# Finite Element Model Updating

Rian SEGHIR, Julien RETHORE

Research Institute in Civil Engineering and Mechanic (**GeM**)
Centrale Nantes, France


UNIVERSITÉ DE NANTES


# Why ?

# Experimental continuum mechanics

Find $(\mathbf{u}, \boldsymbol{\sigma})$ such that

**Admissibility for u**

$$\mathbf{u} = \mathbf{u}_d \text{ on } \Gamma_u + \text{regularity}$$

**Balance of momentum**

$$\text{div}(\boldsymbol{\sigma}) = 0$$

**Compatibility**

$$\boldsymbol{\varepsilon} = \frac{1}{2}(\nabla \mathbf{u} + \nabla^T \mathbf{u})$$

**Balance of external forces**

$$\boldsymbol{\sigma}(\mathbf{n}) = \mathbf{F}_d \text{ on } \Gamma_F$$

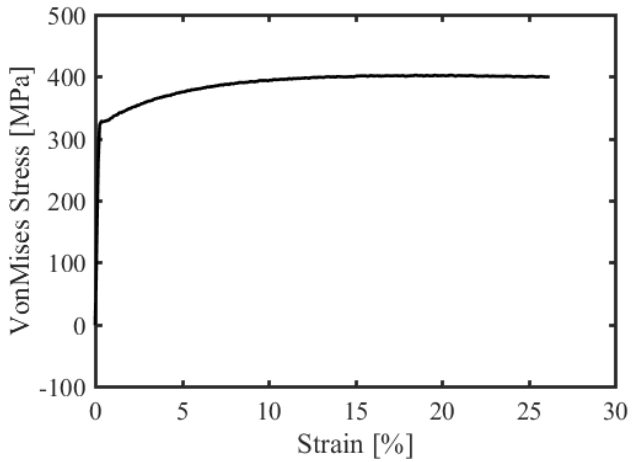
**Regularization**

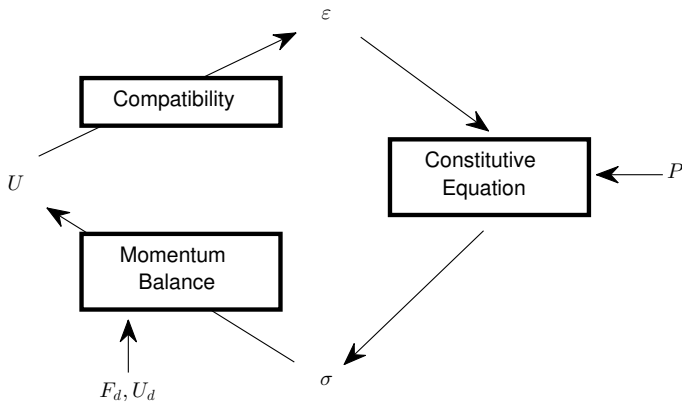
Sample with a geometry such that the solution is unique.

Usual specimen geometry for uni-, bi-, tri-axial testing,...

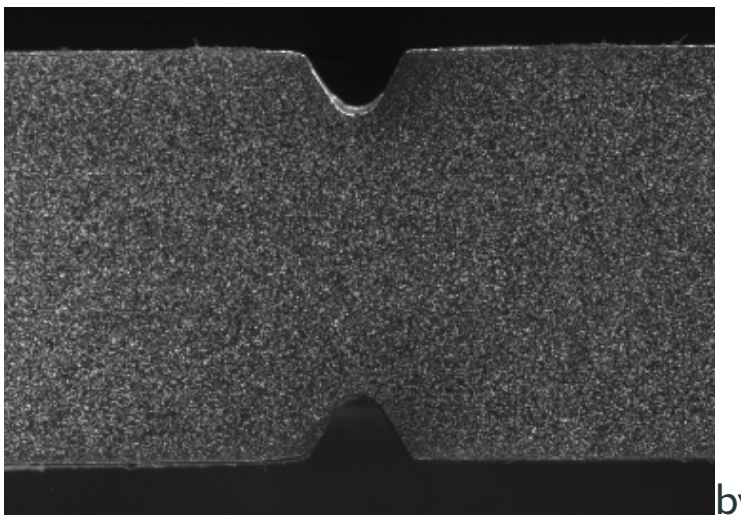
# Identification of constitutive laws

- "Engineering" approach



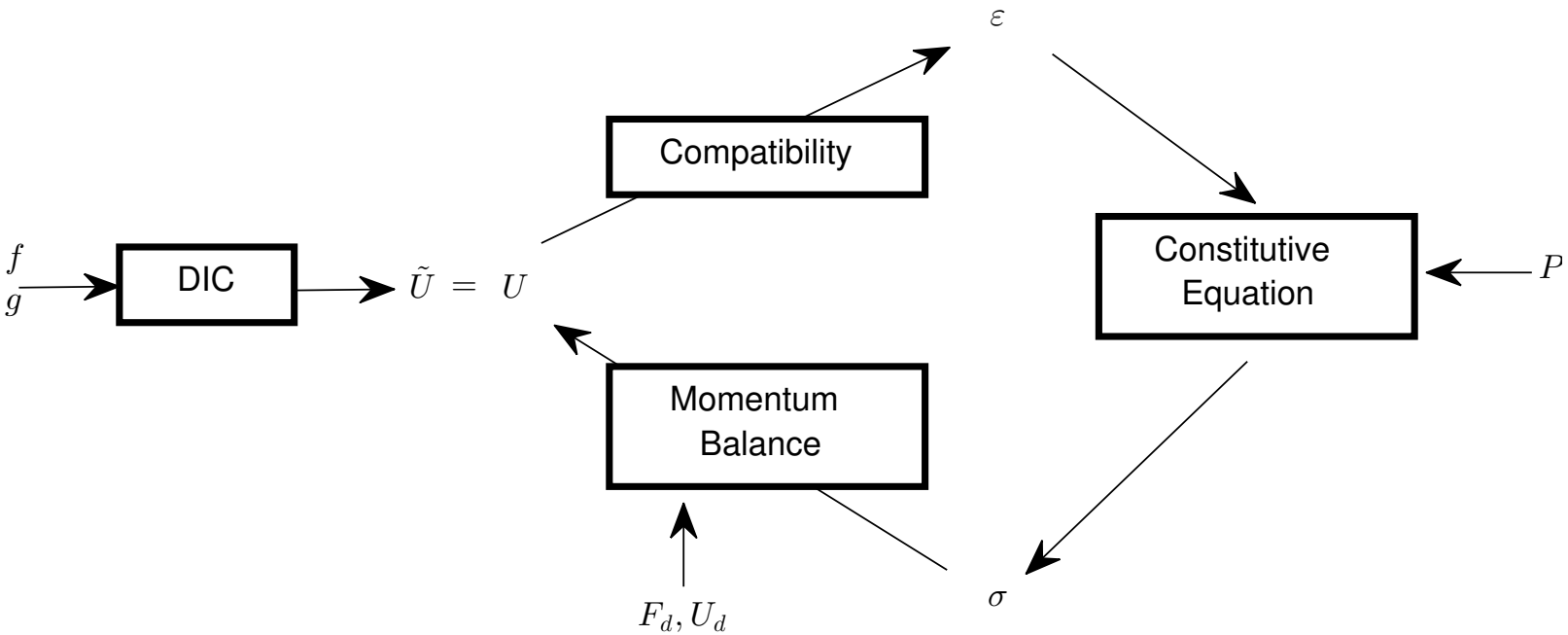

- Validation ?

