FreeFem++ Lessons 5-8

F. Hecht and I. Danaila

Laboratoire Jacques-Louis Lions
Université Pierre et Marie Curie
Paris, France

http://www.freefem.org mailto:frederic.hecht@upmc.fr

Outline

Outline

Linear Lamé equation and Hooke's Law

Let us consider a beam and with transverse section Ω, subject to a force $\mathbf{f}$, perpendicular to the axis. The components along x and y of the displacement $\mathbf{u}(x)$ in the section Ω are governed by the Lamé's system of linear equations.
Remark: we do not use this equation because the associated variational form does not give the correct boundary conditions! We simply use the equilibrium between efforts and constraints:

$$-\nabla.(\sigma) = \rho \boldsymbol{f} \quad \text{in} \quad \Omega,$$

where the constraint tensor $\sigma(\boldsymbol{u})$ is related to deformations using the Hooke's law:

$$\sigma(\boldsymbol{u}) = \lambda tr(\varepsilon(\boldsymbol{u}))I + 2\mu\varepsilon(\boldsymbol{u}).$$

λ, μ are the physical Lamé coefficients and the strain tensor is $\varepsilon(\boldsymbol{u}) = \frac{1}{2}(\nabla\boldsymbol{u} + {}^t\nabla\boldsymbol{u})$.
The corresponding variational (weak) form is:

$$\int_\Omega \sigma(\boldsymbol{u}) : \varepsilon(\boldsymbol{v})\, dx - \int_\Omega \boldsymbol{f}\boldsymbol{v}\, dx - \int_{\partial\Omega} (\sigma(\boldsymbol{u}).\boldsymbol{n})\, \boldsymbol{v} = 0, \qquad \boldsymbol{a} : \boldsymbol{b} = \sum_{i,j} a_{ij} b_{ij}.$$

Finally, the variational form can be written as :

$$\int_\Omega \lambda\, \nabla.\boldsymbol{v}\nabla.\boldsymbol{u} + 2\mu\, \varepsilon(\boldsymbol{u}) : \varepsilon(\boldsymbol{v})\, dx - \int_\Omega \boldsymbol{f}\boldsymbol{v}\, dx - \int_{\partial\Omega} (\sigma(\boldsymbol{u}).\boldsymbol{n})\, \boldsymbol{v} = 0.$$

Outline

Static Linear Lamé equations: weak formulation

Let $\Omega \subset \mathbb{R}^d$ be a domain with a partition of $\partial\Omega = \Gamma_d \cup \Gamma_n$.
Find $\boldsymbol{u}$, the displacement field, such that:

$$-\nabla.\sigma(\boldsymbol{u}) = \rho\, \boldsymbol{f} \text{ in } \Omega, \quad \boldsymbol{u} = \mathbf{0} \text{ on } \Gamma_d, \quad \sigma(\boldsymbol{u}).\boldsymbol{n} = 0 \text{ on } \Gamma_n, \tag{1}$$

where $\sigma(\boldsymbol{u}) = \mathcal{A}\varepsilon(\boldsymbol{u})$, with $\mathcal{A}$ a linear positive operator (symmetric $d \times d$ matrix) corresponding to material properties. Let us denote $V_{\boldsymbol{g}} = \{\boldsymbol{v} \in H^1(\Omega)^d / \boldsymbol{v}_{|\Gamma_d} = \boldsymbol{g}\}$.
The basic (displacement) variational formulation is: find $\boldsymbol{u} \in V_{\mathbf{0}}(\Omega)$, such that:

$$\int_\Omega \varepsilon(\boldsymbol{v}) : \mathcal{A}\varepsilon(\boldsymbol{u}) = \int_\Omega \rho\, \boldsymbol{v}.\boldsymbol{f} + \int_\Gamma ((\mathcal{A}\varepsilon(\boldsymbol{u}))\boldsymbol{n}).\boldsymbol{v}, \quad \forall \boldsymbol{v} \in V_0(\Omega). \tag{2}$$

The Hooke's law says that $\mathcal{A} = \lambda\, \boldsymbol{I}_d + 2\mu\, \mathbf{1}_{d,d}$, where $\boldsymbol{I}_d$ is the Identity $d \times d$ matrix and $\mathbf{1}_{d,d}$ the constant $d \times d$ matrix filled with 1.
Question: How to code this equation with FreeFem++?
Remark: the contraction operator (:) exists, but its priority is low: try to avoid it!

Outline

Representation of the Strain tensor ε: the Voigt notation

More details on `https://en.wikipedia.org/wiki/Voigt_notation`
We denote by `lambda` $= \lambda$, `mu` $= \mu$, `twomul` $= 2\lambda + \mu$, and define
In 2d:

```
func A =    [[twomul,lambda,    0.   ],
             [lambda,twomul,    0.   ],
             [   0.   ,   0.   ,  mu    ]] ;

macro epsV(u1,u2)   [dx(u1),dy(u2),dy(u1)+dx(u2)] // EOM
macro div(u1,u2)  ( dx(u1)+dy(u2) ) // EOM
```

In 3d:

```
func A =    [[twomul,lambda,lambda, 0. ,   0. , 0. ],
             [lambda,twomul,lambda, 0. ,   0. , 0. ],
             [lambda,lambda,twomul, 0. ,   0. , 0. ],
             [   0.   ,   0.   ,   0.   , mu ,   0. , 0. ],
             [   0.   ,   0.   ,   0.   , 0. ,   mu , 0. ],
             [   0.   ,   0.   ,   0.   , 0. ,   0. , mu ]   ] ;

macro epsV(u1,u2,u3)   [dx(u1), dy(u2), dz(u3),
                         dz(u2)+dy(u3), dz(u1)+dx(u3), dy(u1)+dx(u2)  ] // EOM
macro div(u1,u2,u3) ( dx(u1)+dy(u2)+ dz(u3)   ) // EOM
```

