IDMI310 - Introducción a la Cuantificación de Incertidumbre en Ingeniería

Estudiante:
Francisca Fernanda Cárdenas Fuentealba

Profesor:
Luis Ulises Medina Uzcátegui

Fecha:
8 de diciembre de 2022

Asignación 4

Universidad Austral de Chile

Facultad de Ciencias de la Ingeniería

Magíster en Ingeniería Mecánica y Materiales

1. Desarrollo:

El modelo mecánico para simular la dinámica del sistema es un modelo discreto lineal masa-resorte-amortiguador viscoso equivalente. Considere que los parámetros a determinar en el modelo son la rigidez y amortiguación, ya que se conoce la masa equivalente para el sistema. Basado en el archivo Asignacion4.py se requiere:

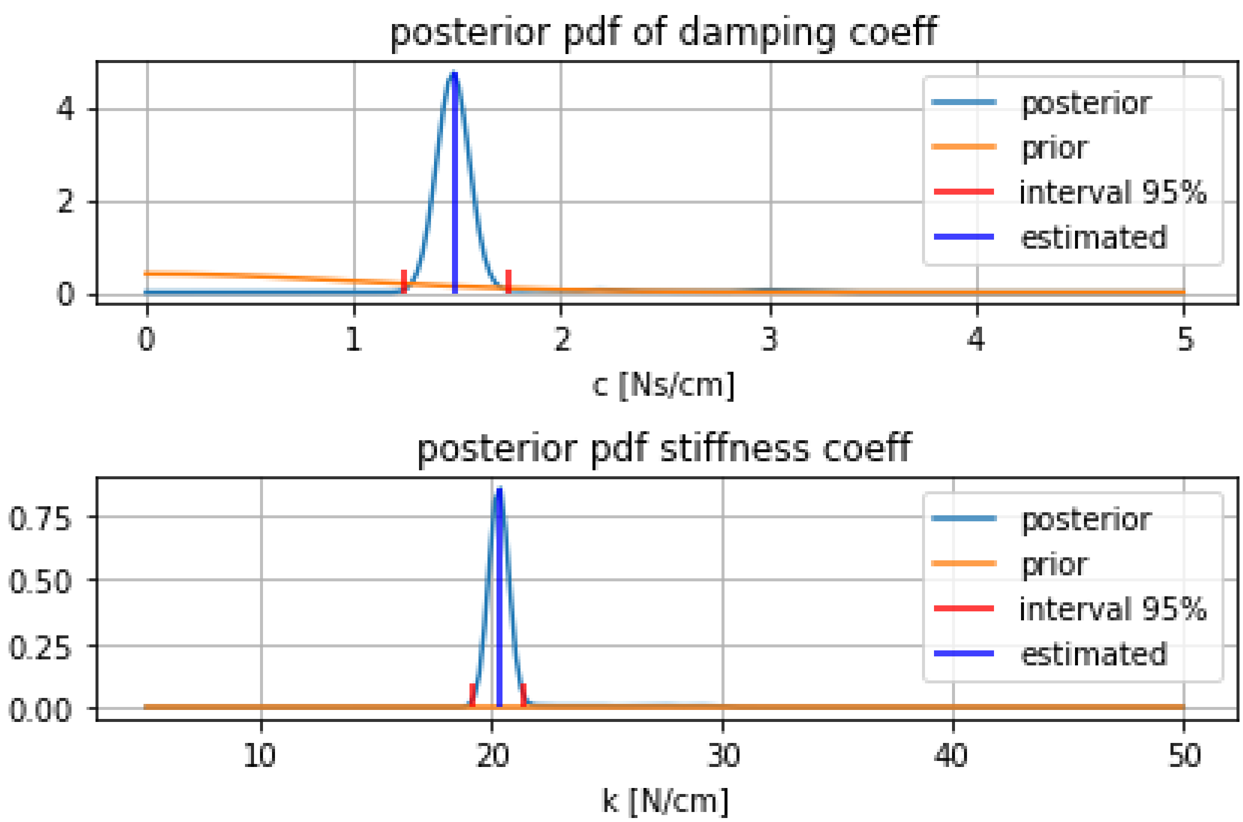

Figura 1: Grafico de la distribución de los parametros, su valor estimado y el intervalo de confianza del 95 %

