Notas sobre FreeFem++ 2D y 3D: traducción del manual de F. Hecht

Eliseo Chacón Vera
Departamento de Ecuaciones Diferenciales y Análisis Numérico,
Facultad de Matemáticas, Universidad de Sevilla

4 de julio de 2011

1. Introducción a FreeFem++ 2D

Los archivos de órdenes se guardan con la terminacion **.edp**, el nombre no es importante pero si lo es la terminación. Normalmente, en un entorno Linux, se ejecutan mediante la orden

FreeFem++ nombre.edp

y en un entorno Windows presionando dos veces con el botón derecho del ratón sobre el archivo. La gramática es similar a la del lenguaje C/C++:

- las variables se declaran siempre,
- las distintas órdenes terminan con ;
- **cout** indica la salida por defecto, pantalla o archivo
- $\backslash n$ significa un salto de linea, etc.

Algunos ejemplos de aritmética es **aritmetica.edp**.
Archivo **aritmetica.edp**:

```
real x=3.14,y;
int i,j;
complex c;
cout<< " x = "<< x << "\n";
x=1;y=2;x=y;i=0;j=1;
cout<< 1+3 << "  " << 1./3 << "\n";
cout<< 10^10. <<  "\n";
cout<< 10^-10. <<  "\n";
cout<< -10^-2.+5 <<  "==4.99 \n";
cout<< 10^-2.+5 <<  "==5.01 \n";
cout<< " 8^(1/3) = "<< (8)^(1./3.) <<"\n";
cout<< "-------------complex  ---------- \n";
cout<< 10-10i <<"\n";
cout<< " -1^(1/3) = "<< (-1+0i)^(1./3.) <<"\n";
```

Con **ciclofor.edp** y **ciclowhile.edp** se puede ver el uso de **ciclos**. $i++$ que equivale a $i = i + 1$.
Archivo **ciclofor.edp**:

```
int i;
for (i=0;i<10;i=i+1)
cout <<i <<"\n";
```

El siguiente ciclo no va a terminar.
Archivo **ciclowhile.edp**:

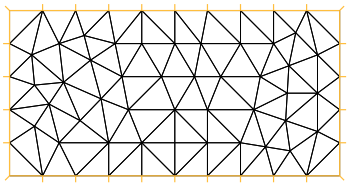
```
int i;
i=0;
real eps=1;
while (eps+1!=eps)
{
eps=eps/2;
  i++;
if(eps+1==eps) break;
cout <<i <<"  eps = "<<eps<<"  eps+1 = "<<eps+1<<"\n";
}
```

2. Generación de mallas

Para crear una malla en un rectángulo (o una superficie cualquiera del plano) primero se usa una descripción paramétrica del contorno de la superficie y de cada lado del mismo.

A cada porción descrita paramétricamente se le puede asignar una etiqueta, mediante la orden **label=** que sirva para asociar condiciones de contorno. Por defecto, si no se usa **label=**, la etiqueta será el nombre dado al trozo de frontera.
Archivo **rectangulo.edp**:

```
border a(t=0,2){x=t;y=0;label=1;};
border b(t=0,1){x=2;y=t;label=2;};
border c(t=0,2){x=2-t;y=1;};
border d(t=0,1){x=0;y=1-t;label=4;};
int n=5;
mesh th=buildmesh(a(2*n)+b(n)+c(2*n)+d(n));
plot(th,wait=1,ps="rectangulo.eps");
```



Malla no uniforme generada con **rectangulo.edp**

Con la orden **border** se define un segmento del contorno de la geometría y con las ordenes **mesh** y **buildmesh** se genera la malla. Se puede alterar la numeración de los lados mediante la orden **label**. Varios lados pueden tener el mismo número de referencia si van a tener el mismo dato de contorno.

Es importante observar que el contorno de la superficie se debe de **describir en el sentido contrario a las agujas del reloj**, es decir, de derecha a izquierda.

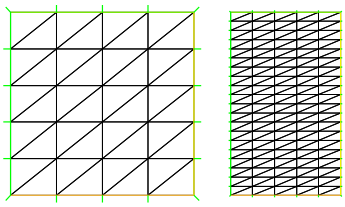
La orden **square** genera por defecto una malla uniforme del rectángulo unidad $[0,1]^2$ y la **numeración por defecto de los lados** es 1,2,3,4 para

base, derecha, superior, izquierda. También puede servir para generar un rectángulo cualquiera en su formato general.

En el caso de rectángulos, la diferencia entre la orden **square** y la orden **buildmesh** es que la primera genera una malla uniforme mientras que la segunda **buildmesh** genera una malla no uniforme.

Archivo **dosrectangulos.edp**

```
real x0=1.2,x1=1.8;
real y0=0.,y1=1.;
int n=5,m=20;
mesh Th=square(n,m,[x0+(x1-x0)*x,y0+(y1-y0)*y]);
mesh th=square(4,5);
plot(Th,th, ps="dosrectangulos.eps");
```



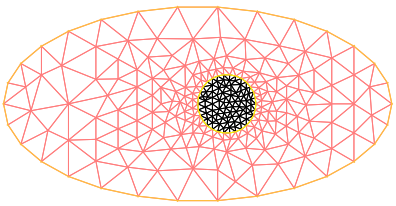
Dos mallas uniformes generadas con **dosrectangulos.edp**

Se pueden dibujar dos mallas al mismo tiempo con la orden **plot** y la opción **ps = ...** genera un fichero postscript.

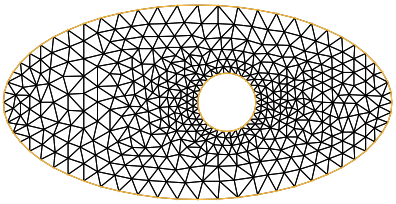
Para generar dominios con agujeros simplemente se cambia el sentido de orientación, o de la descripción paramétrica, de la frontera interna:

Archivo **hueco.edp**

```
real pi=4*atan(1.0);
border a(t=0,2*pi){x=2*cos(t);y=sin(t);label=1;}
border b(t=0,2*pi){x=.3+.3*cos(t);y=.3*sin(t);label=2;}
mesh thwithouthole=buildmesh(a(50)+b(30));
mesh thwithhole=buildmesh(a(50)+b(-30));
plot(thwithouthole,wait=1,ps="thsinhueco.eps");
plot(thwithhole,wait=1,ps="thconhueco.eps");
```



Dos mallas complementarias generadas con **hueco.edp**



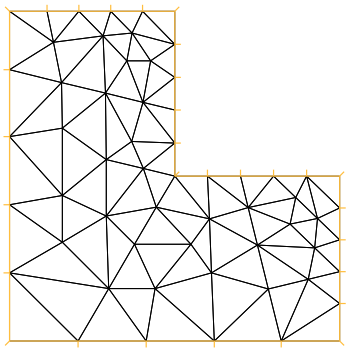
Mallas con hueco generada con **hueco.edp**

El siguiente ejemplo muestra una geometría poligonal y se usan las órdenes **savemesh** y **readmesh** para guardar y leer una malla:

Archivo **ele.edp**

```
border a(t=0,1){x=t;y=0;label=1;};
border b(t=0,0.5){x=1;y=t;label=1;};
border c(t=0,0.5){x=1-t;y=0.5;label=1;};
border d(t=0.5,1){x=0.5;y=t;label=1;};
border e(t=0.5,1){x=1-t;y=1;label=1;};
border f(t=0,1){x=0;y=1-t;label=1;};
int n=5;
mesh rh=buildmesh(a(n)+b(n)+c(n)+d(n)+e(n)+f(n));
```

```
plot(rh,wait=1);
savemesh(rh,"ele.msh");
mesh th=readmesh("ele.msh");
plot(th,wait=1,ps="ele.eps");
```



Malla para L generada con **ele.edp**

Observación 1 *Si se usa el nombre por defecto de las partes de la frontera, este nombre se pierde cuando se guarda la malla. Por lo tanto, en este caso es mejor usar la orden* ***label=...*** *y asignarle un número que sí se guarda junto con el resto de información de la malla.*

3. Problemas variacionales

En FreeFem++ los problemas se describen en la forma variacional. Por lo tanto hace falta una forma bilineal **a(u,v)**, una forma lineal **l(f,v)**, y condiciones de contorno. La expresión general es

```
problem P(u,v) = a(u,v) - l(f,v)
             + (boundary condition);
```

Nos encontramos con tres palabras clave: **problem**, **solve** y **varf**

- **problem ejemplo()...;** define el problema variacional construyendo el sistema lineal y asociandole una manera de invertirlo. El problema variacional se resuelve al usar la orden **ejemplo;**
- **solve ejemplo()...;** define y resuelve el problema variacional construyendo el sistema lineal y asociandole una manera de invertirlo.
- **varf** construye las partes del problema variacional y permite extraer la matriz y el termino independiente del problema.

Resolutores lineales: Básicamente tres: **CG**, **UMFPACK** y **GMRES**. Por defecto siempre se usa UMFPACK pero para sistemas grandes es mejor usar GMRES. Se fija el resolutor del sistema lineal con la orden **solver=...**

- **CG** Gradiente conjugado, es un método directo.
- **UMFPACK** es un método directo, permite manejar cualquier tipo de matriz mediante una factorización.
- **GMRES** método iterativo para matrices grandes y huecas.

El ejemplo **Poisson-escalon.edp** resuelve el problema de Poisson en un escalón hacia adelante con datos de contorno de tipo Dirichlet:

Archivo **Poisson-escalon.edp**

```
real v1=4,v2=6,w=v1+v2;
border a(t=0,v1){x=t;y=0;label=1;}; //lado y=0, x en [0,v1]
border b(t=0,1){x=v1;y=-t;label=2;};//lado x=v1, y en [0,-1]
border c(t=0,v2){x=v1+t;y=-1;label=3;}; //lado y=-1, x en [v1,w]
border d(t=0,2){x=w;y=-1+t;label=4;};//lado x=w, y en [-1,1]
border e(t=0,w){x=w-t;y=1;label=5;};//lado y=1, x en [0,w]
border ff(t=0,1){x=0;y=1-t;label=6;};//lado x=0, y en [1,0]
int n=10;
mesh th=buildmesh(a(2*n)+b(n)+c(2*n)+d(2*n)+e(4*n)+ff(n));
plot(th,wait=1,ps="malla-escalon.eps");
fespace Vh(th,P1);
Vh u,v;
func f=0;
problem laplace(u,v) =int2d(th)(dx(u)*dx(v)+dy(u)*dy(v))  //  bilinear part
                    +int2d(th)(-f*v)               // right hand side
      +on(1,2,3,4,u=0)+on(5,u=1)+on(6,u=0);// Dirichlet boundary condition
laplace;  //solve the pde
plot(u,wait=1,ps="Poisson-escalon.eps");
```

Observación 2 *Las funciones* ***int2d(th)(...) y similares****, se usan*

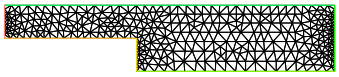
- ***con la función test y la incognita*** *o*
- ***sólo con la función test***
- *pero* ***no ambas*** *situaciones.*

Es decir, ***es incorrecto***

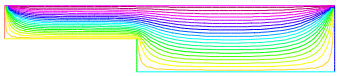
```
int2d(th)(dx(u)*dx(v)+dy(u)*dy(v)-f*v)
      +on(1,u=0)
```

y es correcto

```
int2d(th)(dx(u)*dx(v)+dy(u)*dy(v))    //  bilinear part
      +int2d(th)(-f*v)                // right hand side
      +on(1,u=0)
```



Malla generada con **Poisson-escalon.edp**



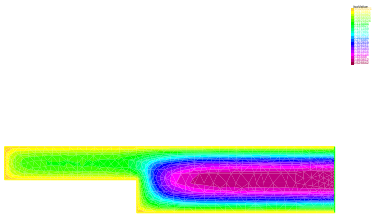
Solución obtenida con **Poisson-escalon.edp**

Incluir una condición de contorno de tipo Neumann con derivada normal cero equivale a no dar una especificación sobre este trozo de frontera:

Archivo **Poisson-escalonNeumann.edp**

real v1=4,v2=6,w=v1+v2;
border a(t=0,v1){x=t;y=0;label=1;}; //lado y=0, x en [0,v1]
border b(t=0,1){x=v1;y=-t;label=1;};//lado x=v1, y en [0,-1]

```
border c(t=0,v2){x=v1+t;y=-1;label=1;}; //lado y=-1, x en [v1,w]
border d(t=0,2){x=w;y=-1+t;label=2;};//lado x=w, y en [-1,1]
border e(t=0,w){x=w-t;y=1;label=2;};//lado y=1, x en [0,w]
border ff(t=0,1){x=0;y=1-t;label=1;};//lado x=0, y en [1,0]
int n=10;
mesh th=buildmesh(a(2*n)+b(n)+c(2*n)+d(2*n)+e(4*n)+ff(n));
plot(th,wait=1);
fespace Vh(th,P1);
Vh u,v;
func f=1;
problem laplace(u,v) =
int2d(th)(dx(u)*dx(v)+dy(u)*dy(v)) // bilinear part
+int2d(th)(-f*v) // right hand side
+on(1,u=0); // Dirichlet boundary condition
laplace; //solve the pde
plot(u,wait=1,value=1,ps="Poisson-escalonNeumann.eps");
```



Solucíon obtenida con **Poisson-escalonNewmann.edp**

El siguiente código **Poisson–elipse.edp** resuelve mediante elementos finitos P2 el problem variacional

$$(\nabla u, \nabla v) = (f, v)$$

con dato de contorno $u = 0$ en una elipse:
Archivo **Poisson-elipse.edp**

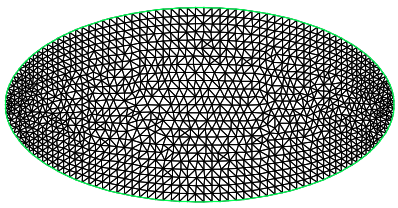
```
border a(t=0,2*pi){x=2*cos(t);y=sin(t);label=5;};
mesh Th= buildmesh (a(150));
plot(Th,wait=1,ps="malla-elipse.eps");
fespace Vh(Th,P2);
```

```
Vh u, v;
func f=sin(x*y);
problem laplace(u,v) =
           int2d(Th)(dx(u)*dx(v)+dy(u)*dy(v)) //bilinear part
-int2d(Th)(f*v)                        // right hand side
+on(5,u=0);           // Dirichlet boundary condition
real cpu=clock();
laplace;        //solve the pde
plot(u,wait=1,ps="u-Poisson-elipse.eps");
cout<< " CPU = " << clock() -cpu << endl;
```

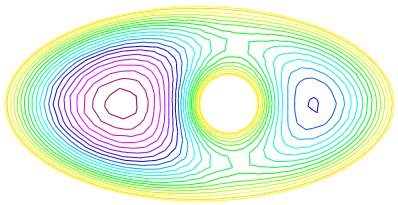
Observación 3 *El tiempo usado en el cálculo se puede obtener con*

```
real d=clock();\\Get current CPU clock time
...
cout<< " CPU = " << clock() - d << endl; \\Gives CPU time taken
                                          \\for the process
```

se obtiene el tiempo de CPU empleado por todas las instrucciones entre ambas líneas.



Malla para elipse generada con **Poisson-elipse.edp**



Solución obtenida con **Poisson-elipse.edp**

Adaptación de mallado: Se hace tomando como referencia el Hessiando de una función dada, que puede incluso ser la solución obtenida de un problema: Archivo **Poisson-adap.edp**

```
int n=5;
mesh Th=square(n,n,[10*x,5*y]);// Malla inicial uniforme en [0,10]x[0,5]
plot(Th,wait=1,ps="malla0.eps");
fespace Vh(Th,P1);
Vh u=0,v=0,zero=0;
func f=-sin(x*y);
func g=x+y;
int i=0;
real error=0.1, coef=0.1^(1./5.);
real cpu=clock();
problem Problem1(u,v,solver=CG,init=i,eps=-1.0e-6)=
int2d(Th)( dx(u)*dx(v)+dy(u)*dy(v)) +int2d(Th)( v*f)
+on(1,2,3,4,u=g);
Problem1;
plot(u,zero,wait=1,ps="uinicial.eps");
Th= adaptmesh(Th,u,inquire=1,err=error);
real d=clock();
cout << " CPU = "<<cpu -d <<endl;
for (i=1;i<5;i++)
{
Problem1;
plot(Th,u,zero,wait=1);
```

```
Th= adaptmesh(Th,u,inquire=1,err=error);
cout << " CPU = "<<clock() -d <<endl;
d=clock();
error=error*coef;
};
cout << " CPU = "<<clock() -cpu <<endl;
plot(Th,u,zero,wait=1,ps="ufinal.eps");
plot(Th,wait=1,ps="mallafin.eps");
```

La opción **init=i** con $i \neq 0$ fuerza a reconstruir el sistema lineal cada vez. Esto es lógico puesto que se ha remallado.

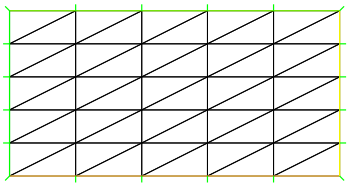
El parámetro **eps=...** indica un criterio de parada para la resolución del sistema lineal.

