Vector Finite Elements and C++ *

Frédéric Hecht (INRIA),
Olivier Pironneau (Université Paris 6)

October 29, 2000

Abstract

The authors propose a few tools for programming efficiently in C++ the finite element method and discuss the best features of the language in a research environment where each program is written by a single programmer at a time. The tools make extensive use of operator overloading, templates and vectors. Applications are shown for Laplace equations, Lamé s Stokes and Navier-Stokes equations. An new programming strategy for the P1-iso-P2/P1 element is proposed whereby the matrix of the linear system is obtained by elementary manipulations of rows and columns of the P1/P1 matrix.

email: Olivier.Pironneau@inria.fr

1 Introduction

Research in numerical analysis involves both theoretical and programming work. The computer program, once written, is hardly exploited, except may be in the framework of a contract with industry, and once the method validated the program is often discarded; however bits and pieces are re-used by the author or passed on to others. Most often programs are not products of a team but of an individual. Therefore the buz words of C++, Òdata hiding, reusability, object orientedÓ are not necessarily relevant to such research. C++ is definitely catching in numerical analysis even though more than half of the programmers still use FORTRAN 77 and are perhaps not so eager to invest in another language if FORTRAN 90 does the job. But we will show here how C++ offers serious benefits for scientific computing, because of:

The class structure of the language is important of course but it is not such a revolution over the struct of C that it is worth the intellectual investment that C++ requires; the learning curve of C++ is long, too long indeed to become an expert! However operator overloading is really a powerful tool. As we will show, it is possible to write a code in scalar mode and change only one line to make it work in vectorial mode. On the

*Our thanks to D. Bernardi and M. Dicesare for very valuable help. Source codes are on http://www.ann.jussieu.fr/Pironneau

theorectical side the only thing to understand is that whenever a discrete approximation of a function u is decomposed on a basis like

$$u_h(x) = \sum_i u_i w^i(x)$$

it is necessary to keep the type of w^i real and scalar and change u_i according to the problem: complex or vector if the problem is with complex coefficients or is a system.

Thus going from a simple Laplace equation to the full elasticity system or complex valued Helmholtz equations requires no effort!

By redefining the bracket operator which access an array (ex c[2]) it is possible to switch on/off a range checking mode without losing efficiency and that also greatly facilitates debugging. Similarly dangling pointers can be avoided somewhat by overwriting the new and delete operators. A final remark concerning the popularity of C++ : there is a serious advange in using a widespread computer language in that its compilers are near to perfect and cheap; and in our case even a laptop computer is perfect for developping such C++ programs.

2 Safegard tools

2.1 Array range check

Borrowing an idea of Shapiro [4], with the following **template** Ê which essentially overload the operator [ ] every time $a[i]$ is executed in the program, it checks that i is within the bounds which have been allocated to a by a **new** function.

```
#include <assert.h>
template <class T> class A{  public: T *cc; long size;
       A(long csize = 0)
          { size = csize;
            if (size > 0 ) cc = new T[size]; else cc = 0;
          }
T&     operator [] (long i) const
          {assert ( cc&&(i >= 0) && (i < size) ); return cc[i]; }
int    no( T* t){ return t - cc;}  // return the place in array
      ~A() { delete [] cc;size = 0; }
void   init(long ssize);  // allocates already declared arrays
};
```

Dynamic arrays will be created with the class A<type>. So in all our programs we try to avoid using arrays directly and prefer this template structure, except when the array has fixed size (like 2 or 3). Note that there is very little speed lost because the function assert of the library stdlib.h can be disabled by the line: #define NDEBUG.

2.2 Example: the triangulation class

Two dimensional triangulations are implemented with the winged edge structures where each vertex knows its neighbor vertices and his neighbor triangles; each triangle knows its 3 edges and each edge knows its left and right triangles.

```
class Vertex { public:
        float x, y;        // cordinates
        int where;         // on which boundary
        int nsupp, nmate;  // nb of neighbors
        A<Triangle*> supp; // all triangles having the vertex
        A<Vertex*> mate;   // all neighbor vertices
 };
class Triangle { public:
        Vertex* v[3];      // 3 vertices of the triangle
        Edge* e[3];        // 3 edges opposite each vertex
        int where;         // in which region
        float area;
};
class Edge { public:
        Vertex *in, *out;  // oriented by first to last
        Triangle *left, *right;// triangles on each side
        int where;         // on which curve (or boundary)
        float length;
};
class Grid { public:
        int nt, nv, ne, nbholes, bdth; // bdth is bandwidth
        A<Vertex> v;        // all vertices
        A<Triangle> t;      // all triangles
        A<Edge> e;          // all edges
int     no(Triangle* tt) { return t.no(tt);}
int     no(Vertex* tt) { return v.no(tt);}
int     no(Edge* tt) { return e.no(tt);}    // Place in ar-
ray of Edge tt
        Grid():v(),t(),e(){nt=nv=ne=0;nbholes=50;}// default
        Grid(const char *path ):v(),t(),e();// reads a triangulation
void    square(int nx, int ny); // triangulate a square
void    save(const char* path); // save the mesh in Gfem format
};
```

Notice a peculiarity which is not common in that each triangle is not defined by the internal numbers of its 3 vertices but by 3 pointers to them. This way, renumbering can be performed without disrupting the geometrical data. The cost of this procedure is that each vertex must have an access function to recover its number or place in the array of vertices. This function is no().