Outline

Solving the static linear elasticity equation in 2d with FreeFem++

For the values of material constants, see
`http://www.mstrtech.com/WebPages/matexam.htm`

```
// Steel  λ = 9.695 10^10 N/m^2,
// μ = 7.617 10^10 N/m^2,  ρ = 7700kg/m^3.
real rho =  7700, mu =  7.617e10, lambda = 9.69e10 ;
real gravity = -9.81, twomul=2*mu+lambda; // Optimisation
cout << "lambda,mu,gravity␣="<<lambda<< "␣" << mu << "␣" << gravity << endl;
```

The `FreeFem++` code:

```
int[int] labs=[1,1,1,2];
mesh Th=square(50,5,[x*10,y],label=labs);
fespace Vh(Th,[P1,P1]);

Vh [u1,u2], [v1,v2],[un1,un2],[up1,up2];
solve Lame([u1,u2],[v1,v2])=  int2d(Th)(  epsV(u1,u2)'*A*epsV(v1,v2))
  - int2d(Th) ( rho*gravity*v2)  + on(2,u1=0,u2=0) ;

real dmax= u1[].linfty, coef= 3/dmax;
cout << "␣max␣deplacement␣=␣" << dmax << "␣coef␣" << coef << endl;
mesh Thm = change(movemesh(Th,[x+u1*coef,y+u2*coef]),fregion=1);
plot(Th,Thm,wait=1,cmm="coef␣␣amplification␣=␣"+coef);
```

Run:Beam-Static-2d.edp

Outline

Solving the static linear elasticity equation in 3d with FreeFem++

Using the same physical parameters: $\lambda = 9.695\ 10^{10} N/m^2$, $\mu = 7.617\ 10^{10} N/m^2$, $\rho = 7700 kg/m^3$.

The FreeFem++ code:

```
int[int] labs=[1,1,1,2,1,1];
mesh3 Th=cube(50,5,5,[x*10,y,z],label=labs);

fespace Vh(Th,[P1,P1,P1]);
Vh [u1,u2,u3], [v1,v2,v3],[un1,un2,un3],[up1,up2,up3];
solve Lame([u1,u2,u3],[v1,v2,v3])=
    int3d(Th)(  epsV(u1,u2,u3)'*A*epsV(v1,v2,v3))
  - int3d(Th)  ( rho*gravity*v3)
  + on(2,u1=0,u2=0,u3=0)
  ;
real dmax= u1[].linfty, coef= 5/dmax;
cout << " max deplacement = " << dmax << " coef " << coef << endl;
int[int] llm=[1,3];// just to change the color of plot mesh
mesh3 Thm=movemesh(Th,[x+u1*coef,y+u2*coef,z+u3*coef],label=llm);
plot(Th,Thm, wait=1,cmm="coef  amplification = "+coef);
```

Run:Beam-Static-3d.edp

Outline

Solving the time-dependent linear elasticity equation in 2d and 3d

The problem is (strong formulation)

$$\rho \partial_{tt} \boldsymbol{u} - \nabla.(\sigma(\boldsymbol{u})) = \rho \boldsymbol{f} \quad \text{in} \quad \Omega.$$

We use a classical explicit 2nd order finite difference scheme for the time derivative:

$$\rho \frac{u^{n+1} - 2u^n + u^{n-1}}{(\delta t)^2} - \nabla.(\sigma(\boldsymbol{u}^n)) = \rho \boldsymbol{f}^n$$

Let us denote by `un` $= u^{n+1}$, `u` $= u^n$, `up` $= u^{n-1}$;
the matrix formulation of the problem is:

$$\mathrm{un} = \boldsymbol{M}^{-1} b, \quad b = \boldsymbol{M}(-\mathrm{up}) + \boldsymbol{A}\mathrm{u} + r, \quad +B.C$$

$$\boldsymbol{M} \equiv \int_\Omega \rho \frac{\boldsymbol{u}.\boldsymbol{v}}{(\delta t)^2} + B.C, \quad \boldsymbol{A} \equiv \int_\Omega -\varepsilon(\boldsymbol{v}) : \mathcal{A}\varepsilon(\boldsymbol{u}) + 2\rho \frac{\boldsymbol{u}.\boldsymbol{v}}{(\delta t)^2}, \quad r \equiv \int_\Omega \rho\, g\, \boldsymbol{e}_3.\boldsymbol{v} + B.C$$

Program for the time-dependent linear elasticity equation in 2d and 3d

```
include "Beam-Static-2d.edp"
real dt =1e-5,     rhodt2= rho/dt/dt;
varf vA(  [u1,u2],[v1,v2]) =   int2d(Th)(  -1*epsV(u1,u2)'*A*epsV(v1,v2)
                + 2*rhodt2*[u1,u2]'*[v1,v2]);
varf vM(  [u1,u2],[v1,v2]) =   int2d(Th)(  rhodt2*[u1,u2]'*[v1,v2])
                + on(2,u1=0,u2=0);
varf vB(  [u1,u2],[v1,v2]) =   int2d(Th)(  rho*[0,gravity]'*[v1,v2])
                + on(2,u1=0,u2=0);

matrix AA=vA(Vh,Vh),       M=vM(Vh,Vh,solver=CG);
real[int]  Rhs = vB(0,Vh);

func BB=[[-0.5,-7],[10.5,1.4]];// for fixe bounding box of the plot ..
up1[]=u1[]=0;
for(int i=0; i<100000; ++i)  {
    real[int] b = AA*u1[];   up1[]=-up1[];   b += Rhs;    b +=  M*up1[];
    un1[]=  M^-1*b;
    up1[]=u1[];    u1[]=un1[];
    if(i%100==0)   {       cout << i << "␣" << u1[].linfty << endl;
     mesh Thmm =movemesh(Th,[x+u1*coef,y+u2*coef]);
```

Run:Beam-Vibration2d.edp   Run:Beam-Vibration3d.edp

Outline

Outline

2. Non-linear problems (Lesson 6)
   - Algorithms for solving non-linear problems: fixed point algorithm, Newton method
   - Example of a non-linear problem: the Minimal Surface problem
   - A fixed-point method to solve the Minimal Surface problem
   - A Newton method to solve the Minimal Surface problem

The fixed-point algorithm

Consider the non-linear problem $F(u,u)=0$, with $F(.,u)$ affine with respect to the first variable. To find a solution, you can try the following basic method, with no guaranty of convergence:

1. set u^0 an initial guess
2. do (iterations following n)
    1. find u^{n+1}, the solution to $F(u^{n+1},u^n)=0$,
    2. if($||u^{n+1}-u^n|| < \varepsilon$) break;

The difficulty in this algorithm is to find an initial guess; sometimes this algorithm explodes. The convergence is generally slow.