- Si $eps < 0$ el criterio es el error absoluto menor que -eps

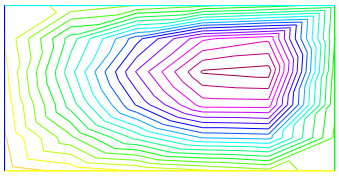
$$\|Ax - b\| \leq |eps|.$$

- Si $eps > 0$ el criterio es el error relativo

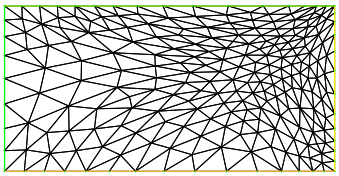
$$\|Ax - b\| \leq eps \cdot \|Ax_0 - b\|.$$



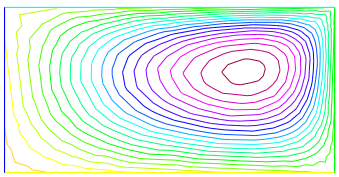
Malla inicial generada con **Poisson-adap.edp**



Solucion inicial obtenida con **Poisson-adap.edp**



Malla final generada con **Poisson-adap.edp**



Solucion final obtenida con **Poisson-adap.edp**

Observación 4 *Observemos que*

- *Los comentarios se escriben después de //*
- **fspace Vh(Th,P1)** *construye un espacio de elementos finitos de nombre Vh y con elementos P1 sobre Th.*

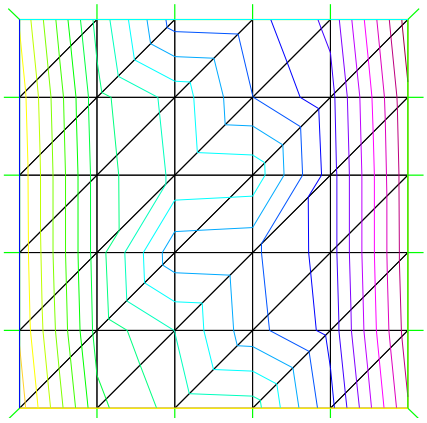
- **Vh u,v,...** *declara distintos elementos de Vh.*
- **func g=...** *declarar una funcion mediante una expresión analítica.*
- *Los sistemas lineales se resuelven mediante CG que equivale a Conjugate Gradient o mediante LU que equivale a la factorización LU.*
- *La malla se adapta mediante* **adaptmesh**

Otro ejemplo, se obtiene con la función

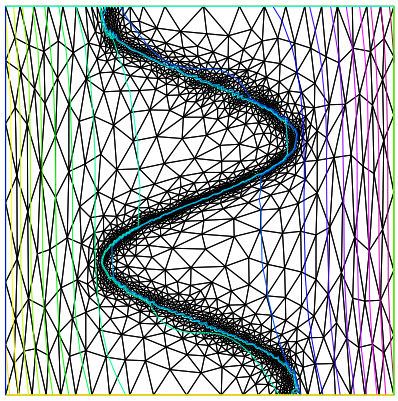
$$f(x,y) = 10\,x^3 + y^3 + atan^{-1}[\epsilon/(sin(5\,y) - 2\,x)], \quad \epsilon = 0{,}0001$$

que cambia muy rapido y una malla uniforme no puede verla. Observemos como se adapta la malla:
Código **remallar.edp**

```
real eps = 0.0001;
real h=1;
real hmin=0.05;
func f = 10.0*x^3+y^3+h*atan2(eps,sin(5.0*y)-2.0*x);
mesh Th=square(5,5,[-1+2*x,-1+2*y]);
fespace Vh(Th,P1);
Vh fh=f;
plot(Th,fh, ,ps="remallainit.eps");
for (int i=0;i<5;i++)
{
Th=adaptmesh(Th,fh);
fh=f; // old mesh is gone
plot(Th,fh,wait=1,ps="remallafin.eps");
}
```



Malla y función inicial generada con **Poisson-adap.edp**



Malla final y función generada con **Poisson-adap.edp**

4. Gráficas

Existen varias opciones una vez que se tiene una gráfica en pantalla con el comando plot. La tecla ? genera un menú con las distintas opciones. Algunas de ellas son:

- p se muestra la gráfica anterior.
- f ó click del ratón avanza el proceso
- +,- hace un zoom + ‘o - en torno a la posición del ratón
- = restaura el zoom

- g cambia de grís a color
- m muestra la malla
- 3 muestra una versión 3D de la gráfica
- b cambia de color a blanco y negro
- c y C aumenta o disminuye el coeficiente de las flechas
- f rellena color entre líneas de corriente
- v muestra los valores numéricos en las líneas de corriente

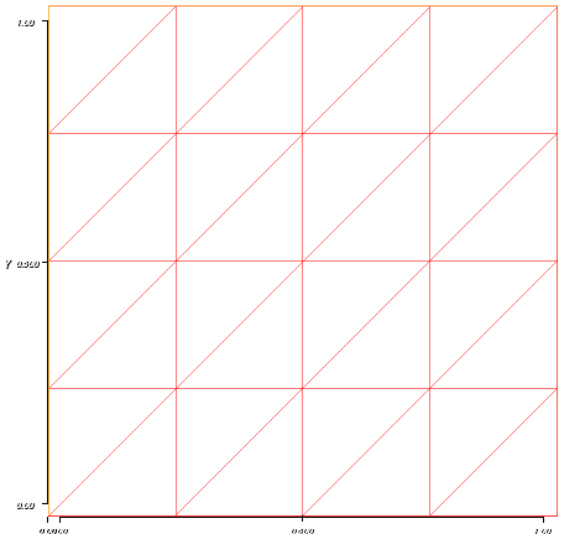
En el siguiente ejemplo se resuelve en un rectángulo y mediante elementos finitos P2 el problem variacional

$$(\nabla u, \nabla v) + (g\,u, v) = (f, v)$$

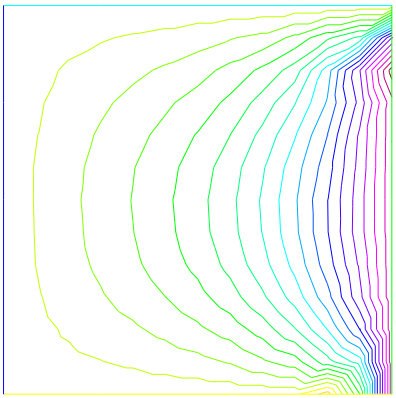
con dato de contorno $u = 0$.
Archivo **PoissonReaccion-adap.edp**

```
real a=1,b=1;
int n=4,m=4;
mesh Th=square(n,m,[a*x,b*y]);// Uniform mesh on  [0,a]x[0,b]
plot(Th,wait=1,ps="PRmalla0.eps");
fespace Vh(Th,P2);
Vh u=0,v=0;
func f=1;
func g=10*x*y;
int i=0;
real error=0.1, coef=0.1^(1./5.);
problem Problem1(u,v,solver=CG,init=i,eps=-1.0e-6)=
        int2d(Th)( dx(u)*dx(v)+dy(u)*dy(v)+g*u*v)
        -int2d(Th)( v*f)+on(1,u=0)
        +on(2,u=1)+on(3,4,u=0);
real cpu=clock();
Problem1;
plot(u,wait=1,ps="PRuinicial.eps");
for (i=0;i<5;i++){
real d=clock();
Problem1;
cout << " CPU = " << clock() -d << endl;
```

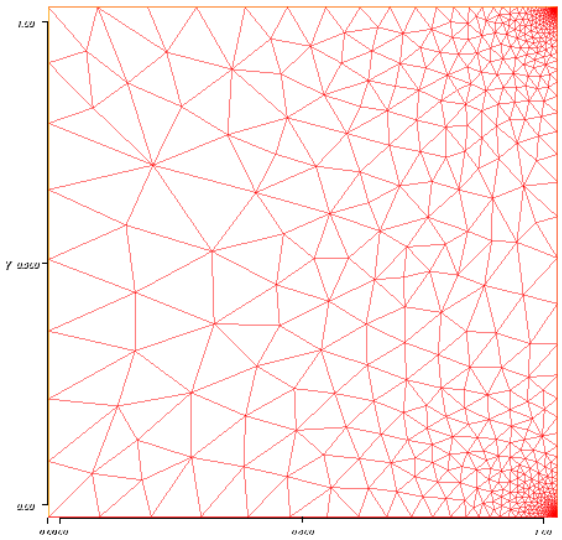
```
plot(u,wait=1);
Th= adaptmesh(Th,u,inquire=1,err=error);
error=error*coef;
};
plot(u,wait=1,ps="PRufinal.eps");
plot(Th,wait=1,ps="PRmallafin.eps");
```



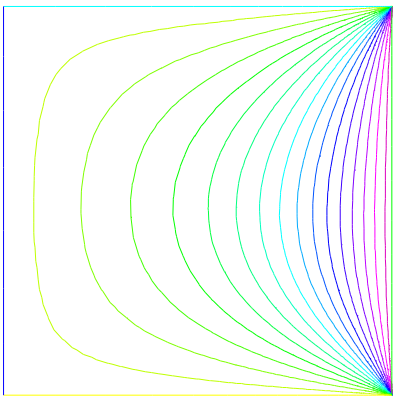
Malla inicial generada con **PoissonReaccion-adap.edp**



Solucion inicial obtenida con **PoissonReaccion-adap.edp**



Malla final generada con **PoissonReaccion-adap.edp**

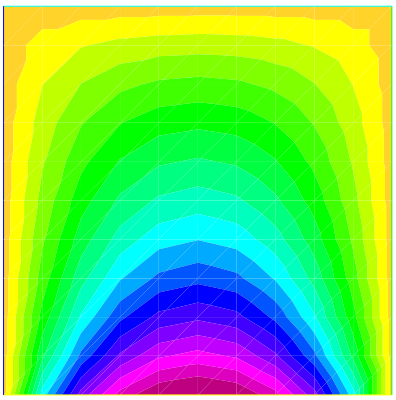


Solucion final obtenida con **PoissonReaccion-adap.edp**

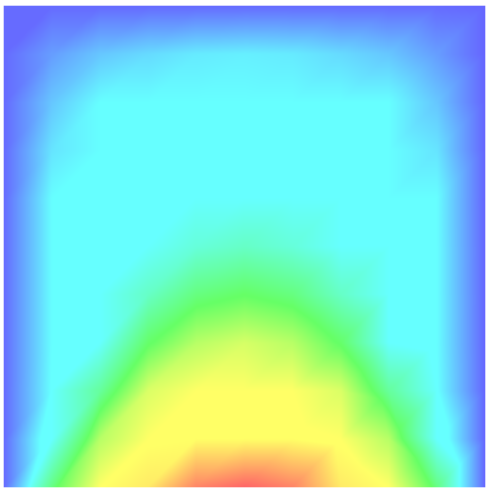
Otra forma de obtener salida gráfica 2d es mediante la libreria **medit** de Pascal Frey.
Archivo **LaplaceP1.edp**:

```
load "medit";
 mesh Th=square(10,10);
 fespace Vh(Th,P1);      // P1 FE space
 Vh uh,vh;               // unkown and test function.
 func f=1;                  //  right hand side function
 func g=0;                  //  boundary condition function
```

```
problem laplace(uh,vh) =
   int2d(Th)( dx(uh)*dx(vh) + dy(uh)*dy(vh) ) //  bilinear form
 + int1d(Th,1)( uh*vh)
 - int1d(Th,1)( vh)
 - int2d(Th)( f*vh )                          //  linear form
 + on(2,3,4,uh=g) ;                      //  boundary condition form

 laplace; // solve the problem plot(uh);
 plot(uh,ps="Laplace.eps",fill=1);// to see the result
  medit("Laplace",Th,uh);
```



Solución generada con **PLaplaceP1.edp** usando **plot**



Solución generada con **PLaplaceP1.edp** usando **medit**

5. Problemas variacionales

FreeFem++ provee de varios tipos de elementos finitos:

- P0
- P1
- P1dc (P1 discontinuo)
- P1b (P1 + burbuja)
- P2
- P2dc
- RT0 (Raviart-Thomas)

Ejemplo de como se define el espacio de elementos finitos dada la malla es:

```
real a=1.0,b=1.0;
int n=10,m=10;
mesh Th=square(n,m,[a*x,b*y]);// Malla uniforme en [0,a]x[0,b]
plot(Th,wait=1);
```

```
fespace Vh(Th,P2);
Vh u,v,zero;
Vh[int] uh(5);
...
```

Donde la última orden permite definir una función vectorial de 5 componentes donde cada componente es un elemento de Vh.

Orden de convergencia

Se puede comprobar el orden de convergencia de los elementos finitos con el siguiente código
Código **orden2d.edp**:

```
verbosity =0;
real a=1.5,b=1.;
//border Gamma(t=0,2*pi){x =a*cos(t);y=b*sin(t);}
real ratio;
int npasos=10;
func phiexact=sin(x)*sin(y);
func fexact=2*phiexact;
real[int] ErrorH1(npasos),ErrorL2(npasos),ta(npasos);
for(int n=0;n<npasos;n++)
 {
//       mesh Th=buildmesh(Gamma(10*(n+1)));
        mesh Th=square(10*(n+1),10*(n+1));
        int nbverts=Th.nv;
        fespace Mh(Th,P0);
        Mh hhh = hTriangle;
cout << " mesh size = "<< hhh[].max
    <<" nb of vertices = " << nbverts<<endl;
        ta[n]=hhh[].max;
        fespace Vh(Th,P2);
        Vh phi,w,errorh,phih,fh;
        phih=phiexact;
        fh=fexact;
        solve laplace(phi,w)=int2d(Th)(dx(phi)*dx(w) + dy(phi)*dy(w))
                    - int2d(Th)(fh*w)
//                    +on(Gamma,phi=phih);
                     +on(1,2,3,4,phi=phih);
         errorh=phi-phih;
```

```
        ErrorH1[n]= int2d(Th)(dx(errorh)^2+dy(errorh)^2);
        ErrorL2[n]= int2d(Th)(errorh^2);
};
cout << "+++++++++++++++++++++++++++++++++++ " <<endl;
for(int n=0;n<npasos;n++)
{
cout << " Error seminorm H1 " << n << " = "<<sqrt( ErrorH1[n]) <<endl;
};
for(int n=0;n<npasos-1;n++)
{
ratio=log(sqrt(ErrorH1[n])/sqrt(ErrorH1[n+1]));
ratio=ratio/log(ta[n]/ta[n+1]);
cout <<" convergence rate error semi norm H1= "<< ratio <<endl;
};
cout << "+++++++++++++++++++++++++++++++++++ " <<endl;
for(int n=0;n<npasos;n++)
{
cout << " ErrorL2 " << n << " = "<< sqrt(ErrorL2[n]) <<endl;
};
for(int n=0;n<npasos-1;n++)
{
ratio=log(sqrt(ErrorL2[n])/sqrt(ErrorL2[n+1]));
ratio=ratio/log(ta[n]/ta[n+1]);
cout <<" convergence rate error L2= "<< ratio <<endl;
};
for(int n=0;n<npasos;n++)
{
cout << " Error norm H1 " << n << " = "<< sqrt(ErrorL2[n]+ErrorH1[n]) <<endl;
};
for(int n=0;n<npasos-1;n++)
{
ratio=log(sqrt(ErrorL2[n]+ErrorH1[n])/sqrt(ErrorL2[n+1]+ErrorH1[n+1]));
ratio=ratio/log(ta[n]/ta[n+1]);
cout <<" convergence rate error norm H1= "<< ratio <<endl;
};
```

los resultados para elementos finitos P2 sobre una triangulación uniforme son

```
size of mesh = 0.141421 nb of vertices = 121
```

```
size of mesh = 0.0707107 nb of vertices = 441
size of mesh = 0.0471405 nb of vertices = 961
size of mesh = 0.0353553 nb of vertices = 1681
size of mesh = 0.0282843 nb of vertices = 2601
size of mesh = 0.0235702 nb of vertices = 3721
size of mesh = 0.0202031 nb of vertices = 5041
size of mesh = 0.0176777 nb of vertices = 6561
size of mesh = 0.0157135 nb of vertices = 8281
size of mesh = 0.0141421 nb of vertices = 10201
++++++++++++++++++++++++++++++++++++
 Error seminorm H1 0 = 1.29502e-005
 Error seminorm H1 1 = 1.69264e-006
 Error seminorm H1 2 = 5.08725e-007
 Error seminorm H1 3 = 2.1613e-007
 Error seminorm H1 4 = 1.11122e-007
 Error seminorm H1 5 = 6.44848e-008
 Error seminorm H1 6 = 4.06886e-008
 Error seminorm H1 7 = 2.72984e-008
 Error seminorm H1 8 = 1.91945e-008
 Error seminorm H1 9 = 1.40056e-008
 convergence rate error semi norm H1= 2.93563
 convergence rate error semi norm H1= 2.96483
 convergence rate error semi norm H1= 2.9756
 convergence rate error semi norm H1= 2.98129
 convergence rate error semi norm H1= 2.98481
 convergence rate error semi norm H1= 2.98722
 convergence rate error semi norm H1= 2.98896
 convergence rate error semi norm H1= 2.99028
 convergence rate error semi norm H1= 2.99133
++++++++++++++++++++++++++++++++++++
 ErrorL2 0 = 2.89877e-007
 ErrorL2 1 = 1.85985e-008
 ErrorL2 2 = 3.69895e-009
 ErrorL2 3 = 1.17389e-009
 ErrorL2 4 = 4.81636e-010
 ErrorL2 5 = 2.3251e-010
 ErrorL2 6 = 1.25613e-010
 ErrorL2 7 = 7.36696e-011
 ErrorL2 8 = 4.60503e-011
 ErrorL2 9 = 3.02595e-011
```

```
convergence rate error L2= 3.96218
convergence rate error L2= 3.98316
convergence rate error L2= 3.98955
convergence rate error L2= 3.99246
convergence rate error L2= 3.99434
convergence rate error L2= 3.99433
convergence rate error L2= 3.99618
convergence rate error L2= 3.98916
convergence rate error L2= 3.98559
Error norm H1 0 = 1.29534e-005
Error norm H1 1 = 1.69274e-006
Error norm H1 2 = 5.08739e-007
Error norm H1 3 = 2.16133e-007
Error norm H1 4 = 1.11123e-007
Error norm H1 5 = 6.44852e-008
Error norm H1 6 = 4.06888e-008
Error norm H1 7 = 2.72985e-008
Error norm H1 8 = 1.91946e-008
Error norm H1 9 = 1.40056e-008
convergence rate error norm H1= 2.9359
convergence rate error norm H1= 2.96491
convergence rate error norm H1= 2.97564
convergence rate error norm H1= 2.98131
convergence rate error norm H1= 2.98483
convergence rate error norm H1= 2.98723
convergence rate error norm H1= 2.98897
convergence rate error norm H1= 2.99029
convergence rate error norm H1= 2.99133
```

Observación 5 *Se ve perfectamente el orden de convergencia en el caso de una triangulación uniforme. Pero* ***no se puede estimar de la misma forma*** *cuando la triangulación deja de serlo.*

Se puede obtener información sobre la triangulación como sigue

```
mesh Th=square(10*(n+1),10*(n+1));
int nbvertices=Th.nv;
fespace Mh(Th,P0);
Mh hhh = hTriangle;
cout << "size of mesh = "<< hhh[].max
    <<" nb of vertices = " << nbvertices<<endl;
```

Se usa que **hTriangle** es el tamaño de la arista más larga de cada triángulo y que **Th.nv** es el número de vértices de la triangulación.