1. Obtenga una estimación para cada uno de los parámetros requeridos del sistema (rigidez y amortiguación), cuando se utiliza el algoritmo de Metropolis-Hastings, tal como se indica en el código del archivo Asignacion4.py. Según se indica en el archivo Asignacion4.py, observe que aunque son desconocidos ambos parámetros, se conoce un dominio de los posibles valores para cada uno de ellos, esto es:

- Para el coeficiente de rigidez se sabe que su valor está comprendido entre 5 y 50 N/cm.
- Para el coeficiente de amortiguación, su valor se encuentra en el rango 0 y 5 $N\dot{s}/cm$.

R: Utilizando el comando *stats.mode()* de la librería *Scipy* de *Python*, se calculó la moda de los vectores de valores estimados y se obtuvo un valor estimado de rigidez de 20.289 N/m y un valor estimado de amortiguación de 1.494 $N\dot{s}/cm$. LLos que se ven representados en la linea azul de la *Figura 1*.

2. Para un nivel de confianza del 95 %, calcule los intervalos de confiabilidad para cada uno de los parámetros estimados en el punto anterior (rigidez y amortiguación).

R: Utilizando una adaptación del código desarrollado en *Example2_ 2nn.py* parte *#4 Calculating the confidence interval for the damping coefficient estimate* y del codigo *utility_ 2.py*, se calculó los intervalos de confianza al 95 % para la rigidez, [19.140 - 21.437] y para el amortiguamiento, [1.238 - 1.750]. Los que estan representados en la linea roja de la *Figura 1*.

