

Calibration of a Fluid-Structure Problem with Keras

Olivier Pironneau

▶ **To cite this version:**

Olivier Pironneau. Calibration of a Fluid-Structure Problem with Keras. 2019. hal-02281069

HAL Id: hal-02281069

https://hal.sorbonne-universite.fr/hal-02281069

Preprint submitted on 7 Sep 2019

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Calibration of a Fluid-Structure Problem with Keras

Olivier Pironneau [1]

Abstract

In this short paper we report on an inverse problem issued from a physical system, namely a fluid structure problem where the parameters are the rigidity constant, the solid-fluid density ratio and the fluid viscosity. We have chosen a simple geometry so that the numerical solution of the system is easy. We compare the solution of this inverse problem by a Neural Network with a more classical solution obtained with a genetic algorithm. The Neural Network does much better.

Keywords: Fluid-Structure Interaction, Calibration, Incompressible material, Neural Networks, Partial differential equations,.

1. Introduction

Let $\mathcal{L}_p w = f$ be the Partial Differential Equations (PDE) of a physical system which defines w for a given set of parameters p; the inverse problem is to recover p when w is known. Such inverse problems may be solved by optimization (see [7]):

$$p = \text{argmin}_{p\in P_{ad}}\{\|u^p - w\|^2 \ : \ \mathcal{L}_p u^p = f\} \tag{1}$$

where P_{ad} is the set of admissible values. If we know only a finite number of values of w, say at $\{z_k\}_1^K$ then we may solve, with $w^k = w(z_k)$,

$$p = \text{argmin}_{p\in P_{ad}}\{\frac{1}{K}\sum_1^K |u^p(z_k) - w^k|^2 \ : \ \mathcal{L}_p u^p = f\} \tag{2}$$

Deep Learning takes another approach; a neural network attempts to build an approximation of the inverse PDE and computes $\{w^k\}_1^K \mapsto p_{NN}$. The neural network is defined by a set of parameters π. Training the network means adjusting the parameters π so that $\{p_{NN}^m, \}_1^M$ is closest to $\{p^m\}_1^M$ for M"samples" $[\{w_m^k\}_1^K]_1^M$. Once trained it is hoped that the network produce a good approximation $\bar{p}$ for a new set of data $\{\bar{w}^k\}_1^K$.
This kind of deep learning [2] is not just another inversion method because one has also the capacity to store π and reuse the network in another context, another machine, like a tablet, and obtain inversion results very cheaply and fast.
In this paper we wish test the potential of DNN (deep neural network) to find the characteristics of an incompressible hyper-elastic material and of the fluid around it from the knowledge of a few data point in the space-time response to an external sollicitation. The performance of the DNN inverter will be compared with a genetic algorithm inverter using CMA-ES[3].

[1] *olivier.pironneau@gmail.com* , LJLL, Sorbonne University, Paris, France.

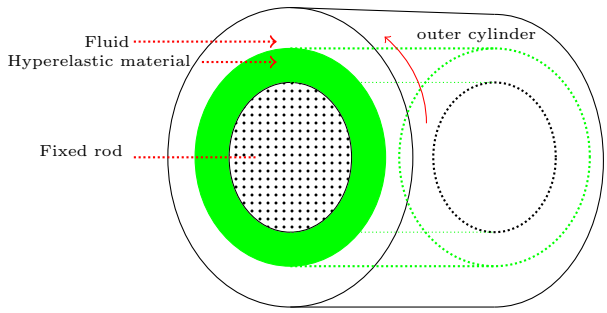


Fig. 1: *A fluid-structure system inside a rotating cylinder with a fixed rod in its center . Sketch of the system.*

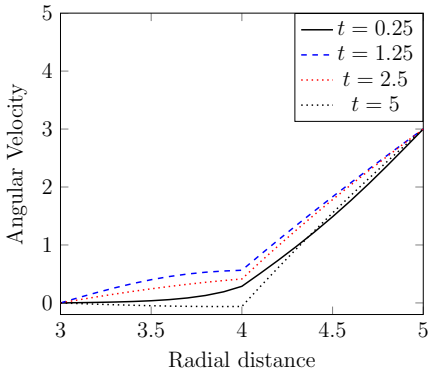


Fig. 2: *A 2d calculation showing the angular velocities along the radial axis at several times.*