Además, dado un espacio de elementos finitos Vh(Th,*) donde * denota el tipo de funciones, entonces existen los siguientes valores:

- **Vh.nt** es el número de elementos en Vh
- **Vh.ndof** es el número de grados de libertad asociados a las funciones de Vh, es decir, la dimensión del espacio de elementos finitos Vh
- **Vh.ndofK** es el número de grados de libertad en cada elemento K de Th
- **Vh(i,k)** da el número del grado de libertad i asociado al elemento k

Código **FE.edp**:

```
mesh Th=square(5,5);
fespace Wh(Th,P2);
cout << " nb of degree of freedom : " << Wh.ndof << endl;
cout << " nb of degree of freedom / ELEMENT : " << Wh.ndofK << endl;
int k= 2;
int kdf= Wh.ndofK ;
cout << " df of element " << k << ":" ;
for (int i=0;i<kdf;i++)
cout << Wh(k,i) << " ";
cout << endl;
```

genera una salida

```
Nb Of Nodes = 121
Nb of DF = 121 F
ESpace:Gibbs: old skyline = 5841 new skyline = 1377
nb of degree of freedom : 121
nb of degree of freedom / ELEMENT : 6
df of element 2:78 95 83 87 79 92
```

Visualización de funciones de elementos finitos

Existen dos operadores para introducir o eliminar triángulos de una malla o para eliminarlos. Tenemos **trunc** que tiene dos parámetros:

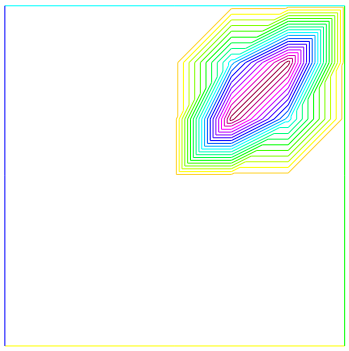
- **label=** fija el número de etiqueta del nuevo trozo de frontera (uno por defecto).
- **split=** fija el nivel n de triángulos en los que se dividen los triángulos. Cada uno se divide en $n \times n$ (uno por defecto) .

Para crear una malla Th3 donde los triangulos de una malla Th se dividen en 3×3 simplemente se escribe:

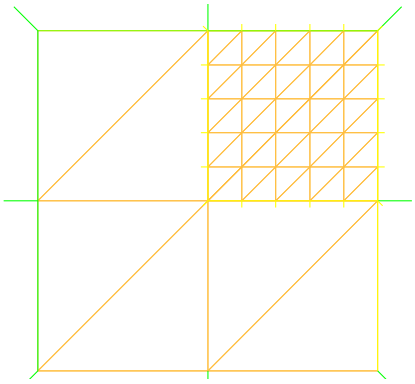
```
mesh Th3 = trunc(Th,1,split=3);
```

En el siguiente ejemplo mostramos los soportes de las funciones de base. Código **base.edp**:

```
mesh Th=square(3,3);
fespace Vh(Th,P1);
Vh u;
int i,n=u.n;
u=0;
for (i=0;i<n;i++)    // all degree of freedom
{
u[][i]=1;                          // the basic function i
plot(u,wait=1,ps="base"+i+".eps");
mesh Sh1=trunc(Th,abs(u)>1.e-10,split=5,label=2);
plot(Th,Sh1,wait=1,ps="trunc"+i+".eps"); // plot the mesh of
                                 //the functions support
u[][i]=0;               // reset
}
```



Función de base 10 obtenida con **base.edp**



Soporte función de base 10 obtenido con **base.edp**

Salida y lectura a archivos de datos

Para guardar un resultado usamos

```
{
ofstream u1data("u1.txt");
u1data << uu1[];
ofstream u2data("u2.txt");
u2data << uu2[];
}
```

y para leer un dato de un fichero y guardarlo en una función de elementos finitos usamos

```
{
ifstream u1data("u1.txt");
u1data >> uu1[];
ifstream u2data("u2.txt");
u2data >> uu2[];
}
```

donde **uu1** o **uu2** son las funciones de elementos finitos y **uu1**[] el vector de valores asociado a cada funcion de elementos finitos.

Observación 6 *Al igual que en C/C++ el entorno*

```
{
....
}
```

plantea un uso local de memoria, es decir, las variables definidas dentro son locales.

6. Ecuaciones de Stokes y Navier-Stokes

Resolvemos ahora el problema de Stokes en un cuadrado $[0,1] \times [0,1]$ con una malla rectangular y con los datos de contorno del problema de la cavidad. Se usa el par de elementos finitos (P2,P1) para las incógnitas velocidad-presión. También se obtienen las lineas de corriente dadas por Ψ tal que

$$rot(\Psi) = u, \quad \text{o mejor } -\Delta\Psi = rot(u) = \partial_y u_1 - \partial_x u_2.$$

Observación 7 *La compatibilidad de la condición* ***inf-sup*** *a nivel continuo la tienen los espacios* $(H_0^1(\Omega))^d$ *y* $L_0^2(\Omega)$*. Por lo tanto, el par (P2,P1)* ***no es compatible****. La compatibilidad correcta se encuentra con el par (P2,P1/R). Para evitar esta situación, que también produciría además una matriz llena, se* ***le añade el término*** $\epsilon\, p\, q$ *que hace que el problema continuo se plantee en los espacios* $(H_0^1(\Omega))^d$ *y* $L^2(\Omega)$ *siendo ya compatible el sistema y distando la solución de la correcta un orden de* ϵ*.*

Observación 8 *Se puede observar la inestabilidad en la presión que se genera si se usa el par de elementos finitos (P2,P1).*

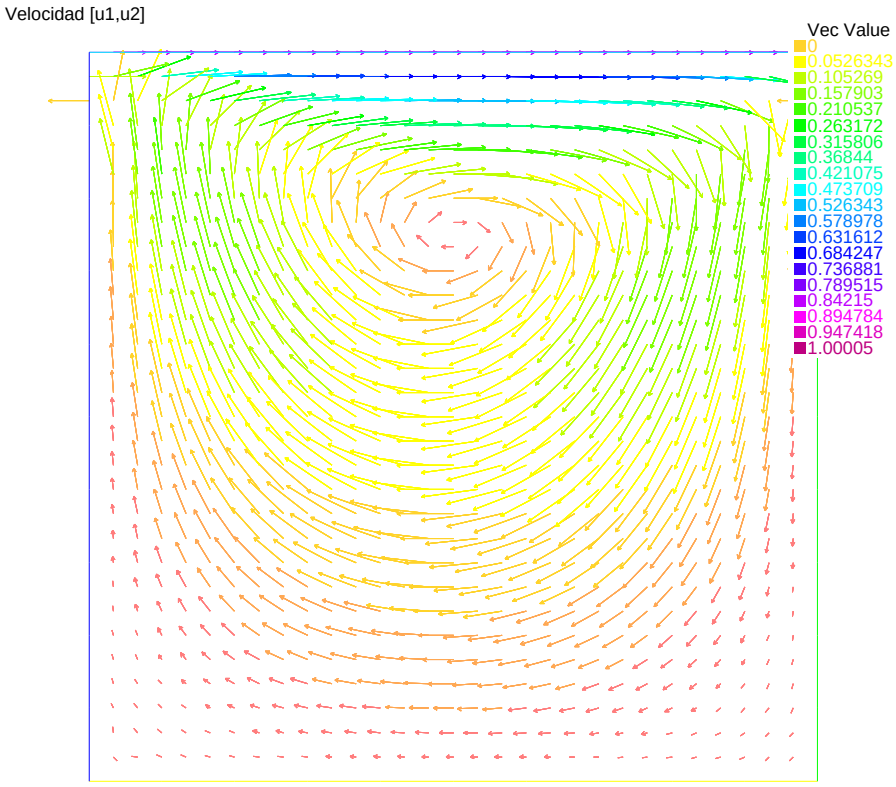
Observación 9 *UMFPACK elimina los autovalores cero de la matriz del sistema lineal que resuelve. Es por ello que en el caso* $\epsilon = 0$ *también da la solución correcta.*

Observación 10 *UMFPACK utiliza la técnica de bloqueo mediante un valor grande para fijar los datos de tipo Dirichlet.*

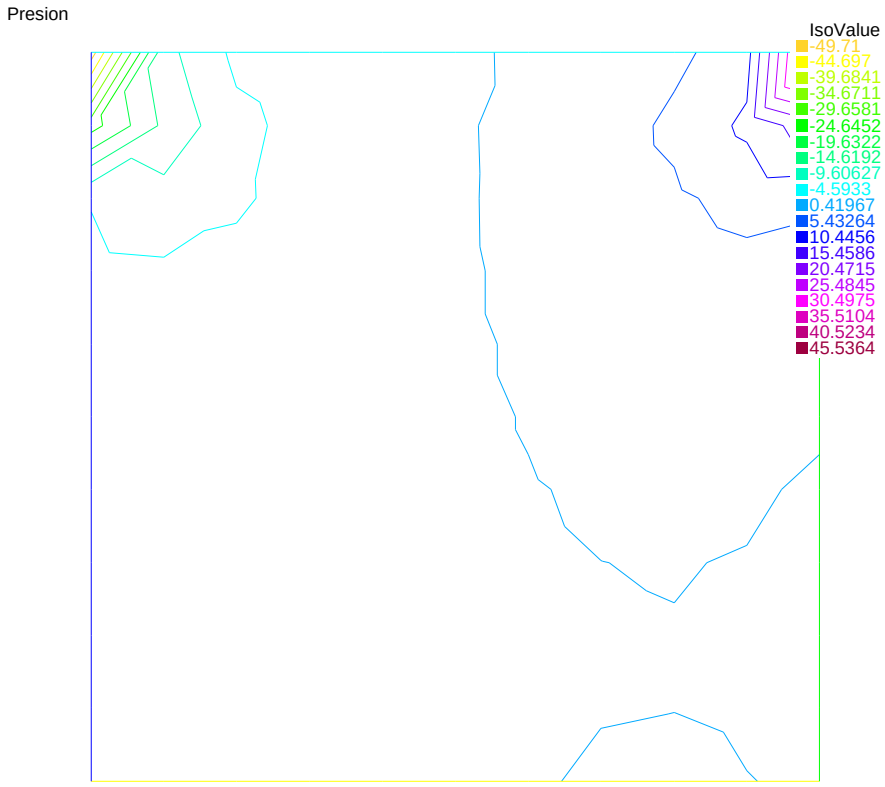
Código **Stokes2dCavidad.edp**:

```
mesh Th=square(10,10);
fespace Xh(Th,P2);
fespace Mh(Th,P1);
Xh u1,v1,u2,v2;
Mh p,q;
solve Stokes ([u1,u2,p],[v1,v2,q]) =
           int2d(Th)(
                       (dx(u1)*dx(v1)+dy(u1)*dy(v1)
                        +
                        dx(u2)*dx(v2)+dy(u2)*dy(v2)
                       )
```

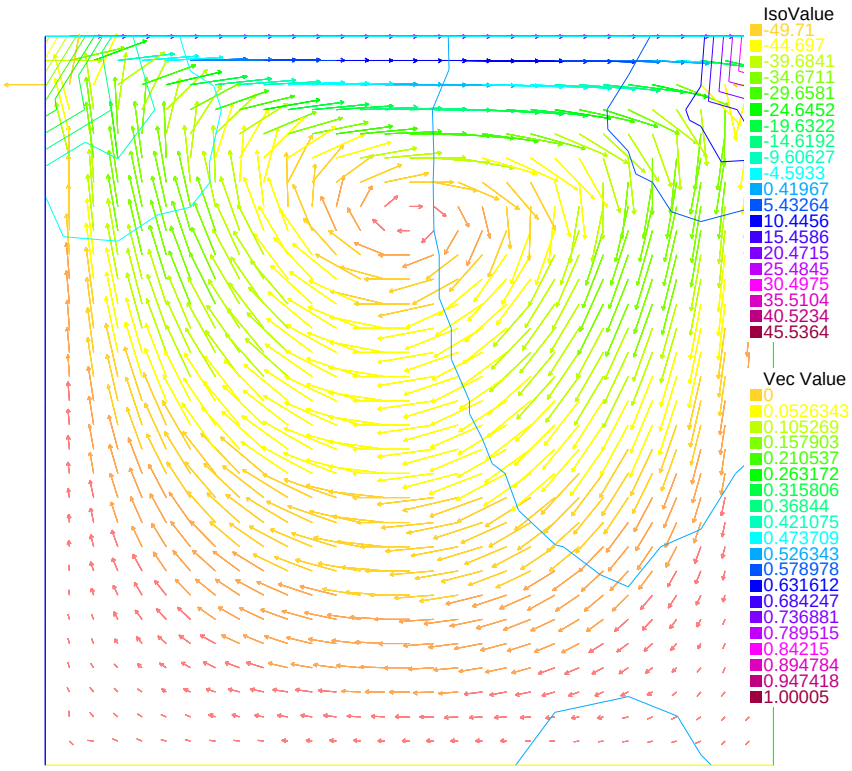
```
                        +p*q*0.000001
                        -p*dx(v1)-p*dy(v2)
                        +dx(u1)*q+dy(u2)*q
                    )
                        +on(1,2,4,u1=0,u2=0)+on(3,u1=1,u2=0);
plot(coef=.5,cmm=" Velocidad [u1,u2] ",
    value=true,[u1,u2],wait=1,ps="Stokes-vel.eps");
plot(cmm=" Presion ",p,value=true,wait=1,ps="Stokes-pres.eps");
plot([u1,u2],p,wait=1,value=true,coef=0.5,ps="Stokes-velypres.eps");
Xh psi,phi;
solve streamlines(psi,phi) =
             int2d(Th)(dx(psi)*dx(phi) + dy(psi)*dy(phi)
                              )
          +int2d(Th)( -phi*(dy(u1)-dx(u2))
                               )
          +on(1,2,3,4,psi=0);
plot(cmm=" Streamlines "psi,wait=1, ps="Stokes-stream.eps");
```



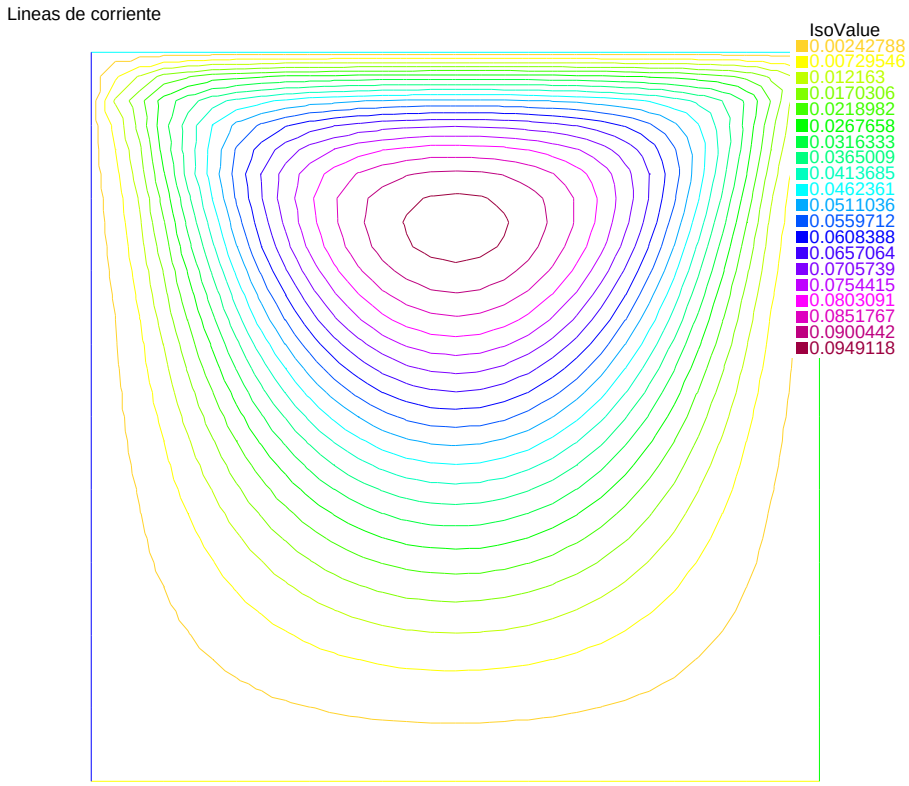
Velocidad obtenida con **Stokes2dCavidad.edp**



Presión obtenida con **Stokes2dCavidad.edp**



Velocidad y Presión obtenida con **Stokes2dCavidad.edp**



Función de corriente obtenida con **Stokes2dCavidad.edp**

Observar como

- la orden **square** genera el cuadrado $[0,1]\times[0,1]$ y además etiqueta los contornos por defecto por 1,2,3,4 empezando por la base y en sentido contrario al del giro de un reloj.
- **coef= ...** da la razón de aspecto de los módulos de los vectores con respecto a la dimensión del dominio.
- **cmm= "..."** pone texto en la gráfica
- La función de corriente es solución del problema

$$\begin{aligned} -\Delta\Psi &= \nabla\times\mathbf{u}, \quad \Omega \\ \Psi &= 0, \quad \partial\Omega. \end{aligned}$$

Podemos simular ahora el problema de la cavidad para las ecuaciones de Navier-Stokes también con el par P2-P1 para velocidad-presión y tomando como dato inicial velocidad cero.

