Open-Source pyMAPOD Framework

User Manual

v.beta

Developed by Computational Design Laboratory (CODE Lab)

Department of Aerospace Engineering
Iowa State University
Ames, Iowa

Leading Developers: Leifur Thor Leifsson
Xiaosong Du
Jethro Nagawkar

We gratefully acknowledge the support of the Center for Nondestructive Evaluation Industry/University Cooperative Research Program at Iowa State University

Code and updates are available at the link: https://github.com/CODE-Lab-IASTATE/MAPOD

CONTENTS

1 INTRODUCTION

This user manual aims at providing details and utilization examples on the open-source pyMAPOD framework. MAPOD essentially stands for model-assisted probability of detection. For the theoretical background, please go through the associated document, Theory Manual of this framework.

The Computational Design (CODE) lab would like to thank the Center of Nondestructive Evaluation (CNDE) for funding this program. In addition, we would also like to say thanks to all the colleagues, Dr. Jiming Song, Dr. William Meeker, Dr. Ronald Roberts and Dr. Leonard Bond for providing the physics-based NDT simulation models as well as their valuable ideas and suggestions.

2 OPERATION ENVIRONMENT

2.1 Python on Linux / Windows systems

This open-source framework is written in python, which is well known for its cross-platform capabilities. Therefore, the users can run it on any system. The authors wrote and tested the code in python v2.7, Window x64 as well as Ubuntu 8.04.1 LTS local desktop and laptop machine.

2.2 Suggested compiler and necessary modules

The code is written within the Enthought Canopy compiler, which is very powerful and convenient for integrating python toolboxes. Users simply select the Package Manager under the toolbar Tools, to add any necessary modules. However, the user is free to use another other compiler of their choice. In this work, the authors have avoided using additional modules as much as possible. The necessary modules and corresponding utilizations are:

1. collections: the module 'OrderedDict' is used to specify random inputs with statistical distributions.
2. pyDOE: randomly generate sample points in the latin hypercube sampling (LHS) scheme.
3. numpy: performs various numerical operations, such as numpy.array.
4. pandas: reads data frames from excel files.
5. matplotlib.pyplot: used to view imported data, generate the "$\hat{a}$ vs. a" plots, and POD curves.
6. sys: sys.exit() is used when the data format provided is incorrect.
7. mlab: links python with Matlab, making Matlab a callable module for python.
8. sklearn: linear_model from sklearn is implemented, in particular, the linear_model.LinearRegression and linear_model.LassoLarsCV are utilized for linear regression.

9. math: used for calculating factorial of integral values.
10. scipy.stats: used for generating normally distributed sample points.
11. xlrd: used to read in Excel files.

3 UTILIZATION

This MAPOD framework is written for various types of implementations.

Firstly, it can handle existing data directly. The users only need to provide their data in specified format, irrespective of the type of physics-based simulation model the data is generated in.

Secondly, the framework can be linked with real physics-based simulation models, which can be in either Python or Matlab format. The only requirement is that the users need to modify their physics model scripts slightly into a specified format. This will be discussed later in this section.

Thirdly, this open-source framework provides the users the option of constructing various meta-models namely the polynomial chaos expansions (PCE), kriging as well as polynomial chaos-based kriging (PC-Kriging). A detailed description of these metamodels can be found in the Theory manual. Then an arbitrary number of predicted model response can be generated for the calculation of the "$\hat{a}$ vs. a" linear regression as well as the probability of detection (POD) analysis. The user also has an option to validate the metamodels before constructing the POD curves.

Fourthly, the user can perform sensitivity analysis (SA) in order quantify how the model response is affected by the various uncertainty parameters. This framework uses the variance-based SA, namely the Sobol indices to perform SA. The user has an option to calculate these indices with or without constructing the metamodel.

And of course, the users are very welcome to go through the open-source code and provide any feedback. We are always glad to keep improving our work. In the following parts of this section, the overview as well as detailed description of the MAPOD framework is provided.