2. A simple Fluid-Strucuture Testing Device

In 2 dimensions an incompressible hyper-elastic Mooney-Rivlin solid material (also called Neo-Hookian) is characterised by only one parameter, the Helmholtz potential constant c_1, besides its density ρ^s. Guessing the viscosity of a viscous fluid is not much simpler; so there is an interest for a simple probe to measure all three.
Consider the following axisymmetric device: a rigid fixed cylinder coated by an incompressible material , immersed in a viscous fluid (see figure 1); the whole aparitus is inside an outer cylinder. The outer cylinder can rotate; it is an external sollicitation which induces motions in the fluid and the solid. First the system is at rest and then a constant angular velocity is given to the outer cylinder. This causes the fluid to rotate with an angular velocity function of the radial distance r to the main axis; in turn due to the friction of the fluid at the interface, the incompressible material will be have an angular velocity u_θ which is also only a function of r and time . Due to elasticity u_θ will oscillate with time until numerical dissipation and fluid viscosity damps it.
In a two dimensional cut perpendicular to the main axis, the velocities and displacements are two dimensional and axisymmetric as well. Hence the geometry is a ring of inner and outer radii, R_0 and R_1, with the incompressible material between R_0 and R and fluid between R and R_1. Because of the incompressibility of the fluid and axial symmetry, R is constant.

In this paper $R_0 = 3$, $R = 4$, $R_1 = 5$. The radial velocity of the outer cylinder is also always kept at 3 rpm. So the parameters of the problem are c_1, ρ^s, μ^f.
As everything is axisymmetric the computation can be done in polar coordinates r, θ, and the fluid-solid system reduces to

$$\rho\partial_t u_\theta - \frac{1}{r}\partial_r[\xi^f r\partial_r u_\theta + \xi^s r\partial_r d] = 0, \quad \partial_t d = u_\theta, \ r \in (R_0, R_1), \quad v_{|R_0} = 0, \ u_{\theta|R_1} = 3, \tag{3}$$

with $\rho = \rho^s \mathbf{1}_{r\le R} + \rho^f \mathbf{1}_{r>R}$, $\xi^s = 2c_1\mathbf{1}_{r\le R}$, $\xi^f = \mu\mathbf{1}_{r>R}$, and with $d(r, 0) = 0$.
The system is approximated in time by an implicit Euler scheme and in space by the finite element method of degree 2. The linear systems are solved by LU factorization. All parameters are shown in Appendix 5.1 which is a freefem [4] implementation. Figure 3 shows results at 4 time instants.

Comparison between this one dimensional system and the full Eulerian 2D fluid-structure system has been reported in [5]

3. Solution with a Genetic Algorithm

Consider (2) with $K = 10 \times 5$ data points and $w^k = u_\theta(r^i, t^j)$ for 10 (resp 5) uniformly distributed r^i (resp. t^j) , CMAES [3] is a global stochastic optimizer which, in our case, stops when it reaches the "tolerance" 10^{-6} for the function evaluations. In the process it has done 569 function evaluations and the relative error on the approximate solution is $[0.087\%, 0.17\%, 0.085\%]$ for $[c_1, \rho^s, \mu^f]$. With 5×5 data points, the relative error is $[0.13\%, 0.08\%, 0.15\%]$.
It is computationally expensive (15min on a Mac book pro) but it indicates that the problem has a solution and the precision can be improved to any amount, it seems.