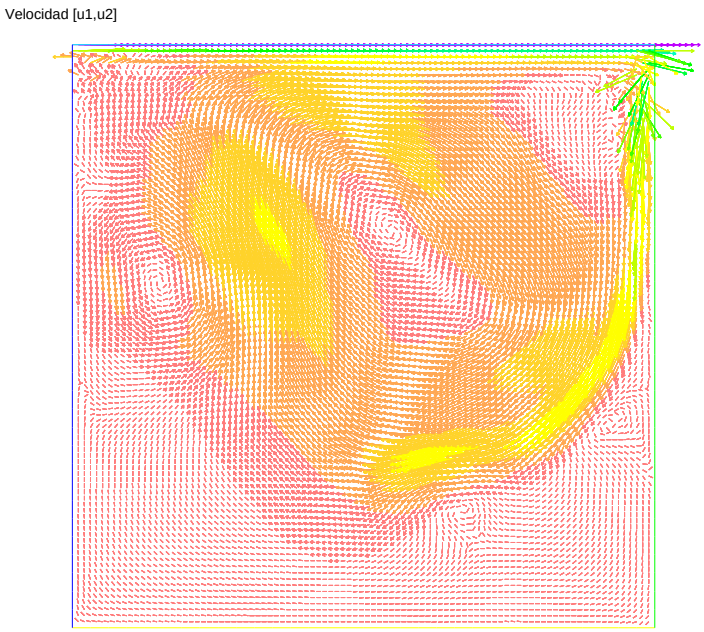
Observación 11 *La expresión* (**a**%**b**) *en C++ equivale a la división en módulo de a entre b y* ! *es el operador lógico de negación. La expresión* !(**i**%**10**) *permite mostrar gráfica cada vez que i es múltiplo de 10.*

if (!(*i*%10))
{
plot(...)
};

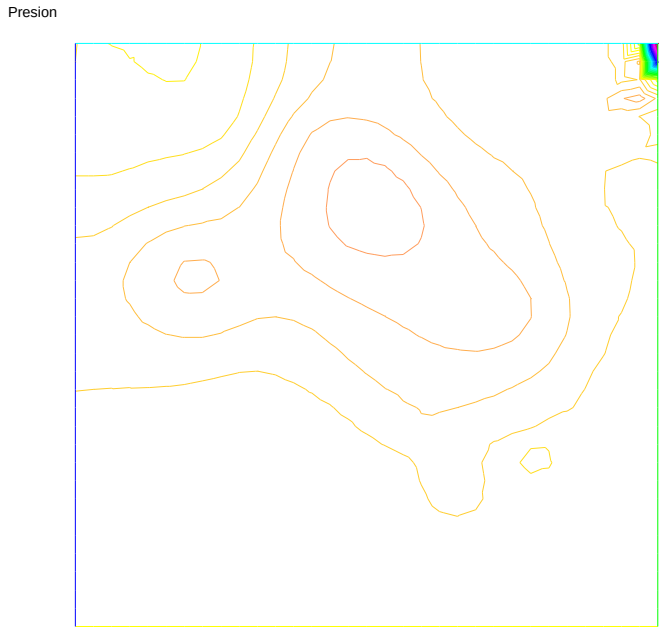
La orden **nbiso=...** indica el número de isolvalores que se van a representar. Por defecto sólo son 20.
Código **cavidad2d.edp**:

```
mesh Th=square(32,32);
fespace Xh(Th,P2);
fespace Mh(Th,P1);
Xh u2,v2, u1,v1,up1=0,up2=0;
Mh p,q;
real  nu=0.00005;
real d=clock();
int i=0;
real dt=0.1;
real alpha=1/dt;
problem  NS(u1,u2,p,v1,v2,q,solver=UMFPACK,init=i) =
   int2d(Th)(
             alpha*(u1*v1+u2*v2)
             +nu*(dx(u1)*dx(v1)+dy(u1)*dy(v1)
                  +dx(u2)*dx(v2)+dy(u2)*dy(v2)
                 )
            + p*q*(0.00000001)
             -p*dx(v1)- p*dy(v2)
             +dx(u1)*q+ dy(u2)*q
            )
     +
        int2d(Th)(
            -alpha*convect([up1,up2],-dt,up1)*v1
            -alpha*convect([up1,up2],-dt,up2)*v2
```

```
                )
     +on(1,2,4,u1=0,u2=0)
     +on(3,u1=1,u2=0);
for (i=0;i<=400;i++)
{
NS;
up1=u1;
up2=u2;
//if ( !(i % 20))
//plot(coef=0.3,cmm=" [u1,u2]  y   p  ",p,[u1,u2],fill=0,value=1);
plot(coef=.5,value=1, fill=1,cmm=" Velocidad  ",[u1,u2]);
};
cout<< " CPU = "<< clock() - d << endl;
plot(coef=.3,cmm=" Velocidad [u1,u2] ",[u1,u2],wait=1,ps="vel-ns.eps");
plot(cmm=" Presion ",p,wait=1,nbiso=50,ps="pres-ns.eps");
```



Velocidad obtenida con **Cavidad2d.edp**



Presión obtenida con **Cavidad2d.edp**

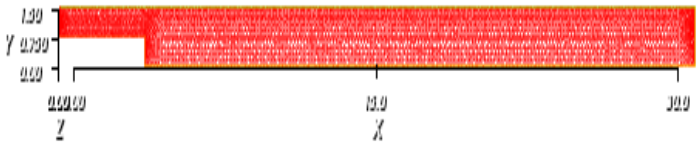
En el siguiente ejemplo obtenemos el flujo en un escalon hacia adelante partiendo también de la solución nula y además **usamos dos mallas distintas** para ver mas cerca la solución:

Código **NavierStokes-step.edp**:

```
real w1=4,w2=26,w=w1+w2,wcut=6,altura=1.5,step=0.75;
//
//                          e
// y=1.5->   |-----------------------------------|
//         ff|                                   |
// y=0.75->  |___________                        | d
//                  a     |                     |
//                       b|                     |
// y=0-->                 |_____________________|
//          x=0           x=4       c           x=30
//
//
border ff(t=0,step){x=0;y=altura-t;label=1;};// x=0, y en [step,altura]
border a(t=0,w1){x=t;y=step;label=2;};     // y=step,x en [0,w1]
border b(t=0,step){x=w1;y=step-t;label=3;} // x=w1,y en [0,step]
```

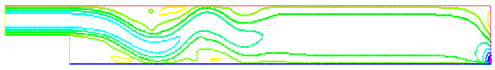
```
border c(t=0,w2){x=w1+t;y=0;label=4;};     //y=0,x en [w1,w1+w2]
border d(t=0,altura){x=w;y=t;label=5;};   //x=w, y en [0,altura]
border e(t=0,w){x=w-t;y=altura;label=6;};  // y=1.5,x en [0,w]
border ccut(t=0,wcut){x=w1+t;y=0;label=4;};    // y=0, x en [w1,w1+w2]
border dcut(t=0,altura){x=w1+wcut;y=t;label=5;};   //x=w, y en [0,altura]
border ecut(t=0,w1+wcut){x=w1+wcut-t;y=altura;label=6;}; // y=1.5,
                                                    // x en [0,w]
int n=25;

mesh th=buildmesh(ff(n)+a(2*n)+b(n)+c(6*n)+d(2*n)+e(12*n));
mesh thcut=buildmesh(ff(n)+a(2*n)+b(n)+ccut(6*n)+dcut(2*n)+ecut(12*n));
plot(th,wait=0,ps="step-mesh.eps");

fespace Xh(th,P2);
fespace Mh(th,P1);
fespace Xhcut(thcut,P2);
fespace Mhcut(thcut,P1);

Xh u1,v1,u2,v2,up1=0,up2=0;
Mh p,q;
Xhcut u1cut,v1cut;
Mhcut pcut;
func f1=0;
func f2=0;
func g=(y-0.75)*(1.5-y)/((2.25/2-0.75)*(1.5-2.25/2));// flujo de entrada
real  nu=0.0025;
int i=0,nmax=100;
real dt=0.1,alpha=1/dt;
problem  ns(u1,u2,p,v1,v2,q,init=i) =
      int2d(th)(  alpha*(u1*v1+u2*v2)+
                  nu*(dx(u1)*dx(v1)+dy(u1)*dy(v1)    //  bilinear part
                     +dx(u2)*dx(v2)+dy(u2)*dy(v2)
                     )
                 +p*dx(v1)+p*dy(v2)+q*dx(u1)+q*dy(u2)
               )    //  bilinear part
      +int2d(th)(
                -alpha*convect([up1,up2],-dt,up1)*v1
                -alpha*convect([up1,up2],-dt,up2)*v2
                -f1*v1-f2*v2
                 )   // right hand side
```

```
      +on(2,u1=0,u2=0)
      +on(3,u1=0,u2=0)
      +on(4,u1=0,u2=0)
      +on(6,u1=0,u2=0)
      +on(1,u1=g,u2=0);   // Dirichlet boundary condition

for (i=0;i<=nmax;i++)
{
   ns;
   up1=u1;
   up2=u2;
   cout <<  " Valor de i = " << i << " \n";
if( !(i%3) )
        {
        u1cut=u1;
plot( u1cut);
        };
};
plot(cmm= "u1  ", u1,value=0,nbiso=10,wait=1,ps="ns-step-vel.eps");
plot(cmm= "presion   ", p,nbiso=10,value=0,ps="ns-step-pres.eps");
u1cut=u1;
pcut=p;
plot(cmm= "u1cut  ", u1cut,value=0,wait=1,ps="ns-step-velcut.eps");
plot(cmm= "presioncut   ", pcut,value=0,ps="ns-step-prescut.eps");
```



Malla usada con **NavierStokes-step.edp**

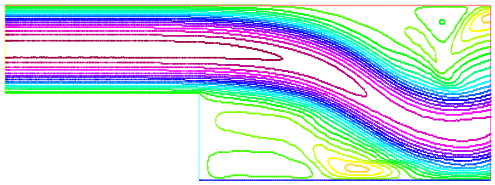
u1



Velocidad obtenida con **NavierStokes-step.edp**

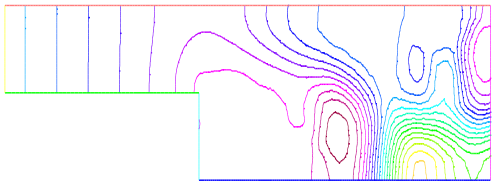
Presión obtenida con **NavierStokes-step.edp**

u1cut



Ampliacion de la velocidad obtenida con **NavierStokes-step.edp**

presioncut



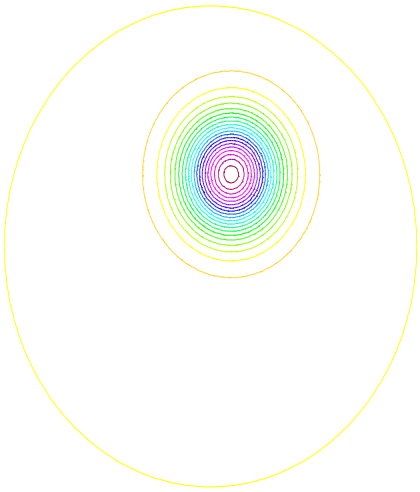
Ampliacion de la presión obtenida con **NavierStokes-step.edp**

Se usa la orden **convect** para calcular la derivada material por el método de las características:

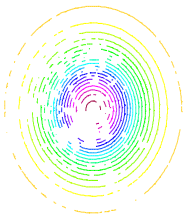
$$\partial_t \phi + \mathbf{u} \nabla \phi = 0, \quad \phi(x, t = 0) = \phi_0(x).$$

Ya hemos visto su aplicación para Navier-Stokes. Otro ejemplo es **convec-cion.edp**

```
real pi=4*atan(1.0),dt=0.01;
int i;
border a(t=0,2*pi){x=cos(t);y=sin(t);};
mesh th=buildmesh(a(300));
plot(th,wait=1);
fespace vh(th,P2);
vh u,u1,u2,phi;
func phi0=exp(-20*((x-0.1)^2+(y-0.3)^2));
u=phi0;
plot(u,wait=1,ps="conveccioninicial.eps");
u1=-y;
u2=x;
for (i=0; i<100;i++)
      {
        phi=convect([u1,u2],-dt,u);
        plot(phi,value=1,ps="conveccionfinal.eps");
        u=phi;
      };
```



Solución inicial para **conveccion.edp**



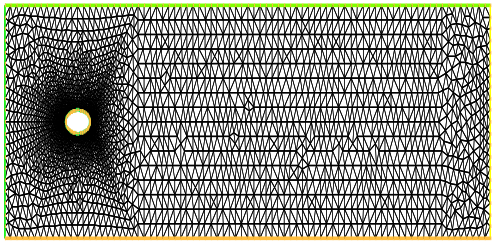
Solución final con **conveccion.edp**

En el siguiente ejemplo se resuelve Navier-Stokes en un canal con un obstáculo
Código **obstaculo.edp**:

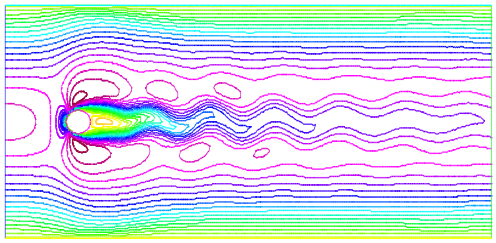
```
real pi=4*atan(1.0),w=20;
//
//                        c (3)
```

```
// y=4       ---------------------------------------
//          |    __                                 |
//   d (4)  |   |__| obstaculo                      | b (2)
//          |                                       |
// y=0       ---------------------------------------
//    x=0               a (1)                x=w
//
//
border a(t=0,w){x=t;y=0;label=1;};
border b(t=0,4){x=w;y=t;label=2;};
border c(t=w,0){x=t;y=4;label=3;};
border d(t=4,0){x=0;y=t;label=4;};
int n=20;
border circulo(t=0,2*pi){x=3+.5*cos(t);y=2+.2*sin(t);label=5;}

mesh th=buildmesh(a(4*n)+b(n)+c(4*n)+d(n)+circulo(-5*n));
plot(th,ps="obstacle.eps");

fespace Xh(th,P2);
fespace Mh(th,P1);
Xh u1,v1,u2,v2,up1=0,up2=0;
Mh p,q;
int i=0,nmax=500;
real nu=0.0025,dt=0.1,alpha=1/dt;
func f1=0;
func f2=0;
func g=-y*(y-4.0)/4.0;


problem  ns(u1,u2,p,v1,v2,q,init=i) =
          int2d(th)(
                    alpha*(u1*v1+u2*v2)+
                    nu*(dx(u1)*dx(v1)+dy(u1)*dy(v1) //  bilinear part
                   +dx(u2)*dx(v2)+dy(u2)*dy(v2))
                   +p*q*0.000000001
                   -p*dx(v1)-p*dy(v2)+
                    q*dx(u1)+q*dy(u2)
                   )    //  bilinear part
      +int2d(th)(-alpha*convect([up1,up2],-dt,up1)*v1
                 -alpha*convect([up1,up2],-dt,up2)*v2
```

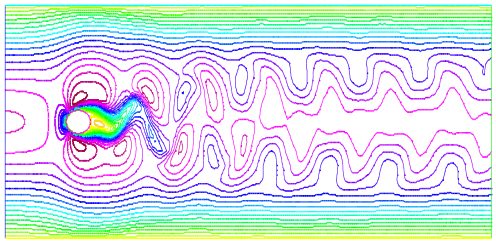
```
                -f1*v1-f2*v2) //right hand side
      +on(1,u1=0,u2=0)
      +on(3,u1=0,u2=0)
      +on(4,u1=g,u2=0)
      +on(5,u1=0,u2=0)
      ;                 // Dirichlet boundary condition
for (i=0;i<=nmax;i++)
{
ns;
up1=u1;
up2=u2;
cout<< "Valor de i = "<< i <<"\n";
plot(u1);
//plot(coef=0.5,value=0,cmm= "Velocidad [u1, u2]   ", [u1,u2]);
};
plot(u1,wait=1,ps="vel2-obst.eps");
plot(cmm= "presion   ", p,value=0,wait=1,ps="pres-obst.eps");
```



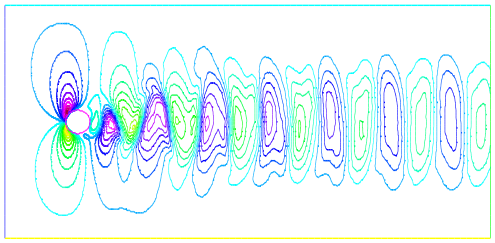
Malla usada con **obstaculo.edp**



Velocidad obtenida con **obstaculo.edp**

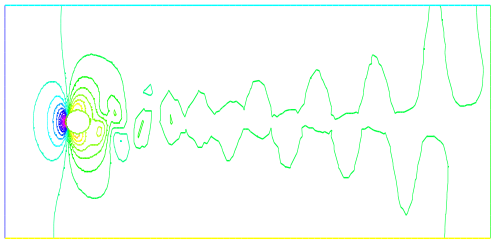


Velocidad horizontal con **obstaculo.edp**



Velocidad vertical con **obstaculo.edp**

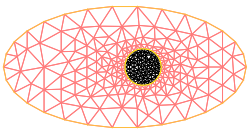
presion



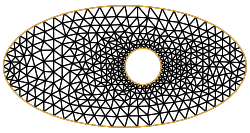
Presión obtenida con **obstaculo.edp**

7. Descomposición de dominios

Con **hueco.edp** podemos generar una malla en un círculo grande menos un círculo pequeño. Las triangulaciones se generan a la izquierda del sentido de recorrido de la curva. Por lo tanto, **un cambio de signo invierte el sentido de recorrido**.



Dos mallas complementarias generadas con **hueco.edp**

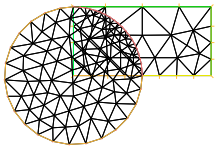


Mallas con hueco generada con **hueco.edp**

En este ejemplo solapamos dos mallas. Aquí las mallas se superponen pero no son coincidentes (el origen de esta geometría se remonta a los tiempos en los que resolver en un círculo o en un cuadrado era mas fácil).
Código **solapamiento.edp**:

```
real pi=4*atan(1.0);
int n=5;
border a(t=0,1){x=t;y=0;};
border a1(t=1,2){x=t;y=0;};
border b(t=0,1){x=2;y=t;};
border c(t=2,0){x=t;y=1;};
border d(t=1,0){x=0;y=t;};
border e(t=0,pi/2){x=cos(t);y=sin(t);};
border e1(t=pi/2,2*pi){x=cos(t);y=sin(t);};
mesh sh=buildmesh(a(n)+a1(n)+b(n)+c(n)+d(n));
```

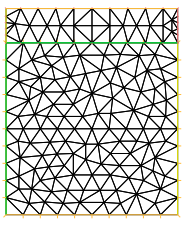
```
mesh SH=buildmesh(e(5*n)+e1(5*n));
plot(sh,SH,wait=1,ps="pito.eps");
```



Dos mallas superpuestas generadas con **solapamiento.edp**

En el siguiente ejemplo se consiguen dos mallas coincidentes sobre una interfaz.
Código **ddm-noover.edp**:

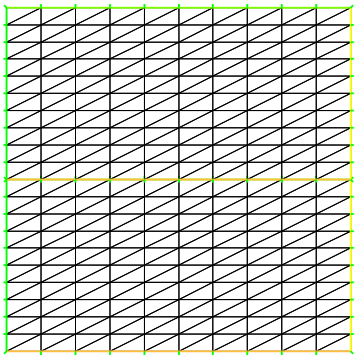
```
int n=5;
//
// Definicion del rectangulo [0,1]x[0,1]
//
border a(t=0,1){x=t;y=0;};
border b(t=0,1){x=1;y=t;};
border c(t=0,1){x=1-t;y=1;};//Interfaz
border d(t=0,1){x=0;y=1-t;};
//
// Definicion de un rectangulo sobre [0,1]x[0,1]
//
border c1(t=0,1){x=t;y=1;};//Interfaz
border e(t=0,0.2){x=1;y=1+t;};
border f(t=0,1){x=1-t;y=1.2;};
border g(t=0,0.2){x=0;y=1.2-t;};
mesh th=buildmesh(a(10*n)+b(10*n)+c(10*n)+d(10*n));
mesh TH=buildmesh(e(5*n)+f(10*n)+g(5*n)+c1(10*n));
plot(th,TH,wait=1,ps="ddm-noover.eps");
```



Dos mallas coincidentes en la interfaz generadas con **ddm-noover.edp**

Lo mismo, pero usando la orden **square** (mallas uniformes).
Código **ff13-ddm.edp** :

```
//
// Rectangulo [x0,x1]x[y0,y1] junto con el [x0,x1]x[y1,y2]
//
real x0=0,x1=1;
real y0=0.,y1=0.5,y2=1;
int n=10,m=10;
mesh Th=square(n,m,[x0+(x1-x0)*x,y0+(y1-y0)*y]);
mesh th=square(n,m,[x0+(x1-x0)*x,y1+(y2-y1)*y]);
plot(Th,th,wait=1);
```



