向量问题是指偏微分方程组。在这样的问题中，解构成一个向量函数。这个模块解释了如何使用类：FESystem，它使得我们对向量问题的代码可以很类似于标量方程的代码。

比如对于欧拉方程组而言，守恒形式的自变量为：



若每个分量都使用一次标量基函数来近似，则共需16个向量基函数，分别为：









使用函数：

std::pair< unsigned int, unsigned int > FiniteElement< dim, spacedim >::

**system\_to\_component\_index (const unsigned int index )** const

返回一个整数pair：

第一个数表示了当前向量基函数的唯一非零分量是第几个分量component；

第二个数表示此向量基函数的非零分量对应的标量基函数在其所属的全部标量基函数中的index(是第几个标量基函数)。当然，通常在用户代码中不需要关心这个值。

比如，对于上面的向量基函数，调用此函数后，pair的第一个数为2，表示component 2是非零分量，即代表了当前基函数是用于近似第3个分量密度(在编写用户代码时，自由度qi代表了哪个分量，需通过询问对应的向量基函数的非零分量是第几个来确定)。第二个数为0，表示对应的底层标量基函数是第0个标量基函数。

**Examples of vector-valued problems**

我们处理向量值问题的方式基本上与处理标量值问题的方式是一样的：首先需要一个弱形式，之后生成系统矩阵。

**Linear elasticity**

以step-8中的弹性问题为例，为了简化问题突出概念，选取λ=0及μ=1。因此，考虑下述弱形式：寻找

**Mixed elliptic problems**

三维中的混合laplace方程，速度场与压力梯度有关，且速度场有源项：



这里，有四个未知数：标量，速度向量的三个分量。 注意，速度空间**V**不仅仅是三个相同空间的拷贝。

通过如下步骤将问题转化为代码：

* **step1**：先将解写为向量形式，将运算符写为矩阵形式：

 →

注：通过所谓的extractor可以提取出**U**的某些成分，如定义symbolic names：“velocities”和“pressure”，则有**U**[velocities]=**u**，是dim维的向量；**U**[pressure]=p，是个标量。

* **step2**：左乘一个向量测试函数并积分：



注：这里同样包含分量为：，且可提取出：，

是个向量，是个标量。

* **step3**：把不同的项乘出来：

****

* **step4：**必要的时候采用分部积分（这里是可以把对p的正则性要求转移到上去）：



* **step5**：把解向量展开为:



注：这里的展开方式，是把基函数视为向量函数，而非把基函数看作标量函数并把系数看作向量。

带入到step4的式子中去，测试函数取为某一个基函数，则：



* **step6**：把双线性型(右端项类似)：翻译成源代码，需要用到下列等式：

**=**fe\_values[velocities].value(i,q)

**=**fe\_values[pressure].value(i,q)

**=**fe\_values[velocities].divergence(i,q)

注：fe\_values代表了基函数(shape function)(在积分点处)的值。fe\_values[velocities]则代表了基函数的“速度”成分。从而，fe\_values[velocities].value(i,q)表示：**第i个基函数(向量)的速度成分在q积分点处的值**，这也就是****的含义。

最终，得到的代码应该是这样的：

for(int q=0; q<n\_q\_points; ++q)

for(int i=0; i<dofs\_per\_cell; ++i){

Tensor<1, dim> phi\_i\_u = fe\_values[velocities].value(i,q); //即，是个向量

double div\_phi\_i\_u = fe\_values[velocities].divergence(i,q); //即，是个标量

double phi\_i\_p = fe\_values[velocities].value(i,q); //即，是个标量

for(int j=0; j<dofs\_per\_cell; ++j){

Tensor<1, dim> phi\_j\_u = fe\_values[velocities].value(j,q); //即，是个向量

double div\_phi\_j\_u = fe\_values[velocities].divergence(j,q); //即，是个标量

double phi\_j\_p = fe\_values[velocities].value(j,q); //即，是个标量

local\_matrix(i,j) += (phi\_i\_u \* phi\_j\_u - div\_phi\_i\_u \* phi\_j\_p – phi\_i\_p \* div\_phi\_j\_u)

\* fe\_values.JxW(q);

//即

}

}

**Describing finite element spaces**

一旦定好了双线性型及泛函设定，我们就需要寻找一种方式来描述向量有限元空间（从中绘制解和测试函数）。由此引入FESystem类：它由简单有限元组成。

我们定义了解、形函数、测试函数都为向量，即含有多个分量。而每个分量通常都由简单有限元构建。

例如：用于2d Stokes流的Taylor-Hood有限元：

 (或)

可表示为：

FESystem<2> stokes\_element(FE\_Q<2>(2),1 //one copy of FE\_Q (2) for ux (二次有限元)

FE\_Q<2>(2),1 //one copy of FE\_Q (2) for uy (二次有限元)

FE\_Q<2>(1),1) //one copy of FE\_Q (1) for p（线性有限元）

或表示为：

FESystem<2> stokes\_element(FE\_Q<2>(2),dim //dim copy of FE\_Q (2) for u(二次有限元)

