Computational Reality XIX

Evolution of probability distribution

B. Emek Abali, Wolfram Martens @ LKM - TU Berlin

Abstract

How likely an event occurs, depends on the statistical data. Suppose we pulled the mass on a spring-dashpot system away from its equilibrium position and left it free to move. It gets a velocity and swings back to the equilibrium position, due to the spring back-force. We may measure the position, $x$, and velocity, $\dot{x}$, of the mass over time. Assume we can measure without any errors. When we redo the same test ten times, but if at a specific time only eight times the same state $x, \dot{x}$ is measured and two times something else, then the probability of this state is 0.8 which is somehow strange. Mathematically it should be always the same path in the state space $x, \dot{x}$, however, in reality a small error in the initial condition or even a tiny misalignment in the system may lead to not deterministic processes. We have seen the notion of probability in a discrete dataset in Comp.Real.XIV where the measurement was error-prone. This time the uncertainty is in the mechanical system or in the initial conditions which we will describe or bring that uncertainty to the equations in a probabilistic way.

1 Motivation

Suppose we apply a force $F(t) + f(t)$ to a mechanical oscillator, simply a spring-dashpot system, where the $F(t)$ represents the deterministic, transient force and $f(t)$ a random (in Greek stochastic) force. The stochastic force simulates the uncertainty, i.e., deviation of the system from the ideal path due to the imperfections in the system. It is of importance to set the external force $F(t)$ zero since we will use the FOKKER-PLANCK equations in the next section. The FOKKER-PLANCK equations cannot model a system with the external deterministic force. Of course this stochastic deviation is small, however, later we will see a chaotic\(^1\) system where this stochastic part is quite important. First, let us start with a system, modeled with a linear spring responding to a force linearly: $cx$, connected parallel with a linear dashpot: $\kappa \dot{x}$. The differential equation of the system reads: $m\ddot{x} + \kappa \dot{x} + cx = f(t)$, where the mass, $m$, oscillates back and forth. This equation can be solved analytically or numerically. We solved it by discretizing with finite difference method and plot the position ($x,t$) and the velocity $\dot{x} = \dot{x}(t)$. Moreover we can plot the motion in the state space $(x, \dot{x}, t)$ or its projection $(x, \dot{x})$, often called the phase space. For the deterministic loading, i.e., $f(t) = 0$ the results are in Fig. 1 and in Fig. 2. This phase space\(^2\) represents the motion in an abstract way which Poincare used for analysis. Now we put the

\(^1\)When a small change of the input creates an extravagant change in output then the response of the system is called chaotic.
\(^2\)The name phase space is misleading, therefore often this is also called state space since it is the projection in the state space. The state space was called in 20’s first by Cartan in French as being the phase space and time. We will also use state for phase space, since it is the standard use in thermodynamics.

Figure 1: Linear harmonic oscillator under deterministic loading, the position in time at left, the velocity in time in the middle and the motion path in the phase space at right.
stochastic part upon and show the path of the same system with a stochastic perturbation on the (projected) state space. Let us recall that stochastic means random, i.e., if we solve it again the result changes. Thus we solve once and then five times and show the results in Fig. 3. Computational difficulties by convergence occur thus we used small stochastic part, tiny time steps and solved only the fifth in time regarding the Fig. 1. The random part produces another result therefore a unique solution does not exist. However, we can solve the deterministic part uniquely, which is the expected value, and upon that we can produce a probabilistic distribution. When we solve it many times, say 100, and then count in every small range the solutions, say for the time \( t \) in some range\(^3\) 30 solutions exist, then the probability in time \( t \) reads 0.3. This method needs many iterations since we have hundred different datasets and we want to count for a specific time range. For every time range we should scan every dataset once. Instead of counting data in a range we can distribute every data and sum them up. This costs computationally less because we scan the dataset once and span the value in space by means of a distribution function. A simple example in one-dimension is shown in Fig. 4 to clarify the idea. We choose eight random numbers between zero and one and distribute them in space \([0, 1]\) via GAUSS normal distribution (black lines on the left figure). Their sum is the pdf (red line on the left figure).

\(^3\) Concretely the range is: \( t \pm \epsilon \) where \textit{epsilon} is small with respect to \( t \).