4. Solution with a Neural Network

A neural network is characterized by the number of hidden layers (depth) and the number of neurons in each layer (width). A neuron takes an input vector $\mathbf{x}$ and produces an output vector $\mathbf{y}$ with $y_i = \phi([\mathbf{A}\mathbf{x} + \mathbf{b}]_i)$; $\mathbf{A}$ is a rectangular matrix, $\mathbf{b}$ is a constant vector and ϕ is a non-linear function the so-called "activation". Here we have used systematically $\texttt{ReLU}(x) := x^+$ (rectified linear unit).
The DNN-parameters (called π in the introduction), $\mathbf{A}, \mathbf{b}$ of each hidden layer are adjusted by a stochastic gradient method to minimize a "loss" function.
The loss function is the least square error between the output of the DNN and the output of the PDE. Hence M solutions of the PDE are computed, each with its own parameters $p = (c_1, \rho^s, \mu^f)$. For the training phase the data of each sample consists of p and K values of the solution of the PDE.
The down side of deep learning is that a large number of samples required and for each sample the PDE needs to be solved. A few hours are needed to compute 5000 samples while the learning phase takes a few tens of seconds and the test phase microseconds.
A Keras code is given in Appendix 5.2. Note the simplicity of the `Python` program.

4.1. Influence of the number of Samples

We have taken a DNN of width 100 and depth 1. Training is done with M samples each containg $K = 10 \times 5$ time-space values . The influence of the number of samples M on the precision of the recovered parameters c_1, ρ^s and μ^f is shown on Table 1 and Figure 3. Table 2 illustrates the precision with some values given by the DNN and compared with the true values.

Table 1: **Precision** versus the number of samples: average absolute (left) and relative (in %) errors.

Samples	$\|c_1 - c_{1true}\|$	$\|\rho^s - \rho^s{}_{true}\|$	$\|\mu^f - \mu^f{}_{true}\|$	error$(c_1)/c_1$	error$(\rho^s)/\rho^s$	error$(\mu^f)/\mu^f$
250	0.1110471	0.1489467	0.06382506	2.8%	3.8%	1.7%
500	0.0506839	0.08803635	0.03628453	1.2%	2.2%	0.9%
1000	0.04044557	0.0534637	0.02502107	1.1%	1.4%	0.70%
2000	0.02899077	0.04001569	0.01182538	0.89%	0.97%	0.36%

Table 2: **Example of results with a DNN** with 100 neurons and using 1000 samples

c_{1NN}	${\rho^s}_{NN}$	${\mu^f}_{NN}$	c_{1true}	${\rho^s}_{true}$	${\mu^f}_{true}$
1.4074545	1.0255045	1.1367583	1.42379	1.04122	1.15782
0.64601064	0.93555903	1.4175779	0.620108	0.905501	1.42252
0.6114049	1.1150271	1.44915	0.611026	1.10804	1.46802
[0.7821889	1.2131171	0.49625444	0.751256	1.14957	0.505141
[0.69889516	1.10933	1.3671701	0.705757	1.13354	1.37266

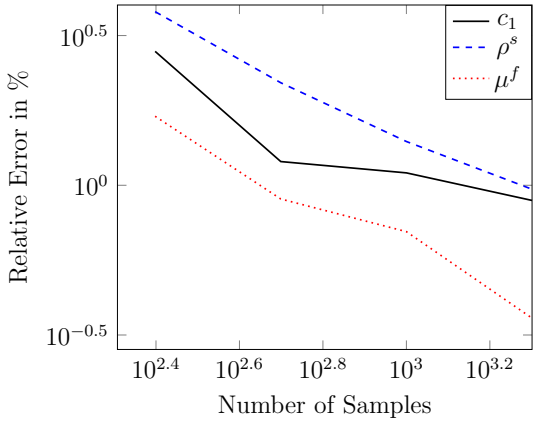


Fig. 3: Relative error versus the number of samples (log-log scales).

4.2. Influence of the Number of Data Points

With 5 points in r-axis and 2 points in time, the relative precision is $[2.4\%, 5.2\%, 1.5\%]$.
With 10 points in r-axis and 2 points in time, the relative precision is $[2.0\%, 6.0\%1.3\%]$.
With 5 points in r-axis and 5 points in time, the relative precision is $[1.1\%, 2.1\%, 0.8\%]$.
With 10 points in r-axis and 5 points in time, the relative precision is $[1.1\%, 0.97\%, 0.36\%]$.
In all 4 computations the number of samples is 1000.

