector add (Vector y)	x=x+y	-
Vector add (double a, Vector y)	x=x+a*y	-
Vector ax () or Vector scale ()	x=a*x	dscal(...)
Vector axpy (double a, Vector y)	x=a*x+y	daxpy(...)*
double dot (Vector y)	dot(x,y)	ddot(...)
double norm1 ()	norm(x,1)	dasum(...)
double norm2 ()	norm(x,2)	dnrm2(...)
double normInf ()	norm(x,inf)	idamax(...)**

* y=a*x+y

** index of max abs value

It should be noted that this **Vector** interface and its implementations are not designed for efficient linear algebra operations. Its existence is only for convenience in the finite element shape function construction and assemble procedures. For efficient linear algebra operations see interface **AlgebraVector**.

Class **SpaceVectorFunction** is an implementation of interface **VectorFunction**. Its instance represents vector valued functions in mathematic context. The following code segments will construct the function

$$\vec{f}(x, y, z) = \begin{pmatrix} x + 1 \\ 2y + 2 \\ 3z + 3 \end{pmatrix}$$

and do some basic operations:

```

1  Function f1 = new Fxpb("x",1.0,1.0);
2  Function f2 = new Fxpb("y",2.0,2.0);
3  Function f3 = new Fxpb("z",3.0,3.0);
4
5  SpaceVectorFunction svf = new SpaceVectorFunction(f1,f2,f3);
6  System.out.println("svf(x,y,z)="+svf);
7  System.out.println("svf(1,1,1)="+svf.value(
8      new Variable().set("x",1).set("y",1).set("z",1))
9      );
10
11 System.out.println(svf.dot(new SpaceVector(1,2,3)));
12 Function svfSquare = svf.dot(svf);
13 System.out.println(svfSquare);
14 System.out.println(svfSquare.value(
15     new Variable(
16         new VarPair("x",1),
17         new VarPair("y",1),
18         new VarPair("z",1)
19     )));

```

The outputs are:

```

1  svf(x,y,z)=(x+1.0  2.0*y+2.0  3.0*z+3.0  )
2  svf(1,1,1)=(2.0  4.0  6.0  )
3  x+1.0 + (2.0*y+2.0) * 2.0 + (3.0*z+3.0) * 3.0
4  (x+1.0) * (x+1.0) + (2.0*y+2.0) * (2.0*y+2.0) + (3.0*z+3.0) *
5  (3.0*z+3.0)
56.0

```

First, three single variable functions f_1, f_2, f_3 are created. Then they are passed to the constructor of **SpaceVectorFunction**. Thus, we get the vector valued function $svf(x, y, z) = (x + 1.0, 2.0 * y + 2.0, 3.0 * z + 3.0)^T$. We can evaluate it by passing variable name and value pairs. The code `new Variable().set("x",1).set("y",1).set("z",1)` will create an object of **Variable**. Function `Variable.set(Stringname, doubleval)` will return the object itself, so the chain expression can be used to simplify code typing. The most simplified expression can be `new Variable("x",1).set("y",1).set("z",1)`. The constructor will also be used to set a name and value pair. There are another way to construct variable objects. The class **VarPair** represents the name and value pair of a variable object, its instance can be passed to the constructor to construct variables. The last print part of the code segment shows the usage of this method.

Now, we discuss the combinations of vector valued functions and composite vector valued functions. If we take out a component of a vector valued function, this component can be treated as a single valued function that is already described in Section 2.4. The following table gives the similar operations with respect to every component of a vector valued functions.

Operation functions in AbstractFunction	Operations
VectorFunction A (VectorFunction g)	$\vec{f}(x) + \vec{g}(x)$
VectorFunction A (Vector g)	$\vec{f}(x) + \vec{g}$
VectorFunction S (VectorFunction g)	$\vec{f}(x) - \vec{g}(x)$
VectorFunction S (Vector g)	$\vec{f}(x) - \vec{g}$
VectorFunction M (VectorFunction g)	$f_i(x) * g_i(x), i = 1, \dots, n$
VectorFunction M (Vector g)	$f_i(x) * g_i, i = 1, \dots, n$
VectorFunction D (VectorFunction g)	$f_i(x)/g_i(x), i = 1, \dots, n$
VectorFunction D (Vector g)	$f_i(x)/g_i, i = 1, \dots, n$

The above functions are defined in class **AbstractVectorFunction**. The returned values are instance of class **SpaceVectorFunction** which extends class **AbstractVectorFunction**. Any class that extends from **AbstractVectorFunction** will automatically has the implemented member functions listed in the above table. These implemented functions will be convenient for user defined classes. Of course, these functions can be over rided if necessary.

Vector algebra operations of vector valued function is similar as in the interface **Vector**. The following table lists the operations.

edu.uta.futureye.function.intf.VectorFunction	Operations
VectorFunction set (VectorFunction g)	$\vec{f}(x) = \vec{g}(x)$
VectorFunction set (double a, VectorFunction g)	$\vec{f}(x) = a * \vec{g}(x)$
VectorFunction add (VectorFunction g)	$\vec{f}(x) = \vec{f}(x) + \vec{g}(x)$
VectorFunction add (double a, VectorFunction g)	$\vec{f}(x) = \vec{f}(x) + a * \vec{g}(x)$
VectorFunction ax (double a) or VectorFunction scale (double a)	$\vec{f}(x) = a * \vec{f}(x)$
VectorFunction axpy (double a, VectorFunction g)	$\vec{f}(x) = a * \vec{f}(x) + \vec{g}(x)$
double dot (VectorFunction g)	$\vec{f}(x) \cdot \vec{g}(x)$

It should be noted that these functions are implemented in class **AbstractVectorFunction**, but are not the best implementation, because the actual strategy of storage is not known by **AbstractVectorFunction**. The class **SpaceVectorFunction** implements all these functions. So, it's a good decision to form new vector valued function based on class **SpaceVectorFunction**.

Composite vector valued functions are easy to form. What you need to do is to compose each component of a vector valued function function. Class **AbstractVectorFunction** has the function *compose()* that has already implemented this procedure:

```

1  public VectorFunction compose(Map<String,Function> fInners) {
2      for(int i=1; i<=dim; i++)
3          this.set(i, this.get(i).compose(fInners));
4      return this;
5  }
```

2.9 Class FMath

We have introduced some static objects defined in class **FMath** in previous sections for example $C0, PI, X, Y, Z$ etc. Class **FMath** also contains methods for performing basic numeric operations for objects of type **Function**, **Vector** and **VectorFunction**. The following table shows all the public static methods defined in class **FMath**.

Methods in FMath	Comment
Function sqrt(final Function f)	$\sqrt{f(\mathbf{x})}$
Function pow(Function f, double p)	$f(\mathbf{x})^p$
Function power(final Function f1, final Function f2)	$f_1(\mathbf{x})^{f_2(\mathbf{y})}$
Function abs(final Function f)	$ f(x) $
Function sum(Function ...fi)	$\sum_{i=1}^N f_i(\mathbf{x})$
Function linearCombination(double c1, Function f1, double c2, Function f2)	$c_1 f_1(\mathbf{x}) + c_2 f_2(\mathbf{x})$
Function linearCombination(double []ci, Function []fi)	$\sum_{i=1}^N c_i f_i(\mathbf{x})$
VectorFunction grad(Function fun)	$Grad f(\mathbf{x}) = \nabla f(\mathbf{x})$
VectorFunction grad(Function fun, ObjList<String> varNames)	$Grad_{\mathbf{x}} f = \nabla_{\mathbf{x}} f$
Function div(VectorFunction vFun)	$Div \vec{f}(\mathbf{x})$
Function div(VectorFunction vFun, ObjList<String> varNames)	$Div_{\mathbf{x}} \vec{f}$
Function curl(VectorFunction vFun)	$Curl \vec{f}(\mathbf{x})$
Function curl(VectorFunction vFun, ObjList<String> varNames)	$Curl_{\mathbf{x}} \vec{f}$
Vector sum(Vector ...vi)	$\vec{v}_1 + \vec{v}_2 + \dots + \vec{v}_n$
double sum(Vector v)	$v_1 + v_2 + \dots + v_n$
Vector log(Vector v)	$(\ln v_1, \ln v_2, \dots, \ln v_n)^T$
Vector exp(Vector v)	$(e^{v_1}, e^{v_2}, \dots, e^{v_n})^T$
Vector abs(Vector v)	$(v_1 , v_2 , \dots, v_n)^T$
double max(Vector v)	$\max(v_1, v_2, \dots, v_n)$
double min(Vector v)	$\min(v_1, v_2, \dots, v_n)$

We give some examples to show how to use class **FMath** to speed up our coding. In the main function of the following class **FMathTest**, two functions u, v are defined first,

$$u(x, y, z) = x^2 + y^2 + z^2$$

$$v(x, y, z) = xyz$$

Then we compute the expression

$$u_x v_x + u_y v_y + u_z v_z$$

by the following code

```

1 import static edu.uta.futureye.function.operator.FMath.*;
2 ...
3 sum(
4     u._d("x").M(v._d("x")),
5     u._d("y").M(v._d("y")),
6     u._d("z").M(v._d("z"))
7 )

```

The result is

$$u_x v_x + u_y v_y + u_z v_z = 2xyz + 2yxz + 2zxy = 6xyz$$

Actually, the above expression can be written as

$$\nabla u \cdot \nabla v = u_x v_x + u_y v_y + u_z v_z$$

So, the following code equally gives the expression above in form of function operators

```
1 import static edu.uta.futureeye.function.operator.FMath.*;
2 ...
3 grad(u).dot(grad(v))
```

Next, we show another example, define

$$\text{invR}(x, y, z) = \frac{1}{\sqrt{x^2 + y^2 + z^2}}$$

Then we compute

$$\text{Div}(\text{Grad}(\text{invR})) = \nabla^2 \text{invR} = \Delta \text{invR}$$

with code

```
1 div(grad(invR))
```

In order to demo how to use the second parameter of function operators, we define

$$\text{invR2}(r) = \frac{1}{r}$$

where $r = r(x, y, z) = \sqrt{x^2 + y^2 + z^2}$.

When we compute gradient of invR2 with respect to different variables, the result will be different:

$$\nabla_r(\text{invR2}) = \nabla_r\left(\frac{1}{r}\right) = \left(\frac{1}{r}\right)' = -\frac{1}{r^2}$$

$$\nabla_{x,y,z}(\text{invR2}) = \nabla_{x,y,z}\left(\frac{1}{r(x,y,z)}\right) = \begin{pmatrix} -\frac{x}{(x^2+y^2+z^2)^{3/2}} \\ -\frac{y}{(x^2+y^2+z^2)^{3/2}} \\ -\frac{z}{(x^2+y^2+z^2)^{3/2}} \end{pmatrix}$$

The code for the first equation is

```
1 grad(invR2)
```

and for the second equation is

```
1 ObjList<String> coordSys =
2     new ObjList<String>("x", "y", "z");
3 grad(invR2, coordSys);
```

That is to say, for composite functions, we must specify the seconde parameter of function operator methods to the coordinate which we want to perform the work.

Interested users can check the operation results of function invR and invR2 as below:

```
1 div(grad(invR))
2 div(grad(invR2, coordSys), coordSys)
```

They are all equals to 0, because that

$$\Delta\left(\frac{1}{r}\right) = \frac{1}{r^3} \left(-3 + 3 \frac{x^2 + y^2 + z^2}{r^2}\right) = 0$$

The whole example code is listed bellow:

```

1 package edu.uta.futureye.test;
2
3 import java.util.HashMap;
4 import java.util.Map;
5 import edu.uta.futureye.algebra.SpaceVector;
6 import edu.uta.futureye.function.intf.Function;
7 import edu.uta.futureye.util.container.ObjList;
8 import static edu.uta.futureye.function.operator.FMath.*;
9
10 public class FMathTest {
11     public static void main(String[] args) {
12         //u(x,y,z) = x^2+y^2+z^2
13         Function u = sum(X.M(X), Y.M(Y), Z.M(Z));
14
15         //v(x,y,z) = x*y*z
16         Function v = X.M(Y).M(Z);
17
18         //Grad(u) \cdot Grad(v)
19         System.out.println(
20             sum(
21                 u._d("x").M(v._d("x")),
22                 u._d("y").M(v._d("y")),
23                 u._d("z").M(v._d("z"))));
24
25         //Grad(u) \cdot Grad(v)
26         System.out.println(grad(u).dot(grad(v)));
27
28         //invR = 1/sqrt(x^2+y^2+z^2)
29         Function invR = C1.D(sqrt(u));
30         //Div(Grad(invR)) = Laplacian(invR)
31         System.out.println(div(grad(invR)));
32
33         //invR = 1/r(x,y,z) = 1/sqrt(x^2+y^2+z^2)
34         Function invR2 = C1.D(R);
35         Map<String, Function> fInners =
36             new HashMap<String, Function>();
37         fInners.put("r", sqrt(u));
38         invR2 = invR2.compose(fInners);
39
40         //Grad(invR2), gradient respect to r
41         System.out.println(grad(invR2));
42         ObjList<String> coordSys =
43             new ObjList<String>("x","y","z");
44
45         //Div_{x,y,z}(Grad_{x,y,z}(invR2))=Laplacian_{x,y,z}(invR2)
46         //gradient with respect to x,y,z
47         //divergence with respect to x,y,z

```

```

48     System.out.println(div(grad(invR2, coordSys), coordSys));
49     }
50 }

```

2.10 User Defined Functions

In this section, we describe how to define yourself function classes. Sometimes, a special function may be required in the application and the function can not be created from existing functions through function combinations and/or composition. The key thing to define yourself function classes is to implement the interface **Function** or to extend the abstract class **AbstractFunction**. Three examples will be given to demonstrate the way of how to define user functions.

Example 1. **DiscreteIndexFunction**

This is a general function that can be evaluated at points referenced by discrete positive integer indices. It is useful when you have a list of external data and you want to use that data by it's indices. The full definition of the class **DiscreteIndexFunction** is listed below:

```

1  package edu.uta.futureeye.function.basic;
2
3  import java.util.HashMap;
4  import java.util.Map;
5
6  import edu.uta.futureeye.function.AbstractFunction;
7  import edu.uta.futureeye.function.Variable;
8  import edu.uta.futureeye.util.FutureeyeException;
9  import edu.uta.futureeye.util.PairDoubleInteger;
10 import edu.uta.futureeye.util.container.ObjList;
11
12 public class DiscreteIndexFunction extends AbstractFunction {
13     protected Map<Integer, Double> data = new HashMap<Integer,
14         Double>();
15
16     public DiscreteIndexFunction() {
17     }
18
19     public DiscreteIndexFunction(ObjList<PairDoubleInteger> list)
20     {
21         for(int i=1;i<=list.size();i++) {
22             PairDoubleInteger pair = list.at(i);
23             this.data.put(pair.i, pair.d);
24         }
25     }
26
27     public void set(int index, double value) {
28         this.data.put(index, value);
29     }
30
31     @Override
32     public double value(Variable v) {
33         if(v.getIndex() <= 0)
34             throw new FutureeyeException("v.getIndex()="+v.getIndex());
35         return data.get(v.getIndex());
36     }
37 }

```



```

34     }
35 }

```

A member variable **data** with type *Map<Integer, Double>* is used to contain the pair of index and value. Two constructor are given to construct an empty object and an object from a list containing elements with type **PairDoubleInteger**. Member function **set(...)** is used to set special index and value pair. Member function **value(...)** is the key function. The positive integer index can be retrieved from the parameter **v**. Then the value required corresponds to the index will be returned.

Class **Variable** is designed to take additional objects beside the regular values when used as function parameters. Code segment below shows how to use **Variable** object to take those additional objects. This feature allows user defined functions to implement functions having special requirements that associate with the additional parameter objects.

```

1     Variable v = new Variable("x",5.0).set("y",6.0);
2     System.out.println(v.get("x"));
3     System.out.println(v.get("y"));
4     v.setIndex(20); //additional data: index
5     System.out.println(v.getIndex());
6     v.setElement(new Element()); //additional data: element
7     System.out.println(v.getElement());

```

Example 2. **Vector2Function**

This function class is similar to **DiscreteIndexFunction**. They are all evaluated at positive integer indices. But the difference is that the constructor of this class needs an object of type **Vector**. The definition of the class below is very short, but it's useful in the case that you need an object of type **Function** instead of an object of type **Vector**. That is to say, the two object represent the same quantity or mathematical function. For example, when you get a vector result (e.g. temperature distribution) from a solver and want to pass the result further to other object function members as a parameter of type **Function**, you can use class **Vector2Function** to do the conversion. The real implementation of class **Vector2Function** is a bit more complicated. It also allows one to evaluate the function at data point that different from any point associated with a index. In this case, interpolation is used on the correspond element of the mesh that related to the vector.

```

1 package edu.uta.futureye.function.basic;
2
3 import edu.uta.futureye.algebra.intf.Vector;
4 import edu.uta.futureye.function.AbstractFunction;
5 import edu.uta.futureye.function.Variable;
6 import edu.uta.futureye.util.FutureyeException;
7
8 public class Vector2Function extends AbstractFunction {
9     Vector u = null;
10
11     public Vector2Function(Vector u) {
12         this.u = u;
13     }
14
15     @Override
16     public double value(Variable v) {
17         int index = v.getIndex();

```

```

18         if(index <= 0) {
19             throw new FutureyeException("Error:_Vector2Function_
                index="+index);
20         } else {
21             return u.get(index);
22         }
23     }
24 }

```

Example 3. DuDn

In the last example, we will give a function that usually used in FEM context. Class **DuDn** represents the function

$$\frac{\partial u}{\partial n}$$

in mathematical context. Where n is the outer normal vector of the domain that function u defined in it. The key part of implementation is according

$$\frac{\partial u}{\partial n} = \mathbf{n} \cdot \nabla u = (n_1, n_2, n_3) \begin{pmatrix} u_x \\ u_y \\ u_z \end{pmatrix}$$

There are two constructors defined in class **DuDn** for convenience. Class **Element** will be explained in next section.

```

1  public class DuDn extends AbstractFunction implements
2  ElementDependentFunction {
3      protected Element e = null;
4      protected Function u = null;
5      protected Function u_x = null;
6      protected Function u_y = null;
7      protected Function u_z = null;
8      protected Vector norm = null;
9
10     public DuDn(Function u) {
11         this.u = u;
12     }
13
14     public DuDn(Function u_x, Function u_y, Function u_z) {
15         this.u_x = u_x;
16         this.u_y = u_y;
17         this.u_z = u_z;
18     }
19
20     @Override
21     public void setElement(Element e) {
22         this.e = e;
23         //Compute outer normal vector
24         GeoEntity ge = e.getGeoEntity();
25         if(ge instanceof Edge) {
26             norm = ((Edge)ge).getNormVector();
27         } else if(ge instanceof Face) {
28             norm = ((Face)ge).getNormVector();
29         } else {
30             throw new FutureyeException("Unsuported_element_type");

```

```

31     }
32 }
33
34 @Override
35 public double value(Variable v) {
36     Function rlt = null;
37     this.setElement(v.getElement());
38     if(u != null) {
39         //u is passed into constructor
40         rlt = FMath.grad(u).dot(norm);
41     } else if(this.norm.getDim() == 2) {
42         //2D case
43         rlt = u_x.M(new FC(norm.get(1)))
44             .A(
45                 u_y.M(new FC(norm.get(2)))
46             );
47     } else if(this.norm.getDim() == 3) {
48         //3D case
49         rlt = FMath.sum(
50             u_x.M(new FC(norm.get(1))),
51             u_y.M(new FC(norm.get(2))),
52             u_z.M(new FC(norm.get(3)))
53         );
54     } else {
55         throw new FutureyeException(
56             "Error: u="+u+", this.norm.getDim()="+this.norm.getDim());
57     }
58     return rlt.value(v);
59 }
60
61 public String toString() {
62     return "DuDn";
63 }
64 }

```

2.11 Speedup Evaluation of Functions

The cost of function evaluation may be time consuming if a function created from combination of many basic functions. For example, the Jacobian in the integrand of finite element quadrature could occur several times, a typical item of integrand in Laplace problem will be

$$(2.11.1) \quad \int_K \frac{\partial N_i}{\partial x} \frac{\partial N_j}{\partial x} dK = \int_K \left(\frac{\partial N_i}{\partial r} \frac{\partial r}{\partial x} + \frac{\partial N_i}{\partial s} \frac{\partial s}{\partial x} \right) \left(\frac{\partial N_j}{\partial r} \frac{\partial r}{\partial y} + \frac{\partial N_j}{\partial s} \frac{\partial s}{\partial y} \right) |J| d\hat{K}$$

where

$$\frac{\partial r}{\partial x} = \frac{1}{|J|} \frac{\partial y}{\partial s}, \quad \frac{\partial r}{\partial y} = -\frac{1}{|J|} \frac{\partial x}{\partial s}, \quad \frac{\partial s}{\partial x} = -\frac{1}{|J|} \frac{\partial y}{\partial r}, \quad \frac{\partial s}{\partial y} = \frac{1}{|J|} \frac{\partial x}{\partial r}$$

$$|J| = \begin{vmatrix} \frac{\partial x}{\partial r} & \frac{\partial x}{\partial s} \\ \frac{\partial y}{\partial r} & \frac{\partial y}{\partial s} \end{vmatrix}$$

$$x = \sum_i x_i N_i, y = \sum_i y_i N_i$$

K and \hat{K} are real and reference element respectively. N_i are shape functions defined on reference element. In this example, when evaluate the integrand Jacobian will be evaluated 5 times at each quadrature point. The first derivative of shape functions will further be evaluated more times. In order to speedup evaluation of functions, we have the following strategies

(1) Using user defined function to create shape function instead of combination of basic function. Because in current implementation of function combination we use nested function mechanism to create new functions from basic functions. There is no effort to reduce the expression of a function. Take shape function $N_i = (1 - r)(1 - s)/4$ for example, if we use combination of basic functions, N_1 will be defined as below:

```

1 Function f1 = new Fxpb("r", -1.0, 1.0);
2 Function f2 = new Fxpb("s", -1.0, 1.0);
3 Function N1 = f1.M(f2).D(4.0);

```

The evaluation procedure of N_1 is shown in Fig.3

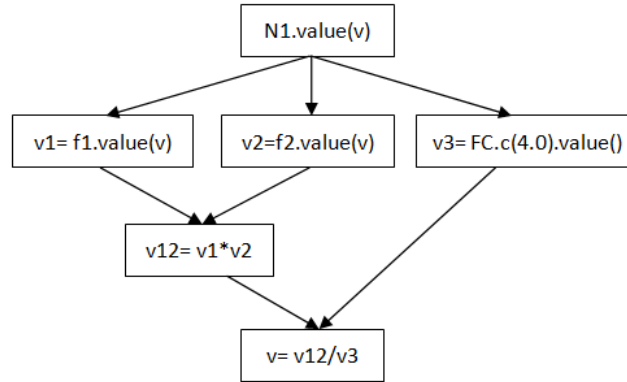


Figure 3: Evaluation procedure of N_1

If we use user defined function, N_1 will be defined as below:

```

1 public class N1 extends AbstractFunction implements {
2     @Override
3     public double value(Variable v) {
4         double r = v.get("r");
5         double s = v.get("s");
6         return (1-r)*(1-s)/4.0;
7     }
8 }

```

From the definition above, it is easy to see that evaluation of user defined function N_1 will cost much less than combination of basic functions especially for functions which have a long expression.

(2) Use cached evaluation mechanism to reduce repeated evaluation of the same function in an expression. Remember that the Jacobian need to be evaluated 5 times in the example above. Due to nested function evaluation mechanism, it is hard to maintain a local cache at the definition of combination of functions. While a global cache is easy to lead conflicts. Our implementation is to define a new member function in the interface **Function**

```
1  double value(Variable v, Map<Object, Object> cache);
```

In this manner of implementation, a local cache will be created together with the variable at the moment of the beginning of every distinct evaluation. Then the cache will be passed down to subsequent nested function calls. With the help of the cache, repeated evaluations can be detected and only one evaluation of the same function will be performed. The result of the first evaluation can be stored in the cache at the same time and be taken out later when the same function with respect to the same variable is called. An example of how to use this cache mechanism in a user defined function is shown in the following definition of class **Jacobian2D** which is an inner class of **CoordinateTransform**.

```
1  public static class Jacobian2D extends AbstractFunction {
2      Function[] funks = null;
3      public Jacobian2D(Function[] funks) {
4          this.funks = funks;
5      }
6
7      @Override
8      public double value(Variable v) {
9          return value(v, null);
10     }
11
12     @Override
13     public double value(Variable v, Map<Object, Object> cache)
14     {
15         Double detJ = null;
16         if(cache != null) {
17             detJ = (Double)cache.get(1);
18         }
19         if(detJ == null) {
20             double J11 = funks[0].value(v);
21             double J12 = funks[1].value(v);
22             double J21 = funks[2].value(v);
23             double J22 = funks[3].value(v);
24             detJ = J11*J22-J12*J21;
25             if(cache != null) {
26                 cache.put(1, detJ);
27             }
28         }
29         return detJ;
30     }
31
32     public String toString() {
33         return "Jacobian2D";
34     }
35 }
```

We can see that in the second **value()** function **cache** object is used to detect repeated evaluations of the Jacobian. Here the key 1 is used to map the result. In fact, any object can be used to map the result. Because it is a local cache with respect to a specific variable you just need to keep that there is no same keys used in all other nested function objects. Obviously, you can store as many objects as you wish in the cache. It is very easy to use this cached evaluation feature to speedup the evaluation. Take equation 2.11.1 for example. Assume that we have an function object **f** which is the integrand of $\int_K \frac{\partial N_i}{\partial x} \frac{\partial N_j}{\partial x} dK$. An evaluation of **f** with cache enabled is shown in the following code segment.

```
1 Variable v = new Variable("r",0.5).set("s","0.5");
2 f.value(v,new HashMap<Object, Object>());
```

What you need to do is just new a map object and pass it to function **value()**. Remember that there are 5 times of evaluation of Jacobian in **f**. From the definition of class **Jacobian2D** we can see that there will be only one real evaluation of the Jacobian.

(3) Using batch evaluation interface of a function. For more efficiency, the following function is defined in **Function**.

```
1 double[] valueArray(VariableArray valAry, Map<Object, Object>
    cache);
```

This function will return an array of function values at an corresponding array of variables. Thus, repeatedly function calls at every variable will be avoid. The following code segment in numerical integration class **FOIntegrate** demonstrates how to use the batch evaluation function

```
1 VariableArray valAry = new VariableArray();
2 ...
3 double []ra = new double[27];
4 double []sa = new double[27];
5 double []ta = new double[27];
6 ... //assign values to ra,sa,ta
7 valAry.set("r", ra);
8 valAry.set("s", sa);
9 valAry.set("t", ta);
10 double[] rltAry = integrand.valueArray(valAry,new HashMap<Object,
    Object>());
```

At the end, we will give a small test of assembling a 3D Laplace problem on a mesh with 8000 nodes and 6859 elements. Trilinear hexahedron element with 8 integral points in element quadrature is used. Assembling procedure run with single thread on Intel(R) Core(TM) i5 2.40GHz CPU and 64bit java 1.7 JDK. Without using user defined function, cache mechanism and batch evaluation the total time consuming is 380 seconds, while using the three speedup tricks the total time consuming is only 15 seconds. It is almost 25 times faster than that without optimization of function evaluation. This optimized function evaluation cost can be acceptable. Fortunately, the computing amount of assembling process grows with the number of element linearly and can be parallelized easily by multi-thread or on computer cluster. So it will not be a big barrier in the whole process of solving a PDE.