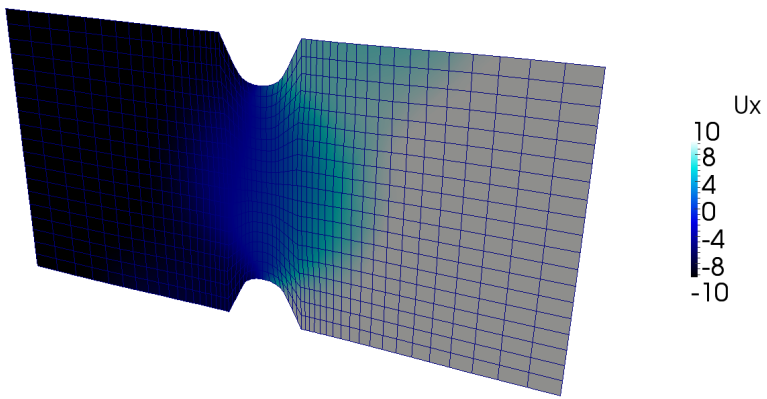


by G. Portemont at ONERA Lille

# Identification of constitutive laws

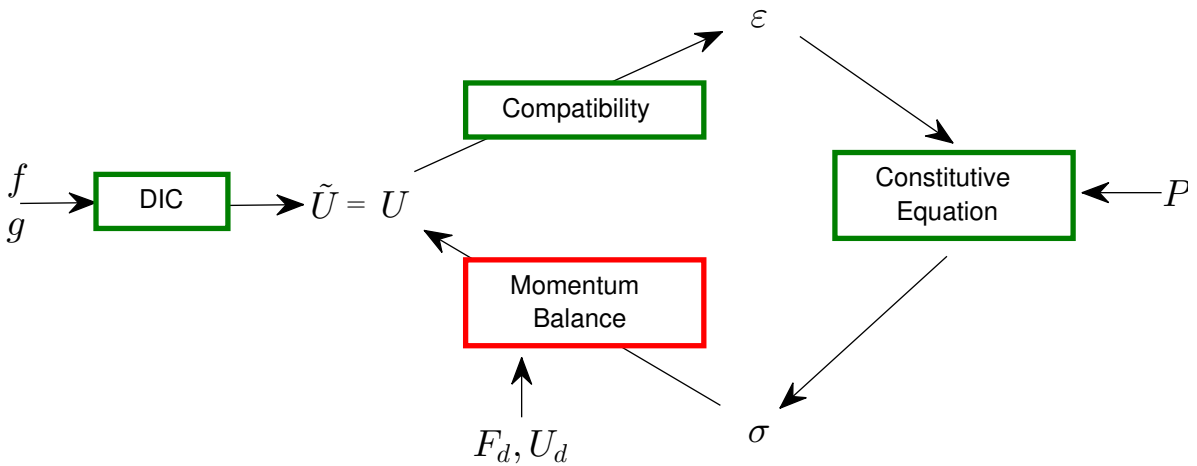
- Photomechanics



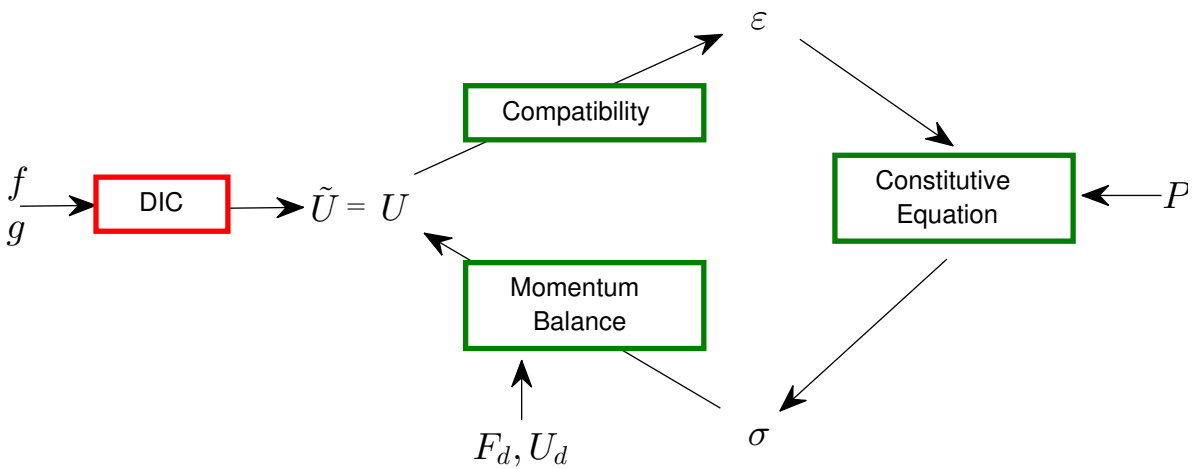



# Identification of constitutive laws

- Stress calculation

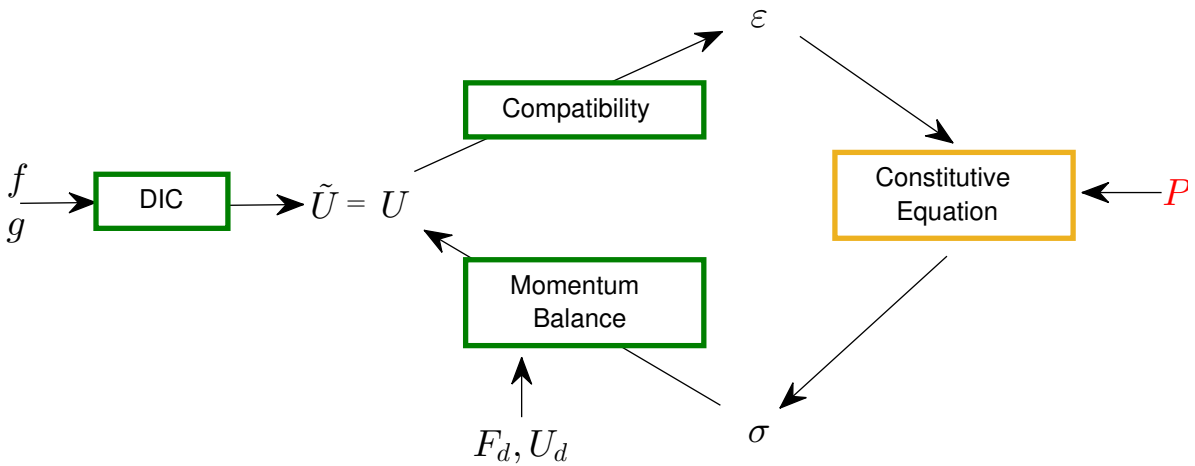



- Numerical simulation

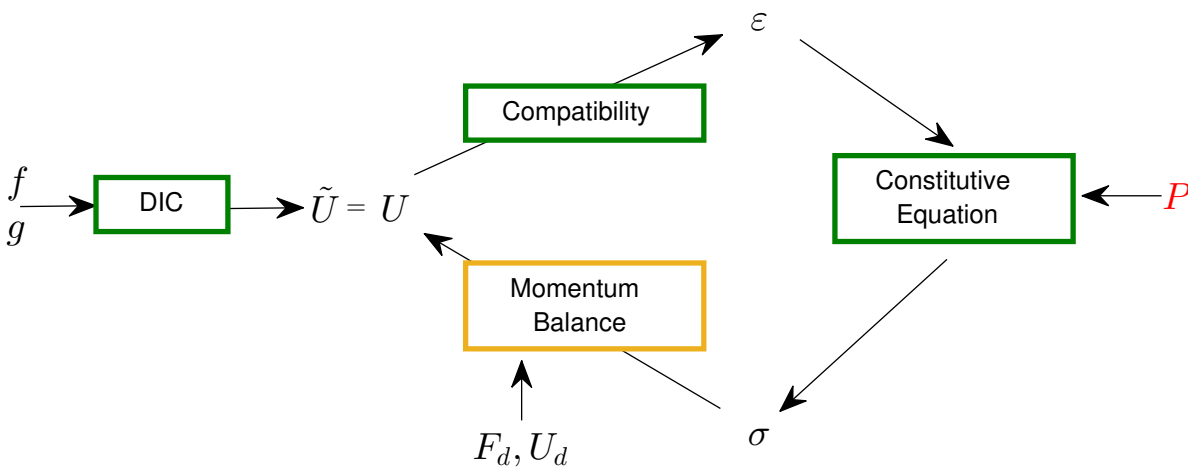


# Identification of constitutive laws

- Constitutive Equation Gap [Chrysochoos *et al.*]

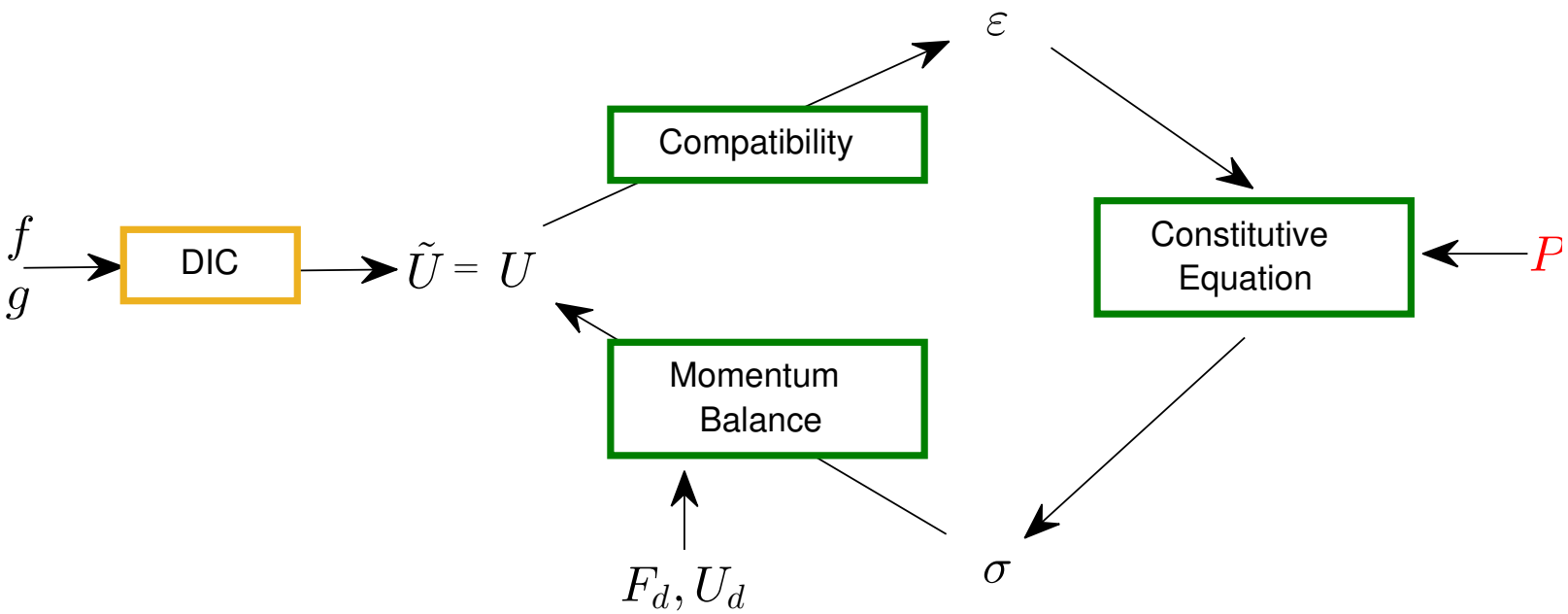



- Equilibrium [Claire et al., 2004], VFM [Grédiac et al., 2006]



# Identification of constitutive laws

- FEMU [Lecompte et al., 2007, Leclerc et al., 2009]

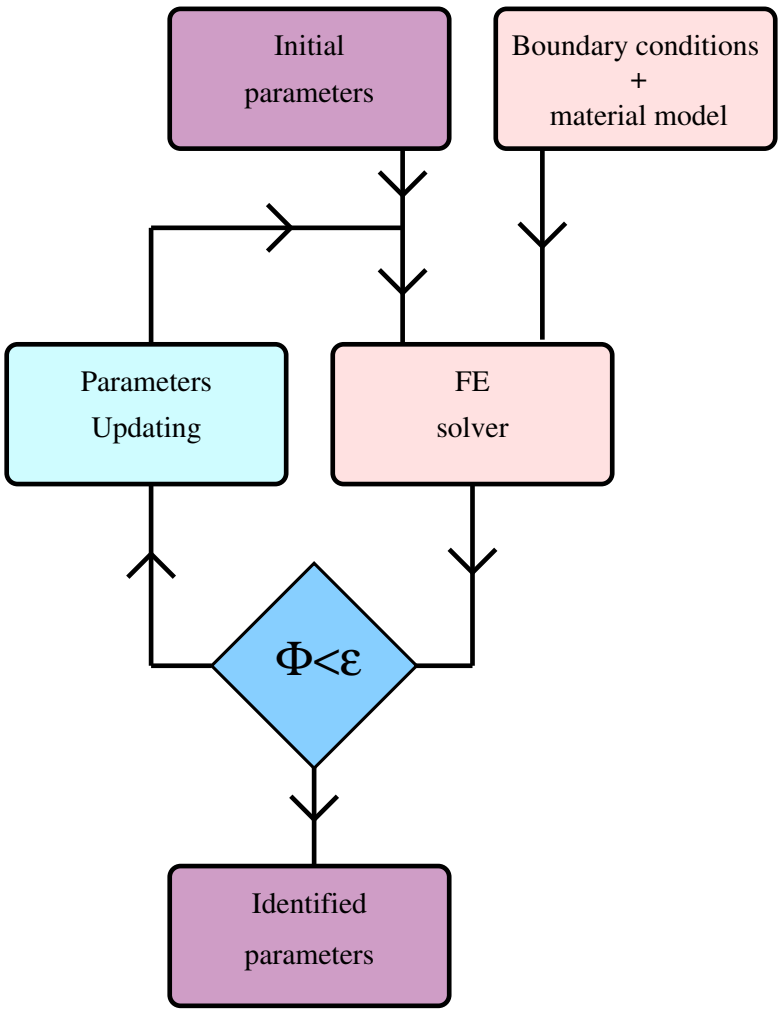



Overview in [Avril et al., 2008]

# Formulation

# FEMU

FEMU (Finite Element Model Updating) principle is the following:




$\Lambda = \{\text{set of material parameters}\}$

$$\Phi^2(\Lambda) = |F_{exp} - F_{num}(\Lambda)|^2$$

'Curve fitting' in Abaqus, Ansys, LS Dyna...

K. Kavanagh *et al.*, IJSS, 1971.

# 1D example

a linear elastic bar: length $L$, section $S$, Young modulus $E$

- analytical solution:

$$u(x) = \frac{F_{num}}{S}\frac{1}{E}x = \frac{U_{num}}{L}x$$

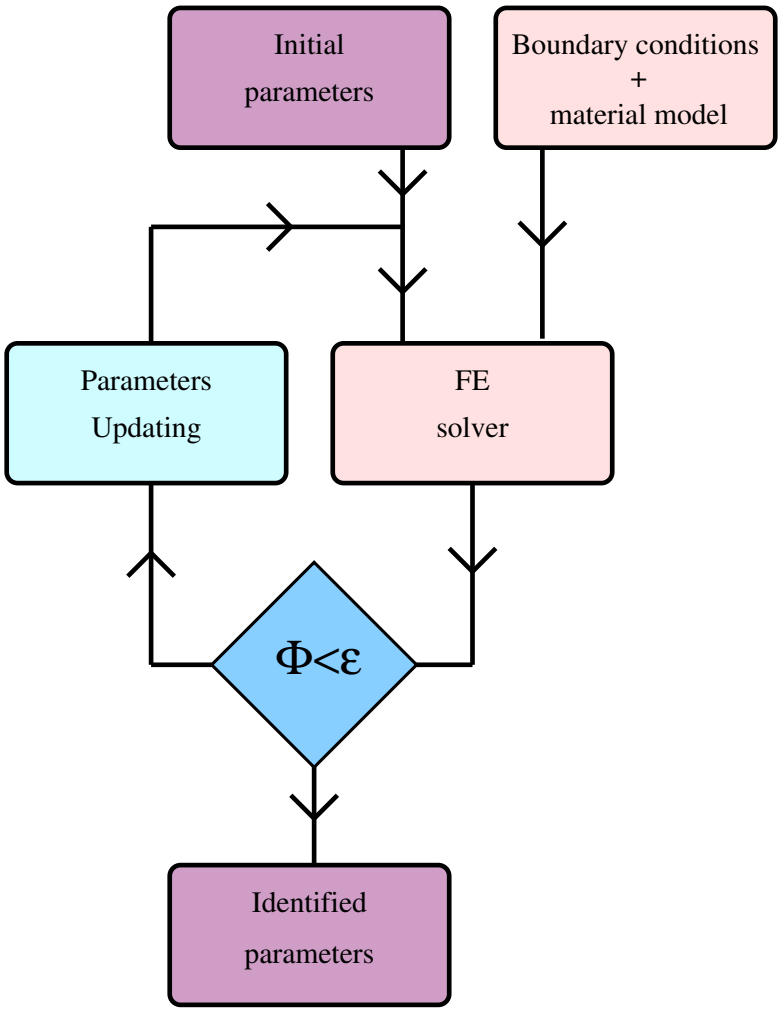
BC choice: $u(L) = U_{exp}$

- cost function $(F_{exp} - F_{num}(E))^2$
- $F_{num}(E) = \frac{ES}{L}U_{exp}$

Optimization

- Min $(F_{exp} - \frac{ES}{L}U_{exp})^2$
- $E = \frac{L}{S}\frac{F_{exp}}{U_{exp}}$
- simple because 1D linear elastic

# FEMU+DIC

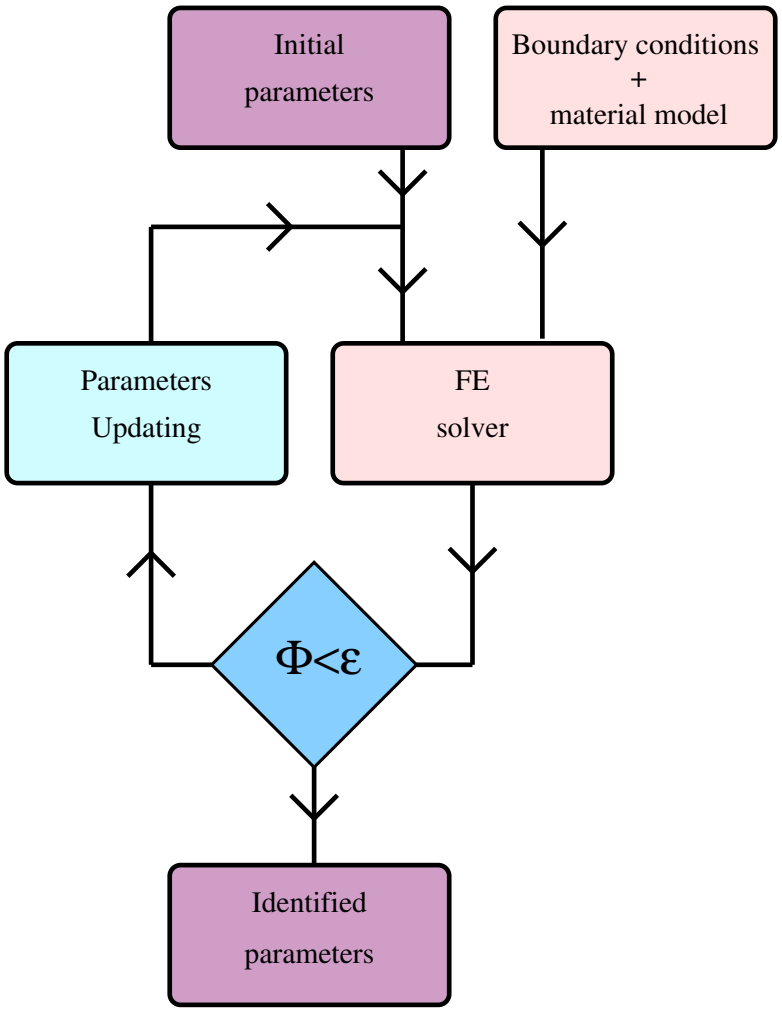



$$\Phi^2(\Lambda) = |P(\varepsilon(U_{exp})) - \varepsilon(U_{num})(\Lambda)|^2$$

P is a projection operator

Lecompte *et al.*, IJSS, 2007. Genovese *et al.*, JB, 2005.