**Figure 2:** State space and the phase space as its projection for the linear harmonic oscillator under deterministic loading.
The integral of the pdf is the sigmoid function (red line on the right figure). This would be exactly one if we would taken the whole space into consideration. The whole space is always \((-\infty, +\infty)\) which is of course not possible for numerics. It is of importance to notice in Fig. 4 that the variance, \(\sigma\), for the Gauss distribution:

\[
P(x, \mu) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{1}{2}\frac{(x-\mu)^2}{\sigma^2}\right)
\]

is arbitrary and thus difficult to find out any physical meaning if not started from the discrete values. However, this value is only needed for creating a continuous pdf out of discrete values.

The chosen Gauss distribution function is one of the many distribution functions. Many other probabilistic distributions, like Gauss normal, Bernoulli, Cauchy, Pareto, Poisson, Rayleigh, Laplace, Weibull, logistic, log-normal, geometric, gamma, beta, F, student’s-t, chi-square exist in the literature. For finite elements we utilize a similar procedure to create a field function from the nodal values by interpolating them with the shape functions. The distribution functions are not changing in time, as in fem the shape functions does not change in time. The totality of the distribution functions in every node or discrete data configures the space, the set of the distribution or shape functions and it does not evolve in time. But the nodal values do regarding a differential equation. For mechanics we use the balance equations. In probabilistic evolutions this differential equation is the Fokker-Planck equation if the process is a Markov process. The process that lacks history is called a Markov process.

\section{Fokker-Planck equation}

Consider an evolution of the probability distribution. The stochastic part of the loading is represented with a probability density function, \(p(x_1, x_2)\), in (projected) state space of \(x_1 = x\) and \(x_2 = \dot{x}\). If the space is big enough, theoretically \(x_1 \in (-\infty, +\infty)\) and \(x_2 \in (-\infty, +\infty)\), then the probability density function integrated over the whole space reads the probability that is equal to one. No matter how the pdf evolves in time, the probability reads one. The state space spans a \(n\)-dimensional abstract space where every single dimension \(x_i\) is a real number \(x_i \in (-\infty, +\infty), i = 1, 2, 3, ..n.\)
Now suppose that the pdf is a conserved quantity in the state space. We do not use the space-time notation therefore the meaning of the conserved quantity is misleading. It is actually being conserved in space-time, however, it is easier to give the physical meaning for using space and time separately. A conserved quantity cannot vanish in one place and pop out in another simultaneously, it has to follow a path. Being a conserved quantity is the first axiom in Liouville’s theorem, thus in state space, i.e., \( \{x_1, x_2, \ldots, x_n\} \) a balance equation for the pdf, \( p = p(x_1, x_2, \ldots, x_n) \), can be formulated:

\[
\frac{d\tilde{p}(\mu = 1|x_k)}{dt} = \frac{d}{dt} \int_{\Omega} p\,dx_1 \ldots \,dx_n \, , \quad dx = dx_1 \ldots \,dx_n ,
\]

\[
\frac{d}{dt} \int_{\Omega} p\,dx = - \int_{\partial\Omega} C_i p\,dx^i + \int_{\partial\Omega} N_i \,dx^i , \quad dx^i = dx_1 \ldots \,dx_{i-1} \,dx_{i+1} \ldots \,dx_n , \tag{1}
\]

where summation convention over multiple indices are used although indices denote just the members of the argument lists, no tensorial relation or any transformation laws must exist. This is the usual balance equation for open systems, where \( C_i \) and \( N_i \) represent convective and non-convective (also called diffusive) fluxes into and out of the probability. When we assume that \( \frac{dx}{dt} = v \) and neglect convective flux, i.e. \( C_i = 0 \), then after using GAUSS theorem and defining \( N_i = \frac{1}{2} \frac{\partial (B_{ij}p)}{\partial x_j} \), the balance equation reads:

\[
\int_{\Omega} \left( \frac{\partial p}{\partial t} + v_i \frac{\partial p}{\partial x_i} + p \frac{\partial v_i}{\partial x_i} \right) \,dx = \int_{\partial\Omega} \frac{1}{2} \frac{\partial (B_{ij}p)}{\partial x_j} \,dx^i ,
\]

\[
\int_{\Omega} \left( \frac{\partial p}{\partial t} + \frac{\partial (v_i p)}{\partial x_i} - \frac{1}{2} \frac{\partial^2 (B_{ij}p)}{\partial x_i \partial x_j} \right) \,dx = 0 \, . \tag{2}
\]

The diffusive matrix \( B_{ij} = G_{ik}G_{jk} \) is actually the connection to the stochastic process for the features \( x_i \) as written in Ito form:

\[
dx_i = v_i \,dt + G_{ij} \,dW_j , \tag{3}
\]

describes in a Brownian motion or Wiener process with aforementioned noise \( G_{ij} \,dW_j \) by Langevin in 190’s.