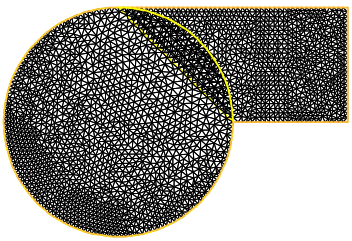
Dos mallas coincidentes en la interfaz generadas con **ff13-ddm.edp**

Finalmente se resuelve un problema de Poisson mediante el método de descomposición de dominios de Schwarz con solapamiento:
Código **ff24-swarz-ov.edp** :

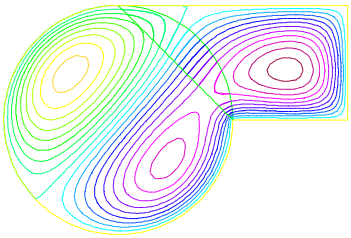
```
int inside = 2;
int outside = 1;
real pi=4*atan(1.0),velH1,velH1old=0,uH1,error,tol=1e-7;
int n=5;
border a(t=1,2){x=t;y=0;label=outside;};
border b(t=0,1){x=2;y=t;label=outside;};
border c(t=2,0){x=t;y=1;label=outside;};
border d(t=1,0){x=1-t;y=t;label=inside;};
border e(t=0,pi/2){x=cos(t);y=sin(t);label=inside;};
border e1(t=pi/2,2*pi){x=cos(t);y=sin(t);label=outside;};
mesh th=buildmesh(a(5*n)+b(5*n)+c(10*n)+d(5*n));
```

```
mesh TH=buildmesh(e(5*n)+e1(25*n));
plot(th,TH,wait=1);
fespace vh(th,P1);
fespace VH(TH,P1);
vh u=0,v,uold=0;
VH U,V,Uold=0;
func f=x-y;
int i=0;
//
// Definicion del problema en el circulo
//
problem PB(U,V,init=i,solver=Cholesky)=
       int2d(TH)(dx(U)*dx(V)+dy(U)*dy(V))
       +int2d(TH)(-f*V)
       +on(inside,U=u)
       +on(outside,U=0);
//
// Definicion del problema en el rectangulo
//
problem pb(u,v,init=i,solver=Cholesky)=
       int2d(th)(dx(u)*dx(v)+dy(u)*dy(v))
       +int2d(th)(-f*v)
       +on(inside,u=U)
       +on(outside,u=0);
//
//Ciclo de calculo
//
for (int j=0; j<10;j++)
{
PB;
pb;
//
// Calculo del error
//
vh uerr=u-uold;
VH Uerr=U-Uold;
uH1=int2d(th)(dx(uerr)^2+dy(uerr)^2);
uH1=uH1+int2d(TH)(dx(Uerr)^2+dy(Uerr)^2);
velH1=sqrt(uH1);
cout<< "-----------------------------------------------"<<"\n";
```

```
cout<< "---- Iteracion  j= "<<j <<"\n";
cout<< "Error H1 velocidad = "<< velH1 <<"\n";
cout<< "Errores consecutivos= "<< velH1old << " "<<velH1<<"\n";
cout<<" diferencia = "<<abs(velH1-velH1old) <<"\n";
cout<< "-------------------------------------------"<<"\n";
error=abs(velH1-velH1old);
velH1old=velH1;
if (error<tol)
{
cout<< "Velocidades H1 = "<< velH1 <<"\n";
break;};
uold=u;
Uold=U;
plot(U,u,wait=0);
cout<< "Iteracion = "<< j <<"\n";
};
```



Dos mallas solapadas generadas con **ff24-swarz-ov.edp**



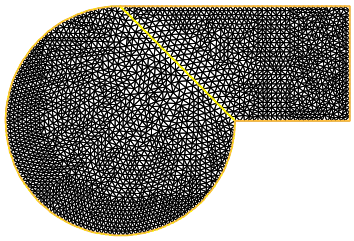
Solución obtenida con **ff24-swarz-ov.edp**

En el ejemplo siguiente se resuelve un problema de Poisson mediante un multiplicador para la forzar el salto a cero (aqui la **inf sup uniforme no se cumple**):
Código **ff25-swarz-nov.edp**:

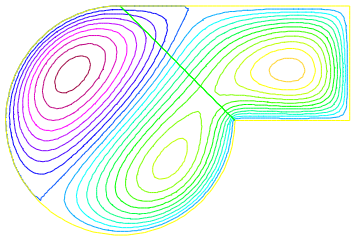
```
int inside = 2;
int outside = 1;
real pi=4*atan(1.0),tol=1e-6,velH1,velH1old=0,error,uH1;
int n=5;
border a(t=1,2){x=t;y=0;label=outside;};
border b(t=0,1){x=2;y=t;label=outside;};
border c(t=2,0){x=t;y=1;label=outside;};
border d(t=1,0){x=1-t;y=t;label=inside;};
border e(t=0,1){x=1-t;y=t;label=inside;};
border e1(t=pi/2,2*pi){x=cos(t);y=sin(t);label=outside;};
mesh th=buildmesh(a(5*n)+b(5*n)+c(10*n)+d(5*n));
mesh TH=buildmesh(e(5*n)+e1(25*n));
plot(TH,th,wait=1);
fespace vh(th,P1);
fespace VH(TH,P1);
vh u=0,v,uold=0;
VH U,V,Uold=0;
vh lambda=0;
func f=-x+y;
int i=0,nmax=100;
//
// Definicion del problema en el circulo
//
```

```
problem PB(U,V,init=i,solver=Cholesky)=
       int2d(TH)(dx(U)*dx(V)+dy(U)*dy(V))
       +int2d(TH)(-f*V)
       +int1d(TH,inside)(-lambda*V)
       +on(outside,U=0);
//
// Definicion del problema en el rectangulo
//
problem pb(u,v,init=i,solver=Cholesky)=
       int2d(th)(dx(u)*dx(v)+dy(u)*dy(v))
       +int2d(th)(-f*v)
       +int1d(th,inside)(+lambda*v)
       +on(outside,u=0);
//
//Ciclo de calculo
//
i=1;
for (int j=0; j<nmax;j++)
{
PB;
pb;
//
// Calculo del error
//
vh uerr=u-uold;
VH Uerr=U-Uold;
uH1=int2d(th,qft=qf2pT)(dx(uerr)^2+dy(uerr)^2);
uH1=uH1+int2d(TH,qft=qf2pT)(dx(Uerr)^2+dy(Uerr)^2);
velH1=sqrt(uH1);
cout<< "-----------------------------------------------"<<"\n";
cout<< "---- Iteracion  j= "<<j <<"\n";
cout<< "Error H1 velocidad = "<< velH1 <<"\n";
cout<< "Errores consecutivos= "<< velH1old << " "<<velH1<<"\n";
cout<<" diferencia = "<<abs(velH1-velH1old) <<"\n";
cout<< "-----------------------------------------------"<<"\n";
error=abs(velH1-velH1old);
velH1old=velH1;
if (error<tol)
{
cout<< "Velocidades H1 = "<< velH1 <<"\n";
```

```
break;};
uold=u;
Uold=U;
lambda=lambda+(u-U)/2;
plot(U,u,wait=0);
};
```



Dos mallas nosolapadas generadas con **ff25-swarz-nov.edp**



Solución obtenida con **ff24-swarz-nov.edp**

Observación 12 *Se puede observar como la función sobre la frontera lambda se define sobre todo Vh. Esto es porque no se pueden definir trazas y se usan los valores de U y u sobre la frontera para actualizar lambda.*

8. Problemas de evolución en tiempo

Podemos resolver la ecuación del calor

$$\partial_t u - \nu \Delta u \quad = \quad f, \quad (x,t) \in \Omega \times (0,T)$$

$$\begin{aligned} u(x,0) &= u_0, \quad x \in \Omega \\ u(x,t) &= g, \quad (x,t) \in \Gamma_D \times (0,T) \\ \partial_{\mathbf{n}} u(x,t) &= 0, \quad (x,t) \in \Gamma_N \times (0,T) \end{aligned}$$

siendo $\partial\Omega = \Gamma_N \cup \Gamma_D$ y usando elementos finitos en espacio y diferencias finitas en tiempo. Para la función

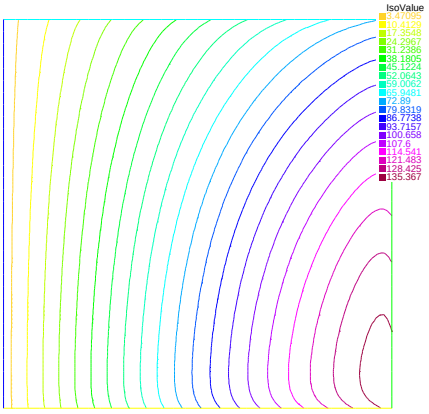
$$w(x,y,t) = \sin(x)\cos(y)e^t$$

el siguiente código **evolucion.edp** calcula w resolviendo

$$\begin{aligned} \partial_t u - \nu\Delta u &= f \qquad\qquad (x,y,t) \in \Omega \times (0,T) \\ u(x,y,0) &= \sin(x)\cos(y) \qquad (x,y) \in \Omega \\ u(x,y,t) &= \sin(x)\cos(y)e^t \qquad (x,y,t) \in \partial\Omega \times (0,T) \end{aligned}$$

siendo $f = (1+2\nu) * u$ y $\Omega = (0,1) \times (0,1)$ con $T = 3$.
Archivo **evolucion.edp**:

```
mesh Th=square(32,32);
fespace Vh(Th,P2);
Vh u,v,uu,f,g;
real dt=0.1, nu=0.001,error;
problem dcalor(u,v) =
   int2d(Th)( u*v+dt*nu*(dx(u)*dx(v) + dy(u)*dy(v)))
  +int2d(Th)(-uu*v-dt*f*v)
  + on(1,2,3,4,u=g);
real t=0;
uu=sin(x)*cos(y)*exp(t);
for (int m=0;m<=5/dt;m++)
{
t=t+dt;
f=(1+2*nu)*sin(x)*cos(y)*exp(t);
g=sin(x)*cos(y)*exp(t);
dcalor;
plot(u,wait=0,value=1);
error=sqrt(int2d(Th)((u-sin(x)*cos(y)*exp(t))^2));
 uu=u;
cout<< "t= "<< t << " L2-Error = " << error << endl;
 };
 plot(u,wait=1,value=1,ps="dcalor.eps");
```



Solucion final obtenida con **evolucion.edp**

Ejemplo aplicado: Buscamos la distribución de temperatura en una placa 3D de sección rectangular $\Omega = (0,6) \times (0,1)$. La placa recibe y mantiene una fuente de calor $u = u_0$ desde sus extremos laterales y está rodeada de aire a temperatura ambiente u_c. La temperatura cambia poco con la coordenada z y por lo tanto podemos considerar el problema como 2D.
Si ponemos $\Gamma_1 = \{y = 0, y = 1\}$ y $\Gamma_2 = \{x = 0, y = 6\}$, entonces se plantea como resolver la ecuación del calor

$$\begin{aligned} \partial_t u - \nabla \cdot (\kappa \nabla u) &= 0, \quad (x,t) \in \Omega \times (0,T) \\ u(x,y,0) &= u_0, \quad x \in \Omega \end{aligned}$$

con condiciones de contorno

$$\begin{aligned} \kappa \partial_{\mathbf{n}} u + \alpha(u - u_c) &= 0, \quad (x,t) \in \Gamma_1 \times (0,T) \\ u &= u_0 \quad (x,t) \in \Gamma_2 \times (0,T) \end{aligned}$$

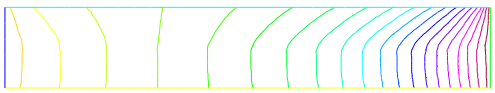
y en donde el coeficiente de difusión κ toma dos valores, $\kappa = 0.2$ sobre $y = 0.5$ y $\kappa = 2$ bajo $y = 0.5$.
Aplicamos el método de Euler implícito y tendremos

$$((u^n - u^{n-1})/dt, v)_\Omega + (k\nabla u^n, \nabla v)_\Omega + (\alpha(u^n - u_c), v)_\Gamma = 0$$

Archivo **termico.edp**:

```
func u0 =10+90*x/6;//temp. inicial
func k = 1.8*(y<0.5)+0.2;
real ue = 25;//temp. exterior
```

```
real alpha=0.25, T=10, dt=0.1;
mesh Th=square(30,5,[6*x,y]);//[0,6]x[0,1]
plot(Th,wait=0);
fespace Vh(Th,P2);
Vh u=u0,v,uold;
problem thermic(u,v)=
            int2d(Th)(u*v/dt+k*(dx(u)*dx(v) +dy(u) * dy(v)))
          + int1d(Th,1,3)(alpha*u*v)
               - int1d(Th,1,3)(alpha*ue*v)
               - int2d(Th)(uold*v/dt)
               + on(2,4,u=u0) ;
for(real t=0;t<T;t+=dt){
    uold=u;
     thermic;
    plot(u,wait=0);
};
plot(u,wait=1,value=1,ps="termico.eps");
```

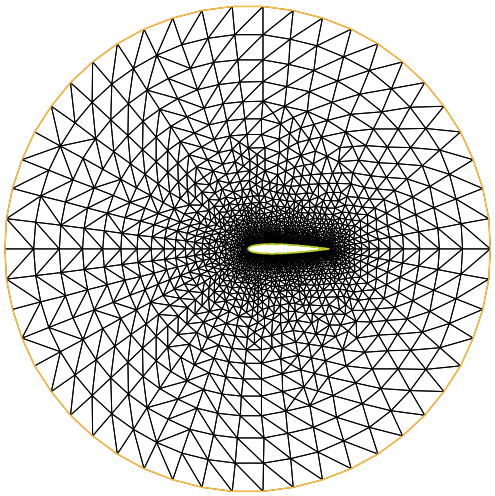


Solucion final obtenida con **termico.edp**

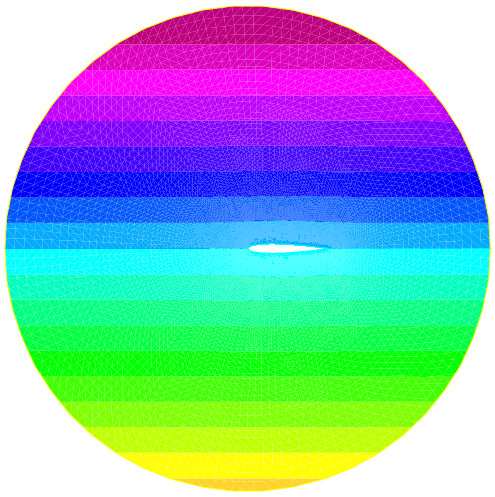
Finalmente, otro ejemplo lo constituye el calor en torno a un perfil de acero: Archivo **potencial.edp**

```
real S=99;
border C(t=0,2*pi) { x=3*cos(t); y=3*sin(t);}
border Splus(t=0,1){ x = t; y = 0.17735*sqrt(t)-0.075597*t
```

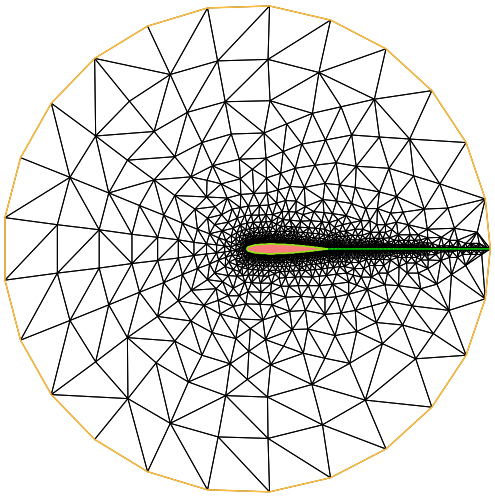
```
               - 0.212836*(t^2)+0.17363*(t^3)-0.06254*(t^4); label=S;}
border Sminus(t=1,0){ x =t; y= -(0.17735*sqrt(t)-0.075597*t
              -0.212836*(t^2)+0.17363*(t^3)-0.06254*(t^4)); label=S;}
mesh Th= buildmesh(C(50)+Splus(70)+Sminus(70));
plot(Th,wait=1,ps="mesh1.eps");
fespace Vh(Th,P2);
Vh psi,w;
solve potential(psi,w)=
      int2d(Th)(dx(psi)*dx(w)+dy(psi)*dy(w))+
      on(C,psi = y) +
        on(S,psi=0);
plot(psi,fill=1,wait=1,ps="potencial.eps");
border D(t=0,2){x=1+t;y=0;} // added to have a fine mesh at trail
mesh Sh = buildmesh(C(25)+Splus(-90)+Sminus(-90)+D(200));
plot(Sh,wait=1,ps="mesh2.eps");
fespace Wh(Sh,P1);
Wh v,vv;
int steel=Sh(0.5,0).region, air=Sh(-1,0).region;
fespace W0(Sh,P0);
W0 k=0.01*(region==air)+0.1*(region==steel);
W0 u1=dy(psi)*(region==air), u2=-dx(psi)*(region==air);
Wh vold = 120*(region==steel);
real dt=0.1, nbT=20;
int i;
problem thermic(v,vv,init=i,solver=LU)=
    int2d(Sh)(v*vv/dt+ k*(dx(v) * dx(vv) + dy(v) * dy(vv))
                + 10*(u1*dx(v)+u2*dy(v))*vv
             )
   -int2d(Sh)(vold*vv/dt);
for(i=0;i<nbT;i++){
v=vold;
 thermic;
plot(v, fill=1);
}
 plot(v,wait=1,fill=1,value=1,ps="calor.eps");
```



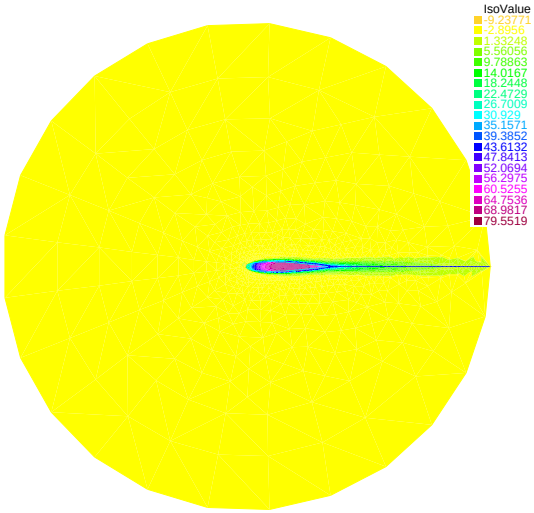
Red inicial obtenida con **potencial.edp**