# FEMU+FEM-DIC




$$\Phi^2(\Lambda) = |U_{exp} - U_{num}(\Lambda)|^2$$

$U_{exp}$ can be used as boundary conditions

Leclerc *et al.*, Computer Vision, 2009.

# 1D example

a linear elastic bar: length $L$, section $S$, Young modulus $E$

- analytical solution:

$$u(x) = \frac{F_{num}}{S}\frac{1}{E}x = \frac{U_{num}}{L}x$$

BC choice: $u(L) = U_{exp}$

- cost function $(U_{exp} - U_{num}(E))^2$
- $U_{num}(E) = u(L) = U_{exp}$

Optimization

- Min $(U_{exp} - U_{exp})^2$
- ill-posed, does not depend on $E$ !!!

# 1D example

a linear elastic bar: length $L$, section $S$, Young modulus $E$

- analytical solution:

$$u(x) = \frac{F_{num}}{S}\frac{1}{E}x = \frac{U_{num}}{L}x$$

BC choice: $F_{num} = F_{exp}$

- cost function $(U_{exp} - U_{num}(E))^2$
- $U_{num}(E) = \frac{F_{exp}}{S}\frac{L}{E}$

Optimization

- Min $(U_{exp} - \frac{F_{exp}}{S}\frac{L}{E})^2$
- $E = \frac{L}{S}\frac{F_{exp}}{U_{exp}}$
- simple because 1D linear elastic
- same solution as for the *force* cost function

but

- the BCs are different
- the sensitivity to $E$ are different
- stiffness *v.s.* compliance.....

# General case

Cost function:

$$\Phi^2(\Lambda) = |U_{exp} - U_{num}(\Lambda)|^2$$

Minimization: $\Lambda = \Lambda_o + d\Lambda$

- $U_{num}(\Lambda) = U_{num}(\Lambda_o) + \frac{\partial U_{num}}{\partial \Lambda} d\Lambda$
- $\frac{\partial U_{num}}{\partial \Lambda}$ is a (sensitivity) matrix when $U$ is a vector of computed/measured displacement

$$\left(\frac{\partial U_{num}}{\partial \Lambda}\right)_{ij} = \frac{\partial U_{num}(x_i)}{\partial \Lambda_j}$$

- parameters increment

$$\left(\frac{\partial U_{num}}{\partial \Lambda}\right)^T \left(\frac{\partial U_{num}}{\partial \Lambda}\right) d\Lambda = \left(\frac{\partial U_{num}}{\partial \Lambda}\right)^T (U_{exp} - U_{num}(\Lambda_o))$$

- $\left(\frac{\partial U_{num}}{\partial \Lambda}\right)^T \left(\frac{\partial U_{num}}{\partial \Lambda}\right)$ Hessian matrix
- in practice $\frac{\partial U_{num}}{\partial \Lambda}$ is computed using finite differences

# Noise sensitivity

Any measurement is corrupted by noise. Quantifying the impact of noise on the "mesurandes" is important. Trying to minimize the influence of noise is a long route....
Let's suppose that the displacement vector $\mathbf{U}$ is affected by a spatially uncorrelated noise:

$$\mathbf{U} = \mathbf{U} + \delta\mathbf{U} \text{ with } \langle\delta\mathbf{U}\rangle = 0 \quad \langle\delta\mathbf{U}.\delta\mathbf{U}^T\rangle = \sigma_u^2\mathbf{I}$$

$\langle . \rangle$ means average over the realizations. In the case of a Least-Squares projection,

$$\mathbf{L}^T\mathbf{L}\mathbf{P} = \mathbf{L}^T\mathbf{U}$$

the identified parameters $\mathbf{P}$ is thus affected by a perturbation $\delta\mathbf{P} = (\mathbf{L}^T\mathbf{L})^{-1}\mathbf{L}^T\delta\mathbf{U}$.
The features of this noise are:

$$\langle\delta\mathbf{P}\rangle = (\mathbf{L}^T\mathbf{L})^{-1}\mathbf{L}^T\langle\delta\mathbf{U}\rangle = 0 \quad \langle\delta\mathbf{P}.\delta\mathbf{P}^T\rangle = (\mathbf{L}^T\mathbf{L})^{-1}\mathbf{L}^T\langle\delta\mathbf{U}\delta\mathbf{U}^T\rangle VL(\mathbf{L}^T\mathbf{L})^{-1} = \sigma_u^2(\mathbf{L}^T\mathbf{L})^{-1}$$

The inverse of the Hessian matrix $\mathbf{L}^T\mathbf{L}$ is thus the kernel (the covariance) of the perturbation affecting the identified parameters [Réthoré, 2010].
Qualitatively, the low value in the Hessian matrix leads to high noise sensitivity.....conditioning is thus a crucial issue !

# Implementation

# FEMU_introduction

October 5, 2025

## 1 FEMU Introduction

J. Réthoré / R. Seghir, 01/2021

We start with a simple example that can be computed without using FEM..... This is a simple plate under uniaxial tension. The problem definition is the following: - the plate's dimension are $L$, $h$ and $e$ along $\mathbf{x}$, $\mathbf{y}$, and $\mathbf{z}$ direction respectively - the loading is a force applied on the $x = L$ edge along the $\mathbf{x}$ direction - the $\mathbf{x}$ displacement is fixed on the $x = 0$ edge - the $\mathbf{y}$ displacement is 0 at the lower left corner $ (x,y)=(0,0)$ - the material is supposed to be isotropic with a linear elastic behaviour - the coefficients defining this behaviour are the Young modulus $E$ and the Poisson ratio $\nu$

Under these assumptions, an analytical solution can be obtained. The displacement solution is - $u_x = \frac{F}{Ehe}x$ - $u_y = -\nu\frac{F}{Ehe}y$

The following steps will be followed: - first the analytical solution with the reference material parameters is computed. - then the sensitivity of this solution to the material parameters is computed together with the covariance matrix of FEMU - then a FEMU is performed to retrieve the material parameters of the reference solution from a given initial guess

[11]:
```
# Chargement des librairies
import numpy as np
import matplotlib.pyplot as plt
import scipy
from scipy import ndimage
```

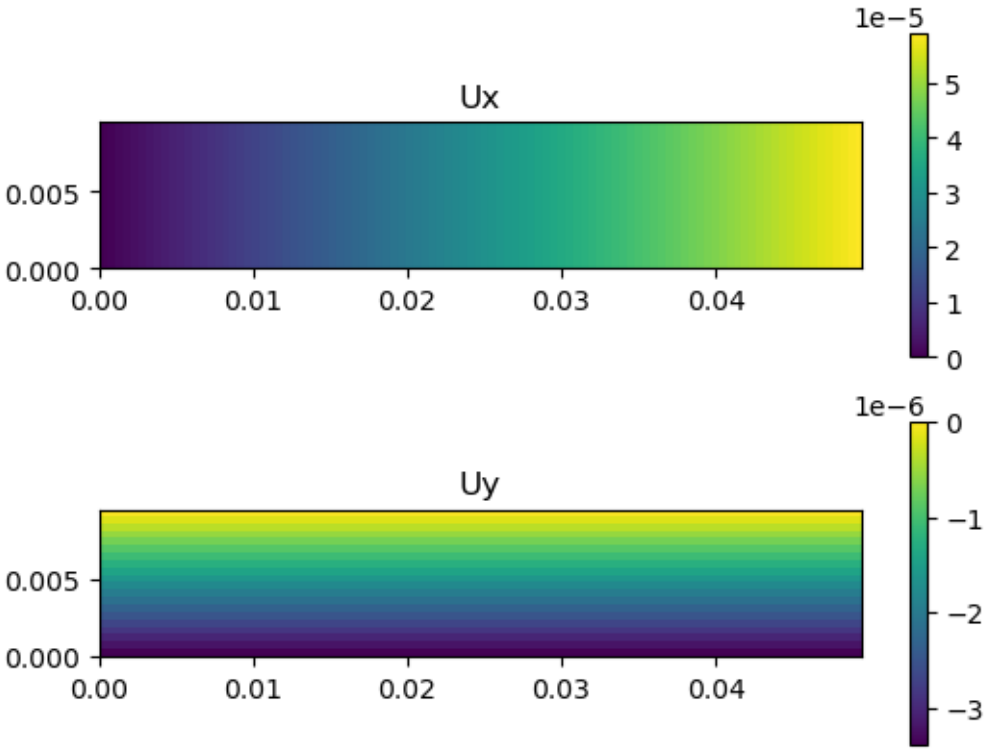
[12]:
```
# geometry
L = 5e-2
h = 1e-2
e = 1e-3
dx=0.01*L
stdu=L/1000/10
# material parameters
E = 210.e9
nu = 0.3

# loading
F = 2.5e3
print('Corresponding axial strain' , F/(e*h)/E)
```

```
print('Maximum axial displacement', F/(e*h)/E*L, ' m')
print('Standard deviation of the displacement noise' , stdu, ' m')
```

```
Corresponding axial strain 0.0011904761904761904
Maximum axial displacement 5.9523809523809524e-05  m
Standard deviation of the displacement noise 5e-06  m
```

### 1.1 Reference solution

[13]:
```
# point coordinates
xm,ym = np.meshgrid(np.arange(0,L,dx),np.arange(0,h,dx))
sizeim = np.shape(xm)


uxm = F/(h*e)/E*xm
uym = -nu*F/(h*e)/E*ym


plt.subplot(211)
plt.imshow(uxm, extent=(np.amin(xm), np.amax(xm), np.amin(ym), np.amax(ym)),␣
 ↪aspect = 'equal')
plt.title('Ux')
plt.colorbar()
plt.subplot(212)
plt.imshow(uym, extent=(np.amin(xm), np.amax(xm), np.amin(ym), np.amax(ym)),␣
 ↪aspect = 'equal')
plt.title('Uy')
plt.colorbar();
```



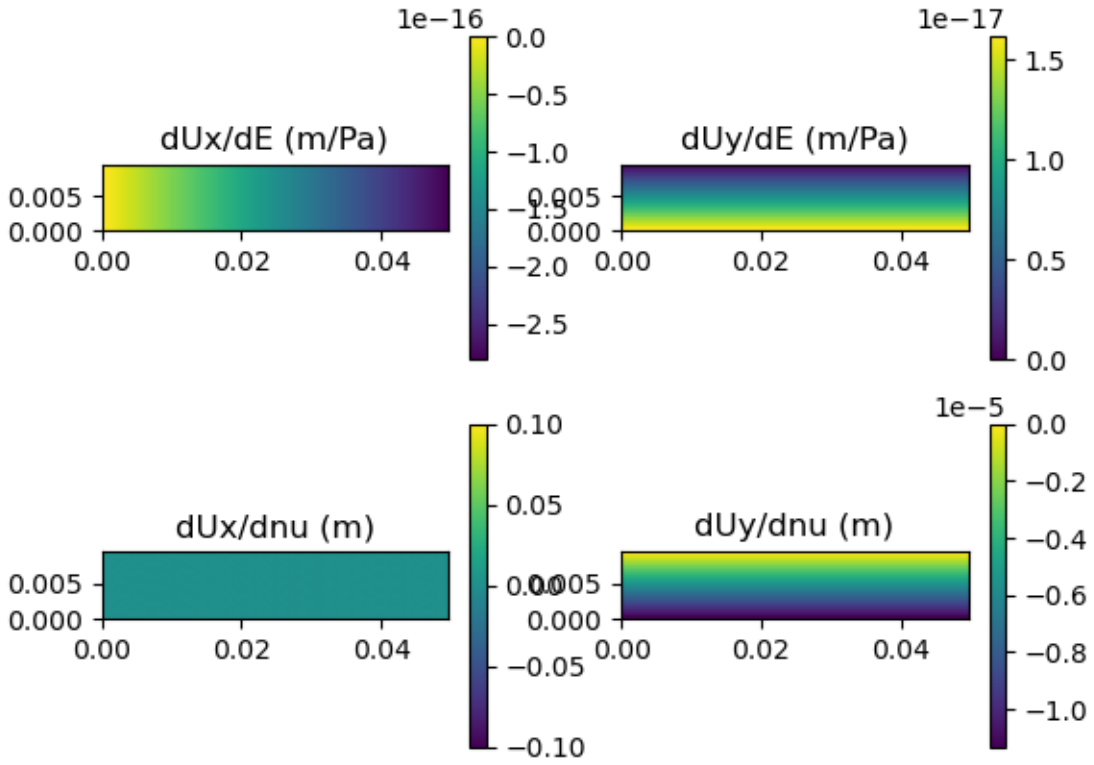

### 1.2 Sensitivity

From this analytical solution, it is possible to compute the sensitivity of the displacement with respect to the material parameters, *i.e.* the derivative of the displacement with respect to the material parameters: - $\frac{\partial u_x}{\partial E} = -\frac{F}{E^2he}x$ - $\frac{\partial u_y}{\partial E} = \nu\frac{F}{E^2he}y$ - $\frac{\partial u_x}{\partial \nu} = 0$ - $\frac{\partial u_y}{\partial \nu} = -\frac{F}{Ehe}y$