The white noise implies that the mean value, or the expected value is zero: \( E[W_j] = 0 \) so that the stochastic part changes the deterministic part as we have seen in the last section.

Although the real derivation is out of scope, we show the first and second moments. The rate of the first moment of the pdf\(^5\) is associated with the velocity:

\[
\frac{d}{dt} \int_{|x_i - X| < \epsilon} (x_i - X)p(x_k,t + dt|X,t) \,dx = v_i(X,t) + O(\epsilon) \, . \tag{4}
\]

Similarly the rate of the second moment of the pdf\(^6\) provides the diffusion matrix:

\[
\frac{d}{dt} \int_{|x_i - X| < \epsilon} (x_i - X)(x_j - X)p(x_k,t + dt|X,t) \,dx = B_{ij}(X,t) + O(\epsilon) \tag{5}
\]

EINSTEIN used the latter connection and described the molecular fluctuation with the differential equation explained below in 1917.

Finally, the FOKKER-PLANCK equation from the 10’s reads in local:

\[
\frac{\partial p}{\partial t} + \frac{\partial (v_i p)}{\partial x_i} - \frac{1}{2} \frac{\partial^2 (B_{ij}p)}{\partial x_i \partial x_j} = 0 \, . \tag{6}
\]

By utilizing method of weighted residuals the form reads in state space:

\[
\int_{\Omega} \left( \frac{\partial p}{\partial t} \delta p + \frac{\partial (v_i p)}{\partial x_i} \delta p - \frac{1}{2} \frac{\partial^2 (B_{ij}p)}{\partial x_i \partial x_j} \delta p \right) \,dx = 0 , \quad dx = dx_1 \,dx_2 \ldots \,dx_n , \tag{7}
\]

\(^4\)also called the ensemble average \( E[\ldots] \)

\(^5\)First moment is the mean or expected value.

\(^6\)Second moment is the correlation function.
where \( \delta p \) is the test function. Thus, we discretize in \( x_i \) space using finite element method and in time with finite difference method, as usual:

\[
\int_{\Omega} \left( \frac{p - p^0}{dt} \delta p + \frac{\partial(v_ip)}{\partial x_i} \delta p - \frac{1}{2} \frac{\partial ^2 (B_{ij} p) \delta p}{\partial x_i \partial x_j} \right) dx \equiv 0 . \tag{8}
\]

Now we employ integration by parts only once to the diffusion matrix since it is in second-order. The boundary terms leads to zero because the probability density vanishes asymptotically and we integrate on a big enough domain so that on the boundaries \( p \) vanishes:

\[
\int_{\Omega} \left( \frac{p - p^0}{dt} \delta p - v_i \frac{\partial \delta p}{\partial x_i} + \frac{1}{2} \frac{\partial (B_{ij} \delta p)}{\partial x_i \partial x_j} \right) dx_1 dx_2 = 0 . \tag{9}
\]

This can be solved after declaring the diffusion matrix or the second-moment, \( B_{ij} \), and the drift vector, the velocity or the first-moment, \( v_i \), which is the aim in the coming section.

### 3 Applications

Again consider the mass connected to the parallel spring-dashpot system and the differential equation that describes the motion \( X \):

\[
m \ddot{X} + \kappa \dot{X} + cx = \sigma \zeta(t) , \quad x_1 = X , \quad x_2 = \dot{X} ,
\]

\[
m \frac{dx_2}{dt} + \frac{c}{m} x_1 dt = \sigma \zeta(t) dt , \quad x_2 dt = dx_1 , \tag{10}
\]

or even as a set of equations:

\[
\begin{pmatrix} dx_1 \\ dx_2 \end{pmatrix} = \begin{pmatrix} 0 & \frac{c}{m} \\ \frac{c}{m} & 0 \end{pmatrix} dt + \begin{pmatrix} 0 \\ \sigma \end{pmatrix} \begin{pmatrix} dW_1(t) \\ dW_2(t) \end{pmatrix} , \tag{11}
\]

where we separated the stochastic part to an instationary \(^7\) amplitude \((0 , \sigma)\) and the white noise \((dW_1(t), \ dW_2(t))\).