3.1 Flowchart

The flowchart of this MAPOD framework is given in Fig. 1. The user starts by defining the variables that will be used in the MAPOD framework. This includes the defect size 'a' as well as other uncertainty parameters. The user then has the option to either construct a metamodel is his/her choice or use the expensive physics-based simulations to either generate the POD curves and/or perform the SA. Details of each step are provided in the following sections.

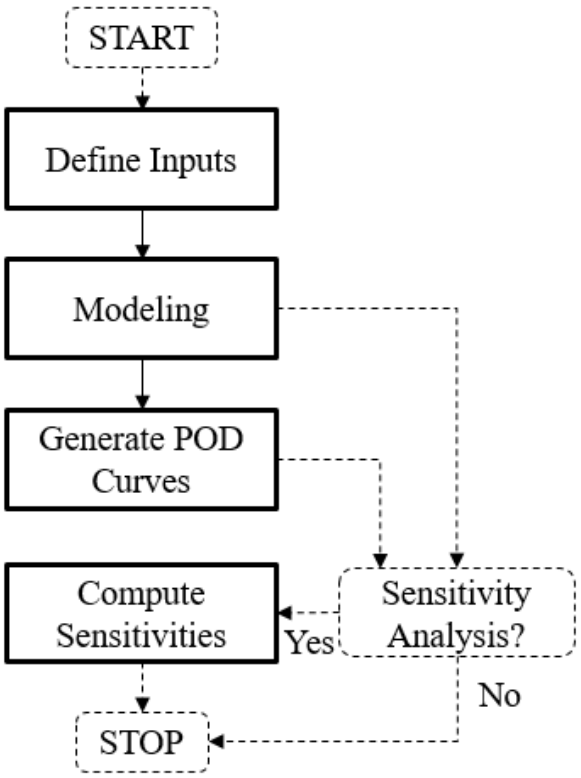

Figure 1: General process of MAPOD framework.

3.2 Define Inputs

This MAPOD framework is able to connect with existing data from any physics-based NDT simulation model or experiments or is able to is able to connect with real physics-based NDT simulation models directly. These model can either be in Python or Matlab format.

The required format of the preexisting data is shown in Fig. 2. Details on this required format are as following:

1. Generate model response for various defect sizes, put them into an Excel file, and specify this

index	size	response
1	0.1	0.803879
2	0.1	1.180379
3	0.1	0.642051
4	0.1	0.124154
5	0.1	1.047103
6	0.1	1.208215
7	0.1	0.773787
8	0.1	0.447201

Figure 2: Formatted data in the Excel file, taking "test_MAPOD.xlsx" as an example.

file name in pyMAPOD.py as mentioned in Section 2.1.

2. Specify the name of each Excel sheet within the data file. Specify this sheet name in pyMAPOD.py as mentioned in Section 2.1.

3. Each Excel sheet containing the data of interest has three columns in total. The first column specifies the index of the data, the second column the current defect size, and the third column the corresponded model response.

4. The Excel sheet should have a title in each column, for example in Fig. 2, we give "index", "size" and "response" for each column respectively.

To link the physics-based simulation model, two arguments need to be specified. The first one is the defect size and the second one is an array containing all random inputs. The main function of this framework, namely pyMAPOD.py either reads the preexisting data or links to the physics-based simulation model specified in test_func.py.

3.3 Modeling

The user has the option to either construct a metamodel of his/her choice or is free to use the expensive physics-based simulation model to generate the POD curves and/or perform SA. In this framework, three metamodels are provided for the user to choose from, namely PCE, Kriging and PC-Kriging. The metamodels are constructed using the Latin Hypercube sampling (LHS) in this framework. The user has the option to first validate the constructed metamodels. The root mean squared error (RMSE) and the normalized RMSE (NRMSE) are used to quantify the errors. The Monte Carlo sampling (MCS) is performed to generate the testing points for validation. pyPCE.py is used to construct and validate the PCE metamodel. pyKriging.py and pyPCKriging.py is used to do the same for the Kriging and PC-Kriging metamodels respectively.