3 Basic Finite Element Components

3.1 Mesh, Element and Node Classes

3.1.1 Basic Grid Properties

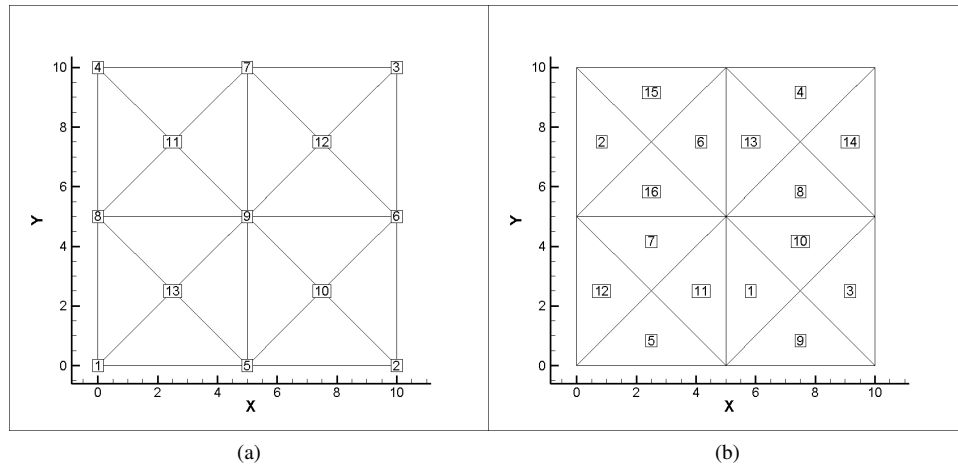
Class **Mesh** is designed to store any grid of dimensions 1,2 or 3 and different element types that generated from other software. The current version of FuturEye doesn't provide grid generation function. The following code gives an example of reading a grid from a file with ASCII UCD format that generated by Gridgen:

```
1 MeshReader reader = new MeshReader("patch_triangle.grd");
2 Mesh mesh = reader.read2DMesh();
```

The content of the file "patch_triangle.grd" is:

```
1 # UCD geometry file from Gridgen 15.08R1,
2 # a product of Pointwise, Inc.
3 # 01-Nov-10 23:08:59
4 #
5 13 16 0 0 0
6 1 0.0000000000e+000 0.0000000000e+000 0.0000000000e+000
7 2 1.0000000000e+001 0.0000000000e+000 0.0000000000e+000
8 3 1.0000000000e+001 1.0000000000e+001 0.0000000000e+000
9 4 0.0000000000e+000 1.0000000000e+001 0.0000000000e+000
10 5 5.0000000000e+000 0.0000000000e+000 0.0000000000e+000
11 6 1.0000000000e+001 5.0000000000e+000 0.0000000000e+000
12 7 5.0000000000e+000 1.0000000000e+001 0.0000000000e+000
13 8 0.0000000000e+000 5.0000000000e+000 0.0000000000e+000
14 9 5.0000000000e+000 5.0000000000e+000 0.0000000000e+000
15 10 7.5000000000e+000 2.5000000000e+000 0.0000000000e+000
16 11 2.5000000000e+000 7.5000000000e+000 0.0000000000e+000
17 12 7.5000000000e+000 7.5000000000e+000 0.0000000000e+000
18 13 2.5000000000e+000 2.5000000000e+000 0.0000000000e+000
19 1 0 tri 9 5 10
20 2 0 tri 11 4 8
21 3 0 tri 10 2 6
22 4 0 tri 3 7 12
23 5 0 tri 1 5 13
24 6 0 tri 9 7 11
25 7 0 tri 9 8 13
26 8 0 tri 9 6 12
27 9 0 tri 10 5 2
28 10 0 tri 9 10 6
29 11 0 tri 13 5 9
30 12 0 tri 1 13 8
31 13 0 tri 12 7 9
32 14 0 tri 3 12 6
33 15 0 tri 11 7 4
34 16 0 tri 9 11 8
```

Fig.4(a) shows the grid with nodes numbers and Fig.4(b) shows the grid with elements numbers. We can see that there are total 13 nodes and 16 elements. Actually, the basic components of a **Mesh** object are nodes and elements. FuturEye provides two classes

Figure 4: *patch_triangle.grd*

NodeList and **ElementList**, which are list containers, the following code segments shows how they works:

```

1  NodeList nodes = mesh.getNodeList();
2  ElementList elements = mesh.getElementList();
3
4  for(int i=1;i<=nodes.size();i++) {
5      System.out.println(nodes.at(i));
6  }
7  for(int i=1;i<=elements.size();i++) {
8      System.out.println(elements.at(i));
9  }

```

It should be noted that the index starts from 1. It's in keeping with the mathematic context of FEM convention for object indexing. The output is:

```

1  GN1( 0.0 0.0 )
2  GN2( 10.0 0.0 )
3  GN3( 10.0 10.0 )
4  GN4( 0.0 10.0 )
5  GN5( 5.0 0.0 )
6  GN6( 10.0 5.0 )
7  GN7( 5.0 10.0 )
8  GN8( 0.0 5.0 )
9  GN9( 5.0 5.0 )
10 GN10( 7.5 2.5 )
11 GN11( 2.5 7.5 )
12 GN12( 7.5 7.5 )
13 GN13( 2.5 2.5 )
14 GE1( 9_ 5_ 10_ )
15 GE2( 11_ 4_ 8_ )
16 GE3( 10_ 2_ 6_ )

```



```

17 GE4( 3_ 7_ 12_ )
18 GE5( 1_ 5_ 13_ )
19 GE6( 9_ 7_ 11_ )
20 GE7( 9_ 8_ 13_ )
21 GE8( 9_ 6_ 12_ )
22 GE9( 10_ 5_ 2_ )
23 GE10( 9_ 10_ 6_ )
24 GE11( 13_ 5_ 9_ )
25 GE12( 1_ 13_ 8_ )
26 GE13( 12_ 7_ 9_ )
27 GE14( 3_ 12_ 6_ )
28 GE15( 11_ 7_ 4_ )
29 GE16( 9_ 11_ 8_ )

```

The capital prefix letters “GN” represents “Global Node” and “GE” represents “Global Element”. We will explain the “_” that following the numbers in the brackets of global elements later (section 3.1.3).

The global index of a node and it’s coordinate can be accessed in the following ways:

```

1 Node node = nodes.at(9);
2 System.out.println("GN"+node.globalIndex+"_coord:"+
3     node.coord(1)+","+"node.coord(2));
4 double []coord = node.coords();
5 System.out.println("GN"+node.getIndex()+"_coord:"+
6     coord[0]+","+"coord[1]);

```

The outputs are:

```

1 GN9 coord:5.0,5.0
2 GN9 coord:5.0,5.0

```

To access all the nodes associated with an element is pretty easy, for example

```

1 Element ele = elements.at(1);
2 System.out.println("GE"+ele.globalIndex+": "+
3     ele.nodes);

```

We will get

```

1 GE1:NodeList[GN9( 5.0 5.0 ), GN5( 5.0 0.0 ), GN10( 7.5 2.5 )]

```

Where *ele.nodes* is an object of **NodeList** that contains all the nodes belong to the element. From the above output, we can find out that element 1 contains global nodes 9,5 and 10.

Remark. All the list classes that have the index starts from 1 are extends from class **ObjList < T >**. It’s a template class that acts as a container class in keeping with the mathematic context of FEM convention for object indexing. It’s also easy to convert from/to java native array or **java.util.List**. For example, if we need to sort the nodes by global index, all we need to do is call *toList()* and sort it with the Java Collections Framework.

```

1 List<Node> list = nodes.toList();
2 Collections.sort(list, new Comparator<Node>(){
3     @Override
4     public int compare(Node o1, Node o2) {
5         //desc
6         if(o1.globalIndex > o2.globalIndex)
7             return -1;

```

```

8         else
9             return 1;
10     }
11 }

```

The following table gives a brief description of the member functions in **ObjList** < **T** >.

edu.uta.futureye.util.list.ObjList< <i>T</i> >	Comment
int size ()	return list size
<i>T</i> at (int <i>i</i>)	return <i>i</i> th object
ObjList< <i>T</i> > add (<i>T</i> <i>e</i>)	add one object
ObjList< <i>T</i> > addAll (ObjList< <i>T</i> > list)	add multi objects
void clear ()	clear list
<i>T</i> remove (int <i>i</i>)	remove <i>i</i> th object
boolean remove (<i>T</i> <i>e</i>)	remove object <i>e</i>
ObjList< <i>T</i> > subList (int begin,int end)	return sublist[begin:end]
ObjList< <i>T</i> > subList (int begin,int end,int step)	return sublist[begin:step:end]
ObjList< <i>T</i> > subList (List<Integer> set)	return a sublist that index in the <i>set</i>
List< <i>T</i> > toList ()	convert to java.util.List
Object[] toArray ()	convert to java native array
ObjList< <i>T</i> > fromList (List< <i>T</i> > list)	construct from java.util.List
ObjList< <i>T</i> > fromArray (<i>T</i> [] array)	construct form java native array

3.1.2 Advanced Grid Properties

Now, we come to the advanced properties contained in **Mesh**, **Element** and **Node** classes. Given a mesh, the following things may need to be known in some applications:

- (1) A node belongs to which elements
- (2) A node's neighbor nodes
- (3) An element's neighbor elements
- (4) Inner nodes and border nodes of a mesh
- (5) The global edge(2D,3D) list or face(3D) list

Class **Mesh** is designed to compute these properties based on the basic nodes and elements information.

edu.uta.futureye.util.list.Mesh	Comment
void computeNodeBelongsToElements ()	(1)A node belongs to which elements
void computeNeighborNodes ()	(2)A node's neighbor nodes
void computeNeighborElements ()	(3)An element's neighbor elements
void markBorderNode (Map<NodeType,Function> mapNTF)	(4)Inner nodes and border nodes of a mesh
void computeGlobalEdge ()	(5)The global edge list (2D,3D)
void computeGlobalFace ()	(5)The global face list (3D)

To know a node belongs to which element is important, because we can get (2) and (4) listed above based on this information. In most applications, after reading a mesh file and getting a **Mesh** object, call the function *computeNodeBelongsToElements*() first. In order to get (3), *computeGlobalEdge*() must be called first in 2D case and call *computeGlobalFace*() additionally in 3D case.

The elements that a node is contained in are stored in the node's public member **belongToElements**. It's an instance of **ElementList** so all the operations defined in **ObjList** < **T** >

can be used to manipulate these elements. Neighbor nodes or elements are similar. The following code segment gives a demo. All the information of these relations are printed out. We omit the long output results. Interested users can run this demo by yourself.

```

1  MeshReader reader = new MeshReader("patch_triangle.grd");
2  Mesh mesh = reader.read2DMesh();
3  NodeList nodes = mesh.getNodeList();
4  ElementList elements = mesh.getElementList();
5
6  mesh.computeNodeBelongsToElements();
7  //Print all elements that a node is contained in them
8  for(int i=1;i<=nodes.size();i++) {
9      System.out.println(nodes.at(i)+":_"+
10         nodes.at(i).belongsToElements);
11  }
12
13  mesh.computeNeighborNodes();
14  //Print all neighbors of a node
15  for(int i=1;i<=nodes.size();i++) {
16      System.out.println(nodes.at(i)+":_"+
17         nodes.at(i).neighbors);
18  }
19
20  mesh.computeGlobalEdge();
21  EdgeList edges = mesh.getEdgeList();
22  mesh.computeNeighborElements();
23  //Print all global edges
24  for(int i=1;i<=edges.size();i++) {
25      System.out.println(edges.at(i));
26  }
27  //Print all neighbors of an element
28  for(int i=1;i<=elements.size();i++) {
29      System.out.println(elements.at(i)+":_"+
30         elements.at(i).neighbors);
31  }

```

3.1.3 Mark Boundary Types

Boundary types of nodes on a grid are needed when you solve PDE problems on the grid with different boundary conditions. The basic way to mark boundary types for the nodes on boundary is that you find out boundary nodes first and call the member functions of the node which defined in class **Node**, listed below

```

1  //for scalar valued problems
2  public void setNodeType(NodeType nodeType)
3  //for vector valued problems
4  public void setNodeType(int nVVFComponent, NodeType nodeType)

```

Enum type **NodeType** defined 4 types of node,

```

1  public enum NodeType {
2      Inner, //Inner
3      Dirichlet, Neumann, Robin //Border
4  }

```

One thing should be noted that for the nodes that are not on the boundary should also be marked as type **NodeType.Inner**.

In practical applications, information of boundary node types may be provided along with grid files or separate boundary condition files. What you should do is that you read the files first and call **setNodeType(...)** to set node types for the nodes which are marked out with different node types in the files.

However, we provide another way to mark node types. It is easy to use for grid with simple geometry of boundary. Class **Mesh** provides a method below to mark node types.

```
1 void markBorderNode(Map<NodeType, Function> mapNTF)
```

The parameter of the function is a map, so different node types can be marked at the same time. The key of the map is enumerate values of **NodeType**. The value of the map is a user defined function that indicates which part of the domain to be marked as the node type corresponding to the key of the map. If a positive number is returned by the user defined function, the node will be marked. For example, if we have an object **mesh**, a 2D rectangle domain $[-3,3] \times [-3,3]$, the following code segments will mark the nodes on the **mesh**.

```
1 HashMap<NodeType, Function> mapNTF =
2     new HashMap<NodeType, Function>();
3 mapNTF.put(NodeType.Robin, new AbstractFunction("x","y"){
4     @Override
5     public double value(Variable v) {
6         if(3.0-v.get("x") < Constant.meshEps)
7             return 1.0;
8         else
9             return -1.0;
10    }
11 });
12 mapNTF.put(NodeType.Dirichlet, null);
13 mesh.markBorderNode(mapNTF);
```

Nodes on the right side of the rectangle domain are marked as *Robin* type and other nodes are marked as *Dirichlet* type. The underlining procedure of the function goes like this: It first checks a node whether it is a boundary node. If yes, iterates the map and evaluate the user defined functions at the coordinate of the node and marks the node as the type in the key when the function returns a positive number. If the the function is null, the type in the key will be default type for nodes that having non-positive number returned by the function. If the node is not a boundary node, it will be marked as *Inner* type.

For some cases, more than one types must be marked on a single node, e.g. vector valued problems. The following code shows how to mark boundary nodes types for a 2D Stokes problem with unknown function (u, v, p) .

```
1 ElementList eList = mesh.getElementList();
2 for(int i=1; i<=eList.size(); i++) {
3     System.out.println(i+"└─" + eList.at(i));
4 }
5
6 //Mark border types for Stokes problem: (u v p)
7 HashMap<NodeType, Function> mapNTF_uv =
8     new HashMap<NodeType, Function>();
9 //mapNTF.put(NodeType.Dirichlet, null);
```

```

10     mapNTF_uv.put(NodeType.Dirichlet, new AbstractFunction("x","y"
11         ) {
12             @Override
13             public double value(Variable v) {
14                 ...
15             }
16     });
17     mapNTF_uv.put(NodeType.Neumann, null);
18     HashMap<NodeType, Function> mapNTF_p =
19         new HashMap<NodeType, Function>();
20     mapNTF_p.put(NodeType.Dirichlet, new AbstractFunction("x","y")
21         {
22             @Override
23             public double value(Variable v) {
24                 ...
25             }
26     });
27     mapNTF_p.put(NodeType.Neumann, null);
28     mesh.markBorderNode(new ObjIndex(1,2),mapNTF_uv);
29     // or
30     //mesh.markBorderNode(1,mapNTF_uv);
31     //mesh.markBorderNode(2,mapNTF_uv);
32     mesh.markBorderNode(3,mapNTF_p);
33
34     for(int i=1;i<=eList.size();i++) {
35         System.out.println(i+"_" + eList.at(i));
36     }

```

The output is:

```

1  1  GE1( 9_ 113_ 10_ 232_ 233_ 234_ )
2  2  GE2( 4_ 56_ 64_ 235_ 236_ 237_ )
3  3  GE3( 3_ 38_ 39_ 238_ 239_ 240_ )
4  4  GE4( 1_ 63_ 9_ 241_ 242_ 243_ )
5  5  GE5( 61_ 5_ 114_ 244_ 245_ 246_ )
6  ...
7  1  GE1( 9DDN 113III 10DDN 232III 233III 234DDN )
8  2  GE2( 4DDD 56DDN 64NND 235DDN 236III 237NND )
9  3  GE3( 3DDN 38DDN 39DDN 238DDN 239III 240DDN )
10 4  GE4( 1DDN 63DDN 9DDN 241DDN 242III 243DDN )
11 5  GE5( 61DDN 5DDN 114III 244DDN 245III 246III )
12 ...

```

We can see that before marking the boundary types, there is a “_” that following the node numbers in the brackets and after marking the boundary types, there are capital letters following the node numbers. These letters are abbreviation of the node types

- I: Inner ndoe
- D: Dirichlet node
- N: Neumann node
- R: Robin node

3.1.4 Create Element Objects

We have talked about **Mesh** class. The objects of **Node** and **Element** are contained in **Mesh** object. They are created when you read in a grid file by using the class **MeshReader**. If we look into the code of class **MeshReader**, we will see how these objects are created. By knowing these, you can read the grid file that has a format **FuturEye** doesn't support and create your own element objects.

```
1 Node node = new Node(index, x, y, z);
```

This line of code creates a node object with global index and 3D coordinate specified. When you have all the nodes that are needed to form an finite element you wanted, you can create an **Element** object like this:

```
1 NodeList list = new NodeList();
2 list.add(node1);
3 list.add(node2);
4 list.add(node3);
5 Element ele = new Element(list);
```

What you need is just a list of nodes in the element. Of cause, there are assumptions for the sequence of the list of nodes. Fig.5 shows a part of a grid. The numbers surrounded with small rectangles are global node indices of the grid. Take element (41,79,82) for example, this element has global nodes 41,79 and 82. The global indices are created along with nodes themselves. If we focus on the element itself, these nodes also have their own local indices. (see Fig.5 (41 → 1, 79 → 2, 82 → 3))

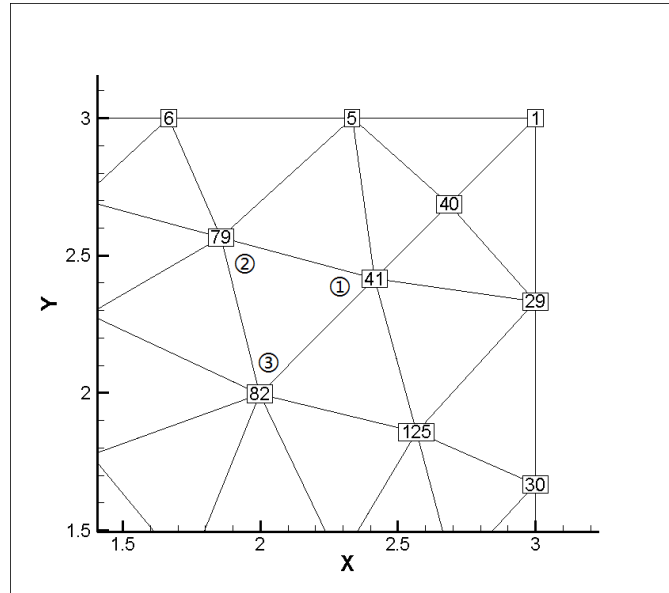


Figure 5: Global nodes and local nodes on elements

Local node indices are important in creating of an element object. Sequence of local node indices is the same as the sequence of parameter of the constructor of class **Element**.

Different sequence and nodes dimension and total nodes number mean different element types. We summarize these in the following tables:

Table 1: Local indices of 1D elements

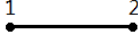
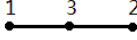
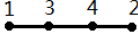
Linear	Quadratic	Cubic
		

Table 2: Local indices of 2D elements

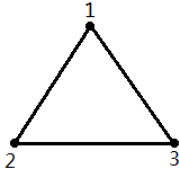
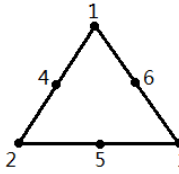
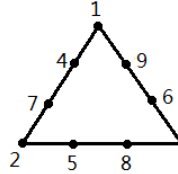
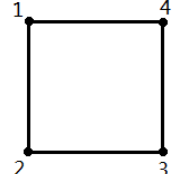
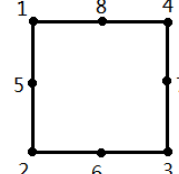
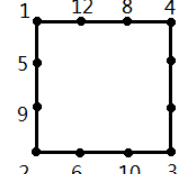
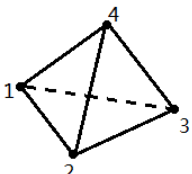
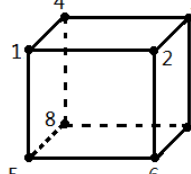
Linear	Quadratic	Cubic
		
		

Table 3: Local indices of 3D elements

Linear tetrahedron	Linear hexahedron
	

Another way to create complicated element is passing an object of **GeoEntity0D** to the constructor of **Element**. The class **GeoEntity0D** is the super class of template classes **GeoEntity1D**, **GeoEntity2D** and **GeoEntity3D**.

Using these classes, any complicated element can be created. Actually, the above method with local indices assumption creates an element also needs to create the object that extends from the super type **GeoEntity0D**. The structure of these classes are described in the following items:

- **GeoEntity0D**: contains all the vertices of a geometry entity. The instance of this class presents the two ends of an edge, the vertices of a face or the vertices of a volume.
- **GeoEntity1D**: presents an edge(1D) of a geometry entity and contains all the nodes on the edge (except the two nodes at the ends). It also contains the two ends (vertices) inherited from super class **GeoEntity0D**.
- **GeoEntity2D**: presents a face(2D) of a geometry entity and contains all the nodes on the face (except the vertices) and contains additionally the edges of the face. It also contains the vertices inherited from super class **GeoEntity0D**.
- **GeoEntity3D**: presents a volume(3D) of a geometry entity and contains all the nodes in the volume (except the vertices) and contains additionally the faces of the volume. It also contains the vertices inherited from super class **GeoEntity0D**.

Let's take this element for example,

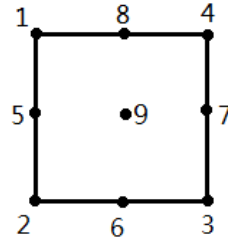


Figure 6: *Creating element with GeoEntity*

the following code segments create an element of the above type:

```

1  NodeList nodes = new NodeList();
2  nodes.add(new Node(1, 0.0,1.0));
3  nodes.add(new Node(2, 0.0,0.0));
4  nodes.add(new Node(3, 1.0,0.0));
5  nodes.add(new Node(4, 1.0,1.0));
6  nodes.add(new Node(5, 0.0,0.5));
7  nodes.add(new Node(6, 0.5,0.0));
8  nodes.add(new Node(7, 1.0,0.5));
9  nodes.add(new Node(8, 0.5,1.0));
10 nodes.add(new Node(9, 0.5,0.5));
11
12 Element e = new Element();
13 GeoEntity2D<EdgeLocal,NodeLocal> face =
14     new GeoEntity2D<EdgeLocal,NodeLocal>();
15 //vertices of face: 1,2,3,4
16 for(int i=1;i<=4;i++)

```



```

17         face.addVertex(new Vertex(i,new NodeLocal(i,nodes.at(i))))
18         ;
19
20     int[] idxLoop = {1,2,3,4,1};
21     int[] edgeIdx = {5,6,7,8};
22     for(int i=0;i<4;i++) {
23         EdgeLocal el = new EdgeLocal(i,e);
24         int idxBeg = idxLoop[i];
25         int idxEnd = idxLoop[i+1];
26         //vertices of edge: idxBeg,idxEnd
27         el.addVertex(new Vertex(idxBeg,
28             new NodeLocal(idxBeg,nodes.at(idxBeg))));
29         el.addVertex(new Vertex(idxEnd,
30             new NodeLocal(idxEnd,nodes.at(idxEnd))));
31         //node on edge
32         el.addEdgeNode(new NodeLocal(edgeIdx[i],nodes.at(edgeIdx[i
33             ])));
34         //edge on face
35         face.addEdge(el);
36     }
37     //node on face
38     face.addFaceNode(new NodeLocal(9,nodes.at(9)));
39     //create the element
40     e.setGeoEntity(face);
41     //output info
42     System.out.println(e);
43     for(int i=1;i<=e.nodes.size();i++)
44         System.out.println(e.nodes.at(i));

```

The output is:

```

1 GE( 1_ 2_ 3_ 4_ 5_ 6_ 7_ 8_ 9_ )
2 GN1( 0.0 1.0 )
3 GN2( 0.0 0.0 )
4 GN3( 1.0 0.0 )
5 GN4( 1.0 1.0 )
6 GN5( 0.0 0.5 )
7 GN6( 0.5 0.0 )
8 GN7( 1.0 0.5 )
9 GN8( 0.5 1.0 )
10 GN9( 0.5 0.5 )

```

3.2 Shape Functions and Mappings

Finite element basis functions are constructed by combining shape functions on neighboring elements. There are many shape function classes defined in **FuturEye**. Usually, users do not need to define shape functions by themselves. However, it's easy to implement new type of shape functions by implementing the interfaces defined for shape functions in **FuturEye**.

A shape function in **FuturEye** should implement two interfaces. For scalar valued shape functions, interfaces **ShapeFunction** and **Function** must be implemented. For vector valued shape functions, interfaces **ShapeFunction** and **VectorFunction** must be implemented. There are two interfaces **ScalarShapeFunction** and **VectorShapeFunction**

that defined in package `edu.uta.futureye.function.intf` which extend the interfaces that a shape function need to implement. It will be easier and more clear to use these two interfaces instead of the previous ones.

3.2.1 Create Shape Functions

Let's explain how to create a shape function by examples. In mathematical context, 3 nodes linear shape functions defined on a 2D triangle in local coordinates (or area/barycentric coordinates) (r, s, t) are:

$$(3.2.1) \quad N_1(r, s, t) = r$$

$$(3.2.2) \quad N_2(r, s, t) = s$$

$$(3.2.3) \quad N_3(r, s, t) = t$$

$$(3.2.4) \quad r + s + t = 1$$

Any function u can be approximated linearly on the element in local coordinates:

$$(3.2.5) \quad u \approx U = u_1 r + u_2 s + u_3 t$$

Where u_1, u_2 and u_3 are values of u defined on nodes of the triangle. In many application problems, the derivatives of u with respect to the global Cartesian coordinates are needed. It can be achieved by coordinates transform from local coordinates to global coordinates. The relation is given by

$$(3.2.6) \quad x = x_1 r + x_2 s + x_3 t$$

$$(3.2.7) \quad y = y_1 r + y_2 s + y_3 t$$

$$(3.2.8) \quad r + s + t = 1$$

where $(x_i, y_i), i = 1, 2, 3$ are nodes coordinates of the triangle. This relation can be derived intuitively if we think the global coordinates x and y as functions of r, s and t just like u in (3.2.5). For example, $u(1, 0, 0) = u_1, u(0, 1, 0) = u_2, u(0, 0, 1) = u_3$ correspond to $x(1, 0, 0) = x_1, x(0, 1, 0) = x_2, x(0, 0, 1) = x_3$. Now, considering the derivatives of u with respect to the global Cartesian coordinates.

$$u_x(r, s, t) \approx U_x(r, s, t) = u_r r_x + u_s s_x + u_t t_x = u_1 r_x + u_2 s_x + u_3 t_x$$

$$u_y(r, s, t) \approx U_y(r, s, t) = u_r r_y + u_s s_y + u_t t_y = u_1 r_y + u_2 s_y + u_3 t_y$$

The unknown items in the above equations are r_x, s_x, t_x, r_y, s_y and t_y .

A final step that concerned with shape function in FEM is the numerical integration in local coordinates. It is performed in order to get the algebra systems. So, arrived at here, we can see that a 3 nodes linear shape function defined on a triangle element should be a composite function and must provides the following minimum required abilities:

- It can associate with a specified physical element
- It can be evaluated with respect to variable r, s, t
- The derivative with respect to variable x, y can be obtained and evaluated with respect to r, s, t

There are 3 shape functions on a triangle element, but we don't want to define 3 classes for each shape function. What we do here is define only one class and pass a function index (e.g. 1,2,3) to the class constructor to create 3 shape function objects. In order to create composite functions, we need the outer functions and a map of outer variables to inner functions. Of course, to create real composite functions is not necessary according to the minimum abilities requirements listed above. At first, we give out the way of creating real composite functions.