The Newton method

To solve $F(u) = 0$ we can also use the Newton method (DF is the differential of F):

```
1. set $u^0$, an initial guess
2. do (iterations following $n$)
    1. find $w^n$, solution to $DF(u^n)w^n = F(u^n)$
    2. $u^{n+1} = u^n - w^n$
    3. if( $||w^n|| < \varepsilon$) break;
```

The Optimized Newton Method:
if $F = C + L + N$, with C the constant, L the linear, and N the non-linear part of F.
We obtain that $DF = L + DN$ and the Newton method can be written as:
$DF(u^n)u^{n+1} = DF(u^n)u^n - F(u^n) = DN(u^n)u^n - N(u^n) - C$.
The new version of the algorithm is:

```
1. do
    1. find $u^{n+1}$ solution to
       $DF(u^n)u^{n+1} = DN(u^n)u^n - N(u^n) - C = DF(u^n)u^n - F(u^n)$
    2. if( $||u^{n+1} - u^n|| < \varepsilon$) break;
```

The weakness of this algorithm is the need to start from an initial guess sufficiently close to a solution.

Outline

Example of a non-linear problem: the Minimal Surface problem

Let us solve the following geometrical problem: Find a function $u : \Omega \mapsto \mathbb{R}$, where u is given on $\Gamma = \partial\Omega$, (i.e. $u_{|\Gamma} = g$) such as the area of the surface S, parametrized by $(x, y) \in \Omega \mapsto (x, y, u(x, y))$ is minimal.
The mathematical formulation of the problem is:

$$\arg\min J(u) = \int_\Omega \left\| \left(\begin{array}{c} 1 \\ 0 \\ \partial_x u \end{array} \right) \times \left(\begin{array}{c} 0 \\ 1 \\ \partial_y u \end{array} \right) \right\| d\Omega = \int_\Omega \sqrt{1 + (\partial_x u)^2 + (\partial_y u)^2}\, d\Omega.$$

The Euler-Lagrange equation associated to the minimization of $J(u)$ is:

$$\forall v / v_{|\Gamma} = 0 \quad : \quad DJ(u)v = \int_\Omega \frac{(\partial_x v \partial_x u + \partial_y v \partial_y u)}{\sqrt{1 + (\partial_x u)^2 + (\partial_y u)^2}}\, d\Omega = 0.$$

We consider the case: $\Omega =]0, \pi[^2$ and $g(x, y) = \cos(nx)\cos(ny)$,
$n = 1$ (simplest problem) and $n = 2$ or 4 (harder to solve).
We shall use the fixed-point algorithm and the Newton method.

Outline

2. Non-linear problems (Lesson 6)
    - Algorithms for solving non-linear problems: fixed point algorithm, Newton method
    - Example of a non-linear problem: the Minimal Surface problem
    - A fixed-point method to solve the Minimal Surface problem
    - A Newton method to solve the Minimal Surface problem

A fixed-point method to solve the Minimal Surface problem

```
int nn=100,n=4;// n= 1 ,2,4
int[int]  l1=[1,1,1,1];
mesh Th= square(nn,nn,[x*pi,y*pi],label=l1);
func g = cos(n*x)*cos(n*y);
fespace Vh(Th,P1);
Vh un,u,v;
for(int i=0; i< 1000; ++i)
{    verbosity =0;
     solve Pb(un,v) = int2d(Th)( (dx(un)*dx(v)+ dy(un)*dy(v))
                        / sqrt( 1. +  (dx(u)*dx(u)+ dy(u)*dy(u)) )
     + on(1,un = g);
     real J = int2d(Th)( sqrt( 1. +  (dx(un)*dx(un)+ dy(un)*dy(un)) );
     plot(un,dim=3,fill=1, wait=0);
     u[]-=un[]; // diff
     real err= u[].linfty;
     cout << "␣iter␣" << i << "␣" << err <<"␣" << "␣J␣" << J <<  endl;
     if( err < 1e-6) break;
     u[]=un[]; }
```

Run:Min-Surf-FixPoint.edp

Outline

A Newton method to solve the Minimal Surface problem

```
// macro of compute all differentiel
macro grad2(u,v)  ( dx(u)*dx(v)+ dy(u)*dy(v) ) //
macro JJ(u) sqrt( 1. +  grad2(u,u) ) //
macro dJJ(u,du)  ( grad2(u,du) / JJ(u) ) //
macro ddJJ(u,du,ddu)  ( grad2(ddu,du)/JJ(u)
                       -  (grad2(u,du)*grad2(u,ddu)/JJ(u)^3) ) // For Newton
fespace Vh(Th,P1);
Vh u,v,w;
// Stating point ...
solve Pb0(u,v) = int2d(Th)( grad2(u,v) )  + on(1,u = g);
plot(u,dim=3,wait=0);
 // Newton loop
for(int i=0; i< 100; ++i)
{  verbosity =0;
    solve PbTangent(w,v) = int2d(Th)( ddJJ(u,w,v) ) - int2d(Th)( dJJ(u,v) )
    + on(1,2,3,4,w = 0);
    u[] -=w[];
    real J = int2d(Th)( JJ(u) );
    plot(u,dim=3,fill=1, wait=0,cmm=" J = "+J);
    real err= w[].linfty;
    cout << " iter " << i << " err= " << err <<" " << " J " << J  << endl;
    if( err < 1e-6 || err >100) break;   }
```