3.4 Generate POD Curves

In order to generate the POD curves, first the "$\hat{a}$ vs. a" linear regression needs to be performed in order to calculate the required constants, namely, β_0, β_1, and σ_ϵ. This constants are found using a maximum likelihood estimate (see Theory manual) and in this framework is calculated by the ahat_vs_a.py function. These constants are then fed into the pod_gen.py function in order to generate the POD curves.

3.5 Compute Sensitivities

The user has the option to compute sensitivities. The variance-based first-order and total order Sobol indices are computed by the framework. The user is recommended to go through the Theory manual to get a better understanding of the Sobol indices. The Sobol indices are computed by the sens_analysis.py function.

4 EXAMPLES

This section shows how to apply the MAPOD framework to some test cases. These tutorials covers all the important functionalities of the MAPOD framework. This ranges from generating POD curves as well as performing SA directly from the physics-based simulation models or by constructing the different metamodels as an intermediate step. A tutorial on how to connect existing data to the MAPOD framework is also presented. The necessary mathematical background can be found in the Theory manual.

4.1 Analytical test function

4.1.1 Problem description

This MAPOD framework can be connected with physics-based simulation models directly. A simple numerical case is taken as an example.

$$y = e^{kln(a)+b}, \tag{1}$$

where 'k' has the distribution of Uniform (3, 4), 'b' has the distribution of Normal (5, 0.5) and 'a' can be taken as five discrete points [0.1, 0.2, 0.3, 0.4, 0.5] or with a distribution of Uniform (0.1, 0.5).

4.1.2 python function

This section demonstrates the following:

- Linking a physics-based simulation model to pyMAPOD, which is in python format.
- Inputting the defect size 'a', which could either be discrete or continuous.
- Specifying the distribution of the uncertainty parameters 'b' and 'k'.
- Constructing the "$\hat{a}$ vs. a" and POD curves

The setup in the pyMAPOD.py is shown on the next page. The following are important settings to be specified

- input_type = 'func' specifies that the physics-based simulation model is to be used.
- funcForm specifies the format of physics-based simulation model.
- funcName specifies the name of the function for pyMAPOD to read.
- a_type = 'discrete' is used when 'a' takes discrete values.
- Uniform1 and Gaussian1 is the distributions of the 'k' and 'b' uncertainty parameter respectively.
- A LHS is used for the uncertainty parameters. Here, 100 for each 'a'.

```
input_type = 'func'

gen_metamodel = False

val_metamodel = False

calc_sobol = False

calc_sobol_meta = False

funcForm = 'python'

funcName = test_func.simpleFunc

a_type = 'discrete'

a = npy.array([0.1, 0.2, 0.3, 0.4, 0.5])

x_prob = collections.OrderedDict([('Uniform1', (3, 4)),
                                  ('Gaussian1', (5, 0.5))])

x_sample = {'LHS' : 100}

PCE_gen = False

Kriging_gen = False

PCKriging_gen = False

beta0, beta1, tau = ahat_vs_a.regression(data)

ahat_vs_a.view_reg(data, beta0, beta1, tau)

threshold = 0.5

mu, sigma, pcov = pod_gen.pod_cal(data, beta0, beta1, tau, threshold)

a_50, a_90, a_90_95 = pod_gen.pod_para(mu, sigma, pcov)

pod_gen.pod_view(mu, sigma, pcov)
```

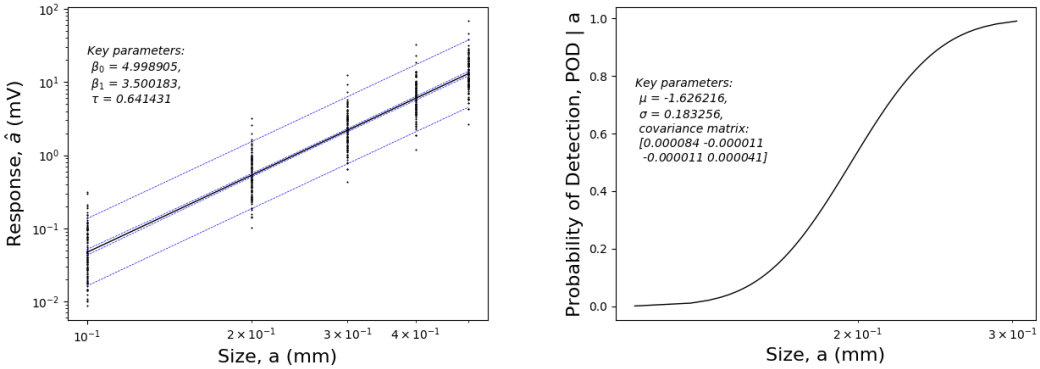

Figure 3: "$\hat{a}$ vs. a" plots and POD curves.