2.3 Dangling Pointers

The basic idea is to overload operators new and delete . Then whenever a pointer is deleted (see MyDeleteOperator) we check first that it has been allocated properly; this requires a special function for allocation, here MyNewOperator . Also at the end of the program we check that all pointers have been destroyed. To avoid name conflicts, everything is encapsulated in the class AllocExtern. Thanks to the variable AllocExternData everything is done without modification of the source code on which it is used. The program given below is fast so long as the number of pointers is not too large, else an efficient searching function should be added; if the pointers are sorted then they are fast to find in the list of pointers.

```
inline void *operator new(size_t, void *place) { return place; }
void *operator new(size_t);
void operator delete(void * pp );

class AllocExtern {
class OneAlloc {              // for each pointer we store:
    void * p;                  // the pointer
    size_t l;                  // size of allocation
    long n;                    // a counter
}; // to avoid allocation of each pointer separatly we use packets
class AllocData {              // a packet of pointers
    public:   OneAlloc *a;     // the array of pointers
    AllocData * next;          // the next packet
}; //  Global variables
static size_t AllocSize ;      // total memory allocated
static size_t MaxUsedSize;     // maximum memory used
static AllocData * AllocHead ; // list head of pointer packets
static long NbAlloc;           // nb of pointers allocated
static void * NextFree;        // first empty pointer on OneAlloc

AllocData * NewAllocData();    // allocation of a packet of pointers
OneAlloc *Alloc();             // allocation of one pointer

public:
void *    MyNewOperator(size_t ll);
void      MyDeleteOperator(void * pp);
          AllocExtern();     //  constructor called before  main()
         ~AllocExtern(); }; //  destructor called after main()
static    AllocExtern AllocExternData; // init of global variables
```

3 Vector variational formulation

3.1 The scalar case

To compute $u(x,y)$, when $f(x,y), g(x,y)$ are given functions of 2 variables of Ω bounded in R^2, with

$$au - \Delta u = f \text{ in } \Omega u|_{\partial\Omega} = g,$$

we use the variational form:

$$\int_\Omega (au.v + \nabla u.\nabla v)dxdy = \int_\Omega fvdxdy, \ \forall v \in H^1(\Omega) \text{ with } v|_{\partial\Omega} = 0,$$

The Finite Element Method being a particular Galerkin method, we seek to approximate u by u_h with

$$u_h(x,y) = \sum_0^{M-1} u_i v^i(x,y),$$

where $\{v^i\}_0^{M-1}$ is a base of H^h, a finite dimensional approximation of $H^1(\Omega)$; the discrete problem is

$$\int_\Omega (au_h v^i + \nabla u_h.\nabla v^i)dxdy + p\sum_{j\in\partial\Omega}(u_j - g_j)\delta_{ij} = \int_\Omega fv^i dxdy, i = 0, ..., M-1$$

Here p is a very large penalisation parameter and the notation $i \in \partial\Omega$ indicates that the basis function v^i is attached to the boundary of Ω. To build $\{v^i\}$ the simplest is to triangulate Ω and bind to each vertex q^i a continuous piecewise linear function v^i which is 1 at q^i and 0 at $q^j, j \neq i$. Then the discrete problem becomes a linear system for the values u_i of u_h at point q^i:

$$AU = F \quad \text{with} \quad A_{ij} = \int_\Omega (av^j v^i + \nabla v^j.\nabla v^i)dxdy + p\delta_{\{ij, i\in\partial\Omega\}}$$

and

$$F_i = \int_\Omega fv^i dxdy + pg_i\delta_{\{ij, i\in\partial\Omega\}}$$

3.2 The case with complex coefficient: Helmholtz' equation

Le u be the amplitude of a wave of frequency ω. It verifies

$$\omega^2 u + \nabla\cdot(\epsilon\nabla u) = 0, \text{ in } \Omega \quad iu + \frac{\partial u}{\partial n} = 0 \text{ on } \partial\Omega_\infty$$

Non reflecting boundary conditions involve the complex number i. The numerical solution will be done with complex coefficients. Nothing changes in the algebra and all the formula above are valid. In the decomposition of u_h, *the coefficients are complex numbers but the hat functions v^j remain real valued*. The only difference is that the algebra is C instead of R. Thus redefining the basic operations $*/+-$ will do.