Run:Min-Surf-Newton.edp   Run:Min-Surf-Newton-V2.edp

Outline

Outline

Mesh adaptivity: Metrics and Unit Mesh

In Euclidean geometry the length $|\gamma|$ of a curve γ of $\mathbb{R}^d$ parametrized by $\gamma(t)_{t=0..1}$ is

$$|\gamma| = \int_0^1 \sqrt{< \gamma'(t), \gamma'(t) >}\, dt$$

We introduce the metric $\mathcal{M}(x)$ as a field of $d \times d$ symmetric positive definite matrices, and the length ℓ of Γ w.r.t $\mathcal{M}$ is:

$$\ell = \int_0^1 \sqrt{< \gamma'(t), \mathcal{M}(\gamma(t))\gamma'(t) >} dt$$

The key-idea is to construct a mesh for which the lengths of the edges are close to 1, accordingly to $\mathcal{M}$.

Metrics intersection

For a metric $\mathcal{M}$, the unit ball $\mathcal{BM}$ (obtained by plotting the maximum mesh size in all directions), is a ellipse.
If you we have two unknowns u and v, we just compute the metrics $\mathcal{M}_u$ and $\mathcal{M}_v$, find a metric $\mathcal{M}_{uv}$, called intersection, defined by the biggest ellipse such that:

$$\mathcal{B}(\mathcal{M}_v) \subset \mathcal{B}(\mathcal{M}_u) \cap \mathcal{B}(\mathcal{M}_v)$$

Example of an adaptive mesh

$$u = (10x^3 + y^3) + tanh(500(sin(5y) - 2x)));$$

$$v = (10y^3 + x^3) + tanh(5000(sin(5y) - 2*)));$$

Run:Adapt-uv.edp

Outline

Building the metrics from the solution u

For P_1 continuous Lagrange finite elements, the optimal metric norms for the interpolation error (used in the function `adaptmesh` in FreeFem++) are:

- L^∞ : $\mathcal{M} = \frac{1}{\varepsilon}|\nabla\nabla u| = \frac{1}{\varepsilon}|\mathcal{H}|$, where $\mathcal{H} = \nabla\nabla u$
- L^p : $\mathcal{M} = \frac{1}{\varepsilon}|det(\mathcal{H})|^{\frac{1}{2p+2}}\,|\mathcal{H}|$, (result by F. Alauzet, A. Dervieux)

For the norm $W^{1,p}$, the optimal metric $\mathcal{M}_\ell$ for the P_ℓ Lagrange finite element is given by (with only acute triangles) (thanks to J-M. Mirebeau)

$$\mathcal{M}_{\ell,p} = \frac{1}{\varepsilon}(det\mathcal{M}_\ell)^{\frac{1}{\ell p+2}}\,\mathcal{M}_\ell$$

and (see `MetricPk` plugin and function)

- for P_1: $\mathcal{M}_1 = \mathcal{H}^2$ (sub-optimal: for acute triangles, take $\mathcal{H}$)
- for P_2: $\mathcal{M}_2 = 3\sqrt{\begin{pmatrix} a & b \\ b & c \end{pmatrix}^2 + \begin{pmatrix} b & c \\ c & a \end{pmatrix}^2}$ with $D^{(3)}u(x,y) = (ax^3 + 3bx^2y + 3cxy^2 + dy^3)/3!$,

Run:adapt.edp

Run:AdaptP3.edp

Outline

The problem with a corner singularity (adaptivity with metrics)

The domain is a L-shaped polygon $\Omega =]0,1[^2 \backslash [\frac{1}{2},1]^2$ and the PDE is

$$\text{find } u \in H^1_0(\Omega) \text{ such that } \quad -\Delta u = 1 \text{ in } \Omega.$$

The solution has a singularity at the re-entrant angle and we wish to capture it numerically.

The problem with a corner singularity (FreeFem++ program)

```
int[int] lab=[1,1,1,1];
mesh Th = square(6,6,label=lab);
Th=trunc(Th,x<0.5 | y<0.5, label=1);
fespace Vh(Th,P1);
Vh u,v;
real error=0.01;
problem Problem1(u,v,solver=CG,eps=1.0e-6) =
     int2d(Th)(  dx(u)*dx(v) + dy(u)*dy(v))  -   int2d(Th)( v)
     + on(1,u=0);
for (int i=0;i< 7;i++)
{
   Problem1;                            // solving the pde problem
   plot(u,Th,wait=1);

   Th=adaptmesh(Th,u,err=error,nbvx=100000);   // the adaptation with Hessian of u
   u=u;
} ;
```

Run:CornerLap.edp

Outline

Solving the 3d Poisson equation using mesh adaptation

```
load "msh3" load "tetgen" load "mshmet" load "medit"
int nn  = 6;  int[int] l1=[1,1,1,1,1,1];
mesh3 Th3=trunc(  cube(nn,nn,nn,label=l1)
                ,(x<0.5)|(y < 0.5)|(z < 0.5), label=1);
fespace Vh(Th3,P1);      Vh u,v,h;
macro Grad(u) [dx(u),dy(u),dz(u)] // EOM
problem Poisson(u,v,solver=CG) = int3d(Th3)( Grad(u)'*Grad(v) )
       -int3d(Th3)( 1*v ) + on(1,u=0);
real errm=1e-2;// level of error
for(int ii=0; ii<5; ii++)
{ Poisson;
  cout <<" u min, max = " <<  u[].min << " "<< u[].max << endl;
  h=0. ;// for resizing h
  h[]=mshmet(Th3,u,normalization=1,aniso=0,nbregul=1,hmin=1e-3,hmax=0.3,err=errm);
  cout <<" h min, max = " <<  h[].min << " "<< h[].max << " " << h[].n << " "
       << Th3.nv << " " << Th3.nt << endl;
  plot(u,wait=1);
  errm*= 0.8;// change the level of error
  Th3=tetgreconstruction(Th3,switch="raAQ",sizeofvolume=h*h*h/6.); }
Poisson;
medit("U-adap-iso-"+5,Th3,u,wait=1);
```