[14]:
```
#sensitivity fields
duxdE = -F/(h*e)/pow(E,2)*xm
duydE = nu*F/(h*e)/pow(E,2)*ym
duxdnu = 0*xm
duydnu = -F/(h*e)/E*ym

plt.subplot(221)
plt.imshow(duxdE, extent=(np.amin(xm), np.amax(xm), np.amin(ym), np.amax(ym)),␣
 ↪aspect = 'equal')
plt.title('dUx/dE (m/Pa)')
plt.colorbar()
plt.subplot(222)
plt.imshow(duydE, extent=(np.amin(xm), np.amax(xm), np.amin(ym), np.amax(ym)),␣
 ↪aspect = 'equal')
plt.title('dUy/dE (m/Pa)')
plt.colorbar()
```

```
plt.subplot(223)
plt.imshow(duxdnu, extent=(np.amin(xm), np.amax(xm), np.amin(ym), np.amax(ym)),␣
↪aspect = 'equal')
plt.title('dUx/dnu (m)')
plt.colorbar()
plt.subplot(224)
plt.imshow(duydnu, extent=(np.amin(xm), np.amax(xm), np.amin(ym), np.amax(ym)),␣
↪aspect = 'equal')
plt.title('dUy/dnu (m)')
plt.colorbar()
```

[14]: <matplotlib.colorbar.Colorbar at 0x7decfbf29df0>




### 1.3 FEMU

The cost function

$\phi(\mathbf{P}) = \int ||\mathbf{u}^{DIC}(\mathbf{x}) - \mathbf{u}^{FEM}(\mathbf{x}, \mathbf{P})||^2 \mathbf{dx}$

is to be minimized with respect to $\mathbf{P}$, a vector collecting the value of the $n_p$ parameters to be identified.

In the general case, there is non-linear dependence of the FEM solution upon $\mathbf{P}$. An iterative searched is initiated from an initial guess of the parameters value. At iteration *iter*, a solution

increment $\mathbf{dP} = \mathbf{P} - \mathbf{P}_o$ is searched for, $\mathbf{P}_o$ being the solution at the previous iteration.

A linearization of the problem is adopted:

$\mathbf{u}^{FEM}(\mathbf{x}, \mathbf{P}) = \mathbf{u}^{FEM}(\mathbf{x}, \mathbf{P}_o + \mathbf{dP}) = \mathbf{u}^{FEM}(\mathbf{x}, \mathbf{P}_o) + \frac{\partial \mathbf{u}^{FEM}}{\partial \mathbf{P}}(\mathbf{x}, \mathbf{P}_o)\ \mathbf{dP}$

The derivatives of the displacement with respect to the parameters is called the sensitivity. Then we obtain for the cost function

$\phi(\mathbf{P}_o + \mathbf{dP}) = \int ||\mathbf{u}^{DIC}(\mathbf{x}) - \mathbf{u}^{FEM}(\mathbf{x}, \mathbf{P}_o) - \frac{\partial \mathbf{u}^{FEM}}{\partial \mathbf{P}}(\mathbf{x}, \mathbf{P}_o)\ \mathbf{dP}||^2 \mathbf{dx}$

Developing the square one gets

$\phi(\mathbf{P}_o + \mathbf{dP}) = \int ||\mathbf{u}^{DIC}(\mathbf{x}) - \mathbf{u}^{FEM}(\mathbf{x}, \mathbf{P}_o)||^2 \mathbf{dx} - 2 \int (\mathbf{u}^{DIC}(\mathbf{x}) - \mathbf{u}^{FEM}(\mathbf{x}, \mathbf{P}_o)).\frac{\partial \mathbf{u}^{FEM}}{\partial \mathbf{P}}(\mathbf{x}, \mathbf{P}_o)\mathbf{dx}\ \mathbf{dP} + \mathbf{dP}^T \int (\frac{\partial \mathbf{u}^{FEM}}{\partial \mathbf{P}}(\mathbf{x}, \mathbf{P}_o))^T.(\frac{\partial \mathbf{u}^{FEM}}{\partial \mathbf{P}}(\mathbf{x}, \mathbf{P}_o))\mathbf{dx}\ \mathbf{dP}$

Then the stationnarity of the cost function with respect to is searched for

$\frac{\partial \phi}{\partial P} = 0 \rightarrow \int (\frac{\partial \mathbf{u}^{FEM}}{\partial \mathbf{P}}(\mathbf{x}, \mathbf{P}_o))^T.(\frac{\partial \mathbf{u}^{FEM}}{\partial \mathbf{P}}(\mathbf{x}, \mathbf{P}_o))\mathbf{dx}\ \mathbf{dP} = \int (\mathbf{u}^{DIC}(\mathbf{x}) - \mathbf{u}^{FEM}(\mathbf{x}, \mathbf{P}_o)).\frac{\partial \mathbf{u}^{FEM}}{\partial \mathbf{P}}(\mathbf{x}, \mathbf{P}_o)\mathbf{dx}$

This procedure is repeated until convergence is reached *i.e.* **dP** is small.

### 1.4 Numerical integration

In pratice the integration over the plate is performed using a discrete sum over sampling points (the number of points is $n$). Each of these points has a weight $w$ that is the areea of the plate divided by the number of points: $w = (h * L)/n$.

The values of the displacement are then collected in matrices having $n$ lines and 2 columns: $\mathbf{U} = \begin{bmatrix} u_x^1, u_y^1 \\ - \\ \vdots \\ - \\ u_x^n, u_y^n \end{bmatrix}_{(n,2)}$.

The third dimension refers to the parameters. This means that the sensitivity $\frac{\partial \mathbf{u}^{FEM}}{\partial \mathbf{P}}$ recasts as a matrix with 3 indices refering to the point id, the component and the parameters:

$(\frac{\partial \mathbf{U}^{FEM}}{\partial \mathbf{P}})_{i,j,k} = \frac{\partial u_j^{FEM}}{\partial P_k}(\mathbf{x}_i)$

Then the system of $n_p$ equations indexed by $i$ resulting from the stationarity condition of the cost function is

$\sum_{p=1}^{n} \sum_{c=1}^{2} \sum_{j=1}^{n_p} \frac{\partial u_c^{FEM}}{\partial P_i}(\mathbf{x}_p) \frac{\partial u_c^{FEM}}{\partial P_j}(\mathbf{x}_p) w dP_j = \sum_{p=1}^{n} \sum_{c=1}^{2} \frac{\partial u_c^{FEM}}{\partial P_i}(\mathbf{x}_p)(u_c^{DIC}(\mathbf{x}_p) - u_c^{FEM}(\mathbf{x}_p))w$

Using the above defined matrices, we have

$\sum_{p=1}^{n} \sum_{c=1}^{2} \sum_{j=1}^{n_p} (\frac{\partial \mathbf{U}^{FEM}}{\partial \mathbf{P}})_{p,c,i} (\frac{\partial \mathbf{U}^{FEM}}{\partial \mathbf{P}})_{p,c,j} w dP_j = \sum_{p=1}^{n} \sum_{c=1}^{2} (\frac{\partial \mathbf{U}^{FEM}}{\partial \mathbf{P}})_{p,c,i} (U_{p,c}^{DIC} - U_{p,c}^{FEM}) w$

This can be written in a compact form $\mathbf{HdP} = \mathbf{b}$ with

$H_{ij} = \sum_{p=1}^{n} \sum_{c=1}^{2} (\frac{\partial \mathbf{U}^{FEM}}{\partial \mathbf{P}})_{p,c,i} (\frac{\partial \mathbf{U}^{FEM}}{\partial \mathbf{P}})_{p,c,j} w \ \ , \ \ b_i = \sum_{p=1}^{n} \sum_{c=1}^{2} (\frac{\partial \mathbf{U}^{FEM}}{\partial \mathbf{P}})_{p,c,i} (U_{p,c}^{DIC} - U_{p,c}^{FEM}) w$

**H** is called the Hessian matrix.

### 1.5 Computation of the covariance matrix

[15]:
```
#Hessian matrix
x = np.reshape(xm,sizeim[0]*sizeim[1],order='F')
y = np.reshape(ym,sizeim[0]*sizeim[1],order='F')

u=np.array([F/(h*e)/(E)*x,-(nu)*F/(h*e)/(E)*y]).T
dudp=np.array([[-F/(h*e)/pow(E,2)*x,nu*F/(h*e)/pow(E,2)*y],[0*x,-F/(h*e)/E*y]]).
 ↪T
w=L*h/np.shape(x)[0]


def hessian(dudp,w):
    import numpy as np
    C = np.zeros((np.size(dudp,2),np.size(dudp,2)))
    for ii in range(np.size(dudp,2)):
        for jj in range(np.size(dudp,2)):
            for cc in range(np.size(dudp,1)):
                for pp in range(np.size(dudp,0)):
                    C[ii][jj]=C[ii][jj]+(dudp[pp,cc,ii]*dudp[pp,cc,jj])*w
    return C

H=hessian(dudp,w)

print('H = \n', H)
```

```
H =
 [[ 1.32348043e-35 -3.12550615e-26]
 [-3.12550615e-26  2.18785431e-14]]
```

### 1.6 Remark

- *Conditioning ?*
- *Units ?*

[16]:
```
# compute the eigen values
eigVal,eigVec = np.linalg.eig(H)
print('Eigen values of H = \n', eigVal)
```

```
Eigen values of H =
 [0.00000000e+00 2.18785431e-14]
```