3.3 The vector Laplace equation

Consider

$$au - \Delta u = f \text{ in } \Omega, \quad u|_{\partial\Omega} = g.$$

where $u(x), f(x), g(x)$ are N-vecteurs with complex values, and where a is a NxN complex valued matrix. The variational formulation is the same but A_{ij} is now a NxN-matrix and U_i and F_i are N-vectors. The algebra for the linear system is that of the NxN-matrices.

3.4 Application to elasticity

Let $u(x)$ denote the (vector) displacement of x in a beam Ω subjected to volumic forces $g(x)$. It is solution of the Lamé equations:

$$\int_\Omega [\lambda(\nabla u + \nabla u^T) : (\nabla v + \nabla v^T) + \mu \nabla \cdot u \nabla \cdot v] = \int_\Omega g \cdot v, \quad \forall v \text{ such that } v|_C = 0.$$

Recall the notation A:B for the trace of AB. As before the hat functions are still scalar

$$\vec{u}_h(x) = \sum_i \vec{u}_i v^i(x)$$

The elements , A_{ij} of the matrix of the linear system are themselves 2x2 matrices:

$$Aijkl = \int_\Omega [\lambda(\nabla(v^i e^k) + \nabla(v^i e^k)^T) : (\nabla(v^j e^l) + \nabla(v^j e^l)^T) + \mu \nabla \cdot (v^i e^k) \nabla \cdot (v^j e^l)]$$

4 Algebra of basic types

Normally a multiplication, for example, a * b , is with a and b real or complex. But in numerical analysis many algorithms are still valid for other algebras. In C++, programming can be extended to any algebra by giving to the compiler a definition for the basic operations (+,-,*,/). So we will define the multiplication, for example, a*b, for a of type T and b of type R and

```
T,R float or Complex,  or T NxN matrix, R N-vector.
```

Complex numbers are now part of the language definition (Koenig[2]; they are described also in most books on C++ (Buzzi-Ferrari[1], Stroustrup[5]). The principle of such a construction is quite classical in C++.

4.1 Fixed size vectors and matrices

Next, when the base type is a vector of fixed length N, operations are provided below. Each element of the vector is of type T destined to be either float or Complex. N-vectors and NxN-matrices of fixed size is also quite classical in C++.

```
template <class T, int N> class VectN{ public:   T val[N];
VectN(const VectN& r) ;
VectN() ;
VectN(T r) ;
T&   operator[] ( int i)
     { assert((i< N)&&(i>=0));  return val[i];}
VectN  operator*    (const T& a);
...
VectN&   gauss  (const MatN<T,N>& a);
};

template <class T,  int N> class MatN{ public:   T val[N][N];
MatN(const MatN& r) ;
MatN();
T& operator() ( int i, int j)
{     assert((i< N)&&(i>=0)&&(j< N)&&(j>=0));  return val[i][j];}
MatN&     operator+=    (const MatN& a);
...
VectN<T,N> operator*(const VectN<T,N>& x);
};
```

5 P1 functions and band matrices

The linear system generated by the PDE is stored in a band matrix (it is trivial to change the band structure by a skyline structure). The band matrix is stored in an array cc with an access function, so that $cc(i,j)$ actually calls $cc[size*(i-j+bdth)+j]$, where bdth is the bandwidth and size is the dimension of the square matrix. For the problem described above, $cc(i,j)$ is a real number but for other applications like Helmholtz equation, it could be a complex number or even an NxN matrix, so let us say that in all generality the matrix elements are of type T. Furthermore the matrix is destined to be multiplied by a vector whose type R could also vary and be a float or a complex or an N-vector. In this last example Thus the band matrix is defined via a template where T and R are undefined. In one instance T will be itself a matrix mxm and R will be a vector of size m.

5.1 Band matrices

```
template <class T, class R> class Bandmatrix { public:
      int bdth,size,csize;  // bandwidth, matrix and array size
      T* cc;
Bandmatrix(int n,int bdth1);
Bandmatrix(Bandmatrix& b);
```

```
~Bandmatrix(){ delete [] cc; }
T& operator()( int i,  int j) const  // access function
{ int k = size*(i-j + bdth) + j;
  assert(cc && (k>=0)&&(k<csize)&& (i>=0)&&(i<size));
  return cc[k];
 }
Vector<R> operator * (const Vector<R>& );  // matrix vector mul
};
```

5.2 P1 functions as vectors

A piecewise linear continuous function is known from its values at the vertices. Thus it can be stored in a one dimensional array very similar to vectors since additions, multiplications... have the same syntax. Vectors are classically defined in C++ by a template because vectors are constructed on any agebra - the base type - reals or complex numbers or other. Each element of the vector being of type T, the template is

```
template <class T> class Vector{ public:
          T *cc; int size;
Vector(int csize = 0)
{  size = csize;
   if (size > 0 )
    {  cc = new T[size]; for(int i=0;i<size;i++) cc[i] = 0; }
   else cc = NULL;
}
~Vector()  { delete [] cc; size = 0; }
T&  operator [](int i) const
      { assert ( cc &&(i >= 0)&&(i < size) ); return cc[i];}
Vector ( const Vector& v );  // copy constructor
const Vector& operator = (const T& r);
...
Vector operator/ (const float& a);
};
```