Run:Laplace-Adapt-3d.edp

Outline

A Newton method with mesh adaptation for the Minimal Surface problem

```
real errA=0.1;
for(int adap=0; adap<7; adap++)
{ verbosity =0;
  for(int i=0; i< 100; ++i)
   {   // ALGO NEWTOW OPTIMIZE
    solve PbTangent(un,v) = int2d(Th)( ddJJ(u,un,v) ) - int2d(Th)( ddJJ(u,u,v) -
        dJJ(u,v) )
    + on(1,2,3,4,un = g);
    w[] =u[] -un[];   u[]=un[];
    real J = int2d(Th)( JJ(u) );
    plot(u,dim=3,fill=1, wait=0,cmm="␣J␣="+J);
    real err= w[].linfty;
    cout << "␣iter␣" << i << "␣" << err <<"␣" << "␣J␣" << J << "␣"  << "␣" << errA
         <<  endl;
    if( err < 1e-5) break;
    assert(err<10); }
  cout << "adaptmesh␣␣" << endl;
  Th = adaptmesh(Th,u,err=errA,nbvx=100000,ratio = 1.5);
  plot(Th,WindowIndex=1);
  v=0;u=u; w=0; un=un; // resize
  errA = errA/2;
}
```

Run:Min-Surf-Newton-Adapt.edp

Outline

Outline

The stress tensor for a Newtonian fluid

In the domain Ω of $\mathbb{R}^d$, we denote by u the velocity field, p the pressure field and μ the dynamic viscosity. The classical mechanical stress $\boldsymbol{\sigma}^\star$ of the fluid is:

$$\boldsymbol{\sigma}^\star(\boldsymbol{u},p) = 2\mu\mathbb{D}(\boldsymbol{u}) - p\,I_d, \qquad \mathbb{D}(\boldsymbol{u}) = \frac{1}{2}(\nabla\boldsymbol{u} + {}^t\nabla\boldsymbol{u}) \tag{3}$$

or in the math formulation:

$$\boldsymbol{\sigma}^\bullet(\boldsymbol{u},p) = \mu\nabla\boldsymbol{u} - p\,I_d \tag{4}$$

So $\boldsymbol{\sigma}$ is one of these two stress tensors. Remark: if $\nabla.\boldsymbol{u} = 0$ and μ is constant, then $\nabla.2\mathbb{D}(\boldsymbol{u}) = \mu\nabla.\nabla\boldsymbol{u} + \mu\nabla.{}^t\nabla\boldsymbol{u} = \mu\nabla.\nabla\boldsymbol{u} + \mu\nabla\underbrace{\nabla.\boldsymbol{u}}_{=0} = \mu\nabla^2\boldsymbol{u} = \mu\Delta\boldsymbol{u}.$

Stokes Equation: find the velocity field $\boldsymbol{u}$ and the pressure field p, satisfying :

$$\begin{aligned} -\nabla.\boldsymbol{\sigma}(\boldsymbol{u},p) &= \boldsymbol{f} && (5)\\ -\nabla.\boldsymbol{u} &= 0 && (6)\end{aligned} \qquad \text{or} \qquad \begin{aligned} -\mu\Delta\boldsymbol{u} + \nabla p &= \boldsymbol{f} && (7)\\ -\nabla.\boldsymbol{u} &= 0 && (8)\end{aligned}$$

where $\boldsymbol{f}$ is the density of external forces.
+ Boundary conditions that will be defined through the variational (weak) form.

Outline

Stokes equation: variational formulation

Mechanical variational form of the Stokes equation:

$$\forall \boldsymbol{v}, q; \quad \int_\Omega 2\mu \mathbb{D}(\boldsymbol{u}) : \mathbb{D}(\boldsymbol{v}) - q\nabla.\boldsymbol{u} - p\nabla.\boldsymbol{v} = \int_\Omega \boldsymbol{f}.\boldsymbol{v} + \int_\Gamma {}^t\boldsymbol{n}\boldsymbol{\sigma}^\star(\boldsymbol{u},p)\boldsymbol{v}$$

Mathematical variational form of the Stokes equation:

$$\forall \boldsymbol{v}, q; \quad \int_\Omega \mu\nabla\boldsymbol{u} : \nabla\boldsymbol{v} - q\nabla.\boldsymbol{u} - p\nabla.\boldsymbol{v} = \int_\Omega \boldsymbol{f}.\boldsymbol{v} + \int_\Gamma {}^t\boldsymbol{n}\boldsymbol{\sigma}^\bullet(\boldsymbol{u},p)\boldsymbol{v}$$

But remember that ${}^t\boldsymbol{n}\boldsymbol{\sigma}^\bullet(\boldsymbol{u},p)$ are boundary density forces $\boldsymbol{f}_\Gamma$ and not ${}^t\boldsymbol{n}\boldsymbol{\sigma}^\star(\boldsymbol{u},p)$.