The white noise has the mean value zero \( E[dW_i(t)] = 0 \) and it is independent in time, that means there are no correlation between the time steps \( E[dW_i(t) dW_i(t + dt)] = \delta(dt) \), where the delta function is one at zero vanishes elsewhere. Therefore in different times, the latter expected value vanishes, an abstract way of saying that the noise is fluctuating in time independently in each time step, no continuity or smoothness is needed for that function. Let us give an example in two subsequent time steps. Suppose at the same position for these subsequent times, the values 0.2,0.8 are given. We think always that a quantity evolves in time. Thus, if we would have taken the second time instant more closer to the first one it should read, e.g., 0.2,0.4 since the function is not changing instantaneously. Exactly this fact does not hold for the noise. It fluctuates as quick as it is necessary, i.e., different times lack of correlation. We give the codes at the end, where we simply have chosen random numbers without any dependence at all. Finally it is an easy task to relate the Eq. (12) to the Ito Eq. (3) such that:

\[
v_i = \begin{pmatrix} \frac{c}{m} x_2 - \frac{c}{m} x_1 \\ \frac{c}{m} x_2 \end{pmatrix} \quad , \quad G_{ij} = \begin{pmatrix} 0 & 0 \\ 0 & \sigma \end{pmatrix} \quad , \quad B_{ij} = G_{ik} G_{jk} . \tag{13}
\]

For the linear oscillator we have solved the variational form (9) obtained from the FOKKER-PLANCK equation by using the latter drift vector and diffusion matrix, cf. the Fig. 5. This is a transient solution, i.e., with a given initial condition the evolution of the pdf in time and in state space has been computed. The initial condition is a GAUSS normal distribution around the 0,0 point, i.e., initially the mass is at the \( x_1 = X = 0 \) position and has the \( x_2 = dX/dt = 0 \) velocity. The distribution function spreads out due to the diffusion matrix, \( B_{ij} \). Upon

\(^7\)The “white” noise is not affected from the system, thus \( f \) is constant. There are some systems that the stochastic term depends on time.

5
that the drift vector, \( \mathbf{v}_i \), leads the pdf in association with the deterministic part of the differential equation. In the end, without solving the differential equation of motion, we get the path in the state space to be seen in Fig. 5. Compare with the Fig. 3, where the differential equation has been solved. Two quite different way of solving the same reality which is deterministic and stochastic.

4 Code

The code for the harmonic linear oscillator:

```python
# from dolfin import *
import numpy
import scipy
import scipy.optimize
import pylab
import mpl_toolkits.mplot3d.axes3d as p3

freq = 5.

f = lambda t: numpy.random.random_sample()  # returns a random number between [0, 1)
m = 1.
c = 5.
kappa = 1.
dt = 0.1
t = 0.
tend = 10.

# initial conditions
x0, x00 = 5., 5.

# the residual
ode = lambda x: m*(x-2.*x0+x00)/dt/dt + kappa*(x-x0)/dt + c*x

#pylab.ion()  # uncomment for draw() in time loop
```

Figure 5: Transient solution of the pdf in state space for the linear oscillator.
The code for the deterministic and stochastic part of loading solved many times:

```python
# from dolfin import *
import numpy
import scipy
import scipy.optimize
import pylab
import mplotkits.mplot3d.axes3d as p3

freq=5.
sigma = 0.26
f = lambda t: sigma*numpy.random.uniform(-1.,+1.)  # returns a random number between -1, +1
m=1.
c=5.
kappa=1.
```
dt=0.01
t=0.
x0, x00=5., 5. # initial conditions
xi0, xi00=5., 5.
# the residual
ode_x= lambda x: m*(x-2.*x0+x00)/dt/dt + kappa*(x-x0)/dt + c*x
ode_xi= lambda xi: m*(xi-2.*xi0+xi00)/dt/dt + kappa*(xi-xi0)/dt + c*xi - f(t)