The resulting plots are shown in Fig. 3.

We can also use the independent variable, 'a', as Uniform (0.1, 0.5). Then all the settings are the same as above, except the followings:

```
a_type = 'continuous'

a = npy.array([])

x_prob = collections.OrderedDict([('Uniform1', (0.1, 0.5)),
                                  ('Uniform2', (3, 4)),
                                  ('Gaussian1', (5, 0.5))])

x_sample = {'LHS' : 1000}
```

Note that now Uniform1 refers to 'a' and Uniform2 to 'k'. The resulting plots are shown in Fig. 4.

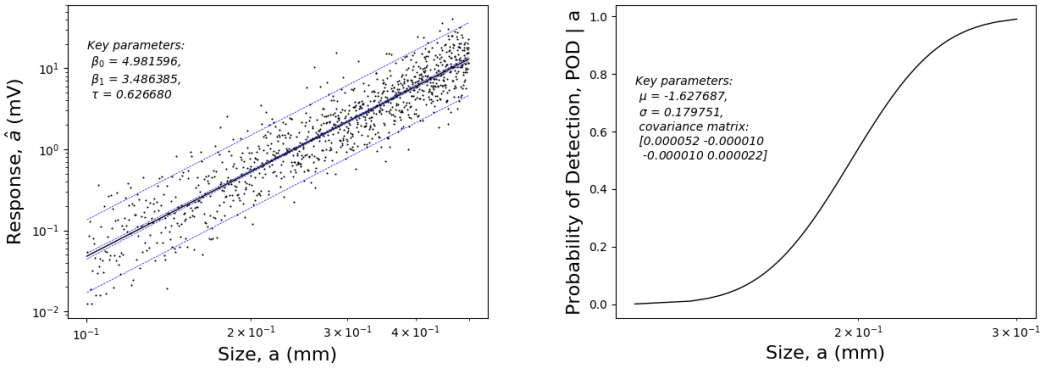

Figure 4: "$\hat{a}$ vs. a" plots and POD curves.

4.1.3 PCE model on the Python function

This section demonstrates the use of the PCE metamodel to construct the POD curve, for the same problem as mentioned in Section 4.1.2. The following settings need to be made:

- gen_metamodel is set to True to generate the metamodel.
- PCE_gen specifies that the PCE metamodel is to be used.
- n_deg specifies the order of the PCE metamodel.

```
input_type = 'func'

gen_metamodel = True

val_metamodel = False

calc_sobol = False

calc_sobol_meta = False

funcForm = 'python'

funcName = test_func.simpleFunc

a_type = 'discrete'

a = npy.array([0.1, 0.2, 0.3, 0.4, 0.5])

x_prob = collections.OrderedDict([('Uniform1', (3, 4)),
                                  ('Gaussian1', (5, 0.5))])

x_sample = {'LHS' : 45}

PCE_gen = True

Kriging_gen = False

PCKriging_gen = False

n_deg = 8

x_meta = {'LHS' : 1000}

beta0, beta1, tau = ahat_vs_a.regression(data)

ahat_vs_a.view_reg(data, beta0, beta1, tau)

threshold = 0.5

mu, sigma, pcov = pod_gen.pod_cal(data, beta0, beta1, tau, threshold)

a_50, a_90, a_90_95 = pod_gen.pod_para(mu, sigma, pcov)

pod_gen.pod_view(mu, sigma, pcov)
```