If the B.C. is $\boldsymbol{u} = \boldsymbol{u}_\Gamma$ for all boundaries, then the two formulations are identical.
The pressure p is defined up to an additive constant and the weak formulation can use a small regularization (to remove the problem of the additive constant and impose a zero mean value for the pressure).

$$\forall \boldsymbol{v} \in (H^1_0)^d, q \in L^2; \quad \int_\Omega \mu\nabla\boldsymbol{u} : \nabla\boldsymbol{v} - q\nabla.\boldsymbol{u} - p\nabla.\boldsymbol{v} - \varepsilon pq = \int_\Omega \boldsymbol{f}.\boldsymbol{v}$$

Solving the Stokes equation with FreeFem++: the entrained cavity flow

```
int nn=10;
mesh Th=square(nn,nn);
macro grad(u)  [dx(u),dy(u)] //
macro Grad(u1,u2) [grad(u1),grad(u2)]  //
macro D(u1,u2)  [  [dx(u1),(dy(u1)+dx(u2))*.5] ,  [(dy(u1)+dx(u2))*.5,dy(u2)] ]   //
macro div(u1,u2)  (dx(u1)+dy(u2))//
real epsp =1e-8, mu = 1;
```

Choose the correct finite-element couple for velocity and pressure: (P2,P1), (P1b,P1), (P1nc, P0), ...

```
fespace Vh(Th,P2); fespace Ph(Th,P1); // Taylor Hood Finite element


Vh u1,u2, v1,v2;   Ph p,q ;
solve Stokes([u1,u2,p],[v1,v2,q]) =
int2d(Th) ( mu*(Grad(u1,u2):Grad(v1,v2))
            - div(u1,u2)*q   - div(v1,v2)*p  -epsp*p*q )
 + on(1,2,4,u1=0,u2=0) + on(3,u1=1,u2=0) ;
 plot([u1,u2],p,wait=1);
 cout << " mean value pressure= " << int2d(Th)(p)/Th.area<<endl;
```

Run:Stokes-Cavity.edp

Outline

Incompressible Navier-Stokes equation: steady states

Computing steady states of the Incompressible Navier-Stokes equation: In the domain Ω of $\mathbb{R}^d$, find the velocity field $\boldsymbol{u}$ and the pressure field p, solution to:

$$
\begin{aligned}
(\boldsymbol{u}.\nabla)\boldsymbol{u} - \nabla.\boldsymbol{\sigma}(\boldsymbol{u}, p) &= \boldsymbol{f}, && (9)\\
-\nabla.\boldsymbol{u} &= 0, && (10)
\end{aligned}
$$

+ Boundary conditions.

First idea: use the Optimized Newton Method (see page 19)! the only non-linear term is $N(u) = (\boldsymbol{u}.\nabla)\boldsymbol{u}$ and the differential is $DN(u)w = (\boldsymbol{u}.\nabla)\boldsymbol{w} + (\boldsymbol{w}.\nabla)\boldsymbol{u}$; so, the iteration ℓ of the Newton algorithm is:
Find $\boldsymbol{u}^\ell, p^\ell$ such that

$$
\begin{aligned}
\forall \boldsymbol{v} \in (H^1_0)^d, \qquad & q \in L^2;\\
& \int_\Omega \mu(\nabla\boldsymbol{u}^\ell : \nabla\boldsymbol{v}) - q\nabla.\boldsymbol{u}^\ell - p^\ell\nabla.\boldsymbol{v} + \boldsymbol{v}.((\boldsymbol{u}^\ell.\nabla)\boldsymbol{u}^{\ell-1} + (\boldsymbol{u}^{\ell-1}.\nabla)\boldsymbol{u}^\ell) - \varepsilon p^\ell q\\
= \quad & \int_\Omega \boldsymbol{v}.((\boldsymbol{u}^{\ell-1}.\nabla)\boldsymbol{u}^{\ell-1}) + \boldsymbol{f}.\boldsymbol{v}
\end{aligned}
$$

Incompressible Navier-Stokes equation: steady states of the entrained cavity

```
real epsp =1e-8, mu = 1./Reynold , eps= 1e-5;
Vh u1=0,u2=0, un1,un2, v1,v2;   Ph p,pn,q ;
macro UGradW( u1,u2, w1,w2) [ [u1,u2]'*grad(w1) , [u1,u2]'*grad(w2)]//
verbosity=0;
for(int iter=0; iter<20; ++iter)
{  // DF(u)un = DN(u)u - N(u)  = UGradW(u1,u2,u1,u2)
   solve Tangent ([un1,un2,pn],[v1,v2,q]) =
     int2d(Th) ( UGradW(u1,u2,   un1,un2)'*[v1,v2]
               +  UGradW(un1,un2, u1,u2)'*[v1,v2]
               +  mu*(Grad(un1,un2):Grad(v1,v2))
              - div(un1,un2)*q   - div(v1,v2)*pn  -epsp*pn*q
               )
   - int2d(Th) ( UGradW(u1,u2,u1,u2)'*[v1,v2]      )
  + on(1,2,4,un1=0,un2=0) + on(3,un1=1,un2=0) ;
   u1[]-=un1[];   u2[]-=un2[]; p[]-=pn[]; //diff err
   real err1=u1[].linfty, err2 =u2[].linfty , errp = p[].linfty;
   cout << "␣iter␣=" <<iter << "␣errs=␣" << err1 << "␣"<< err2 << "␣" << errp <<
       endl;
   u1[]=un1[];   u2[]=un2[];  p[]=pn[];
   plot([u1,u2],p,wait=1,cmm=iter);
   if( err1 < eps & err2 < eps &   errp < eps) break;
}
```

Run:Navier-Stokes-Cavity.edp

Incompressible Navier-Stokes equation: unsteady computations

In the domain Ω of $\mathbb{R}^d$, find the velocity field $\boldsymbol{u}$ and the pressure field p, solution to:

$$\begin{aligned} \partial_t \boldsymbol{u} + (\boldsymbol{u}.\nabla)\boldsymbol{u} - \nabla.\boldsymbol{\sigma}(\boldsymbol{u}, p) &= \boldsymbol{f}, \quad (11) \\ -\nabla.\boldsymbol{u} &= 0, \quad (12) \end{aligned}$$

+ Initial conditions + Boundary conditions.

We try to compute the classical Benchmark: Computations of Laminar Flow Around a Cylinder form, by M. Schäfer, S. Turek, F. Durst, E. Krause, R. Rannacher `http://www.mathematik.tu-dortmund.de/lsiii/cms/papers/SchaeferTurek1996.pdf` We compute the 2d case.