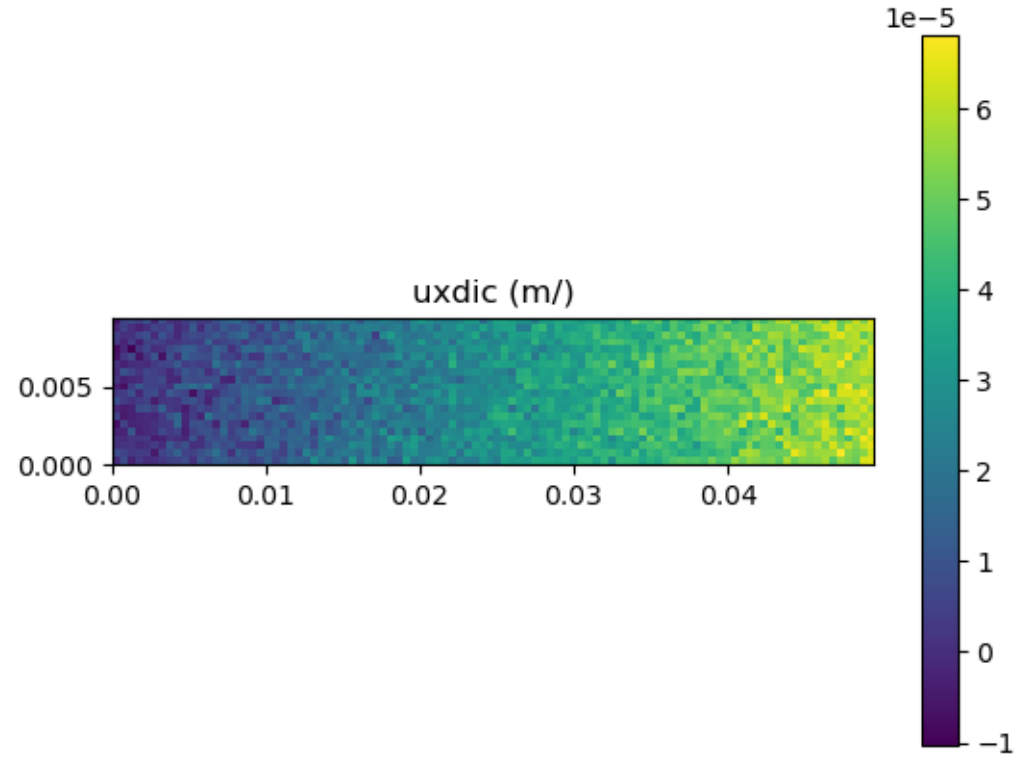
### 1.7 Test case

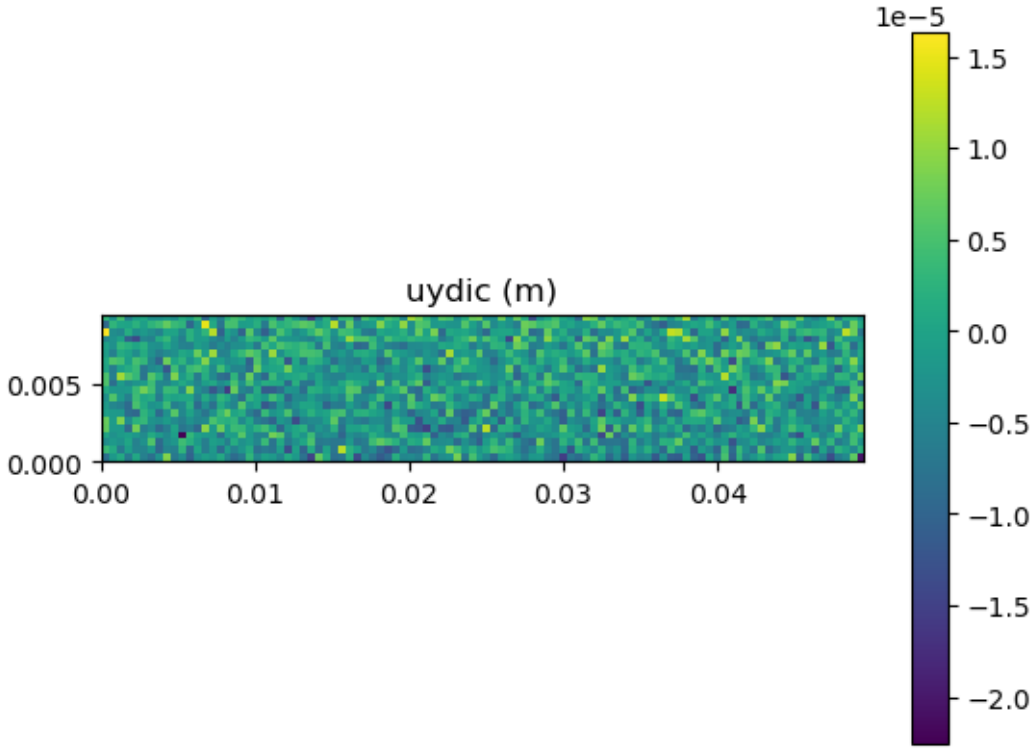
FEMU is run adding a noise to the reference solution computed earlier to simulate DIC and starting from parameters different from those used for the reference solution $(E_o, \nu_o)$. Due to the problem mentioned above, what is actually searched for are multiplying coefficients to the initial values of the parameters. This means that the value for the Young modulus for the current iteration is $E_o * P(0)$.

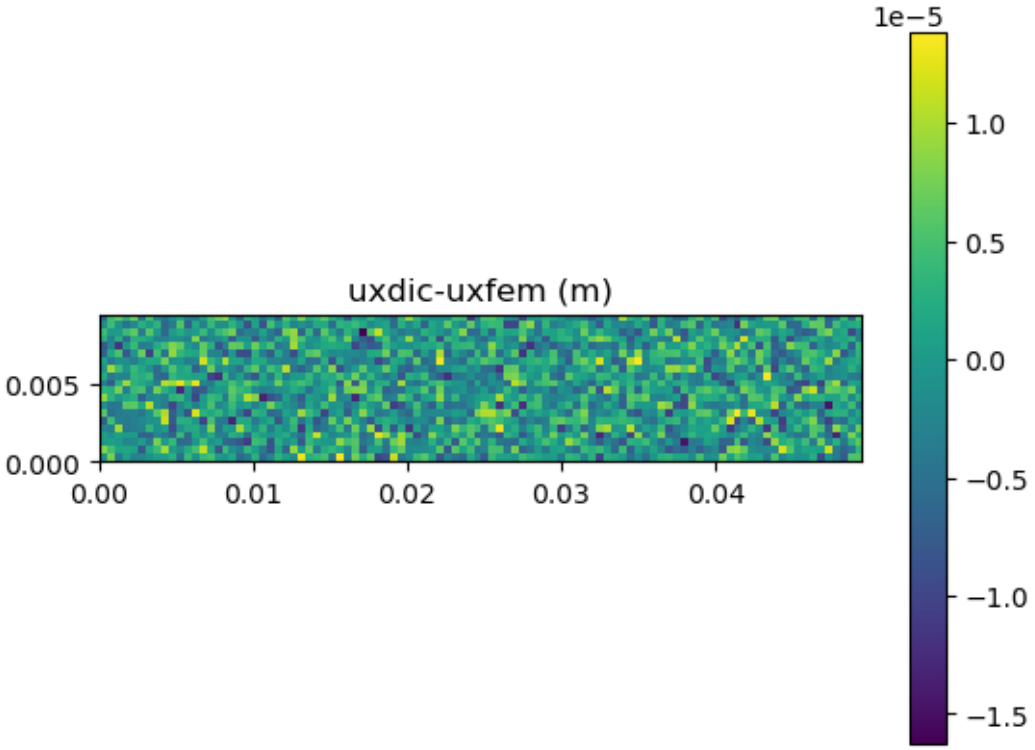
[17]:
```
# initial material parameters
Eo = 200.e9
nuo = 0.25
du=1*stdu*np.random.randn(x.shape[0],2)
udic=u+du
P= np.array([1. ,1.])
b= np.array([0. ,0.])

def sensitivity(x,y,S,E,nu,P):
    import numpy as np
    u=np.array([S/(E*P[0])*x,-(nu*P[1])*S/(E*P[0])*y]).T
    dudp=np.array([[-S/E/pow(P[0],2)*x,nu*S/E/pow(P[0],2)*y],
                   [0*x,-nu*S/(E*P[0])*y]]).T
    return (u, dudp)

def residual(dudp,r,w):
    import numpy as np
    b = np.zeros((np.size(dudp,2)))
    for ii in range(np.size(dudp,2)):
        for cc in range(np.size(dudp,1)):
            for pp in range(np.size(u,0)):
                b[ii]=b[ii]+(dudp[pp,cc,ii]*r[pp,cc])*w
    return b
iter=0

while (iter < 5):
    (ui,dudpi)= sensitivity(x,y,F/(h*e),Eo,nuo,P)
    H=hessian(dudpi,w)

    Hinv=np.linalg.inv(H)
    #print('Ci = \n',Cinv)
    b=residual(dudpi,udic-ui,w)
    #print('b  = \n',b)

    dP=np.dot(Hinv,b)
    print('dP = \n',dP)
    P=P+dP
    #print('P = \n',P)
    iter=iter+1
print('H ' ,H)
eigVal,eigVec = np.linalg.eig(H)
print('Eigen values of H = \n', eigVal)
Ei=Eo*P[0]
nui=nuo*P[1]
print('E identif' , Ei,' nu identif', nui)
print('relative errors on E', (E-Ei)/E, ' on nu', (nu-nui)/nu)
```

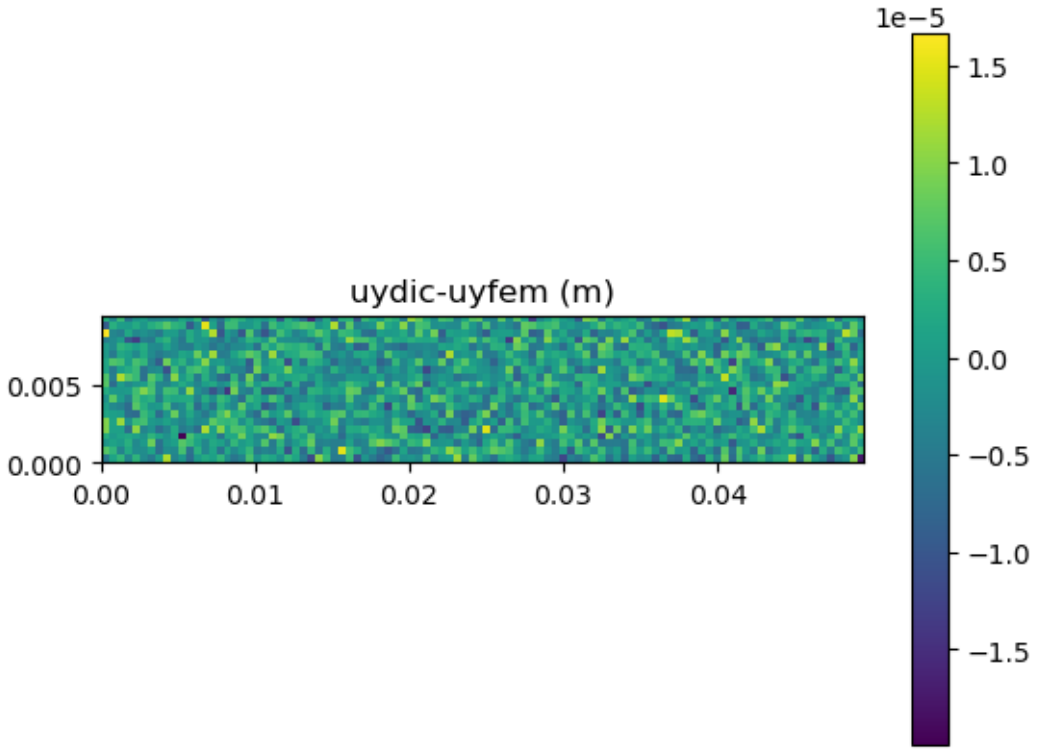
```
uxdic= np.reshape(udic[:,0],[sizeim[0],sizeim[1]],order='F')
uydic= np.reshape(udic[:,1],[sizeim[0],sizeim[1]],order='F')
uxfem= np.reshape(ui[:,0],[sizeim[0],sizeim[1]],order='F')
uyfem= np.reshape(ui[:,1],[sizeim[0],sizeim[1]],order='F')

#plt.subplot(411)
ff = plt.figure()
plt.imshow(uxdic, extent=(np.amin(xm), np.amax(xm), np.amin(ym), np.amax(ym)),␣
 ↪aspect = 'equal')
plt.title('uxdic (m/)')
plt.colorbar()
#plt.subplot(412)
ff = plt.figure()
plt.imshow(uydic, extent=(np.amin(xm), np.amax(xm), np.amin(ym), np.amax(ym)),␣
 ↪aspect = 'equal')
plt.title('uydic (m)')
plt.colorbar()
#plt.subplot(413)
ff = plt.figure()
plt.imshow(uxdic-uxfem, extent=(np.amin(xm), np.amax(xm), np.amin(ym), np.
 ↪amax(ym)), aspect = 'equal')
plt.title('uxdic-uxfem (m)')
plt.colorbar()
#plt.subplot(414)
ff = plt.figure()
plt.imshow(uydic-uyfem, extent=(np.amin(xm), np.amax(xm), np.amin(ym), np.
 ↪amax(ym)), aspect = 'equal')
plt.title('uydic-uyfem (m)')
plt.colorbar();
```

```
dP =
 [0.04526581 0.10972131]
dP =
 [0.00214174 0.00496662]
dP =
 [4.39740969e-06 2.34994844e-04]
dP =
 [1.84619925e-11 4.82489540e-07]
dP =
 [4.65794693e-17 2.02550746e-12]
H  [[ 5.34092755e-13 -1.31197161e-15]
 [-1.31197161e-15  1.37417475e-15]]
Eigen values of H =
 [5.34095986e-13 1.37094366e-15]
E identif 209482390308.29044  nu identif 0.2787308521948414
relative errors on E 0.0024648080557598297  on nu 0.07089715935052865
```


uxdic (m/)
1e−5
6
5
4
3
2
1
0
−1
0.005
0.000
0.00
0.01
0.02
0.03
0.04


uydic (m)
1e−5
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
−2.0
0.005
0.000
0.00
0.01
0.02
0.03
0.04


uxdic-uxfem (m)
0.005
0.000
0.00
0.01
0.02
0.03
0.04
1e−5
1.0
0.5
0.0
−0.5
−1.0
−1.5




### 1.8 Remarks

- *Conditioning ?*: much better with the relative identification strategy
- *Accuracy ?*
- *Robustness ?*
- The covariance matrix usually depand on the material parameters, and should be updated

### 1.9 Alternative resolution strategy

In the previous cell a *full* Gauss-Newton algorithm is used. Alernatively, a fixed point or staggered algorithm could be used. This somehow simply consists in 'lumping' the Hessian matrix, *i.e.* neglecting the coupling terms.

