PROGRAMMING FINITE ELEMENTS

NICOLA CAVALLINI

1. Introduction

We write a brief paper meant to support MHPC lectures on scientific programming environment. The goal of this classes is to provide students scientific programming tools to develop scientific applications. The author is an engineer, with a Ph.D. in numerical analysis and hopefully these brief notes will hold the best habits of these two education paths. Achieving my goal no matter what, and in the most efficient way. Being passionate about sophisticated mathematical structures. I believe the Finite Element method is a good way acquire programming skills in scientific computing.

2. Strong Problem

Scientists solve problems. Among the pletora of existing problems we need to pick one, simple enough to be solved during the time dedicated to a single class, and sophisticated enough to be representative for more sophisticated ones. The answer is the Poisson's problem, modeling the diffusion of temperature in a body.

Given a domain $\Omega \subset \mathbb{R}^2$. Find u such that:

$$\begin{cases} -\Delta u = f & \text{in } \Omega \\ u = 0 & \text{on } \partial\Omega \end{cases}$$

Just one remark:

$$\Delta u = \partial_{xx} u + \partial_{yy} u,$$

where x and y are the two coordinates in $\mathbb{R}^2$.

In other words, the Poisson's problem is asking for a type of solution that is characterised by having two continuous derivatives, in mathematical sense we ask for $u \in C^2$. *It can be proved* that such a solution exists and is unique. It is also true that such a solution can be *found* in simple and specific cases. In the next section we explore an alternative strategy.

3. Continous Weak Problem

In the previous section we looked at the Poisson's problem and noticed that searching for its solution in a strong form requires looking for a C^2 solution. This requirement is that strong, that can only be fulfilled in simple cases. The Finite Element (Method) si not really a method, it is rather the art of looking for solutions in vector spaces simpler than the one needed by the strong form.

We understand that a key role is played by the space V substituting the C^2 space. Assuming V is continuous (has an infinite set of basis functions) the first idea is to project the problem onto this particular space. Mathematically this is written as:

$$-\int_\Omega \Delta u\, v = \int_\Omega f\, v \quad \forall v \in V$$

If the projection procedure in functional spaces looks cryptical, we can compare it to the geometrical projection. Consider u a vector and V the $\mathbb{R}^2$ space. The projection of u onto V is:

$$u = u_1 \cdot \mathbf{e}_1 + u_2 \cdot \mathbf{e}_2,$$

being $\mathbf{e}_1$, $\mathbf{e}_2$ the basis vectors for $V = \mathbb{R}^2$. The analogy here should be clear once we think that $\mathbf{e}_i$ play the same role as v.

If we want to really get rid of the second derivatives in our problem we can integrate by parts:

$$\int_\Omega \nabla u \cdot \nabla v - \int_{\partial\Omega} u\, \nabla v \cdot \mathbf{n} = \int_\Omega f\, v.$$

The boundary integral calls for the boundary conditions to come into play. Homogenous boundary conditions require $u = 0$ on $\partial\Omega$. This requirement has a very simple meaning: "we do not need to project our solution on the boundary", as consequence: $v = 0$ on $\partial\Omega$. We are left with:

$$\int_\Omega \nabla u \cdot \nabla v = \int_\Omega f\, v.$$

For simplicity, we look for u in the very same space as v. Notice that this is a choice that works pretty well in this case, but remember, Finite Elements is an art, not a method. This choice is arbitrary and there are cases in which this choice fails. The weak formulation of our original problem, looks like:

Find $u \in V$, such that:

$$\int_\Omega \nabla u \cdot \nabla v = \int_\Omega f\, v, \quad \forall v \in V.$$

The relation between the weak problem and the strong one is clarified by the Lax-Milgram theorem. This proofs that the weak form is equivalent to the strong one, meaning we can approximate the continuous weak problem to get the solution of the strong one. We conclude this section recalling that:

$$\int_\Omega \nabla u \cdot \nabla v = \int_\Omega \begin{pmatrix} \partial_x u \\ \partial_y u \end{pmatrix} \cdot \begin{pmatrix} \partial_x v \\ \partial_y v \end{pmatrix} = \int_\Omega (\partial_x u\, \partial_x v + \partial_y u\, \partial_y v)$$

4. Discrete Weak Problem

To plug the solution of the Poisson's problem inside a computer we need to discretise it. F. Brezzi has a very nice way to explain the relation between the continuous form and the discrete one: "Just plug h everywhere!"