t=0.
tend=0.5
xiplot, xidotplot=[]
while t<tend:
    x=scipy.optimize.fsolve(ode_x, x0)
    x_dot=(x-x0)/dt
    pylab.plot(x, x_dot, 'r', marker='o', markersize=7)
    xi=scipy.optimize.fsolve(ode_xi, xi0)
    xiplot.append(xi)
    xi_dot=(xi-xi0)/dt
    xidotplot.append(xi_dot)
    pylab.plot(xiplot, xidotplot, 'b-', marker='d', markersize=4, linewidth=2, alpha=0.8)
    x00, xi00 = x0, xi0
    x0, xi0 = x, xi
    t+=dt

for k in range(5):
    print k
    t=0.
x0, x00=5., 5. # initial conditions
xi0, xi00=5., 5.
xiplot, xidotplot=[]
while t<tend:
    x=scipy.optimize.fsolve(ode_x, x0)
    x_dot=(x-x0)/dt
    pylab.plot(x, x_dot, 'r', marker='o', markersize=7)
    xi=scipy.optimize.fsolve(ode_xi, xi0)
    xiplot.append(xi)
    xi_dot=(xi-xi0)/dt
    xidotplot.append(xi_dot)
    pylab.plot(xiplot, xidotplot, 'b-', marker='d', markersize=4, linewidth=2, alpha=0.8)
    x00, xi00 = x0, xi0
    x0, xi0 = x, xi
    t+=dt

pylab.savefig('CR19_oscillator_f_01.png')

The code for the probability density function and its cumulative distribution for a random set of numbers between zero and one:

```python
import numpy
import scipy
cdf = stats.norm.cdf(r)
pdf = stats.norm.pdf(r)
```
import pylab

data=numpy.random.random_sample((8,))
normalize=1./float(len(data))

sigma=0.1

def distribution(x,mean):
    return 1./(2.*numpy.pi*sigma**2)*(0.5)*numpy.exp((-1./2.)*((x-mean)/sigma)**2)*normalize

def p(x):
    p=0.
    for i in range(len(data)):
        p+=distribution(x,data[i])
    return p

h=100.
dx=1./h
space=numpy.linspace(0.,1.,int(h))

def cumulative(x):
    out=0.
    for i in range(len(space)):
        if space[i]<x:
            out+=p(space[i])*dx
    return out

pylab.rc('font', size=26)
pylab.figure(1, figsize=(30,8), dpi=100)
pylab.subplot(1, 2, 1)
pylab.xlabel(' $x$ ')
pylab.ylabel(' pdf, $p(x)$ ')
pylab.title('Probability density function')
pylab.plot(data, 0.*data, 'b', marker='d', markersize=22)

pylab.subplot(1, 2, 2)
pylab.xlabel(' $x$ ')
pylab.ylabel(' $\int_0^x p(x) dx$ ')
pylab.title('Cumulative probability distribution')

for i in range(len(space)):
    x=space[i]
    pylab.plot(x,p(x), 'r', marker='o', markersize=12)
    for k in range(len(data)):
        x=0.*data[k]
        pylab.plot(x,distribution(x, data[k]), 'k', marker='o', markersize=4, alpha=0.7)

for i in range(len(space)):
    x=0.*space[i]
    pylab.plot(x,cumulative(x), 'r', marker='o', markersize=12)

print float(i)/float(len(space))*100., '% is done'

pylab.savefig('CR19_p.png')

The code for the transient FOKKER-PLANCK equation of the linear oscillator:
# it under the terms of the GNU Lesser General Public License as published by
# the Free Software Foundation, either version 3 of the License, or
# (at your option) any later version.
#
# This code is distributed in the hope that it will be useful,
# but WITHOUT ANY WARRANTY; without even the implied warranty of
# MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
# GNU Lesser General Public License for more details.
#
# For the GNU Lesser General Public License see <http://www.gnu.org/licenses/>.
#
#--------------------------------------------------------------------------------
from dolfin import *
import numpy
from numpy import array

parameters["form_compiler"]["optimize"] = True
parameters["form_compiler"]["cpp_optimize"] = True

class InitialConditions(Expression):
    def __init___.(self, sigma):
        self.s = sigma
    def eval(self, out, x):
        out[0] = 1. / (2. * pi * self.s**2) ** 0.5 * exp(-((x[0] - 5.)**2 + (x[1] - 0.)**2) / self.s**2)