2. Código:

```
# -*- coding: utf-8 -*-
"""
IDMI310-22
Introducción a la cuantificación de incertidumbres en Ingeniería
Magister en Ingeniería Mecánica y Materiales
Facultad de ciencias de la ingeniería
Universidad Austral de Chile
----------------------------------------
Created on Mon Aug 29 16:28:30 2022

References:
[1] A. Olivier, D. G. Giovanis, B. S. Aakash, M. Chauhan, L. Vandanapu,
   and M. D. Shields, ‘‘UQpy: A general purpose Python package and development
   environment for uncertainty quantification,’’ Journal of Computational
   Science, vol. 47, p. 101204, 2020.


[2] R. C. Smith, Uncertainty quantification: theory, implementation, and
    applications, vol. 12. Siam, 2013., Example 7.15,


[3] Singiresu S. Rao. Mechanical vibrations; SI conversion by Philip Griffin.
Harlow, United Kingdom : Pearson, [2018].



@author: Luis Medina
"""




#Basic libraries
import numpy as np
import matplotlib.pyplot as plt

#Importing from UQpy [1] to apply MCMC via Metropolis Hastings :
from UQpy import PythonModel
from UQpy.sampling.mcmc.MetropolisHastings import MetropolisHastings
from UQpy.inference.inference_models.ComputationalModel import ComputationalModel
from UQpy.run_model.RunModel import RunModel
from UQpy.inference import BayesParameterEstimation
from sklearn.neighbors import KernelDensity  # for the plots
from UQpy.distributions import JointIndependent, Normal, Uniform


#For statistics purposes
from scipy import stats as st
from scipy.stats import norm
from scipy.stats import t
from scipy.optimize import minimize_scalar

# # an library created for this example
```

```
from utility_2_old import *

#System's parameter values  [2]
#Note: units are assumed since example in ref. [2] does not specified units
k_true=20.5 #[N/cm]
m_true=1.0 # [Kg]
c_true=1.5 # [Ns/cm]

#initial condition values [2]
y0_val=2.0       #[cm]
y0_dot_val=0.0   #[cm/s]

#Define a time vector to get  system's response and
to=0.0 #[s]
tf=5.0 #[s]
n=500 #number of points
t_vec=np.linspace(to, tf,n)


# Defining functions
#_____________________________________________________________________________

#Kernel Density Estimation function
def pdf_from_kde(domain, samples1d):
    bandwidth = 1.06 * np.std(samples1d) * samples1d.size ** (-1 / 5)
    kde = KernelDensity(bandwidth=bandwidth).fit(samples1d.reshape((-1, 1)))
    log_dens = kde.score_samples(domain)
    return np.exp(log_dens)

#_____________________________________________________________________________


# 1 Define noise to be added to sistem´s response. This is a strategy  to
# generate synthetic data since there are not measurements available

#Assuming that error is iid and error ~ N(error_mu, error_var ) [2]

sigma_error=.1                      # error's standard deviation
error_mu=0.0                        # error's mean value
error_var=sigma_error**2            # error variance
error=sigma_error*np.random.randn(n)  # errors



#2.System's response "measurements"



# Generate data

param_true = np.array([ c_true,k_true]).reshape((1, -1))


model = PythonModel(model_script='utility_functions.py', model_object_name='y',
                    var_names=['c', 'k'])
```

```
h_func = RunModel(model=model)
h_func.run(samples=param_true)
y_vec = np.array(h_func.qoi_list[0]) # Quantity of interest
n_data=len(y_vec)                    # Number of measurements' points

# Add noise, using a random_state for reproducible results
error_covariance = 0.1**2
noise = Normal(loc=0., scale=np.sqrt(error_covariance)).rvs(nsamples=n_data, random_state=123).reshape((
#"artificial" measurements
y_obs = y_vec + noise




#MCMC implemented using Metropolis Hastings algorithm

# Prior density matrix

p0 = Normal()
p1 = Normal()
prior = JointIndependent(marginals=[p0, p1])


inference_model = ComputationalModel(n_parameters=2, runmodel_object=h_func, error_covariance=error_cova
                                     prior=prior)
# Proposal density J
J = JointIndependent([Normal(scale=0.1), Normal(scale=0.1)])

c_initial=3.0  #Enter here a initial value for the Metropolis-Hastings algoritm (damping coeff)
k_initial=30.0 #Enter here a initial value for the Metropolis-Hastings algoritm (stiffness coeff)

# Invoking Metropolis Hastings algorithm


mh1 = MetropolisHastings(jump=10, burn_length=0, proposal=J, seed=[c_initial, k_initial],
                         random_state=456)

"""
 Info about   some inputs requested by the  Metropolis Hastings algorithm used:
 jump: Thinning parameter, used to reduce correlation between samples. Setting :code:`jump=n` corresponds
 to skipping :code:`n-1` states between accepted states of the chain. Default is :math:`1` (no thinning)