FE\_Q<2>(1),1) //one copy of FE\_Q (1) for p（线性有限元

在弹性问题的例子中，我们需要dim个同样有限元的拷贝，例如：

FESystem<dim> elasticity\_element(FE\_Q<dim>(1), dim);

这会产生一个dim维的向量值空间，其中每个分量都是一个类型为FE\_Q的连续双线性有限元。它会有对应于FE\_Q的dim倍数量的基函数。

对于混合Laplacian问题，情况要复杂一些。首先因为，，所以我们必须选定一对离散空间：。一种选择是稳定Raviart-Thomas对：

FESystem<dim> rt\_element (FE\_RaviartThomas<dim>(k), 1, //**u**

FE\_DGQ<dim>(1), 1); //p

这个系统中的第一个有限元本身就已经是一个dim维的向量值有限元，而第二个是个常规的标量有限元。

...//省略了一段

**Internal structure of FESystem**

FESystem有一些内部变量，反映了由构造函数建立的内部结构。这些也可以在之后被用户程序用于传递结构给矩阵组装和线性代数。我们针对上面的例子给出这些变量的名称和值：

|  |  |  |  |
| --- | --- | --- | --- |
| **System Element** | **FiniteElementData::n\_blocks()** | **FiniteElementData::n\_components()** | **FiniteElement::n\_base\_elements()** |
| elasticity\_element | dim | dim | 1 |
| rt\_element | 2 | dim+1 | 2 |
| ldg\_equal\_element | 2 | dim+1 | 2 |
| ldg\_convoluted\_element\_1 | dim+1 | dim+1 | 1 |
| ldg\_convoluted\_element\_2 | dim+1 | dim+1 | 2 |

**Assembling linear systems**

下一步是组装线性系统。对于上述弱形式的不同特征及创建的不同有限元系统，我们有一些选择。整个概念可能用一个例子来解释最好：我们考虑，在上面的混合laplace方程组中，如何把弱形式在一个cell上的局部贡献组装起来。

**A single FEValues and FEValuesExtractors**

这实际上是step-20中的处理办法：

const FEValuesExtractors::Vector velocities (0);

const FEValuesExtractors::Scalar pressure (dim);

...

typename DoFHandler<dim>::active\_cell\_iterator

cell = dof\_handler.begin\_active(),

endc = dof\_handler.end();

for (; cell!=endc; ++cell)

{

fe\_values.reinit (cell);

local\_matrix = 0;

local\_rhs = 0;

right\_hand\_side.value\_list (fe\_values.get\_quadrature\_points(), rhs\_values);

for (unsigned int q=0; q<n\_q\_points; ++q)

for (unsigned int i=0; i<dofs\_per\_cell; ++i)

{

for (unsigned int j=0; j<dofs\_per\_cell; ++j)

local\_matrix(i,j) += (fe\_values[velocities].value (i, q) \*

fe\_values[velocities].value (j, q)

-

fe\_values[velocities].divergence (i, q) \*

fe\_values[pressure].value (j, q)

-

fe\_values[pressure].value (i, q) \*

fe\_values[velocities].divergence (j, q)) \*

fe\_values.JxW(q);

local\_rhs(i) += - fe\_values[pressure].value (i, q) \*

rhs\_values[q] \*

fe\_values.JxW(q);

}

代码中发生了这样一些事：

* 我们做的第一件事是声明“extractors”(参见FEValuesExtractors命名空间)。这些对象基本上就只是存储了一个向量有限元的哪个分量构成了一个单一的标量分量，或一个一阶张量(即所谓的“物理向量“，总由dim个分量构成)。这里，我们声明了一个表示速度的对象，由dim个分量组成，从0分量开始. 以及一个表示压力的extractor，是个在dim位置的标量。
* 然后进行cells上、形函数上、积分点上的循环。在最内层循环中，我们计算一对形函数对全局矩阵的局部贡献以及右端向量。一个cell对双线性型的贡献是这样的

（基于形函数为 ）：



其实现是这样的：

local\_matrix(i,j) += (fe\_values[velocities].value (i, q) \*

fe\_values[velocities].value (j, q)

-

fe\_values[velocities].divergence (i, q) \*

fe\_values[pressure].value (j, q)

-

fe\_values[pressure].value (i, q) \*

fe\_values[velocities].divergence (j, q)

) \*

fe\_values.JxW(q);

* 事实上，上面代码发生的事是这样的：当你使用fe\_values[pressure]时，一个所谓的“view“被创建，即，一个不像全FEValues对象的对象，而是只代表了一个有限元的部分分量，即由extractor对象pressure或velocities代表的那些部分。
* 这些view在之后被询问关于这些独立分量的信息。例如，当你写fe\_values[pressure].value(i,q)时，你得到第i个形函数**V**i的pressure分量在第q个积分点处的值。因为extractor pressure代表了一个标量分量，这个操作的结果是一个标量值。但调用fe\_values[velocities].value(i,q)会产生全部dim个分量的值，其类型是Tensor<1, dim>
* 还可以询问view关于形函数的某个分量的梯度值。例如，fe\_values[pressure].gradient(i,q)代表了标量分量pressure的梯度值，属于类型Tensor<1,dim>，然而速度分量的梯度，fe\_values[velocities].gradient(i,q)是一个Tensor<2,dim>，即一个由元素组成的矩阵 。