The local coordinates of shape functions have 3 variables, but there are only 2 variables are free and the third one depends on the other two variables. For convenience, we can still create the shape function with three outer variables r, s, t . To do that, we need to define a special outer function class that keeps the relation $r + s + t = 1$. For example, inner class **SF123** which contained in class **SFLinearLocal2D** represents shape functions $N1 = r, N2 = s, N3 = t$. The following simplified code segment from **SFLinearLocal2D** shows the basic idea

```

1 public class SFLinearLocal2D extends AbstractFunction
2                               implements ScalarShapeFunction {
3     protected int funIndex;
4     ...
5     class SF123 extends AbstractFunction {
6         public SF123() {
7             super(SFLinearLocal2D.this.varNames);
8         }
9         @Override
10        public Function _d(String var) {
11            if(varNames.get(funIndex).equals(var)) {
12                //d(N1)/dr = 1.0
13                //d(N2)/ds = 1.0
14                //d(N3)/dt = 1.0
15                return new FC(1.0);
16            } else if(funIndex == 2){
17                //N3 = r = 1 - s - t, not free variable
18                //d(N3)/ds = -1.0
19                //d(N3)/dt = -1.0
20                return new FC(-1.0);
21            } else {
22                return new FC(0.0);
23            }
24        }
25        @Override
26        public double value(Variable v) {
27            return v.get(varNames.get(funIndex));
28        }
29        public String toString() {
30            return varNames.get(funIndex);
31        }
32    }
33    ...
34    public void SFLinearLocal2D(int funID) {
35        funIndex = funID - 1;
36        //3 variable names
37        varNames.add("r");
38        varNames.add("s");
39        varNames.add("t");
40        ...
41        //Composite function
42        innerVarNames = new ObjList<String>("x","y");
43        Map<String, Function> fInners = new HashMap<String,
44            Function>();
45        final String varName = varNames.get(funIndex);

```

```

45         fInners.put(varName,
46             new AbstractFunction(innerVarNames.toList()) {
47                 ...
48             }
49     });
50     funOuter = new SF123();
51     funCompose = funOuter.compose(fInners);
52 }
53 ...
54 public String toString() {
55     String varName = varNames.get(funIndex);
56     return "N"+(funIndex+1)+"("+varName+"(x,y))="+
57         funOuter.toString();
58 }
59 }

```

When we created the 3 shape function objects and print them out

```

1     SFLinearLocal2D[] shapeFun = new SFLinearLocal2D[3];
2     shapeFun[0] = new SFLinearLocal2D(1);
3     shapeFun[1] = new SFLinearLocal2D(2);
4     shapeFun[2] = new SFLinearLocal2D(3);
5     System.out.println(shapeFun[0]);
6     System.out.println(shapeFun[1]);
7     System.out.println(shapeFun[2]);

```

we will get the result

```

1     N1( r(x,y) )=r
2     N2( s(x,y) )=s
3     N3( t(x,y) )=t

```

Another way to create shape functions that just have 2 free variables, can be implemented like this:

```

1 public class SFLinearLocal2DRS extends AbstractFunction
2                               implements ScalarShapeFunction {
3     protected int funIndex;
4     ...
5     public void SFLinearLocal2DRS(int funID) {
6         funIndex = funID - 1;
7         //Just 2 free variables
8         varNames.add("r");
9         varNames.add("s");
10        ...
11        //Composite function
12        innerVarNames = new ObjList<String>("x","y");
13        Map<String, Function> fInners = new HashMap<String,
14            Function>();
15        for(final String varName : varNames) {
16            fInners.put(varName,
17                new AbstractFunction(innerVarNames.toList()) {
18                    ...
19                });
20        }

```

```

20 //Construct shape functions:
21 //r,s are free variables, t = 1 - r - s
22 if(funIndex == 0)
23     funOuter = R; //N1=r
24 else if(funIndex == 1)
25     funOuter = S; //N2=s
26 else
27     funOuter = C1.S(R).S(S); //N3=1-r-s, (N3=t)
28     funOuter.setVarNames(varNames);
29     funCompose = funOuter.compose(fInners).M(FC.c(this.coef));
30 }
31 ...
32 public String toString() {
33     return "N"+(funIndex+1)+"(r,s)="+funOuter.toString();
34 }
35 }

```

The result will be

```

1 N1(r,s)=r
2 N2(r,s)=s
3 N3(r,s)=1.0 - r - s

```

The difference of the two ways of constructing shape functions is that in the second way the variable t is replaced explicitly by $1 - r - s$, while in the first way this replacement is implemented implicitly in class **SF123**.

Now, we come to the inner map of composite functions. The derivatives of r, s, t with respect to x, y should be returned by the inner function. From relation (3.2.6)-(3.2.8), replace t with $1 - r - s$, we can get the Jacobian

$$\det(J_e) = \begin{vmatrix} x_r & x_s \\ y_r & y_s \end{vmatrix} = \begin{vmatrix} x_1 - x_3 & x_2 - x_3 \\ y_1 - y_3 & y_2 - y_3 \end{vmatrix}$$

where

$$J_e = \begin{pmatrix} x_r & x_s \\ y_r & y_s \end{pmatrix}$$

is the Jacobian matrix.

By inverting the mapping (3.2.6)-(3.2.8) or by using the chain rule:

$$U_x(r(x, y), s(x, y)) = U_r r_x + U_s s_x$$

$$U_y(r(x, y), s(x, y)) = U_r r_y + U_s s_y$$

and let $U(r, s) = x(s, t)$, we have

$$1 = x_r r_x + x_s s_x$$

$$0 = x_r r_y + x_s s_y$$

and let $U(r, s) = y(s, t)$, we have

$$0 = y_r r_x + y_s s_x$$

$$1 = y_r r_y + y_s s_y$$

Rewrite the above 4 equations in matrix form, we have

$$\begin{pmatrix} x_r & x_s \\ y_r & y_s \end{pmatrix} \begin{pmatrix} r_x & s_x \\ r_y & s_y \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$$

We may show that

$$(3.2.9) \quad \begin{pmatrix} r_x & s_x \\ r_y & s_y \end{pmatrix} = \begin{pmatrix} x_r & x_s \\ y_r & y_s \end{pmatrix}^{-1} = J_e^{-1}$$

Solve for the inverse of J_e

$$(3.2.10) \quad J_e^{-1} = \frac{1}{\det(J_e)} \begin{pmatrix} y_s & -y_r \\ -x_s & x_r \end{pmatrix}$$

we get

$$(3.2.11) \quad r_x = \frac{y_s}{\det(J_e)} = \frac{y_2 - y_3}{\begin{vmatrix} x_1 - x_3 & x_2 - x_3 \\ y_1 - y_3 & y_2 - y_3 \end{vmatrix}}$$

$$(3.2.12) \quad r_y = -\frac{x_s}{\det(J_e)} = -\frac{x_2 - x_3}{\begin{vmatrix} x_1 - x_3 & x_2 - x_3 \\ y_1 - y_3 & y_2 - y_3 \end{vmatrix}}$$

$$(3.2.13) \quad s_x = -\frac{y_r}{\det(J_e)} = -\frac{y_1 - y_3}{\begin{vmatrix} x_1 - x_3 & x_2 - x_3 \\ y_1 - y_3 & y_2 - y_3 \end{vmatrix}}$$

$$(3.2.14) \quad s_y = \frac{x_r}{\det(J_e)} = \frac{x_1 - x_3}{\begin{vmatrix} x_1 - x_3 & x_2 - x_3 \\ y_1 - y_3 & y_2 - y_3 \end{vmatrix}}$$

The following code segment gives the definition of the inner map of composite functions in the second way described above. The first way of definition of the inner map is similar.

```

1 public class SFLinearLocal2DRS extends AbstractFunction
2                               implements ScalarShapeFunction {
3     protected int funIndex;
4     private Function funCompose = null;
5     private Function funOuter = null;
6     protected ObjList<String> innerVarNames = null;
7
8     protected Element e = null;
9     private double jac = 0.0;
10    private double[] x = new double[3];
11    private double[] y = new double[3];
12
13    public void SFLinearLocal2DRS(int funID) {
14        //Just 2 free variables
15        varNames.add("r");
16        varNames.add("s");
17        ...
18        //Composite function
19        innerVarNames = new ObjList<String>("x","y");
20        Map<String, Function> fInners = new HashMap<String,
21        Function>();
22        for(final String varName : varNames) {
23            fInners.put(varName,
24                new AbstractFunction(innerVarNames.toList()) {
25                public Function _d(String var) {
26                    if(varName.equals("r")) {

```

```

26         if(var.equals("x"))
27             return new FC( (y[1]-y[2]) / jac);
28         if(var.equals("y"))
29             return new FC( (x[2]-x[1]) / jac);
30     } else if(varName.equals("s")) {
31         if(var.equals("x"))
32             return new FC( (y[2]-y[0]) / jac);
33         if(var.equals("y"))
34             return new FC( (x[0]-x[2]) / jac);
35     } else
36         throw new FuturEyeException(
37             "\nERROR:\n└varName="+varName);
38     return null;
39 }
40 @Override
41 public double value(Variable v) {
42     throw new FuturEyeException(
43         "\nERROR:\n└Not└supported└evaluate:└"+v);
44 }
45 });
46 }
47 //Construct shape functions:
48 //r,s are free variables, t = 1 - r - s
49 if(funIndex == 0)
50     funOuter = new FX("r"); //N1=r
51 else if(funIndex == 1)
52     funOuter = new FX("s"); //N2=s
53 else
54     funOuter = FC.cl.S(FX.fr).S(FX.fs); //N3=1-r-s, (N3=t)
55 funOuter.setVarNames(varNames);
56 funCompose = funOuter.compose(fInners).M(FC.c(this.coef));
57 }
58 @Override
59 public void assignElement(Element e) {
60     this.e = e;
61     VertexList vList = e.vertices();
62     x[0] = vList.at(1).coord(1);
63     x[1] = vList.at(2).coord(1);
64     x[2] = vList.at(3).coord(1);
65     y[0] = vList.at(1).coord(2);
66     y[1] = vList.at(2).coord(2);
67     y[2] = vList.at(3).coord(2);
68     jac = (x[0]-x[2])*(y[1]-y[2])-(x[1]-x[2])*(y[0]-y[2]);
69 }

```

The last thing that a shape function should provides is the implementation of function

```
1 public ScalarShapeFunction restrictTo(int funIndex)
```

This function gives the shape functions in lower dimension when it is restricted to one of the element borders (edges or faces). In our implementation, instead of really restrict the shape function to the borders, we use directly the lower dimension shape functions. For example:

```

1 public class SFLinearLocal2DRS extends AbstractFunction
2                               implements ScalarShapeFunction {
3     ...
4     ScalarShapeFunction sf1d1 = new SFLinearLocal1D(1);
5     ScalarShapeFunction sf1d2 = new SFLinearLocal1D(2);
6     @Override
7     public ScalarShapeFunction restrictTo(int funIndex) {
8         if(funIndex == 1) return sf1d1;
9         else return sf1d2;
10    }
11    ...
12 }

```

By this way, the border integral can be carried out easily.

3.2.2 3D Shape Functions

Any function u can be approximated by tri-linearly shape function on hexahedron element in local coordinates

$$(3.2.15) \quad u \approx U = \sum_{i=1}^8 u_i N_i$$

where

$$(3.2.16) \quad N_i = \frac{1}{8}(1 + r_i r)(1 + s_i s)(1 + t_i t), i = 1, \dots, 8$$

$u_i, i = 1, \dots, 8$ are function values of $u(x, y, z)$ at vertices on real hexahedron element and $(r_i, s_i, t_i), i = 1, \dots, 8$ are local coordinates of vertices on reference hexahedron element.

By using the chain rule:

$$U_x(r(x, y, z), s(x, y, z), t(x, y, z)) = U_r r_x + U_s s_x + U_t t_x$$

$$U_y(r(x, y, z), s(x, y, z), t(x, y, z)) = U_r r_y + U_s s_y + U_t t_y$$

$$U_z(r(x, y, z), s(x, y, z), t(x, y, z)) = U_r r_z + U_s s_z + U_t t_z$$

and let $U(r, s, t) = x(r, s, t)$, we have

$$1 = x_r r_x + x_s s_x + x_t t_x$$

$$0 = x_r r_y + x_s s_y + x_t t_y$$

$$0 = x_r r_z + x_s s_z + x_t t_z$$

and let $U(r, s, t) = y(r, s, t)$, we have

$$0 = y_r r_x + y_s s_x + y_t t_x$$

$$1 = y_r r_y + y_s s_y + y_t t_y$$

$$0 = y_r r_z + y_s s_z + y_t t_z$$

and let $U(r, s, t) = z(r, s, t)$, we have

$$0 = z_r r_x + z_s s_x + z_t t_x$$

$$0 = z_r r_y + z_s s_y + z_t t_y$$

$$1 = z_r r_z + z_s s_z + z_t t_z$$

Rewrite the above 9 equations in matrix form, we have

$$\begin{pmatrix} x_r & x_s & x_t \\ y_r & y_s & y_t \\ z_r & z_s & z_t \end{pmatrix} \begin{pmatrix} r_x & s_x & t_x \\ r_y & s_y & t_y \\ r_z & s_z & t_z \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

We may show that

$$(3.2.17) \quad \begin{pmatrix} r_x & s_x & t_x \\ r_y & s_y & t_y \\ r_z & s_z & t_z \end{pmatrix} = \begin{pmatrix} x_r & x_s & x_t \\ y_r & y_s & y_t \\ z_r & z_s & z_t \end{pmatrix}^{-1} = J_e^{-1}$$

It's easy to know that the inverse matrix of a general 3*3 matrix A is

$$(3.2.18) \quad A^{-1} = \frac{1}{\det(A)} \begin{pmatrix} \begin{vmatrix} a_{22} & a_{23} \\ a_{32} & a_{33} \end{vmatrix} & \begin{vmatrix} a_{13} & a_{12} \\ a_{33} & a_{32} \end{vmatrix} & \begin{vmatrix} a_{12} & a_{13} \\ a_{22} & a_{23} \end{vmatrix} \\ \begin{vmatrix} a_{23} & a_{21} \\ a_{33} & a_{31} \end{vmatrix} & \begin{vmatrix} a_{11} & a_{13} \\ a_{31} & a_{33} \end{vmatrix} & \begin{vmatrix} a_{13} & a_{11} \\ a_{23} & a_{21} \end{vmatrix} \\ \begin{vmatrix} a_{21} & a_{22} \\ a_{31} & a_{32} \end{vmatrix} & \begin{vmatrix} a_{12} & a_{11} \\ a_{32} & a_{31} \end{vmatrix} & \begin{vmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{vmatrix} \end{pmatrix}$$

So, we have for example

$$(3.2.19) \quad r_x = \frac{\begin{vmatrix} a_{22} & a_{23} \\ a_{32} & a_{33} \end{vmatrix}}{\det(J_e)}$$

and others are similar.

3.2.3 Vector Valued Shape Functions

In the following, we will briefly describe how to create vector valued shape functions. For example, in solving the 2D Stokes problem, we may need quadratic shape functions for velocity and linear functions for pressure on a triangle element. For vector valued shape function, one should implements interface **VectorShapeFunction**.

```

1 public class QuadraticV_LinearP extends AbstractVectorFunction
2                                 implements VectorShapeFunction {
3     ...
4 }
```

Every component of the vector valued shape function actually is a object of type **ScalarShapeFunction**. So it becomes easier by using existing classes that already defined for scalar shape functions. Take the 2D Stokes problem for example, we have to create vector valued shape function (u_1, u_2, p) , where $u_i, i = 1, 2$ can be defined by class **SFQuadraticLocal2D** and p can be defined by class **SFLinearLocal2D**. We order shape function u_1 first, u_2 second and p last with local indices.

```

1 /**
2  * Velocity: Quadratic shape function
3  * 3
4  * / \
5  * / \
6  * 6 5
```

```

7  * /      \
8  * /      \
9  * 1--4-- 2
10 *
11 * NV = NV(r,s,t) = NV( r(x,y), s(x,y), t(x,y) )
12 * NV1 = (2*r-1)*r
13 * NV2 = (2*s-1)*s
14 * NV3 = (2*t-1)*t
15 * NV4 = 4*r*s
16 * NV5 = 4*s*t
17 * NV6 = 4*r*t
18 *
19 * Pressure: Linear shape function
20 * 3
21 * /      \
22 * /      \
23 * /      \
24 * /      \
25 * 1---- 2
26 *
27 * NP = NP(r,s,t) = NP( r(x,y), s(x,y), t(x,y) )
28 * NP1 = r
29 * NP2 = s
30 * NP3 = t
31 *
32 * 2D vector valued shape functions
33 * Ni = (v1,v2,p)', i=1,...,15
34 *
35 * N1 = (NV1, 0, 0)'
36 * N2 = (NV2, 0, 0)'
37 * N3 = (NV3, 0, 0)'
38 * N4 = (NV4, 0, 0)'
39 * N5 = (NV5, 0, 0)'
40 * N6 = (NV6, 0, 0)'
41 * N7 = (0, NV1, 0)'
42 * N8 = (0, NV2, 0)'
43 * N9 = (0, NV3, 0)'
44 * N10 = (0, NV4, 0)'
45 * N11 = (0, NV5, 0)'
46 * N12 = (0, NV6, 0)'
47 * N13 = (0, 0, NP1)'
48 * N14 = (0, 0, NP2)'
49 * N15 = (0, 0, NP3)'
50 *
51 */
52 public class QuadraticV_LinearP extends AbstractVectorFunction
53                                 implements VectorShapeFunction {
54     //(u1,u2,p)
55     protected SpaceVectorFunction sf = new SpaceVectorFunction(3);
56     protected int funIndex;
57     protected ObjList<String> innerVarNames = null;
58     ...
59     public QuadraticV_LinearP(int funID) {

```

```

60     dim = 3;
61     funIndex = funID - 1;
62     varNames.add("r");
63     varNames.add("s");
64     varNames.add("t");
65     innerVarNames = new ObjList<String>("x","y");
66     if(funIndex>=0 && funIndex<=5) {
67         sf.set(1, new SFQuadraticLocal2DFast(funIndex+1));
68         sf.set(2, new SF0(innerVarNames));
69         sf.set(3, new SF0(innerVarNames));
70     } else if(funIndex>=6 && funIndex<=11) {
71         sf.set(1, new SF0(innerVarNames));
72         sf.set(2, new SFQuadraticLocal2DFast(funIndex-5));
73         sf.set(3, new SF0(innerVarNames));
74     } else if(funIndex>=12 && funIndex<=14) {
75         sf.set(1, new SF0(innerVarNames));
76         sf.set(2, new SF0(innerVarNames));
77         sf.set(3, new SFLinearLocal2D(funIndex-11));
78     }
79 }
80 @Override
81 public Vector value(Variable v) {
82     return (Vector) this.sf.value(v);
83 }
84 @Override
85 public Function dot(VectorFunction b) {
86     return sf.dot(b);
87 }
88 ...
89 }

```

3.3 Degrees of Freedom

Assume domain Ω is partitioned by a finite element triangulation. Any function $u(\mathbf{x})$ defined in Ω can be approximated by its interpolant $U(\mathbf{x})$

$$u(\mathbf{x}) \approx U(\mathbf{x}) = \sum_{i=1}^N c_i \phi_i(\mathbf{x}), \mathbf{x} \in \Omega$$

where i is global indices of degrees of freedom (DOF), ϕ_i is the base function associated with DOF i , and c_i is the value to be determined. N is the total number of DOF. If we restrict $u(\mathbf{x})$ into an finite element $e \subset \Omega$, then

$$u(\mathbf{x})|_e \approx U(\mathbf{x})|_e = \sum_{j=1}^{N_e} c_j \psi_j(\mathbf{x}), \mathbf{x} \in e$$

where N_e is the total number of DOF in element e . Base function $\phi_i(\mathbf{x})$ becomes shape functions $\psi_j(\mathbf{x})$ after restricting to element e . In **FuturEye**, the items in the sum symbols above correspond to class *DOF*. Table 4 shows the relationship between the mathematical description above and members defined in class *DOF*.

Degrees of freedom objects can be associated to element objects. Specifically, DOF objects are allowed to associate with geometry entities such as node(or vertex), edge, face

Symbols in $\sum_{i=1}^N c_i \phi_i(\mathbf{x})$, $\sum_{j=1}^{N_e} c_j \psi_j(\mathbf{x})$	Corresponding members defined in class <i>DOF</i>	Remarks
i	int getGlobalIndex () void setGlobalIndex (int globalIndex)	Global index of DOF
j	int getLocalIndex () void setLocalIndex (int localIndex)	Local index of DOF
$\phi_j(\mathbf{x})$	ShapeFunction getSF () ScalarShapeFunction getSSF () VectorShapeFunction getVSF () void setShapeFunction (ShapeFunction sf)	Shape function associated with local DOF j
k in $\psi_{jk} \neq 0$, If $\vec{\psi}_j(\mathbf{x}) = \begin{pmatrix} \cdots \\ \psi_{jk} \\ \cdots \end{pmatrix}$	int getVVFComponent () void setVVFComponent (int nComponent)	Nontrivial component index of vector valued shape function

Table 4: Degrees of freedom and class **DOF**

Functions in class Element	Comments
void addNodeDOF (int localNodeIndex,DOF dof) DOFList getNodeDOFList (int localNodeIndex) DOFList getAllNodeDOFList () int getNodeDOFNumber ()	Associate DOFs with nodes
void addEdgeDOF (int localEdgeIndex,DOF dof) DOFList getEdgeDOFList (int localEdgeIndex) DOFList getAllEdgeDOFList () int getEdgeDOFNumber ()	Associate DOFs with edges, in 2D and 3D case only
void addFaceDOF (int localFaceIndex,DOF dof) DOFList getFaceDOFList (int localFaceIndex) DOFList getAllFaceDOFList () int getFaceDOFNumber ()	Associate DOFs with faces, in 3D case only
void addVolumeDOF (DOF dof) DOFList getVolumeDOFList () DOFList getAllVolumeDOFList () int getVolumeDOFNumber ()	Associate DOFs with ‘volumes’, here ‘volume’ means element itself, in 3D is nature, but in 1D,2D should be especially noted
DOFList getAllDOFList (DOFOrder order) int getAllDOFNumber ()	Get all DOFs in an element, ordered by order
DOFList getAllDOFListByVvfIndex (DOFOrder order,int vvfIndex)	For vector valued shape function, get the component with index vvfIndex in an element,ordered by order

Table 5: Association DOFs with mesh entities

or volume (interior of element) object through the following member functions define in class *Element*.

```
public void addNodeDOF(int localNodeIndex,DOF dof)
public void addEdgeDOF(int localEdgeIndex,DOF dof)
public void addFaceDOF(int localFaceIndex,DOF dof)
public void addVolumeDOF(DOF dof)
```

More than one DOF objects can be associated with one entity object. Member function

```
public DOFList getAllDOFList(DOFOrder order)
```

returns all DOF objects associated with an element object in the order specified by enum type *DOFOrder*. It is useful when one assembles global matrix and vector by looping over all DOFs on an element. Additionally, a dozen of member functions are provided for users to manage their own degrees of freedom objects. All these member functions are shown in table 5. This kind of design makes it possible that a vast amount of finite element types can be implemented base on the toolkit.

In applications, there are two ways to associate objects of class DOF to elements. First way is that users can direct associate DOFs to elements through calling functions that provided in class **Element** shown in table 5. This is the fundamental way to do the association. It can be used to associate any new types of finite element. The following code segment shows the association procedure for 2D linear triangle elements

```
1 //Create 2D linear triangle shape function
2 SFLinearLocal2D[] shapeFun = new SFLinearLocal2D[3];
3 for(int i=0;i<3;i++)
4     shapeFun[i] = new SFLinearLocal2D(i+1);
5
6 //Assign degrees of freedom(DOF) to element
7 for(int i=1;i<=mesh.getElementList().size();i++) {
8     Element e = mesh.getElementList().at(i);
9     for(int j=1;j<=e.nodes.size();j++) {
10        //Create degree of freedom(DOF) object
11        DOF dof = new DOF(j,e.nodes.at(j).globalIndex,shapeFun[j-1]);
12        e.addNodeDOF(j, dof);
13    }
14 }
```

Another way to associate DOF to element is much easier, there are several classes defined in package **edu.uta.futureye.lib.element**. They have done all the works just mentioned above. For example, the follow code segment do the same thing

```
1 //Use element library to assign degrees
2 //of freedom (DOF) to element
3 ElementList eList = mesh.getElementList();
4 FELinearTriangle linearTriangle = new FELinearTriangle();
5 for(int i=1;i<=eList.size();i++)
6     linearTriangle.assignTo(eList.at(i));
```

3.4 Weak Form Classes

In order to solve a PDE problem, the strong form of a PED have to be rewritten to a weak form in FEM. Different weak forms may be involved in different techniques for solving the same PDE problem with finite element method. Any class that implements

Table 6: Element library classes

Classes	Comments
FELinear1D	Linear 1D element
FEBilinearRectangle	Bilinear element on rectangle
FEBilinearRectangleRegular	Bilinear element on regular rectangle
FEBilinearRectangleVector	Bilinear vector valued element on rectangle
FEBilinearV_ConstantP	Bilinear velocity and constant pressure element on rectangle
FELinearTriangle	Linear element on triangle
FEQuadraticTriangle	Quadratic element on triangle
FELinearTriangleVector	Linear vector valued element on triangle
FEQuadraticV_ConstantP	Quadratic velocity and constant pressure element on triangle
FEQuadraticV_LinearP	Quadratic velocity and linear pressure element on triangle
FELinearTetrahedron	Linear element on tetrahedron
FETrilinearHexahedron	Trilinear element on hexahedron
FETrilinear_ConstantP	Trilinear velocity and constant pressure element on hexahedron

interface **WeakForm** will represents a weak form of a concrete or a class of PED problems. There are two approaches defined in interface **WeakForm**. In the first approach (common approach), you just provide the expression of left and right hand side of the integrand in the weak form. The remain work will be done by assembling classes in a standard way. In the second approach (fast approach), you should do all the things to compute the local stiffness matrix and load vector. Generally, the second approach is faster than the first approach. Because optimization of the algorithms is allowed in the second case.

There is another quick way to create weak forms other than creating a new class that implements interface **WeakForm**. Class **WeakFormBuilder** which is designed for this purpose provides a way that allows users only focus on the expression of the weak form itself only. We will discuss about how to use **WeakFormBuilder** later.

We fist begin with some concrete examples in the following subsections and then talk more about the interface **WeakForm**. Finally we talk about how to use class **WeakFormBuilder** to build weak forms quickly.

3.4.1 Elliptic Problem

Let Ω be a bounded Lipschitz domain with unit normal \mathbf{n} on the boundary $\Gamma = \Gamma_D \cup \Gamma_N$, where Γ_D and Γ_N are Dirichlet and Neumann boundary respectively. Given $f, g \in L^2(\Omega)$, $u_0 \in H^1(\Omega) \cap C(\bar{\Omega})$, $q \in L^2(\Gamma_N)$, $k, c, d \in L^2(\Omega)$, seek $u \in H^1(\Omega)$ such that

$$(3.4.1) \quad -\nabla(k\nabla u) + cu = f, \text{ in } \Omega$$

$$(3.4.2) \quad u = u_0, \text{ on } \Gamma_D$$

$$(3.4.3) \quad du + k \frac{\partial u}{\partial n} = g, \text{ on } \Gamma_N$$

The weak form of the above reads: Given $f, g \in L^2(\Omega)$, $u_0 \in H^1(\Omega) \cap C(\overline{\Omega})$, $k, c, d \in L^2(\Omega)$, seek $u \in H^1(\Omega)$ such that

$$(3.4.4) \quad (k \nabla u, \nabla v)_\Omega - (du, v)|_{\Gamma_N} + (cu, v)_\Omega = (f, v)_\Omega + (g, v)|_{\Gamma_N}$$

where $v \in H_0^1(\Omega)$ is arbitrary function.