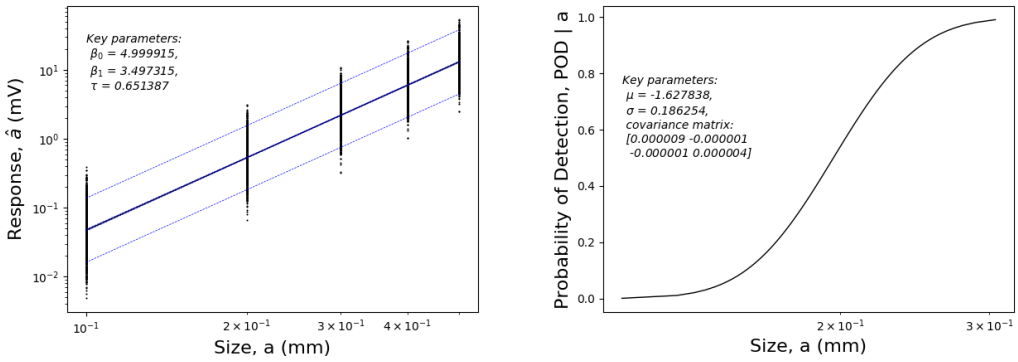

Figure 5: "$\hat{a}$ vs. a" plots and POD curves.

The resulting plots are in Fig. 5.

'a' can be set as a random variable, with Uniform (0.1, 0.5) as shown below. Then,

```
a_type = 'continuous'

a = npy.array([])

x_prob = collections.OrderedDict([('Uniform1', (0.1, 0.5)),
                                  ('Uniform2', (3, 4)),
                                  ('Gaussian1', (5, 0.5))])

x_sample = {'LHS' : 120}

n_deg = 7
```

The resulting plots are shown in Fig. 6.

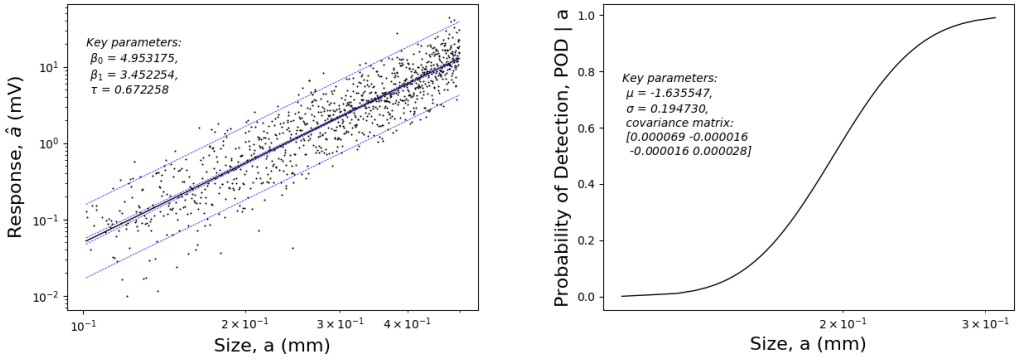

Figure 6: "$\hat{a}$ vs. a" plots and POD curves.

4.1.4 Validation of the Kriging model on the Matlab function

The following are demonstrated in this section:

- Linking a Matlab-format physics-based simulation model to the MAPOD framework.
- Construction and validation of the Kriging metamodel.
- Sampling plans for constructing and validating the metamodel.
- Kriging construction settings.

```
input_type = 'func'

gen_metamodel = False

val_metamodel = True

calc_sobol = False

calc_sobol_meta = False

funcForm = 'matlab'

funcName = 'test_func_mat'

a_type = 'discrete'

a = npy.array([0.1, 0.2, 0.3, 0.4, 0.5])

x_prob = collections.OrderedDict([('Uniform1', (3, 4)),
                                  ('Gaussian1', (5, 0.5))])

x_sample = {'LHS' : 20}

PCE_gen = False

Kriging_gen = True

PCKriging_gen = False

x_vali = {'MCS' : 1000}

setting['trendType'] = 'linear'
setting['estimator'] = 'maximum_likelihood'
setting['optimizer'] = 'SLSQP'
```

The following options are to be used:

- val_metamodel constructs and validates the metamodel.
- funcForm is set to Matlab to link the Matlab simulation model to the pyMAPOD framework.
- Kriging_gen to select the Kriging metamodel.
- x_sample to specify the sampling plan along with the number of samples to construct the metamodel.
- x_vali does the same, but for the validation points.
- 'trendType' specifies the global trend function of the Kriging metamodel.
- 'maximum_likelihood' to find the maximum likelihood of the hyperparameters of the metamodel.
- 'SLSQP' the optimizer used to find these hyperparameters.