Potencial en torno obtenido con **potencial.edp**



Malla total para **potencial.edp**

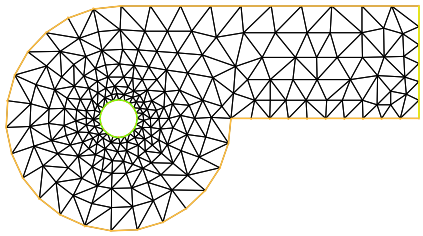


Calor obtenido con **potencial.edp**

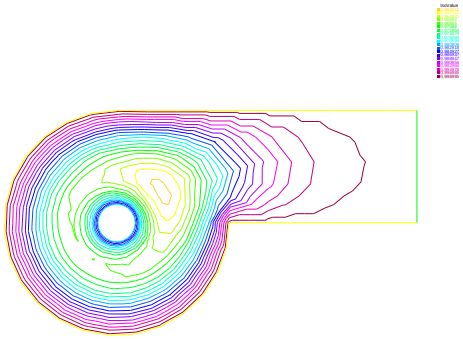
Otro problema es la ecuacion del calor con una turbina.
Archivo **turbina.edp**:

```
border a(t=pi/2,2*pi){x=1.5*cos(t);y=1.5*sin(t);label=1;};//exterior
border b(t=1.5,4){x=t;y=0;label=1;};//exterior
border c(t=0,1.5){x=4;y=t;label=2;};//salida turbina
border d(t=4,0){x=t;y=1.5;label=1;};//exterior
border e(t=2*pi,0){x=0.25*cos(t);y=0.25*sin(t);label=3;};//Interior
```

```
                                                          //de turbina
mesh Th= buildmesh (a(20)+b(10)+c(6)+d(10)+e(20));
plot(Th,wait=1,ps="malla-turbina.eps");
fespace Vh(Th,P1);
Vh u, v,uold,f,g,s,ue=1;
func u0 =0;
func k=1;
real N=100;
real  T=5*pi, dt=T/N;
uold=u0;
problem turbina(u,v)=
             int2d(Th)(u*v/dt +k*(dx(u)*dx(v)+dy(u)*dy(v))
                      )
                - int2d(Th)(uold*v/dt)
        +on(1,u=ue)+
        on(3,u=g);
for(real tt=0;tt<T;tt+=dt)
{
     g=abs(sin(tt+pi/2));
     turbina;
     uold=u;
     plot(u,wait=0,value=0);
};
plot(u,wait=1,fill=0,value=1,ps="u-turbina.eps");
```



Red inicial obtenida con **turbina.edp**

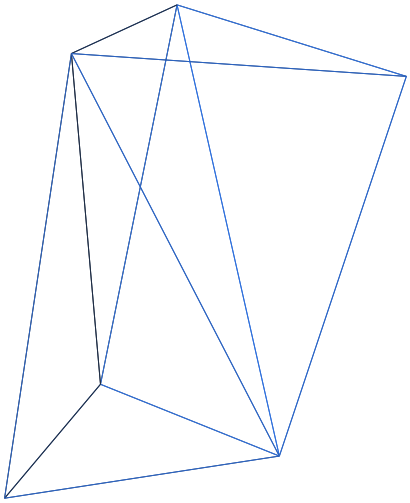


Solucion obtenida con **turbina.edp**

9. FreeFem 3d

Con este primer ejemplo vemos como generar una malla 3D a partir de una 2D sobre un triangulo con una sola capa:
Algoritmo **triangulo.edp:**

```
load "msh3"
load "medit"
border a(t=0,1){x=t;y=0;};
border b(t=0,1){x=1-t;y=t;};
border c(t=0,1){x=0;y=1-t;};
int nn=1;
mesh Th2=buildmesh(a(nn)+b(nn)+c(nn));
int ncapas=1;
real zmin=0,zmax=2;
mesh3 Th=buildlayers(Th2,ncapas,zbound=[zmin,zmax]);
medit(" Triangulo1capa ", Th);
savemesh(Th,"Th3d.mesh");
```



Triangulación básica de un triángulo con **triangulo.edp** y una sóla capa

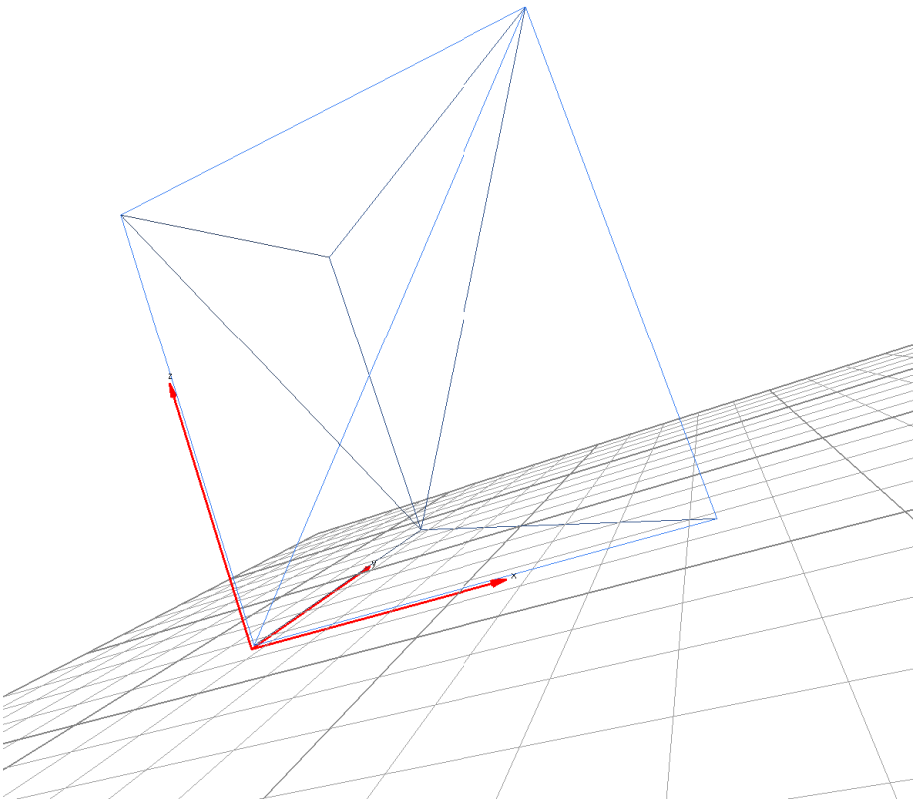
Se puede obervar que la capa está constituida por

- 6 vértices
- 8 triángulos
- 3 tetraedros

los parámetros zmin=0 y zmax=2 sirven para indicar la altura de la capa. Además, el archivo **Th3d.mesh** contiene toda la descripción de la malla

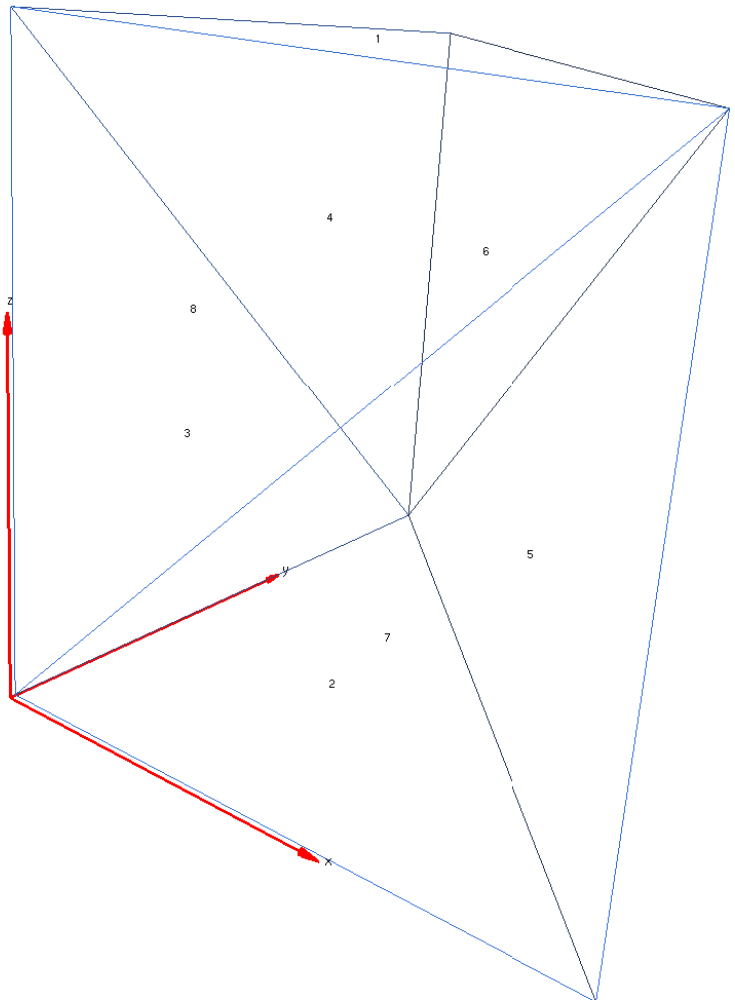
```
    MeshVersionFormatted 1
Dimension 3
Vertices
6
0 1 0 3
0 1 1 3
0 0 0 3
0 0 1 3
1 0 0 2
1 0 1 2
Tetrahedra
3
```

```
1 6 3 5 0
1 4 3 6 0
1 6 2 4 0
Triangles
8
2 4 6 0
5 3 1 1
1 3 4 2
4 2 1 2
5 1 6 3
2 6 1 3
3 5 6 4
6 4 3 4
End
```

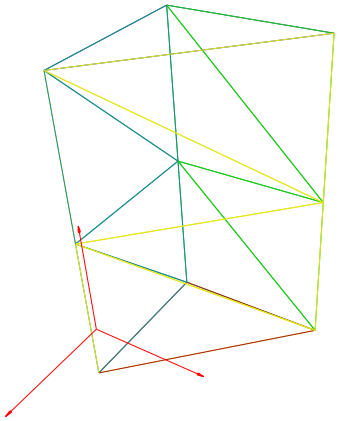


Otra vista donde se observan los tres tetraedros con **triangulo.edp**

Ahora se pueden ver las numeraciones de las caras



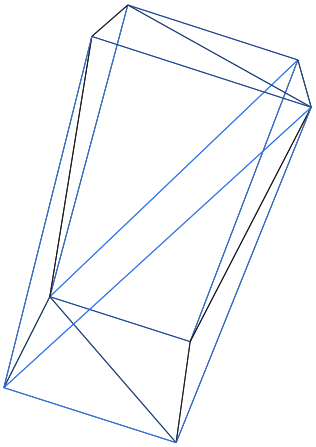
Numeracion de caras



Triangulación básica de un triángulo con **triangulo.edp** y dos capas

Cubo con sólo una capa y con la partición en triangulos de la base lo mas simple posible:
Algoritmo **cubo.edp:**

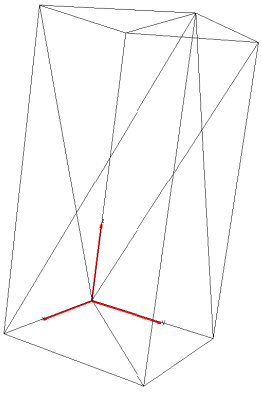
```
// build the mesh
// ---------------
load "msh3"
load "medit"
int nn=1;
mesh Th2=square(nn,nn);
int ncapas=1;
real zmin=0,zmax=2;
mesh3 Th=buildlayers(Th2,ncapas,zbound=[zmin,zmax]);
medit(" Sol ", Th,order=1);//  to see the sol in P1
```



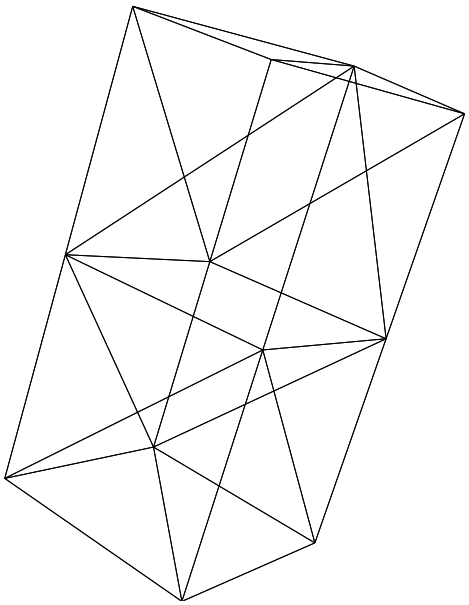
Triangulación básica de un cubo con **cubo.edp**

Se puede obervar que la capa está constituida por

- 8 vértices
- 12 triángulos
- 6 tetraedros



Triangulación de un cubo con **cubo.edp** y una capa

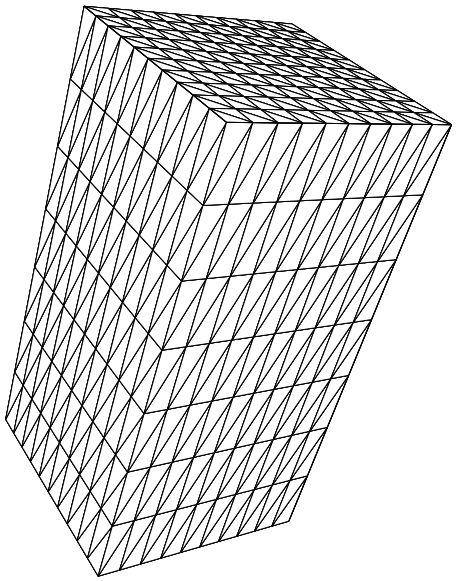


Triangulación de un cubo con **cubo.edp** y dos capas

En total se tienen ahora

- 12 vértices
- 20 triángulos
- 12 tetraedros

Un cubo ms grande se muestra en



Triangulación de un cubo con **cubo.edp** y siete capas.

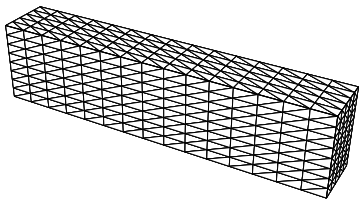
Uso de Tetgen

Otra forma de generar mallas 3D es mediante la funcion **tetgen**. Para eso hay que construir primero las superficies laterales del volumen 3D que se quiere partir en tetraedros y triangularlas de manera conforme con respecto a las interfaces. A continuación se crea el volumen y la partición.
Código **tetgencubo.edp:**

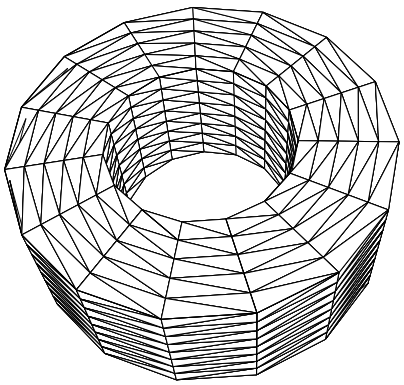
```
load "msh3"
load "tetgen"
load "medit"
//
// Rectangulo [1,2]x[0,2*pi]: tapas laterales en la direccion z
//                            en z=0 y z=1.5
//
real x0,x1,y0,y1;
x0=1.; x1=2.; y0=0.; y1=2*pi;
mesh Thsq1 = square(5,15,[x0+(x1-x0)*x,y0+(y1-y0)*y]);
func ZZ1min = 0;
func ZZ1max = 1.5;
func XX1 = x;
func YY1 = y;
```

```
mesh3 Th31h = movemesh2D3Dsurf(Thsq1,transfo=[XX1,YY1,ZZ1max]);
mesh3 Th31b = movemesh2D3Dsurf(Thsq1,transfo=[XX1,YY1,ZZ1min]);
//
//  Rectangulo [1,2]x[0,1.5]: tapas laterales en la direccion y
//                            en y=0 e y=2*pi
//
x0=1.; x1=2.; y0=0.; y1=1.5;
mesh Thsq2 = square(5,8,[x0+(x1-x0)*x,y0+(y1-y0)*y]);
func ZZ2 = y;
func XX2 = x;
func YY2min = 0.;
func YY2max = 2*pi;
mesh3 Th32h = movemesh2D3Dsurf(Thsq2,transfo=[XX2,YY2max,ZZ2]);
mesh3 Th32b = movemesh2D3Dsurf(Thsq2,transfo=[XX2,YY2min,ZZ2]);
//
//  Rectangulo [0,2*pi]x[0,1.5]: tapas laterales en la direccion x
//                            en x=1 x=2
//
x0=0.; x1=2*pi; y0=0.; y1=1.5;
mesh Thsq3 = square(15,8,[x0+(x1-x0)*x,y0+(y1-y0)*y]);
func XX3min = 1.;
func XX3max = 2.;
func YY3 = x;
func ZZ3 = y;
mesh3 Th33h = movemesh2D3Dsurf(Thsq3,transfo=[XX3max,YY3,ZZ3]);
mesh3 Th33b = movemesh2D3Dsurf(Thsq3,transfo=[XX3min,YY3,ZZ3]);
//
// Se construye la malla de la superficie del cubo sumando las mallas
//
mesh3 Th33 = Th31h+Th31b+Th32h+Th32b+Th33h+Th33b;
//
// Se graba la malla
//
savemesh(Th33,"Th33.mesh");
//
// Se construye finalmente la malla con tetraedros de un cubo paralelo
// al eje y usando TetGen
//
real[int] domain =[1.5,pi,0.75,145,0.0025];
mesh3 Thfinal =tetg(Th33,switch="paAAQY",nbofregions=1,regionlist=domain);
```

```
savemesh(Thfinal,"Thfinal.mesh");
medit("mallafinal",Thfinal);
//
// Construimos ahora una malla de la mitad de una placa cilindrica
// de radio interior 1., radio exterior 2 y altura 1.5
func mv2x = x*cos(y);
func mv2y = x*sin(y);
func mv2z = z;
mesh3 Thmv2 = movemesh3D(Thfinal, transfo=[mv2x,mv2y,mv2z]);
savemesh(Thmv2,"halfcylindricalshell.mesh");
medit("mallafinalmodificada",Thmv2);
```



Partición de un cubo con **tetgencubo.edp** usando tetgen



Placa cilindrica usando tetgen

Uso de buildlayermesh

Se obtiene una malla 3D por capas a partir de una 2D. Tenemos

$$\Omega_{3D} = \Omega_{2D} \times [zmin, zmax]$$

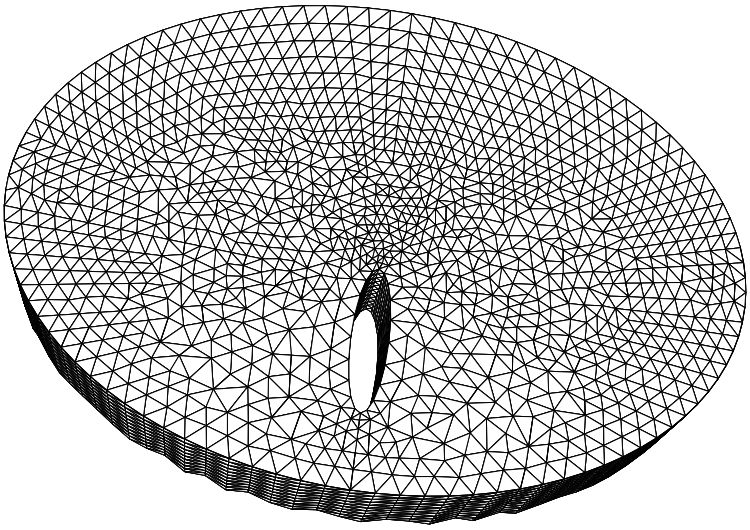
donde *zmin* y *zmax* son funciones definidas sobre Ω_{2D} que determinan las capas superior e inferior del dominio Ω_{3D}. En general, los elementos que se usan son **prismas**. Los argumentos principales son una malla 2D y el número de capas M.