Find $u_h \in V_h$, such that:

$$\int_\Omega \nabla u_h \cdot \nabla v_h = \int_\Omega f\, v_h, \quad \forall v_h \in V_h.$$

Of course the this graphical operation has some deep mathematical consequences that are explained in a very elegant way by the Céa's Lemma. We express the meaning of this lemma with the sketch in Figure 1. If V_h is a two dimensional plane and u a point in three dimensions. The distance between u and u_h is the minimal among the all possible distances between u and any v_h.

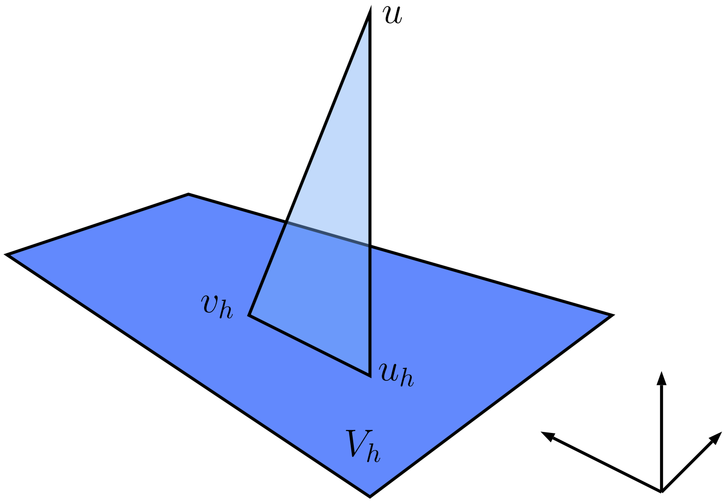


FIGURE 1. We represent in a three dimensional sketch the meaning of the Céa's Lemma. If V_h is a two dimensional plane and u a point in three dimensions. The distance between u and u_h is the minimal among the all possible distances between u and any v_h.

5. Computational Solution of the Discrete Problem

The first step toward the discretisation of the problem, is the discretisation of the domain. In Figure 2 we represent the domain Ω and its discretisation, namely the triangulation $\mathcal{T}_h$. In this lecture notes we are going to consider triangular elements and linear shape functions.

There are a pletora of possible basis functions that can be adopted on a discretised domain. A subset of this pletora are defined on triangles, and among these we pick continuous linear functions, namely called P_1. Mathematically we can define the vector space V_h:

$$V_h = \{\phi \in H^1_0\Omega : \phi|_K \in P_1(K) \forall K \in \mathcal{T}_h\}.$$

$H^1_0(\Omega)$ is an Hilbert space, and ϕ is the collection of our basis functions, K denotes the single triangle. We agree with the reader that this expression can be cryptic. In Figure 3 we sketched the three linear basis functions on a triangle. The reader should notice that

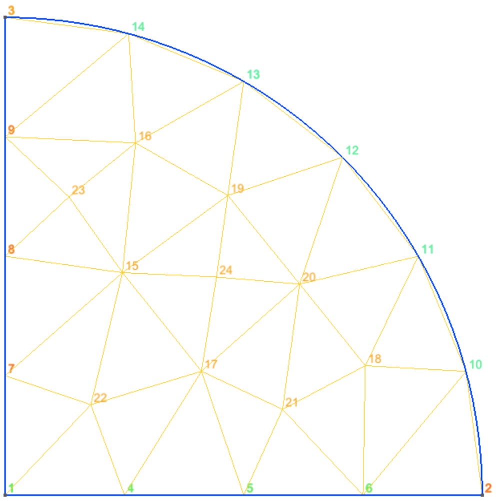


Figure 2. A typical domain discretisation, in this case the original domain is a quarter of a circle.