5.3 Exemple: Gauss factorization

The matrix A is stored in an array a. Its elements are of type T. The vector x in a product Ax has components of type R. For instance T and R could be both float or Complex but they could also be 2x2 matrix and 2-vector as in block algebra. By using templates the gaussband function will work in all these cases. The input boolean ÒfirstÓ is used to switch from factorization + solution to solution only. Notice also how the access function which overloads $a(,)$ simplifies references to $a(i,j)$. The function below is programmed as if T was the scalar real basic type 'float' and full instead of band

matrices (except in the arguments of the do loops). Yet without modification it becomes a block Gauss factorization as well, by changing T.

```
template <class T,class R>
      float gaussband (Bandmatrix<T,R>& a, Vector<R>& x, int first)
{  int i,j,k, n = a.size, bdthl = a.bdth;
   T s, s1;
   R s2;
   float smin = 1e9, eps = 1/smin;
   if (first)    // factorization
     for (i=0;i<n;i++)
      {  for(j=mmax(i-bdthl,0);j<=i;j++)
         {  s=0; for (k=mmax(i-bdthl,0); k<j;k++)
                     s += a(i,k)*a(k,j);
            a(i,j) -= s ;
         }
         for(j=i+1;j<=mmin(n-1,i+bdthl);j++)
         {  s= a(i,j);
            for (k=mmax(j-bdthl,0);k<i;k++) s -= a(i,k)*a(k,j);
            s1 = a(i,i);
            if(s1.modul() < smin) smin=s1.modul();
            if(s1.modul() < eps) s1 = eps;
            a(i,j) = s / s1;
         }
      }
    for (i=0;i<n;i++)                    // resolution
      {    s2 = x[i];
         for (k=mmax(i-bdthl,0);k<i;k++) s2 -= a(i,k) * x[k];
         x[i] = s2 / a(i,i) ;
      }
    for (i=n-1;i>=0;i--)
      { s2=0; for (k=i+1; k<=mmin(n-1,i+bdthl);k++)
                  s2 += a(i,k)* x[k];
       x[i] -= s2 ;
      }
   return smin;
}
```

6 Finite Element functions

6.1 Defining the base type

For the real scalar case one will write:

```
typedef VectN<float,1> vectNB; // base type is float(=vectN of size 1)
typedef MatN<float,1> matNB;   // same for matrices
```

6.2 Structure of the main() function

The main() calls 3 functions to: Build the matrix, compute the right and side and solve the linear system

```
void main()
{  Grid g;
   g.rectInit(15,15);// triangulates the unit square with a 15x15 grid
   g.prepgrid();     // computes areas of triangles, neigh-
bor points...
   Vector<vectNB> f(g.nv), sol(g.nv); //for -Delta(sol)=f
   for(int i=0;i< g.nv;i++) f[i] = 1 ; // here f=1 and zero Dirichlet.
   Bandmatrix<matNB,vectNB> aa(nv,g.bdth);
   buildmatlaplace(g,aa);
   for(int i=0;i< g.nv;i++)if(g.v[i].where) aa(i,i)= 1e10; // p=1e10
   sol = rhs(g,f);      // computes right hand side of lin-
ear system
   cout<< "pivot=" << gaussband(aa,sol,1) << endl;
}
```

6.3 Function buidmatlaplace()

Besides the fact that it is written with templates so as to handle the vector case without modification, this function proceeds in the usual way by a loop on the triangles and a computation of elementary contributions to the matrix on each triangle. The case $a \neq 0$ is not considered here, for clarity.

```
template <class T, class R>
   void buildmatlaplace (Grid& g, Bandmatrix<T,R>& aa, T al-
pha )
{     const int next[4] = {1,2,0,1};
      int i,j,k,ip,jp,ipp,jpp,iloc,jloc;
      T alph,dwidxa,dwjdxa,dwidya,dwjdya,aaloc;
      for ( k=0; k<g.nt; k++)
      {  Triangle& tk = g.t[k];
         alph = alpha * (tk.area / 12) ;
         for ( iloc=0; iloc<3; iloc++)
            { i = g.no(tk.v[iloc]);
```

```
              ip = g.no(tk.v[next[iloc]]);
              ipp = g.no(tk.v[next[iloc+1]]);
              dwidya = -(g.v[ip].x - g.v[ipp].x)/(tk.area * 4);
              dwidxa = (g.v[ip].y - g.v[ipp].y)/(tk.area * 4);
              for ( jloc=0; jloc<=iloc; jloc++)
              { j = g.no(tk.v[jloc]);
                jp = g.no(tk.v[ next[jloc]]);
                jpp = g.no(tk.v[next[jloc+1]]);
                dwjdxa = g.v[jp].y - g.v[jpp].y;
                dwjdya = -(g.v[jp].x - g.v[jpp].x);
                aaloc =  dwidxa * dwjdxa + dwidya * dwjdya + alph
                aa(i,j) += aaloc;
                if(i != j) aa(j,i) += aaloc; else aa(i,i) += alph;
               }
            }
      }
}
```