The results are as follows:

```
Defect size 0.1
The RMSE: 0.0046
THE NRMSE: 0.0120

Defect size 0.2
The RMSE: 0.0191
THE NRMSE: 0.0060

Defect size 0.3
The RMSE: 0.03834
THE NRMSE: 0.0035

Defect size 0.4
The RMSE: 0.0639
THE NRMSE: 0.0025

Defect size 0.5
The RMSE: 0.0888
THE NRMSE: 0.0017
```

4.1.5 Sobol index calculation

The calculation of the first-order and total order Sobol indices is demonstrated in this section. Note, that no metamodel is constructed for this calculation. However, the user may construct a metamodel of his/her choice. This is demonstrated in the following section. The setup is as follows:

```
input_type = 'func'

gen_metamodel = False

val_metamodel = False

calc_sobol = True

calc_sobol_meta = False

funcForm = 'python'

funcName = test_func.simpleFunc

a_type = 'discrete'

a = npy.array([0.1, 0.2, 0.3, 0.4, 0.5])

x_prob = collections.OrderedDict([('Uniform1', (3, 4)),
                                  ('Gaussian1', (5, 0.5))])

x_sample = {'LHS' : 100}

x_sobol = {'LHS' : 100}

PCE_gen = False

Kriging_gen = False

PCKriging_gen = False
```

The options used are as follows:

- cal_sobol is set to True to calculate the Sobol indices.
- x_sobol is used to sample the design space to calculate the Sobol indices.

The corresponding results is as follows:

```
Defect size 0.1
1st order Sobol index
1st uncertainty parameter: 0.35703404189985277
2nd uncertainty parameter: 0.7806280431422407

Total order Sobol index
1st uncertainty parameter: 0.7562060063870426
2nd uncertainty parameter: 0.6249739413194544

Defect size 0.2
1st order Sobol index
1st uncertainty parameter: 0.26043166646438726
2nd uncertainty parameter: 0.8868110926918888

Total order Sobol index
1st uncertainty parameter: 0.5863655174912262
2nd uncertainty parameter: 0.7516066668195023

Defect size 0.3
1st order Sobol index
1st uncertainty parameter: 0.17997208056393615
2nd uncertainty parameter: 0.9815926711790979

Total order Sobol index
1st uncertainty parameter: 0.4338379196853705
2nd uncertainty parameter: 0.8664995456286563

Defect size 0.4
1st order Sobol index
1st uncertainty parameter: 0.11525234435622732
2nd uncertainty parameter: 1.0594463447978524

Total order Sobol index
1st uncertainty parameter: 0.3034595428480572
2nd uncertainty parameter: 0.9661042598786521

Defect size 0.5
1st order Sobol index
1st uncertainty parameter: 0.06610328536547028
2nd uncertainty parameter: 1.1183418200236852

Total order Sobol index
1st uncertainty parameter: 0.19798904322619632
2nd uncertainty parameter: 1.0479762423916583
```

4.1.6 Sobol index calculation using PC-Kriging

This section covers the following:

- Calculation of Sobol indices using a metamodel.
- Constructing the PC-Kriging metamodel.
- Options used to construct the PC-Kriging metamodel.