the i-th basis function is one on the i-th vertex and zero on the others. The three points are linearly connected in space. The linear functions are connected at each vertex forming a sort of tent. Figure 2 highlights the global numbering of the vertices. Recall that in our problem $u_h \in V_h$, meaning that on every single element:

$$u_h(x, y) = \sum_{i=1}^{3} u^j \phi_j(x, y)$$

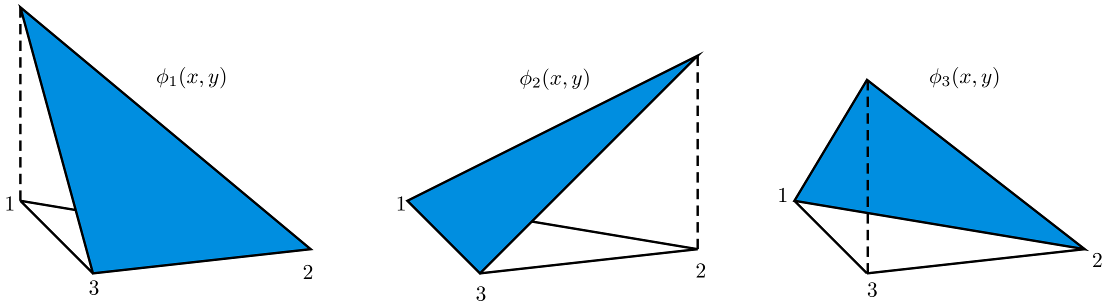


Figure 3. Local linear shape functions on a triangular element.

Figure 4 explains how the local basis functions are connected releasing the continuity of the solution space.

The last ingredient to establish an automatic strategy to solve our problem is to numerically compute an integral. Techniques that perform this task are called quadrature

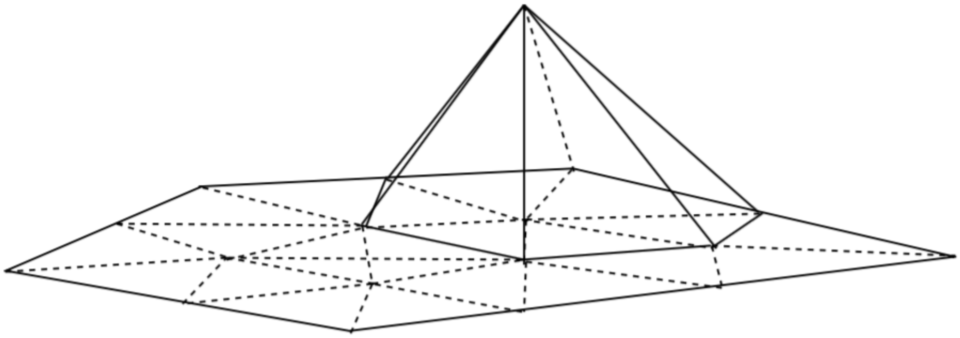


FIGURE 4. Global shape function on a domain. We can notice how the interelement continuity is achieved.

formulas. Among these, one that is particular suited for our linear case is the trapezoidal rule:

$$\int_K \psi(x,y) \approx \text{area}(K)\frac{1}{3}\sum_{q=1}^{3}\psi(x_q,y_q)$$

where $\psi(x_q, y_q)$ is the integrand evaluated at the q-th quadrature point.