6.4 The right hand side rhs()

The computation of the volume integral of the right hand side also offers no additional difficulty. It is a loop on the triangles with the use of a Gauss quadrature formula based on the mid edges.

```
template <class T> Vector<T> rhs(Grid& g, const Vector<T>& f)
{     const int next[4] = {1,2,0,1};
      Vector<T> r(g.nv);
      for (int k=0; k<g.nt; k++)
      {     Triangle& tk = g.t[k];
            int i = g.no(tk.v[0]);
            T f0 = f[i] + f[g.no(tk.v[1])] + f[g.no(tk.v[2])];
            for (int iloc=0; iloc<3; iloc++)
                  r[i] += ( f[g.no(tk.v[iloc])] + f0 ) *  (tk.area/12) ;
      }
      return r;
}
```

7 Stokes' Problem

Consider the problem of finding a velocity vector field $\mathbf{u}$ and a pressure scalar field p solution of

$$\alpha u - \nabla \cdot (\nu \nabla u) + \nabla p = 0, \quad \nabla \cdot u = 0 \text{ in } \Omega, \quad u|_{\partial\Omega} = g$$

The variational formulation is

$$a(u,v) + b(p,v) + c(u,q) = (f,v) \;\; \forall v \in H_0^1(\Omega)^2, \;\; \forall q \in L^2(\Omega)$$

with

$$a(u,v) = \int_\Omega [\alpha u + \nu \nabla u : \nabla v], \quad b(p,v) = \int_\Omega v \cdot \nabla p, \quad c(u,q) = \int_\Omega \nabla \cdot u \, q$$

Let us call $U = (u_1, u_2, p)^T$ and $V = (v_1, v_2, q)$, then, with obvious notations

$$A(U,V) = (F,V), \;\; \forall V \in W = H_0^1(\Omega)^2 \times L^2(\Omega), \;\; U - (g,0)^T \in W.$$

Two well known discretizations are used:

The $P1 - P1$ stabilized element where by all variables u and p are continuous and piecewise linear on a triangulation but the triangulation for u is obtained by dividing each triangle of the triangulation of p into 3 triangles by joining the vertices to the center of gravity of each triangle.

The second method does the same but divides the triangles of p into 4 subtriangles for u by joining the mid-edges. It is refered as the P1-iso-P2/P1 element ([3]).

7.1 The P1-bubble / P1 element

The first case is much easier to implement because the unknows at the center of the triangles (bubbles) can be eliminated and the resulting linear system is of the form

$$\begin{pmatrix} A & B^T \\ B & D \end{pmatrix} \begin{pmatrix} U \\ P \end{pmatrix} = \begin{pmatrix} F \\ 0 \end{pmatrix}.$$

In tnfem it is programmed exactly as the Laplacian except that N=3 instead of 1 and the formula for aaloc(i,j) is different.

```
template <class T, class R>
  void buildmatstokes (Grid& g, float nu, float alpha, float beta,
                Bandmatrix<T,R>& aa )
{   const float alph = alpha/12.0, bet = beta*0.5/nu;
    int i,j,k,kp,ip,jp,ipp,jpp,iloc,jloc;
    float dwidxa,dwjdxa,dwidya,dwjdya;
    T aaloc;
      aa.zero();
    for ( k=0; k<g.nt; k++)
    {   Triangle& tk = g.t[k];
        float tka = tk.area * 2;
        for ( iloc=0; iloc<3; iloc++)
        {   i = g.no(tk.v[iloc]);
```

```
            ip = g.no(tk.v[next[iloc]]);
            ipp = g.no(tk.v[next[iloc+1]]);
                 dwidxa = (g.v[ip].y - g.v[ipp].y)/tka;
                 dwidya = -(g.v[ip].x - g.v[ipp].x)/tka;
            for ( jloc=0; jloc<3; jloc++)
            {    j = g.no(tk.v[jloc]);
                 jp = g.no(tk.v[ next[jloc]]);
                 jpp = g.no(tk.v[next[jloc+1]]);
                 dwjdxa = (g.v[jp].y - g.v[jpp].y)/tka;
                 dwjdya = -(g.v[jp].x - g.v[jpp].x)/tka;
  aaloc(0,0) =  nu*(dwidxa * dwjdxa + dwidya * dwjdya) + alph ;
  aaloc(1,1) =  nu*( dwidxa * dwjdxa + dwidya * dwjdya) + alph ;
  aaloc(2,2) =  bet*tk.area*( dwidxa * dwjdxa + dwidya * dwjdya)+1e-
10 ;
  aaloc(1,0) =  aaloc(0,1) =  0;
  aaloc(2,0) =  aaloc(0,2) =dwidxa/3; aaloc(2,1) =aaloc(1,2) =dwidya/3;
  if(i==j){ aaloc(0,0) += alph;  aaloc(1,1) += alph; aaloc(2,2) += 1e-
10;}
                 aa(j,i) += aaloc * tk.area;
            }
        }
    }
}
```