As aconsequence, at each iteration the parameters increments are obtained *explicitly*:

$\mathbf{H}_{ii}\mathbf{dP}_i = \mathbf{b}_i$

[18]:
```
P= np.array([1. ,1.])
dP= np.array([0. ,0.])
iter=0
while (iter < 5):
    (ui,dudpi)= sensitivity(x,y,F/(h*e),Eo,nuo,P)
    H=hessian(dudpi,w)
```

```
    #print('Ci = \n',Cinv)
    b=residual(dudpi,udic-ui,w)
    #print('b  = \n',b)
    for ii in range(np.size(dudp,2)):
        dP[ii]=b[ii]/H[ii,ii]
    print('dP = \n',dP)
    P=P+dP
    #print('P = \n',P)
    iter=iter+1
Ei=Eo*P[0]
nui=nuo*P[1]
print('E identif' , Ei,' nu identif', nui)
print('relative errors on E', (E-Ei)/E, ' on nu', (nu-nui)/nu)
```

```
dP =
 [0.04500849 0.0644555 ]
dP =
 [0.00227491 0.04790953]
dP =
 [0.00012229 0.00242154]
dP =
 [5.93040285e-06 1.30173683e-04]
dP =
 [3.18201994e-07 6.31264991e-06]
E identif 209482387050.03522  nu identif 0.2787307631815414
relative errors on E 0.00246482357126087  on nu 0.07089745606152868
```

The main difference is on the convergence speed on such a *simple* problem. For more complex constitutive law to identify it may cause additionnal difficulties.

### 1.10 Reformulation for improving conditioning

The conditioning problem mentionned earlier are due to: - the Young modulus is a denominator - Young modulus and Poisson ratio don't have the same unit To improve the conditioning, the Hooke law

$\epsilon = \frac{1+\nu}{E}\sigma - \frac{\nu}{E}\mathrm{tr}(\sigma)$

is parametrized a bit differently. Instead of identifying $(E, \nu)$ as intrinsic parameters, one can try to identify $(\frac{1+\nu}{E}, \frac{\nu}{E})$.

[19]:
```
P= np.array([(1.+nuo)/Eo ,nuo/Eo])
b= np.array([0. ,0.])

def sensitivity2(x,y,S,E,nu,P):
    import numpy as np
    u=np.array([S*(P[0]-P[1])*x,-(P[1])*S*y]).T
    dudp=np.array([[S*x,0*y],[-S*x,-S*y]]).T
    return (u, dudp)
```

```
iter=0

while (iter < 5):
    (ui,dudpi)= sensitivity2(x,y,F/(h*e),Eo,nuo,P)
    H=hessian(dudpi,w)

    Hinv=np.linalg.inv(H)
    #print('Ci = \n',Cinv)
    b=residual(dudpi,udic-ui,w)
    #print('b  = \n',b)

    dP=np.dot(Hinv,b)
    print('dP = \n',dP)
    P=P+dP
    #print('P = \n',P)
    iter=iter+1
print('H ' ,H)

eigVal,eigVec = np.linalg.eig(H)
print('Eigen values of H = \n', eigVal)
Ei=1/(P[0]-P[1])
nui=(P[1])*Ei
#Ei=1/((1+nuo)/Eo*P[0]-nuo/Eo*P[1])
#nui=(nuo/Eo*P[1])*Ei
print('E identif' , Ei,' nu identif', nui)
print('relative errors on E', (E-Ei)/E, ' on nu', (nu-nui)/nu)
```

```
dP =
 [-1.45759685e-13  8.05693705e-14]
dP =
 [-1.96023526e-27 -2.62842211e-27]
dP =
 [-2.03774656e-28  7.02317197e-30]
dP =
 [-2.03774656e-28  7.02317197e-30]
dP =
 [-2.03774656e-28  7.02317197e-30]
H  [[ 2.56523438e+10 -2.56523438e+10]
 [-2.56523438e+10  2.66171875e+10]]
Eigen values of H =
 [4.77886026e+08 5.17916452e+10]
E identif 209482390308.29047  nu identif 0.27873085219484145
relative errors on E 0.0024648080557596844  on nu 0.07089715935052847
```

- The conditioning is improved in the same way as when using the relative strategy.
- A good parametrization is one of the key

# Application to FE data

# FEMU_from_FEM

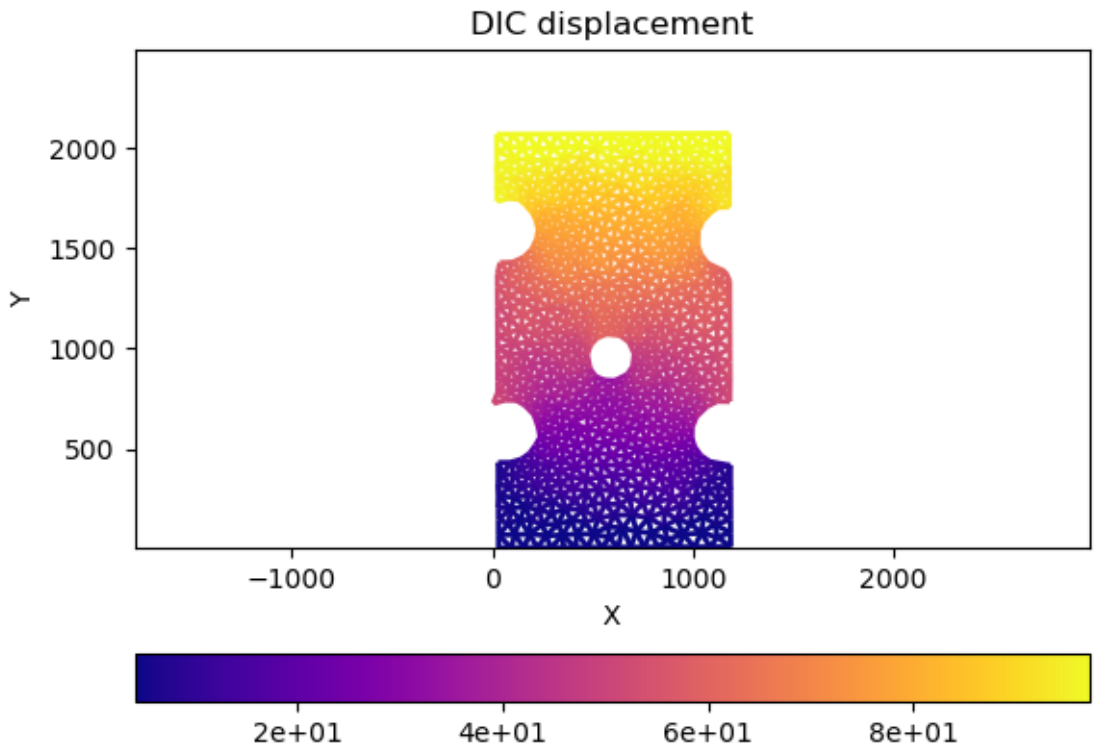
October 5, 2025

[30]:
```
#### Import Libraries
import numpy as np
import matplotlib.pyplot as plt
import scipy
from scipy import ndimage
from scipy.sparse import csr_matrix as smatrix
import scipy.sparse.linalg as splinalg
import os
import sys
import fem
```

[31]:
```
# Units m->pixel=m/pix2m
# Pa=kg/m/s^2->kg/pixel/s^2=Pa*pix2m
# N = kg.m/s^2->kg.pixel/s^2=N/pix2m
# N/m = kg/s^2->kg/s^2=N/m
# Parameters
Ns=10 # Number of loading steps in the FE simulations
pix2m=25.e-6; # pixel to m conversion
thickness=3e-3 # specimen thicness in m
stdu=0.01 # amplitude of the noise on displacement

# Loading FE model and displacement data
npz=np.load('fem-from-dic.npz')
Uref=npz['U']
dUref=max(abs(Uref[:]))*stdu*np.random.randn(Uref.size)
Uref+=dUref
Fres=npz['Fres']/thickness
inp='dic-coarse.res'
(X,conn)=fem.readDICmesh(inp)
model=fem.FEModel()
model.X=X
model.conn=conn
model.Assemble()
W=model.W
B=model.B
```

```
Xg=0.5*(X[conn[:,0]]+X[conn[:,1]])
Nnodes=X.shape[0]
Nelems=conn.shape[0]
fig, ax = plt.subplots()
model.show_field(fig, ax,Uref[Nnodes::], name = "DIC displacement")
```

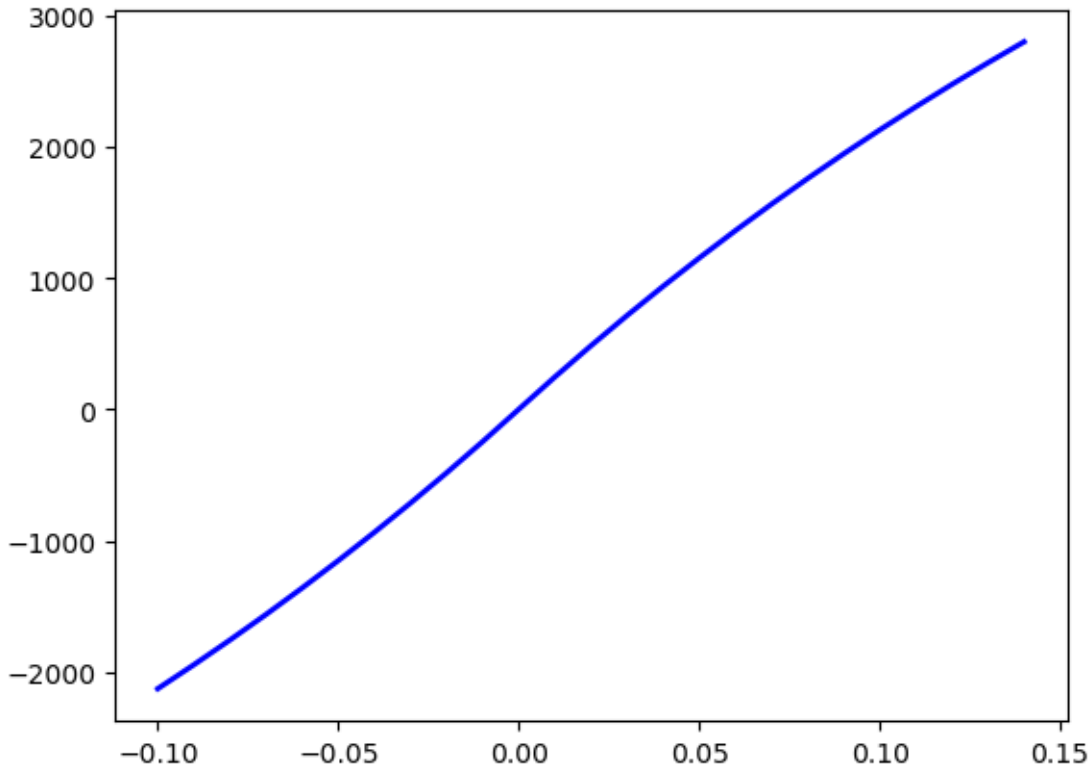
[31]:
```
(<matplotlib.collections.LineCollection at 0x7247097001d0>,
 <matplotlib.colorbar.Colorbar at 0x724709b03e90>)
```




The material constitutive law is non-linear elastic given by $\sigma = K(1 - exp(-|\epsilon|)/\epsilon_o))sign(\epsilon)$

[32]:
```
# Setting the initial material model
Kini=3e8
Eoini=0.3
mat=fem.MatModel(pix2m)
mat.K=Kini
mat.Eo=Eoini
model.material=mat
epsi=np.arange(-0.1,0.15,0.01)
stress=mat.GetStress(epsi)
ff=plt.figure()
plt.plot(epsi,stress/pix2m*1.e-6,'b-', linewidth=2)
```

[32]: [<matplotlib.lines.Line2D at 0x7247080c93a0>]