We list below the definition of weak form class **WeakFormLaplace** for problem (3.4.4) by the first approach declared in class **WekForm** of creating weak forms. In order to decrease the coding load, we provide two abstract classes **AbstractScalarWeakForm** and **AbstractVectorWeakform**. A new class that represents a weak form can just extend the proper abstract class.

```

1 package edu.uta.futureye.lib.weakform;
2
3 import edu.uta.futureye.core.Element;
4 import edu.uta.futureye.function intf.Function;
5 import edu.uta.futureye.util.Utils;
6 import static edu.uta.futureye.function.operator.FMath.*;
7
8 public class WeakFormLaplace extends AbstractScalarWeakForm {
9     protected Function g_f = null;
10    protected Function g_k = null;
11    protected Function g_c = null;
12    protected Function g_q = null;
13    protected Function g_d = null;
14
15    //right hand side function (source term)
16    public void setF(Function f) {
17        this.g_f = f;
18    }
19
20    //Robin: d*u + k*u_n = q
21    public void setParam(Function k, Function c, Function q, Function d)
22    {
23        this.g_k = k;
24        this.g_c = c;
25        this.g_q = q;
26        this.g_d = d;
27    }
28
29    @Override
30    public Function leftHandSide(Element e, ItemType itemType) {
31        if(itemType==ItemType.Domain) {
32            //Integrand part of Weak Form on element e
33            Function integrand = null;
34            if(g_k == null) {
35                integrand = grad(u, u.innerVarNames()).dot(
36                    grad(v, v.innerVarNames()));
37            } else {
38                Function fk = Utils.interpolateOnElement(g_k, e);
39                Function fc = Utils.interpolateOnElement(g_c, e);
40                integrand = fk.M(
41                    grad(u, u.innerVarNames()).dot(
42                    grad(v, v.innerVarNames()))).A(

```

```

42         fc.M(u.M(v)));
43     }
44     return integrand;
45 }
46 else if(itemType==ItemType.Border) {
47     if(g_d != null) {
48         Element be = e;
49         Function fd = Utils.interpolateOnElement(g_d, be);
50         Function borderIntegrand = fd.M(u.M(v));
51         return borderIntegrand;
52     }
53 }
54 return null;
55 }
56
57 @Override
58 public Function rightHandSide(Element e, ItemType itemType) {
59     if(itemType==ItemType.Domain) {
60         Function ff = Utils.interpolateOnElement(g_f, e);
61         Function integrand = ff.M(v);
62         return integrand;
63     } else if(itemType==ItemType.Border) {
64         if(g_q != null) {
65             Element be = e;
66             Function fq = Utils.interpolateOnElement(g_q, be);
67             Function borderIntegrand = fq.M(v);
68             return borderIntegrand;
69         }
70     }
71     return null;
72 }
73 }

```

We can see that in function **leftHandSide**, $(k\nabla u, \nabla v)_\Omega + (cu, v)_\Omega$ in (3.4.4) corresponds to the following code segment

```

1     integrand = fk.M(
2         grad(u,u.innerVarNames()).dot(
3         grad(v,v.innerVarNames()))).A(
4         fc.M(u.M(v)));

```

and the border integration $(du, v)|_{\Gamma_N}$ in (3.4.4) corresponds to

```

1     Function borderIntegrand = fd.M(u.M(v));

```

In function **rightHandSide**, $(f, v)_\Omega$ in (3.4.4) corresponds to

```

1     Function integrand = ff.M(v);

```

and $(g, v)|_{\Gamma_N}$ in (3.4.4) corresponds to

```

1     Function borderIntegrand = fq.M(v);

```

Static method **Utils.interpolateOnElement(...)** can be used to interpolate any function on an finite element. All the parameters such as k, c, f, d, g must be interpolated on all

finite elements, because the standard integral progress that defined in the abstract classes **AbstractScalarWeakForm** needs a local coordinate system.

It should be noted that class **WeakFormLaplace** can be used to solve both 2D and 3D problems. There are also two classes **WeakFormLaplace2D** and **WeakFormLaplace3D** defined in package **edu.uta.futureye.lib.weakform** which are used to solve 2D and 3D problems respectively. The strongpoints of these two class are that optimization of the code has been implemented for both of the classes and the two approach of implementing interface **WeakForm** have also been implemented.

3.4.2 Mixed Laplace Problem

In this section, we will describe the second approach of creating weak forms defined in interface **WeakForm** and will show how to implement weak forms for vector valued problems. The following example is also an example of showing different weak forms in different techniques for the same PDE problems. Considering 2D case, problem (3.4.1) can be split into two equations

$$(3.4.5) \quad \nabla \cdot \mathbf{p} + f = 0, \text{ in } \Omega$$

$$(3.4.6) \quad \mathbf{p} = \nabla u, \text{ in } \Omega$$

for unknown $u \in H^1(\Omega)$ and $\mathbf{p} \in H(\text{div}, \Omega)$, where

$$H(\text{div}, \Omega) := \{\mathbf{q} \in L^2(\Omega)^2 : \nabla \cdot \mathbf{q} \in L^2(\Omega)\}$$

Here for simplicity, we just let $k = 1, c = 0$ in (3.4.1). The weak form of (3.4.5)(3.4.6) is seeking $q \in H_g(\text{div}, \Omega)$ and $u \in L^2(\Omega)$ such that

$$(3.4.7) \quad (\mathbf{p}, \mathbf{q})_\Omega + (u, \nabla \cdot \mathbf{q})_\Omega = (u_D, \mathbf{q} \cdot \mathbf{n})_{\Gamma_D}$$

$$(3.4.8) \quad (v, \nabla \cdot \mathbf{p})_\Omega = -(v, f)_\Omega$$

for all $q \in H_0(\text{div}, \Omega)$ and $v \in L^2(\Omega)$. Where where

$$H_0(\text{div}, \Omega) := \{\mathbf{q} \in H(\text{div}, \Omega) : \mathbf{q} \cdot \mathbf{n} = 0, \text{ on } \Gamma_N\}$$

$$H_g(\text{div}, \Omega) := \{\mathbf{q} \in H(\text{div}, \Omega) : \mathbf{q} \cdot \mathbf{n} = g, \text{ on } \Gamma_N\}$$

Here we consider Neumann boundary condition $\mathbf{q} \cdot \mathbf{n} = g, \text{ on } \Gamma_N$.

We choose an edge-oriented basis of RT_0 for solving (3.4.7),(3.4.8), then a linear system of the weak form can be got

$$(3.4.9) \quad \begin{pmatrix} B & C \\ C^T & 0 \end{pmatrix} \begin{pmatrix} \mathbf{p} \\ u \end{pmatrix} = \begin{pmatrix} 0 \\ b_f \end{pmatrix}$$

In this case, by using block matrix and block vector the assembling and post processing procedures can be simplified. The method

```
1 void assembleElement(Element e,
2 Matrix globalStiff, Vector globalLoad)
```

defined in weak form class **WeakFormMixedLaplace** should be implemented to assemble local stiffness matrix and vector into global stiffness matrix and vector respectively. But you need not to care about whether the matrix and the vector that passed in by function parameters are block ones or not block ones. Because classes implemented the interfaces **BlockMatrix** and **BlockVector** are designed to access their components by the ways of either globally or locally.

The following code gives a demo of how to implement the function **assembleElement** in weak form class. Assuming that the elements are all triangles for simplicity.

```

1 public void assembleElement(Element e,
2     Matrix globalStiff, Vector globalLoad){
3     DOFList edgeDOFs = e.getAllEdgeDOFList();
4     DOFList eleDOFs = e.getVolumeDOFList();
5     int nEdgeDOF = edgeDOFs.size();
6     int nElementDOF = eleDOFs.size();
7
8     BlockMatrix blockMat = (BlockMatrix)globalStiff;
9     BlockVector blockVec = (BlockVector)globalLoad;
10    //The type cast here above is not necessary in this case,
11    //but it gives a demo of possible applications that
12    //when the block matrix and vector are needed in
13    //assembling procedure.
14    //Matrix m11 = blockMat.getBlock(1, 1);
15    //Matrix m12 = blockMat.getBlock(1, 2);
16    //Matrix m21 = blockMat.getBlock(2, 1);
17    //Matrix m22 = blockMat.getBlock(2, 2);
18
19    //The Jacobin of element e must be updated first
20    e.updateJacobianLinear2D();
21    for(int i=1;i<=nEdgeDOF;i++) {
22        edgeDOFs.at(i).getVSF().assignElement(e);
23    }
24
25    //Loop for edge degrees of freedom for p
26    for(int j=1;j<=nEdgeDOF;j++) {
27        DOF dofV = edgeDOFs.at(j);
28        VectorShapeFunction vecV = dofV.getVSF();
29        //Loop for edge degrees of freedom for q
30        for(int i=1;i<=nEdgeDOF;i++) {
31            DOF dofU = edgeDOFs.at(i);
32            VectorShapeFunction vecU = dofU.getVSF();
33            //B = (p,q)_{\Omega}
34            Function integrandB = vecU.dot(vecV);
35            double val = F0Integrate.intOnTriangleRefElement(
36                integrandB.M(e.getJacobian()),4);
37            blockMat.add(dofU.getGlobalIndex(), dofV.
38                getGlobalIndex(),
39                val);
40        }
41    }
42    //Loop for area degrees of freedom for u
43    for(int k=1;k<=nElementDOF;k++) {
44        DOF dofE = eleDOFs.at(k);
45        //C = (u,\div{q})_{\Omega}
46        Function integrandC = div(vecV);
47        Function val = F0Integrate.intOnTriangleRefElement(
48            integrandC.M(e.getJacobian()),4);
49        blockMat.add(dofV.getGlobalIndex(), dofE.
50            getGlobalIndex(),
51            val);
52        //C' = (v,\div{p})_{\Omega}

```

```

50         blockMat.add(dofE.getGlobalIndex(), dofV.
51             getGlobalIndex(),
52             val);
53     }
54     //Loop for area degrees of freedom for (v,f)
55     for(int k=1;k<=nElementDOF;k++) {
56         DOF dofE = eleDOFs.at(k);
57         Function integrand = Utils.interpolateOnElement(g_f, e);
58         //bf = -(v,f)_{\Omega}
59         integrand = FC.c(-1.0).M(integrand);
60         double val = F0Integrate.intOnTriangleRefElement(
61             integrand.M(e.getJacobin()),4
62         );
63         blockVec.add(dofE.getGlobalIndex(), val);
64     }
65 }

```

3.4.3 Other Weak Forms

Convection-diffusion equation

$$(3.4.10) \quad \frac{\partial c}{\partial t} = \nabla \cdot (k \nabla c) - \mathbf{v} \cdot \nabla c + f$$

where $c = c(x, y, z, t)$ represents particles or energy (e.g. salt density, heat...) are transferred inside a physical system due to two processes: diffusion and convection. $k = k(x, y, z)$ is the diffusion coefficient. $\mathbf{v} = (v_1, v_2, v_3)'$ is the convection velocity vector. $f = f(x, y, z)$ is the source term.

The implicit Euler time discrete form of (3.4.10) reads: let

$$\frac{\partial c}{\partial t} = \frac{c_{n+1} - c_n}{\delta t}$$

where δt is time step size, then we have,

$$(3.4.11) \quad -\delta t \nabla \cdot (k \nabla c_{n+1}) + \delta t \mathbf{v} \cdot \nabla c_{n+1} + c_{n+1} = \delta t f + c_n$$

The weak formulation of the time discrete form reads: seek $u \in H^1(\Omega)$, such that, for any $w \in H_0^1(\Omega)$

$$(3.4.12) \quad \begin{aligned} & \delta t (k \nabla u, \nabla w) + \\ & \delta t (\mathbf{v} \cdot \nabla u, w) + \\ & (bu, w) = (\delta t f + c_n, w) \end{aligned}$$

where $u := c_{n+1}$. We can write explicitly

$$(3.4.13) \quad \begin{aligned} & \delta t ((ku_x, w_x) + (ku_y, w_y) + (ku_z, w_z)) + \\ & \delta t ((v_1 u_x, w) + (v_2 u_y, w) + (v_3 u_z, w)) + \\ & (bu, w) = (\delta t f + c_n, w) \end{aligned}$$

where $b = 1$, or $b = b(x, y, z)$, when left hand side of equation (3.4.10) has the term ac , where $a(x, y, z) = b(x, y, z) - 1$.

Dirichlet and Neumann (or Robin) boundary conditions

$$c = c_0, \text{ on } \Gamma_D$$

$$dc + k \frac{\partial c}{\partial n} = g, \text{ on } \Gamma_N$$

The following weak form class gives one step computation of c from c_n to c_{n+1} .

```

1  public class WeakFormConvectionDiffusion extends
2  AbstractScalarWeakForm {
3  ...
4  public Function leftHandSide(Element e, ItemType itemType) {
5      if(itemType==ItemType.Domain) {
6          //Interplate functions on element e
7          Function fk = Utils.interpolateOnElement(g_k,e);
8          Function fb = Utils.interpolateOnElement(g_b,e);
9          VectorFunction fv = new SpaceVectorFunction(g_v.getDim());
10         for(int dim=1;dim<=g_v.getDim();dim++)
11             fv.set(dim,
12                 Utils.interpolateOnElement(g_v.get(dim),e));
13
14         //Dt*(k*\nabla{u},\nabla{w}) +
15         //Dt*( (v1*u_x,w)+(v2*u_y,w)+(v3*u_z,w) ) +
16         //b*(u,w)
17         Function integrand = null;
18         integrand = fk.M(
19             FMath.grad(u,u.innerVarNames()).
20                 dot(
21                     FMath.grad(v,v.innerVarNames())
22                 ).A(
23                     fv.dot(FMath.grad(u,u.innerVarNames()))
24                 ).M(FC.c(Dt)).A(
25                     fb.M(u).M(v)
26                 );
27         return integrand;
28     }
29     else if(itemType==ItemType.Border) {
30         if(g_d != null) {
31             Element be = e;
32             Function fd = Utils.interpolateOnElement(g_d, be);
33             Function borderIntegrand = fd.M(u.M(v));
34             return borderIntegrand;
35         }
36     }
37     return null;
38 }
39
40 public Function rightHandSide(Element e, ItemType itemType) {
41     if(itemType==ItemType.Domain) {
42         //(Dt*f + c_n,w)
43         Function ff = Utils.interpolateOnElement(g_f, e);
44         Function fcn = Utils.interpolateOnElement(g_cn, e);
45         Function integrand = ff.M(FC.c(Dt)).A(fcn).M(v);
46         return integrand;
47     } else if(itemType==ItemType.Border) {
48         Element be = e;
49         Function fq = Utils.interpolateOnElement(g_g, be);

```

```

50         Function borderIntegrand = fq.M(v);
51         return borderIntegrand;
52     }
53     return null;
54 }
55 }

```

Stokes equation Let's consider Stokes equations which are basic components of more complicated fluent problems(e.g. Navier-Stokes equations). The unknown of this problem is a vector valued function. So, this example can be a good template for other vector valued problems. For simplicity, considering 2D case:

$$(3.4.14) \quad -\nabla(k\nabla \cdot \mathbf{u}) + \nabla p = \mathbf{f}$$

$$(3.4.15) \quad \nabla \cdot \mathbf{u} = 0$$

where $\mathbf{u} = (u_1, u_2)$ is velocity vector field, $\mathbf{f} = (f_1, f_2)$ is body force. The weak formulation reads:

find $\mathbf{u} \in H_0^1(\text{div}, \Omega)$, $p \in L^2(\Omega)$ such that, for all $\mathbf{v} \in H_0^1(\text{div}, \Omega)$, $q \in L^2(\Omega)$

$$(3.4.16) \quad (\nabla \mathbf{v}, k\nabla \mathbf{u}) - (\nabla \cdot \mathbf{v}, p) + (q, \nabla \cdot \mathbf{u}) = (\mathbf{v}, \mathbf{f})$$

Write explicitly

$$(3.4.17) \quad \begin{aligned} & (v_{1x}, ku_{1x}) + (v_{1y}, ku_{1y}) + (v_{2x}, ku_{2x}) + (v_{2y}, ku_{2y}) - \\ & \quad (v_{1x} + v_{2y}, p) + \\ & \quad (q, u_{1x} + u_{2y}) \\ & = (v_1, f_1) + (v_2, f_2) \end{aligned}$$

The definition of the corresponding weak form class **WeakFormStokes** is listed below

```

1 package edu.uta.futureye.lib.weakform;
2
3 import edu.uta.futureye.algebra.intf.Vector;
4 import edu.uta.futureye.core.Edge;
5 import edu.uta.futureye.core.Element;
6 import edu.uta.futureye.function.intf.*;
7 import edu.uta.futureye.util.Utills;
8 import static edu.uta.futureye.function.operator.FMath.*;
9
10 public class WeakFormStokes extends AbstractVectorWeakForm {
11     protected VectorFunction g_f = null;
12     protected Function g_k = null;
13     //Robin: k*u_n + d*u - p\vec{n} = 0
14     protected VectorFunction g_d = null;
15
16     public void setF(VectorFunction f) {
17         this.g_f = f;
18     }
19
20     public void setParam(Function k) {
21         this.g_k = k;
22     }
23
24     //Robin: k*u_n + d*u - p\vec{n} = 0
25     public void setRobin(VectorFunction d) {

```

```

26         this.g_d = d;
27     }
28
29     @Override
30     public Function leftHandSide(Element e, ItemType itemType) {
31         if(itemType==ItemType.Domain) {
32             //Integrand part of Weak Form on element e
33             Function integrand = null;
34             Function fk = Utils.interpolateOnElement(g_k,e);
35             Function u1 = u.get(1), u2 = u.get(2), p = u.get(3);
36             Function v1 = v.get(1), v2 = v.get(2), q = v.get(3);
37             //(v1_x,k*u1_x) + (v1_y,k*u1_y) +
38             //(v2_x,k*u2_x) + (v2_y,k*u2_y) -
39             //(v1_x+v2_y,p) + (q,u1_x+u2_y)
40             Function uv1 = grad(u1,"x","y").dot( grad(v1,"x","y") );
41             Function uv2 = grad(u2,"x","y").dot( grad(v2,"x","y") );
42             Function div_v = v1._d("x").A(v2._d("y"));
43             Function div_u = u1._d("x").A(u2._d("y"));
44             integrand = fk.M( uv1.A(uv2) ).S( div_v.M(p) ).A( div_u.M(
45                 q ) );
46             return integrand;
47         }
48         else if(itemType==ItemType.Border) {
49             if(g_d != null) {
50                 Element be = e;
51                 Function fd1 = Utils.interpolateOnElement(g_d.get(1), be);
52                 Function fd2 = Utils.interpolateOnElement(g_d.get(2), be);
53                 Function u1 = u.get(1), u2 = u.get(2);
54                 Function v1 = v.get(1), v2 = v.get(2);
55                 //Robin: - k*u_n = d*u - p\vec{n}
56                 //d1*u1 + d2*u2
57                 Function borderIntegrand = fd1.M(u1.M(v1)).A(fd2.M(u2.M(v2
58                     ))) );
59                 return borderIntegrand;
60             }
61         }
62         return null;
63     }
64
65     @Override
66     public Function rightHandSide(Element e, ItemType itemType) {
67         if(itemType==ItemType.Domain) {
68             Function f1 = Utils.interpolateOnElement(g_f.get(1), e);
69             Function f2 = Utils.interpolateOnElement(g_f.get(2), e);
70             Function v1 = v.get(1);
71             Function v2 = v.get(2);
72             //(v1*f1+v2*f2)
73             Function integrand = v1.M(f1).A(v2.M(f2));
74             return integrand;
75         }
76         else if(itemType==ItemType.Border) {
77             Element be = e;
78             Function v1 = v.get(1), v2 = v.get(2), p = v.get(3);
79             //Robin: - k*u_n = d*u - p\vec{n}

```

```

77      //- p\vec{n} = - p*n1*v1 - p*n2*v2
78      Edge edge = (Edge)be.getGeoEntity();
79      Vector n = edge.getNormVector();
80      Function n1 = C(-1.0*n.get(1));
81      Function n2 = C(-1.0*n.get(2));
82      Function borderIntegrand = p.M(v1.M(n1)).A(p.M(v2.M(n2)));
83      return borderIntegrand;
84  }
85  return null;
86  }
87
88  public boolean isVVFComponentCoupled(int nComponent1, int
      nComponent2) {
89      if(nComponent1 == nComponent2) return true;
90      else if(nComponent1 == 3 || nComponent2 == 3) return true;
91      else return false;
92  }
93  }

```

3.4.4 Interface WeakForm

A class that implements interface **WeakForm** is not necessary to implement all the methods defined in this interface. That is to say either the common approach or the fast approach be implemented. We have shown some examples in the above subsections about how to create weak form classes that extends from **AbstractScalarWeakform** or **AbstractVectorWeakform**. Below is the definition of interface **WeakForm**.

```

1  public interface WeakForm {
2      //1st: Common approach
3      void setDOF(DOF trialDOF, DOF testDOF);
4      DOF getTrialDOF();
5      DOF getTestDOF();
6      Function leftHandSide(Element e, ItemType itemType);
7      Function rightHandSide(Element e, ItemType itemType);
8
9      //2nd: Fast approach
10     void assembleElement(Element e, Matrix globalStiff, Vector
        globalLoad);
11
12     //Integrate on element
13     double integrate(Element e, Function fun);
14
15     //For vector valued problems only
16     boolean isVVFComponentCoupled(int nComponent1, int nComponent2);
17
18 }

```

If one want to implement this interface directly without extends from the abstract classes he should also implement other member functions that are not mentioned in the previous examples which are define in interface **WeakForm**. Function **setDOF(...)** is called by the assembler before calling **leftHandSide(...)** and **rightHandSide(...)**, so that the information of DOF can be obtained through functions **getTrialDOF(...)** and **getTestDOF(...)** when you are writing the expressions for a weak form in functions **leftHandSide(...)** and

rightHandSide(...). Function **integrate(...)** is also called by the assembler, the expression of left hand side and right hand side will be the integrand and be integrated on finite elements. Function **isVVFComponentCoupled(...)** is used in the case of vector valued problems. In the loop of assembling, if two components do not couple with each other, the computation can be skipped, so the cost of time is saved. For example, in 2D Stokes problem (u,v, p), u and v do not couple with each other while u, v will both couple with p.

3.4.5 Class WeakFormBuilder

Class **WeakFormBuilder** is a helper class that allows users to build weak forms with less lines of code than to write a class that implements interface **WeakForm** or extends from the abstract weak form classes. By using this helper class one can focus on the expressions of the weak forms itself. For example, problem (3.4.4) can be written as below

```

1 WeakFormBuilder wfBuilder = new WeakFormBuilder() {
2     /**
3      * Override this function to define weak form
4      */
5     @Override
6     public Function makeExpression(Element e, Type type) {
7         ScalarShapeFunction u = getScalarTrial();
8         ScalarShapeFunction v = getScalarTest();
9         Function ff = getParam("f",e);
10        Function fk = getParam("k",e);
11        Function fc = getParam("c",e);
12        Function fd = getParam("d",e);
13        Function fg = getParam("g",e);
14        switch(type) {
15            case LHS_Domain:
16                //(k*grad(u),grad(v))_Omega + (c*u,v)_Omega
17                return fk.M( grad(u,"x","y").dot(grad(v,"x","y")) ).A(
18                    fc.M(u).M(v) );
19            case LHS_Border://(d*u,v)_Gamma_N
20                return fd.M(u.M(v));
21            case RHS_Domain://(f,v)_Omega
22                return ff.M(v);
23            case RHS_Border://(g,v)_Gamma_N
24                return fg.M(v);
25        }
26        return null;
27    }
28 };
29 //f=-2*(x^2+y^2)+36
30 wfBuilder.addParam("f",X.M(X).A(Y.M(Y)).M(-2.0).A(36.0));
31 wfBuilder.addParam("k",C(0.1));
32 wfBuilder.addParam("c",C1);
33 wfBuilder.addParam("d",C1);
34 wfBuilder.addParam("g",X.M(X).A(Y.M(Y)));
35 WeakForm weakForm = wfBuilder.getScalarWeakForm();

```

The code segment first creates an anonymous class of type **WeakFormBuilder** and overrides member function **makeExpression(...)**; then the expression of the weak form defined in (3.4.4) is given by the switch clause. The different integral part of the weak form expression

can be distinguished by the value of parameter *type*. **Type** is an enum defined in class **WeakFormBuilder** as below

```

1 public static enum Type {
2     LHS_Domain, //Left Hand Side, integration on domain
3     LHS_Border, //Left Hand Side, integration on Border
4     RHS_Domain, //Right Hand Side, integration on domain
5     RHS_Border //Right Hand Side, integration on Border
6 };

```

Member function *getScalarTrial()* and *getScalarTest()* return the trial and test function respectively. The coefficients in (3.4.4) can be passed by member function *addParam(...)* and retrieved by member function *getParam(...)*.

3.5 Assembly Process

The summation process of the integration term in a weak form over elements is regarded as an assembly process. In this process, the element stiffness matrices are added into their proper places located in the global stiffness matrix.

Interface **Assembler** provides several standard methods that every assembling class should implements those methods. Users just call the standard functions, the assembly process for the specified problem will be performed by the assembler class. The details will be hidden, so users doesn't need to care about those.

There are two classes that implement interface **Assembler**: **AssemblerScalar** and **AssemblerVector**, which are provided for most application cases. Generally, They are enough to handle the process of assembly, except for some special applications, e.g. class **AssemblerMixedLaplace** for the case of mixed Laplace problem. **AssemblerScalar** is used for problems with scalar valued unknowns and **AssemblerVector** is for problems with vector valued unknowns. The following code segment shows a standard assembly process for a problem with scalar valued unknowns

```

1 Assembler assembler = new AssemblerScalar(mesh, weakForm);
2 assembler.assemble();
3 Matrix stiff = assembler.getStiffnessMatrix();
4 Vector load = assembler.getLoadVector();

```

The code is similar for a problem with vector valued unknowns.

AssemblerScalar and **AssemblerVector** will call functions *leftHandSide(...)* and *rightHandSide(...)* define in weak form classes to get the expressions and evaluate it. They are not the most efficient way of assembling. Class **AssemblerScalarFast** will call function *assembleElement(...)* defined in weak form classes, this function can be designed by users in a more efficient way than the first way mentioned. However, you may need to do more works on how to cook the optimized code to assemble local stiffness matrices when implementing the method *assembleElement(...)*.