Now all the tools are in place, let's substitute in the left hand side (the most tricky one):

$$\begin{aligned}
\int_\Omega (\partial_x u\, \partial_x v + \partial_y u\, \partial_y v) \quad \forall v_h = \\
\sum_K \int_K (\partial_x u\, \partial_x v + \partial_y u\, \partial_y v) \quad \forall v_h = \\
\sum_K \int_K (\partial_x u\, \partial_x \phi_i + \partial_y u\, \partial_y \phi_i) \quad \forall i = \\
\sum_K \underbrace{\sum_j u^j \int_K (\partial_x \phi_j\, \partial_x \phi_i + \partial_y \phi_j\, \partial_y \phi_i) \quad \forall i}_{a_{ij}\, u^j}
\end{aligned}$$

Easy enough the last step in our derivation can be expressed in matrix form as:

$$\sum_K a_{ij} u^j.$$

$$a_{ij}u^j = \begin{pmatrix} \int_\Omega (\partial_x\phi_1\,\partial_x\phi_1 + \partial_y\phi_1\,\partial_y\phi_1) & \int_\Omega (\partial_x\phi_2\,\partial_x\phi_1 + \partial_y\phi_2\,\partial_y\phi_1) & \int_\Omega (\partial_x\phi_3\,\partial_x\phi_1 + \partial_y\phi_3\,\partial_y\phi_1) \\ \int_\Omega (\partial_x\phi_1\,\partial_x\phi_2 + \partial_y\phi_1\,\partial_y\phi_2) & \int_\Omega (\partial_x\phi_2\,\partial_x\phi_2 + \partial_y\phi_2\,\partial_y\phi_2) & \int_\Omega (\partial_x\phi_3\,\partial_x\phi_2 + \partial_y\phi_3\,\partial_y\phi_2) \\ \int_\Omega (\partial_x\phi_1\,\partial_x\phi_3 + \partial_y\phi_1\,\partial_y\phi_3) & \int_\Omega (\partial_x\phi_2\,\partial_x\phi_3 + \partial_y\phi_2\,\partial_y\phi_3) & \int_\Omega (\partial_x\phi_3\,\partial_x\phi_3 + \partial_y\phi_3\,\partial_y\phi_3) \end{pmatrix}$$

a_{ij} is called the local system matrix. It can be evaluated using the trapezoidal roule:

$$\begin{aligned}
a_{ij} &= \frac{\text{area}(K)}{3}\sum_{q=1}^{3}\left(\partial_x\phi_j\,\partial_x\phi_i + \partial_y\phi_j\,\partial_y\phi_i\right) = \\
&= \text{area}(K)\left(\partial_x\phi_j\,\partial_x\phi_i + \partial_y\phi_j\,\partial_y\phi_i\right) = \\
&= \text{area}(K)\left(\begin{pmatrix}\partial_x\phi_1\\ \partial_x\phi_2\\ \partial_x\phi_3\end{pmatrix}\begin{pmatrix}\partial_x\phi_1 & \partial_x\phi_2 & \partial_x\phi_3\end{pmatrix} + \begin{pmatrix}\partial_y\phi_1\\ \partial_y\phi_2\\ \partial_y\phi_3\end{pmatrix}\begin{pmatrix}\partial_y\phi_1 & \partial_y\phi_2 & \partial_y\phi_3\end{pmatrix}\right)
\end{aligned}$$

notice that derivatives of liner functions are constant. The same procedure applies to the right and side:

$$\tilde{f}_i = \frac{\text{area}(K)}{3}\sum_{q=1}^{3}\left(f(x_q,y_q)\,\phi_i(x_q,y_q)\right)$$