class iterate(NonlinearProblem):
    def __init___.(self, a, L, bc, inter_B, exter_B):
        NonlinearProblem.__init__(self)
        self.L = L
        self.a = a
        self.bc = bc
        self.inter = inter_B
        self.exter = exter_B
    def F(self, b, x):
        assemble(self.L, tensor=b, interior_facet_domains=self.inter, 
        exterior_facet_domains=self.exter)
        for condition in self.bc: condition.apply(b, x)
    def J(self, A, x):
        assemble(self.a, tensor=A, interior_facet_domains=self.inter, 
        exterior_facet_domains=self.exter)
        for condition in self.bc: condition.apply(A)

class space_as_field(Expression):
    def __init___.(self, mesh):
        self.mesh = mesh
    def eval_cell(self, out, x, ufc_cell):
        out[0] = x[0]
        out[1] = x[1]
    def value_shape(self):
        return (2,)

x1Min, x1Max, x1Dof = 0., 6., 200
x2Min, x2Max, x2Dof = -9., 1., 200
mesh = Rectangle(x1Min, x2Min, x1Max, x2Max, x1Dof, x2Dof)
D = mesh.topology().dim()

space = FunctionSpace(mesh, 'CG', 1)
interior = MeshFunction("uint", mesh, D-1)
interior.set_all(0)
exterior = MeshFunction("uint", mesh, D-1)
exterior.set_all(0)
bc=[]
dp = TrialFunction(space)
\[ \text{delp} = \text{TestFunction}(\text{space}) \]
\[ p = \text{Function}(\text{space}) \]
\[ p_0 = \text{Function}(\text{space}) \]
\[ x = \text{space}_\text{as_field}(\text{mesh}) \]
\[ t, dt, t_{\text{end}} = 0.0, 0.1, 0.6 \]
\[ m, \text{kappa}, c = 1.0, 1.0, 5.0 \]
\[ \text{sigma} = 0.26 \]
\[ i, j, k, l = \text{indices}(4) \]
\[ v = \text{as_vector}([x[1], -\text{kappa}/m*x[1] - c/m*x[0]]) \]
\[ G = \text{as_matrix}([[0.0, 0.0], [0.0, \text{sigma}]]) \]
\[ B = \text{as_matrix}(G[i,k] \cdot G[j,k], (i, j)) \]
\[ x = \text{variable}(x) \]
\[ \text{Form} = ((p - p_0)/dt \cdot \text{delp} + \text{diff}(v, x)[i,i] \cdot p \cdot \text{delp} + v[i] \cdot p \cdot dx(i) \cdot \text{delp} \]
\[ + 1/2 \cdot B[i,j] \cdot p \cdot dx(i) \cdot \text{delp} \cdot dx(j)) \cdot dx \]
\[ \text{Gain} = \text{derivative}(\text{Form}, p, dp) \]

```python
import matplotlib.pyplot as plt
import matplotlib as mpl
import matplotlib.pyplot as p3

def plot_statespace(p, time):
    print 'plotting...
    fig = plt.figure(1, figsize=(20,20), dpi=100)
    ...
    x1_data = numpy.linspace(x1Min, x1Max, x1Dof)
    x2_data = numpy.linspace(x2Min, x2Max, x2Dof)
    ...
    for i in range(x1Dof):
        for j in range(x2Dof):
            p[i, j] = p((X1[i,j], X2[i,j]))
            surf = plot3d.plot_surface(X1, X2, p, rstride=1, cstride=1, cmap=cm.jet, 
            linewidth=0, antialiased=False)
            fig.colorbar(surf, shrink=0.5, aspect=5)
    plt.savefig('CR19_fpe%s.png' % t)
    plt.close()

def normalize(p):
    probability = assemble(p*dx)
    print 'normalize:
    ...
    p.init = InitialConditions(sigma)
    p.interpolate(p.init)

while t<t_end:
```

```
normalize(p)
plot_statespace(p, t)
t+=dt
p0.assign(p)

print 'solving for the time:
problem = iterate(Gain, Form, bc, interior, exterior)
solver = NewtonSolver('lu')
solver.parameters['convergence_criterion'] = 'incremental'
solver.parameters['relative_tolerance'] = 1e-3
solver.solve(problem, p.vector())