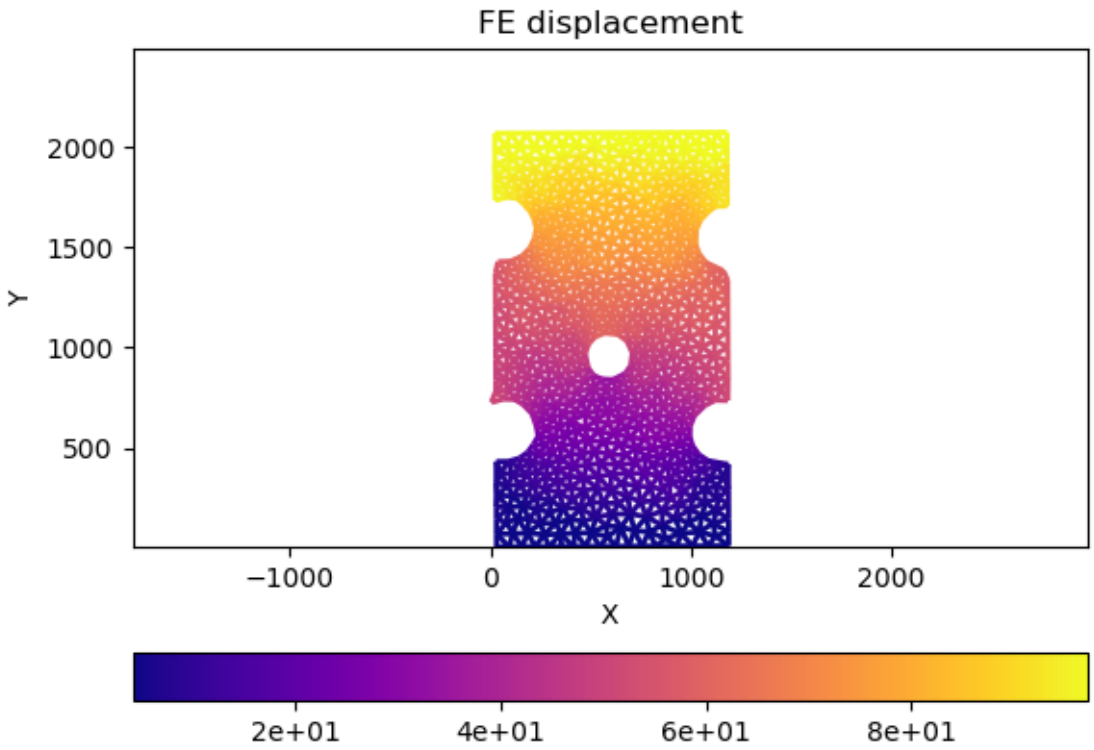



[33]:
```
# Boundary conditions for the FE simulation
# Displacement control -> ux,uy from DIC on bottom, ux,uy from DIC on top
# Load control -> ux,uy from DIC on bottom, ux from DIC on top, Fy on top
top=X[:,1]>max(X[:,1])*0.99
bot=X[:,1]<max(X[:,1])*0.01

nodes_index=np.arange(Nnodes)
top_nodes=nodes_index[top]
bot_nodes=nodes_index[bot]
const_ddls_f=np.r_[top_nodes,bot_nodes,bot_nodes+Nnodes]
const_ddls_u=np.r_[top_nodes,top_nodes+Nnodes,bot_nodes,bot_nodes+Nnodes]

Fexto=np.zeros(2*Nnodes)
Fext=np.zeros(2*Nnodes)
Uimp=np.zeros(2*Nnodes)
Uimp=Uref.copy()
U=np.zeros(2*Nnodes)

Fint=model.Solve(Ns,const_ddls_u,Uimp,Fexto,U,True)
U_u=U.copy()
```

```
#Scaling the traction profile to the experimental Force
print(Fres/sum(Fint[top_nodes+Nnodes]))
scal=Fres/sum(Fint[top_nodes+Nnodes])
Fext[top_nodes+Nnodes]=Fint[top_nodes+Nnodes]*scal
print(Fres/sum(Fext[top_nodes+Nnodes]))

# To check and have a initial displacement field
test=model.Solve(1,const_ddls_f,Uimp,Fext,U,True);
print(Fres/sum(test[top_nodes+Nnodes]))
Uini=U.copy()
fig, ax = plt.subplots()
model.show_field(fig, ax,Uref[Nnodes::], name = "FE displacement")
```

```
Solving…
For load factor 0.100 after 6 iterations R/Fext = 1.000e+00 dU/U=2.056e-08
For load factor 0.200 after 6 iterations R/Fext = 1.000e+00 dU/U=1.034e-07
For load factor 0.300 after 6 iterations R/Fext = 1.000e+00 dU/U=2.853e-07
For load factor 0.400 after 6 iterations R/Fext = 1.000e+00 dU/U=6.032e-07
For load factor 0.500 after 7 iterations R/Fext = 1.000e+00 dU/U=7.670e-08
For load factor 0.600 after 7 iterations R/Fext = 1.000e+00 dU/U=1.501e-07
For load factor 0.700 after 7 iterations R/Fext = 1.000e+00 dU/U=2.656e-07
For load factor 0.800 after 7 iterations R/Fext = 1.000e+00 dU/U=4.359e-07
For load factor 0.900 after 7 iterations R/Fext = 1.000e+00 dU/U=6.746e-07
For load factor 1.000 after 7 iterations R/Fext = 1.000e+00 dU/U=9.962e-07
0.9703736646537541
0.9999999999999998

Solving…
For load factor 1.000 after 7 iterations R/Fext = 1.000e+00 dU/U=2.270e-07
0.9999999032051472
```

[33]:
```
(<matplotlib.collections.LineCollection at 0x7247090701d0>,
 <matplotlib.colorbar.Colorbar at 0x724703f6cd10>)
```

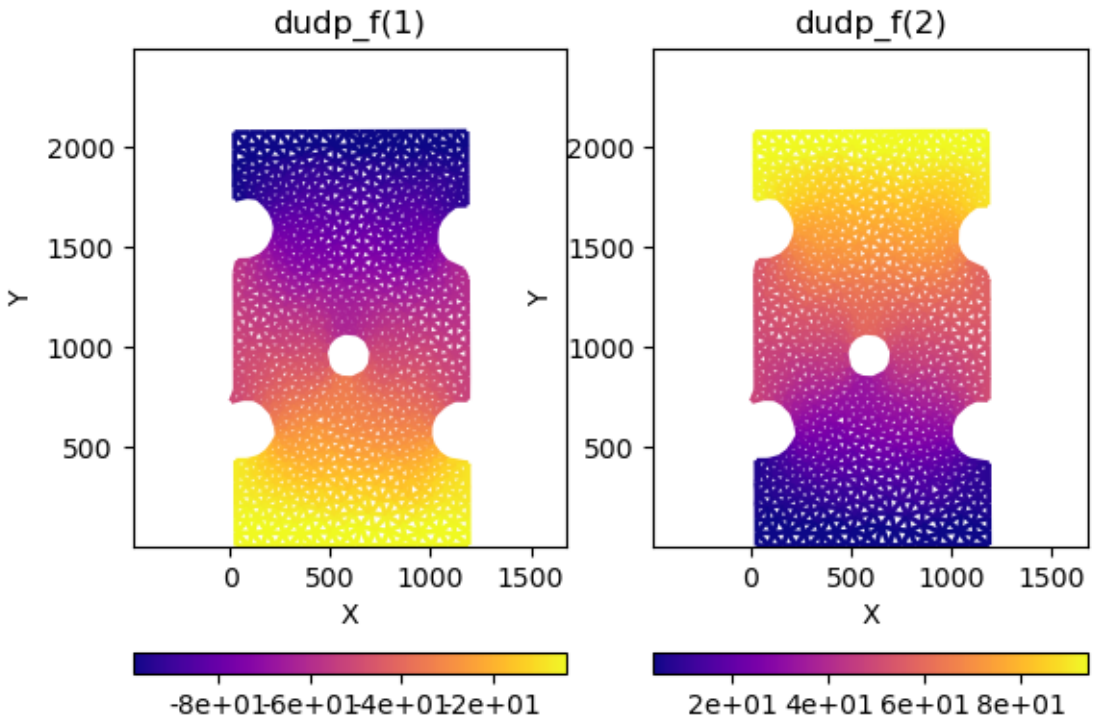


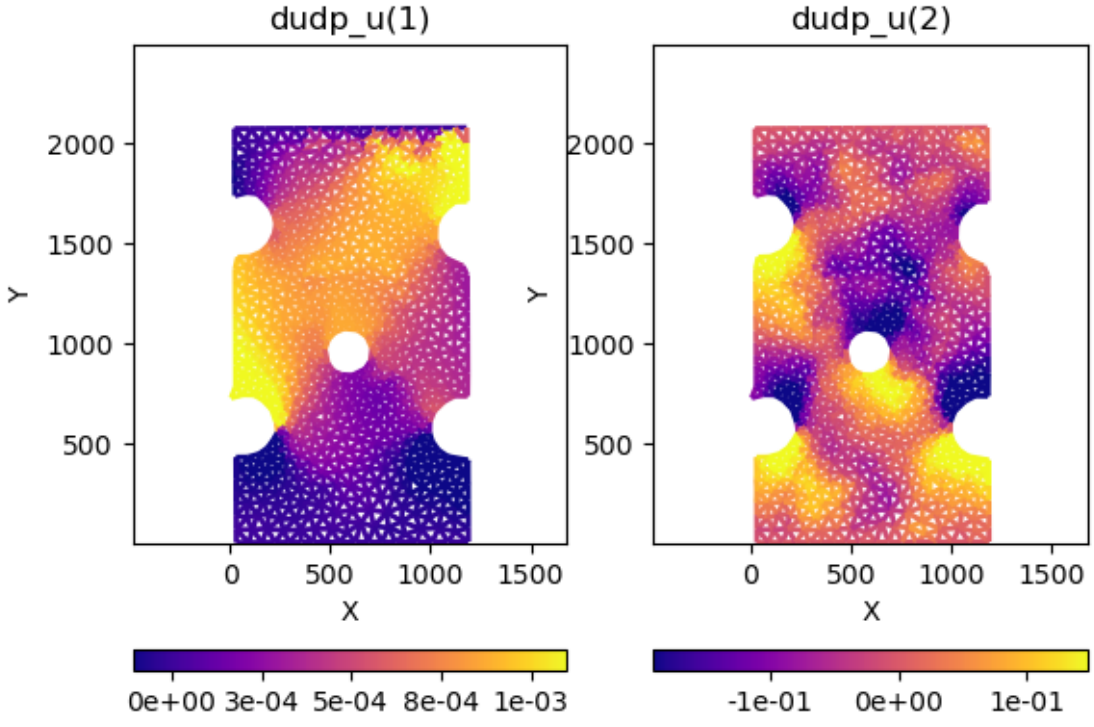

[34]:
```
def updatematerial(P):
    mat=fem.MatModel(pix2m)
    mat.K=Kini*P[0]
    mat.Eo=Eoini*P[1]
    return mat

def sensitivity_f(P,Ui):
    dudp = np.zeros((Ui.shape[0],P.size))
    dP=0.01
    mati=model.material
    Umod=Ui.copy();

    for ii in range(len(P)):
        Pi=P+0.;
        Pi[ii]=Pi[ii]+dP
        newmat=updatematerial(Pi)
        model.material=newmat
        model.Solve(1,const_ddls_f,Uimp,Fext,Umod,False);
        dudp[:,ii]=(Umod-Ui)/dP
    model.material=mati
    return dudp
def sensitivity_u(P,Ui):
```

```
    dudp = np.zeros((Ui.shape[0],P.size))
    dP=0.01
    mati=model.material
    Umod=Ui.copy();

    for ii in range(len(P)):
        Pi=P+0.;
        Pi[ii]=Pi[ii]+dP
        newmat=updatematerial(Pi)
        model.material=newmat
        model.Solve(1,const_ddls_u,Uimp,Fexto,Umod,False);
        dudp[:,ii]=(Umod-Ui)/dP
    model.material=mati
    return dudp

def hessian(dudp):
    H = np.zeros((np.size(dudp,1),np.size(dudp,1)))
    for ii in range(np.size(dudp,1)):
        for jj in range(np.size(dudp,1)):
                H[ii][jj]=np.dot(dudp[:,ii],dudp[:,jj])
    return H


def residual(dudp,r):
    b = np.zeros((np.size(dudp,1)))
    for ii in range(np.size(dudp,1)):
                b[ii]=np.dot(dudp[:,ii],r)
    return b
```

[35]:
```
P=np.array((1.,1.))
dudp_f=sensitivity_f(P, Uini)
fig=plt.figure()
ax = plt.subplot(121)
model.show_field(fig, ax,dudp_f[Nnodes:,0], name = "dudp_f(1)")
ax = plt.subplot(122)
model.show_field(fig, ax,dudp_f[Nnodes:,1], name = "dudp_f(2)")
dudp_u=sensitivity_u(P, U_u)
fig=plt.figure()
ax = plt.subplot(121)
model.show_field(fig, ax,dudp_u[Nnodes:,0], name = "dudp_u(1)")
ax = plt.subplot(122)
model.show_field(fig, ax,dudp_u[Nnodes:,1], name = "dudp_u(2)")
```

[35]:
```
(<matplotlib.collections.LineCollection at 0x724703d9a6f0>,
 <matplotlib.colorbar.Colorbar at 0x724703cbd970>)
```


dudp_f(1)
dudp_f(2)
2000
1500
1000
500
Y
0
500
1000
1500
X
-8e+01 -6e+01 -4e+01 -2e+01
2e+01 4e+01 6e+01 8e+01


dudp_u(1)
dudp_u(2)
2000
1500
1000
500
Y
0
500
1000
1500
X
0e+00 3e-04 5e-04 8e-04 1e-03
-1e-01 0e+00 1e-01

[36]:
```
H_f=hessian(dudp_f)
print("H_f", H_f)
H_u=hessian(dudp_u)
print("H_u",H_u)
```

```
H_f [[ 2488941.62308898 -2359079.8961313 ]
 [-2359079.8961313   2236001.92723434]]
H_u [[ 4.76394141e-04 -2.33849762e-02]
 [-2.33849762e-02  9.96255611e+00]]
```