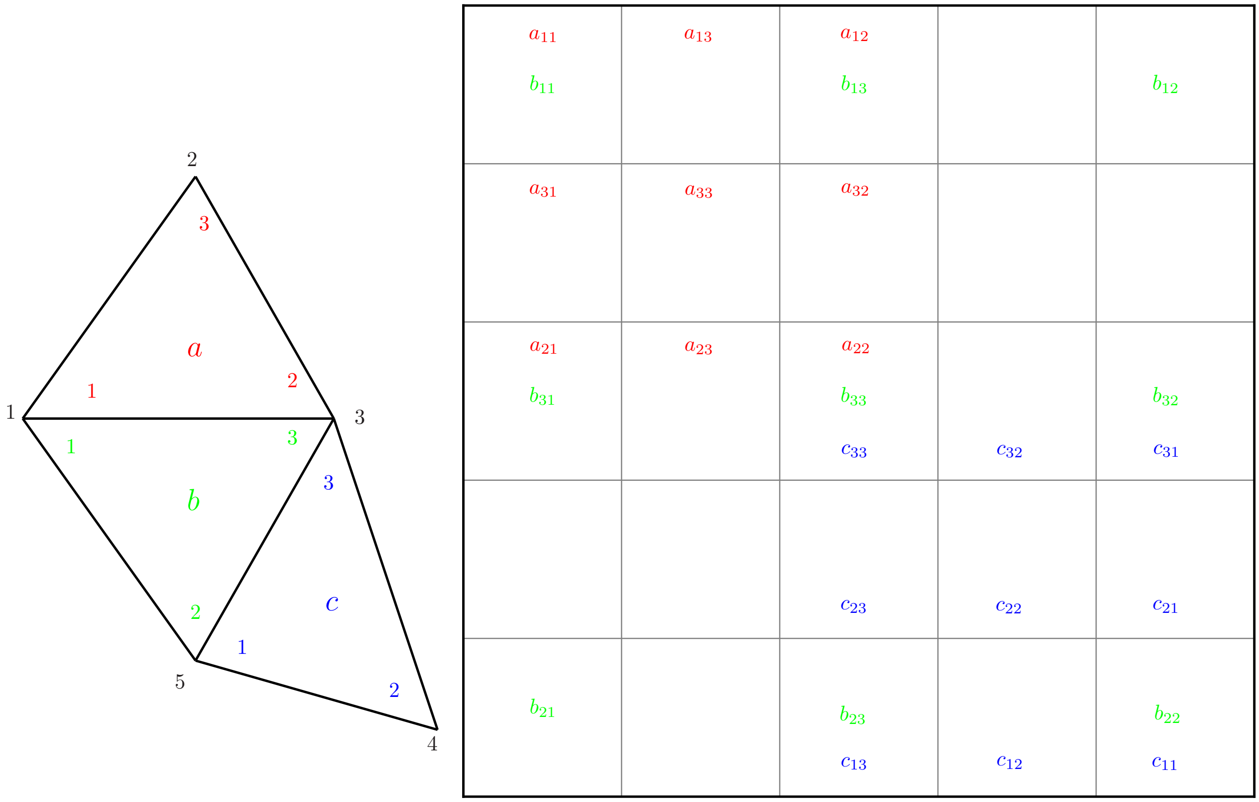
$$\begin{aligned}
\tilde{f}_i =& \frac{\text{area}(K)}{3}\begin{pmatrix} f(x_1,y_1)\,\phi_1(x_1,y_1) + f(x_2,y_2)\,\phi_1(x_2,y_2) + f(x_3,y_3)\,\phi_1(x_3,y_3)\\ f(x_1,y_1)\,\phi_2(x_1,y_1) + f(x_2,y_2)\,\phi_2(x_2,y_2) + f(x_3,y_3)\,\phi_2(x_3,y_3)\\ f(x_1,y_1)\,\phi_3(x_1,y_1) + f(x_2,y_2)\,\phi_3(x_2,y_2) + f(x_3,y_3)\,\phi_3(x_3,y_3)\end{pmatrix}\\
=& \frac{\text{area}(K)}{3}\begin{pmatrix} f(x_1,y_1)\,1 + f(x_2,y_2)\,0 + f(x_3,y_3)\,0\\ f(x_1,y_1)\,0 + f(x_2,y_2)\,1 + f(x_3,y_3)\,0\\ f(x_1,y_1)\,0 + f(x_2,y_2)\,0 + f(x_3,y_3)\,1\end{pmatrix}\\
=& \frac{\text{area}(K)}{3}\begin{pmatrix} f(x_1,y_1)\\ f(x_2,y_2)\\ f(x_3,y_3)\end{pmatrix}
\end{aligned}$$