The setup is as follows:

```
input_type = 'func'

gen_metamodel = False

val_metamodel = False

calc_sobol = True

calc_sobol_meta = True

funcForm = 'python'

funcName = test_func.simpleFunc

a_type = 'continuous'

a = npy.array([])

x_prob = collections.OrderedDict([('Uniform1', (0.1, 0.5)),
                                  ('Uniform2', (3, 4)),
                                  ('Gaussian1', (5, 0.5))])

x_sample = {'LHS' : 45}

x_sobol = {'LHS' : 100}

PCE_gen = False

Kriging_gen = False

PCKriging_gen = True

setting['trendType'] = 'pce'
setting['estimator'] = 'maximum_likelihood'
setting['optimizer'] = 'SLSQP'
```

The options used are as follows:

- cal_sobol to calculate the Sobol indices.
- cal_sobol_meta to generate the metamodel from which the Sobol indices will be calculated.
- PCKriging_gen to generate the PC-Kriging metamodel.
- 'trendType' is set to 'pce' in order to use PCE as the global trend function.

The corresponding results are as follows:

```
1st order Sobol index
1st uncertainty parameter: 0.5817513252954871
2nd uncertainty parameter: 0.10688476229776851


Total order Sobol index
1st uncertainty parameter: 0.8761138815847912
2nd uncertainty parameter: 0.06476004876052731
```

4.2 Spherical void defect

This tutorial covers link preexisting data from the spherically-void-defect case, developed by the World Federal Nondestructive Evaluation Center in 2004, to generate the POD curves. A detailed description of this case can be found in the Theory manual. This tutorial covers the following:

- Connecting existing data to the MAPOD framework.
- Plotting the "$\hat{a} vs. a$ regression line.
- Plotting the POD curve

The setup for this tutorial is on the next page. The following options are used:

- input_type is set to dat to specify that the MAPOD framework is to be connected to the preexisting data.
- ahat_vs_a.regression returns the coefficients of the "$\hat{a}$ vs. a" regression line.
- ahat_vs_a.view_reg is used to plot the regression line.
- pod_gen.pod_cal returns the μ and σ of the POD curves.
- pod_gen.pod_view generates the POD curve.

```
import func_data
import ahat_vs_a
import pod_gen
import collections
import prob_distrs
import numpy as npy

input_type = 'dat'

beta0, beta1, tau = ahat_vs_a.regression(data)

ahat_vs_a.view_reg(data, beta0, beta1, tau)

threshold = 6.5

mu, sigma, pcov = pod_gen.pod_cal(data, beta0, beta1, tau, threshold)

a_50, a_90, a_90_95 = pod_gen.pod_para(mu, sigma, pcov)

pod_gen.pod_view(mu, sigma, pcov)
```

Results on "$\hat{a}$ vs. a" regression are shown in Fig. 7a and the related results of the POD are shown in Fig. 7b. The plots also show the values of the parameters β_0, β_1 and τ as well as μ and σ.

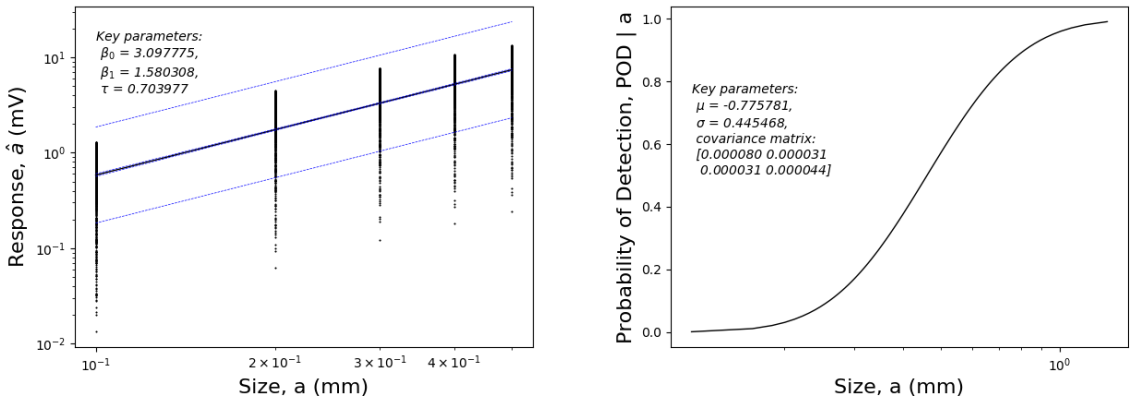

Figure 7: "$\hat{a}$ vs. a" plots and POD curves.