[37]:
```
P=np.array((1.,1.))
nIter=10 # number of iteration to perform
for iter in range(nIter):
    # a. compute the sensitivity matrix $dudp$
    dudp=sensitivity_f(P, U)
    # b. compute the covariance matrix $C$
    H=hessian(dudp)
    # c. compute the residual vector $b$
    b=residual(dudp, Uref-U)
    # d. solve the system $C\; dP=b$ for getting the parameter increment
    dP=np.linalg.solve(H,b)
    # e. update the parameter values $P=P+dP$
    P+=.9*dP
    # f. update the FE solution $U$ with the updated parameters
    material=updatematerial(P)
    model.material=material
    Fint=model.Solve(1,const_ddls_u,Uimp,Fext,U,False);
    scal=Fres/sum(Fint[top_nodes+Nnodes])
    Fext[top_nodes+Nnodes]=Fint[top_nodes+Nnodes]*scal
    Fint=model.Solve(1,const_ddls_f,Uimp,Fext,U,False);

    # g. compute the norm of the gap between $U$ and $U_{DIC}$
    norm = np.linalg.norm(Uref-U)
    normp = np.linalg.norm(dP)/np.linalg.norm(P)
    print('***FEMU loop Iteration %02d: |U-Uref| =%6.3e  dP/P =%6.4f***' %␣
 ↪(iter,norm,normp))

print('IDENTIFIED PARAMETERS\n K = %5.3f MPa,Eo %5.3f' %  (Kini*P[0]*1.
 ↪e-6,Eoini*P[1] ))
print('REFERENCE PARAMETERS\n K = %5.3f MPa,Eo %5.3f' %  (2e8*1.e-6,0.2 ))
np.savez('femu-from-fem',U=U,mat=model.material)
```

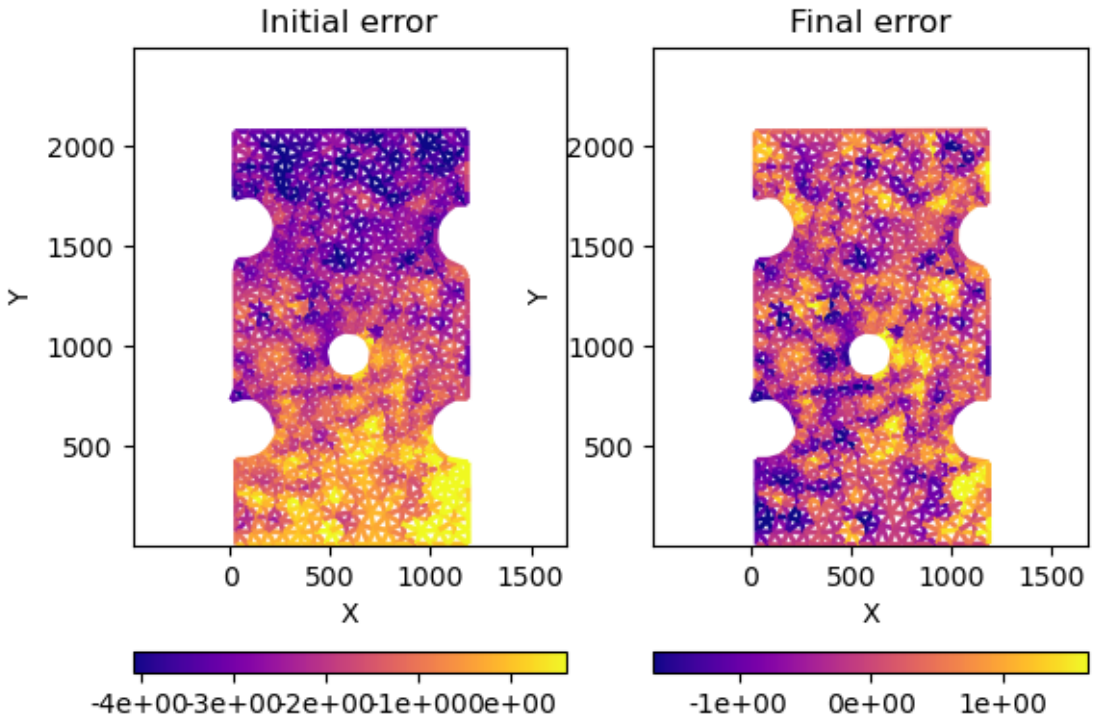
```
***FEMU loop Iteration 00: |U-Uref| =1.439e+02  dP/P =1.8837***
***FEMU loop Iteration 01: |U-Uref| =5.113e+01  dP/P =0.2529***
***FEMU loop Iteration 02: |U-Uref| =3.728e+01  dP/P =0.1788***
***FEMU loop Iteration 03: |U-Uref| =3.719e+01  dP/P =0.0683***
***FEMU loop Iteration 04: |U-Uref| =3.716e+01  dP/P =0.0108***
***FEMU loop Iteration 05: |U-Uref| =3.716e+01  dP/P =0.0010***
```

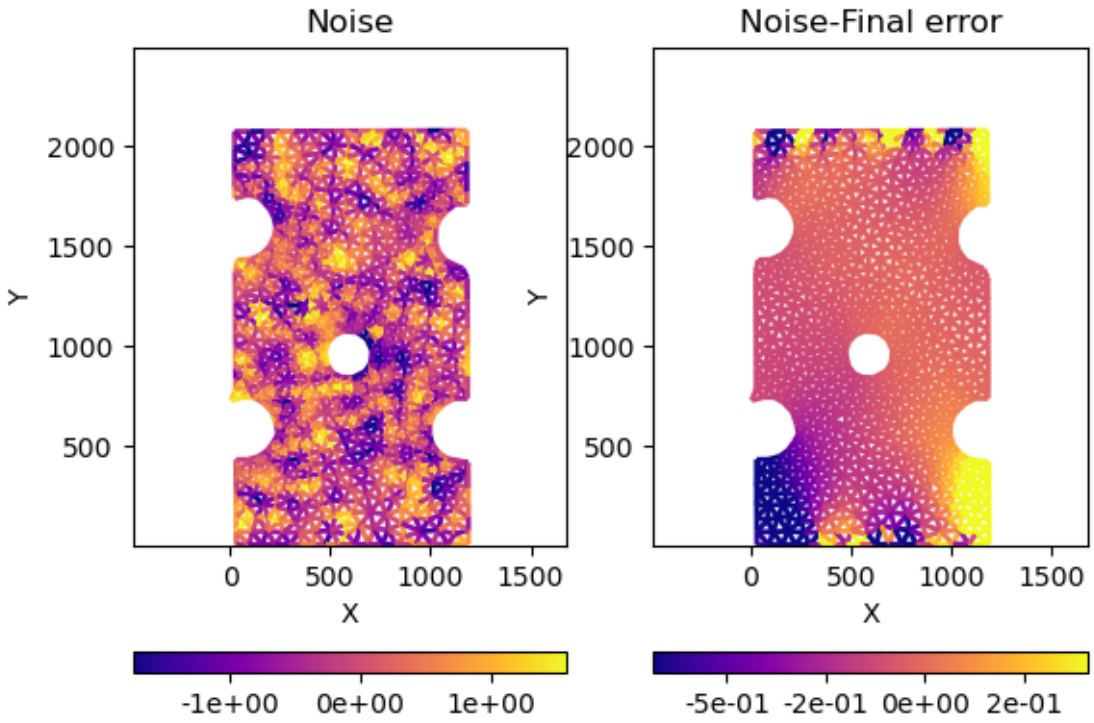
```
***FEMU loop Iteration 06: |U-Uref| =3.716e+01  dP/P =0.0001***
***FEMU loop Iteration 07: |U-Uref| =3.716e+01  dP/P =0.0000***
***FEMU loop Iteration 08: |U-Uref| =3.716e+01  dP/P =0.0000***
***FEMU loop Iteration 09: |U-Uref| =3.716e+01  dP/P =0.0000***
IDENTIFIED PARAMETERS
 K = 185.576 MPa,Eo 0.184
REFERENCE PARAMETERS
 K = 200.000 MPa,Eo 0.200
```

[ ]:

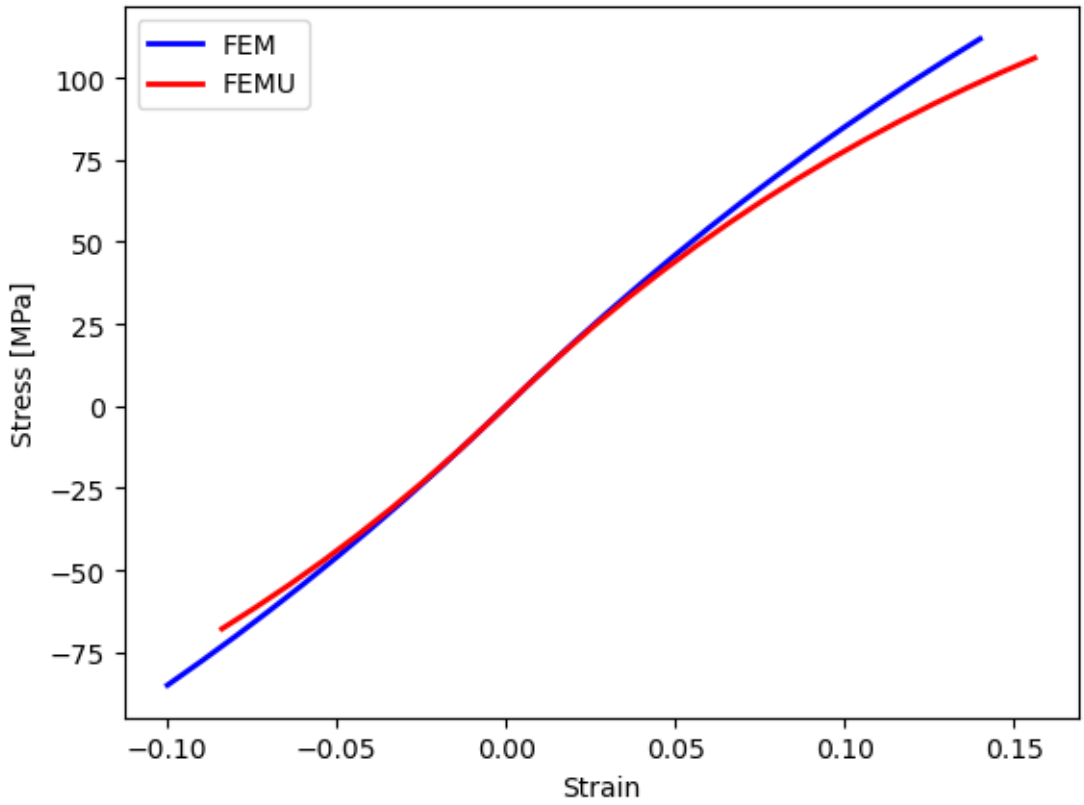
[38]:
```
fig=plt.figure()
ax = plt.subplot(121)
model.show_field(fig, ax,Uini[Nnodes::]-Uref[Nnodes::], name = "Initial error")
ax = plt.subplot(122)
model.show_field(fig, ax,U[Nnodes::]-Uref[Nnodes::], name = "Final error")
fig=plt.figure()
ax = plt.subplot(121)
model.show_field(fig, ax,dUref[Nnodes::], name = "Noise")
ax = plt.subplot(122)
model.show_field(fig, ax,U[Nnodes::]-(Uref[Nnodes::]-dUref[Nnodes::]), name =␣
↪"Noise-Final error")
```

[38]:
```
(<matplotlib.collections.LineCollection at 0x72470380e120>,
 <matplotlib.colorbar.Colorbar at 0x72470380d010>)
```






[41]:
```
Efem=B.dot(Uref)
epsii=np.arange(min(Efem),max(Efem),0.01)
stressi=material.GetStress(epsii)
ff=plt.figure()
plt.plot(epsi,stress/pix2m*1.e-6,'b-', linewidth=2,label='FEM')
plt.plot(epsii,stressi/pix2m*1.e-6,'r-', linewidth=2,label='FEMU')
plt.legend()
plt.xlabel('Strain')
plt.ylabel('Stress [MPa]');
```


FEM
FEMU
100
75
50
25
0
−25
−50
−75
Stress [MPa]
−0.10
−0.05
0.00
0.05
0.10
0.15
Strain

[ ]:

# Class's references i



Avril, S., Bonnet, M., Bretelle, A., Grediac, M., Hild, F., Ienny, P., Latourte, F., Lemosse, D., Pagano, S., Pagnacco, E., et al. (2008). **Overview of identification methods of mechanical parameters based on full-field measurements.** Experimental Mechanics, 48(4):381–402.


Claire, D., Hild, F., and Roux, S. (2004). **A finite element formulation to identify damage fields: The equilibrium gap method.** International Journal for Numerical Methods in Engineering, 61:189–208.


Grédiac, M., Pierron, F., Avril, S., and Toussaint, E. (2006). **The virtual fields method for extracting constitutive parameters from full-field measurements: a review.** Strain, 42(4):233–253.


Leclerc, H., Perie, J., Roux, S., and Hild, F. (2009). **Computer Vision/Computer Graphics CollaborationTechniques, chapter Integrated Digital Image Correlation for the Identification of Mechanical Properties.** Springer, Berlin.


Lecompte, D., Smits, A., Sol, H., Vantomme, J., and Van Hemelrijck, D. (2007). **Mixed numerical–experimental technique for orthotropic parameter identification using biaxial tensile tests on cruciform specimens.** International Journal of Solids and Structures, 44(5):1643–1656.

# Class's references ii

Réthoré, J. (2010). **A fully integrated noise robust strategy for the identification of constitutive laws from digital images.** International Journal for Numerical Methods in Engineering, 84:631–660.