4.3. Influence of the Network Architecture

We consider the case 10×5 data points and 1000 samples.

- With 1 layer of width 1000, the number of epochs to trigger *early stopping* is 111 and the loss is 0.166689.
- With 2 layers of width 100, the number of epochs is 196 and the loss is 0.10234474.
- With 1 layer of width 100, the number of epochs is 374 and the loss is 0.13168786.
- With a 2 layer of width 50, the number of epochs is 596 and the loss is 0.06836466.
- With 3 layers of width 34, the number of epochs is 393 and the loss is 0.10246510.
- With a 2 layer of width 50 and a layer of width 10 in between, the number of epochs is 888 and the loss is 0.06401216.

In view of the fact that all these numbers are realisations of random processes, it is not easy to draw any conclusion, but it seems to indicate that the architecture of the network is not very important, despite the following results:

Theorem 1. *[9] Functions of $W^{n,\infty}([0,1]^d)$ can be δ-approximated by ReLU networks with depth $O(\log \frac{1}{\delta})$ and width $O(\delta^{-\frac{d}{n}})$.*

Theorem 2. *[10] A bounded function of $x \in \mathbb{R}$ with bounded first derivative can be approximated with precision $O(1/(N \log N))$ by the network of Figure 4 of fixed width (here 5) and of depth N.*

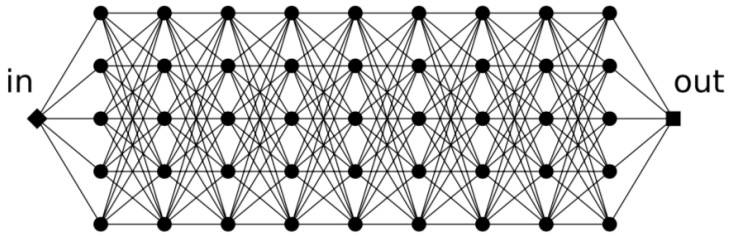


Fig. 4: A Deep network with constant width and large depth.

4.4. Critical Convergence

Let us explain why the precision is hard to adjust. Assuming M samples $\{x_i, u^\epsilon(x_i)\}_1^M$ known with stochastic error ϵ to simulate

$$\mathbb{R}^d \ni x \mapsto u(x) \in \mathbb{R}^s \text{ by a DNN } x \mapsto U(x) \tag{4}$$

Then the most sensitive thing to do is to

- Rescale $x_i \in (-1,1)^d$ and to
- Minimize with respect to the parameters which define U the loss function using a DNN of width and depth M_d, and observe the loss

$$\mathcal{E}_N(U) = \frac{1}{M} \sum_{i=1}^{M} \|U(x_i) - u^\epsilon(x_i)\|^2 \tag{5}$$

- The AdaGrad-Norm algorithm [8] will converge at rate $O(\frac{logP}{P})$, after P iterations, to a local minimum U^*.

Then, for some $s > 1$,

$$\|\mathbb{E}[u^\epsilon|x] - U^*(x)\| \leq C M_d^{-s} + \mathcal{E}_N(U). \tag{6}$$

Hence the value of the "loss" (the last term above) at convergence will control the precision because the first term on the right can be decreased to any small number by increasing the number of neurons. Unfortunately the magnitude fo this last term can only be discovered a posteriori.

5. Conclusion

The physical system chosen here is sufficiently simple to be sure of the accuracy of the numerical method. This has allowed us to concentrated on the parameters of the DNN. The bottleneck is the number of samples needed to train the network, although here a few hundreds are sufficient for a precision around 1%. Consequently the DNN performs very well. The architecture of the network is not important; the size of the data in each sample should also not be too small.
We intend to build on this experience to solve a more difficult FSI problem with a free boundary as in [6].

References

[1] Chollet, F. (2015): Keras, Deep learning library for Theano and Tensorflow. https://keras.io/k

[2] Goodfellow I. and Y. Bengio and A. Courville (2016): *Deep Learning*, MIT-Bradford.