Although, the assembler class above can handle most cases, but in some case, users should define their own assembler classes to perform special assembly process for their problems. Below, we will show the class **AssemblerMixedLaplace** for assembling of mixed Laplace problems using block matrix and vector. The global stiffness matrix and global load vector of mixed Laplace problem are actually block matrix and block vector. See (3.5.1), we write out the algebra system again:

$$(3.5.1) \quad \begin{pmatrix} B & C \\ C^T & 0 \end{pmatrix} \begin{pmatrix} \mathbf{p} \\ u \end{pmatrix} = \begin{pmatrix} 0 \\ b_f \end{pmatrix}$$

Block matrix and block vector will benefit the solving of linear system by Schur complement and post processing process. For this reason, we need to define a block matrix and a block vector as the parameters to the function **assembleElement()** of the weak form class. In the following code segment, the global block stiffness matrix and the global block load vector are created in the class constructor. The assembly process is performed in function *assemble()* through calling the member function *assembleElement(...)* of the weak form.

```

1  public class AssemblerMixedLaplace implements Assembler {
2      protected Mesh mesh;
3      protected WeakForm weakForm;
4      protected BlockMatrix globalStiff;
5      protected BlockVector globalLoad;
6
7      public AssemblerMixedLaplace(Mesh mesh, WeakForm weakForm) {
8          this.mesh = mesh;
9          this.weakForm = weakForm;
10
11         int edgeDOF = mesh.getEdgeList().size();
12         int elementDOF = mesh.getEdgeList().size();
13
14         globalStiff = new SparseBlockMatrix(2,2);
15         globalStiff.setBlock(1, 1,
16             new SparseMatrixRowMajor(edgeDOF, edgeDOF));
17         globalStiff.setBlock(1, 2,
18             new SparseMatrixRowMajor(edgeDOF, elementDOF));
19         globalStiff.setBlock(2, 1,
20             new SparseMatrixRowMajor(elementDOF, edgeDOF));
21         globalStiff.setBlock(2, 2,
22             new SparseMatrixRowMajor(elementDOF, elementDOF));
23
24         globalLoad = new SparseBlockVector(2);
25         globalLoad.setBlock(1,
26             new SparseVectorHashMap(edgeDOF));
27         globalLoad.setBlock(2,
28             new SparseVectorHashMap(elementDOF));
29
30     }
31
32     @Override
33     public void assemble() {
34         ElementList eList = mesh.getElementList();
35         int nEle = eList.size();
36         for(int i=1; i<=nEle; i++) {
37             eList.at(i).adjustVerticeToCounterClockwise();
38
39             weakForm.assembleElement(eList.at(i),
40                 globalStiff, globalLoad);
41         }
42         return;
43     }
44
45     public Vector getLoadVector() {
46         return globalLoad;

```

```

47     }
48
49     public Matrix getStiffnessMatrix() {
50         return globalStiff;
51     }
52 }

```

3.6 Solving Linear System

The current version of **FuturEye** provides only limited methods for solving linear system that directly implemented in this toolkit. But there are several matrix and vector classes defined in package **edu.uta.futureye.algebra** which includes useful wrapper classes that allow one to use other linear solver libraries, for example Java BLAS Interface, Matrix-Toolkits-Java(MTJ), Colt, EJML, Jama etc.

3.6.1 Matrix and Vector Classes

There are four kinds of matrix and vector interfaces

- Matrix and Vector
- SparseMatrix and SparseVector
- BlockMatrix and BlockVector
- AlgebraMatrix and AlgebraVector

It is easy to understand the key features of the four kinds interfaces from their names.

Interfaces **Matrix** and **Vector** are basic interfaces for general purpose matrix and vector operations.

SparseMatrix and **SparseVector** which extend interfaces **Matrix** and **Vector** respectively are designed to handle operations for sparse matrices and vectors. It enables users to iterate on non-zero values. Usually, the stiffness matrix and load vector generated from PDE assemble process are sparse matrix and vector.

BlockMatrix and **BlockVector** which extend interfaces **Matrix** and **Vector** are designed to handle operations for block (or partitioned) matrix and vector. Block matrix is a matrix broken into sections called blocks. That is to say, the matrix is written in terms of smaller matrices. In block matrix, rows and columns must be formed consistently: the matrix is split into blocks by horizontal and vertical lines, which must cut the matrix completely in the given direction. Block vector is similar.

AlgebraMatrix and **AlgebraVector** are designed to handle algebra operations. Only functions of algebra operations are defined in these interfaces. There is no fine-grained operations are required to be implemented in classes which are implementing these interfaces. So fast algebra operations can be implemented in this way.

We list the implemented classes of the above interfaces in the tables below:

Table 7: Classes implement interfaces Matrix and Vector directly

Classes	Comments
SpaceVector	Space vector of arbitrary dimension

Table 8: Classes implement interfaces SparseMatrix and SparseVector

Classes	Comments
SparseMatrixRowMajor	Sparse row-major storage matrix, mainly used in assemble process
SparseMatrixColMajor	Sparse column-major storage matrix
SparseVectorHashMap	Sparse vector based on HashMap storage, mainly used in assemble process

Table 9: Classes implement interfaces BlockMatrix and BlockVector

Classes	Comments
SparseBlockMatrix	Sparse block matrix, mainly used in assemble process of vector valued problems
SparseBlockVector	Sparse block vector, mainly used in assemble process of vector valued problems

Table 10: Classes implement interfaces AlgebraMatrix and AlgebraVector

Classes	Comments
CompressedRowMatrix	The compressed row storage (CRS) format of matrix
CompressedColMatrix	The compressed col storage (CCS) format of matrix
FullMatrix	Full storage of matrix
FullVector	Full storage of vector

3.6.2 Internal and External Solvers

Internal solvers are solvers implemented in FuturEye, while external solvers are which implemented by other packages. Now, there are two kinds of internal solvers in FutueEye. One is iterative solvers and the other one is direct solvers.

Class **Slover** provides several iterative methods that directly implemented in **FuturEye** for solving sparse linear systems.

```

1 public AlgebraVector solveCG(AlgebraMatrix A, AlgebraVector b,
2     AlgebraVector x)

```

Function **solveCG** is the Conjugate Gradients iterative method that solves symmetric positive definite linear system.

```

1 public AlgebraVector solveCGS(AlgebraMatrix A, AlgebraVector b,
2     AlgebraVector x)

```

Function **solveCGS** is the Conjugate Gradients squared iterative method that solves unsymmetric linear system.

After assembly process, we get the global stiffness matrix and global load vector. But usually, they are sparse matrix and vector. The interfaces they implemented are **Matrix** and **Vector**. In order to use the solvers above, users should convert the global matrix and vector to the right ones. However, it's an easy thing. The following code segment shows how to do this:

```

1 ...
2 Matrix stiff = assembler.getStiffnessMatrix();

```

```

3      Vector load = assembler.getLoadVector();
4      ...
5      Solver solver = new Solver();
6      AlgebraMatrix A =
7          new CompressedRowMatrix((SparseMatrix)stiff,true);
8      FullVector b = new FullVector((SparseVector)load);
9      //Initial value for iteration solvers
10     SparseVector x = new SparseVector(load.getDim(),1.0);
11     FullVector algX = new FullVector(x);
12     solver.solveCG(A, B, algX);
13
14     //Sometimes, the solution will be output to a file,
15     //we need object having interface Vector, so assign
16     //the values of algX to vector x
17     double[] data = algX.getData();
18     for(int i=0;i<data.length;i++) {
19         x.set(i+1, data[i]);
20     }

```

In the current version of **FuturEye**, we have implemented the following methods of class **Solver** that provide iterative methods with parameters of type **Matrix** and **Vector** for fast coding and testing. They convert the objects of type **Matrix** and **Vector** automatically to the objects of type **AlgebraMatrix** and **AlgebraVector**.

Table 11: Methods in Class Solver

Methods in Solver	Comments
Vector solveCG(Matrix A, Vector b, Vector x)	CG iterative method
Vector solveCG(Matrix A, Vector b)	CG iterative method with initial x=1.0
Vector solveCGS(Matrix A, Vector b, Vector x)	CGS iterative method
Vector solveCGS(Matrix A, Vector b)	CGS iterative method with initial x=1.0

Class **LUDecomposition** provides methods to solve sparse or dense linear systems by LU decomposition.

```

1  /**
2      * Matrix RHS
3      * Solve A*X=F with LU decomposition, where A is full matrix
4      *
5      * @param A (Input) Coefficient matrix
6      * @param L (Output) Lower triangular matrix with ones on its
7      *       diagonal
8      * @param U (Output) Upper triangular matrix
9      * @param P (Output) Permutation matrix
10     * @param X (Input) Unknown vectors in matrix form
11     * @param F (Input) Right hand sides in matrix form
12     * @return
13     */
14     public static FullMatrix solve(FullMatrix A,
15         FullMatrix L, FullMatrix U, SparseMatrix P,
16         FullMatrix X, FullMatrix F)

```

```

1  /**
2   * Solve A*X=F with LU decomposition, where A is sparse matrix
3   *
4   * @param A (Input) Coefficient matrix
5   * @param L (Output) Lower triangular matrix with ones on its
6   *   diagonal
7   * @param U (Output) Upper triangular matrix
8   * @param P (Output) Permutation matrix
9   * @param X (Input) Unknown vectors in matrix form (column
10  *   based)
11  * @param F (Input) Right hand sides in matrix form (column
12  *   based)
13  * @return
14  */
15 public static SparseMatrix solve(SparseMatrixRowMajor A,
16   SparseMatrixRowMajor L, SparseMatrixRowMajor U,
17   SparseMatrix P,
18   SparseMatrix X, SparseMatrix F)

```

The solver for sparse matrix is more efficient for sparse linear systems from FEM context. There are also two counterpart functions exist for vector unknown x and right hand side f . See the class reference document for details.

We also have several classes for solving linear systems with block coefficient matrix which is usually generated from vector valued PDE problems or PDE based optimization problems.

Class **SchurComplementMixSolver** is designed to solve linear systems $Ax = b$ where A has the following form

$$\begin{pmatrix} M & B' \\ B & 0 \end{pmatrix}$$

Class **SchurComplementStokesSolver** is designed to solve linear systems $Ax = b$ where A has the following form

$$\begin{pmatrix} B_1 & 0 & C_1 \\ 0 & B_2 & C_2 \\ C_1' & C_2' & C \end{pmatrix}$$

Class **SchurComplementLagrangianSolver** is designed to solve linear systems $Ax = b$ where A has the following form

$$\begin{pmatrix} M & A' & 0 \\ A & 0 & C \\ 0 & C' & R \end{pmatrix}$$

and for multi-measurement case

$$\begin{pmatrix} M_1 & & & A'_1 & & 0 \\ & M_2 & & A'_2 & & 0 \\ & & \dots & & \dots & \dots \\ & & & M_n & & A'_n \\ A_1 & & & 0 & & C_1 \\ & A_2 & & 0 & & C_2 \\ & & \dots & & \dots & \dots \\ & & & A_n & & 0 \\ 0 & 0 & \dots & 0 & C'_1 & C'_2 & \dots & C'_n & R \end{pmatrix}$$

External solvers are defined in package **edu.uta.futureye.algebra.solver.external**. These classes actually are interfaces between FuturEye and other solver packages.

solvers which Class **SloverJBLAS** is a wrapper of Java BLAS library. The following table shows the methods defined in it:

Class **SloverJBLAS** is a wrapper of Java BLAS library. The following table shows the methods defined in it:

Table 12: Methods in Class SloverJBLAS

Methods in SloverJBLAS	Comments
Vector solveDGESV(Matrix m, Vector v)	DGESV

Class **SloverMTJ** is a wrapper of Matrix-Toolkits-Java library. The following table shows the methods defined in it:

Table 13: Methods in Class SloverMTJ

Methods in SloverMTJ	Comments
Vector solveCG(SparseMatrix A, SparseVector b, SparseVector x)	CG iterative method

3.6.3 Convert Matrices Between Different Packages

In this section, we talk about how to convert a matrix to another matrix from a different package. Usually, we will get a linear system with a sparse stiffness matrix from assemble process of FEM method. In order to solve the system by using solvers from other packages, we need to convert the sparse stiffness matrix to the specific matrix that the packages support. Below, we give several demos to show how to convert a sparse matrix in FuturEye to package Jama, Colt, MTJ and EJML.

Convert matrices which implement interface **SparseMatrix** to instances of **Jama.Matrix**. Suppose we have a sparse matrix A with dimensions m by n.

```

1 public static Jama.Matrix toJamaMatrix(SparseMatrix A) {
2     FullMatrix tmpA = new FullMatrix(A);
3     Jama.Matrix A2 = new Jama.Matrix(
4         tmpA.getData(), A.getRowDim(), A.getColDim());
5     return A2;
6 }
```

The storage structure of `Jama.Matrix` is a two dimensional array. Fundamental matrix decompositions are used to solve linear systems. The most direct way to do the conversion is that construct an m-by-n matrix of zeros and iterate on sparse matrix A to assign (call function `set(int i, int j, double s)`) non-zero values to the full Jama matrix. However, our way here which follows the same principle is more concise. We first construct an instance `tmpA` of **FullMatrix** by passing A to the constructor and then create the object of **Jama.Matrix** by calling proper functions of `tmpA` which providing exactly the parameters that the constructor of **Jama.Matrix** requires. So, here **FullMatrix** can be seen as a bridge between our **SparseMatrix** and **Jama.Matrix**.

Convert matrices with interface **SparseMatrix** to instances of **DenseDoubleMatrix2D** and **SparseDoubleMatrix2D** in Colt.

```

1 public static DenseDoubleMatrix2D toColtDenseDoubleMatrix2D(
    SparseMatrix A) {
2     FullMatrix tmpA = new FullMatrix(A);
3     DenseDoubleMatrix2D A2 = new DenseDoubleMatrix2D(tmpA.getData
        ());
4     return A2;
5 }
6
7 public static SparseDoubleMatrix2D toColtSparseDoubleMatrix2D(
    SparseMatrix A) {
8     SparseDoubleMatrix2D A2 = new SparseDoubleMatrix2D(
9         A.getRowDim(), A.getColDim());
10    for(MatrixEntry e : A) {
11        A2.setQuick(e.getRow(), e.getCol(), e.getValue());
12    }
13    return A2;
14 }

```

Convert matrices with interface **SparseMatrix** to instances of **CompRowMatrix** and **CompColMatrix** in MTJ and convert them back.

```

1 public static CompRowMatrix toMTJCompRowMatrix(SparseMatrix A) {
2     CompressedRowMatrix tmpA = new CompressedRowMatrix(A, false);
3     CompRowMatrix cnvtA = new CompRowMatrix(
4         A.getRowDim(), A.getColDim(), tmpA.getColIndex());
5     for(MatrixEntry e : A) {
6         cnvtA.set(e.getRow()-1, e.getCol()-1, e.getValue());
7     }
8     return cnvtA;
9 }
10
11 public static CompColMatrix toMTJCompColMatrix(SparseMatrix A) {
12     CompressedColMatrix tmpA = new CompressedColMatrix(A, false);
13     CompColMatrix cnvtA = new CompColMatrix(
14         A.getRowDim(), A.getColDim(), tmpA.getRowIndex());
15     for(MatrixEntry e : A) {
16         cnvtA.set(e.getRow()-1, e.getCol()-1, e.getValue());
17     }
18     return cnvtA;
19 }
20 }

```



```

21 public static SparseMatrix fromMTJ(CompRowMatrix A) {
22     SparseMatrix A2 = new SparseMatrixRowMajor(
23         A.numRows(), A.numColumns());
24     for(no.uib.cipr.matrix.MatrixEntry e : A) {
25         A2.set(e.row()+1, e.column()+1, e.get());
26     }
27     return A2;
28 }
29
30 public static SparseMatrix fromMTJ(CompColMatrix A) {
31     SparseMatrix A2 = new SparseMatrixRowMajor(
32         A.numRows(), A.numColumns());
33     for(no.uib.cipr.matrix.MatrixEntry e : A) {
34         A2.set(e.row()+1, e.column()+1, e.get());
35     }
36     return A2;
37 }

```

Convert matrices with interface **SparseMatrix** to instances of **DenseMatrix64F** in EJML

```

1 public static DenseMatrix64F toEJMLDenseMatrix64F(SparseMatrix A)
2 {
3     FullMatrix tmpA = new FullMatrix(A);
4     DenseMatrix64F A2 = new DenseMatrix64F(tmpA.getData());
5     return A2;
6 }

```

All these static functions can be found in class edu.uta.futureye.tutorial.MatrixConvert.

4 Whole Examples

In this section, we will give several whole examples about how to use the basic components in **FuturEye** to solve specific Partial Differential Equations(PDE) with Finite Element Methods(FEM).

4.1 Example 1: Elliptic Problem

The first example is an elliptic problem with true solution known. The problem is find $u \in H^1(\Omega)$ such that

$$(4.1.1) \quad -\Delta u(x, y) = f(x, y), \quad (x, y) \in \Omega$$

$$(4.1.2) \quad u(x, y) = 0, \quad (x, y) \in \partial\Omega$$

$$(4.1.3) \quad f(x, y) = -2(x^2 + y^2) + 36$$

$$(4.1.4) \quad \Omega = [-3, 3] \times [-3, 3]$$

The true solution is

$$(4.1.5) \quad u = (x^2 - 9)(y^2 - 9), \quad (x, y) \in \Omega$$

The complete code file for solving the elliptic problem is listed below and a description for each part of the code segment follows.

```

1 package edu.uta.futureye.tutorial;
2
3 import java.util.HashMap;
4 import edu.uta.futureye.algebra.intf.Matrix;
5 import edu.uta.futureye.algebra.intf.Vector;
6 import edu.uta.futureye.algebra.solver.external.SolverJBLAS;
7 import edu.uta.futureye.core.Mesh;
8 import edu.uta.futureye.core.NodeType;
9 import edu.uta.futureye.function.intf.Function;
10 import edu.uta.futureye.io.MeshReader;
11 import edu.uta.futureye.io.MeshWriter;
12 import edu.uta.futureye.lib.assembler.AssemblerScalar;
13 import edu.uta.futureye.lib.element.FELinearTriangle;
14 import edu.uta.futureye.lib.weakform.WeakFormLaplace2D;
15 import edu.uta.futureye.util.container.ElementList;
16 import static edu.uta.futureye.function.operator.FMath.*;
17
18 public class T02Laplace {
19     public Mesh mesh;
20     public Vector u;
21     public void run() {
22         //1.Read in a triangle mesh from an input file with
23         // format ASCII UCD generated by Gridgen
24         MeshReader reader = new MeshReader("triangle.grd");
25         Mesh mesh = reader.read2DMesh();
26         //Compute geometry relationship of nodes and elements
27         mesh.computeNodeBelongsToElements();
28
29         //2.Mark border types
30         HashMap<NodeType, Function> mapNTF =
31             new HashMap<NodeType, Function>();
32         mapNTF.put(NodeType.Dirichlet, null);
33         mesh.markBorderNode(mapNTF);
34
35         //3.Use element library to assign degrees of
36         // freedom (DOF) to element
37         ElementList eList = mesh.getElementList();
38         FELinearTriangle feLT = new FELinearTriangle();
39         for(int i=1;i<=eList.size();i++)
40             feLT.assignTo(eList.at(i));
41
42         //4.Weak form
43         WeakFormLaplace2D weakForm = new WeakFormLaplace2D();
44         //Right hand side(RHS): f = -2*(x^2+y^2)+36
45         weakForm.setF(X.M(X).A(Y.M(Y)).M(-2.0).A(36.0));
46
47         //5.Assembly process
48         AssemblerScalar assembler =
49             new AssemblerScalar(mesh, weakForm);
50         assembler.assemble();
51         Matrix stiff = assembler.getStiffnessMatrix();
52         Vector load = assembler.getLoadVector();

```

```

53      //Boundary condition
54      assembler.imposeDirichletCondition(C0);
55
56      //6.Solve linear system
57      SolverJBLAS solver = new SolverJBLAS();
58      Vector u = solver.solveDGESV(stiff, load);
59      System.out.println("u=");
60      for(int i=1;i<=u.getDim();i++)
61          System.out.println(String.format("%.3f", u.get(i)));
62
63      //7.Output results to an Techplot format file
64      MeshWriter writer = new MeshWriter(mesh);
65      writer.writeTechplot("./tutorial/Laplace2D.dat", u);
66
67      this.mesh = mesh;
68      this.u = u;
69  }
70
71  public static void main(String[] args) {
72      T02Laplace ex1 = new T02Laplace();
73      ex1.run();
74  }
75  }

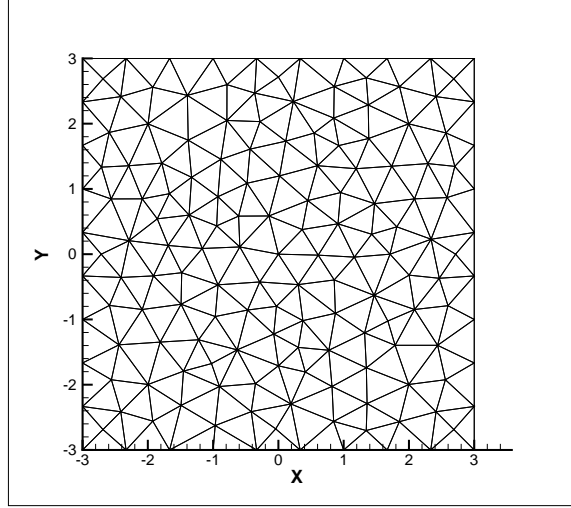
```

The class **Laplace** defined in the package **edu.uta.futureye.tutorial**. We call it ‘Laplace’ mainly because we want to emphasize the Laplace operator in the equation. There are many example classes defined in package **edu.uta.futureye.tutorial**. It’s a good idea for new users to start learning **FuturEye** by reading the classes in it and also a good draft of coding for whom that want begin a real application by selecting similar classes in the package.

For simplicity, we start our work directly in the function ‘main’. In code segment 1 (numbers in comment lines), an input file “triangle.grd” with format ASCII UCD that generated by Gridgen is read in. The object **reader** is constructed by the file name, and an object **mesh** is created by calling **reader.read2DMesh()**. When you have an object of type **Mesh**, the node list and element list can be accessed through it’s methods, see section 3.1.1 for details. Before continuing the work, the necessary geometry information about nodes and elements must be known, see section 3.1.2 for details. Here we call **mesh.computeNodeBelongsToElements()** to get the relationship of nodes and elements. The mesh is shown in Fig.7.

In code segment 2, we mark the border node types of the grid. A map of **NodeType** and **Function** is needed to indicate what type of node should be marked to the border nodes. The nodes selected to be marked to a node type are determined by the return value of a function. If the return value > 0 the node will be marked to the specified type, otherwise the node will be marked to the type that have a function of null(default value, see details in section 3.1.3). About how to specify the boundary values will be described later.

In code segment 3, we assign degrees of freedom to every element in object **mesh**. Element library class **FELinearTriangle** is used to do the work. It is possible to create DOF objects directly and select proper shape functions by users, see section 3.3 for details.

Figure 7: Mesh *triangle.grd* for example 1

In code setment 4, an object of class **WeakFormLaplace2D** is defined. The weak formulation of (4.1.1) reads: Given $f \in L^2(\Omega)$, seek $u \in H^1(\Omega)$ such that

$$(4.1.6) \quad (\nabla u, \nabla v)_\Omega = (f, v)_\Omega$$

where $v \in H_0^1(\Omega)$ is arbitrary function. This is a simplified case of weak formulation that class **WeakFormLaplace2D** represented. Class **WeakFormLaplace2D** is designed to solve 2D general elliptic PDE equations (see section 3.4.1), for example steady state thermal problems and electrostatic problems etc. For this example, we just specify the right hand side function $f(x, y) = -2(x^2 + y^2) + 36$ by function operations (see 2.4 for details)

```
1 weakForm.setF(X.M(X).A(Y.M(Y)).M(-2.0).A(36.0));
```

In code segment 5, assembly process is performed. In this example, we use class **AssemblerScalar** to assemble global stiffness matrix and global load vector. Details description of assembly process see section 3.5. When we get the global stiffness matrix and global load vector, we need impose Dirichlet boundary condition to the nodes on the boundary of Ω . This is done by

```
1 assembler.imposeDirichletCondition(FC.c0);
```

The parameter of the method is a **Function** object. This method will loop over every boundary node and pass the coordinates and global index of the node to the **Function** object. The object then evaluate the proper values for the node according to the parameters passed in. In our case, all the values on nodes of Dirichlet boundary is 0. In other case, users should define their own function object for imposing Dirichlet condition. For example, if we have global node index and value pairs for Dirichlet boundary condition, the function class **DiscreteIndexFunction** (see section 2.10) will be useful and easy to do the job.

Now, we have finished the part of most important in the example. Next, in code segment 6, is the solver part. We use wrapper class **SolverJBLAS** if the Java BLAS library to solve the linear system. We print the solution vector to standard output.

In the last code segment(segment 7), an object of **MeshWriter** is used to write the solution vector to the file “tutorial_Laplace.dat” with format of Techplot.

The output of this demo listed below, we omit the long values list of the solution vector. The solution is shown in Fig.8

```

1 Assemble [0%-----100%]
2 Progress [*****] Done !
3 u=
4 0.000
5 0.000
6 0.000
7 ...
8 14.234
9 71.861
10 80.284

```

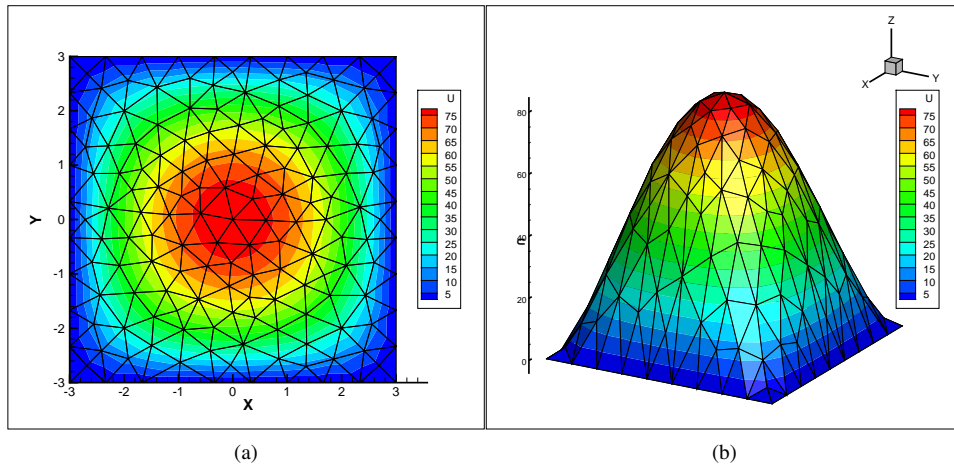


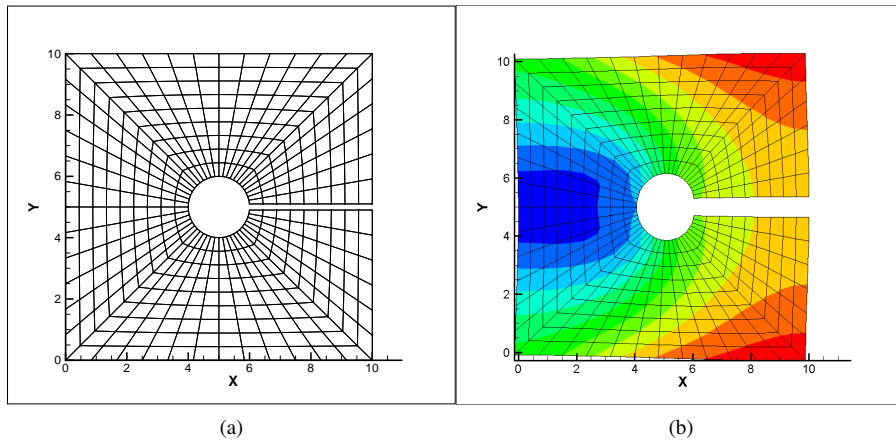
Figure 8: *Mesh and Result of example 1*

4.2 Example 2 Plane Stress

```

1 package edu.uta.futureye.tutorial;
2 ...
3 public class T09PlaneElasticHole

```

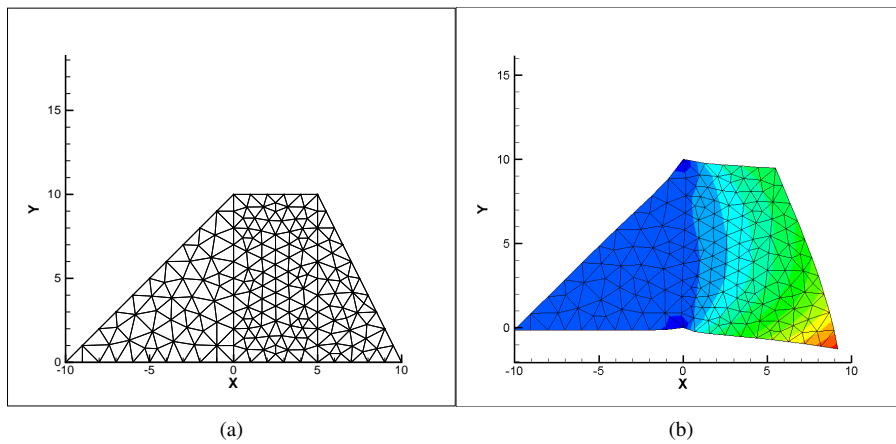
Figure 9: *Mesh and Result of example 2*