The Geometry and the physical constant are defined in file Run:2d-data-Turek-bm.edp.

One of the difficulties is to obtain the correct Strouhal number of the Bénard-von Karman vortex street.

We need a high-order scheme for the time integration: we use a multi-step BDF scheme of order 1, 2 or 3: BDF1 is Euler,

BDF2 is $\partial_t \boldsymbol{u} \sim \frac{3u^{n+1} - 4u^n + u^{n-1}}{2\delta t}$ and BDF3 is $\partial_t \boldsymbol{u} \sim \frac{11u^{n+1} - 18u^n + 9u^{n-1} - 2u^{n-2}}{6\delta t}$

(`https://en.wikipedia.org/wiki/Backward_differentiation_formula`)

Incompressible Navier-Stokes equation: unsteady computations (program)

```
real[int,int] BDF= [ [1,-1, 0,0],
                     [3./2.,-4/2., 1/2.,0],
                     [11./6.,-18./6., 9./6., -2./6.]];
```

to empty the file

```
{ofstream ff(datafile); }// empty file ..
```

to write in a file,

```
    drag = -int1d(Th,3) ( 2*nu* ([1.,0]'*D(un1,un2)*[N.x,N.y]) - p*N.x)   ;
    lift = -int1d(Th,3) ( 2*nu* ([0.,1.]'*D(un1,un2)*[N.x,N.y]) - p*N.y)   ;
    TCd[itime]=Cd = ccdrag*drag;
    TCl[itime]=Cl = ccdrag*lift;
    real deltap = p(xa,ya)-p(xe,ye);

    cout << "␣Time␣"<< time+dt << "␣at␣" << time/ccfreq << "␣Cd␣" << Cd << "␣Cl␣"
        << Cl
         << "␣Delta␣P=" << deltap << "/␣max:␣␣" << Cdx << "␣" << Clx << "␣" << Cpx
              << endl;
    ofstream ff(datafile,append);
    ff << time << "␣" << time/ccfreq << "␣" << Cd << "␣" << Cl << "␣"<< deltap
       << Cdx << "␣" << Clx << "␣" << Cpx <<endl;
```

Run:NS-Newton-Turek-bm.edp

Unsteady incompressible Navier-Stokes equation: method of characteristics

For a flow field $\boldsymbol{u}$ the total (or material) derivative is

$$\frac{D\boldsymbol{u}}{Dt} = \frac{\partial \boldsymbol{u}}{\partial t} + (\boldsymbol{u}.\nabla)\boldsymbol{u},$$

A correct numerical scheme used to approximate $\frac{D\boldsymbol{u}}{Dt}$ has to take into account the movement of a particle: let us denote by x^n (resp. x^{n+1}) the particle position at time t^n (resp. t^{n+1}); we can write

$$\frac{D\boldsymbol{u}}{Dt}(x^{n+1}) \sim \frac{\boldsymbol{u}^{n+1}(x^{n+1}) - \boldsymbol{u}^n(x^n)}{\delta t}$$

Defining the characteristic flow (passing at time t through the point $\boldsymbol{x}$)

$$\begin{cases} \frac{\partial \boldsymbol{X}}{\partial \tau}(\tau, t, \boldsymbol{x}) = \boldsymbol{u}(\tau, \boldsymbol{X}(\tau, t, \boldsymbol{x})), \quad \tau \in (0, t_{max}) \\ \boldsymbol{X}(t, t, \boldsymbol{x}) = \boldsymbol{x}, \end{cases} \tag{13}$$

one can express the total derivative of any function $\Phi(t, \boldsymbol{x})$ as

$$\frac{D\Phi}{Dt}(t, \boldsymbol{x}) = \left(\frac{\partial \Phi}{\partial t} + \boldsymbol{u}.\nabla\Phi\right)(t, \boldsymbol{x}) = \frac{\partial}{\partial t}\left(\Phi(\tau, \boldsymbol{X}(\tau, t, \boldsymbol{x}))\right)|_{\tau=t} \tag{14}$$

Method of characteristics in FreeFem++

We use the time discretization:

$$\left(\frac{D\Phi}{Dt}\right)^{n+1}(\boldsymbol{x}) \approx \frac{\Phi^{n+1}(\boldsymbol{x}) - \Phi^n \circ \boldsymbol{X}^n(\boldsymbol{x})}{\delta t}, \tag{15}$$

with $\boldsymbol{X}^n(\boldsymbol{x})$ a suitable approximation of $\boldsymbol{X}(t_n, t_{n+1}, \boldsymbol{x})$, obtained by an integration back in time of (13) from t_{n+1} to t_n for each grid point $\boldsymbol{x}$. The Galerkin characteristic method is implemented in Freefem++ as an operator computing $\Phi \circ \boldsymbol{X}^n$ for given: mesh, convection velocity field and time step.

The FreeFem++ operator `convect([u1,u2],-dt, ..)` computes:

$$\frac{D\boldsymbol{u}}{Dt}(x^{n+1}) \sim \frac{\boldsymbol{u}^{n+1} - \boldsymbol{u}^n \circ X^n}{\delta t}$$

Example: solve the convection equation with given velocity $\boldsymbol{u}$

$$\partial_t a + (\boldsymbol{u}.\nabla)a = 0, \quad + \text{ initial condition}$$

```
for (int i=0; i< 20 ; i++) {
    t += dt;    vo[]=v[];
    v=convect([u1,u2],-dt,vo);     // convect v by u1,u2, dt seconds, results in f
    plot(v,fill=1,wait=0,dim=3,cmm="convection:␣t="+t
         + ",␣min=" + v[].min + ",␣max=" +  v[].max); }
```

Run:convect.edp Exercise: use the characteristics method for the unsteady Navier-Stokes computation of the entrained cavity flow.