[3] Hansen N. (2006): The CMA Evolution Strategy : A Comparing Review, in www.lri.fr/∼hansen/cmaesintro.html.

[4] Hecht F. (2012): New development in FreeFem++, J. Numer. Math., 20, pp. 251-265. (see also www.freefem.org.)

[5] (2017) Hecht H. and O. Pironneau An energy stable monolithic Eulerian fluid-structure finite element method Int. J. Numer. Methods in Fluids. Vol. 85, Issue 7, 10, pp 430–446.

[6] Pironneau, O. (2019) Parameter Identification of a Fluid-Structure System by Deep-Learning with an Eulerian Formulation. NCTS conference in honor of R. Glowinski (to appear).

[7] Tarantola A. (1987): Inverse Problem Theory. Elsevier Science.

[8] Ward R. and Xiaoxia Wu and Léon Bottou, (2018) AdaGrad stepsizes: sharp convergence over nonconvex landscapes, from any initialization arxiv 1806.01811

[9] Yarotsky D. (2017) Error bounds for approximations with deep ReLU networks. Neural Netw., 94:103-114.

[10] Yarotsky D. (2017) Quantified advantage of discontinuous weight selection in approximations with deep neural networks, arXiv: 1705.01365v1.

[11] Zhang X. and J Zhao and Y LeCun (2015): Character-level convolutional networks for text classification. Advances in neural information processing systems, 649-657.

Appendix: Programs

5.1. freefem program to generate the data

The freefem PDE solver for Mac, PC, Linux, is available for free download at texttt-twww.freefem.org.

```
load "pipe"
load "ff-cmaes"
load "ffrandom"
verbosity=0;
bool iscmaes=0;

real RR=5, r=4,r0=3, L=0.1;
border a1(t=r0,r){x=t;y=0;label=1;}
border a2(t=r,RR){x=t;y=0;label=1;}
border b(t=0,L){x=RR;y=t;label=2;}
border cc1(t=RR,r){x=t;y=L;label=3;}
border cc2(t=r,r0){x=t;y=L;label=3;}
border dd(t=L,0){x=r0;y=t;label=4;}
border e(t=0,L){x=r;y=t;label=5;}
int m=3,mm=5;
mesh Th=buildmesh(a1(50*m)+a2(50*m)+b(mm)+cc1(50*m)+cc2(50*m)+dd(mm)+e(mm));
fespace Vh(Th,P2,periodic=[[1,x],[3,x]]);
fespace V0(Th,P1dc);

real eps=1e-15;
int n,NN=21;
real T=5, dt=T/NN;
real nu,c1,rhos,nu0=1, c10=0.833,rhof=1,rhos0=2;

int beam=Th(r-0.1,0.05).region, fluid=Th(r+0.1,0.05).region;
cout<<beam<<" "<<fluid<<endl;
Vh d=x,w,wh,wold=0;
V0 nnu, rho;

int I=5;
int NV=3;
real[int] cc(NV);
real dr = (RR-r0)/(I-1);
srandomdev();
randinit(random());

problem aa(w,wh,init=n)
    = int2d(Th)(rho*(x+eps)*w*wh/dt + nnu*((x+eps)*dx(w)*dx(wh) + w*wh/(x+eps)))
    - int2d(Th)(rho*(x+eps)*wold*wh/dt - 2*c1*(x<=r)*((x+eps)*dx(d)*dx(wh) +
d*wh/(x+eps)))
    + on(2,w=3)+ on(4,w=0);

real[int, int] uref(NN,I), u(NN,I);
int funceval=0;

func real cost(real[int] & cc){
    c1=c10*cc[0];
    rhos=rhos0*cc[1];
    nu=nu0*cc[2];
    nnu=nu*(x>r)+2*c1*dt*(x<=r);
    rho=rhof*(x>r)+rhos*(x<=r);
    d=x;
    wold=0;
    real JJ=0;
```

```
    for(n=0;n<NN;n++){
        aa;
        wold=w;
        d=d+dt*w;
        for(int i=0;i<I;i++) {
            u(n,i)=w(r0+i*dr,0.05);
            if(iscmaes && n>0 && (n/4)*4==n ) JJ += (u(n,i)-uref(n,i))^2;
        }
    }
    funceval +=1;
    cout<<funceval<<" cc= "<<cc[0]<<" "<<cc[1]<<"  "<<cc[2]<<" JJ= "<<JJ/(NN+I)<<
endl;
    return JJ/(NN+I);
}

if(!iscmaes){
    int kmax = 1000;
    ofstream myfile("datafsi.txt");

    for(int k=0;k<kmax;k++){
        cc[0]=0.5+randreal1();
        cc[1]=0.5+randreal1();
        cc[2]=0.5+randreal1();
        myfile<<cc[0]<<" "<<cc[1]<<" "<<cc[2];
        cost(cc);
        for(n=1;n<NN;n+=4) for(int i=0;i<I;i++)  myfile<<"  "<<u(n,i);
        myfile<<endl;
    }
}else{
    cc[0]=1; cc[1]=1; cc[2]=1;
    cost(cc);
    uref=u;
    cc[0]=2;   cc[1]=0.5; cc[2]=1;
    cout<<"initial start "<<cc[0]<<" "<<cc[1]<< " "<<cc[2]<<endl;
    cost(cc);
    real minimum =
cmaes(cost,cc,stopTolFun=0.1e-5,stopMaxFunEval=1000,stopMaxIter=100000);
    cout<<"minimum= "<<minimum<<" Solution: c1/c1ref= "<<cc[0]<<" rhos/rhosf=
"<<cc[1]<<" nu/nuref= "<<cc[2]<<endl;
}
```