4.3 Example 3 Plane Stain

```

1 package edu.uta.futureye.tutorial;
2 ...
3 public class T09PlaneElasticDam

```

Figure 10: *Mesh and Result of example 3*

4.4 Example 4 Stokes: Cylinders

```

1 package edu.uta.futureye.tutorial;
2 ...
3 public class T10Stokes

```

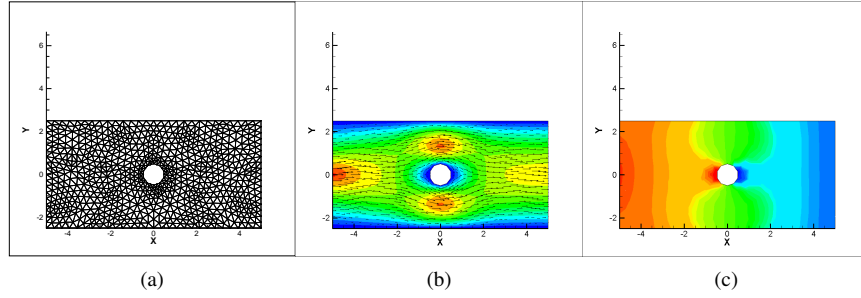


Figure 11: Results of example 4: 1 cylinder. (a) Mesh (b) Velocity (c) Pressure

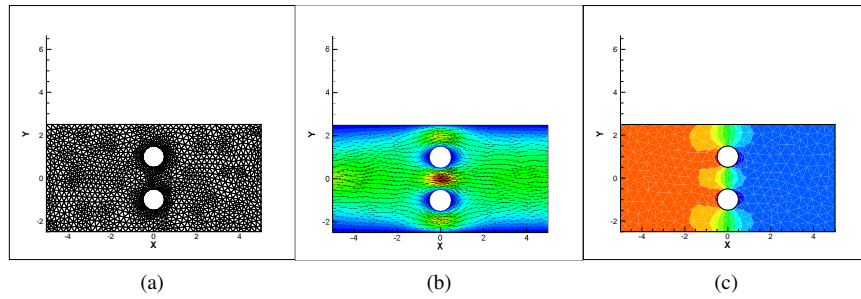


Figure 12: Results of example 4: 2 cylinders. (a) Mesh (b) Velocity (c) Pressure

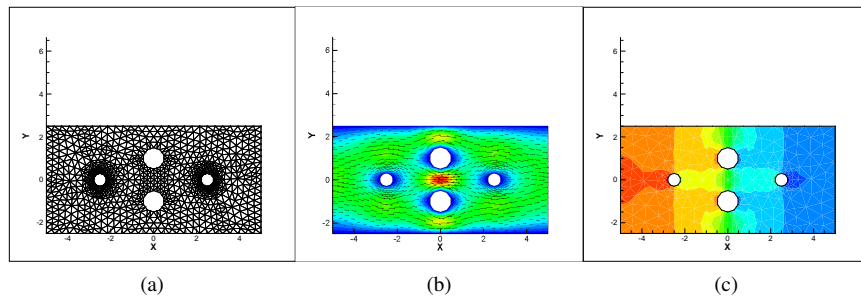


Figure 13: Results of example 4: 4 cylinders. (a) Mesh (b) Velocity (c) Pressure

4.5 Example 5 Stokes: Box

```

1 package edu.uta.futureye.tutorial;
2 ...
3 public class T10StokesBox

```

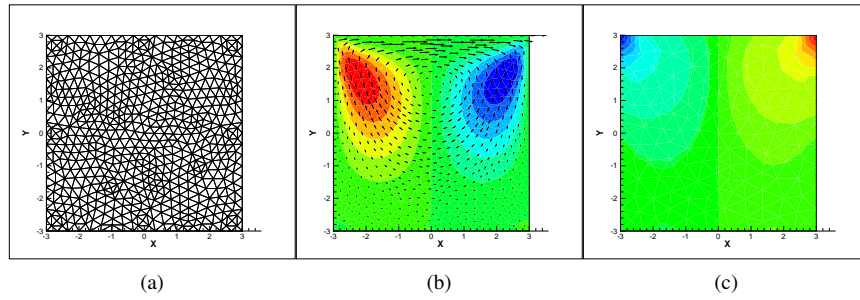


Figure 14: Results of example 5. (a) Mesh (b) Velocity (c) Pressure

4.6 Example 6 Stokes: U Shape Channel

```

1 package edu.uta.futureye.tutorial;
2 ...
3 public class T10Stokes

```

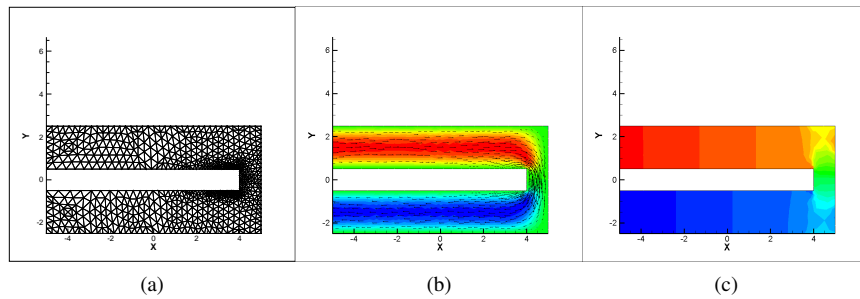


Figure 15: Results of example 6. (a) Mesh (b) Velocity (c) Pressure

4.7 Example 7 Convection Diffusion

```

1 package edu.uta.futureye.tutorial;
2 ...
3 public class T07ConvectionDiffusion

```

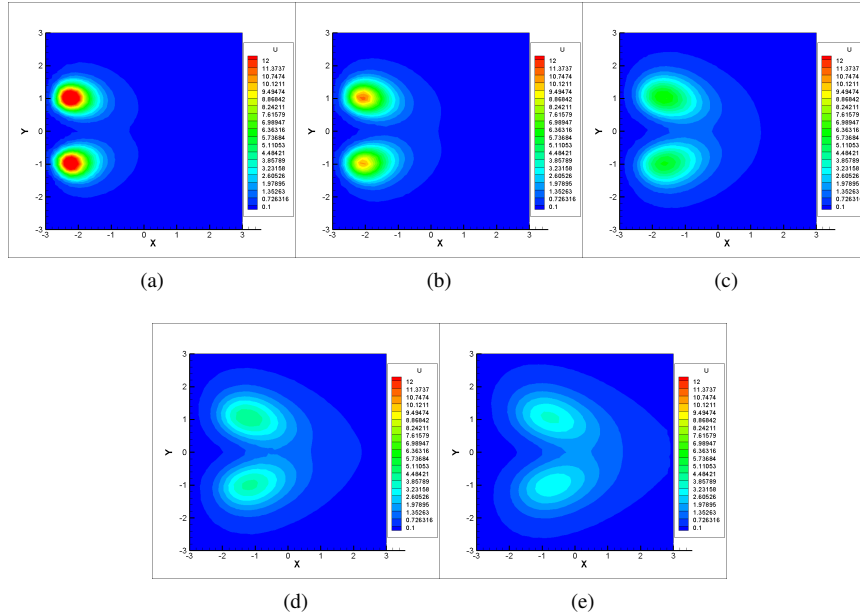


Figure 16: Results of example 7

4.8 Example 8 Wave

```

1 package edu.uta.futureye.tutorial;
2 ...
3 public class T06Wave

```

4.9 Example 9 3D Elliptic

```

1 package edu.uta.futureye.test;
2 ...
3 public class Laplace3DTest

```

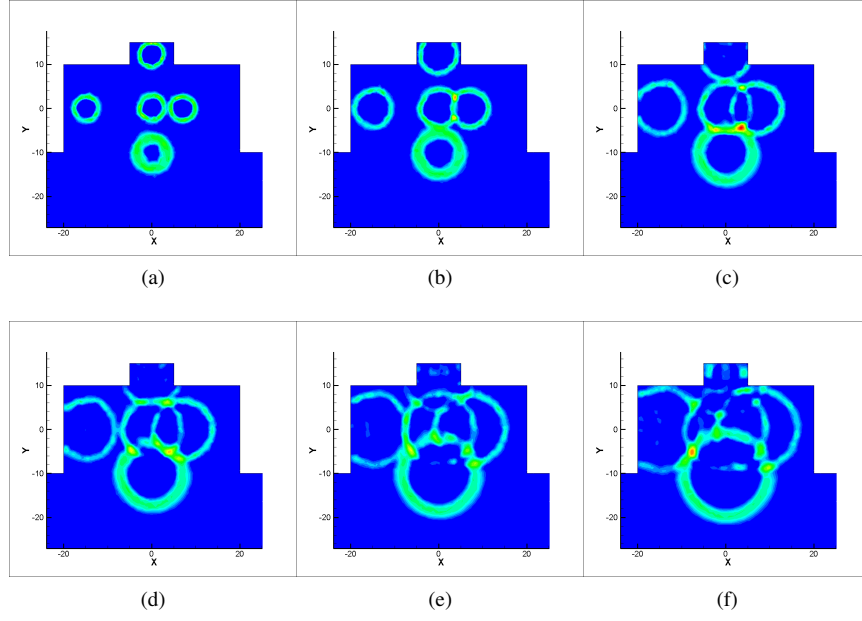


Figure 17: Results of example 8.

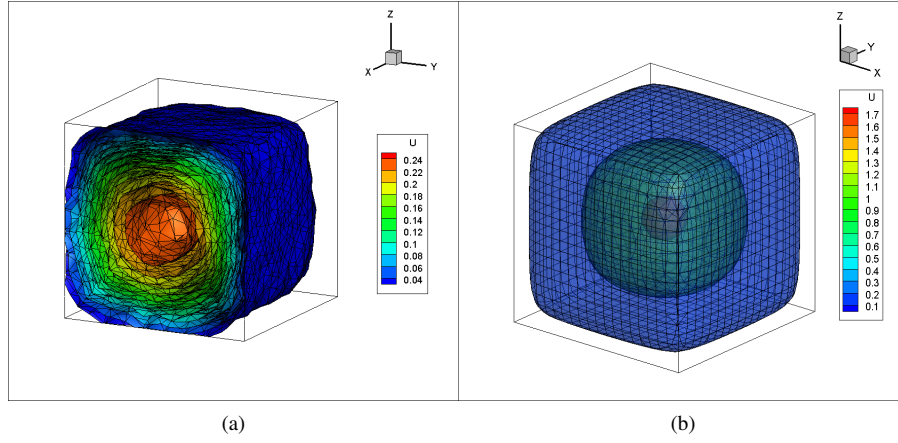


Figure 18: Results of example 9.

4.10 Example 10 Navier-Stokes Equation

This example follows the paper 'benchmark computations of laminar flow around a cylinder' given by M. Schäfer and S. Turek. An incompressible Newtonian fluid is considered for which the conservation equations of mass and momentum are

$$(4.10.1) \quad \frac{\partial U_i}{\partial x_i} = 0$$

$$(4.10.2) \quad \rho \frac{\partial U_i}{\partial t} + \rho \frac{\partial}{\partial x_j} (U_j U_i) = \rho \nu \frac{\partial}{\partial x_j} \left(\frac{\partial U_i}{\partial x_j} + \frac{\partial U_j}{\partial x_i} \right) - \frac{\partial P}{\partial x_i}$$

The notations are time t , cartesian coordinates $(x_1, x_2, x_3) = (x, y, z)$, pressure P and velocity components $(U_1, U_2, U_3) = (U, V, W)$. The kinematic viscosity is defined as $\nu = 10^{-3} \text{ m}^2/\text{s}$, and the fluid density is $\rho = 1.0 \text{ kg/m}^3$. In operator notation, this equation can be written as

$$(4.10.3) \quad \nabla \cdot \mathbf{v} = 0$$

$$(4.10.4) \quad \rho \partial_t \mathbf{v} - \rho \nu \Delta \mathbf{v} + \rho \mathbf{v} \cdot \nabla \mathbf{v} + \nabla p = \mathbf{f}$$

Where $\mathbf{v} = (U, V, W)$ and $\mathbf{f} = 0$.

For the 2D test cases the flow around a cylinder with circular cross-section is considered. The geometry and the boundary conditions are indicated in Fig. 19. The outflow condition is chosen as free boundary. $H = 0.41 \text{ m}$ is the channel height and $D = 0.1 \text{ m}$ is the cylinder diameter. The Reynolds number is defined by $Re = \bar{U}D/\nu$ with the mean velocity $\bar{U}(t) = 2U(0, H/2, t)/3$.

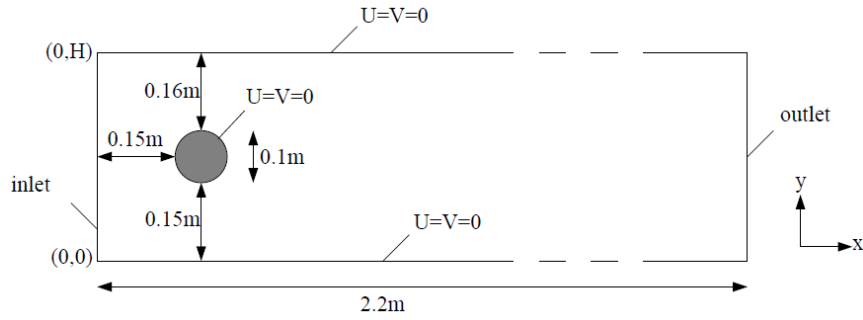


Figure 19: Geometry of 2D test cases with boundary conditions

Consider the unsteady case. The inflow condition is

$$U(0, y, t) = 4U_m y(H - y)/H^2, \quad V = 0$$

with $U_m = 1.5 \text{ m/s}$, yielding the Reynolds number $Re = 100$. In order to make the vortex street significant, we changed the boundary condition on sides $y = 0$ and $y = H$ to $U = 1, V = 0$.

The following code segment gives the definition of weak form class **WeakFormNavier-Stokes2D** for 2D Navier-Stokes equation with upwind technique.

```

1 package edu.uta.futureye.lib.weakform;
2
3 public class WeakFormNavierStokes2D extends AbstractVectorWeakForm
4 {
5     ...
6
7     @Override
8     public Function leftHandSide(Element e, ItemType itemType)
9     {
10         @Override
11         public Function leftHandSide(Element e, ItemType itemType) {
12
13             if(itemType==ItemType.Domain) {
14                 //Integrand part of Weak Form on element e

```

```

15     Function integrand = null;
16     Function fk = Utils.interpolateOnElement(g_k,e);
17     Function fc = Utils.interpolateOnElement(g_c,e);
18     VectorFunction fU = Utils.interpolateOnElement(g_U, e);
19
20     Function u1 = u.get(1), u2 = u.get(2), p = u.get(3);
21     Function v1 = v.get(1), v2 = v.get(2), q = v.get(3);
22
23     //upwind: v1 and v2 in (u,v,p)
24     if(this.testDOF.getVVFComponent() != 3) {
25         //\tidle{v} = v + \tidle{k}*\hat{U}*v,j/\norm{U}
26         Vector valU = g_U.value(new Variable().setIndex(node1.
27             globalIndex));
28         double normU = valU.norm2();
29         Vector valU_hat = valU.scale(1.0/normU);
30         double h = e.getElementDiameter();
31         double k = g_k.value(Variable.createFrom(g_k, node1, node1
32             .globalIndex));
33         double alpha = normU*h/(2*k);
34         //double k_tidle = 2*(MathEx.coth(alpha)-1.0/alpha)*normU*
35             h;
36         double k_tidle = (MathEx.coth(alpha)-1.0/alpha)*normU*h;
37         if(!(v1.isConstant())) {
38             v1.A(grad(v1,"x","y").dot(valU_hat).M(k_tidle).D(valU.
39                 norm2()));
40         }
41         if(!(v2.isConstant())) {
42             v2.A(grad(v2,"x","y").dot(valU_hat).M(k_tidle).D(valU.
43                 norm2()));
44         }
45     }
46
47     /**
48     * k* [(v1_x,u1_x) + (v1_y,u1_y) + (v2_x,u2_x) + (v2_y,u2_y) ]
49     * + [(U1*u1_x,v1)+(U2*u1_y,v1)] + [(U1*u2_x,v2)+(U2*u2_y,v2)]
50     * + c*[(u1,v1)+(u2,v2)]
51     * - (v1_x+v2_y,p)
52     * + (q,u1_x+u2_y)
53     * = (v1,f1)+(v2,f2)
54     */
55
56     VectorFunction grad_u1 = grad(u1,"x","y");
57     VectorFunction grad_u2 = grad(u2,"x","y");
58     Function uv1 = grad_u1.dot(grad(v1,"x","y"));
59     Function uv2 = grad_u2.dot(grad(v2,"x","y"));
60     Function div_v = v1._d("x").A(v2._d("y"));
61     Function div_u = u1._d("x").A(u2._d("y"));
62     Function cvect = fU.dot(grad_u1).M(v1).A(fU.dot(grad_u2).M
63         (v2));
64     Function cuv = fc.M(u1.M(v1).A(u2.M(v2)));
65     integrand = fk.M( uv1.A(uv2) ).A( cvect ).A( cuv ).S(
66         div_v.M(p) ).A( div_u.M(q) );

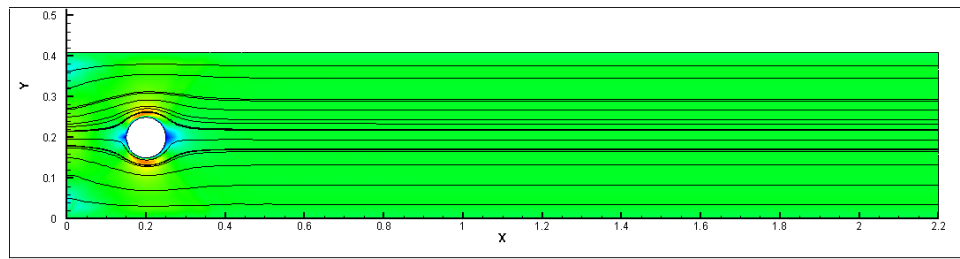
```

```

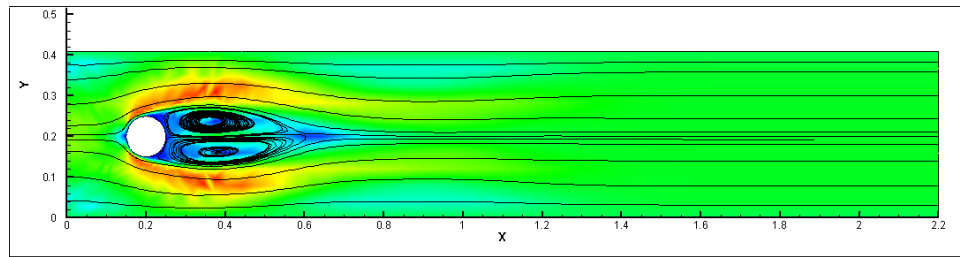
61         return integrand;
62     } else if(itemType==ItemType.Border) {
63         if(g_d != null) {
64             Element be = e;
65             Function fd1 = Utils.interpolateOnElement(g_d.get(1), be);
66             Function fd2 = Utils.interpolateOnElement(g_d.get(2), be);
67             Function u1 = u.get(1), u2 = u.get(2);
68             Function v1 = v.get(1), v2 = v.get(2);
69             //Robin: - k*u_n = d*u - p\vec{n}
70             //d1*u1 + d2*u2
71             Function borderIntegrand = fd1.M(u1.M(v1)).A(fd2.M(u2.M(v2)
72                 ));
73             return borderIntegrand;
74         }
75     }
76     return null;
77 }
78 @Override
79 public Function rightHandSide(Element e, ItemType itemType) {
80     if(itemType==ItemType.Domain) {
81         Function f1 = Utils.interpolateOnElement(g_f.get(1), e);
82         Function f2 = Utils.interpolateOnElement(g_f.get(2), e);
83         Function v1 = v.get(1), v2 = v.get(2);
84         //(v1*f1+v2*f2)
85         Function integrand = v1.M(f1).A(v2.M(f2));
86         return integrand;
87     } else if(itemType==ItemType.Border) {
88         Element be = e;
89         Function v1 = v.get(1), v2 = v.get(2), p = v.get(3);
90         //Robin: - k*u_n = d*u - p\vec{n}
91         //- p\vec{n} = - p*n1*v1 - p*n2*v2
92         Edge edge = (Edge)be.getGeoEntity();
93         Vector n = edge.getNormVector();
94         Function n1 = C(-1.0*n.get(1)), n2 = C(-1.0*n.get(2));
95         Function borderIntegrand = p.M(v1.M(n1)).A(p.M(v2.M(n2)));
96         return borderIntegrand;
97     }
98     return null;
99 }
100
101 public boolean isVVFComponentCoupled(int nComponent1, int
102     nComponent2) {
103     if(nComponent1 == nComponent2) return true;
104     else if(nComponent1 == 3 || nComponent2 == 3) return true;
105     else return false;
106 }

```

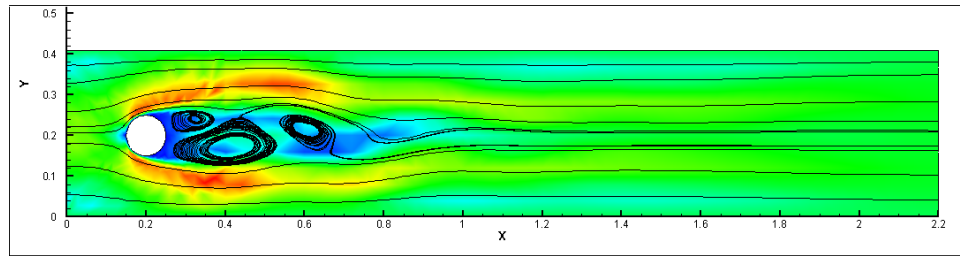
Class **T11NavierStokesCylinder** defined in package **edu.uta.futureye.tutorial** is a solver class for 2D unsteady case described above. It uses class **WeakFormNavierStokes2D** to represent the weak form for each time step. The selected results of streamline and velocity from the solutions of all time steps depicted in Fig. 20.



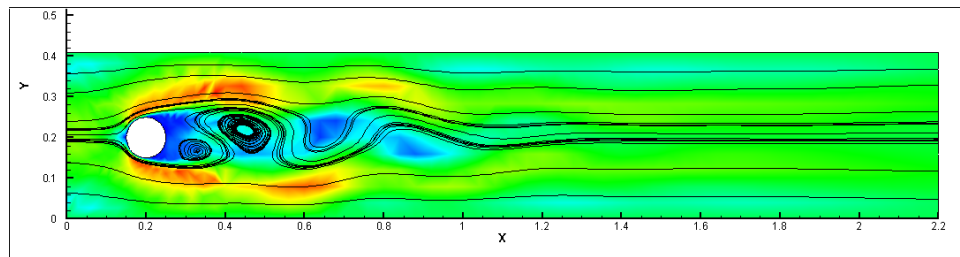
(a)



(b)



(c)



(d)

Figure 20: *Streamline and velocity*

5 Mesh Refinement

1 Adaptive:
2

```

3 mesh.computeNodesBelongToElement();
4 mesh.computeNeighborNode();
5 mesh.computeNeighborElement();
6
7
8 public static void adaptiveTestRectangle() {
9     // MeshReader reader = new MeshReader("patch_rectangle.grd");
10    MeshReader reader = new MeshReader("patch_rectangle2.grd");
11    // MeshReader reader = new MeshReader("patch_rectangle_refine
12    .grd");
13
14    Mesh mesh = reader.read2DMesh();
15    HashMap<NodeType, Function> mapNTF = new HashMap<NodeType,
16    Function>();
17    mapNTF.put(NodeType.Dirichlet, null);
18
19    mesh.computeNodeBelongsToElements();
20    mesh.computeNeighborNodes();
21    mesh.markBorderNode(mapNTF);
22    mesh.computeGlobalEdge();
23    mesh.computeNeighborElements();
24
25    ElementList eList = mesh.getElementList();
26    ElementList eToRefine = new ElementList();
27
28    //first refine
29    eToRefine.add(eList.at(6));
30    eToRefine.add(eList.at(7));
31    eToRefine.add(eList.at(10));
32    eToRefine.add(eList.at(11));
33    Refiner.refineOnce(mesh, eToRefine);
34    mesh.markBorderNode(mapNTF);
35
36    //refine again
37    eToRefine.clear();
38    eToRefine.add(eList.at(17));
39    eToRefine.add(eList.at(18));
40    Refiner.refineOnce(mesh, eToRefine);
41    mesh.markBorderNode(mapNTF);
42
43    SFBilinearLocal2D[] shapeFun = new SFBilinearLocal2D[4];
44    for(int i=0;i<4;i++)
45        shapeFun[i] = new SFBilinearLocal2D(i+1);
46    SFBilinearLocal2D[] shapeFun2 = new SFBilinearLocal2D[4];
47    for(int i=0;i<4;i++) {
48        shapeFun2[i] = new SFBilinearLocal2D(i+1,0.5);
49    }
50
51    //Assign degree of freedom to element
52    for(int i=1;i<=mesh.getElementList().size();i++) {
53        Element e = mesh.getElementList().at(i);
54        int nDofLocalIndexCounter = 0;
55        for(int j=1;j<=e.nodes.size();j++) {

```

```

54      //Assign shape function to DOF
55      if(e.nodes.at(j) instanceof NodeRefined) {
56          NodeRefined nRefined = (NodeRefined)e.nodes.at(j);
57          if(nRefined.isHangingNode()) {
58              DOF dof = new DOF(++nDofLocalIndexCounter,
59                              nRefined.constrainNodes.at(1).
60                                  globalIndex,
61                                  shapeFun2[j-1]);
62              e.addNodeDOF(j, dof);
63              DOF dof2 = new DOF(++nDofLocalIndexCounter,
64                                nRefined.constrainNodes.at(2).
65                                    globalIndex,
66                                    shapeFun2[j-1]);
67              e.addNodeDOF(j, dof2);
68          } else {
69              DOF dof = new DOF(++nDofLocalIndexCounter,
70                                e.nodes.at(j).globalIndex, shapeFun[j
71                                  -1]);
72              e.addNodeDOF(j, dof);
73          }
74      } else {
75          DOF dof = new DOF(++nDofLocalIndexCounter,
76                            e.nodes.at(j).globalIndex, shapeFun[j-1]);
77          e.addNodeDOF(j, dof);
78      }
79
80      //User defined weak form of PDE (including bounder conditions)
81      WeakFormLaplace2D weakForm = new WeakFormLaplace2D();
82      //-\Delta{u} = f
83      //u(x,y)=0, (x,y)\in\partial\Omega
84      //\Omega = [0,10]*[0,10]
85      //u=[(x-5)^2-25]*[(y-5)^2-25]
86      //f=-2*((x-5)^2 + (y-5)^2) + 100
87      Function fxm5 = new Fxpb("x",1.0,-5.0);
88      Function fym5 = new Fxpb("y",1.0,-5.0);
89      weakForm.setF(
90          FC.c(-2.0).M(FMath.pow(fxm5, new FC(2.0))) ).A(
91              FC.c(-2.0).M(FMath.pow(fym5, new FC(2.0))) )
92              ).A(FC.c(100.0))
93      );
94
95      AssemblerScalar assembler = new AssemblerScalar(mesh, weakForm
96      );
97      System.out.println("Begin Assemble...");
98      assembler.assemble();
99      Matrix stiff = assembler.getStiffnessMatrix();
100      Vector load = assembler.getLoadVector();
101      assembler.imposeDirichletCondition(new FC(0.0));
102      System.out.println("Assemble done!");
103
104      SolverJBLAS solver = new SolverJBLAS();

```



```

103     Vector u = solver.solveDGESV(stiff, load);
104
105     //hanging node
106     for(int i=1;i<=mesh.getElementList().size();i++) {
107         Element e = mesh.getElementList().at(i);
108         for(int j=1;j<=e.nodes.size();j++) {
109             if(e.nodes.at(j) instanceof NodeRefined) {
110                 NodeRefined nRefined = (NodeRefined)e.nodes.at(j);
111                 if(nRefined.isHangingNode()) {
112                     double hnValue =
113                         (u.get(nRefined.constrainNodes.at(1).
114                             globalIndex)+
115                             u.get(nRefined.constrainNodes.at(2).
116                                 globalIndex))/2.0;
117
118                     u.set(nRefined.globalIndex, hnValue);
119                 }
120             }
121         }
122
123         System.out.println("u=");
124         for(int i=1;i<=u.getDim();i++)
125             System.out.println(String.format("%.3f", u.get(i)));
126
127         MeshWriter writer = new MeshWriter(mesh);
128         writer.writeTechplot("patch_rectangle.dat", u);
129     }

```

6 Handling Exceptions

To be continue...