Outline

Outline

A free-boundary problem: modelling the water infiltration

We use a simple model to study water infiltration = the process by which water on the ground surface enters the soil.
Let Ω be a trapezoidal domain, defined in `FreeFem++` by:

```
real L=10,h=2.1 h1=0.35;       //Lenght, Left and Right Height
// trapeze
border a(t=0,L){x=t;y=0;};         // bas
border b(t=0,h1){x=L;y=t;};        // droite
border f(t=L,0){x=t;y=t*(h1-h)/L+h;}; // free surface
border d(t=h,0){x=0;y=t;};        // gauche
    int n=10;
mesh Th=buildmesh (a(L*n)+b(h1*n)+f(sqrt(L^2+(h-h1)^2)*n)+d(h*n));
plot(Th);
```

A free-boundary problem: modelling the water infiltration 2/2

The model is: find p and Ω such that:

$$\begin{cases} -\Delta p = 0 & \text{in } \Omega \\ p = y & \text{on } \Gamma_b \\ \dfrac{\partial p}{\partial n} = 0 & \text{on } \Gamma_d \cup \Gamma_a \\ \dfrac{\partial p}{\partial n} = \frac{q}{K} n_x & \text{on } \Gamma_f \qquad (Neumann) \\ p = y & \text{on } \Gamma_f \qquad (Dirichlet) \end{cases}$$

where the input water flux is $q = 0.02$, and $K = 0.5$.
The velocity u of the water is given by $u = -\nabla p$.

Modelling the water infiltration: the algorithm

We use the following fixed-point method: (*with bad main B.C. Run:freeboundaryPB.edp*)
Let $k = 0$, $\Omega^k = \Omega$. For the first step, we forget the Neumann B.C. and we solve the problem: find p in $V = H^1(\Omega^k)$, such as $p = y$ on Γ_b^k and Γ_f^k, and

$$\int_{\Omega^k} \nabla p \nabla p' = 0, \quad \forall p' \in V \text{ with } p' = 0 \text{ on } \Gamma_b^k \cup \Gamma_f^k$$

With the residual of the Neumann boundary condition, we build a domain transformation $\mathcal{F}(x,y) = [x, y - v(x)]$, where v is solution to: $v \in V$, such than $v = 0$ on Γ_a^k (bottom) and

$$\int_{\Omega^k} \nabla v \nabla v' = \int_{\Gamma_f^k} \left(\frac{\partial p}{\partial n} - \frac{q}{K} n_x\right) v', \quad \forall v' \in V \text{ with } v' = 0 \text{ sur } \Gamma_a^k$$

Remark: we can use the previous equation to evaluate

$$\int_{\Gamma^k} \frac{\partial p}{\partial n} v' = -\int_{\Omega^k} \nabla p \nabla v'$$

Modelling the water infiltration: implementation

The new domain is: $\Omega^{k+1} = \mathcal{F}(\Omega^k)$.
Warning: if is the displacement is too large we can have triangle overlapping.

```
Vh  u,v,uu,vv;
problem Pu(u,uu,solver=CG)  = int2d(Th)( dx(u)*dx(uu)+dy(u)*dy(uu))
  + on(b,f,u=y) ;
problem Pv(v,vv,solver=CG)  = int2d(Th)( dx(v)*dx(vv)+dy(v)*dy(vv))
  +  on (a, v=0) + int1d(Th,f)(vv*((q/K)*N.y- (dx(u)*N.x+dy(u)*N.y)));
real errv=1;
while(errv>1e-6)  {  j++;
  Pu;         Pv;
  plot(Th,u,v ,wait=0);
  errv=int1d(Th,f)(v*v);
```

Here tricky code to take account the triangle overlapping

```
  Th=movemesh(Th,[x,y-coef*v]); // calcul de la deformation
  cout << "\n\n"<<j <<"------------ errv = " << errv << "\n\n";
}
```

Run:freeboundary.edp

Outline

Computing eigenvalues and eigenvectors: an example

Find the first λ, u_λ such as:

$$a(u_\lambda, v) = \int_\Omega \nabla u_\lambda \nabla v = \lambda \int_\Omega u_\lambda v = \lambda b(u_\lambda, v)$$

Boundary conditions are imposed using exact penalization: we set to $1e30 = tgv$ the diagonal terms corresponding to locked degrees of freedom. Consequently, we impose Dirichlet boundary conditions only for the variational form of a and not for the variational form of b, because we compute eigenvalue of

$$\frac{1}{\lambda} v = A^{-1} B v$$

Otherwise we can get spurious mode.
FreeFem++ uses an Arpack interface:

```
int k=EigenValue(A,B,sym=true,value=ev,vector=eV);
```

Computing eigenvalues and eigenvectors: the program

```
real sigma = 0;  // value of the shift
varf  a(u1,u2)= int2d(Th)(  dx(u1)*dx(u2) + dy(u1)*dy(u2) - sigma* u1*u2 )
                        +  on(1,2,3,4,u1=0) ;  // Boundary condition
varf b([u1],[u2]) = int2d(Th)(  u1*u2 ) ; // no Boundary condition
matrix A= a(Vh,Vh,solver=UMFPACK);
matrix B= b(Vh,Vh,solver=CG,eps=1e-20);


.....

for (int i=0;i<k;i++)
{ u1=eV[i];
  real gg = int2d(Th)(dx(u1)*dx(u1) + dy(u1)*dy(u1));
  real mm= int2d(Th)(u1*u1) ;
  real err = int2d(Th)(dx(u1)*dx(u1) + dy(u1)*dy(u1) - (ev[i])*u1*u1) ;
  if(abs(err) > 1e-6) nerr++;
  if(abs(ev[i]-eev[i]) > 1e-1) nerr++;
  cout << " ---- " <<  i<< " " << ev[i] << " == " << eev[i] << " err= " << err <<
      " --- "<<endl;
  plot(eV[i],cmm="Eigen  Vector "+i+" valeur =" + ev[i]  ,wait=1,value=1,dim=3,
      fill=1);
}
```

Run:Lap3dEigenValue.edp   Run:LapEigenValue.edp   Run:free-cyl-3d.edp