Figure 5 should be explicative on how to account for local contributions in the global matrix. The final problem reduces to a linear system solution:

$$A\,u^j = f_i.$$

6. Homogeneous Boundary Conditions

$$\begin{pmatrix} A_{11} & A_{12} & A_{13} & A_{14} & A_{15}\\ A_{21} & A_{22} & A_{23} & A_{24} & A_{25}\\ A_{31} & A_{32} & A_{33} & A_{34} & A_{35}\\ A_{41} & A_{42} & A_{43} & A_{44} & A_{45}\\ A_{51} & A_{52} & A_{53} & A_{54} & A_{55}\end{pmatrix}\begin{pmatrix} u^1\\ u^2\\ u^3\\ u^4\\ u^5\end{pmatrix} = \begin{pmatrix} f_1\\ f_2\\ f_3\\ f_4\\ f_5\end{pmatrix}$$



a_{11} b_{11}	a_{13}	a_{12} b_{13}		b_{12}
a_{31}	a_{33}	a_{32}		
a_{21} b_{31}	a_{23}	a_{22} b_{33} c_{33}	c_{32}	b_{32} c_{31}
		c_{23}	c_{22}	c_{21}
b_{21}		b_{23} c_{13}	c_{12}	b_{22} c_{11}

Figure 5. The assembly explained in one picture. On the left an example mesh. On the right, the grid represents the global matrix. Highlighted inside the grid the local contributions a_{ij}, b_{ij}, c_{ij}.

Say that nodes 2 and 4 belong to the boundary:

$$\begin{pmatrix} A_{11} & A_{12} & A_{13} & A_{14} & A_{15} \\ 0 & A_{22} & 0 & 0 & 0 \\ A_{31} & A_{32} & A_{33} & A_{34} & A_{35} \\ 0 & 0 & 0 & A_{44} & 0 \\ A_{51} & A_{52} & A_{53} & A_{54} & A_{55} \end{pmatrix} \begin{pmatrix} u^1 \\ u^2 \\ u^3 \\ u^4 \\ u^5 \end{pmatrix} = \begin{pmatrix} f_1 \\ 0 \\ f_3 \\ 0 \\ f_5 \end{pmatrix}$$

Appendix A. Basis Functions

$$\phi(x, y) = a\,x + b\,y + c$$

$$\begin{aligned}
\phi_1(x_1, y_1) &= a\,x_1 + b\,y_1 + c = 1\\
\phi_1(x_2, y_2) &= a\,x_2 + b\,y_2 + c = 0\\
\phi_1(x_3, y_3) &= a\,x_3 + b\,y_3 + c = 0
\end{aligned}$$

$$\begin{aligned}
a\,x_1 + b\,y_1 + c &= 1\\
a\,x_2 + b\,y_2 + c &= 0\\
a\,x_3 + b\,y_3 + c &= 0
\end{aligned}$$

$$\begin{pmatrix} x_1 & y_1 & 1\\ x_2 & y_2 & 1\\ x_3 & y_3 & 1 \end{pmatrix}\begin{pmatrix} a\\ b\\ c \end{pmatrix} = \begin{pmatrix} 1\\ 0\\ 0 \end{pmatrix}$$

$$\begin{aligned}
\phi_2(x_1, y_1) &= a\,x_1 + b\,y_1 + c = 0\\
\phi_2(x_2, y_2) &= a\,x_2 + b\,y_2 + c = 1\\
\phi_2(x_3, y_3) &= a\,x_3 + b\,y_3 + c = 0
\end{aligned}$$

$$\begin{aligned}
\phi_3(x_1, y_1) &= a\,x_1 + b\,y_1 + c = 0\\
\phi_3(x_2, y_2) &= a\,x_2 + b\,y_2 + c = 0\\
\phi_3(x_3, y_3) &= a\,x_3 + b\,y_3 + c = 1
\end{aligned}$$

Appendix B. Assemble Coding Hints

There are plenty of ways to write the same thing. Coding is all about style. *It Works!* is a good thing, but sometimes this may collide the sense of good style. The assembly code is a great way to exemplify bad and good styles. Of course the style depends on the language. Of course the style depends on the goal of our piece of code.