JStack, JMap

7 Inverse Problems

In this section, we consider Coefficient Inverse Problem (CIP) for PDE.

7.1 Useful Tools

When you get the results from algebra solver, there may be some requires to plot the results, smooth result functions or compute the derivative of the result functions. This section we will describe how to manipulate vectors and functions.

7.1.1 Plot Vectors and Functions

In order to plot vectors and functions, we must output them as files and these files must have special formats that can be recognized by some post processing softwares, e.g. Techplot. The following class **MeshWriter** is implemented to output vectors with Techplot file format. Other file formats can be implemented similarly.

A vector usually corresponds to a mesh that means the components of the vector correspond to the nodes of the mesh. The following code segment gives a demo about how to plot a vector \mathbf{u} into the folder **results** that based on the current file system path.

```

1  ...
2  MeshWriter writer = new MeshWriter(mesh);
3  writer.writeTechplot("./results/"+fileName, u);

```

The important thing is that the vector object and the mesh object must keep consistency with each other.

There is no direct methods defined in **MeshWriter** to plot functions. But we can convert a function to a vector, then it's easy to plot the vector instead of plotting the function directly. The following code segment shows how to convert a function to a vector. The idea is evaluating the function at each node of a corresponding mesh.

```

1  NodeList list = mesh.getNodeList();
2  int nNode = list.size();
3  Variable var = new Variable();
4  Vector v = new SparseVector(nNode);
5  for(int i=1;i<=nNode;i++) {
6      Node node = list.at(i);
7      var.setIndex(node.globalIndex);
8      for(int j=1;j<=node.dim();j++) {
9          if(fun.varNames().size()==node.dim())
10             var.set(fun.varNames().get(j-1), node.coord(j));
11      }
12      v.set(i, fun.value(var));
13  }

```

There is a tool class “**Tools**” defined in package **edu.uta.futureeye.tutorial**. Two static methods can be used directly to plot vectors and functions.

```

1  public static void plotVector(Mesh mesh, String outputFolder,
2                               String fileName, Vector v, Vector ...vs)
3  public static void plotFunction(Mesh mesh, String outputFolder,
4                                 String fileName, Function fun, Function ...funs)

```

The last parameters of these two methods are optional. More than one vector or function can be output to a file with these parameters. For example, if you want to plot a vector field of velocity $\mathbf{u} = (u, v)$, you just can write

```

1  ...
2  MeshWriter writer = new MeshWriter(mesh);
3  writer.writeTechplot("./results/"+fileName, u, v);

```

The following code segment gives a concrete example. The mesh is in domain $[-3, 3] \times [-3, 3]$ and the function $f(x, y) = x^2 + y^2$ is defined on this mesh.

```

1  MeshReader reader = new MeshReader("triangle.grd");
2  Mesh mesh = reader.read2DMesh();
3  //fun(x,y)=x^2+y^2
4  Function fun = FX.fx.M(FX.fx).A(FX.fy.M(FX.fy));
5  //Fig.(a)
6  plotFunction(mesh, ".", "testPlotFun.dat", fun);
7  VectorFunction gradFun = FMath.grad(fun);

```

```

8 //Fig. (b)
9 plotFunction(mesh, ".", "testPlotFunGrad.dat",
10 gradFun.get(1), gradFun.get(2));

```

Fig.21 shows the results viewed in Techplot.

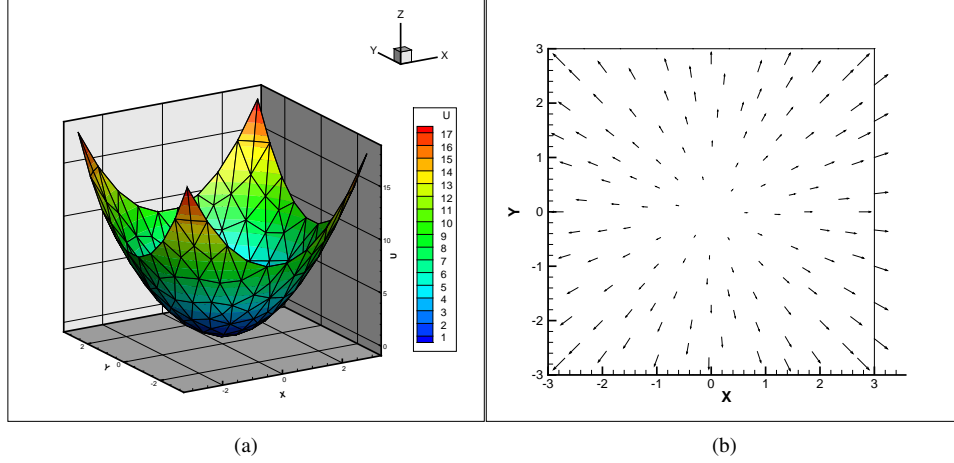


Figure 21: Plot of function $f(x,y) = x^2 + y^2$ and $\nabla f(x,y) = (2x, 2y)^T$ in $[-3, 3] \times [-3, 3]$

7.1.2 Data Manipulation

In this section, we give some examples of data manipulation.

Example 1. Gaussian Smoothing. The Gaussian smoothing operator is a 2-D convolution operator that can be used to smooth functions. Noise in the measurement data can be removed by this method. Static method **gaussSmooth** is defined in class **Utils** in the package **edu.uta.futureye.util**.

```

1 public static Vector gaussSmooth(Mesh mesh, Vector u, int
2 neighborBand, double weight)

```

Example 2. Derivative of a vector. The vector object is the discrete form of a function that usually comes from the results of some algebra solvers. The derivative of a vector equals the derivative of a function that correspond to the vector. However, most times we don't know the expression of the function, we only have the discrete vector object. Fortunately, the derivative operation can be done by using the weak form class **WeakFormDerivative**. This is based on solving the equation

$$(w, v) = (U_x, v)$$

where w is unknown and U_x is the piecewise derivative of U on the given mesh. We can say w is a good approximation of U_x .

It is easy to use class **WeakFormDerivative** to get the derivative of a vector object. There is a static method defined in class **edu.uta.futureye.tutorial.Tools**

```

1 public static Vector computeDerivative(Mesh mesh, Vector U,
2 String varName)

```

can be called directly to do the derivative.

Example 3. ΔU , U is a vector object. This operation can be achieved by using method **computeDerivative()** 4 times according to the formula

$$\Delta U = U_{xx} + U_{yy}$$

See static method

```
1 public static Vector computeLaplace2D(Mesh mesh, Vector U)
```

defined in class **edu.uta.futureye.tutorial.Tools** for details. The following code segment gives an example of data manipulation.

```
1 Laplace model = new Laplace();
2 model.run();
3 Vector u = model.u;
4 Vector ux = computeDerivative(model.mesh, u, "x");
5 Vector uy = computeDerivative(model.mesh, u, "y");
6 Vector uLaplace = computeLaplace2D(model.mesh, u);
7 plotVector(model.mesh, ".", "testPlotUx.dat", ux);
8 plotVector(model.mesh, ".", "testPlotUy.dat", uy);
9 plotVector(model.mesh, ".", "testPlotULaplace.dat", uLaplace);
10 //method gaussSmooth() need to know neighbor nodes
11 model.mesh.computeNeighborNodes();
12 uLaplace = Utils.gaussSmooth(model.mesh, uLaplace, 1, 0.5);
13 uLaplace = Utils.gaussSmooth(model.mesh, uLaplace, 1, 0.5);
14 uLaplace = Utils.gaussSmooth(model.mesh, uLaplace, 1, 0.5);
15 plotVector(model.mesh, ".", "testPlotULaplaceSmooth.dat",
    uLaplace);
```

We take the result vector u of class **Laplace** to show data operations. Do derivative of u with respect to x and y , we get ux and uy . Then call **computeLaplace2D()**, we get Δu . Because second derivative is not smooth, we call **gaussSmooth()** three times to get a good results.

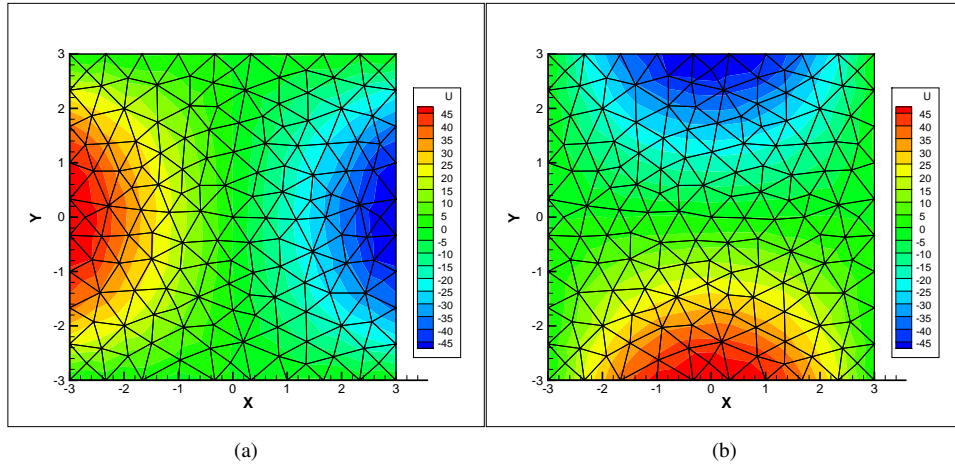


Figure 22: ux and uy in $[-3, 3] \times [-3, 3]$

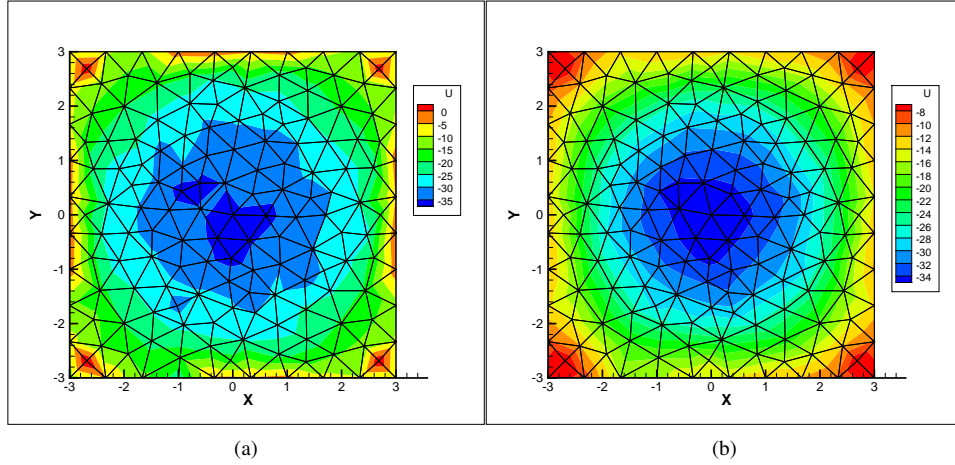


Figure 23: Δu and after 3 times smooth in $[-3, 3] \times [-3, 3]$

7.2 Optical Inverse Problems

7.2.1 Globally Convergent Method

In this section, we consider the Globally Convergent Method (GCM) for Coefficient Inverse Problem (CIP) for an elliptic equation with an unknown potential. CIP is one of problems of Diffusion Optical Tomography. Let's consider the 2D case for the sake of simplicity. The following equation describes the propagation of light (Near-infrared) in tissue.

$$(7.2.1) \quad \Delta u - a(\mathbf{x})u = -\delta(\mathbf{x} - \mathbf{x}_0), \mathbf{x}, \mathbf{x}_0 \in \mathbb{R}^2, \\ \lim_{|\mathbf{x}| \rightarrow \infty} u(\mathbf{x}, \mathbf{x}_0) = 0.$$

Here, u is the light intensity. \mathbf{x}_0 is the light source position, and this position is running along a line to generate the data for the inverse problem. The light source is a continuous-wave (CW). In this case, the coefficient $a(\mathbf{x})$ is

$$(7.2.2) \quad a(\mathbf{x}) = 3(\mu'_s \mu_a)(\mathbf{x}),$$

where $\mu'_s(\mathbf{x})$ is the reduced scattering coefficient and $\mu_a(\mathbf{x})$ is the absorption coefficient of the medium. The Inverse Problem is to determine the function $a(\mathbf{x})$ inside of the domain Ω assuming that the following the constant k and the function $\varphi(\mathbf{x}, \mathbf{x}_0)$ is given

$$(7.2.3) \quad u(\mathbf{x}, \mathbf{x}_0) = \varphi(\mathbf{x}, \mathbf{x}_0), \forall (\mathbf{x}, \mathbf{x}_0) \in \partial\Omega \times \Gamma.$$

where

$$a \in C^1(\mathbb{R}^2), a(\mathbf{x}) \geq k^2 \text{ and } a(\mathbf{x}) = k^2 \text{ for } \mathbf{x} \in \mathbb{R}^2 \setminus \Omega.$$

We assume that source $\{\mathbf{x}_0\}$ is located outside of the domain of interest Ω . That means the right hand side of equation (7.2.1) will be zero, if we consider the problem only in domain Ω . In this case, Dirichlet boundary condition (7.2.3) must be considered.

We consider the function

$$v = \frac{\ln u}{s^2}$$

for $\mathbf{x} \in \Omega$ and denote $s := |\mathbf{x}_0|$ as the parameter. Then obtain from (7.2.1)

$$(7.2.4) \quad \Delta v + s^2 |\nabla v|^2 = \frac{a(\mathbf{x})}{s^2}.$$

Let $q(\mathbf{x}, s) = \frac{\partial}{\partial s} v(\mathbf{x}, s)$, We obtain an nonlinear integral differential equation

$$(7.2.5) \quad \Delta q + 2s^2 \nabla q \nabla v + 2s |\nabla v|^2 = -2 \frac{a(\mathbf{x})}{s^3}.$$

where

$$(7.2.6) \quad v(\mathbf{x}, s) = - \int_s^{\bar{s}} q(\mathbf{x}, \tau) d\tau + T(\mathbf{x}), \mathbf{x} \in \Omega, s \in [\underline{s}, \bar{s}],$$

The Dirichlet boundary condition for q reads

$$(7.2.7) \quad q(\mathbf{x}, s) = \psi(\mathbf{x}, s) = \frac{\partial}{\partial s} \left(\frac{\ln \phi(\mathbf{x}, s)}{s^2} \right), \forall (\mathbf{x}, s) \in \partial\Omega \times [\underline{s}, \bar{s}].$$

$T(\mathbf{x})$ is the so-called “tail function”. The exact expression for this function is of course $T(\mathbf{x}) = v(\mathbf{x}, \bar{s})$. However, since the function $v(\mathbf{x}, \bar{s})$ is unknown, we will use an approximation for the tail function. If we approximate both q and T well (in a certain sense), then the target coefficient $a(\mathbf{x})$ would be easily reconstructed via backwards calculations.

We use Layer Stripping for approximating the function $q(\mathbf{x}, s)$ as a piecewise constant function with respect to the source position s . So, we assume that there exists a partition $\underline{s} = s_N < s_{N-1} < \dots < s_1 < s_0 = \bar{s}, s_{i-1} - s_i = h$ of the interval $[\underline{s}, \bar{s}]$ with a sufficiently small grid step size h such that

$$q(\mathbf{x}, s) = q_n(\mathbf{x}) \text{ for } s \in [s_n, s_{n-1}), n \geq 1$$

$$q_0 := 0.$$

We obtain for $n \geq 1$

$$(7.2.8) \quad \Delta q_n + A_{2,n} \left(h \sum_{j=0}^{n-1} \nabla q_j - \nabla T \right) \nabla q_n =$$

$$A_{1,n} (\nabla q_n)^2 + A_{4,n} \left(h \sum_{j=1}^{n-1} \nabla q_j - \nabla T \right)^2 + A_{3,n} \Delta T,$$

where

$$\begin{aligned}
A_{1,n} &= \frac{1}{I_{0,n}} \int_{s_n}^{s_{n-1}} (s_{n-1} - s)(2s^2 - 4s(s_{n-1} - s))ds, \\
A_{2,n} &= \frac{1}{I_{0,n}} \int_{s_n}^{s_{n-1}} (8s(s_{n-1} - s) - 2s^2)ds, \\
A_{3,n} &= \frac{2}{I_{0,n}} \int_{s_n}^{s_{n-1}} \frac{ds}{s}, \\
A_{4,n} &= -\frac{4}{I_{0,n}} \int_{s_n}^{s_{n-1}} sds, \\
I_{0,n} &= \int_{s_n}^{s_{n-1}} \left(1 - \frac{2}{s}(s_{n-1} - s)\right)ds.
\end{aligned}$$

We approximate the Dirichlet boundary condition (7.2.7) as a piecewise constant function

$$(7.2.9) \quad q_n(\mathbf{x}) = \psi_n(\mathbf{x}) = \frac{1}{h} \int_{s_n}^{s_{n-1}} \psi(\mathbf{x}, s)ds, \mathbf{x} \in \partial\Omega.$$

Assume that

$$\bar{s} \geq 1, h \in (0, 1) \text{ and } \frac{h}{\bar{s}} < \frac{1}{2}.$$

What is important to us here about numbers $A_{k,n}$ is that, given the assumption above, the following estimates hold (see [1])

$$\begin{aligned}
|A_{1,n}| &\leq 4\bar{s}^2 h, \\
|A_{2,n}| &\leq 8\bar{s}^2, |A_{3,n}| \leq 4, |A_{4,n}| \leq 16\bar{s}.
\end{aligned}$$

The estimate tells one that the nonlinearity $A_{1,n}(\nabla q_n)^2$ can be mitigated via decreasing the step size h . Thus, we can solve only linear elliptic Dirichlet boundary value problem for each q_n .

As soon as all functions $q_n, n = 1, \dots, N$ are computed, we can get $v(\mathbf{x}, s)$ from (7.2.6). In principle we reconstruct an approximation $a_n(\mathbf{x})$ for the function $a(\mathbf{x})$ as

$$a(\mathbf{x}) = \Delta \tilde{v}(\mathbf{x}, s_N) + |\nabla \tilde{v}(\mathbf{x}, s_N)|^2,$$

where $s_N = \underline{s}$ and

$$\tilde{v}(\mathbf{x}, s_N) = s_N^2 v(\mathbf{x}, s_N) = -s_N^2 h \sum_{j=0}^N q_j(\mathbf{x}) + s_N^2 T(\mathbf{x}).$$

However, it is numerically unstable to calculate second derivatives, since we use piecewise linear triangular finite elements to solve above equations for functions $q_n(\mathbf{x})$. Fortunately, we can get $u = e^{s^2 v}$, then we use (7.2.1) in the weak form as

$$(7.2.10) \quad \int_{\Omega} a(\mathbf{x}) u \eta_k d\Omega = - \int_{\Omega} \nabla u \nabla \eta_k d\Omega$$

where η_k is finite element test functions. This equation leads to a linear algebraic system. Solving this system we can get the target coefficient $a(\mathbf{x})$.

The main part of implementation of GCM is to solve (7.2.5). That is achieved by solving $q_n, n = 1, \dots, N$ in (7.2.8). Class **WeakFormGCM** is a weak form of (7.2.8) that can be used to solve equations like

$$-k\Delta u + \mathbf{b} \cdot \nabla u + cu = f.$$

Because there is a nonlinear term $A_{1,n}(\nabla q_n)^2$ in (7.2.8), it should be linearized, e.g. $A_{1,n}(\nabla q_n)^2 \approx A_{1,n}\nabla q_{n-1} \cdot \nabla q_n$. Then the coefficient $A_{1,n}\nabla q_{n-1}$ of ∇q_n in this nonlinear term can be merged into the coefficient \mathbf{b} in the equation above. However, this term can be mitigated via decreasing the step size h .

The definition of class **WeakFormGCM** shows below

```

1 public class WeakFormGCM extends AbstractScalarWeakForm {
2     protected Function g_f = null;
3
4     protected Function g_k = null;
5     protected Function g_c = null;
6     protected Function g_b1 = null;
7     protected Function g_b2 = null;
8
9     protected Function g_q = null;
10    protected Function g_d = null;
11
12    public void setF(Function f) {
13        this.g_f = f;
14    }
15
16    public void setParam(Function k, Function c, Function b1,
17        Function b2) {
18        this.g_k = k;
19        this.g_c = c;
20        this.g_b1 = b1;
21        this.g_b2 = b2;
22    }
23
24    //Robin: d*u + k*u_n= q
25    public void setRobin(Function q, Function d) {
26        this.g_q = q;
27        this.g_d = d;
28    }
29
30    @Override
31    public Function leftHandSide(Element e, ItemType itemType) {
32        if(itemType==ItemType.Domain) {
33            //Integrand part of Weak Form on element e
34            Function integrand = null;
35
36            Function fk = Utils.interpolateOnElement(g_k,e);
37            Function fc = Utils.interpolateOnElement(g_c,e);
38            Function fb1 = Utils.interpolateOnElement(g_b1,e);
39            Function fb2 = Utils.interpolateOnElement(g_b2,e);

```



```

40         integrand = FMath.sum(
41             fk.M(
42                 u._d("x").M(v._d("x")).A(
43                     u._d("y").M(v._d("y"))
44                 )),
45             fb1.M(u._d("x").M(v)),
46             fb2.M(u._d("y").M(v)),
47             fc.M(u.M(v))
48         );
49         return integrand;
50     }
51     else if(itemType==ItemType.Border) {
52         if(g_d != null) {
53             Element be = e;
54             Function fd = Utils.interpolateOnElement(g_d, be);
55             Function borderIntegrand = fd.M(u.M(v));
56             return borderIntegrand;
57         }
58     }
59     return null;
60 }
61
62 @Override
63 public Function rightHandSide(Element e, ItemType itemType) {
64     if(itemType==ItemType.Domain) {
65         Function ff = Utils.interpolateOnElement(g_f, e);
66         Function integrand = ff.M(v);
67         return integrand;
68     } else if(itemType==ItemType.Border) {
69         Element be = e;
70         Function fq = Utils.interpolateOnElement(g_q, be);
71         Function borderIntegrand = fq.M(v);
72         return borderIntegrand;
73     }
74     return null;
75 }
76 }

```

The full implementation of GCM is defined in class **GCMMModel**. Method

```

1 public void solveGCM(Mesh mesh, int N, double[]s, Vector[]phi,
2                     Vector tailT) {

```

can be called to get the reconstructed coefficient $a(\mathbf{x})$ by providing parameters:

- mesh: the domain Ω on which to solve $q_n(\mathbf{x})$,
- N: total number of partition of interval $[\underline{s}, \bar{s}]$,
- s: positions of light sources,
- phi: measurement data $\phi(\mathbf{x}, \mathbf{x}_0)$ on boundary $\partial\Omega$,
- tailT: tail function $T(\mathbf{x})$.

We briefly describe the reconstruction of the target coefficient $a(\mathbf{x})$ by backwards calculations. This work is done in the following method **solveParamInverse()**, the key part is

the using of weak form class **WeakFormL22D** which is the weak form of equation

$$(Uu, v) = (f, v) - (k\nabla U, \nabla v)$$

where u is unknown and U, f are know parameters.

```

1 public Vector solveParamInverse(Mesh mesh, Vector U) {
2     HashMap<NodeType, Function> mapNTF2 = new HashMap<NodeType,
3         Function>();
4     mapNTF2.put(NodeType.Dirichlet, null);
5     mesh.clearBorderNodeMark();
6     mesh.markBorderNode(mapNTF2);
7
8     //Weak form
9     WeakFormL22D weakFormL2 = new WeakFormL22D();
10    //Right hand side
11    weakFormL2.setF(this.delta);
12    //Parameters
13    weakFormL2.setParam(
14        this.k, new Vector2Function(U)
15    );
16
17    Assembler assembler = new AssemblerScalar(mesh, weakFormL2);
18    System.out.println("Begin Assemble... solveParamInverse");
19    assembler.assemble();
20    Matrix stiff = assembler.getStiffnessMatrix();
21    Vector load = assembler.getLoadVector();
22    assembler.imposeDirichletCondition(new FC(0.1));
23    System.out.println("Assemble done!");
24
25    SolverJBLAS solver = new SolverJBLAS();
26    Vector u = solver.solveDGESV(stiff, load);
27    return u;
28 }

```

7.2.2 Gradient Method for CW Case

This method is based on the Lagrange of the target functional. The reconstructed coefficient $a(\mathbf{x})$ from GCM can be used as a good approximation of real coefficient of the equation (7.2.1), we state it below again

$$(7.2.11) \quad \Delta u - a(\mathbf{x})u = -\delta(\mathbf{x} - \mathbf{x}_0), \text{ in } \Omega_0$$

$$(7.2.12) \quad u(\mathbf{x}, \mathbf{x}_0), \text{ on } \Omega_0$$

For actual numerical computation, we can not compute u in \mathbb{R}^2 . Alternatively, we choose a big enough domain Ω_0 , such that $\Omega \subset \Omega_0$ and $u \approx 0$ on $\partial\Omega_0$.

Let u and u_0 be the solution of the following two problems respectively:

$$(7.2.13) \quad \Delta u - a(\mathbf{x})u = 0, \text{ in } \Omega$$

$$(7.2.14) \quad \Delta u_0 - k^2 u_0 = 0, \text{ in } \Omega.$$

Let

$$v = \frac{u}{u_0}$$

substitute $u = vu_0$ in (7.2.13), we have

$$(7.2.15) \quad \Delta(vu_0) - a(\mathbf{x})vu_0 = 0, \text{ in } \Omega$$

$$(7.2.16) \quad u_0\Delta v + v\Delta u_0 + 2\nabla v \cdot \nabla u_0 - a(\mathbf{x})vu_0 = 0, \text{ in } \Omega$$

from (7.2.14), we know $\Delta u_0 = k^2 u_0$, substitute Δu_0 in (7.2.16) and eliminate u_0 , we have

$$(7.2.17) \quad \Delta v + 2\nabla v \cdot \nabla \ln u_0 - (a(\mathbf{x}) - k^2)v = 0, \text{ in } \Omega$$

Considering boundary condition

$$(7.2.18) \quad \frac{\partial(u - u_0)}{\partial n} - (u - u_0) = 0, \text{ on } \partial\Omega$$

we have

$$(7.2.19) \quad \frac{\partial v}{\partial n} + (1 + \frac{\partial \ln u_0}{\partial n})v = 1 + \frac{\partial \ln u_0}{\partial n}, \text{ on } \partial\Omega$$

Let $v = v(\mathbf{x}, s; a)$ be the solution of the following problem.

$$(7.2.20) \quad \begin{cases} \Delta v + 2\nabla v \cdot \nabla \ln u_0 - (a(\mathbf{x}) - k^2)v = 0, \text{ in } \Omega, \\ \frac{\partial v}{\partial n} + (1 + \frac{\partial \ln u_0}{\partial n})v = 1 + \frac{\partial \ln u_0}{\partial n}, \text{ on } \partial\Omega. \end{cases}$$

We want to find such a coefficient $a(\mathbf{x})$ which would guarantee that v also satisfy

$$(7.2.21) \quad v \approx \tilde{g}(\mathbf{x}, s), \text{ on } \Gamma$$

where

$$(7.2.22) \quad \tilde{g}(\mathbf{x}, s) = \frac{u(\mathbf{x}, s)}{u_0(\mathbf{x}, s)} \big|_{x \in \Gamma} = \frac{g(\mathbf{x}, s)}{u_0(\mathbf{x}, s)} \big|_{x \in \Gamma}$$

$g(\mathbf{x}, s)$ is measurement data on Γ .