Notice that the Gauss factorization will inverse the diagonal 3x3 blocks, so these must be non-singular. But the pressure being defined up to a constant, the pressure operator must be made coercive and the easiest is to add a small diagonal matrix to the pressure blocs of the P1/P1 matrix.

7.2 The P1-iso-P2 / P1 element

At first sight, it seems difficult to use the vector programming technique for this element for which the number of unknowns at each node depends on the node position, whether at a vertex or at a mid-edge. However, by using an analogy with the hierarchical basis method of Yserantant[6] we can convert this problem into a vector problem. We shall work with only one mesh, the velocity mesh, but we remember that it is a subdivision of a coarser mesh, the pressure mesh. The vertices of the coarse mesh will be called the P1 node and the other nodes (vertices of the fine mesh but not of the coarse mesh) the P2 nodes; We shall need a boolean function p2node(i) to tell if node i is P2 or P1. Then let us denote the linear system of the P1-P1 element on the fine mesh without bubble stabilization by

$$\begin{pmatrix} A & B_1 & B_2 \\ C_1 & 0 & 0 \\ C_2 & 0 & 0 \end{pmatrix} \begin{pmatrix} U \\ P^1 \\ P^2 \end{pmatrix} = \begin{pmatrix} F \\ 0 \\ 0 \end{pmatrix}.$$

where the vector P^1 consists of all the values of the pressure at the coarse mesh (P1 nodes), and $P2$ those at the other (P2) nodes. In view of the P1-iso-P2/P1 element, three things are wrong with this system:

1. at a P2-node q^k between P1-nodes, q^i, q^j, we should have $p_k = (p_i + p_j)/2$.
2. The hat function w^i is wrong because it is 0 at node q^k instead of 0.5.
3. The hat function w^k should not be used.

Let the P1/P1 linear system be decomposed into blocks corresponding to (u, p^1, p^2) where p^1 is the vector of pressure values on all P1 nodes and similarly for p^2. Let the P1/P1 and P1-iso-P2/P1 matrices be

$$\begin{pmatrix} A & B^1 & B^2 \\ C^1 & 0 & 0 \\ C^2 & 0 & 0 \end{pmatrix} \quad \begin{pmatrix} A & \tilde{B}^1 \\ \tilde{C}^1 & 0 \end{pmatrix}$$

It is not hard to see that

$$\begin{pmatrix} A & \tilde{B}^1 & 0 \\ \tilde{C}^1 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} = \begin{pmatrix} I & 0 & 0 \\ 0 & I & D^T \\ 0 & 0 & 0 \end{pmatrix} \begin{pmatrix} A & B^1 & B^2 \\ C^1 & 0 & 0 \\ C^2 & 0 & 0 \end{pmatrix} \begin{pmatrix} I & 0 & 0 \\ 0 & I & 0 \\ 0 & D & 0 \end{pmatrix}$$

where I is the identity matrix and

$$D_{ij} = \frac{1}{2} \quad \text{if} \quad j \in V(i), \quad = 0 \ \text{ otherwise}$$

where $V(i)$ is the set of indices of neighbor nodes, by an edge in the fine mesh of q^i.

Indeed notice that the hat function $\tilde{w}^i$ of the coarse mesh at q^i is expressed in terms of the hat functions of the fine mesh by

$$\tilde{w}^i = w^i + \frac{1}{2} \sum_{k \in V(i)} w^k.$$

Therefore,

$$\int_\Omega \nabla p \cdot w^i = \sum_j p_j \int_\Omega \nabla w^j \cdot w^i = \sum_{j \in P1} p_j \int_\Omega \nabla (w^j + \frac{1}{2} \sum_{k \in V(j)} w^k) \cdot w^i,$$

in other words

$$\tilde{B^1}_{ij} = B^1_{ij} + \sum_{k \in V(j)} B^2_{ik}.$$

Similarly

$$\int_\Omega \nabla \cdot u w^i = \sum_{l=1,2} \sum_j u_{lj} \int_\Omega \frac{\partial \tilde{w^j}}{\partial x_l} w^i = \sum_{l=1,2} \sum_{j \in P1} u_{lj} \int_\Omega (\frac{\partial w^j}{\partial x_l} + \frac{1}{2} \sum_{k \in V(j)} \frac{\partial w^k}{\partial x_l}) \cdot w^i, \ i.e.$$

Loop in i,j

$$\tilde{C^1_{ij}} \leftarrow C^1_{ij} + C^2_{k,j} \text{ if } k \in V(i).$$

However the resulting matrix is singular because the P2-pressure degrees of freedom are useless (problem 3). One way is to replace the last block by the identity matrix, but the correct values of the pressure at these places must be recalculated once the values of the pressure at the P1 nodes are known.