Observación 13 *Cada superficie queda* ***numerada por defecto*** *con el mismo número que la arista sobre la que descansa. Siendo 0 la superficie de base que se extiende por capas y 1 la última capa. Siendo la región de base el 0, entonces*

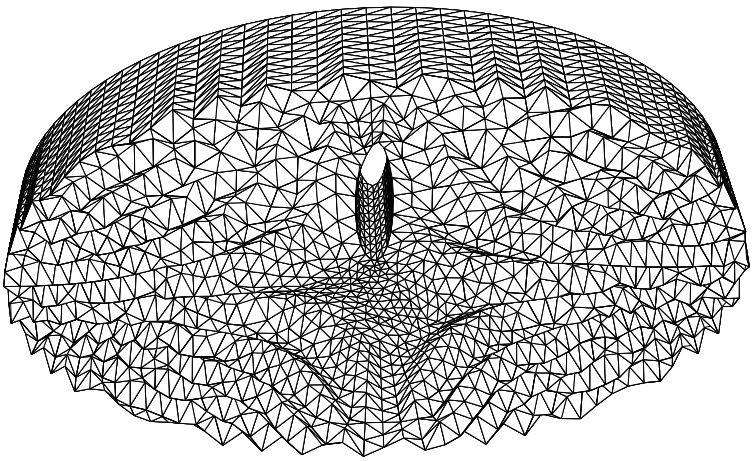
- `rup=[0,2]` *pone la cara superior con la etiqueta 2*
- `rdlow=[0,1]` *pone la cara inferior con la etiqueta 1*
- `rmid=[1,1,2,1,3,1,4,1]` *pone las caras laterales, con etiquetas 1,2,3,4, todas con con la etiqueta 1.*

Código **buildlayermesh.edp**:

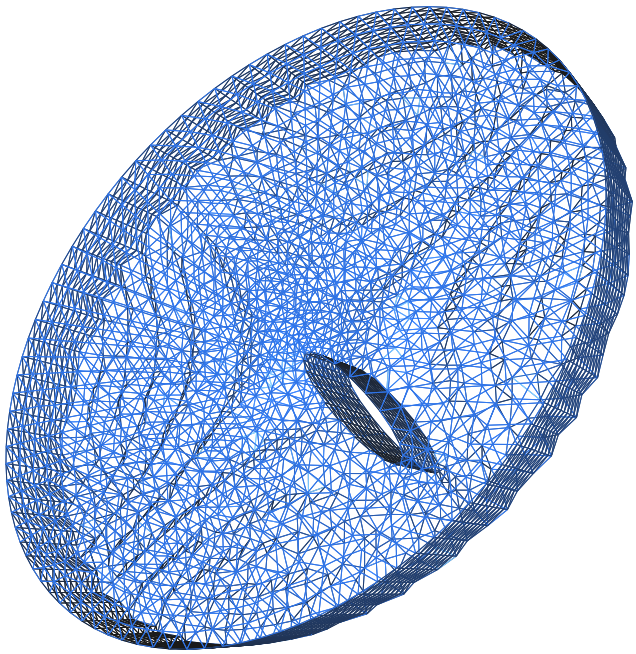
```
load "msh3"
load "medit"
border C0(t=0,2*pi){ x=7*cos(t); y=10*sin(t);};
border C1(t=0,2*pi){ x=3+2*cos(t); y=0.5*sin(t);};
mesh Th=buildmesh(C0(100)+C1(-20));
func zmin=-1+sin(x*y)/3;
func zmax=2;
int MaxLayer=10;
mesh3 Th3=buildlayers(Th, MaxLayer,zbound=[zmin,zmax]);
savemesh(Th3,"region1hole.mesh");
medit("regionconbujero",Th3);
```



Distintas vistas de la region generada con **buildlayermesh.edp**



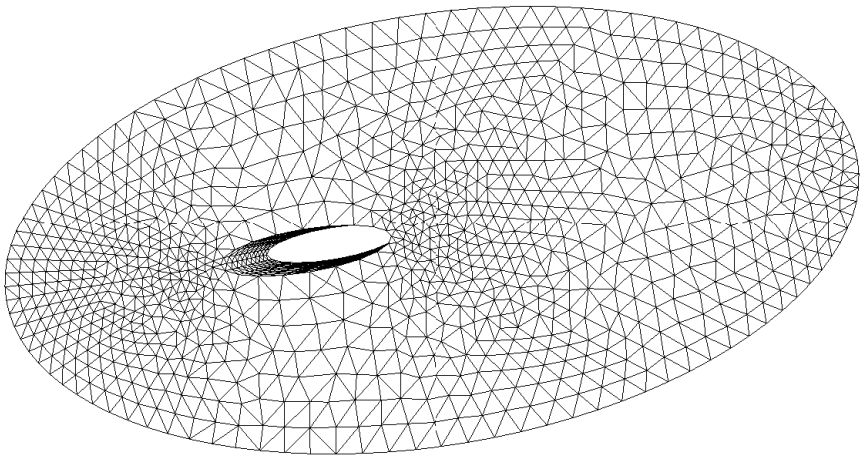
Distintas vistas de la region generada con **buildlayermesh.edp**



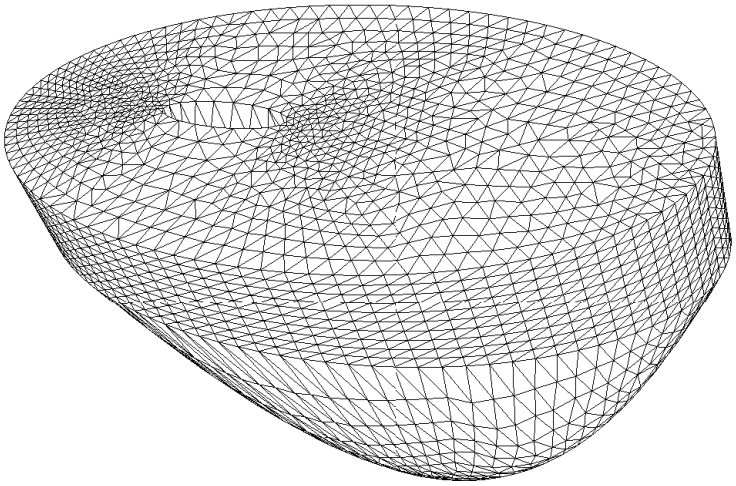
Distintas vistas de la region generada con **buildlayermesh.edp**

Mismo ejemplo con distintos máximos y mínimos para la malla 3D

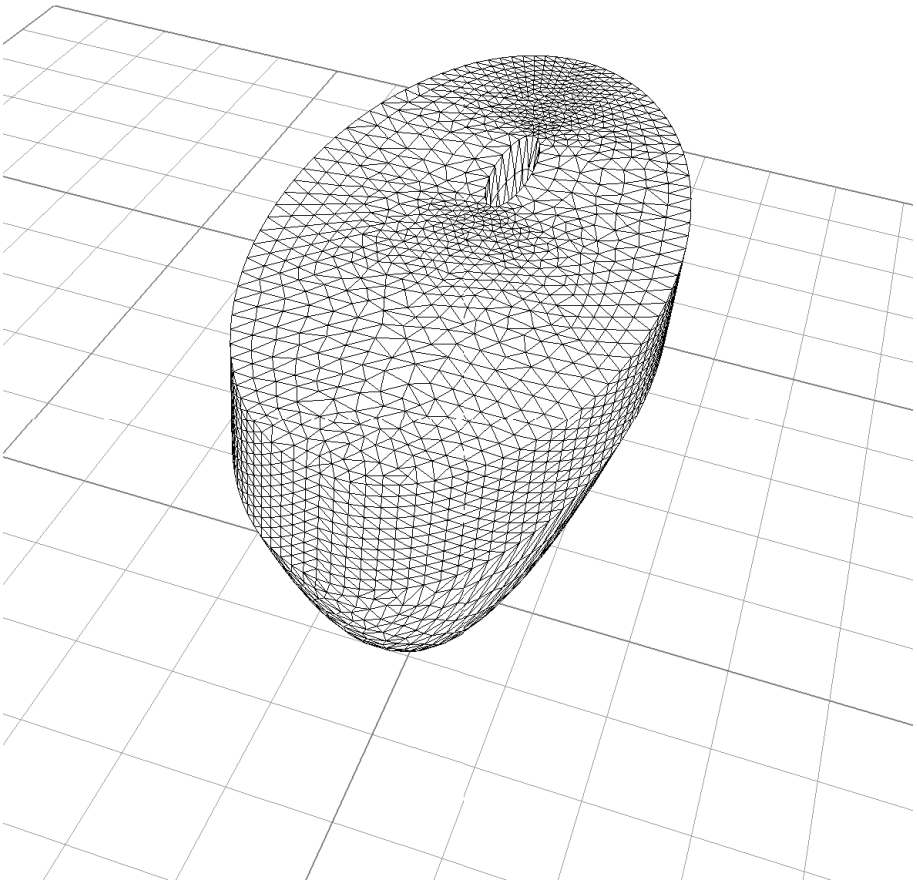
```
load "msh3"
load "medit"
border C0(t=0,2*pi){ x=7*cos(t); y=10*sin(t);};
border C1(t=0,2*pi){ x=3+2*cos(t); y=0.5*sin(t);};
mesh Th=buildmesh(C0(100)+C1(-20));
func zmin=10*((x/10)^2+(y/5)^2-1.2);
func zmax=1;
int MaxLayer=10;
mesh3 Th3=buildlayers(Th, MaxLayer,zbound=[zmin,zmax]);
savemesh(Th3,"region1hole.mesh");
medit("regionconbujero",Th3);
```



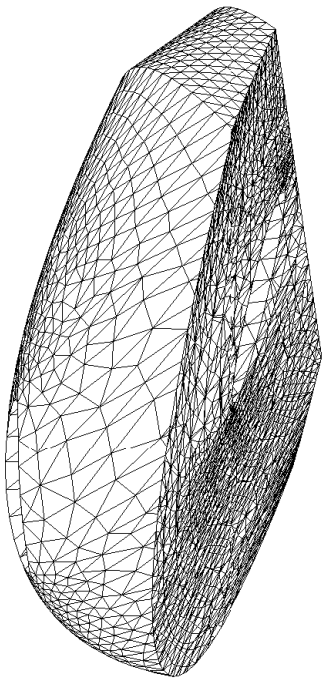
Distintas vistas de la region generada con **buildlayermesh.edp**



Distintas vistas de la region generada con **buildlayermesh.edp**



Distintas vistas de la region generada con **buildlayermesh.edp**

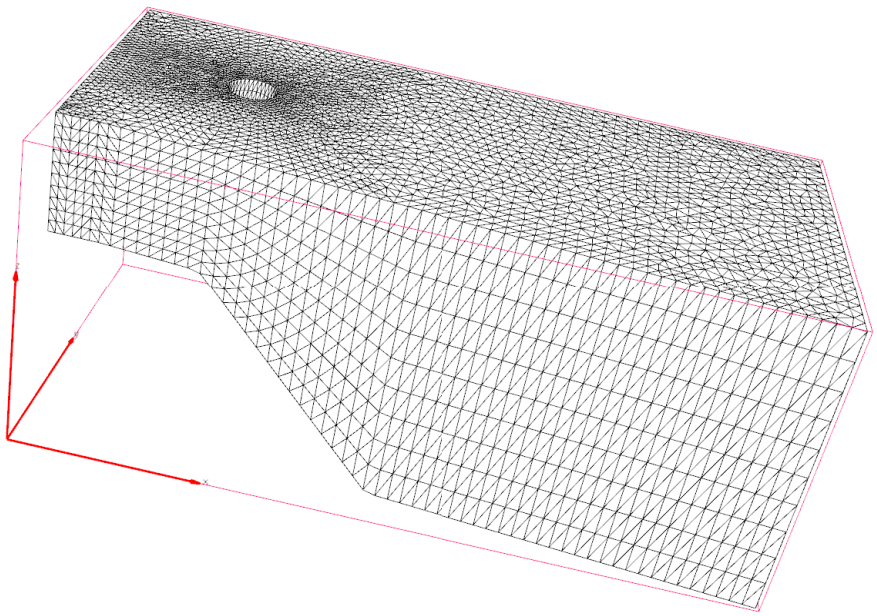


Distintas vistas de la region generada con **buildlayermesh.edp**

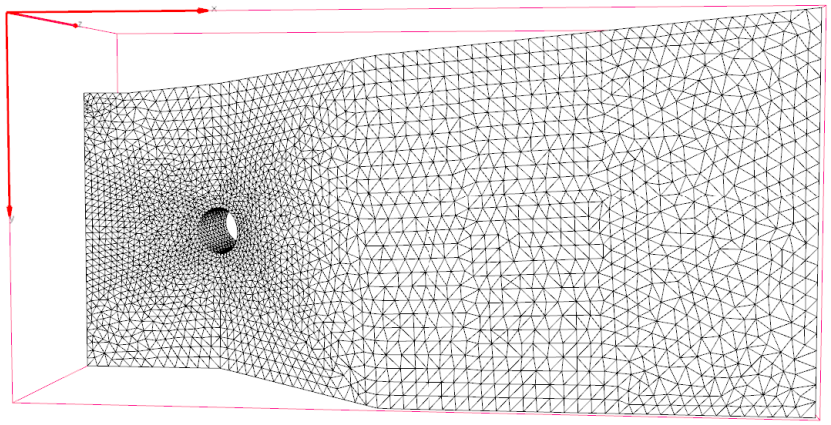
Mismo ejemplo con distintos máximos y mínimos para la malla 3D

```
load "msh3"
load "medit"
real L=6;
verbosity=0;
border aa(t=0,1){x=t; y=0 ;};
border bb(t=0,14){x=1+t; y= - 0.1*t ;};
border cc(t=-1.4,L){x=15; y=t ;};
border dd(t=15,0){x= t ; y = L;};
border ee(t=L,0.5){ x=0; y=t ;};
border ff(t=0.5,0){ x=0; y=t ;};
border C1(t=0,2*pi){ x=3+0.5*cos(t); y=3+0.5*sin(t);};
```

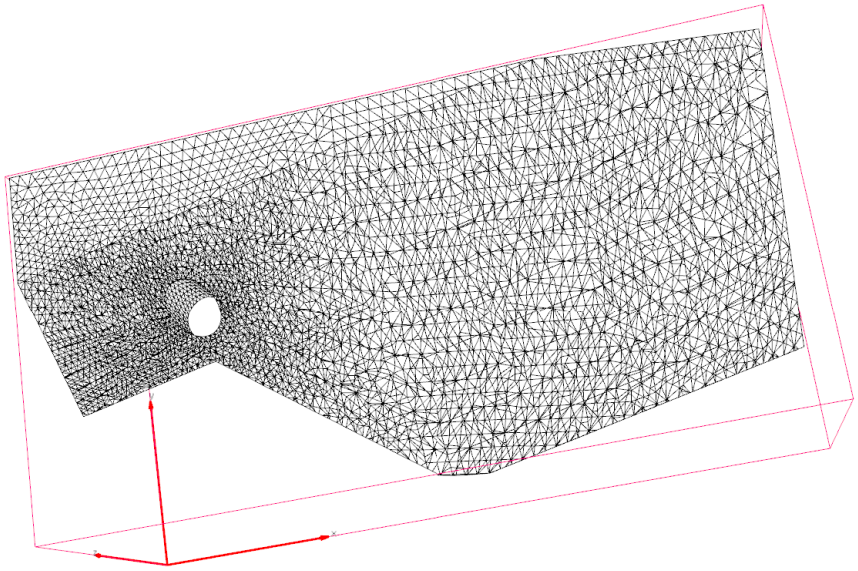
```
int n=6;
mesh Th2d=buildmesh(aa(n)+bb(9*n) + cc(4*n) + dd(10*n)+ee(6*n)
                                    + ff(n)+C1(-6*n));
//plot(Th2d,wait=1);
func zmin=-1.5*(x<3)+(+1.5-x)*(x>3)*(x<7)-5.5*(x>7);
func zmax=1;
int MaxLayer=10;
mesh3 Th3=buildlayers(Th, MaxLayer,zbound=[zmin,zmax]);
savemesh(Th3,"region1hole.mesh");
medit("regionconbujero",Th3);
```



Distintas vistas de la region generada con **buildlayermesh.edp**



Distintas vistas de la region generada con **buildlayermesh.edp**

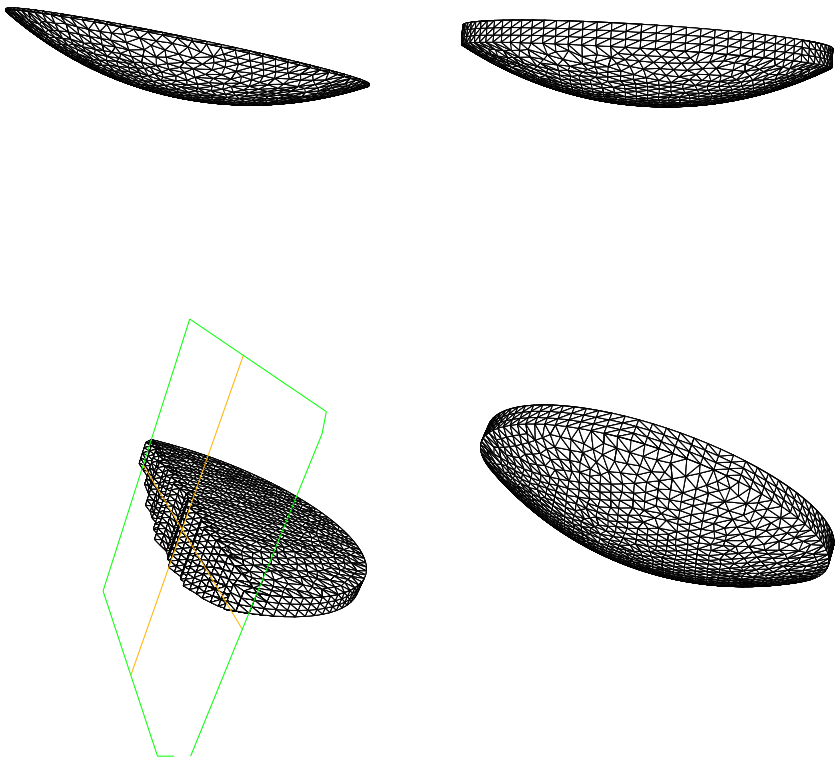


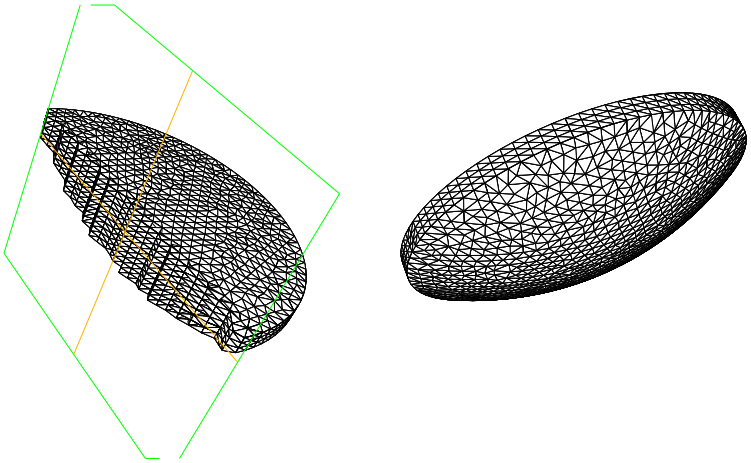
Distintas vistas de la region generada con **buildlayermesh.edp**

Otra forma de obtener profundidad y orilla variable es posible:
Código **orilla.edp**

```
load "msh3"
load "medit"
//
// Orilla con grosor eps
//
real eps=0.1;
//
int nn=10;
border cc(t=0,2*pi){x=cos(t);y=sin(t);};
mesh Th2= buildmesh(cc(100));
```

```
//
// zmin define el fondo
//
func zmin= 2-sqrt(4-(x*x+y*y));
//
// zmax define la superficie
//
func zmax= eps+ 2-sqrt(3.);
mesh3 Th=buildlayers(Th2,nn,coef= max((zmax-zmin)/zmax,1./(nn*2)),
zbound=[zmin,zmax]);
medit("orilla",Th);
```





Distintas vistas de la region generada con **orilla.edp** para un grosor de 0.01 y de 0.1 de la orilla.