 burn_length: number of samples at the beginning of the chain to discard

 proposal: Proposal distribution, must have a log_pdf/pdf and rvs method. Default: standard
  multivariate normal

 seed: Seed of the Markov chain(s)
 random_state: Random seed used to initialize the pseudo-random number generator
 (if defined then it wouuld be possible to get same results every time the code runs)
"""



# Bayes estimator
```

```
bayes_estimator = BayesParameterEstimation(inference_model=inference_model,
                                           data=y_obs,
                                           sampling_class=mh1,
                                           nsamples=500)

# Finally, samples from estimated distributions for both calculated parameters are:
s = bayes_estimator.sampler.samples

c_bayes_samples=s[:, 0]  # Estimated values samples for damping coeff.
k_bayes_samples=s[:, 1]  # Estimated values samples for stiffness coeff.

#%%
"""
Resolucion asignacion 4 - Introduccion a la CUnatificacion de incertidumbre

author: Francisca Cardenas
fecha: 04 de diciembre
"""
#___________________________________________________________________________
damp_est,a = st.mode(c_bayes_samples)   # Estimated values for damping coeff.
stiff_est,b = st.mode(k_bayes_samples)  # Estimated values for stiffness coeff.

#Using a function defined at utility_2 library to get the sensitivity matrix for damping
Sens_matrix_damp=Sensitivity_matrix2(m_true,damp_est[0],k_true,t_vec,y0_val,y0_dot_val)

#Computing an estimation of error variance for damping
est_error_var_d=np.var(error,ddof=1) # Notice that this value should be close to sigma_error**2
                                   # that is the error variance used to generate
                                   # the y_obs data


est_error_std_d=np.sqrt(est_error_var_d) # estimated error standard deviation for damping
                                     # it must be close to sigma_error
#Estimating the damping coeff 's variance
V=np.linalg.inv(Sens_matrix_damp)*est_error_var_d

#Confidence interval for the damping coefficient estimate
SE_d=est_error_std_d*np.sqrt(Sens_matrix_damp) #standard error

#_________________________________________________________________

#Using a function defined at utility_2 library to get the sensitivity matrix for stiffness
Sens_matrix_stiff=Sensitivity_matrix2(m_true,c_true,stiff_est[0],t_vec,y0_val,y0_dot_val)

#Computing an estimation of error variance for stiffness
est_error_var_f=np.var(error,ddof=2) # Notice that this value should be close to sigma_error**2
                                   # that is the error variance used to generate
                                   # the y_obs data


est_error_std_f=np.sqrt(est_error_var_f) # estimated error standard deviation for stiffness
                                     # it must be close to sigma_error
#Estimating the stiffness coeff 's variance
V=np.linalg.inv(Sens_matrix_stiff)*est_error_var_f
```

```
#Confidence interval for the stiffness coefficient estimate
SE_s=est_error_std_f*np.sqrt(Sens_matrix_stiff) #standard error

#Calculating the confidence interval for damping
damp_int=t.interval(alpha = 0.95, df = len(c_bayes_samples) -1, loc = damp_est[0], scale = SE_d[1,1])

#Calculating the confidence interval for stiffness
stiff_int=t.interval(alpha = 0.95, df = len(k_bayes_samples) -1, loc = stiff_est[0], scale = SE_s[0,0])

#%%
fig, axs=plt.subplots(2,1)
domain_for_c = np.linspace(0, 5, 200)[:, np.newaxis]
pdf_c = pdf_from_kde(domain_for_c,c_bayes_samples )
axs[0].plot(domain_for_c, pdf_c, label='posterior')
axs[0].plot(domain_for_c, p0.pdf(domain_for_c), label='prior')
axs[0].vlines(x=[damp_int[0],damp_int[1]],color='r',ymin=0,ymax=0.5, label='interval 95%')
axs[0].vlines(x=[damp_est[0]],color='b',ymin=0,ymax=4.75, label='estimated')
axs[0].set_title('posterior pdf of damping coeff')
axs[0].set_xlabel('c [Ns/cm]')
axs[0].legend()
axs[0].grid(True)

domain_for_k = np.linspace(5, 50, 200)[:, np.newaxis]
pdf_k = pdf_from_kde(domain_for_k, k_bayes_samples)
axs[1].plot(domain_for_k, pdf_k, label='posterior')
axs[1].plot(domain_for_k, p1.pdf(domain_for_k), label='prior')
axs[1].vlines(x=[stiff_int[0],stiff_int[1]],color='r',ymin=0,ymax=0.1, label='interval 95%')
axs[1].vlines(x=[stiff_est[0]],color='b',ymin=0,ymax=0.86, label='estimated')
axs[1].set_title('posterior pdf stiffness coeff')
axs[1].set_xlabel('k [N/cm]')
axs[1].legend()
axs[1].grid(True)

fig.tight_layout()
plt.show()

#%%
print('estimated damping:',damp_est[0])
print('damping interval: (',float(damp_int[0]),':',float(damp_int[1]),')')
print('estimated stiffness:',stiff_est[0])
print('stiffness interval: (',float(stiff_int[0]),':',float(stiff_int[1]),')')
```