5.2. *The Keras/Python program to compute c_1,ρ^s and μ^f with a neural network*

```
import keras
import numpy as np
from keras.models import Sequential
from keras.layers import Dense, Activation, Conv2D, Flatten
from keras import losses
from keras.callbacks import EarlyStopping

Z = np.loadtxt('datafsi2.txt')
n_test=32
n_train = 250 -n_test
I,NN = 10,5
pix2=I*NN
epochs = 1000
batch_size=32
num_classes = 3
x_test  = np.zeros([n_test,pix2],'float32')
x_train = np.zeros([n_train,pix2],'float32')
y_test= np.zeros([n_test, num_classes],'float32')
y_train= np.zeros([n_train, num_classes],'float32')
############################################################
for i in range(n_train):
    for j in range(num_classes):
        y_train[i,j] = Z[i,j]
    for j in range(pix2):
            x_train[i,j] = Z[i,j+num_classes]
for i in range(n_test):
    k=n_train+i
    for j in range(num_classes):
        y_test[i,j] = Z[k,j]
    for j in range(pix2):
            x_test[i,j] = Z[k,j+num_classes]

model = Sequential()
model.add(Dense(100, input_dim=pix2,use_bias=True,
              bias_initializer='random_uniform',activation='relu'))
model.add(Dense(num_classes, use_bias=True, bias_initializer='random_uniform'))

model.compile(loss=losses.mean_squared_error, optimizer=
    keras.optimizers.Adadelta(lr=1.0, rho=0.95, epsilon=None, decay=0.0))
es = EarlyStopping(monitor='val_loss',mode='min',verbose=1,min_delta=1.e-7,
                   patience=40,restore_best_weights=True)
model.fit(x_train, y_train, batch_size=batch_size, epochs=epochs,
          verbose=1, validation_split=0.1,callbacks=[es])
u_test = model.predict(x_test)
score = model.evaluate(x_test, y_test, verbose=0)
print('Test loss:', (score)**0.5)
som,somr=0,0
for i in range(n_test):
    som+=np.abs(u_test[i]-y_test[i])
    somr+=np.abs(u_test[i]-y_test[i])/np.abs(u_test[i]+y_test[i])/2
    print(u_test[i]," ", y_test[i])
print("Average abs error= ",som/n_test," Average relative error= ",somr/n_test
```

1