Let $[\underline{s}, \bar{s}]$ be an interval for the light source distance $s \in [\underline{s}, \bar{s}]$. Consider the Lagrangian

$$(7.2.23) \quad L(a, v, \lambda) = \frac{1}{2} \int_{\underline{s}}^{\bar{s}} ds \int_{\Gamma} (v - \tilde{g})^2 d\sigma + \frac{\theta}{2} \int_{\Omega} (a(\mathbf{x}) - a_{glob}(\mathbf{x}))^2 d\Omega + \int_{\underline{s}}^{\bar{s}} ds \int_{\Omega} \lambda(\mathbf{x}, s) [\Delta v + 2\nabla v \cdot \nabla \ln u_0 - (a(\mathbf{x}) - k^2)v] d\Omega$$

where $a_{glob}(\mathbf{x})$ is the solution obtained on the globally convergent stage and θ is the regularization parameter. The function λ will be defined later.

Let $\varepsilon \in (0, 1)$ be a parameter. Consider $F(\varepsilon) = L(a + \varepsilon\delta a, v + \varepsilon\delta v, \lambda + \varepsilon\delta\lambda)$, where $\delta a, \delta v, \delta\lambda$ are arbitrary “good” functions and assume that

$$(7.2.24) \quad \frac{\partial \delta v}{\partial n} + (1 + \frac{\partial \ln u_0}{\partial n})\delta v = 0, \text{ on } \partial\Omega$$

We consider equation

$$(7.2.25) \quad \frac{dF}{d\varepsilon} \big|_{\varepsilon=0} = 0, \forall (\delta a, \delta v, \delta\lambda) \partial\Omega$$

This equation provides us with the minimizer of the Lagrangian. Since functions $\delta a, \delta v, \delta \lambda$ are arbitrary, then (7.2.25) implies

$$(7.2.26) \quad a(\mathbf{x}) = \frac{1}{\theta} \int_{\underline{s}}^{\bar{s}} \lambda v ds + a_{glob}(\mathbf{x})$$

and

$$(7.2.27) \quad \begin{cases} \Delta \lambda - 2 \nabla \lambda \cdot \nabla \ln u_0 - [2 \Delta \ln u_0 + (a(\mathbf{x}) - k^2)] \lambda = 0, \text{ in } \Omega, \\ \frac{\partial \lambda}{\partial n} + (1 - \frac{\partial \ln u_0}{\partial n}) \lambda = v - \tilde{g}, \text{ on } \partial \Omega. \end{cases}$$

The following algorithm can be used to update $a(\mathbf{x})$

- Step 0.1. Choose an appropriate interval $s \in [\underline{s}, \bar{s}]$ and an appropriate regularization parameter θ . Basically we want the number

$$\beta = \frac{\bar{s} - \underline{s}}{\theta}$$

to be small. In this case, the operator in the right hand side of (7.2.26) is contraction mapping. Therefore, the algorithm will converge.

- Step 0.2. Set $a_0(\mathbf{x}) := a_{glob}(\mathbf{x})$. Divide the interval $[\underline{s}, \bar{s}]$ into k small subintervals

$$\underline{s} = s_0 < s_1 < \cdots < s_k = \bar{s}, s_i - s_{i-1} = h$$

- Step $n \geq 1$. Suppose that the function $a_{n-1}(\mathbf{x})$ is found.

First, solve the boundary values problem (7.2.20) with $a(\mathbf{x}) := a_{n-1}(\mathbf{x})$ for all s_0, s_1, \dots, s_k . This way we find functions $v_{n-1}(\mathbf{x}, s_i), i = 0, 1, \dots, k$.

Next, solve the adjoint boundary value problem (7.2.27) with $a(\mathbf{x}) := a_{n-1}(\mathbf{x})$ for all s_0, s_1, \dots, s_k . In doing so, use in the boundary condition of (7.2.27) the calculated function $v(\mathbf{x}, s_i)|_{\Gamma} := v_{n-1}(\mathbf{x}, s_i)|_{\Gamma}$. We obtain functions $\lambda_{n-1}(\mathbf{x}, s_i), i = 0, 1, \dots, k$.

Finally, find the function $a_n(\mathbf{x})$ from (7.2.26) as

$$a(\mathbf{x}) = \frac{1}{\theta} \int_{\underline{s}}^{\bar{s}} \lambda_{n-1}(\mathbf{x}, s) v_{n-1}(\mathbf{x}, s) ds + a_{glob}(\mathbf{x})$$

where the integral is calculated via a standard rule. Continue this step until converges.

The implementation of the Gradient Method main focus on solving the equation (7.2.20), (7.2.27) and (7.2.26). Observing the expression of the equations of v and λ , we can see that the weak form of them can be presented by class **WeakFormGCM**. What we should do is just prepare the parameters for the object of class **WeakFormGCM**. The following two method defined in class **LagrangianMethod** is designed to solve the two equations.

```

1 public Vector solve_v(int s_i) {
2     Vector2Function fu0 = new Vector2Function(u0);
3     Vector2Function fu0_x = new Vector2Function(u0_x);
4     Vector2Function fu0_y = new Vector2Function(u0_y);
5
6     //-( a(x)-k^2 ) = k^2-a(x)
7     Vector v_c = FMath.axpy(-1.0, a, new SparseVector(a.getDim(), k
8         *k));
9     plotVector(mesh, v_c, "param_c"+s_i+".dat");

```

```

9      Vector2Function param_c = new Vector2Function(v_c);
10
11      //\Nabla{lnu0}
12      //Function b1 = F0Basic.Divi(fu0_x,fu0);
13      //Function b2 = F0Basic.Divi(fu0_y,fu0);
14      Function b1 = new Vector2Function(lnu0_x);
15      Function b2 = new Vector2Function(lnu0_y);
16      plotFunction(mesh, b1, "b1_"+s_i+".dat");
17      plotFunction(mesh, b2, "b2_"+s_i+".dat");
18
19      WeakFormGCM weakForm = new WeakFormGCM();
20      weakForm.setF(FC.c(0.0));
21      weakForm.setParam(
22          FC.c(-1.0),
23          param_c,
24          FC.c(2.0).M(b1),
25          FC.c(2.0).M(b2));
26      //q = d = 1 + \partial_{n}{lnu_0}
27      Function param_qd = FC.c(1.0).A(new DuDn(b1, b2, null));
28      //Robin: d*u + k*u_n = q
29      weakForm.setRobin(FC.c(-1.0).M(param_qd),
30          FC.c(-1.0).M(param_qd));
31
32      mesh.clearBorderNodeMark();
33      HashMap<NodeType, Function> mapNTF = new HashMap<NodeType,
34          Function>();
35      mapNTF.put(NodeType.Robin, null);
36      mesh.markBorderNode(mapNTF);
37
38      AssemblerScalar assembler = new AssemblerScalar(mesh, weakForm
39          );
40      System.out.println("Begin Assemble...solve_v");
41      assembler.assemble();
42      Matrix stiff = assembler.getStiffnessMatrix();
43      Vector load = assembler.getLoadVector();
44      System.out.println("Assemble done!");
45
46      SolverJBLAS solver = new SolverJBLAS();
47      Vector v = solver.solveDGESV(stiff, load);
48      plotVector(mesh, v, "Lagrangian_v"+s_i+".dat");
49      return v;
50  }
51
52  public Vector solve_lambda(int s_i, Vector v) {
53      Vector2Function fu0 = new Vector2Function(u0);
54      Vector2Function fu0_x = new Vector2Function(u0_x);
55      Vector2Function fu0_y = new Vector2Function(u0_y);
56
57      // -2*Laplace(lnu0) - ( a(x)-k^2 ) = -2*Laplace(lnu0) + ( k^2-a
58          (x) )
59      Vector2Function param_c = new Vector2Function(
60          FMath.axy(-2.0, laplace_ln_u0,

```

```

59         FMath.ax(1.0, FMath.axy(-1, a,
60             new SparseVector(a.getDim(), k*k)))));
61
62     // \Nabla\{lnu0\}
63     Function b1 = fu0_x.D(fu0);
64     Function b2 = fu0_y.D(fu0);
65     WeakFormGCM weakForm = new WeakFormGCM();
66
67     //(v - g)\partial\{\Omega\}
68     Vector v_g = FMath.axy(-1.0, FMath.axDivy(1.0, g[s_i], u0), v);
69     NodeList nodes = mesh.getNodeList();
70     Function fv_g = new Vector2Function(v_g);
71
72     weakForm.setF(FC.c(0.0));
73     weakForm.setParam(
74         FC.c(-1.0),
75         param_c,
76         FC.c(-2.0).M(b1),
77         FC.c(-2.0).M(b2)
78     );
79     Function param_d = FC.c(1.0).S(new DuDn(b1, b2, null));
80     weakForm.setRobin(FC.c(-1.0).M(fv_g),
81         FC.c(-1.0).M(param_d));
82
83     mesh.clearBorderNodeMark();
84     HashMap<NodeType, Function> mapNTF = new HashMap<NodeType,
85         Function>();
86     mapNTF.put(NodeType.Robin, null);
87     mesh.markBorderNode(mapNTF);
88
89     AssemblerScalar assembler = new AssemblerScalar(mesh, weakForm
90     );
91     System.out.println("Begin Assemble... solve_lambda");
92     assembler.assemble();
93     Matrix stiff = assembler.getStiffnessMatrix();
94     Vector load = assembler.getLoadVector();
95     System.out.println("Assemble done!");
96
97     SolverJBLAS solver = new SolverJBLAS();
98     Vector lambda = solver.solveDGESV(stiff, load);
99     plotVector(mesh, lambda, "Lagrangian_lambda"+s_i+".dat");
100    return lambda;
101 }

```

8 ScalaFEM

”If I were to pick a language to use today other than Java, it would be Scala” by James Gosling.

”I can honestly say if someone had shown me the Programming in Scala book by Martin Odersky, Lex Spoon & Bill Venner back in 2003 I’d probably have never created Groovy.” by James Strachan.

ScalaFEM provides wrapper classes for the classes in FuturEye to users so that things can be expressed more clearly, more concisely, or in an alternative style that looks more mathematically. Usually, we call this style of syntax within a programming language 'syntactic sugar' that is designed to make things easier to read or to express. The main purpose of the wrapper classes is for the syntactic sugar. We will discuss how it works when using the components: functions, weak forms, matrices and vectors etc. to perform your FEM computing.

8.1 Functions

Let's recall some function operations in FuturEye first. For example, we have a function

$$f(x) = x^2 + 2(x + 1)$$

One can construct this function through function combinations based on function x defined at **FMATH.X** (use: `import static edu.uta.futureeye.function.operator.FMath.*;`):

```
1 //f1 = x^2 + 2(x+1)
2 Function f1 = X.M(X).A( X.A(1.0).M(2.0) );
3 System.out.println(f1);
4 System.out.println(f1.value(new Variable(2.0)));
5 System.out.println(f1._d("x"));
6 System.out.println(f1._d("x").value(new Variable(3.0)));
```

Because Java doesn't allow one take function names as '+, -, *, /' we have to use 'A, S, M, D' instead. It looks a little clumsy. We print out the expression of this function at line 3. In order to evaluate this function, an object of **Variable** must be passed in to function `value(...)` of `f1`. We calculated the derivative of `f1` and print out the expression of the derivative at line 5. Finally, the derivative is evaluated at $x = 3.0$ at the last line of the code segment.

How does these things look like in Scala? Thanks for the syntactic sugar, we can do the same thing in this way:

```
1 //f1 = x^2 + 2(x+1)
2 var f1 = X*X + 2.0*(X + 1.0)
3 println(f1)
4 println(f1(2.0))
5 println(f1._d("x"))
6 println(f1._d("x")(3.0))
```

All things can be expressed more clearly and concisely now. In Scala, we can use '+, -, *, /' as function names and the operator precedence is consistent with that in mathematical context. The result is:

```
1 x * x + 2.0 * (x + 1.0)
2 10.0
3 x + x + 2.0
4 8.0
```

It should be noted that we write '`2.0*(X + 1.0)`' instead of '`(X + 1.0)*2.0`' (It's also correct expression) in the expression of `f1`. In Java, you have to write '`X.A(1.0).M(2.0)`' because `M(...)` is a function defined in interface **Function** you can not write '`2.0.M(A(1.0))`'. Of course, you can write it in this way: '`C(2.0).M(A(1.0))`'. Here `C(...)` is the constant function wrapper defined in **edu.uta.futureeye.function.operator.FMath**. However, it is so clumsy. The expression looks so friendly in Scala. How do you make it possible? The answer is that we have defined the implicit conversion functions in Scala object **FMATH**:

```

1 implicit def int2MathFun(i: Int): MathFun = C(i)
2 implicit def long2MathFun(l: Long): MathFun = C(l)
3 implicit def float2MathFun(f: Float): MathFun = C(f)
4 implicit def double2MathFun(d: Double): MathFun = C(d)

```

So 2.0 is automatically converted to $C(2.0)$ in this case.

The evaluation of function $f1(2.0)$ at line 4 is just like what you see in mathematical context. This syntactic sugar is that when calling $f1(2.0)$, you are actually calling $f1.apply(2.0)$. We have defined the following functions for this purpose

```

1 def apply(v: Double): Double
2 def apply(name: String, value: Double): Double

```

The name of the variable can be omitted when there is only one variable in the function. You can also explicitly specify the name of the variable like this:

```

1 println(f1("x", 2.0))

```

It is a little regret that we didn't define function `^^` for pow operation like X^2 . Because operator precedence in Scala is fixed now and it is not correct for `^^` in mathematical context. If we defined it one have to write parenthesis around the term X^2 . This may cause errors when one forgot to write the parenthesis. However, one can use function `pow(...)` to do the same operation.

Next example, we will show how it works for user defined multivariable functions in Scala. We have function

$$f(x) = 3x^2 + 4y + 1$$

This time, we don't use function combinations instead we define an anonymous class in Java that extends **AbstractFunction**. It will be more efficient than function combinations in the meaning of computing time. What we need to do is over ride functions `value(...)` and `_d(...)` to provide the function of evaluation and derivation.

```

1 //f2 = 3x^2 + 4y + 1
2 Function f2 = new AbstractFunction("x","y") {
3     @Override
4     public double value(Variable v) {
5         double x = v.get("x");
6         double y = v.get("y");
7         return 3*x*x + 4*y + 1;
8     }
9     @Override
10    public Function _d(String vn) {
11        if (vn == "x") return C(6).M(X);
12        else if(vn == "y") return C(4);
13        else throw new FutureyeException("variable_name="+vn);
14    }
15 }.setFName("f(x,y) = 3x+4y+1");
16
17 System.out.println(f2);
18 Variable vv = new Variable("x",2.0).set("y",3.0);
19 System.out.println(f2.value(vv));
20
21 System.out.println(f2._d("x"));
22 System.out.println(f2._d("y"));

```



```
23 System.out.println(f2._d("z")); //Exception
```

Function `setFName(...)` is called here to specify a name for this function and it will be printed out at line 17. This function returns the object itself so it can be call at the end of the definition of the anonymous class and assigned to `f2`. `f2` is evaluated at $x = 2.0, y = 3.0$ at line 18. The last three lines of the code segment print out the derivatives of `f2`. An exception will be generated, because 'z' is not a variable of `f2`. In Scala, you can do this as below:

```
1 //f2 = 3x^2 + 4y + 1
2 val f2 = MathFun("x", "y")(
3     v => 3*v("x")*v("x")+4*v("y")+1,
4     vn => if (vn == "x") 6*x
5           else if(vn == "y") 4
6           else error("variable_name="+vn)
7     ).setFName("f(x,y) = 3x^2+4y+1")
8
9 println(f2)
10 val vv = Variable( ("x",2.0), ("y",3.0) )
11 println(f2.value(vv))
12
13 val vv2 = Variable("x",2.0)("y",3.0)
14 println(f2.value(vv2))
15
16 println(f2( ("x",2.0), ("y",3.0) ))
17
18 println(f2._d("x"))
19 println(f2._d("y"))
20 println(f2._d("z")) //Exception
```

Here, object **MathFun** have member function `apply(...)` defined as:

```
1 def apply(names: String*)(
2     f: edu.uta.futureye.function.Variable => Double,
3     df: String => MathFun = null
4 ): MathFun
```

The first parameter `names` of type **String*** is a vararg parameter, it is used to indicate variable names of a function. The vararg must be the last parameter in the parameter list but we have two other parameters `f` and `df` followed after the first parameter. A solution in Scala is to use function currying. What function currying means is that a function that takes multiple arguments in such a way that it can be called as a chain of functions each with a single argument (partial application). The `apply(...)` method is defined in a way that it takes a vararg parameter and returns a functional as result. Then this functional takes two parameter `f` and `df` and returns the final result.

Let's take a look at the type of arguments `f` and `df`. They are functions. For example, `String => MathFun` means a function which takes a parameter of type **String** and returns a value of type **MathFun**. In Scala, you can pass functions as parameters to a function (we call this function higher-order functions).

Function literals are used to define two parameters of function type at line 3-6. A function literal is an alternate syntax for defining a function. It's useful for when you want to pass a function as an argument to a higher-order function but you don't want to define a separate function. The following function

```
1 v => 3*v("x")*v("x")+4*v("y")+1
```

will be called when function f_2 is evaluated. $v("x")$ and $v("y")$ refer the values of variable x and y . The following function

```
1 vn => if (vn == "x") 6*X
2       else if(vn == "y") 4
3       else error("variable_name="+vn)
```

gives the definition of the derivatives of function f_2 . The return value, for example $\frac{d(f_2)}{dy} = 4$, will be converted to type **MathFun** implicitly if it is a native number.

Three times of function evaluation of f_2 at $x = 2.0$, $y = 3.0$ is performed at line 10-16 of the code segment. In the first evaluation, a variable is created by passing two tuples to the parameters of object **Variable** and evaluated by calling function $value(...)$ of f_2 . The tuples here are only syntactic sugar for the class `tuple2`. The function $apply(...)$ of object **Variable** is defined as

```
1 def apply(tuples: Tuple2[String, Double]*): Variable
```

In the second evaluation, the construction of variable object looks a little bit neat. It's just a small trick. Define the function

```
1 def apply(name: String, value: Double): Variable
```

in both object **Variable** and class **Variable**, then we got this syntactic sugar for the variable creation. In the last evaluation, you can even only use the parenthesis and pass the tuples as the parameters to evaluate function f_2 . It is implemented by defining

```
1 def apply(tuples: Tuple2[String, Double]*): Double
```

in object **MathFun**. It's pretty cool! It can be seen as the standard way to evaluating multivariable functions. Let's finish this example by giving the output of the code segment mentioned above:

```
1 f(x,y) = 3x^2+4y+1
2 25.0
3 25.0
4 25.0
5 6.0 * x
6 4.0
```

Our third example is about how to create vector valued functions. Consider vector function

$$\vec{f}(x,y) = \begin{pmatrix} x+y \\ x*y \\ 1 \end{pmatrix}$$

In Java you can create this function by using class **SpaceVectorFunction**:

```
1 //f3 = (x+y, x*y, 1.0)
2 VectorFunction f3 = new SpaceVectorFunction(
3     new AbstractFunction("x","y") {
4         @Override
5         public double value(Variable v) {
6             return v.get("x") + v.get("y");
7         }
8     })
```

```

8         }.setFName("x+y"),
9         X.M(Y),
10        C1
11    );
12    System.out.println(f3);
13    System.out.println(f3.value(new Variable("x",2.0).set("y",2.0)));

```

Three function components are passed into the constructor of class **SpaceVectorFunction**. In Scala, we defined object **MathVectorFun** to do the same thing:

```

1    val f3 = MathVectorFun(
2        MathFun("x","y"){ v => v("x") + v("y") }.setFName("x+y"),
3        X*Y,
4        1.0
5    )
6    println(f3)
7    println(f3( ("x",2.0),("y",2.0) ))

```

There is no new tricks in grammar. It just should be noted that implicit type conversion occurs at the third component of creating function *f3*. The result is

```

1    [x+y, x * y, 1.0]
2    (4.0  4.0  1.0)

```

8.2 Meshes and Elements

In this section, we talk about how to read a mesh and assign finite element types to elements on a mesh in Scala. This process can be implemented in Java as

```

1    Mesh mesh = new MeshReader("triangle.grd").read2DMesh();
2    mesh.computeNodeBelongsToElements();
3    ...
4    ElementList eList = mesh.getElementList();
5    FELinearTriangle feLT = new FELinearTriangle();
6    for(int i=1;i<=eList.size();i++)
7        feLT.assignTo(eList.at(i));

```

First, we read a 2D mesh from file "triangle.grd" and compute the relation between nodes and elements at line 2. Then we assign **FELinearTriangle** finite element for each elements on the mesh.

In Scala, we can use the wrapper objects and classes of **MeshReader** and **Mesh**. May be you have noted that there are some differences when we call function *read2DMesh()* and function *computeNodeBelongsToElements()*. At the first line we use the dot notation to call a function of arity-0 (no arguments) and at the second line we drop the dot and the parenthesis (suffix notation). However, This style should be used with great care. In order to avoid ambiguity in Scalas grammar, any method which is invoked via suffix notation must be the last item on a given line. Also, the following line must be completely empty, otherwise Scalas parser will assume that the suffix notation is actually infix and will (incorrectly) attempt to incorporate the contents of the following line into the suffix invocation.

The last 2 lines do the work of assigning **FELinearTriangle** finite element for each elements on mesh. The '_' acts as a placeholder for parameters.

```

1  val mesh = MeshReader("triangle.grd").read2DMesh()
2  mesh.computeNodeBelongsToElements
3
4  ...
5  val feLT = new FELinearTriangle()
6  mesh.elements.foreach(feLT.assignTo _)

```

Function *foreach(...)* is defined in wrapper class **ElementList** as below:

```

1  def foreach[B](F: Element => B): Unit = {
2      var i = 0
3      while(i < objs.size()) {
4          F(objs.get(i))
5          i += 1
6      }
7  }

```

It is a functional style in Scala and do what just like the **for** do in Java, but that is an imperative style. The **foreach** of **ElementList** is the same as that in **Array**, **List** etc. Alternatively, we could have defined function *foreach(...)* in Java for class **ElementList** as:

```

1  import scala.Function1;
2  ...
3  public <B> void foreach(Function1<Element, B> F) {
4      int i = 0;
5      while (i < this.objs.size()) {
6          F.apply(this.objs.get(i));
7          i += 1;
8      }
9  }

```

This would give the same result as that defined in the wrapper class **ElementList** in Scala.

8.3 Weak Form

In ScalaFEM, you can still use the weak form classes that defined in package **edu.uta.futureye.lib.weakform**. There is nothing changes for these classes. The express way to create your own weak form by using class **WeakFormBuilder** is almost the same as what you do in Java. For example, the follow code segment demonstrates how to define the weak form $A(u, v) = (f, v)$ where $A(u, v) = (\nabla u, \nabla v)$:

```

1  val wfb = new WeakFormBuilder() {
2      override def makeExpression(e: Element, tp: Type): Function =
3          {
4              val u = MathFun(getScalarTrial())
5              val v = MathFun(getScalarTest())
6              val ff = getParam("f", e)
7              tp match {
8                  // A(u, v) = u_x*v_x + u_y*v_y
9                  case Type.LHS_Domain => return grad(u, "x", "y") dot
10                     grad(v, "x", "y")
11                  case Type.RHS_Domain => return ff.M(v);
12              }
13          }
14  }

```

```

11         null
12     }
13 };
14 //Right hand side(RHS): f = -2*(x^2+y^2)+36
15 wfb.addParam("f", -2.0*(X*X+Y*Y)+36.0);

```

The functions *addParam(...)* and *getParam(...)* is used to pass coefficient parameters for the weak form, e.g. $f = -2(x^2 + y^2) + 36$ which are used in the expression of the weak form. The weak form defined above corresponds to the following problem in mathematical context:

Find $u(x, y)$, such that

$$\begin{aligned}
 -\Delta u(x, y) &= f(x, y), & (x, y) \in \Omega \\
 u(x, y) &= 0, & (x, y) \in \partial\Omega
 \end{aligned}$$

where $f(x, y) = -2(x^2 + y^2) + 36$ On the boundary $\partial\Omega$ it is assigned with Dirichlet condition.

Using the phenomenal powers of closures, currying and higher order functions in Scala, we can reduce the amount of code significantly and make the definition of weak form concisely. We introduced a new class **PDE**. The weak form of the above problem can be defined as:

```

1 val eq1 = new PDE
2 val f = -2.0*(X*X + Y*Y) + 36.0
3 eq1.integrate(LHS, Inner) {
4     (u,v) => (grad(u, "x", "y") dot grad(v, "x", "y"))
5 }.integrate(RHS, Inner) {
6     (u,v) => f.M(v)
7 }

```

First, we defined an object *eq1* of type **PDE** and a function *f* (of type **MathFun**). Then we call the member function *integrate(...){...}* of *eq1*. In the parentheses, the enumeration LHS and RHS which are defined in object **HandSide** represent left hand side and right hand side of the equation. The enumeration Inner (and Border) are defined in object **DomainType** represent the domain on which the integration will be performed. The function literals in the braces give the expression of the weak form. Function *integrate(...){...}* returns the object itself, so we can use function chaining to express different terms in the weak form.

It should be noted that function *f* is not a parameter now. Here *f* is a 'free variable', because it isn't declared as one of the function arguments nor introduced locally. A function containing at least one free variable is called an 'open term' in Scala. In order to get to a fully functional function, all free variables have to be bound (turning that open term into a closed term). For this to be done, the compiler reaches out to the so called lexical environment, in which that function was defined and tries to find a binding. This process is called closing over which results in a closed expression: a 'closure'. Let's see a more complicated example about closure:

Find $u(x, y)$, such that

$$\begin{aligned} -k\Delta u(x, y) + cuv &= f(x, y), & (x, y) \in \Omega \\ u(x, y) &= 0, & (x, y) \in \Gamma_D \\ du(x, y) + k \frac{\partial u}{\partial n} &= q, & (x, y) \in \Gamma_R \end{aligned}$$

where $f(x, y) = -2(x^2 + y^2) + 36$, $k = d = q = 1$ and $c = \sqrt{x^2 + y^2}$. Γ_R is the boundary of Ω that assigned with Robin condition, Γ_D is the boundary of Ω that assigned with Dirichlet condition.

```

1  val eq1 = new PDE
2  val k = C1
3  val c = sqrt(X*X + Y*Y)
4  val f = -2.0*(X*X + Y*Y) + 36.0
5  val d = C1
6  val q = C1
7  eq1.integrate(LHS, Inner) {
8      (u,v) => (grad(u, "x", "y") dot grad(v, "x", "y")) + c*u*v
9  }.integrate(LHS, Border) {
10     (u,v) => d*u*v
11  }.integrate(RHS, Inner) {
12     (u,v) => f*v
13  }.integrate(RHS, Border) {
14     (u,v) => q*v
15  }

```

We can see all the coefficients of the weak form and right hand side f are not parameters in the definition of the weak form. It's very concisely and close to the expression in mathematical context.

8.4 Matrix and Vector Operations

By defining the *apply(...)* and *update(...)* member functions in matrix and vector classes, the following syntactic sugar allows users to access matrix and vector components by parenthesis.

```

1  var v = new SpaceVector(3)
2  v(1) = 1
3  v(2) = 2
4  v(3) = 3
5  println(v(2))
6
7  var m = new SparseMatrixRowMajor(3,3);
8  m(1,2) = 5;
9  println(m(1,2))

```