最后，无论标量还是矢量views都可以被询问二阶导数(Hessians)的信息，矢量views可以被询问对称梯度的信息，对称梯度被定义为： ，以及散度信息：。

**An alternative approach（采用所谓的primitive element，即采用的有限元的每个向量基函数只含一个非零分量）**

在有些情况下，可以优化组装矩阵或右端项的工作，前提是知道使用的有限元。比如考虑弹性方程中的双线性型(最早在step-8中考虑过)：



这里，**u**是有dim个分量的向量函数，**v**是对应的测试函数，λ和μ是材料参数。实现这个双线性型的方式是如下，使用一个extractor对象把有限元的全部dim个分量作为单个向量来解释，而不是离散成标量分量：

const FEValuesExtractors::Vector displacements (0);

...

for (unsigned int q\_point=0; q\_point<n\_q\_points; ++q\_point)

for (unsigned int i=0; i<dofs\_per\_cell; ++i)

{

const Tensor<2,dim> phi\_i\_grad

= fe\_values[displacements].gradient (i,q\_point);

const double phi\_i\_div

= fe\_values[displacements].divergence (i,q\_point);

for (unsigned int j=0; j<dofs\_per\_cell; ++j)

{

const Tensor<2,dim> phi\_j\_grad

= fe\_values[displacements].gradient (j,q\_point);

const double phi\_j\_div

= fe\_values[displacements].divergence (j,q\_point);

cell\_matrix(i,j)

+= (lambda\_values[q\_point] \*

phi\_i\_div \* phi\_j\_div

+

mu\_values[q\_point] \*

double\_contract(phi\_i\_grad, phi\_j\_grad)

+

mu\_values[q\_point] \*

double\_contract(phi\_i\_grad, transpose(phi\_j\_grad))

) \*

fe\_values.JxW(q\_point);

}

}

//省略了部分

**Block solvers**

由上面的方法得到的矩阵往往是非正定甚至非对称的，这使得求解起来没有很好的办法，只能采用迭代方法。但可以通过利用问题的结构来优化。

例如，混合laplace方程的算子形式为：****

最终得到的矩阵形式为：，它与上面的算子形式是对应的。

解释：

* 我们在代码中像这样构建bilinear form是因为这样很方便
* 然而，在实际中，形函数仅仅在u或p分量非零，即：它们要么是要么是

那么，如果index i对应的形函数是属于u-type的，则我们得到一个矩阵行为：



如果index i是属于p-type的，则



对于index j同理。

* 如果我们enumerate dofs的时候使得全部u-type的排在前面，p-type在后面，那么这就会形成一个分块的矩阵，即：



这里的M表示来自于的项，B对应，BT对应

我们在线性求解器中会利用这种结构！

例如在step-20中，我们对线性系统进行block eliminattion：

 

第一个方程乘以并减去第二个方程，得：

这个方程现在只含有压力变量P。在求解了它之后，再使用得到速度场。

其优势在于矩阵和M都是对称正定的。

**Extracting data from solutions**

一旦我们计算出了解，常常需要我们估算这个解在积分点上的值，例如：为了估算下一个牛顿迭代步的非线性残值，为了估算有限元残值从而得到误差指示器，或者为了在时间相关问题中计算下一时间步的右端项。

以上这些，仍然是通过使用FEValues对象，从而估算形函数在积分点处的值的，当然，也还是需要用到有限元函数的值。对于上面例子中的混合laplace问题，考虑下述代码：

std::vector<Vector<double>> local\_solution\_values(n\_q\_points, Vector<double> (dim+1));

//存储积分点处的解

typename DoFHandler<dim>::active\_cell\_iterator cell=dof\_handler.begin\_active(),

endc=dof\_handler.end();

for(; cell!=endc; ++cell)

{

fe\_values.reinit(cell);

fe\_values.get\_function\_values(solution, local\_solution\_values);

}

/\*上面程序有点类似于：

std::vector<types::global\_dof\_index> dof\_indices(dofs\_per\_cell); //存储全局dof索引

std::vector<double> local\_dof\_values; //存储局部自由度值, 即局部解向量

cell->get\_dof\_indices(dof\_indices); //获得局部自由度索引对应的全局索引

local\_dof\_values[i] = solution(dof\_indices[i]); //获得局部自由度值, 即局部解向量

\*/

经过这之后，变量local\_solution\_values就保存了当前网格积分点处的共n\_q\_point个向量，每个向量都代表了当地的解，含dim+1个分量（dim个为速度分量，多一个为压力分量）

那么我们就可以利用这些值来构造其他东西，例如残值。

...

**Generating graphical output**

...