Let's start with a `c-like` python code for the assembly, in Listing 1. This piece of code is good to get the mechanics of what we are asking to the computer. It probably works, the author did not try it. On the other hand the author finds it aesthetically terrible. Python is an interpreted language and does not really get on well with for loops. Moreover to the taste of the author they do really look too much old fashioned. The reader can appreciate how the python syntax helps programmers to develop elegant and readable code.

```
 1 def gradu_gradv(topo,x,y):
 2   A = np.zeros((x.shape[0],x.shape[0]))
 3   for element in topo:
 4       x_l = x[element]
 5       y_l = y[element]
 6       (dx_phi,dy_phi,phi,surf_e) = tri_p1(x_l,y_l,np.zeros((1,2)))
 7       local_matrix = np.zeros((3,3))
 8       for i in range(0,3):
 9           for j in range(0,3):
10               local_matrix[i,j] += surf_e *\
11               (dx_phi[i]*dx_phi[j] + dy_phi[i]*dy_phi[j] )
12 
13       for i in range(0,3):
14           for j in range(0,3):
15               A[element[i],element[j]] += local_matrix[i,j]
16 
17   return A
```

Listing 1: Example of a listing.

Lines 7 to 11 are really horrible. At least horrible in a *pythonc* sense. Our goal is to get:

$$a_{ij} = \text{area}(K)\left(\underbrace{\begin{pmatrix}\partial_x\phi_1\\ \partial_x\phi_2\\ \partial_x\phi_3\end{pmatrix}\begin{pmatrix}\partial_x\phi_1 & \partial_x\phi_2 & \partial_x\phi_3\end{pmatrix}}_{\texttt{local\_x}} + \underbrace{\begin{pmatrix}\partial_y\phi_1\\ \partial_y\phi_2\\ \partial_y\phi_3\end{pmatrix}\begin{pmatrix}\partial_y\phi_1 & \partial_y\phi_2 & \partial_y\phi_3\end{pmatrix}}_{\texttt{local\_y}}\right)$$

If we focus on the operands we recognise that we can get the local x contribution and the local y contribution as vector multiplication. En elegant code would reflect this mathematical property, and of course the closer to the mathematics is the code, the easier is to find errors. The most elegant solution we can think of, is using the Einstein notation. Consider the one dimensional array $\partial_x\phi_i$ with $i = 1\ldots3$ then:

$$\texttt{local\_x}_{i,j} = \partial_x\phi_i\,\partial_x\phi_j$$

The corresponding Numpy command is:

```
local_x = np.einsum('i,j->ij', dx_phi, dx_phi)
```

The sophisticated part is to understand how the indices are combined together. In this simple situation, `'i,j->ij'` the index `i` runs over the first vector, and the index `j` over the second. The entries are multiplied forming a two dimensional array. The situation is analogous for $\texttt{local\_y}_{i,j}$.

Also lines 13 to 15 are really bad. The situation would become easier if we had a one dimensional array of row indices (`rows`) and a one dimensional array of column indices (`cols`). Consider the `element=[5,6,7]`, the expected `cols=[5,6,7,5,6,7,5,6,7]`. This last one dimensional array can be considered the `flatten` version of `cols=[[5,6,7][5,6,7][5,6,7]]`. The two dimensional version of the array `cols` ca be considered as the product of a mesh operation of the `element` array with itself. In a more clear way:

$$\texttt{cols}, \texttt{rows} = \begin{pmatrix} \texttt{5} & \texttt{6} & \texttt{7} \\ \texttt{5} & \texttt{6} & \texttt{7} \\ \texttt{5} & \texttt{6} & \texttt{7} \end{pmatrix}, \begin{pmatrix} \texttt{5} & \texttt{5} & \texttt{5} \\ \texttt{6} & \texttt{6} & \texttt{6} \\ \texttt{7} & \texttt{7} & \texttt{7} \end{pmatrix} = \texttt{np.meshgrid(element,element)}$$