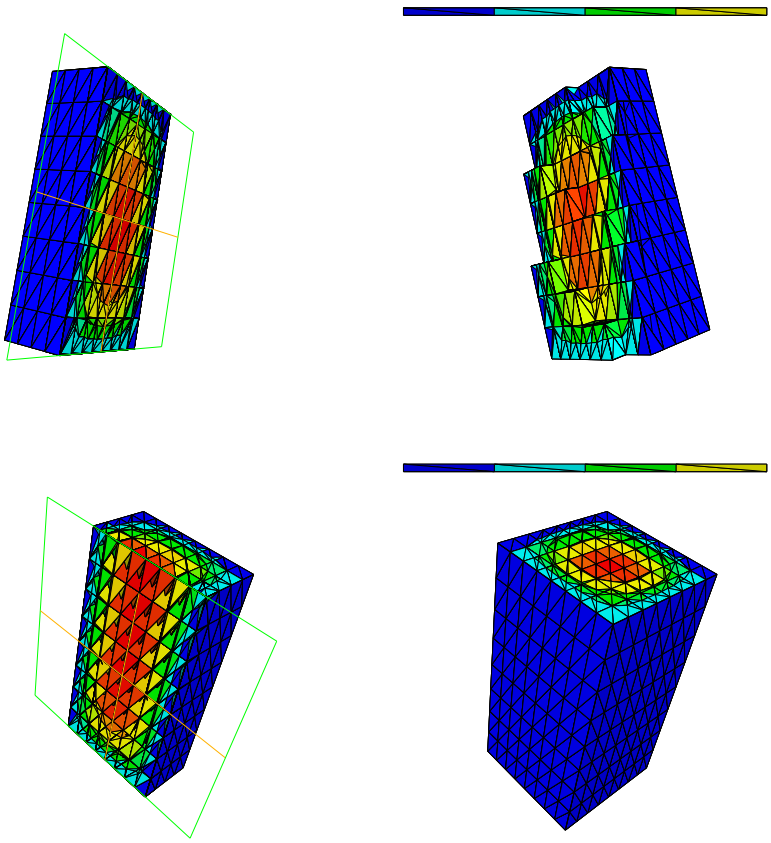
Se resuelve ahora el Laplaciano en un cubo. Se debe de observar que por defecto se etiquetan las caras laterales como sigue:

- Cara superior... 0
- Cara inferior... 1
- Cara laterales... 2,3,4,5

Código **Laplaciano3d.edp**:

```
//
//Construccion de la malla
//
load "msh3"
load "medit"
int nn=8;
mesh Th2=square(nn,nn);
//
real zmin=0,zmax=2;
mesh3 Th=buildlayers(Th2,nn, zbound=[zmin,zmax]);
//
func f=1. ;
fespace Vh(Th,P23d);
Vh u,v;
//
macro Grad3(u) [dx(u),dy(u),dz(u)]  // EOM
```

```
//
problem Lap3d(u,v,solver=CG) =
          int3d(Th)(Grad3(v)' *Grad3(u))
          - int3d(Th)(f*v)
  + on(1,u=0);
Lap3d;
medit(" Laplaciano ", Th, u ,order=1); // para ver la solucion en P1
```

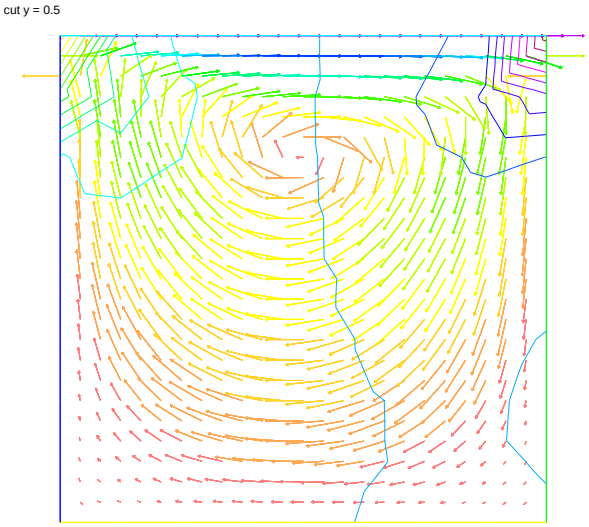


Distintas vistas de la solucion $-\Delta u = 1$ con $u = 0$ en partes de la frontera.

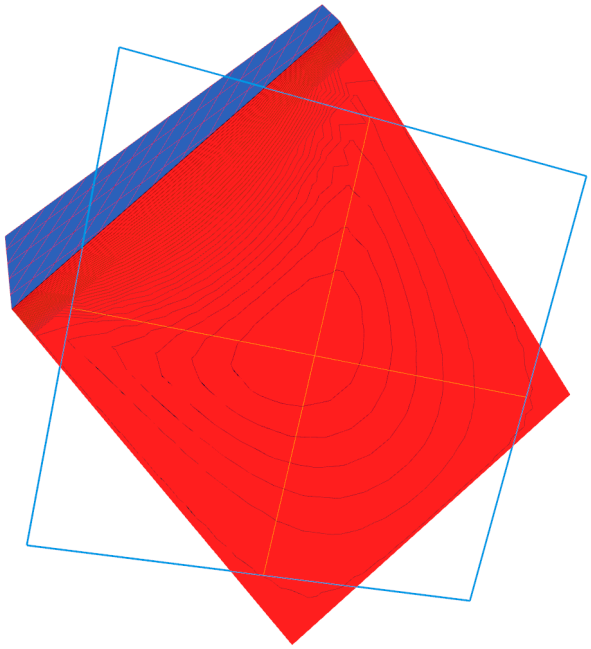
Se resuelve ahora el **problema de Stokes** en un cubo unidad con los datos del **problema de la cavidad**. Se muestran cortes transversales de la solución y para $nn = 15$ el método directo UMFPACK consume toda la memoria y se debe usar GMRES.
Código **Stokes3dcavidad.edp**:

```
load "msh3";
load "medit";
//
// Malla base
//
int nn=15;
mesh Th2=square(nn,nn);
//
// Espacio de elementos finitos para los cortes transversales
//
fespace Vh2(Th2,P2);
Vh2 ux,uz,p2;
//
// Malla 3D por capas
//
real zmin=0,zmax=1;
mesh3 Th=buildlayers(Th2,nn, zbound=[zmin,zmax]);
//
// Espacio de elementos finitos para la solucion de Stokes
//
fespace VVh(Th,[P23d,P23d,P23d,P13d]);

VVh [u1,u2,u3,p];
VVh [v1,v2,v3,q];

macro Grad(u) [dx(u),dy(u),dz(u)]// EOM
macro div(u1,u2,u3) (dx(u1)+dy(u2)+dz(u3)) //EOM
real mu=1,yy;
  varf vStokes([u1,u2,u3,p],[v1,v2,v3,q]) =
  int3d(Th)( mu*(Grad(u1)'*Grad(v1)+Grad(u2)'*Grad(v2)+Grad(u3)'*Grad(v3))
             - div(u1,u2,u3)*q - div(v1,v2,v3)*p
             + 1e-10*q*p
             )
              +on(1,2,3,4,5,u1=0,u2=0,u3=0)
              + on(0,u1=1.,u2=0,u3=0)
                ;
cout << "Construccion de la matriz de rigidez " << endl;
matrix A=vStokes(VVh,VVh);
cout << "le asociamos un resolutor lineal " << endl;
set(A,solver=GMRES);
```

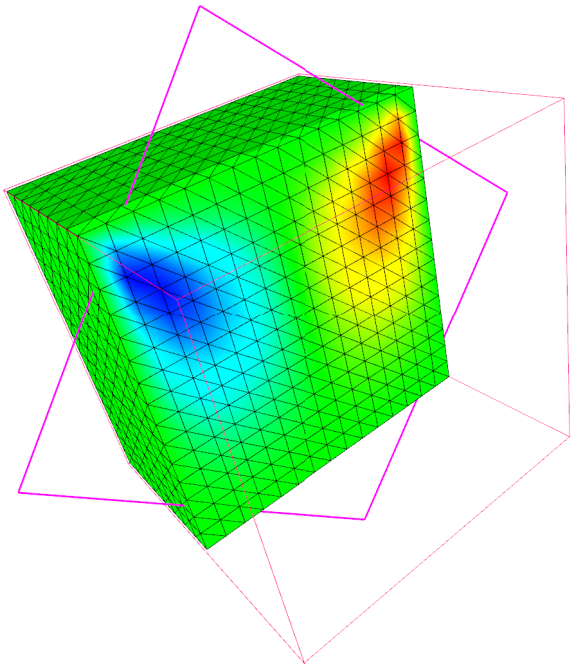
```
cout << "Construimos el termino independiente   " << endl;
real[int] b= vStokes(0,VVh);
cout << "Invertimos el sistema lineal ....." << endl;
p[]= A^-1 * b;
cout << "Hecho" << endl;
cout <<" Cortes seccionales para distintos valores de y ....." << endl;
for(int i=1;i<10;i++)
{
 yy=i/10.;
 ux= u1(x,yy,y);
 uz= u3(x,yy,y);
 p2= p(x,yy,y);
 plot([ux,uz],p2,cmm=" cut y = "+yy,wait= 1);
       if (i==5)
      {plot([ux,uz],p2,cmm=" cut y = "+yy,wait= 1,ps="Stokes3d.ps");};
};
medit("cavidad3d",Th,p,order=1);
medit("cavidad3d",Th,u3,order=1);
medit("cavidad3d",Th,u2,order=1);
medit("cavidad3d",Th,u1,order=1);
```



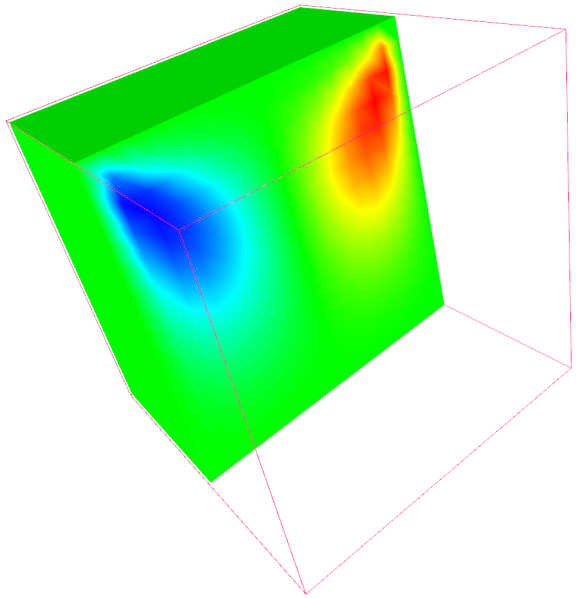
Corte del problema de la cavidad para Stokes con $\mu = 1$.



Corte del problema de la cavidad para Stokes con $\mu = 1$.



Corte del problema de la cavidad para Stokes con $\mu = 1$.



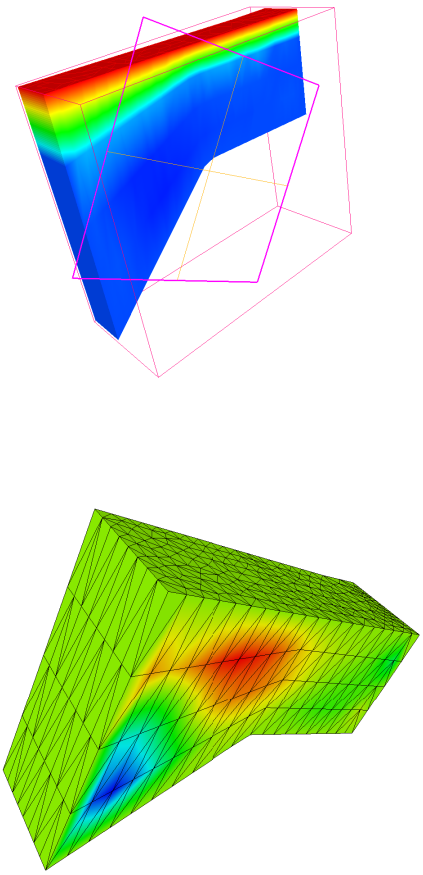
Corte del problema de la cavidad para Stokes con $\mu = 1$.

Otro ejemplo con Navier-Stokes, una geometría extraña y viscosidad anisótropa. Código **tanke3d.edp**:

```
load "msh3"
load "medit"
verbosity=0;
real nux=0.25,nuy=0.25,nuz=0.0025,dt=0.1;
real alpha=1./dt;
real L=2,w=5;
real a=1;
verbosity=0;
border aa(t=0,1){x=t; y=0 ;};
border bb(t=0,w){x=1+t; y= - 0.1*t ;};
border cc(t=-0.1*w,L){x=w+1; y=t ;};
border dd(t=w+1,0){x= t ; y = L;};
border ee(t=L,0){ x=0; y=t ;};
//
```

```
//border C1(t=0,2*pi){ x=2+0.5*cos(t); y=2+0.5*sin(t);};
//
int n=2,i;
mesh Th2d=buildmesh(aa(n)+bb(9*n) + cc(4*n) + dd(10*n)+ee(7*n));// +C1(-6*n));
//plot(Th2d,wait=1);
func zmin=-1.5*(x<3)+(+1.5-x)*(x>3)*(x<7)-5.5*(x>7);
func zmax=1;
int MaxLayer=4;
mesh3 Th3d=buildlayers(Th2d, MaxLayer,zbound=[zmin,zmax]);
fespace VVh(Th3d,[P23d,P23d,P23d,P13d]);
fespace MMh(Th3d,P13d);
VVh [up1,up2,up3,pp];
VVh [u1,u2,u3,p];
VVh [v1,v2,v3,q];
macro Grad(u) [dx(u),dy(u),dz(u)]// EOM
macro div(u1,u2,u3) (dx(u1)+dy(u2)+dz(u3)) //EOM
solve Stokes([up1,up2,up3,pp],[v1,v2,v3,q],solver=UMFPACK) =
  int3d(Th3d)(nux*(Grad(up1)'*Grad(v1))+nuy*(Grad(up2)'*Grad(v2)) +
                   nuz*(Grad(up3)'*Grad(v3))
              +a*( -up2*v1+up1*v2)
                -div(up1,up2,up3)*q - div(v1,v2,v3)*pp
              + 0.000000001*pp*q
             )
  + on(1,2,4,5,6,7,up1=0,up2=0,up3=0)
  + on(0,up1=1,up2=0,up3=0)
  ;
 varf vNS([u1,u2,u3,p],[v1,v2,v3,q],init=i) =
  int3d(Th3d)( alpha*(u1*v1+u2*v2+u3*v3) +
            nux*(Grad(u1)'*Grad(v1))+nuy*(Grad(u2)'*Grad(v2))
                +nuz*(Grad(u3)'*Grad(v3))
               +a*( -u2*v1+u1*v2)
                  -div(u1,u2,u3)*q - div(v1,v2,v3)*p
                  + 0.000000001*p*q
              )
 +  int3d(Th3d)(-alpha*(convect([up1,up2,up3],-dt,up1)*v1  +
                         convect([up1,up2,up3],-dt,up2)*v2  +
                         convect([up1,up2,up3],-dt,up3)*v3
                       )
               )
  + on(1,2,4,5,6,7,u1=0,u2=0,u3=0)
```

```
  + on(0,u1=1,u2=0,u3=0)
    ;
cout << " Construimos A" << endl;
matrix  A = vNS(VVh,VVh);
cout << " Asociamos un solver a  A" << endl;
set(A,solver=UMFPACK);
real t=0;
for(i=0;i<21;++i)
  {t += dt;
 cout << " Calculamos termino independiente " << endl;
real[int]   b=vNS(0,VVh);
cout << " iteration " << i << " t = " << t << endl;
cout << "Invertimos sistema lineal ....  " << endl;
 u1[]= A^-1*b;
 up1[]=u1[];
  };
{
medit("tank3d",Th3d,p,order=1);
medit("tank3d",Th3d,u3,order=1);
medit("tank3d",Th3d,u2,order=1);
medit("tank3d",Th3d,u1,order=1);
};
```



Corte Navier-Stokes.