7.3 Implementation

For the driven cavity problem, discretized by 3 families of parallel lines, the boolean function is simple. The function which sets the pressure at the P2 nodes to their correct value uses the edges:

```
void p2pressure(Grid& g, Vector<vectNB>& f)
{// assumes pressure at p2 vertices is zero
    for(int k =0; k< g.ne; k++)
    {    int i = g.no(g.e[k].in ), j = g.no(g.e[k].out );
         if(p2node(i)&& !p2node(j)) f[i)][2] += f[j][2]/2;
         if(p2node(j)&& !p2node(i)) f[j][2] += f[i)][2]/2;
    }
}
```

Finally the function which builds the P1-iso-P2/P1 matrix from the P1/P1 matrix loops also on the edges to find the nonzero entries of the P1/P1 matrix and then find the neighbors at their neighbors by the neighbor arrays g.v[i].mate[].

```
void fromp1top2(Grid& g, Bandmatrix<matNB,vectNB>& aa)
{ int i,i1,j,j1,k,m,l, ii[2];
  for( i1=0; i1<g.ne; i1++)
  {   ii[0] = g.no(g.e[i1].in); ii[1] = g.no(g.e[i1].out);
      for(l=0;l<2;l++)
      {   i = ii[l==1?0:1];
          if(k=ii[l], p2node(k))
            for( j1=0; j1<g.v[k].nmate; j1++)
                if(j = g.no(g.v[k].mate[j1]),!p2node(j))
                    for(m=0;m<2;m++)
                    {   aa(i,j)(m,2) += aa(i,k)(m,2)/2;
                        aa(j,i)(2,m) += aa(k,i)(2,m)/2;
                    }
          if(p2node(i)) aa(i,i)(2,2) = 1e10;
      }
  }
}
```

8 Numerical Tests

8.1 Micro-wave oven

First we present the numerical solution of Helmholtz equation in the complex plane on a configuration which is close to a 2D micro-wave oven and where the electromagnetic coefficients change in the vertical rectangle. The wave signal comes from the boundary condition on the left (figure 1).

8.2 Lamé Equations

Then the equations of elasticity in 2D are solved by this vector approach. The domain is a rectangle. The external force are the weight of the beam which is clamped on the left. The flection are exagerated, naturally (figure 2).

8.3 Navier-Stokes equations

The driven cavity problem was tested at Re=3000 with a mesh of 50x50 refined near the boundary. The results after 300 time iterations are shown for the P1/P1 and P1-iso-P2/P1 elements with a time step of 0.05 and a treatment of the nolinear terms by the Galerkin-Characteristic method of degree 2 in time:

$$u^{n+1}(x) - u^n(x - u^n(x)\delta t) - \frac{\nu\delta t}{2}\Delta(u^{n+1} + u^n) + \nabla p^{n+1}\delta t = 0, \quad \nabla\cdot(u^{n+1} + u^n) = 0.$$

The results are shown on figures 3 and 4.

In this case the boundary conditions are on u. The vector formulation allows complicated boundary conditions also. For instance, in the driven pipe problem of the boundary conditions are:

- $u = (x, 0)$ on the left, $u = (1, 0)$ on the top, $u = 0$ on the bottom
- $p = 0$ and $u \times n = 0$ on the right

Notice, on figure 5, the perfect exit of the flow in the upper part of the pipe and perfect entrance in the lower part.

References

[1] G. Buzzi-Ferraris: Scientific C++ Addison-Wesley 1993

[2] A. Koenig (ed.) : Draft Proposed International Standard for Information Systems - Programming Language C++. ATT report X3J16/95-087 (ark@research.att.com)

[3] O. Pironneau: Finite Elements for Fluids. Wiley 1989.

[4] J. Shapiro: A C++ Toolkit, Prentice Hall 1991.

[5] B. Stroustrup: The C++ Programming Language Addison Wesley 1991

[6] H. Yserantant: On the multi-level splitting of finite element spaces. Numer. Math. 49; p 379-412, 1986.

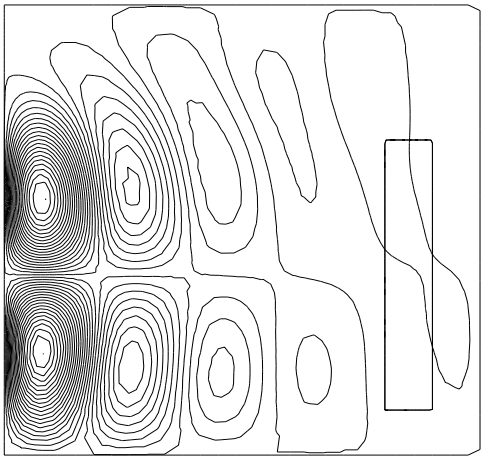

Figure 1: Electromagnetic wave in a micro-wave oven.

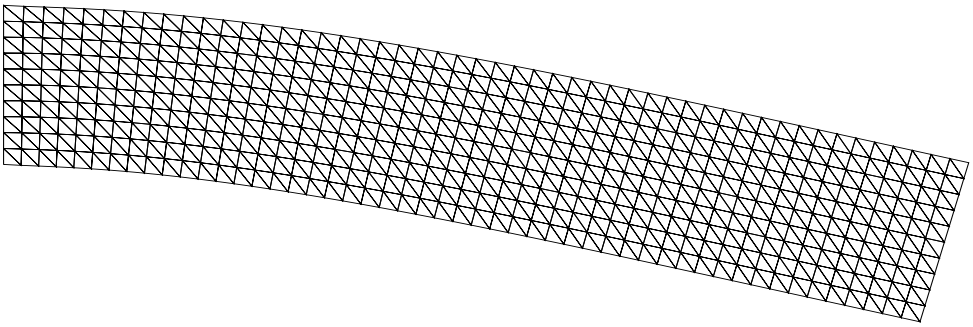

Figure 2: Flection of a clamped beam under its own weight.

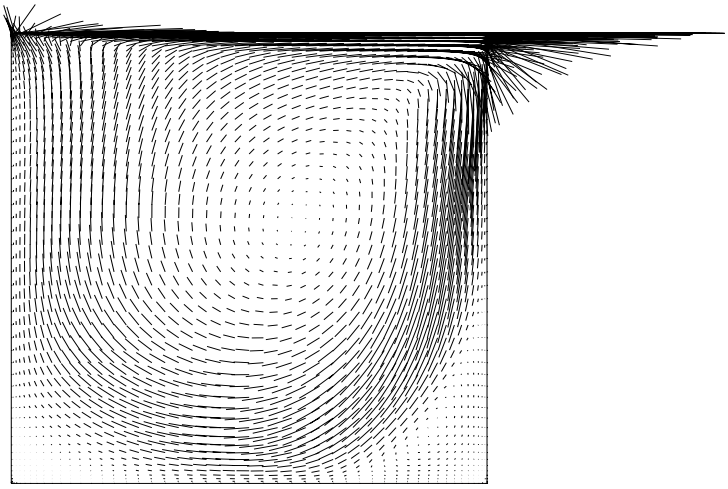

Figure 3: Driven cavity at Re=3000 with 50x50 points and the P1/P1 element.

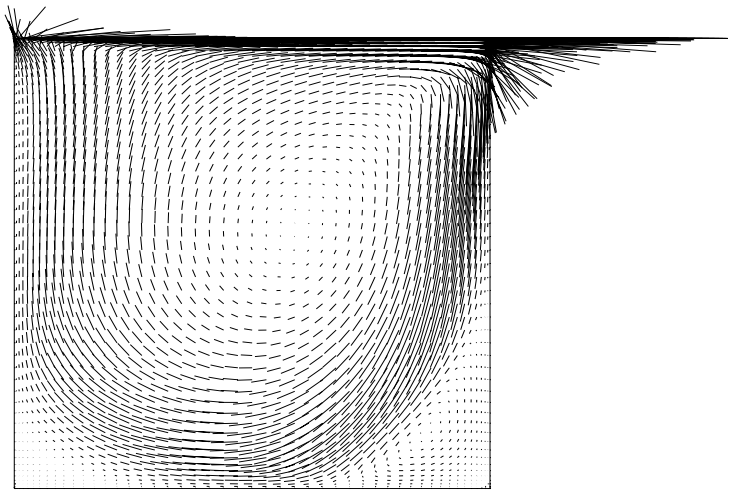

Figure 4: Driven cavity at Re=3000 with 50x50 points and the P1 - iso P2/P1 element.

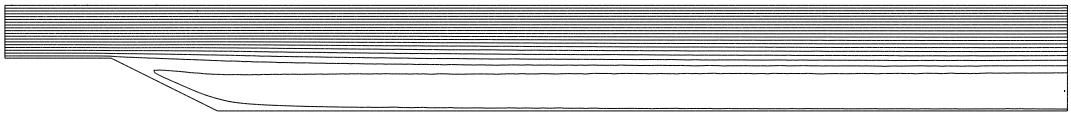

Figure 5: Driven pipe at Re=1000 with 3200 points and the P1 - element.